Automatic Synthesis of Synchronisation Primitives for Concurrent Programs

by

Thorsten Tarrach

7th July 2016

A thesis presented to the
Graduate School
of the
Institute of Science and Technology Austria, Klosterneuburg, Austria
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

The thesis of Thorsten Tarrach, titled *Automatic Synthesis of Synchronisation Primitives for Concurrent Programs*, is approved by:

Supervisor: Thomas A. Henzinger, IST Austria, Klosterneuburg, Austria

Signature: ______________________

Committee Member: Roderick Bloem, TU Graz, Graz, Austria

Signature: ______________________

Committee Member: Krishnendu Chatterjee, IST Austria, Klosterneuburg, Austria

Signature: ______________________

Committee Member: Pavol Černý, UC Boulder, Boulder, CO, USA

Signature: ______________________

Committee Member: Laura Kovács, TU Vienna, Vienna, Austria

Signature: ______________________

Defense Chair: Robert Seiringer, IST Austria, Klosterneuburg, Austria

Signature: ______________________

I hereby declare that this thesis is my own work and that it does not contain other people's work without this being so stated; this thesis does not contain my previous work without this being stated, and the bibliography contains all the literature that I used in writing the dissertation.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee, and that this thesis has not been submitted for a higher degree to any other university or institution.

I certify that any republication of materials presented in this thesis has been approved by the relevant publishers and co-authors.

Signature: ___________________________

Thorsten Tarrach

7th July 2016

Abstract

In this thesis we present a computer-aided programming approach to concurrency. Our approach helps the programmer by automatically fixing concurrency-related bugs, i.e. bugs that occur when the program is executed using an aggressive preemptive scheduler, but not when using a non-preemptive (cooperative) scheduler. Bugs are program behaviours that are incorrect w.r.t. a specification. We consider both user-provided explicit specifications in the form of assertion statements in the code as well as an implicit specification. The implicit specification is inferred from the non-preemptive behaviour. Let us consider sequences of calls that the program makes to an external interface. The implicit specification requires that any such sequence produced under a preemptive scheduler should be included in the set of sequences produced under a non-preemptive scheduler.

We consider several semantics-preserving fixes that go beyond atomic sections typically explored in the synchronisation synthesis literature. Our synthesis is able to place locks, barriers and wait-signal statements and last, but not least reorder independent statements. The latter may be useful if a thread is released to early, e.g., before some initialisation is completed. We guarantee that our synthesis does not introduce deadlocks and that the synchronisation inserted is optimal w.r.t. a given objective function.

We dub our solution *trace-based synchronisation synthesis* and it is loosely based on counterexample-guided inductive synthesis (CEGIS). The synthesis works by discovering a trace that is incorrect w.r.t. the specification and identifying ordering constraints crucial to trigger the specification violation. Synchronisation may be placed immediately (greedy approach) or delayed until all incorrect traces are found (non-greedy approach). For the non-greedy approach we construct a set of global constraints over synchronisation placements. Each model of the global constraints set corresponds to a correctness-ensuring synchronisation placement. The placement that is optimal w.r.t. the given objective function is chosen as the synchronisation solution.

We evaluate our approach on a number of realistic (albeit simplified) Linux device-driver benchmarks. The benchmarks are versions of the drivers with known concurrency-related bugs. For the experiments with an explicit specification we added assertions that would detect the bugs in the experiments. Device drivers lend themselves to implicit specification, where the device and the operating system are the external interfaces. Our experiments demonstrate that our synthesis method is precise and efficient. We implemented objective functions for coarse-grained and fine-grained locking and observed that different synchronisation placements are produced for our experiments, favouring e.g. a minimal number of synchronisation operations or maximum concurrency.

Acknowledgments

I would like to thank all those people who helped me in the last four years to finish this thesis. First and foremost Thomas A. Henzinger whose advice proved invaluable to guide me along the way. Despite his responsibilities as a president he always has time for his students. Pavol Černý and Arjun Radhakrishna helped me get a smooth start with my PhD by involving me into the project that would eventually make up this thesis. But before even starting my PhD at IST I visited Laura Kovács at TU Vienna who encouraged me to persue a PhD degree. I would also like to thank Roderick Bloem and Bettina Könighofer for the many interesting discussions and for inviting me to Graz.

Thanks to the professors from my thesis committee for reviewing this thesis: Roderick Bloem, Krishnendu Chatterjee, Pavol Černý, Thomas A. Henzinger and Laura Kovács.

Further I am grateful to my collaborators Ashutosh Gupta, Pavol Černý, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk and Roopsha Samanta for the fruitful collaborations we had. Without them I would not have been able to conduct this research.

My office mates Przemysław Daca, Mirco Giacobbe, Andreas Pavlogiannis, Arjun Radhakrishna, Jakob Rueß and the usual coffee break crowd consisting of Sergiy Bogomolov, Ventsislav Chonev, Rasmus Ibsen-Jensen, Bernhard Kragl, Petr Novotný, Jan Otop and Roopsha Samanta made my time at IST a much more pleasent experience

Last but not least I would like to thank my family for their continued support even though I did not have as much time for them as I would have liked.

About the Author

Thorsten Tarrach completed a BSc and MSc in Computer Science at the University of Saarland in Saarbrücken, Germany in 2010. After two years in the industry as a software engineer he joined IST in 2012. During his PhD studies he worked on synthesis of synchronisation for concurrent programs. The PhD was concluded in 2016.

Record of Publications

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

The formalisation in Chapter 2 and the synthesis technique presented in Chapter 6 is an extension of our work that appeared in [Černý *et al.*, 2015b]. We extended said paper by including a proof for Theorem 6.4.5 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. This part of the work is to appear in the Formal Methods in System Design journal.

Chapters 3, 4 and 5 contain the research from our publications [Černý *et al.*, 2013], [Černý *et al.*, 2014] and [Gupta *et al.*, 2015] respectively. The presentation has been modified w.r.t. the original papers to match the formalisation presented in Chapter 2. This is especially true for Chapter 3 that was heavily modified to use the same formalisation as Chapter 4.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CEGAR Counterexample guided abstraction refinement

CEGIS Counterexample guided inductive synthesis

HB Happens-before

LOC Lines of Code

mutex mutual exclusion

NFA Non-deterministic finite automaton

PACES Positive- and Counter-Examples in Synthesis

Chapter 1

Introduction

The main goal of this thesis is to introduce a technique for computer-aided programming where the programmer is helped with the hardest aspect of concurrent programs: synchronisation. A concurrent program allows several threads of statements to run at the same time. Such programs typically require synchronisation to ensure threads do not interfere with each other. Our technique helps the programmer by automatically repairing a concurrent program by adding missing synchronisation to the source code.

1.1 Motivation

The recent interest in concurrency in both industry and academia stems from developments in the CPU market. While the single-core performance has stagnated, the number of cores in today's CPUs is growing. This means programmers can only improve the performance of their algorithms by parallelising them. A second motivation for concurrency is to ensure that systems stay responsive even during long running computations. Even single-core systems may create the illusion of running threads in parallel using time-sharing, i.e. quickly switching between threads. Such a switch is called a *context-switch*. In this work we assume a single core that uses frequent context-switches to run threads concurrently. The work would also apply to a hypothetical multi-core system that guarantees sequential consistency of memory accesses. Common multi-core systems today do not offer such guarantees, so that additional synthesis steps would be required to enforce memory consistency.

However, implementing concurrent programs is difficult as it requires great care not to introduce errors due to missing synchronisation. Without synchronisation concurrent systems offer no guarantee as to the relative progress between threads. We consider in this thesis shared-memory concurrency, where each threads has a local state and read/write access to a shared state. Other types of concurrency are discussed in Section 1.3. Lack of synchronisation may lead to an unintended shared state if two threads manipulate the same variables concurrently. Such states outside the specification are considered *bugs*.

Our work aims to discover and fix concurrency bugs. Concurrency bugs only occur because of scheduler choices on the interleaving of threads. The stochastic nature of these bugs make them hard to discover by traditional testing.

Techniques to automatically discover bugs have a long research history and are important for critical applications, such as software used in the automotive or aviation industry. They are also used in hardware design as replacing hardware is much more expensive. The term *Formal Methods* is a term used to describe such techniques. They all have in common that a formal specification is used to enable automatic bug discovery. Bug discovery methods for concurrent programs range from the discovery of common bug patterns [Jin *et al.*, 2012], race detection [Savage *et al.*, 1997] to software verification.

Software verification is a method to prove software conforms to its specification. It was pioneered in the 60s with Floyd-Hoare triples [Floyd, 1967; Hoare, 1969] for manual verification, and later automated using *model checking* [Clarke and Emerson, 1982; Queille and Sifakis, 1982]. To this day automated software verification remains a challenge because depending on whether the state space is finite it is either undecidable or PSPACE-complete [Esparza, 1997; Schnoebelen, 2002], meaning for real-world concurrent programs there is no efficient model-checking algorithm available. This lead to the development of sound, but incomplete algorithms that will find all bugs, but also return false-positives. In practice this leads to a huge number of false-positive bug reports the programmer has to comb through. An example for a model-checker that can deal with concurrent programs is CBMC [Clarke *et al.*, 2004]. In our experience it cannot verify programs with more than 400 lines of code and 5 threads in reasonable time.

We want to go one step further and not just discover concurrency bugs, but also assist programmers in fixing these bugs automatically. The programmer would provide a program that is sequentially correct and tested enough to ensure that there are no bugs in the sequential execution. This is a reasonable assumption as programmers have developed and tested sequential programs for decades. Our tool then finds concurrency bugs and automatically modifies the source code to eliminate the bug by adding synchronisation constructs.

This type of program repair is a form of *synthesis*. Traditional, functional synthesis, which attempts to generate whole programs from specifications, goes back to the 60s [Green, 1969; Waldinger and Lee, 1969]. Despite half a century of research, synthesis of full programs is still out of reach. Our synthesis is less ambitious: We only aim to introduce isolated synchronisation primitives into the code.

Our synthesis is based on verification: A verifier is used to find a bug in the program that we fix using our synthesis. This means we inherit the disadvantages of automated program verification, such as false-positives. In our case false-positives would lead to additional synchronisation, which may negatively impact performance, but not introduce incorrect behaviour. In general, introduction of synchronisation in the program code may cause *deadlocks*. This is a state where some threads of the program are blocked indefinitely and the program cannot terminate. To illustrate this, take the most common type of synchronisation primitive, locks, which ensure that certain areas of the program are mutually exclusive. A thread tid_1 trying to acquire a lock currently held by thread tid_2 has to wait for the lock to become available. If both threads try to acquire a lock currently held by the other, none can make progress. This is a classic example for a deadlock. In our synthesis we take care not to introduce deadlocks.

Finally, we envision a system where the programmer may write code completely ignoring concurrency and the necessary synchronisation is automatically inserted using our synthesis technique. While in this thesis we introduce all the needed techniques, they do not scale to real-world programs.

Though our work is applicable to a wide variety of programs, we evaluated our implementation on Linux device drivers. Device drivers are a crucial pieces of code that facilitate communication between the operating system and the hardware. They must inherently support concurrency and a bug in a device driver is more devastating because it may cause the entire system to crash.

1.1.1 Synthesis approach

We observe that programmers tend to take an iterative approach to bug fixing. After a bug is discovered the programmer would typically try to reconstruct the control flow (trace) that lead to the error and then mentally abstract the trace to narrow down the root cause of the bug. The program is then fixed by modifying the source code. Finally, testing is resumed until the program is found to be correct.

We mimic this iterative approach in our synthesis, so in a sense our automatic approach proceeds in the same way as a human programmer would. Each iteration checks if the program is correct w.r.t. a formal specification. If it is incorrect a counterexample trace is produced. A *trace* is a sequence of program statements that were executed. The trace contains information which branches were taken and at which points the program makes context-switches. A counterexample trace is a witness of a specification failure and we call such traces *bad* traces.

The original counterexample trace is then generalised to a set of bad traces. Intuitively a larger set describes the root cause of the bug. Finally the program is refined by inserting synchronisation into the code, such that the set of bad traces become *infeasible*. An infeasible trace is one that cannot occur because of the semantics of the program. The generalisation is therefore a crucial step in speeding up the process, because it ensures a whole set of bad traces is removed from the program as opposed to just removing the original counterexample trace. The synthesis algorithm continues in a loop finding new bad traces until the refined program is found to be correct. We dub this approach *trace-based synchronisation synthesis*.

We consider two types of specifications: Explicit specification and implicit specification. An explicit specification is provided by the programmer in addition to the actual program, typically in the form of assertions. Assert statements contain an expression that needs to evaluate to true whenever the execution reaches the assertion. If an assertion evaluates to false the trace leading to it is considered a counterexample.

It is difficult for a programmer to provide a complete specification that will expose all possible concurrency bugs. However, our problem set-up is suitable for an implicit specification. For this we introduce two schedulers: A hypothetical *non-preemptive* scheduler that will not switch threads except at specific *preemption points* and the real-world *preemptive* scheduler that may switch threads at any point. Our implicit specification requires that the behaviour of the program under the preemptive scheduler should be the same as executing the program under the non-preemptive scheduler. For the implicit specification the counterexample is a trace that is possible under the preemptive, but not under the non-preemptive scheduler.

In the literature the most common synchronisation primitives employed for concurrent program repair are atomic sections. An atomic section is a sequence of adjacent statements and no context-switch is allowed in between executing these statements. Atomic sections are a theoretical construct and not directly implementable. There exist techniques however to convert them to locks [Cherem *et al.*, 2008]. We focus on synchronisation primitives commonly employed by programmers today, such as locks, barriers and wait-signal statements. A lock marks several sections of statements that are mutually exclusive to each other. This makes them weaker than atomic sections, which are exclusive w.r.t. to any part of the code. We also explore the reordering of independent statements as an alternative to wait-signal statements. Reordering produces more efficient code as it avoids the introduction of additional synchronisation primitives that generally have a negative runtime impact. In our study in Section 3.5.1 we confirm that reordering is a common fix for concurrency bugs in device drivers. For our work we assume a sequentially consistent memory model where the CPU does not reorder statements.

1.1.2 Techniques introduced

In this thesis we present four approaches to trace-based synchronisation synthesis increasing in sophistication. As mentioned above we investigated two kinds of specification: explicit and implicit. For the explicit specification the programmer typically annotates the source code with assertions and an off-the-shelve model-checker [Clarke *et al.*, 2004] can be used to check the program for correctness.

We start with this basic setup for our first implementation of the trace-based synchronisation synthesis technique in Chapter 3. After the model-checker returns a trace we generalise it and insert two kinds of synchronisation primitives to eliminate the bug. Firstly we try to reorder two statements from the same thread in the source code. This eliminates bugs where one threads is released too early. If that fails we place an atomic section in the code. While the model-checker respects the semantics of atomic sections and confirms the final program is correct, the atomic sections cannot be compiled to real machine code.

For statement reordering we consider only independent statements, so that swapping does not affect the semantics of the thread. However, by reordering we may still introduce new bugs in other threads that rely on the original ordering. Chapter 4 improves over the previous technique by not just analysing counterexamples, but also learning from good traces. A *good trace* is one that does not violate the specification. During our synthesis we use the model-checker to generate both good and bad traces. When we eliminate bad traces using statement reordering we guarantee that none of the observed good traces becomes bad. This means that w.r.t. to the good traces observed to this point we guarantee there are no regressions.

As atomic sections are not directly implementable in most commonly used programming languages we improved our synthesis to generate common synchronisation primitives, such as locks, barriers and wait-signals in Chapter 5. Synthesising a lock is more challenging because it is exclusive only w.r.t. code in other threads protected by the same lock. We accomplish this by employing a theorem prover to generalise the counterexample traces. Using the prover we obtain a formula of ordering constraints that includes all bad traces of the program. This formula may contain disjunctions and conjunctions, e.g. statement `A` has to happen before `B` or `C` before `D`. We have patterns that match parts of these formulæ and each pattern corresponds to a synchronisation primitive that eliminates the traces. While this changes the synthesis step we still employ an off-the-shelf model-checker in a loop to find counterexample traces we analyse.

Finally, in Chapter 6 we move to an implicit specification: We require that the program exhibits the same observable behaviour under the preemptive and the non-preemptive scheduler. This can no longer be checked using an off-the-shelf model-checker; we develop an algorithm based on automata language-inclusion. This works by translating the control-flow of the program into two automata, a non-preemptive one that serves as the specification and a preemptive one that represents all possible traces. The language-inclusion checks if all behaviours possible under the preemptive automaton are also possible under the non-preemptive automaton. The language these automata produce is the sequence of read/write access to the memory. If these sequences are identical we know that the result computed is the same. We can relax the restriction that they need to be identical because two reads or access to different memory loctions do not interfere with each other. The synchronisation synthesis discovers critical sections in the same way as in Chapter 5, but does not insert locks greedily. Only once all critical sections are known our tool inserts locks into the source code. This allows us to optimise the lock placement, for example preferring a minimal number of lock statements or to minimise the number of statements protected by a lock. Depending on the platform and workload of the program one may be better than the other in terms of performance.

1.2 Illustrative Example

For a very simple example we assume two threads that increment a shared variable `x` by 1. To make things more interesting a variable `y` is assigned in both as well. The example is given in

Figure 1.1 Example with two threads incrementing $\mathtt{x}$

thread1	thread2
ℓ_1: $\mathtt{l1} := \mathtt{x}$	$\ell_{\mathtt{A}}$: $\mathtt{l22} := \mathtt{y}$
ℓ_2: $\mathtt{l11} := \mathtt{y}$	$\ell_{\mathtt{B}}$: $\mathtt{l2} := \mathtt{x}$
ℓ_3: $\mathtt{x} := \mathtt{l1} + 1$	$\ell_{\mathtt{C}}$: $\mathtt{x} := \mathtt{l2} + 1$

(a) Example

thread1	thread2
ℓ_1: $\mathsf{read}_1(\mathtt{x})$	$\ell_{\mathtt{A}}$: $\mathsf{read}_2(\mathtt{y})$
ℓ_2: $\mathsf{read}_1(\mathtt{y})$	$\ell_{\mathtt{B}}$: $\mathsf{read}_2(\mathtt{x})$
ℓ_3: $\mathsf{write}_1(\mathtt{x})$	$\ell_{\mathtt{C}}$: $\mathsf{write}_2(\mathtt{x})$

(b) Abstraction

Figure 1.1a.

Informally the programmer's expectation is that $\mathtt{x}$ should have increased by 2 after execution of both threads. Without loss of generality let us assume the initial value of $\mathtt{x}$ is 0. Without synchronisation the following trace is possible and results in $\mathtt{x} = 1$: $\ell_1 \to \ell_2 \to \ell_{\mathtt{A}} \to \ell_{\mathtt{B}} \to \ell_{\mathtt{C}} \to \ell_3$. Intuitively the problem with this trace is that both threads read value 0 into their thread-local variables ($\mathtt{l1}$ and $\mathtt{l2}$) and both write back 1 to $\mathtt{x}$.

In order to analyse the trace we first need a formal specification. An explicit specification could be a post-condition $\mathtt{x} = \mathtt{x}' + 2$, where $\mathtt{x}'$ is the original value and $\mathtt{x}$ is the value after both threads completed. Though simple in this case in general it is difficult for the programmer to come up with the exact specification that would detect all possible race-conditions.

Our implicit specification approach ensures that the preemptive execution of the program is equivalent to the non-preemptive execution of the program. One way to define equivalence is to require that the result of the calculation is identical [Bloem *et al.*, 2014]. This approach is not applicable to reactive systems and we therefore require that the interaction with the environment is retained, so that for the environment the preemptive program appears to behave the same as the non-preemptive program. For this example the environment is the shared memory and the interactions are reads and writes. We relax the requirement that the order of memory accesses needs to be exactly the same as the order of reads w.r.t. other reads is unimportant. The same is true for accesses to different variables. In Figure 1.1b we present the abstract program with the access to global variables. There are two traces possible under the non-preemptive scheduler: `thread1` followed by `thread2` and the other way around. This results in two possible sequences of memory accesses:

(1) $\mathsf{read}_1(\mathtt{x}) \to \mathsf{read}_1(\mathtt{y}) \to \mathsf{write}_1(\mathtt{x}) \to \mathsf{read}_2(\mathtt{y}) \to \mathsf{read}_2(\mathtt{x}) \to \mathsf{write}_2(\mathtt{x})$ and

(2) $\mathsf{read}_2(\mathtt{y}) \to \mathsf{read}_2(\mathtt{x}) \to \mathsf{write}_2(\mathtt{x}) \to \mathsf{read}_1(\mathtt{x}) \to \mathsf{read}_1(\mathtt{y}) \to \mathsf{write}_1(\mathtt{x})$

Under a preemptive scheduler more sequences are possible. A sequence is considered within the specification if it can be transformed into a sequence of the non-preemptive scheduler. As a transformation we allow that adjacent accesses can be swapped if they are independent. Accesses are independent if they are both reads or if they refer to different variables. For example assume that the scheduler starts with `thread2` and then switches to `thread1` after executing $\ell_{\mathtt{A}}$.

(3) $\mathsf{read}_2(\mathtt{y}) \rightarrow \mathsf{read}_1(\mathtt{x}) \rightarrow \mathsf{read}_1(\mathtt{y}) \rightarrow \mathsf{write}_1(\mathtt{x}) \rightarrow \mathsf{read}_2(\mathtt{x}) \rightarrow \mathsf{write}_2(\mathtt{x})$

This sequence is within the specification, because it is equivalent to sequence (1). $\mathsf{read}_2(\mathtt{y})$ is independent w.r.t. all other accesses and be moved past the accesses of `thread1`. The sequence

(4) $\mathsf{read}_2(\mathtt{y}) \rightarrow \mathsf{read}_2(\mathtt{x}) \rightarrow \mathsf{read}_1(\mathtt{x}) \rightarrow \mathsf{read}_1(\mathtt{y}) \rightarrow \mathsf{write}_1(\mathtt{x}) \rightarrow \mathsf{write}_2(\mathtt{x})$

is not within the specification as $\mathsf{read}_2(\mathtt{x})$ and $\mathsf{write}_1(\mathtt{x})$ are dependent and cannot be swapped. This sequence corresponds to the bad trace $\ell_{\mathtt{A}} \rightarrow \ell_{\mathtt{B}} \rightarrow \ell_1 \rightarrow \ell_2 \rightarrow \ell_3 \rightarrow \ell_{\mathtt{C}}$. A trace is bad if it violates the specification.

Trace generalisation. A bad trace is subsequently generalised to find the root cause of the bug. We introduce different generalisation techniques in various chapters, which work on the same principle: The sequence of events in the trace are seen as happens-before relations that should be relaxed as long as all traces are still bad. We consider the following bad trace π: $\ell_{\mathtt{A}} \rightarrow \ell_{\mathtt{B}} \rightarrow \ell_1 \rightarrow \ell_2 \rightarrow \ell_3 \rightarrow \ell_{\mathtt{C}}$. The trace can be represented by a happen-before formula:

$$f = hb(\ell_{\mathtt{A}}, \ell_{\mathtt{B}}) \wedge hb(\ell_{\mathtt{B}}, \ell_1) \wedge hb(\ell_1, \ell_2) \wedge hb(\ell_2, \ell_3) \wedge hb(\ell_3, \ell_{\mathtt{C}})$$

where $hb(x, y)$ describes all traces where x occurs before y. The formula f therefore represents exactly our original bad trace. We can construct a larger formula f' that contains also the happens-before relations that are implied by transitivity. For example $hb(\ell_{\mathtt{A}}, \ell_{\mathtt{B}}) \wedge hb(\ell_{\mathtt{B}}, \ell_1)$ also implies $hb(\ell_{\mathtt{A}}, \ell_1)$. We have that

$$f' = \bigwedge \{hb(\ell_x, \ell_y) | \ell_x <_\pi \ell_y\}$$

where $\ell_x <_\pi \ell_y$ denotes that ℓ_x appeared before ℓ_y in trace π.

The generalisation step consists of removing as many happens-before relations as possible from f', such that all traces represented by the HB-formula are bad. For example if we remove $hb(\ell_3, \ell_{\mathtt{C}})$ from f', the HB-formula now represents two traces: the original π and the trace $\ell_{\mathtt{A}} \rightarrow \ell_{\mathtt{B}} \rightarrow \ell_1 \rightarrow \ell_2 \rightarrow \ell_{\mathtt{C}} \rightarrow \ell_3$. As both traces are bad, removing $hb(\ell_3, \ell_{\mathtt{C}})$ is a valid generalisation. If we were to remove $hb(\ell_3, \ell_{\mathtt{C}}), hb(\ell_2, \ell_{\mathtt{C}}), hb(\ell_1, \ell_{\mathtt{C}})$ f' would contain the trace

Figure 1.2 Example with correct locks

thread1	thread2
lock(lk)	$\ell_{\texttt{A}}$: $\texttt{l22} := \texttt{y}$
ℓ_1: $\texttt{l1} := \texttt{x}$	lock(lk)
ℓ_2: $\texttt{l11} := \texttt{y}$	$\ell_{\texttt{B}}$: $\texttt{l2} := \texttt{x}$
ℓ_3: $\texttt{x} := \texttt{l1} + 1$	$\ell_{\texttt{C}}$: $\texttt{x} := \texttt{l2} + 1$
unlock(lk)	unlock(lk)

$\ell_{\texttt{A}} \to \ell_{\texttt{B}} \to \ell_{\texttt{C}} \to \ell_1 \to \ell_2 \to \ell_3$, which conforms to the specification. Therefore the constraint $hb(\ell_1, \ell_{\texttt{C}})$ must be part of generalisation of f'. The most general HB-formula that represents all bad traces in the program is

$$f'' = hb(\ell_1, \ell_{\texttt{C}}) \land hb(\ell_{\texttt{B}}, \ell_3)$$

Informally this means that both threads have to read x before either thread writes to it.

Synchronisation synthesis. As f'' represents all bad traces, the negation represents all good traces of the program.

$$\neg f'' = hb(\ell_{\texttt{C}}, \ell_1) \lor hb(\ell_3, \ell_{\texttt{B}})$$

From this HB-formula we can infer a lock. The formula $\neg f''$ mandates that either the statement $\ell_{\texttt{C}}$ happens before ℓ_1 or ℓ_3 happens before $\ell_{\texttt{B}}$. This is exactly what a lock around ℓ_1, ℓ_2, ℓ_3 and $\ell_{\texttt{B}}, \ell_{\texttt{C}}$ enforces. The final program is shown in Figure 1.2.

1.3 Related Work

Concurrency

There are different notions of concurrency. First let us consider true concurrency as introduced in Petri nets [Petri, 1962]. True concurrency places no restrictions on the order of events. Mazurkiewicz traces [Mazurkiewicz, 1987] are partial-order traces used to describe the behaviour of a concurrent system. The trace does not enforce an order between events that can occur concurrently.

Contrary, interleaved concurrency describes a model where a scheduler orders the events. One model are message-passing systems. A message passing system is a set of independent processes that communicate with each other using one or more channels they send and receive messages from. The messages on the channel synchronise the processes. Hoare introduced a first notation of message-passing concurrency: Communicating Sequential Processes [Hoare, 1978]. This evolved into the Calculus of Communicating Systems [Milner, 1980] and was later extended to the π-calculus [Milner, 1992].

Shared-memory concurrency is another model of interleaved concurrency. We use this model in this thesis as it is the model used by multi-threaded programs. It describes a system where concurrently running processes communicate by accessing a common shared memory. This roots back to [Dijkstra, 1968; Cadiou and Levy, 1973]. Traces of shared-memory concurrent programs represent a total order over the events of all process.

Reasoning about concurrent programs. Software verification was pioneered in the 60s by [Floyd, 1967] and [Hoare, 1969] who independently invented what is today known as the Floyd-Hoare triples. Their research was based on earlier work in [McCarthy, 1960; McCarthy, 1962]. These triples essentially capture the semantics of a statement in an imperative program in a formal way. For each possible program statement the corresponding triple defines a precondition that must hold before executing the statement and a postcondition that is guaranteed to hold afterwards. This was intended to construct manual proofs of a sequential program, which is, however, only feasible for trivially small programs.

Verification was motivated by the need to reason systematically over concurrent programs. Concurrent programs can have hard to identify bugs, such as race conditions, which makes it desirable to reason about them systematically. A proof system for shared-variable concurrency was developed in [Owicki and Gries, 1976] that can show non-interference of parallel threads with respect to the correctness proof of the other threads.

Correctness specifications for concurrent programs. Up to that point all correctness properties of a program were expressed as propositional formulæ, which were proven to be invariants, and termination conditions, which were proven using ranking functions. [Pnueli, 1977] and later also [Lamport, 1980] proposed the use of temporal logic to express correctness conditions of reactive systems. A reactive system runs indefinitely, accepting input and producing output. Temporal logic is suitable to reason over reactive systems because it allows one to express conditions that must be met in future states (as seen from the state currently under observation). This shift was motivated by concurrent programs, for which a new class of properties was explored: liveness properties. Liveness captures the notion that all threads of a program should be able to make progress eventually. If we consider critical sections for example, the safety property is that no two threads may be inside the critical section at the same time. The liveness property is that no thread may wait forever to enter the critical section (starvation). There exist several variants of temporal logic, such as LTL (linear temporal logic), CTL (computational tree logic) and CTL* (a combination of both).

Model checking

Owicki-Gries style proofs can be constructed by hand or automatically using a verification condition generator and a theorem prover. In practice it turns out to be difficult to automatically generate verification conditions for programs with loops or recursion.

[Clarke and Emerson, 1982] as well as [Queille and Sifakis, 1982] independently discovered model checking, a procedure that can mechanically determine if a program with finitely many states meets its temporal logic specification. They expressed the state space of a program as a Kripke structure, named after its author [Kripke, 1963]. A Kripke structure is basically a finite automaton where every state has a successor and a label is assigned to every state. This process is called model checking because it checks if the Kripke structure is a model for the temporal formula. If the check fails a counterexample is produced that we use for our synthesis. The strength of the procedure is that it naturally lends itself to the verification of concurrent programs. In sequential programs the only source of non-determinism is the input, making systematic exploration of all branches a feasible option. Concurrent programs have an additional source of non-determinism: The possible interleavings of concurrent processes. This space cannot be explored easily by testing, making automated verification the preferred option. Both sources of non-determinism lead to a so-called state explosion, meaning that the Kripke structure is exponential in the size of the original program. This model checking approach was later implemented in the EMC Model Checker [Clarke *et al.*, 1983; Clarke *et al.*, 1986].

State explosion due to inputs. The fact that the state explosion makes it infeasible to construct a Kripke structure for complex programs gave birth to the idea of symbolic model checking. In symbolic model checking a set of states is defined by a formula that represents the states. Symbolic execution of a statement results in a new formula that represents the states the system can be in after execution. This technique was pioneered by [McMillan, 1993], who implement it in the SMV model checker. An important precondition for symbolic model checking is a suitable representation of formulæ. [Bryant, 1986] discovered Binary Decision Diagrams (BDDs) that represent Boolean formulæ canonically and allow for inexpensive logical operations needed in verification.

Symbolic model checking represents the set of states exactly which means that the formulæ potentially have to track all program variables and the BDDs can grow exponentially. [Cousot and Cousot, 1977] introduced the concept of abstract interpretation that can be used to abstract states and reason about transitions on abstract states. The abstract state space is smaller, however transitions are sound because they overapproximates the set of concrete states reachable. [Graf and Saidi, 1997] implement this idea for a small process calculus and use the theorem prover PVS to perform operations on the abstract domain.

While abstraction preserves soundness, it is incomplete: overestimating the reachable state may lead to the erroneous conclusion that an error state is reachable. While Graf and Saidi recognised this problem, they suggested that the algorithm is rerun with a different set of predicates, leaving it to the user to refine the abstraction. [Clarke *et al.*, 2000] and [Ball *et al.*, 2001] introduced automated abstraction refinement, where once the spurious counter-example is discovered the abstraction is refined to eliminate the counter-example. In [Henzinger *et al.*, 2002] abstraction refinement was further improved to lazy abstraction. In lazy abstraction not the entire program is reverified after an abstraction refinement, but only those parts that are affected by the new predicates. This approach was implement in the BLAST model checker.

State explosion due to concurrency. In [Lipton, 1975] the concept of reduction was introduced which can greatly simplify program proofs. Proving that certain parts of the program behave the same whether they are interrupted or not allows us to reason about them as if they were atomic sections. We use his notion of left- and right-movers in our work to test if a statements interferes with another statement.

A number of techniques were developed to tackle the state explosion problem that occurs due to concurrency. One is assume-guarantee reasoning [Jones, 1983; Henzinger *et al.*, 2000], which allows a modular approach to proving concurrent programs. This is essentially a divide and conquer strategy where every concurrent process can be analysed separately assuming non-interference from the other processes. Another approach is partial-order reductions [Godefroid *et al.*, 1996] that define an equivalence class of interleavings and for each class only one representative interleaving is explored.

In this thesis we use bounded model checking (BMC), where model checking is bounded by some measure. BMC is unsound because it cannot guarantee the absence of bugs beyond the bound, but it is still able to finding difficult bugs. [Biere *et al.*, 2003] restrict the length of the paths checked, which is increased whenever no error is found (up to a certain point). Later [Qadeer and Rehof, 2005] proved BMC bounded by the number of context-switches to be decidable for Boolean programs. Despite being unsound in general, empirically it was shown that if there is no bug up to a sufficiently high bound the program is most likely bug-free [Musuvathi and Qadeer, 2007]. We use a continuation of this work, that is implemented in the model checker CBMC [Clarke *et al.*, 2004]. CBMC uses SAT instead of BBDs to reason over states.

[Elmas *et al.*, 2009] deal with the state explosion using a combination of abstraction and reduction. These two are applied alternatingly to simplify the program. The abstraction overestimates the program's behaviour while the reduction collapses several statements into a single atomic block, thereby reducing the number of possible interleavings.

Bug summarisation

Verification techniques typically provide the user with a counterexample trace if the program does not match the specification. Counterexamples of concurrent programs can be difficult to understand as they typically contain a lot of spurious context-switches unrelated to the bug. Bug summarisation techniques aim to present the user with easy to understand information, such as classifying the bug (race condition, atomicity violation, two-stage access bugs, etc.) and highlighting problematic context-switches or code sections that lead to the error.

Though this thesis focuses primarily on synthesis we also developed a bug summarisation technique based on our trace generalisation technique from Chapter 5. Our tool is able to present to the user a minimal set of required ordering constraints that will trigger the bug and a bug classification. There is a number of prior work in fault localisation, error explanation, counterexample minimisation and bug summarisation for concurrent programs.

In [Kashyap and Garg, 2008], the authors focus on shortening counterexamples in message-passing programs to a set of "crucial events" that are both necessary and sufficient to reach the bug. In [Jalbert and Sen, 2010], the authors introduce a heuristic to simplify concurrent error traces by reducing the number of context-switches. There are several papers that survey and classify common concurrency bug patterns [Farchi *et al.*, 2003; Lu *et al.*, 2008]. We can extend our bug inference rules using the bug patterns from the papers. Finally, there is a large body of work on automatic detection of specific bugs such as data races and atomicity violations [Savage *et al.*, 1997; Engler and Ashcraft, 2003; Wang *et al.*, 2010; Said *et al.*, 2011].

Synthesis

Synthesis aims to produce code from a specification and has been studied for decades, but remains a hard problem today.

Functional synthesis uses an input-output relation as a specification and was pioneered in early work by [Green, 1969] and [Waldinger and Lee, 1969]. These approaches generally cannot synthesise loops in the programs. A more recent approach by [Solar-Lezama *et al.*, 2006] is less ambitious and assumes the programmer provides the general program structure in addition to the specification, so that only straight-line parts of the program are synthesised. This partial synthesis proved to be more tractable in practice compared to the full program synthesis.

Reactive systems require a different specification because they do not terminate. The synthesis question in the automata-theoretic framework was first asked in [Church, 1962] and solved by [Rabin, 1972]. [Manna and Wolper, 1984] used temporal logics as a specification and synthesised a communicating system. Finally, [Pnueli and Rosner, 1989] and [Ramadge and Wonham, 1987] synthesised a reactive module from a temporal formula. They model the synthesis as a game where player 1 tries to satisfy the specification and player 2 models the environment and tries to violate the specification. The synthesis corresponds to finding a winning strategy for player 1. Assume-guarantee synthesis [Chatterjee and Henzinger, 2007] synthesises concurrent processes one-by-one by assuming the other process will work according to their specification.

Program repair. Our synthesis is a form of program repair. Program repair takes as input a program that almost conforms to the specification and minimally modifies the program to ensure it is correct. The works [Jobstmann *et al.*, 2005; Griesmayer *et al.*, 2006; Samanta *et al.*, 2008] introduce program repair for sequential Boolean programs.

We target concurrent programs that lack appropriate synchronisation, which our synthesis inserts. A large body of work deals with the repair of concurrent programs by inserting atomic sections and fences [Clarke and Emerson, 1982; Vechev *et al.*, 2009; Vechev *et al.*, 2010a]. The approaches in [Clarke and Emerson, 1982; Vechev *et al.*, 2009] were based on inferring synchronisation by constructing and exploring the entire product graph or tableaux corresponding to a concurrent program. In [Vechev *et al.*, 2009] the authors discuss a trade-off between permissiveness and synchronisation cost. The synthesis, however, works by adding guards to statements and the cost is the number of shared accesses in the guards. This is conceptionally very different from the standard synchronisation primitives we introduce into the code. The approach in [Vechev *et al.*, 2010a] combines synthesis with abstraction refinement. If a program violates the specification it may either be due to the abstraction not being precise enough or due to an actual bug. Then either the abstraction is refined or an atomic section is inserted. Similar to our method in Chapter 6 an atomicity constraint formula is derived for each faulty trace that eliminates the context switches. The placement of atomic sections is delayed until no more faulty traces are found (non-greedy approach). What is missing compared to our approach is a trace generalisation step, that would let their synthesis loop terminate more quickly. We do not implement abstraction refinement as it is part of the off-the-shelve verification tool we use (except in Chapter 6). Further, our approach is able to synthesise real-world synchronisation primitives, such as locks, barriers and wait-signals, as well as statement reordering. Using the lock placement technique in [Cherem *et al.*, 2008] atomic sections can be transformed into locks. Nevertheless, that approach will lead to less fine-grained locking because in the lock placement step the original specification that lead to the atomic sections being discovered is not longer taken into account. In [Deshmukh *et al.*, 2010], the authors present a technique that can directly infer locks to ensure linearisability of concurrent libraries. The main difference from this previous work is the added counterexample generalisation step that generalises counterexample traces to formulæ of ordering constraints and eliminate them by inserting appropriate synchronisation. In [Zhang and Wang, 2014] the authors devise a system that prevents harmful traces at runtime. In contrast in this work we propose to add common synchronisation constructs to the source code before compile time. A very recent program repair paper is [Khoshnood *et al.*, 2015], which is very similar to the technique we present in Chapter 5. Our technique was published earlier in [Gupta *et al.*, 2015]. All these techniques require an explicit user-provided specification in the form of assertions or post-conditions. It is very hard for the user to ensure all race conditions are discoverable by assertions. Our technique in Chapter 6 uses an implicit specification that guarantees that the synthesised program behaves as if it were executed non-preemptively.

Our program repair algorithm considers the reordering of independent statements as an alternative to placing wait-signal statements. Statement ordering has been studied before in [Solar-Lezama *et al.*, 2008] and [Vechev and Yahav, 2008], where the input is a partial order of a set of statements and the output of the synthesis is a total order of the statements. Our synthesis in Chapter 3 accepts as input a buggy program, and we reorder statements to remove the bug (while preserving the sequential semantics).

As an alternative to the above techniques [Jin *et al.*, 2012] developed CFIX. It uses predetermined patterns to detect concurrency bugs and a fixing strategy for each pattern of bug. CFIX scales very well and was shown to work on real code bases, but it is not sound in that it cannot find all concurrency bugs.

Trace generalisation

All our approaches use generalisation of counterexample traces to guide the synthesis. This novel approach allows us to focus on traces that can be obtained easily from an off-the-shelve verification tool and the generalisation helps us find the most general bug fix.

In verification, concurrent trace generalisation was used in [Sinha and Wang, 2010; Sinha and Wang, 2011]; and in [Alglave *et al.*, 2013] for detecting errors due to weak memory models. In these works the bounded verification problem is encoded in SMT by encoding the data- and control-flow. Generalisations of good traces was previously used in [Farzan *et al.*, 2013b] to create an inductive data-flow graph (iDFG) to represent a proof of program correctness. They do not attempt to use iDFGs in synthesis.

In [Weeratunge *et al.*, 2011] the authors develop a technique that is based on running the program in a profile phase to gather positive behaviours (traces) and use them as a specification for the synthesis. This has the advantage that the approach is much more scalable as positive traces are cheap to find. The disadvantage is that it may overestimate the required atomicity because some positive traces were not observed during the profiling phase.

Representations of trace sets. The encoding of trace sets used in Chapter 5 was introduced in [Wang *et al.*, 2009], and subsequently generalised in [Kahlon and Wang, 2010; Sinha and Wang, 2011]. We derive from this encoding a succinct, explicit representation of traces using formulæ over happens-before constraints. In other work, interference scenarios have been proposed in [Farzan *et al.*, 2013a] to represent concurrent executions that are behaviourally equivalent under the same input values. For sequential programs, the authors in [Basu *et al.*, 2003] represent all counterexamples of recursive programs using pushdown automata.

Implicit Specification

The concepts of sequential consistency, linearisablity and serialisability have been used as implicit specifications. Sequential consistency and linearisablity are properties of a concurrent datastructure that has a number of methods called from various threads. Sequential consistency [Lamport, 1979] requires that every concurrent execution of the threads is equivalent to a sequential interleaving of the method calls. Linearisability [Herlihy and Wing, 1990] is a stronger constraint that also requires that the relative order of calls between threads is preserved. Serialisability [Eswaran *et al.*, 1976; Papadimitriou, 1986] is a term that originated in the database world and describes a property of modern database systems that guarantee that for every concurrent execution of a set of transactions there exists a sequential execution of those transactions that results in the same state of the database.

There has been a body of work on using a non-preemptive (cooperative) scheduler as an implicit specification. The notion of cooperability was introduced in [Yi and Flanagan, 2010]. 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. While we abstract the program to variable reads and writes, [Yi and Flanagan, 2010] uses Lipton's notion of left- and right-mover to define equivalence between a preemptive and a cooperating trace.

A recent paper [Bloem *et al.*, 2014] uses implicit specifications for synchronisation synthesis. While their specification is given by sequential behaviours, ours is given by non-preemptive behaviours. This makes our approach applicable to scenarios where threads need to communicate explicitly. Further, correctness in [Bloem *et al.*, 2014] 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.

Chapter 2

Formal Framework and Problem Statement

We present the syntax and semantics of a *concrete* concurrent while language $\mathcal{W}$. While $\mathcal{W}$ (and our tools) permits non-recursive function call and return statements, we skip these constructs in the formalisation below. We conclude the section by formalising our notion of correctness for concrete concurrent programs.

2.1 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.

2.1.1 Syntax of $\mathcal{W}$ (Figure 2.1)

A concurrent program is a finite collection of threads $\langle \mathsf{T}1, \ldots, \mathsf{T}n \rangle$ where each thread is a statement written in the syntax of $\mathcal{W}$. Variables in $\mathcal{W}$ can be categorised into

- shared variables ShVar_i,
- thread-local variables LoVar_i,
- lock variables LkVar_i,
- condition variables $\mathit{CondVar}_i$ for wait-signal statements,
- guard variables GrdVar_i for assumptions and
- the special variable `atomic_level`.

The LkVar_i, $\mathit{CondVar}_i$ and GrdVar_i variables are also shared between all threads. All variables range over integers, except for guard variables that range over Booleans $\{\texttt{false}, \texttt{true}\}$. The `atomic_level` variable keeps track how deeply nested the execution is inside atomic sections. It is not a shared or local variable. It is increased by the semantics whenever an atomic section starts and decreased when an atomic section ends. Atomic sections can be nested in the code. Each statement is labelled with a unique location identifier ℓ; we denote by $\mathsf{stmt}(\ell)$ the statement labelled by ℓ.

We use $SV = \{\mathit{ShVar}_i | i \in \mathbb{N}\}$ to denote the set of shared variables, each LV_i is the set of local variables of thread T_i, and $V = SV \cup \bigcup_i LV_i$ is the set of all variables. Let $V_i = SV \cup LV_i$ denote the set of variables that can be read from and written by thread T_i.

The language $\mathcal{W}$ includes standard sequential constructs, such as assignments, loops, conditionals, and goto statements. Additional statements control the interaction between threads, such as lock, wait/signal, and yield statements. In $\mathcal{W}$, 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 $\mathcal{W}$ supports both implicit and explicit specification. The two are not meant to be used together. The assert statement is used to provide an explicit specification.

To allow an implicit specification the language $\mathcal{W}$ has two statements that allow communication with an external system: $\mathsf{input}(ch)$ reads from and $\mathsf{output}(ch, \mathit{ShExp})$ 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. The implicit specification requires that the interaction observable to the external system is identical between the preemptive and non-preemptive execution of the program. 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.

[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 $\mathtt{x}$ is a global variable can be translated to $\mathtt{l} = \mathtt{x};\ \mathtt{x} = \mathtt{l} + 1$, where $\mathtt{l}$ is a fresh local variable.

Figure 2.1 Syntax of $\mathcal{W}$

$\mathit{LbStmt} ::=$	**Labelled Statement**
$\ell : \mathit{stmt}$	Statement annotated with a location
$\mathit{LbStmt}_1; \mathit{LbStmt}_2$	Sequence of statements
$\mathit{stmt} ::=$	Statement
skip	marks the end of the thread
$\mathit{ShVar} := \mathit{LoExp}$	Assignment to shared variable
$\mathit{LoVar} := \mathit{ShExp}$	Assignment to local variable
$\mathit{ShVar} :=$ havoc	Assign non-deterministic value
$\mathit{ShVar} :=$ input(ch)	Read a value from channel ch
output$(\mathit{ch}, \mathit{ShExp})$	Write value of ShExp to channel ch
if (ShExp) then LbStmt_1 else LbStmt_2	conditional
while (ShExp) LbStmt	while loop
assert(ShExp)	Assert ShExp evaluates to $\neq 0$
await(ShExp)	Busy wait for ShExp to become $\neq 0$
lock(LkVar)	Locks the mutex lock
unlock(LkVar)	Unlocks the mutex lock
wait$(\mathit{CondVar})$	Waits for $\mathit{CondVar}$ to be signalled
wait_not$(\mathit{CondVar})$	Waits for $\mathit{CondVar}$ to be reset
signal$(\mathit{CondVar})$	Notifies condition variable
reset$(\mathit{CondVar})$	Resets condition variable
wait_reset$(\mathit{CondVar})$	Waits and resets in an atomic operation
assume(GrdVar)	Assume guard to be `true`
assume_not(GrdVar)	Assume guard to be `false`
$\mathit{GrdVar} \leftarrow \mathit{GrdExpr}$	Assigns GrdVar the result of $\mathit{GrdExpr}$
yield	Allow current thread to be descheduled
goto(ℓ)	Set the next statement to ℓ
atomic_start/atomic_end	Start/End an atomic section
$\mathit{LoExp} ::=$	Local-variable expression
c	Integer constant
LoVar	Thread-local variable
$\mathit{op}(\mathit{LoExp}_1, \ldots, \mathit{LoExp}_n)$	Operator application
$\mathit{ShExp} ::=$	Shared-variable expression
LoExp	Local-variable expression
ShVar	Shared variable
$\mathit{op}(\mathit{ShVar}, \mathit{LoExp}_1, \ldots, \mathit{LoExp}_n)$	Operator application with shared variable
$\mathit{GrdExpr} ::=$	Expression over guard variables
`true`/`false`	Boolean constant
GrdVar	Guard variable
$\mathit{boolop}(\mathit{GrdExpr}_1, \ldots, \mathit{GrdExpr}_n)$	Boolean operation

2.1.2 Semantics of $\mathcal{W}$

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

Single-thread semantics (Figure 2.2). 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 $\langle \mathcal{V}, \ell \rangle$ where $\mathcal{V}$ 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* $\mathcal{G}_{tid}$ for thread tid in a manner similar to the control-flow graph of tid. Every node of $\mathcal{G}_{tid}$ represents a single statement (basic blocks are not merged) and the node is labelled with the location identifier ℓ of the statement. The flow graph $\mathcal{G}_{tid}$ 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 labelled 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 events of tid using $\mathcal{G}_{tid}$. The event last has no successors. We define $\mathsf{succ}(\ell) = \ell'$ if node $\ell : stmt$ in $\mathcal{G}_{tid}$ has exactly one outgoing edge to node $\ell' : stmt'$. 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 $\ell : stmt$ in $\mathcal{G}_{tid}$ has exactly two outgoing edges to nodes $\ell_1 : stmt_1$ and $\ell_2 : stmt_2$. Here succ_1 represents the then or the loop branch, whereas succ_2 represents the else or the $\mathsf{loopexit}$ branch.

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

In Figure 2.2, 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. Note that every expression may reference only one shared variable and all other variables must be thread-local.

We use integer expressions for conditional and loop statements using the standard C interpretation of 0 meaning `false` and all other integers meaning `true`. The only rules with an observable output are:

1. HAVOC: Statement $\ell : \mathit{ShVar} := \mathsf{havoc}$ assigns shared variable ShVar a non-deterministic value (say k) and outputs the observable $(\mathit{tid}, \mathsf{havoc}, k, \mathit{ShVar})$.
2. INPUT, OUTPUT: $\ell : \mathit{ShVar} := \mathsf{input}(\mathit{ch})$ and $\ell : \mathsf{output}(\mathit{ch}, \mathit{ShExp})$ read and write values to the channel ch, and output $(\mathit{tid}, \mathsf{in}, k, \mathit{ch})$ and $(\mathit{tid}, \mathsf{out}, k, \mathit{ch})$, 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 synchronisation statements shown in Figure 2.2 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.

The $\mathsf{await}(\mathit{ShExp})$ statement is a busy-wait, short for $\mathsf{while}\ (\mathit{ShExp})\ \{\}$. The atomic start and end statements atomically increment and decrement the `atomic_level` variable. This allows for nested atomic sections to be handled correctly. A succeeding assertion (where the expression evaluates to true) does not change the state except to advance the location identifier.

Concurrent semantics. A state of a concurrent program is given by $\langle \mathcal{V}, \mathit{ctid}, (\ell_1, \ldots, \ell_n)\rangle$ where $\mathcal{V}$ 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 location identifiers of the statements to be executed next in threads T_1 to T_n, respectively. There are three additional states: $\langle\mathtt{terminated}\rangle$ indicates the program has finished, $\langle\mathtt{failed}\rangle$ indicates an assertion failed and $\langle\mathtt{invalid}\rangle$ indicates an assumption failed. Initially, all integer program variables and ctid equal 0, all guard variables equal `false` and for each $i \in [1, n] : \ell_i = \mathtt{first}_i$. We introduce a non-preemptive and a preemptive semantics. The former is used as an implicit specification of allowed executions, whereas the latter models concurrent sequentially consistent executions of the program.

Non-preemptive semantics (Figure 2.3). 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

Figure 2.2 Single-thread semantics of $\mathcal{W}$

$$\frac{\mathsf{stmt}(\ell) = \mathit{ShVar} := \mathit{LoExp} \qquad \mathit{LoExp}[v/\mathcal{V}[v]] = k}{\langle \mathcal{V}, \ell \rangle \xrightarrow{\epsilon} \langle \mathcal{V}[\mathit{ShVar} := k], \mathsf{succ}(\ell) \rangle} \ \textsc{Assignment}$$

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

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

$$\frac{\mathsf{stmt}(\ell) = \mathsf{output}(\mathit{ch}, \mathit{ShExp}) \qquad \mathit{ShExp}[v/\mathcal{V}[v]] = k}{\langle \mathcal{V}, \ell \rangle \xrightarrow{(\mathit{tid}, \mathsf{out}, k, \mathit{ch})} \langle \mathcal{V}, \mathsf{succ}(\ell) \rangle} \ \textsc{Output}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{if}\ (\mathit{ShExp})\ \mathsf{then}\ \mathit{LbStmt}_1\ \mathsf{else}\ \mathit{LbStmt}_2 \qquad \mathit{ShExp}[v/\mathcal{V}[v]] \neq 0}{\langle \mathcal{V}, \ell \rangle \xrightarrow{\epsilon} \langle \mathcal{V}, \mathsf{succ}_1(\ell) \rangle} \ \textsc{If1}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{if}\ (\mathit{ShExp})\ \mathsf{then}\ \mathit{LbStmt}_1\ \mathsf{else}\ \mathit{LbStmt}_2 \qquad \mathit{ShExp}[v/\mathcal{V}[v]] = 0}{\langle \mathcal{V}, \ell \rangle \xrightarrow{\epsilon} \langle \mathcal{V}, \mathsf{succ}_2(\ell) \rangle} \ \textsc{If2}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{while}\ (\mathit{ShExp})\ \mathit{LbStmt} \qquad \mathit{ShExp}[v/\mathcal{V}[v]] \neq 0}{\langle \mathcal{V}, \ell \rangle \xrightarrow{\epsilon} \langle \mathcal{V}, \mathsf{succ}_1(\ell) \rangle} \ \textsc{While1}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{while}\ (\mathit{ShExp})\ \mathit{LbStmt} \qquad \mathit{ShExp}[v/\mathcal{V}[v]] = 0}{\langle \mathcal{V}, \ell \rangle \xrightarrow{\epsilon} \langle \mathcal{V}, \mathsf{succ}_2(\ell) \rangle} \ \textsc{While2}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{lock}(\mathit{LkVar}) \qquad \mathcal{V}[\mathit{LkVar}] = 0}{\langle \mathcal{V}, \ell \rangle \xrightarrow{\epsilon} \langle \mathcal{V}[\mathit{LkVar} := \mathit{tid}], \mathsf{succ}(\ell) \rangle} \ \textsc{Lock}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{unlock}(\mathit{LkVar}) \qquad \mathcal{V}[\mathit{LkVar}] = \mathit{tid}}{\langle \mathcal{V}, \ell \rangle \xrightarrow{\epsilon} \langle \mathcal{V}[\mathit{LkVar} := 0], \mathsf{succ}(\ell) \rangle} \ \textsc{Unlock}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{wait}(\mathit{CondVar})/\mathsf{wait\_not}(\mathit{CondVar}) \qquad \mathcal{V}[\mathit{CondVar}] = 1/0}{\langle \mathcal{V}, \ell \rangle \xrightarrow{\epsilon} \langle \mathcal{V}, \mathsf{succ}(\ell) \rangle} \ \textsc{Wait/Wait\_not}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{wait\_reset}(\mathit{CondVar}) \qquad \mathcal{V}[\mathit{CondVar}] = 1}{\langle \mathcal{V}, \ell \rangle \xrightarrow{\epsilon} \langle \mathcal{V}[\mathit{CondVar} := 0], \mathsf{succ}(\ell) \rangle} \ \textsc{Wait\_reset}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{signal}(\mathit{CondVar})/\mathsf{reset}(\mathit{CondVar})}{\langle \mathcal{V}, \ell \rangle \xrightarrow{\epsilon} \langle \mathcal{V}[\mathit{CondVar} := 1/0], \mathsf{succ}(\ell) \rangle} \ \textsc{Signal/Reset}$$

$$\frac{\begin{array}{c}\mathsf{stmt}(\ell) = \mathsf{assume}(\mathit{GrdVar})/\mathsf{assume\_not}(\mathit{GrdVar}) \\ \mathcal{V}[\mathit{GrdVar}] = \mathtt{true}/\mathtt{false}\end{array}}{\langle \mathcal{V}, \ell \rangle \xrightarrow{\epsilon} \langle \mathcal{V}, \mathsf{succ}(\ell) \rangle} \ \textsc{Assume/Assume\_not}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{assert}(\mathit{ShExp}) \qquad \mathit{ShExp}[v/\mathcal{V}[v]] \neq 0}{\langle \mathcal{V}, \ell \rangle \xrightarrow{\epsilon} \langle \mathcal{V}, \mathsf{succ}(\ell) \rangle} \ \textsc{Assertion succeeding}$$

$$\frac{\mathsf{stmt}(\ell) = \mathit{GrdVar} \leftarrow \mathit{GrdExpr} \qquad \mathit{GrdExpr}[v/\mathcal{V}[v]] = k \qquad k \in \{\mathtt{true}, \mathtt{false}\}}{\langle \mathcal{V}, \ell \rangle \xrightarrow{\epsilon} \langle \mathcal{V}[\mathit{GrdVar} := k], \mathsf{succ}(\ell) \rangle} \ \textsc{Set guard}$$

Figure 2.2 Single-thread semantics of $\mathcal{W}$ (continued)

$$\frac{\mathsf{stmt}(\ell) = \mathsf{await}(\mathit{ShExp}) \qquad \mathit{ShExp}[v/\mathcal{V}[v]] \neq 0 \qquad \ell' = \mathsf{succ}(\ell)}{\langle \mathcal{V}, \ell \rangle \xrightarrow{\epsilon} \langle \mathcal{V}, \ell' \rangle} \text{ AWAIT1}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{await}(\mathit{ShExp}) \qquad \mathit{ShExp}[v/\mathcal{V}[v]] = 0}{\langle \mathcal{V}, \ell \rangle \xrightarrow{\epsilon} \langle \mathcal{V}, \ell \rangle} \text{ AWAIT2}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{atomic\_start} \qquad \mathcal{V}[\mathtt{atomic\_level}] = k \qquad \ell' = \mathsf{succ}(\ell)}{\langle \mathcal{V}, \ell \rangle \xrightarrow{\epsilon} \langle \mathcal{V}[\mathtt{atomic\_level} := k+1], \ell' \rangle} \text{ ATOMIC START}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{atomic\_end} \qquad \mathcal{V}[\mathtt{atomic\_level}] = k \qquad k > 0 \qquad \ell' = \mathsf{succ}(\ell)}{\langle \mathcal{V}, \ell \rangle \xrightarrow{\epsilon} \langle \mathcal{V}[\mathtt{atomic\_level} := k-1], \ell' \rangle} \text{ ATOMIC END}$$

its current statement is to acquire an unavailable lock, to wait for a condition that is not signalled or the thread reached the last event. We call yield, lock, wait and wait_not statements *preemption points*. Note the difference between wait/wait_not and assume/assume_not. The former allow for a context-switch while the latter transitions to the $\langle \mathtt{invalid} \rangle$ 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, all locks are unlocked and also that we are outside any atomic section. If an assertion fails the execution transitions to the $\langle \mathtt{failed} \rangle$ state (rule ASSERTION FAILURE). The await statement causes a livelock in the non-preemptive semantics if its condition is not fulfilled.

Preemptive semantics (Figure 2.3, Figure 2.4). 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 Figure 2.4. A preemptive context-switch is only possible when outside an atomic section.

2.2 Executions and Traces

Let $\mathbb{W}$ denote the set of all concurrent programs in $\mathcal{W}$.

Executions. A *non-preemptive/preemptive execution* of a concurrent program $\mathcal{C}$ in $\mathbb{W}$ is an alternating sequence of program states and (possibly empty) observable symbols, $S_0 \ell_1 S_1 \ldots \ell_k S_k$, such that

Figure 2.3 Non-preemptive semantics

$$\frac{ctid = i \qquad \langle \mathcal{V}, \ell_i \rangle \xrightarrow{\alpha} \langle \mathcal{V}', \ell_i' \rangle}{\langle \mathcal{V}, ctid, (\ldots, \ell_i, \ldots) \rangle \xrightarrow{\alpha} \langle \mathcal{V}', ctid, (\ldots, \ell_i', \ldots) \rangle}\ \textsc{Seq}$$

$$\frac{ctid = i \quad \ell_i = \mathsf{last}_i \qquad ctid' \in \{1, \ldots, n\} \qquad \neg\mathsf{blocked}(\ell_{ctid'}, \mathcal{V})}{\langle \mathcal{V}, ctid, (\ldots, \ell_i, \ldots) \rangle \xrightarrow{\epsilon} \langle \mathcal{V}, ctid', (\ldots, \ell_i, \ldots) \rangle}\ \textsc{Thread\_end}$$

$$\frac{\begin{array}{c}\mathsf{stmt}(\ell_i) = \mathsf{lock}(lk)/\mathsf{wait}(cv)/\mathsf{wait\_not}(cv)/\mathsf{wait\_reset}(cv)/\mathsf{yield} \\ ctid = i \qquad ctid' \in \{1, \ldots, n\} \qquad \neg\mathsf{blocked}(\ell_{ctid'}, \mathcal{V})\end{array}}{\langle \mathcal{V}, ctid, (\ldots, \ell_i, \ldots) \rangle \xrightarrow{\epsilon} \langle \mathcal{V}, ctid', (\ldots, \ell_i, \ldots) \rangle}\ \textsc{Nswitch}$$

$$\frac{\forall i.\ \ell_i = \mathsf{last}_i \qquad \forall j.\ \mathcal{V}[lk_j] = 0 \qquad \mathcal{V}[\texttt{atomic\_level}] = 0}{\langle \mathcal{V}, ctid, (\ell_1, \ldots, \ell_n) \rangle \xrightarrow{\epsilon} \langle \texttt{terminated} \rangle}\ \textsc{Terminate}$$

$$\frac{ctid = i \qquad \mathsf{stmt}(\ell_i) = \mathsf{assume}(gv)/\mathsf{assume\_not}(gv) \qquad \mathcal{V}[gv] = 0/1}{\langle \mathcal{V}, ctid, (\ell_1, \ldots, \ell_n) \rangle \xrightarrow{\epsilon} \langle \texttt{invalid} \rangle}\ \textsc{Assume/Assume\_not}$$

$$\frac{ctid = i \qquad \mathsf{stmt}(\ell_i) = \mathsf{assert}(ShExp) \qquad ShExp[v/\mathcal{V}[v]] = 0}{\langle \mathcal{V}, ctid, (\ell_1, \ldots, \ell_n) \rangle \xrightarrow{\epsilon} \langle \texttt{failed} \rangle}\ \textsc{Assertion failure}$$

$$\begin{array}{rcl} \mathsf{blocked}(\ell, \mathcal{V}) & = & (\mathsf{stmt}(\ell) = \mathsf{lock}(LkVar) \wedge \mathcal{V}[LkVar] \neq 0) \\ & \vee & (\mathsf{stmt}(\ell) = \mathsf{wait}(CondVar) \wedge \mathcal{V}[CondVar] = 0) \\ & \vee & (\mathsf{stmt}(\ell) = \mathsf{wait\_not}(CondVar) \wedge \mathcal{V}[CondVar] = 1) \\ & \vee & (\mathsf{stmt}(\ell) = \mathsf{wait\_reset}(CondVar) \wedge \mathcal{V}[CondVar] = 0) \\ & \vee & (\exists i :\ \ell = \mathsf{last}_i)) \end{array}$$

Figure 2.4 Additional rule for preemptive semantics

$$\frac{ctid' \in \{1, \ldots, n\} \qquad \neg\mathsf{blocked}(\ell_{ctid'}, \mathcal{V}) \qquad \mathcal{V}[\texttt{atomic\_level}] = 0}{\langle \mathcal{V}, ctid, (\ell_1, \ldots, \ell_n) \rangle \xrightarrow{\epsilon} \langle \mathcal{V}, ctid', (\ell_1, \ldots, \ell_n) \rangle}\ \textsc{Pswitch}$$

(a) S_0 is the initial state of $\mathcal{C}$,

(b) $\forall j \in [0, k-1]$. $S_j = \langle \mathcal{V}_j, ctid_j, (\ell_j^1, \ldots, \ell_j^n) \rangle$,

(c) $\forall j \in [0, k-1]$, according to the non-preemptive/preemptive semantics of $\mathcal{W}$, we have $\ell_{ctid} = \ell_j \wedge \exists \alpha.\ S_j \xrightarrow{\alpha_{j+1}} S_{j+1}$, and

(d) S_k is the state $\langle \texttt{terminated} \rangle$ or $\langle \texttt{failed} \rangle$.

Intuitively an execution must be valid w.r.t. the non-preemptive/preemptive semantics and terminate in the end. An execution is *good* if it satisfies a given specification, and *bad* otherwise.

Traces. A trace is a sequence of location identifiers $\ell_1; \ldots; \ell_n$. We occasionally write $tid_0.\ell_1; \ldots; tid_n.\ell_n$ to highlight the thread of each statement. These two notations are equivalent as the location identifiers are unique and each location identifier is associated with exactly one thread. Alternatively, we also use $\rightarrow$ to separate location identifiers in a trace as in $\ell_1 \rightarrow \ldots \rightarrow \ell_n$. The *language* $\mathcal{L}(\pi)$ of a trace $\pi = \ell_1 \ldots \ell_n$ is the set of all executions $S_0 \ell'_1 S_1 \ldots \ell'_n S_{n+1}$ where $\ell_i = \ell'_i$ for $i \in [1, n]$.

For any two events $\ell_i, \ell_j \in \mathsf{locs}(\pi)$, we say $\ell_i <_\pi \ell_j$ if ℓ_i occurs before ℓ_j in π. A trace π is *feasible* if its language has at least one execution (i.e., $\mathcal{L}(\pi) \neq \emptyset$), and is *infeasible* otherwise. A feasible trace π is *good* if all executions in $\mathcal{L}(\pi)$ are good, and is *bad* otherwise.

We introduce the following functions that help us reason about traces and events:

Function	Explanation
$\mathsf{stmt}(\ell)$	Returns the statement for event ℓ
$\mathsf{tid}(\ell)$	Returns the thread identifier for ℓ
$\mathsf{locs}(tid)$	Returns a set of location identifiers of all statements in thread tid
$\mathsf{locs}(\pi)$	Returns a set of events of trace π

If an event repeats in a trace due to loops it is tagged with its repetition number as in the following example:

```
while(y) {
  $\ell_1$ : y := y − 1
}
$\ell_2$ : y := 2
```

The trace $\ell_1^1;\ \ell_1^2;\ \ell_1^3;\ \ell_2$ repeats event ℓ_1 three times.

2.2.1 Trace Neighbourhoods

We define equivalence classes over traces, called *neighbourhoods*. Since the trace is the result of a program run, loops are unrolled a fixed number of times. This makes the relation $<$ well-defined for loops, as the events in the loop are tagged with their repetition number. The *neighbourhood* $\mathcal{N}_\pi$ of a trace π is a set of traces $\mathcal{N}_\pi = \{\sigma \mid \mathsf{locs}(\sigma) = \mathsf{locs}(\pi) \land \forall \ell_i, \ell_j \in \mathsf{locs}(\pi).\ \mathsf{tid}(\ell_i) = \mathsf{tid}(\ell_j) \land \ell_i <_\pi \ell_j \Rightarrow \ell_i <_\sigma \ell_j\}$. Intuitively, $\mathcal{N}_\pi$ contains all traces having the same events as π and having the same order of events within each thread. A trace in $\mathcal{N}_\pi$ may be infeasible, good, or bad. We denote the subsets of good and bad traces in $\mathcal{N}_\pi$ by $\mathcal{N}_\pi^g$ and $\mathcal{N}_\pi^b$, respectively. We call $\mathcal{N}_\pi^b$ and $\mathcal{N}_\pi^g$ the *bad* and *good* neighbourhoods of π. The languages $\mathcal{L}(\mathcal{N}_\pi)$, $\mathcal{L}(\mathcal{N}_\pi^g)$ and $\mathcal{L}(\mathcal{N}_\pi^b)$ are the unions of the languages of all traces in $\mathcal{N}_\pi$, $\mathcal{N}_\pi^g$ and $\mathcal{N}_\pi^b$, respectively. We say $\ell_i \preceq_\mathcal{N} \ell_j$ if $\ell_i <_\pi \ell_j$ for all traces π in $\mathcal{N}$. We use $\ell_i \preceq \ell_j$ if the neighbourhood is clear from the context.

Note that $\mathcal{N}_\pi$ corresponds to a partial order $(\mathsf{locs}(\pi), \sqsubseteq)$, with $\ell_i \sqsubseteq \ell_j$ iff $\ell_i <_\pi \ell_j$ and $\mathsf{tid}(\ell_i) = \mathsf{tid}(\ell_j)$. However, $\mathcal{N}_\pi^g$ and $\mathcal{N}_\pi^b$ do not, in general, correspond to a partial order.

Representation of concurrent trace sets. There are multiple ways to represent trace sets. Some representations may be more expressive or useful for reasoning about concurrent programs than others. A candidate representation that has been used for certain trace sets is a partial order over events [Wang *et al.*, 2009]. The neighbourhood of a trace, as defined above, can also be represented as a partial order. However, the good neighbourhood or the bad neighbourhood of a trace is, in general, not a partial order. In our work, we represent trace sets as *HB-formulæ*. An HB-formula is a Boolean combination of *happens-before* causality constraints ($\ell_i < \ell_j$) between events. HB-formulæ can represent arbitrary finite sets of finite traces, and in particular, good and bad neighbourhoods. As we will see later, HB-formulæ are not only expressive, but also versatile enough to be usable for diverse objectives.

We represent the sequence of events from thread T with location identifiers between ℓ and ℓ' (inclusive) by $\mathsf{T}[\ell : \ell']$. We also use the symbol L to denote location identifier ranges such as $\ell : \ell'$.

Partial-order HB-formulæ and partial-order neighbourhoods. For our earlier techniques we need to restrict HB-formulæ to those representing partial orders. We call an HB-formula a *partial-order HB-formula* if it does not contain disjunctions ($\lor$) or negations ($\neg$). A *partial-order neighbourhood* is a neighbourhood that can be represented using a partial-order HB-formula.

For a partial-order neighbourhood $\mathcal{N}_\pi$, we use $\mathcal{N}_\pi \setminus hb(A, B)$ to denote the neighbourhood $\mathcal{N}_\pi$ with the constraint $hb(A, B)$ removed.

2.2.2 Representing Trace Neighbourhoods

Representing subsets of trace neighbourhoods. We represent subsets of trace neighbourhoods using *happens-before formulæ*, or, *HB-formulæ*. An HB-formula φ for a trace π is either a: (a) *basic constraint* of the form $hb(\ell_i, \ell_j)$ where $\ell_i, \ell_j \in \mathsf{locs}(\pi)$; or (b) a Boolean combination of HB-formulæ, i.e., one of $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, or $\neg\varphi_1$ where φ_1 and φ_2 are HB-formulæ.

The semantics $[\![\varphi]\!]$ of an HB-formula φ for a trace π is subset of $\mathcal{N}_\pi$, defined as follows: (a) for a basic constraint $hb(\ell_i, \ell_j)$, we have that $[\![hb(\ell_i, \ell_j)]\!] = \{\sigma \in \mathcal{N}_\pi \mid \ell_i <_\sigma \ell_j\}$; and (b) for Boolean combinations, we have that $[\![\varphi_1 \wedge \varphi_2]\!] = [\![\varphi_1]\!] \cap [\![\varphi_2]\!]$, $[\![\varphi_1 \vee \varphi_2]\!] = [\![\varphi_1]\!] \cup [\![\varphi_2]\!]$, and $[\![\neg\varphi_1]\!] = \mathcal{N}_\pi \setminus [\![\varphi_1]\!]$, respectively.

In a slight abuse of notation we write $\ell_i < \ell_j$ for $hb(\ell_i, \ell_j)$.

Remark 2.2.1. Our HB-formulæ only represent constraints on scheduling. One could define more expressive constraints which include constraints not just on scheduling, but also on variable valuations in individual executions. However, our hypothesis is that happens-before constraints on scheduling are sufficient to express many interesting properties of traces and executions. This is also supported by empirical data that shows that most concurrency bugs are due to bad ordering of statements in a trace rather than the interaction between schedules and variable valuations [Lu *et al.*, 2008].

2.3 Program Correctness and the Synthesis Problem

2.3.1 Explicit specification

We call an execution *assertion-safe* if the final state is not $\langle\texttt{failed}\rangle$. A program $\mathcal{C}$ is *assertion-safe* if all executions are assertion-safe. Intuitively, a program is assertion-safe if no failing assertion is reachable.

A trace π is called *assertion-safe* if all executions in $\mathcal{L}(\pi)$ are assertion-safe. Otherwise the trace is called *erroneous*.

2.3.2 Implicit specification

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

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

Preemption-safety. Observable behaviours ω_1 and ω_2 of a program $\mathcal{C}$ 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 $(tid, \mathsf{havoc}, k, x)$ are equal. Intuitively, observable behaviours are equivalent if they have the same interaction with the interface, and the same non-deterministic choices (havoc) in each thread. For sets $\mathcal{O}_1$ and $\mathcal{O}_2$ of observable behaviours, we write $\mathcal{O}_1 \Subset \mathcal{O}_2$ to denote that each sequence in $\mathcal{O}_1$ has an equivalent sequence in $\mathcal{O}_2$.

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

We call a program *correct* if it is preemption-safe in case of implicit specification or if it is assertion-safe in case of explicit specification.

2.3.3 Deadlock-freedom

A state S of concurrent program $\mathcal{C}$ 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 $\mathcal{C}$ can lead to S,

(b) $S \neq \langle \texttt{terminated} \rangle$,

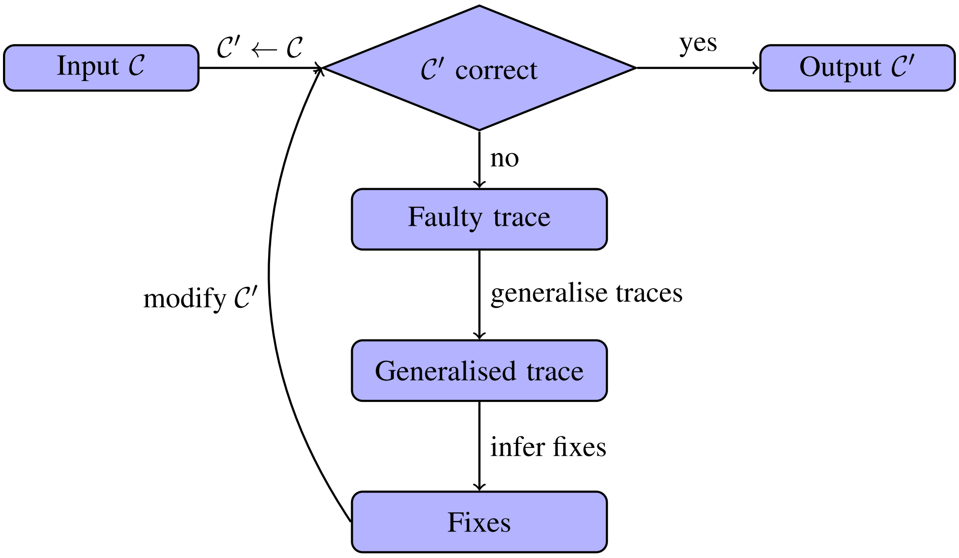
(c) $S \neq \langle \texttt{invalid} \rangle$,

(d) $S \neq \langle \texttt{failed} \rangle$ and

(e) $\neg \exists S'$: $\langle S \rangle \xrightarrow{\alpha} \langle S' \rangle$ according to the non-preemptive/preemptive semantics of $\mathcal{W}$.

Program $\mathcal{C}$ in $\mathbb{W}$ is *deadlock-free under non-preemptive/preemptive semantics* if no non-preemptive/preemptive execution of $\mathcal{C}$ hits a deadlock state. In other words, every non-preemptive/preemptive

Figure 2.5 Basic flow of trace-based synthesis



execution of $\mathcal{C}$ ends in state $\langle$`terminated`$\rangle$, $\langle$`failed`$\rangle$ or $\langle$`invalid`$\rangle$. The $\langle$`invalid`$\rangle$ state indicates an assumption did not hold, which we do not consider a deadlock. We say $\mathcal{C}$ is *deadlock-free* if it is deadlock-free under both non-preemptive and preemptive semantics.

Deadlocks are typically caused by two threads acquiring the same locks in different order or a wait statement with no concurrent thread signalling. Note that the await statement may introduce a livelock (but not a deadlock), i.e. $\langle S \rangle \xrightarrow{\epsilon} \langle S \rangle$ is possible forever while the program makes no progress.

2.3.4 Synthesis Problem

In Chapters 3 to 6 we present solutions to various synchronisation synthesis problems, that are all variants of the basic synchronisation synthesis problem: Given a concurrent program $\mathcal{C}$ in $\mathbb{W}$, the goal is to synthesise a new concurrent program $\mathcal{C}'$ in $\mathbb{W}$ such that:

(a) $\mathcal{C}'$ is obtained by adding synchronisation to $\mathcal{C}$,

(b) $\mathcal{C}'$ is correct *and*

(c) $\mathcal{C}'$ contains no deadlocks not already present in $\mathcal{C}$.

2.4 Trace-based Synthesis

Our synthesis algorithm is loosely based on *counterexample guided inductive synthesis* (CEGIS) [Solar-Lezama *et al.*, 2006]. CEGIS works in a loop that iteratively proposes a candidate solution and then checks if the solution matches the specification. If this is the case the synthesis algorithm found a correct solution and terminates. If the solution does not match the specification a counterexample is produced that is used to refine the solution for the next loop iteration.

We call our synthesis algorithm the *trace-based synchronisation synthesis*. The algorithms presented in Chapters 3 to 6 are variations of this basic synthesis algorithm.

Figure 2.5 shows the basic components of our basic algorithm. The input is the program $\mathcal{C}$ that uses either implicit or explicit specification. The algorithm works in a loop that iteratively refines the program $\mathcal{C}'$ (initially $\mathcal{C}' = \mathcal{C}$). The loop condition is whether $\mathcal{C}'$ is correct according to the specification. For programs with explicit specification we can use for example an off-the-shelf model-checker to test this. If $\mathcal{C}'$ is correct this it is returned and the algorithm terminates. If $\mathcal{C}'$ is incorrect the check will produce a faulty (counterexample) trace that explains the reason for incorrectness, for example a failed assertion is reached. The goal of the synthesis is to eliminate the buggy trace from $\mathcal{C}$. The first step to eliminate the bug is generalisation of the buggy trace. The goal of this step is to capture the “essence” of the bug. The trace may for example contain a large number of context switches, but only few are actually required to trigger the bug. The last step is to eliminate the bug by modifying $\mathcal{C}'$ such that the buggy trace and a set of similar traces become impossible. This may be achieved for example by inserting locks to remove possibilities for context switches.

Chapter 3

Synthesis for an Explicit Specification

3.1 Problem Statement and Illustrative Examples

Our first technique focuses on a specific class of synchronisation bugs that can be fixed using *statement reordering*, i.e., a rearrangement of program statements that changes the program's concurrent behaviour while preserving the sequential semantics of each thread. For example, a pointer initialisation may be moved before a statement releasing another thread that dereferences the pointer. We develop a technique for automating this type of transformation.

Our study of real-world concurrency bugs from device drivers shows that the most common fix used for 28% of driver concurrency bugs is statement reordering and only 17% of bugs are fixed using locks.

We also consider other semantics-preserving transformations inspired by practical bug-fixing techniques. For example, the synthesis tool may repeat idempotent statements multiple times (we give an example where duplicating an statement removes a concurrency bug). As a fall-back option we place atomic sections to fix other concurrency bugs.

In this approach we generalise counterexample traces to partial-order neighbourhoods (see Section 2.2.1). We first find a counterexample trace using an off-the-shelf tool, and then generalise it to a partial-order neighbourhood. We achieve this generalisation by combining ideas from Lipton reduction [Lipton, 1975] and error invariants [Ermis *et al.*, 2012]. We relax the ordering of a pair of statements in the trace if swapping these statements preserves error invariants (and thus the bug can still be reached). Intuitively, the resulting partial-order neighbourhood captures the 'true cause' of the bug. For instance, if the counterexample trace includes context switches that are not necessary to reach the bug, these context switches will not be required by the partial-order neighbourhood.

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

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

We evaluated our tool on (a) five microbenchmarks that are simplified versions of bugs from Linux device drivers, and (b) a simplified driver for the Realtek 8169 Ethernet controller. The latter had 364 LOC, seven threads, and contained five bugs. In the experiments, we found that: (a) bug finding and verification (in Q) dominates time spent generalising counterexamples, and (b) using generalised counterexamples reduces the number of bug-finding iterations.

3.1.1 Illustrative Examples

Generalising buggy traces. In Figure 3.1a, `thread1` and `thread2` concurrently increment `x`. The assertion states that `x` is 2 in the end. It fails in trace $\pi \equiv \mathtt{A} \rightarrow \mathtt{B} \rightarrow \mathtt{1} \rightarrow \mathtt{C} \rightarrow \mathtt{2} \rightarrow \mathtt{D} \rightarrow \mathtt{3} \rightarrow \mathtt{4}$, where both threads read the initial `x` value 0, and then write back 1 to `x`. However, π is just one trace exhibiting this bug. For example, swapping `B` and `1` in π gives another buggy trace. Let $\preceq_\pi$ be the HB-formula $\bigwedge_{\ell_i,\ell_j} hb(\ell_i, \ell_j)$ iff statement ℓ_i occurs before ℓ_j in π. We relax $\preceq_\pi$ by removing all constraints $hb(\ell_i, \ell_j)$ where $\mathtt{stmt}(\ell_i);\ \mathtt{stmt}(\ell_j)$ has the same effect as $\mathtt{stmt}(\ell_j);\ \mathtt{stmt}(\ell_i)$. This gives us the HB-formula $\preceq_\pi^1$ (shown in Figure 3.1c). All traces where the execution order respects $\preceq_\pi^1$ fail the assertion.

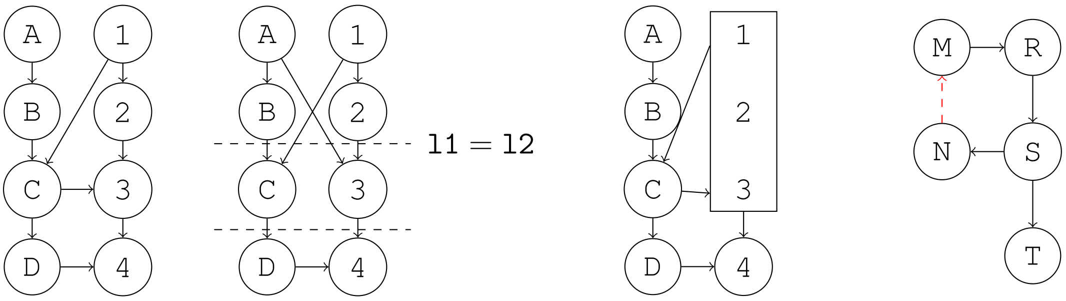
Figure 3.1 Illustrative examples

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

(a) Concurrent increment

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

(b) Interrupt handling



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

For C and 3, the sequence C; 3 is not equivalent to 3; C when $\texttt{l1} \neq \texttt{l2}$. However, in all traces of $\preceq_\pi^1$, it can be seen that $\texttt{l1} = \texttt{l2} = 1$, and further, this is sufficient to trigger the bug. These sufficient conditions to trigger bugs are *error invariants*. Using this information, we can further relax $\preceq_\pi$ to $\preceq_\pi^1$ shown in Figure 3.1d, where the only constraints are that both threads read x before either writes to it, and that D occurs before 4. A main component of our synthesis algorithm is the generalisation of buggy traces to determine their root cause.

Atomic sections. We attempt to eliminate the bug represented by $\preceq_\pi$ by adding atomic sections. For example, adding an atomic section around 1, 2, and 3 in $\preceq_\pi^1$ gives us $\preceq_\pi^{1+}$ from Figure 3.1e, where the atomic section is collapsed into a single node. Note that $\preceq_\pi^{1+}$ represents an empty neighbourhood, as there is a cycle of nodes [1; 2; 3] and C. Intuitively, the cycle implies that [1; 2; 3] happens both before and after C, which is impossible. Hence, adding an atomic section around [1; 2; 3] eliminates all traces represented by $\preceq_\pi^1$ from the program. The atomic section [1; 2; 3] does not eliminate the buggy trace A → [1; 2; 3] → B → C → D → 4. Analysing this trace similarly, we find that another atomic section [A; B; C] is needed to obtain a correct program.

The number of bug fixing iterations can be reduced using error invariants [Ermis *et al.*, 2012].

For example, in $\preceq_\pi^2$, the atomic section $[\texttt{1};\texttt{2};\texttt{3}]$ is not sufficient to create a cycle; instead, we immediately see that both $[\texttt{1};\texttt{2};\texttt{3}]$ and $[\texttt{A};\texttt{B};\texttt{C}]$ are needed. The error invariant $\texttt{l1} = \texttt{l2}$ allows the partial order to be generalised further, removing the edge between C and 3 (Figure 3.1d). This shows that error invariants help the synthesis by allowing more general partial orders to be discovered.

Instruction reordering. The example in Figure 3.1b is inspired by a real bug from a Linux device driver. Thread intr_thread runs when interrupts are enabled, i.e., IntrMask is 1, and attempts to handle them; it fails if the driver is not ready. The init_thread enables interrupts and readies the driver.

The bug is that interrupts are enabled before the driver is ready, for example, in trace $\theta \equiv \texttt{M} \rightarrow \texttt{R} \rightarrow \texttt{S} \rightarrow \texttt{N} \rightarrow \texttt{T}$. Note that statements M and N are independent, i.e., $\texttt{M};\texttt{N}$ is equivalent to $\texttt{N};\texttt{M}$. We construct an HB-formula from θ as before, but remove the constraint $\texttt{M} \preceq_\theta \texttt{N}$, resulting in Figure 3.1f (excluding the dashed edge). Adding the edge $\texttt{N} \rightarrow \texttt{M}$ creates a cycle and eliminates the bug. This edge changes the order of M and N, forcing the order $\texttt{N};\texttt{M}$. This results in a correct program with the driver ready to handle interrupts before they are enabled.

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

3.2 Semantics-preserving Transformations

We consider two kinds of transformations for fixing bugs:

- A *reordering transformation* $\theta = \ell_1 \leftrightsquigarrow \ell_2$ transforms program $\mathcal{C}$ to $\mathcal{C}'$ if location ℓ_1 immediately precedes ℓ_2 in $\mathcal{C}$ and ℓ_2 immediately precedes ℓ_1 in $\mathcal{C}'$ with everything else unchanged. We only consider cases where the sequential semantics are preserved, i.e., if (a) ℓ_1 and l_2 are from the same basic block; and (b) $\ell_1; l_2$ is equivalent to $\ell_2; \ell_1$.
- An *atomic section transformation* $\theta = [\ell_1; \ell_2]$ transforms $\mathcal{C}$ to $\mathcal{C}'$ if neighbouring locations ℓ_1 and ℓ_2 are in an atomic section in $\mathcal{C}'$, but not in $\mathcal{C}$.

We write $\mathcal{C} \xrightarrow{\theta_1 \ldots \theta_k} \mathcal{C}'$ if applying each of θ_i in order transforms $\mathcal{C}$ to $\mathcal{C}'$. We say transformation θ *acts across preemption points* if either $\theta = \ell_1 \leftrightsquigarrow \ell_2$ and one of ℓ_1 or ℓ_2 is a preemption point; or if $\theta = [\ell_1; \ell_2]$ and ℓ_2 is a preemption point.

Given a program $\mathcal{C}$, we define *program constraints* to represent sets of programs that can be obtained through applying program transformations on $\mathcal{C}$.

- *Atomicity constraint*: Program $\mathcal{C}' \models [\ell_i; \ell_j]$ if ℓ_i and ℓ_j are in an atomic block.
- *Ordering constraint*: Program $\mathcal{C}' \models \ell_i \sqsubseteq \ell_j$ if ℓ_i and ℓ_j are from the same basic block and either ℓ_i occurs before ℓ_j, or $\mathcal{C}'$ satisfies $[\ell_i; \ell_j]$.

If $\mathcal{C}' \models \Phi$, we say that $\mathcal{C}'$ *satisfies* Φ. Further, we define conjunction of Φ_1 and Φ_2 by letting $\mathcal{C}' \models \Phi_1 \wedge \Phi_2 \iff (\mathcal{C}' \models \Phi_1 \wedge \mathcal{C}' \models \Phi_2)$. We define implication as $\mathcal{C}' \models \Phi_1 \implies \Phi_2 \iff (\mathcal{C}' \models \Phi_1 \implies \mathcal{C}' \models \Phi_2)$ and its negation as $\mathcal{C}' \models \Phi_1 \not\Longrightarrow \Phi_2 \iff \neg(\mathcal{C}' \models \Phi_1 \implies \mathcal{C}' \models \Phi_2)$

A program constraint Φ is *weaker* than Φ' if $\Phi \neq \Phi'$ and

- $\forall \ell_i, \ell_j.\ (\Phi \implies [\ell_i; \ell_j]) \implies (\Phi' \implies [\ell_i; \ell_j])$ or
- $(\nexists \ell_i, \ell_j.\ \Phi' \implies [\ell_i; \ell_j] \wedge \Phi \not\Longrightarrow [\ell_i; \ell_j]) \wedge (\forall \ell_i, \ell_j.\ (\Phi \implies \ell_i \sqsubseteq \ell_j) \implies (\Phi' \not\Longrightarrow \ell_i \sqsubseteq \ell_j))$.

Intuitively, this means Φ is weaker than Φ' if Φ has fewer atomic sections or if the number of atomic sections are the same than Φ is weaker if it enforces fewer ordering constraints.

As our reorderings need to preserve the sequential semantics of the thread, we can compute some reordering constraints even before considering concurrent executions. The procedure `SemPreservingOrders` computes a program constraint Φ as follows. For each thread M it picks $\ell, \ell' \in \mathsf{locs}(M)$ such that ℓ precedes ℓ' in the original program, and checks if $\mathtt{stmt}(\ell)$ and $\mathtt{stmt}(\ell')$ commute, i.e., we test using a theorem prover two conditions: (a) $\mathtt{stmt}(\ell'); \mathtt{stmt}(\ell)$ can execute to completion from each state $\mathtt{stmt}(\ell); \mathtt{stmt}(\ell')$ can; and (b) they have the same effect. If they do not commute, then $\Phi \leftarrow \Phi \wedge \ell \sqsubseteq \ell'$.

If `SemPreservingOrders` returns Φ on input M, then every M' satisfying Φ (and obtained by reordering) is sequentially equivalent to M, and no weaker Φ' has the same property.

Example 3.2.1. *Running* `SemPreservingOrders` *on the code fragment from Figure 3.1b gives us a single constraint* $\mathtt{S} \sqsubseteq \mathtt{T}$ *as all other pairs of statements are independent of each other. In Figure 3.1a, we get* $\mathtt{A} \sqsubseteq \mathtt{B} \sqsubseteq \mathtt{C} \wedge \mathtt{1} \sqsubseteq \mathtt{2} \sqsubseteq \mathtt{3} \sqsubseteq \mathtt{4}$.

Other transformations. We motivate another sequential-semantics preserving transformation with an example. Some further transformations are in Section 3.5.1.

Example 3.2.2. *In Figure 3.2, the* `timer` *thread is invoked when* `timer_enabled = 1` *to handle requests. The device shutdown thread,* `shutdown`*, handles the remaining requests*

Figure 3.2 Example for copying idempotent statements.

```
init : timer_enabled := 1;  halted := 0
 timer                          shutdown
 atomic_start                   1 : work_queue()              work_queue() {
    A : assume(timer_enabled)   2 : timer_enabled := 0          P : unsafe()
    B : timer_enabled := 0      3 : assert(¬timer_enabled)      Q : timer_enabled := 1
    C : work_queue()                                          }
 atomic_end
```

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

The above example illustrates another useful semantics-preserving transformation, namely, replication of idempotent statements. A statement ℓ occurring after ℓ' can be replicated before ℓ' if $\mathsf{stmt}(\ell);\ \mathsf{stmt}(\ell');\ \mathsf{stmt}(\ell)$ has the same effect as $\mathsf{stmt}(\ell');\ \mathsf{stmt}(\ell)$.

3.3 Generalising Counterexamples to Partial-order Neighbourhoods

In this chapter we consider bad partial-order neighbourhoods represented by a partial-order HB-formula. We consider an execution bad if its last state is $\langle \texttt{failed} \rangle$. A bad partial-order neighbourhood contains only bad traces, i.e., for every trace there exists at least one bad execution. We do not guarantee that the bad partial-order neighbourhood of trace π contains all bad traces in $\mathcal{N}_\pi$

Error invariants. Error invariants were introduced in [Ermis *et al.*, 2012] in a sequential setting. Here we use them to generalise counterexamples to partial-order HB-formulæ. Let π be a trace $S_0\ell_1S_1\ldots\ell_kS_k$. An *error invariant ErrInv* is a function from ℓ to state assertions, such that :

(a) $\mathit{ErrInv}(\ell_1)$ represents exactly state S_0

(b) $\forall i.\ \mathit{ErrInv}(\ell_i)$ over-approximates the set of states reachable at ℓ_i along π. That is, $\mathit{ErrInv}(\ell_i)$ holds for S_i.

(c) $\forall i.\ \mathit{ErrInv}(\ell_i)$ under-approximates the set of states from which we can reach the failed state along π starting from ℓ_i. That is, if $\mathit{ErrInv}(\ell_i)$ holds for S_i, then $S_n = \langle \texttt{failed} \rangle$.

We generalise the notion of error invariant to neighbourhoods. An error invariant for $\mathcal{N}_\pi$ is a function ErrInv from location identifier ℓ to state assertions such that ErrInv is an error invariant for every trace π' in $\mathcal{N}_\pi$.

3.3.1 Generalising Counterexample Traces

Given an erroneous trace π, we now present techniques for generalising it into a bad partial-order neighbourhood $\mathcal{N}$. We assume we have discovered for each location ℓ_i in the original trace an error invariant $\mathit{ErrInv}(\ell_i) = J_{\ell_i}$.

The trace generalisation technique proceeds iteratively. Given a partial-order neighbourhood $\mathcal{N}$ represented by a partial-order HB-formula, in each step, we attempt to relax $\mathcal{N}$ by removing the conjunct $hb(\ell_a, \ell_b)$ for two location identifiers ℓ_a, ℓ_b where $\mathsf{tid}(\ell_a) \neq \mathsf{tid}(\ell_b)$. Further, we require that $\neg\exists \ell : \ell_a \preceq_\mathcal{N} \ell \preceq_\mathcal{N} \ell_b$. However, we need to ensure that the resultant neighbourhood remains bad after the relaxation, i.e., that every trace contained in it is an erroneous trace. We formalise this condition below.

Let $\ell_c, \ell_d \in \mathsf{locs}(\pi)$ be such that $\ell_c \preceq_\mathcal{N} \ell_a \preceq_\mathcal{N} \ell_b \preceq_\mathcal{N} \ell_d$ and $\forall \ell \in \mathsf{locs}(\pi).\ \ell \preceq_\mathcal{N} \ell_c \vee \ell_c \preceq_\mathcal{N} \ell \preceq_\mathcal{N} \ell_d \vee \ell_d \preceq_\mathcal{N} \ell$. Further, let $\kappa \subseteq \mathsf{locs}(\pi)$ be the set $\{\ell | \ell_c \preceq_\mathcal{N} \ell \preceq_\mathcal{N} \ell_d\} \setminus \{\ell_c, \ell_d\}$, i.e., κ represents the set of statements occur between ℓ_c and ℓ_d. We call the triple (ℓ_c, ℓ_d, κ) a *border set* of ℓ_a, ℓ_b and $\mathcal{N}_\pi$.

Let J_{ℓ_c} and J_{ℓ_d} be the error invariants at ℓ_c and ℓ_d. Intuitively, we check that we can get from J_{ℓ_c} to J_{ℓ_d} for every ordering of statements in κ allowed by $\mathcal{N} \setminus hb(\ell_a, \ell_b)$. Formally, let ℓ_1, ℓ_2, $\ldots \ell_n$ be such that each $\ell_i \in \kappa$ and $\forall \ell \in \kappa.\ \exists i.\ \ell_i = \ell$, and $\forall i.\ \ell_i \preceq_\mathcal{N} \ell_{i+1} \vee \ell_i = \ell_b \wedge \ell_{i+1} = \ell_a$. We allow relaxing the condition $hb(\ell_a, \ell_b)$ in a step if and only if the following holds: for every sequence $\ell_1;\ \ell_2;\ \ldots;\ \ell_n$ satisfying the above conditions, the Hoare-triple $\{J_{\ell_c}\}\mathsf{stmt}(\ell_1);\ \mathsf{stmt}(\ell_2);\ \ldots;\ \mathsf{stmt}(\ell_n)\{J_{\ell_d}\}$ is valid.

Therefore, the full technique for generalising a trace is as follows: We start with the neighbourhood that contains only the original trace $\mathcal{N} \equiv \bigwedge\{hb(\ell_i, \ell_j) | \ell_i <_\pi \ell_j\}$. Then, in each step, we pick ℓ_a and ℓ_b, and then check the above conditions. If they hold, we relax by removing the constraint $hb(\ell_a, \ell_b)$ from $\mathcal{N}$. Although this technique is sound and complete for generalising traces, it can be inefficient due to the large number of complex checks needed in each iteration. Instead, we present an alternative algorithm (Algorithm 3.1) that is sound, but incomplete. The outline of the algorithm is the same as the complete technique presented above, i.e., in each iteration, the algorithm attempts to relax $hb(\ell_a, \ell_b)$. However, we use two alternative checks.

Algorithm 3.1 Generalising linear counterexamples

```
Require: counterexample trace $\pi$, error invariant $ErrInv$
Ensure: bad partial-order neighbourhood $\mathcal{N}$ represented by an HB-formula
$\mathcal{N} \leftarrow \bigwedge\{hb(\ell_i, \ell_j) | \ell_i <_\pi \ell_j\}$
for all $hb(\ell_a, \ell_b)$ in $\mathcal{N}$ do
    $\mathcal{N} \leftarrow \mathcal{N} \setminus hb(\ell_a, \ell_b)$
    $\ell_c, \ell_d, \kappa \leftarrow$ borderSet($\ell_a, \ell_b, \mathcal{N}$)
    res $\leftarrow$ true
    for all $U, V \in \kappa$ do
        if $U \not\preceq V \wedge V \not\preceq U$ then
            res $\leftarrow$ res $\wedge$ (check1($\ell_u, \ell_v, \ell_c, \ell_d, \mathcal{N}, ErrInv$) $\vee$
              check2($\ell_u, \ell_v, \ell_c, \ell_d, \mathcal{N}, ErrInv$))
        end if
    end for
    if $\neg$ res then $\mathcal{N} \leftarrow \mathcal{N} \wedge hb(\ell_a, \ell_b)$
end for
return $\mathcal{N}$
```

Rule 1, implemented in procedure check1, allows relaxing the order between statements that commute under certain conditions. To relax the edge from ℓ_u to ℓ_v, we check if there exists K_1 such that $\{J_{\ell_c}\}\mathsf{stmt}(\ell_c)\{K_1\}$ is a valid Hoare-triple and $K_1 \wedge \mathsf{stmt}(\ell_v);\ \mathsf{stmt}(\ell_u) \implies K_1 \wedge \mathsf{stmt}(\ell_u);\ \mathsf{stmt}(\ell_v)$. Intuitively, we are checking if the statements at ℓ_u and ℓ_v commute given the pre-condition K_1. Further, we require that other statements do not interfere with K_1, i.e., for all $\ell \in \kappa$, K_1 is preserved under ℓ, i.e., $\{K_1\}\mathsf{stmt}(\ell)\{K_1\}$ is a valid Hoare-triple.

Rule 2, implemented in procedure check2, allows relaxing the order between statements which do not commute, but ensure the similar post-conditions in both orders. The procedure $\mathtt{check2}(\ell_u, \ell_v, \ell_c, \ell_d, \mathcal{N})$ works as follows. Let J_{ℓ_c} be the error invariant at ℓ_c, and let J_{ℓ_d} be the error invariant at ℓ_d. The procedure returns true if and only if there exists two state assertions K_1 and K_2 such that for all nodes the following conditions hold: (a) $\{J_{\ell_c}\}\mathsf{stmt}(\ell_c)\{K_1\}$, $\{K_1\}\mathsf{stmt}(\ell_u);\ \mathsf{stmt}(\ell_v)\{K_2\}$, and $\{K_1\}\mathsf{stmt}(\ell_v);\ \mathsf{stmt}(\ell_u)\{K_2\}$ are valid Hoare-triples; and (b) $K_2 \implies J_{\ell_d}$. These conditions state that the error invariants are sufficient to prove that ℓ_u and ℓ_v commute. Furthermore, let ℓ be any other node in κ. We require that ℓ preserves K_1 and

K_2, i.e., the following two Hoare-triples are valid: (c) $\{K_1\}\mathsf{stmt}(\ell)\{K_1\}$ (d) $\{K_2\}\mathsf{stmt}(\ell)\{K_2\}$. Intuitively, instead of checking all allowed paths from ℓ_c to ℓ_d, we find state assertions K_1 and K_2 that are strong enough to prove commutativity, but are preserved by other statements in κ.

Example 3.3.1.

- *Consider threads* $(\mathtt{1} : \mathtt{x} := 0;\ \mathtt{2} : \mathtt{x} := \mathtt{x}{+}1)$ *and* $(\mathtt{A} : \mathtt{x} := \mathtt{x}{+}1;\ \mathtt{B} : \mathsf{assert}(\mathtt{x} \leq 1))$*. Here,* $\mathtt{1} \rightarrow \mathtt{A} \rightarrow \mathtt{2} \rightarrow \mathtt{B}$ *is an erroneous trace. However, the ordering of* $\mathtt{A}$ *and* $\mathtt{2}$ *is irrelevant to the bug. This order can be eliminated by applying Rule 1 with precondition* $K_1 \equiv \mathtt{true}$*, as we have* $\mathtt{A}; \mathtt{2} \implies \mathtt{2}; \mathtt{A}$*.*

- *Using Rule 1 in the illustrative example (Figure 3.1a) taking* K_1 *to be* $\mathtt{l1} = 1 \wedge \mathtt{l2} = 1$ *lets us commute the statements* $\mathtt{x} := \mathtt{l1}$ *and* $\mathtt{x} := \mathtt{l2}$*.*

- *Consider two threads each with the code* $(\mathtt{1} : \mathtt{x} := \mathtt{3})$ *and* $(\mathtt{A} : \mathtt{x} := \mathtt{2};\ \mathtt{B} : \mathsf{assert}(\mathtt{x} = \mathtt{0}))$*. The erroneous trace here is* $\mathtt{1} \rightarrow \mathtt{A} \rightarrow \mathtt{B}$*. Here, it is clear that* $\mathtt{1}$ *and* $\mathtt{A}$ *cannot be relaxed by* `check1`*, because scheduling them in reverse order results in a different state. However, in the context of this trace, interchanging* $\mathtt{A}$ *and* $\mathtt{1}$ *still preserves the error. Therefore, using Rule 2 with* $K_1 \equiv \mathtt{true}$ *and* $K_2 \equiv \mathtt{x} > 0$ *relax the ordering between* $\mathtt{A}$ *and* $\mathtt{1}$*.*

We note that Rule 1 and Rule 2 provide only a sound, not a complete proof system for trace generalisation. Application of both these rules involve finding suitable K_1 and K_2. The set of conditions imposed on K_1 and K_2 can be expressed as Horn clauses. Solving Horn clauses (in logics useful for program analysis) is a focus of recent research. Non-recursive version was solved by [Gupta *et al.*, 2011a], and recursive Horn clauses are solved successfully using heuristics, for example, in [Gupta *et al.*, 2011b]. These techniques can be used to implement `check1` and `check2`.

Theorem 3.3.2. *Let* π *be a counterexample trace corresponding to an erroneous trace, and* $ErrInv$ *an error invariant for* π*. If Algorithm 3.1 returns neighbourhood* $\mathcal{N}$ *on* π *and* $ErrInv$*, then* $\mathcal{N}$ *is a bad neighbourhood of* π*.*

Now at step i, we pick a constraint in $\mathcal{N}$, and try to remove it. We remove a constraint $hb(\ell_u, \ell_v)$ if in all traces of $\mathcal{N}$ where ℓ_u is immediately followed by ℓ_v, we can swap the two and still hit the bug. That the border nodes lb and ub exist is easy to see (X is finite).

3.4 Synthesis by Elimination of Bad Neighbourhoods

We now present Algorithm 3.2 to solve the synthesis problem stated in Section 2.3.4 for programs with an explicit specification using atomic sections and reorderings. It works by finding a buggy trace, generalising it, and then eliminating it using either an atomic section, or a code reordering. The algorithm maintains a program constraint Φ. In each iteration, program $\mathcal{C}'$ which satisfies Φ is picked and verified. If correct, it is returned. Otherwise, Φ is strengthened using the generalised counterexample. Note that as `Verify` is solving an undecidable problem, it may not terminate. This results in our algorithm not terminating as well. However, as the constraint is strengthened at each step and only a finite number exist, if all calls to `Verify` terminate, then the algorithm terminates and always returns a correct program. The verification question may be undecidable because our language is Turing-complete. This correct program, in the worst case, will have every thread enclosed in an atomic section.

Algorithm 3.2 Synthesis algorithm

```
Require: Library $\mathcal{C}$
Ensure: Error-free program $\mathcal{C}'$ sequentially equivalent to $\mathcal{C}$
$\Phi \leftarrow$ SemPreservingOrders($\mathcal{C}$)
while true do
    $\mathcal{C}' \leftarrow$ Choose($\Phi$)
    if Verify($\mathcal{C}'$) return $\mathcal{C}'$
    $\mathcal{N} \leftarrow$ Generalise(cex($\mathcal{C}'$), $\Phi$)
    $\Phi \leftarrow \Phi \wedge$ Eliminate($\mathcal{N}, \Phi$)
end while
```

Algorithm `SemPreservingOrders` was defined in Section 3.2. `Generalise` is the Algorithm 3.1. Algorithm `Choose` picks a program satisfying a given constraint. `Eliminate` (see below) finds constraints to eliminate a generalised po-trace.

The basic idea behind generalised trace elimination is that $\mathcal{N}$ encodes the happens-before relation among statements and hence cannot contain loops. Hence, we aim to enforce minimal constraints to introduce a cycle in the HB-formula representing $\mathcal{N}$. We extend the HB-formula representing $\mathcal{N}$ by introducing constraints corresponding to possible atomic sections and reorderings. We then find the smallest cycles, which correspond to the required minimal constraints.

Fix a program $\mathcal{C}$ and a partial-order neighbourhood $\mathcal{N}$ for the remainder of this section. We represent $\mathcal{N}$ with an *elimination graph* $G(\mathcal{N}, \Phi) = (S, E)$, that is a weighted graph with vertices $S = \mathsf{locs}(\pi)$. Here Φ refers to the constraints from `SemPreservingOrders` and previous iterations. The edges $E \subseteq S \times \mathbb{N} \times S$ are described below. Let $\ell, \ell' \in S$. The function cons assigns a constraint to each edge of the elimination graph. We have $(\ell, w, \ell') \in E$ if:

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

Intuitively, an edge (ℓ, w, ℓ') with $\mathit{cons}((\ell, w, \ell')) = \top$ represents a happens-before relation true in any $\mathcal{C}'$ satisfying Φ. Every remaining edge (ℓ, w, ℓ') is a happens-before relation true in any program satisfying $\mathit{cons}((\ell, w, \ell'))$. We pick A much larger than R to prefer solutions having only reorderings rather than atomic sections (picking A and R such that $A > R \cdot |\mathsf{locs}(\pi)|^2$ is sufficient).

Let $\ell_0 \ldots \ell_{n-1}\ell_0$ be a cycle in the elimination graph for the partial-order neighbourhood $\mathcal{N}$ and Φ such that $\ell_0 = \ell \wedge \ell_{n-1} = \ell'$ and $\mathit{tid}(\ell) \neq \mathit{tid}(\ell')$. We call such a cycle an *elimination cycle*. We show that any elimination cycle gives us a constraint that eliminates all traces in $\mathcal{N}$. From the elimination cycle, we obtain the following constraint $\bigwedge_{i=0}^{n-2} \mathit{cons}((x_i, x_{i+1}))$. This is the constraint returned by `Eliminate` (called from Algorithm 3.2). A constraint Φ' *eliminates* a neighbourhood $\mathcal{N}$ iff all libraries satisfying $\Phi \wedge \Phi'$ and sequentially equivalent to $\mathcal{C}$ do not share a trace with $\mathcal{N}$.

Theorem 3.4.1. *Let $G(\mathcal{N}, \Phi)$ contain an elimination cycle $\ell_0\ell_1 \ldots \ell_{n-1}\ell_0$. Then, $\bigwedge_{i=0}^{n-2} \mathit{cons}((\ell_i, \ell_{i+1}))$ eliminates the partial-order neighbourhood $\mathcal{N}$.*

Proof. Assume $\pi = S_0\ell_1S_1 \ldots \ell_kS_k$ to be an execution. Any execution π in $\mathcal{C}'$ satisfying Φ and $\mathit{cons}(\ell_i, \ell_{i+1})$ has $\ell_i < \ell_{i+1}$. Hence, any trace π satisfying $\bigwedge_{i=0}^{n-2} \mathit{cons}((\ell_i, \ell_{i+1}))$ and Φ satisfies

$\ell_0 < \ell_{n-1}$. However, as (ℓ_{n-1}, ℓ_0) is an edge in the elimination graph where ℓ_0 and ℓ_{n-1} come from different threads, we have that $\ell_{n-1} \preceq_{\mathcal{N}} \ell_0$ and hence, $\ell_{n-1} < \ell_0$. This is not possible as ℓ_0 and ℓ_{n-1} correspond to different threads. Hence, every execution $\pi \in \mathcal{N}$ is eliminated by $\bigwedge_{i=0}^{n-2} cons((\ell_i, \ell_{i+1}))$. □

Further, the minimal elimination cycle corresponds to a minimal constraint. As $A > R|\mathsf{locs}(\pi)|^2$, atomic sections are used iff $\mathcal{N}$ cannot be eliminated by reordering.

Theorem 3.4.2. *If Φ is the constraint corresponding to the minimal cycle in the elimination graph, no strictly weaker constraint is sufficient to eliminate $\mathcal{N}$.*

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

Theorem 3.4.3. *Finding minimal elimination cycles in the elimination graph $G(\mathcal{N}, \Phi)$ can be done in time polynomial in the size of $\mathcal{N}$ and Φ.*

3.5 Implementation and Experiments

3.5.1 A study of concurrency bugs in Linux drivers

Our work is motivated by a study of concurrency defects in Linux device drivers. Drivers are required to perform well under concurrent workloads, which calls for sparing and fine-grained use of locks. This, in turn, provokes many concurrency-related bugs, making concurrency a major source of errors in drivers [Chou *et al.*, 2001; Ryzhyk *et al.*, 2009]. Additionally, the kernel imposes a number of constraints on the use of locks. For example, a driver may not perform blocking operations, such as acquiring a mutex, in its interrupt handler routine. Driver threads interact with other kernel threads; however, since the developer can not modify kernel code outside the driver, synchronisation with external threads must be achieved without placing additional locks in the kernel. Our study considered 100 most recent (as of Dec. 2012) concurrency-related defects fixed in Linux device drivers (we used the Linux kernel development archive obtained from `www.kernel.org`). These defects occurred in 68 different drivers, all maintained by different developers. For each bug, we rely on manual code inspection to understand the exact nature of the bug and the fix.

Table 3.1 Synchronisation patterns in Linux device drivers.

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

Figure 3.3 Examples of REORDER subpatterns and corresponding elimination graphs.

(a) REORDER.RELEASE

init : x := 0, run := 0

thread1	thread2
B : x := 1	1 : wait(run)
A : signal(run)	2 : assert(x)
~~B : x := 1~~	

(b) REORDER.LOCK

init : x := 1

thread1	thread2
B : x := 0	1 : lock(l)
D : x := 1	2 : assert(x)
C : unlock(l)	3 : unlock(l)
~~D : x := 1~~	

(c) REORDER.DELAY

init : x := 1

thread1	thread2
~~A : x := 0~~	1 : assert(x)
B : wait(exit)	2 : signal(exit)
A : x := 0	

We observed that many bug fixes involve subtle and seemingly ad hoc code transformations. In-depth analysis reveals several common patterns, shown in Table 3.1. In particular, 28 of 100 fixes were semantic-preserving statement reorderings (the REORDER pattern). These further fall into several subpatterns (see Table 3.2 and Figure 3.3). Reordering statements often involves additional side effects. For example, moving a statement across function boundaries may require adding arguments or return values to functions. Our implementation currently does not perform these, but can be extended to do so.

Interestingly, the LOCK pattern (17%) is rarer than expected. Performance and kernel-imposed constraints often prevent lock usage. This observation confirms that locks are not a universal band-aid for concurrency defects in OS code. We do not discuss remaining bug categories, but note that we encountered 23 bug fixes that did not fit into any pattern (AD HOC in Table 3.1). We expect to discover new patterns among these as we include more defects in our study.

Table 3.2 Subpatterns of the REORDER pattern.

pattern	description	example	#
REORDER.RELEASE	Move a variable assignment to a location before another thread accessing this variable is released	Figure 3.3a	11
REORDER.LOCK	Move statements to existing lock-protected section	Figure 3.3b	10
REORDER.DELAY	Delay assignment to a shared variable until a racing thread accessing this variable has terminated	Figure 3.3c	6
REORDER.RW	Reorder accesses to a pair of shared variables	Figure 3.1b	1
REORDER.ADHOC	Application-specific reordering	–	1

3.5.2 Synthesis case study

We implemented our synthesis algorithm in a tool called CONCURRENCYSWAPPER[1]. It handles a restricted subset of C, avoiding complex parts including pointer arithmetic, aliasing, bit-wise arithmetic, etc. It uses CPAChecker [Beyer and Keremoglu, 2011] to convert C statements into formulæ representing statements, as in Section 2.3.4. We use the bounded model checking tool Q [Q] to detect three kinds of bugs: (a) assertion failures; (b) generic correctness conditions (e.g., initialisation-before-use for pointers); and (c) deadlocks (as Q does not detect deadlocks, we manually encoded these as suitable assertions for our examples). We generalise buggy traces, using the Z3 theorem prover [de Moura and Bjørner, 2008] to perform the required checks for Rules 1 and 2. The current implementation does not compute invariants during generalisation; but even without invariant computation, our tool came up with the right program transformations quickly. To evaluate the effectiveness of trace generalisation, we ran the experiments with and without it.

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

Benchmarks. Our initial evaluation consisted of 5 microbenchmarks each of 15–30 lines of code without comments, and modelling a single concurrency defect found in a real Linux driver. The iterations required and fix patterns are summarised in Table 3.3. All measurements were

[1] available as open-source software along with benchmarks: `https://github.com/thorstent/ConcurrencySwapper`

Table 3.3 Micro-benchmarks

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

Table 3.4 Results for Linux Realtek 8169 driver benchmark

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

done on an Intel core i5-3320M laptop with 8GB of RAM. The synthesis took less than 15 seconds for each case, with trace analysis taking less than 0.5 seconds. Also, in 1 case, not using trace generalisation leads to an additional iteration, leading to a larger execution time.

We evaluate the scalability of CONCURRENCYSWAPPER using a simplified version of the Linux Realtek 8169 driver. This driver is representative of medium to high-end drivers both in terms of overall complexity and the complexity of synchronisation logic. We extracted the driver's complete synchronisation skeleton, including code and variables related to thread synchronisation and communication. The skeleton does not include the actual device management code, which is irrelevant to concurrency, and was additionally simplified to avoid currently unsupported C constructs. We provide an environment model to simulate all (7) OS threads that interact with the driver. The resulting skeleton had 364 LOC, while the original driver had around 7,000 LOC. The skeleton had 5 concurrency defects.

Q was able to find all the defects, and CONCURRENCYSWAPPER was able to find fixes for each defect through statement reordering. The results are summarised in Table 3.4. In each iteration, the trace analysis phase took less than 2 seconds. The extra bug finding times due to additional iterations is reported for the runs without trace generalisation. In one case, the 3 additional iterations were required without trace generalisation. The bug finding times dominate the trace analysis times, justifying the use of complex trace generalisation procedure to avoid additional iterations. The verification phase took around 30 minutes.

Chapter 4

Regression-free Synthesis

4.1 Problem Statement and Illustrative Example

In Chapter 3 we introduced a trace-based algorithm for program repair of programs with explicit specification. We applied two types of program transformations: (1) reordering of adjacent statements $\mathsf{stmt}(\ell_a);\ \mathsf{stmt}(\ell_b)$ within a thread if the statements are sequentially independent (i.e., if $\mathsf{stmt}(\ell_a);\ \mathsf{stmt}(\ell_b)$ is sequentially equivalent to $\mathsf{stmt}(\ell_b);\ \mathsf{stmt}(\ell_a)$), and (2) inserting atomic sections. Reordering of statements is given priority as it may result in a better performance than the insertion of atomic sections.

While placing atomic sections is safe, reordering statements can result in assertion violations becoming reachable that were not reachable in the original program. This is called a *regression*. In this chapter we demonstrate the use of good traces to prevent regressions.

We say that such an algorithm is *regression-free* if after every iteration, we have that: first, all bad traces examined so far are removed, and second, no good trace examined so far is turned into a bad trace of the new program. (Of course, to make this definition precise, we will need to define a correspondence between traces of the original program and the new program.)

In related work by [von Essen and Jobstmann, 2013], the goal is to repair reactive systems (given as automata) according to an LTL specification, with a guarantee that good traces do not disappear as a result of the repair. They do not deal with concurrency bugs or synchronisation.

Our algorithm. Our algorithm learns constraints on the space of candidate solutions from both good traces and bad traces. We explain constraint learning using as an example the program transformation $\ell_1 \leftrightsquigarrow \ell_2$ (Section 3.2), which reorders statements within threads. From a bad trace, we learn reordering constraints that eliminate the counterexample using Algorithm 3.2. While eliminating the counterexample, such reorderings may transform a (not necessarily preemption-free) good trace into a bad trace — this would constitute a regression. In order to avoid regressions, our algorithm learns also from good traces. Intuitively, from a good trace π, we want to learn all the ways in which π can be transformed by reordering without turning it into an error trace— this is expressed as a program constraint. The program constraint is (a) sound, if all programs satisfying the constraint are regression-free; and (b) complete, if all programs violating the constraint have regressions. However, as learning a sound and complete constraint is not possible from a single trace, given a good trace π we learn a sound constraint that only guarantees that π is not transformed into a bad trace. We generate the constraint using data-flow analysis on the statements in π. The main idea of the analysis is that in good traces, the data-flow into passing assertions is protected by synchronisation mechanisms (such as locks) and data-flow into conditionals along the trace. This protection may fail if we reorder statements. We thus find a constraint that prevents such bad reorderings.

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

CEGIS vs PACES. A popular recent approach to synthesis is counterexample-guided inductive synthesis (CEGIS) [Solar-Lezama *et al.*, 2006]. Our algorithm can be viewed as an instance of CEGIS with the important feature that we learn from positive examples. We dub this approach PACES, for *Positive- and Counter-Examples in Synthesis*. The input to the CEGIS algorithm is a specification φ (possibly in multiple pieces – say, as a temporal formula and a language of possible solutions [Alur *et al.*, 2013]). In the basic CEGIS loop, the synthesiser proposes a candidate solution S, which is then checked against φ. If it is correct, the CEGIS loop terminates; if not, a counterexample is provided and the synthesiser uses it to improve S. In practice, the CEGIS loop often faces performance issues, in particular, it can suffer from regressions: new candidate solutions may introduce errors that were not present in previous candidate solutions. We address this issue by *making use of positive examples* (good traces) in addition to counterexamples (bad traces). The good traces are used to learn constraints that ensure that these good traces are preserved in the candidate solution programs proposed by the

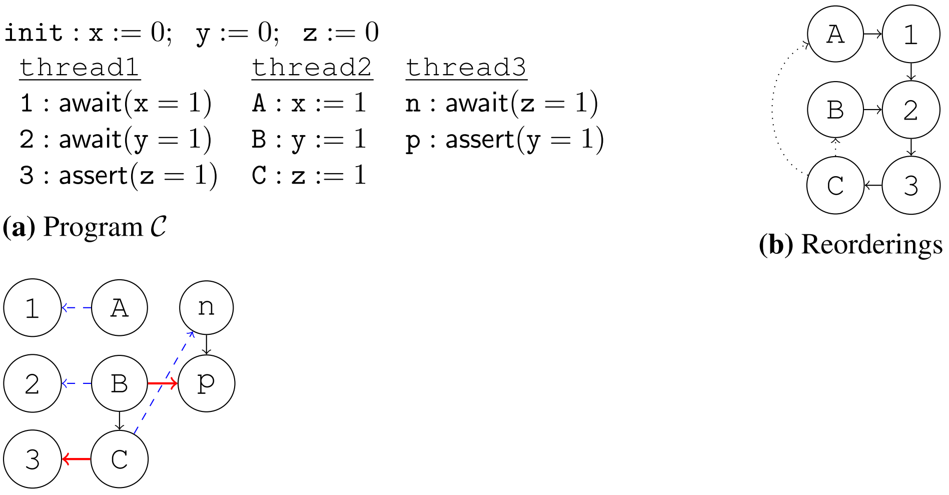
CEGIS loop. The PACES approach applies in many program synthesis contexts, but in this chapter, we focus on program repair for concurrency.

Experimental evaluation. To evaluate our approach, we implemented a program repair tool, named ConRepair, and applied it to a collection of (simplified) open-source Linux device drivers. We looked at concurrency bugs that were reported and fixed in the Linux kernel repository. We used five examples, where we modelled the concurrency skeleton in sufficient detail to reproduce the bug (between 35 and 80 lines of code per example). In addition, we did two larger case studies, of the `rtl8169` and `usb-serial` drivers, modelled in more detail, with about 400 lines of code each. As explained above, our tool tries to fix a bug by reordering statements within threads, which is the preferred solution and/or by inserting atomic sections. In each case, our tool found a solution that fixed the problem (or, in the case of `rtl8169`, multiple problems). To evaluate the impact of using positive examples, we compared ConRepair with two versions of ConcurrencySwapper (Chapter 3), which do not use positive examples. The first version of ConcurrencySwapper (`ce1`) prefers to exhaust all possible statement reorderings before using atomic sections; the second version (`ce2`) heuristically decides to insert atomic sections earlier. We found that (a) the new tool converges to a solution in a significantly smaller number of iterations than (`ce1`), and (b) the new tool finds solutions with fewer atomic sections than (`ce2`) in a comparable number of iterations. We thus conclude that the use of positive examples can substantially improve the performance and quality of counterexample-guided inductive synthesis algorithms. In theory it is possible that `ce2` inserts an atomic section earlier that is not needed.

4.1.1 Illustrative Example

We motivate our approach on the program $\mathcal{C}$ in Figure 4.1a. There is a bug witnessed by the following trace: $\pi_1 = \mathtt{A} \rightarrow \mathtt{B} \rightarrow \mathtt{1} \rightarrow \mathtt{2} \rightarrow \mathtt{3}$ (the assertion at line `3` fails). Let us attempt to fix the bug using the algorithm from Chapter 3. The algorithm discovers possible fixes by first generalising the trace into a partial order (Figure 4.1b, without the dotted edges) representing the happens-before relations necessary for the bug to occur, and second, trying to create a cycle in the partial order to eliminate the generalised counterexample. It finds three possible ways to do this: swapping `B` and `C`, or moving `C` before `A`, or moving `A` after `C`, indicated by the dotted edges in Figure 4.1b. Assume that we continue with swapping `B` and `C` to obtain program $\mathcal{C}_1$ where the

Figure 4.1 Program analysis with good and bad traces



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

(a) Program $\mathcal{C}$

(b) Reorderings from bad traces

(c) Learning from a good trace

first thread is A; C; B. Program $\mathcal{C}_1$ contains an error trace $\pi_2 = \mathtt{A} \rightarrow \mathtt{C} \rightarrow \mathtt{n} \rightarrow \mathtt{p}$ (the assertion at line p fails). This bug was not in the original program, but was introduced by our fix. We refer to this type of bug as a regression.

In order to prevent regressions, the algorithm learns from good traces. Consider the following good trace $\pi_3 = \mathtt{A} \rightarrow \mathtt{B} \rightarrow \mathtt{C} \rightarrow \mathtt{1} \rightarrow \mathtt{2} \rightarrow \mathtt{n} \rightarrow \mathtt{3} \rightarrow \mathtt{p}$. The algorithm analyses the trace, and produces the graph in Figure 4.1c. Here, the thick red edges indicate the reads-from relation for assert commands, and the dashed blue edges indicate the reads-from relation for await commands. Intuitively, the algorithm now analyses why the assertion at line p holds in the given trace. This assertion reads the value written in line B (indicated by the thick red edge). The algorithm finds a path from B to p composed entirely from intra-thread sequential edges ($\mathtt{B} \rightarrow \mathtt{C}$ and $\mathtt{n} \rightarrow \mathtt{p}$) and dashed blue edges ($\mathtt{C} \rightarrow \mathtt{n}$). This path guarantees that this trace cannot be changed by different scheduler choices into a path where p reads from elsewhere and fails. From the good trace π_2 we thus find that there could be a regression unless B precedes C and n precedes p. Having learned this constraint, the synthesiser can find a better way to fix π_1. Of the three options described above, it chooses the only way which does not reorder B and C, i.e., it moves A after C. This fixes the program without regressions.

4.1.2 Problem Statement

We use the definition of program constraints Φ introduced in Section 3.2.

Trace Transformations and Regressions. A trace $\pi = \ell_0 \ldots \ell_m$ *transforms* into a trace $\pi' = \ell'_0 \ldots \ell'_m$ by *switching* if: (a) $\ell_0 \ldots \ell_n = \ell'_0 \ldots \ell'_n$ and the suffixes $\ell_{n+2} \ldots \ell_m$ and $\ell'_{n+2} \ldots \ell'_m$ are equal; and (b) $l_n = \ell'_{n+1} \wedge \ell_{n+1} = \ell'_n$. We label switching transformations as a:

- *Free transformation* if ℓ_n and ℓ_{n+1} are from different threads. We write $\pi' \in f(\pi)$ if a sequence of free transformations takes π to π'.
- *Reordering transformation $\theta = \ell^\sharp \leftrightsquigarrow \ell^\flat$ acting on* π if $\ell_n = \ell^\sharp$ and $\ell_{n+1} = \ell^\flat$ and $\ell^\sharp, \ell^\flat$ from the same thread.

  We have $\pi' \in \theta(\pi)$ if repeated applications of reordering transformations acting on π give π'. Similarly, $\pi' \in \theta^f(\pi)$ if repeated applications of θ and free transformations acting on π give π'.

Reordering is a special form a switching because it requires changes to the order of statements in the actual program. Similarly, π' is obtained by *atomic section transformation $\theta = [\ell_1; \ell_2]$ acting on a trace* π if $\pi' \in f(\pi)$, and there are no context-switches between ℓ_1 and ℓ_2 in π' (Section 3.2).

Trace analysis graphs. We use trace analysis graphs to characterise data-flow and scheduling in a trace. First, given a trace $\pi = \ell_0 \ldots \ell_n$, we define the function $depends$ to recursively find the data-flow edges into the location ℓ_i. Formally, $depends_\pi(i) = \bigcup_v \left(\{(\mathsf{last}(i,v), i)\} \cup depends_\pi(\mathsf{last}(i,v))\right)$ where v ranges over variables read by ℓ_i, and $\mathsf{last}(i,v)$ returns j if ℓ_i reads the value of v written by ℓ_j and $\mathsf{last}(i,v) = \bot$ if no such j exists. As the base case, we define $depends_\pi(\bot) = \emptyset$.

Now, a *trace analysis graph* for trace $\pi = \ell_0 \ldots \ell_n$ is a multi-graph $G(\pi) = \langle V, \rightarrow \rangle$, where $V = \{\bot\} \cup \{i | 0 \leq i \leq n\}$ are the positions in the trace along with $\bot$ (representing the initial state) and $\rightarrow$ contains the following types of edges.

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

transformations that switch ℓ_x and ℓ_y.

Regressions. Suppose $\mathcal{C} \xrightarrow{\theta_1,\ldots,\theta_k} \mathcal{C}'$. We say $\theta_1, \ldots, \theta_k$ introduces a *regression* with respect to a good trace $\pi = \ell_0 \ldots \ell_m$ of $\mathcal{C}$ if there exists a trace $\pi' = \ell'_0 \ldots \ell'_m \in \theta^f_k \circ \ldots \circ \theta^f_1(\pi)$ such that: (a) π' is a bad trace of $\mathcal{C}'$; (b) π does not freely transform into any bad trace of $\mathcal{C}$; and (c) for every data-flow into conditionals edge $x \to y$ (say l_y reads the variables V from l_x) in $G(\pi)$, the edge $p(x) \to p(y)$ is a data-flow into conditionals edge in $G(\pi')$ (where $\ell'_{p(y)}$ reads the same variables V from $\ell'_{p(x)}$). Here, $p(i)$ is the position in π' of the statement at position i in π after the sequence of switching transformations that take π to π'. We say $\theta_1 \ldots \theta_k$ introduces a regression with respect to a set T_G of good traces if it introduces a regression with respect to at least one trace $\pi \in T_G$.

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

Remark 4.1.1. The above definition of regression attempts to capture the intuition that a good trace transforms into a "similar" bad trace. The notion of similar asks that the traces have the same data-flow into conditionals – this condition can be relaxed to obtain more general notions of regression. However, this makes trace analysis and finding regression-free fixes much harder (See Example 4.2.4).

Example 4.1.2. *In Figure 4.1, the trace* $\pi =$ `A; B; C; n; p` *transforms under* `B` $\leftrightsquigarrow$ `C` *to* $\pi' =$ `A; C; B; n; p`*, which freely transforms to* $\pi'' =$ `A; C; n; p; B`*. Hence,* `B` $\leftrightsquigarrow$ `C` *introduces a regression with respect to* π *as* π *does not freely transform into a bad trace, and* π' *is bad while the* await *in* `n` *still reads from* `C`.

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

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

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

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

4.2 Good and Bad Traces

4.2.1 Learning from Good Traces

Given a trace π of $\mathcal{C}$, a program constraint Φ is a *sound regression preventing constraint* for π if no sequence of program transformations $\theta_1, \ldots, \theta_k$, such that $\mathcal{C} \xrightarrow{\theta_1 \ldots \theta_k} \mathcal{C}'$ and $\mathcal{C}' \models \Phi$, introduces a regression with respect to π. Further, if every $\theta_1 \ldots \theta_k$, such that $\mathcal{C} \xrightarrow{\theta_1 \ldots \theta_k} \mathcal{C}'$ and $\mathcal{C}' \not\models \Phi$[1], introduces a regression with respect to π, then Φ is a *complete regression preventing constraint*.

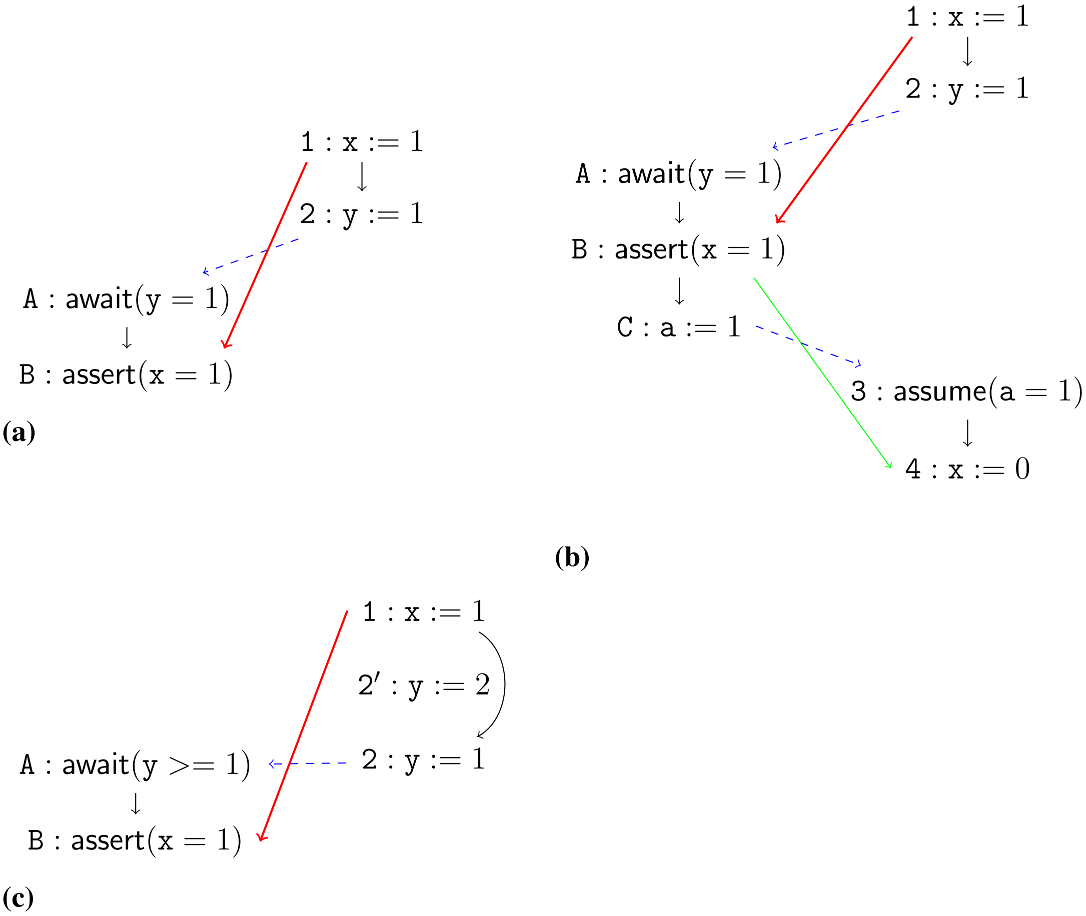
Example 4.2.1. *Let the program $\mathcal{C}$ be $\{\mathtt{1} : \mathtt{x} := \mathtt{1};\ \mathtt{2} : \mathtt{y} := \mathtt{1}\} || \{\mathtt{A} : \mathsf{await}(\mathtt{y});\ \mathtt{B} : \mathsf{assert}(\mathtt{x} = \mathtt{1})\}$. In Figure 4.2a, the constraint $\Phi^* = (\mathtt{1} \sqsubseteq \mathtt{2} \wedge \mathtt{A} \sqsubseteq \mathtt{B})$ is a sound and complete regression-preventing constraint for the trace $\mathtt{1} \rightarrow \mathtt{2} \rightarrow \mathtt{A} \rightarrow \mathtt{B}$.*

Lemma 4.2.2. *For a program $\mathcal{C}$ and a good trace π, the sound and complete regression-preventing constraint Φ^* is computable in exponential time in $|\pi|$.*

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

[1] see Section 3.2 for definition

Figure 4.2 Sample Good Traces for Regression-preventing constraints



(a)

(b)

(c)

While the complexity is exponential, we can show that this cost is unavoidable. We do not present the proof here, but only state that it is non-constructive and is based on Shannon's lower bounds on circuit complexity for boolean functions.

Lemma 4.2.3. *There exist a class of programs $\mathcal{C}_n$, and traces π_n of length $O(n)$ such that the most-general regression-preventing constraint is of size $\Theta(\frac{2^n}{n})$.*

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

Example 4.2.4. *The trace in Figure 4.2b is a simple extension of Figure 4.2a. However, the constraint $(\mathtt{1} \sqsubseteq \mathtt{2} \wedge \mathtt{A} \sqsubseteq \mathtt{B})$ (from Example 4.2.1) does not prevent regressions for Figure 4.2b. An additional constraint $\mathtt{B} \sqsubseteq \mathtt{C} \wedge \mathtt{3} \sqsubseteq \mathtt{4}$ is needed as reordering these statements can lead to the assertion failing by reading the value of $\mathtt{x}$ "too late", i.e., from the statement $\mathtt{4}$ (trace: $\mathtt{1} \rightarrow \mathtt{2} \rightarrow \mathtt{A} \rightarrow \mathtt{C} \rightarrow \mathtt{3} \rightarrow \mathtt{4} \rightarrow \mathtt{B}$).*

Figure 4.2c clarifies our definition of regression, which requires that the data-flow edges into assumptions and awaits need to be preserved. The await *can be activated by both* 2 *and* 2′*; in the trace we analyse it is activated by* 2*. Moving* 2′ *before* 1 *could activate the* await *"too early" and the assertion would fail (trace:* $2' \to \mathtt{A} \to \mathtt{B}$*). However, it is not possible to learn this purely with data-flow analysis – for example, if statement* 2′ *was* $\mathtt{y} := -1$*, then this would not lead to a bad trace. Hence, we exclude such cases from our definition of regressions by requiring that the* await *reads* A *reads from the same location.*

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

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

Given a trace $\pi = \ell_0; \ell_1 \ldots \ell_n$, where statement at position r (i.e., ℓ_r) reads a set of variables (say V) written by a statement at position w (i.e., ℓ_w), the non-interference edges define a sufficient set of happens-before relations to ensure that no other statements can interfere with the read-write pair, i.e., that every other write to V either happens before w or after r. Formally, we have that $\mathit{interfere}(w \to r) = \{r \to w' \mid w' > r \land \mathit{write}(\ell_{w'}) \cap \mathit{write}(\ell_w) \cap \mathit{read}(\ell_r) \neq \emptyset\} \cup \{w' \to w \mid w' < w \land \mathit{write}(\ell_{w'}) \cap \mathit{write}(\ell_w) \cap \mathit{read}(\ell_r) \neq \emptyset\}$ where $\mathit{read}(\ell)$ and $\mathit{write}(\ell)$ are the variables read and written at location ℓ. If $w = \bot$, we have $\mathit{interfere}(w \to r) = \{r \to w' \mid w' > r \land \mathit{write}(\ell_{w'}) \cap \mathit{read}(\ell_r) \neq \emptyset\}$.

Algorithm 4.1 works by ensuring that for each data-flow into assertions edge e, the edge itself is covered and that the interference edges are covered. For each such cover, the set of intra-thread order edges needed for the covering are conjuncted to obtain a constraint. We take the disjunction Φ' of the constraints produced by all covers of one edge and add it to a constraint

Algorithm 4.1 *LearnGoodUnder*

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

Φ to be returned. If an edge cannot be covered, the algorithm falls back by returning a constraint that fixes all current intra-thread orders. The algorithm can be made to run in polynomial time in $|\pi|$ using standard dynamic programming techniques.

Theorem 4.2.5. *Given a trace π, Algorithm 4.1 returns a constraint Φ that is a sound regression-preventing constraint for π and runs in polynomial time in $|\pi|$.*

Proof. The fallback case (line 3) is trivially sound. Let us assume towards contradiction that there is a bad trace $\pi' = \ell'_0; \ell'_1 \ldots \ell'_n$ of $\mathcal{C}' \models \Phi$, that is obtained by transformation of $\pi = \ell_0; \ell_1 \ldots \ell_n$. For each $0 \leq i < n$, let $p(i)$ be such that the statement at position i in π is at position $p(i)$ in π' after the sequence of switching transformations taking π to π'.

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

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

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

- If $a \rightarrow b$ is a intra-thread order edge, we have that $\mathcal{C}' \models \Phi$ and $\Phi \implies a \sqsubseteq b$, and hence, no switching transformation would change the order of a and b.

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

4.2.2 Eliminating Bad Traces

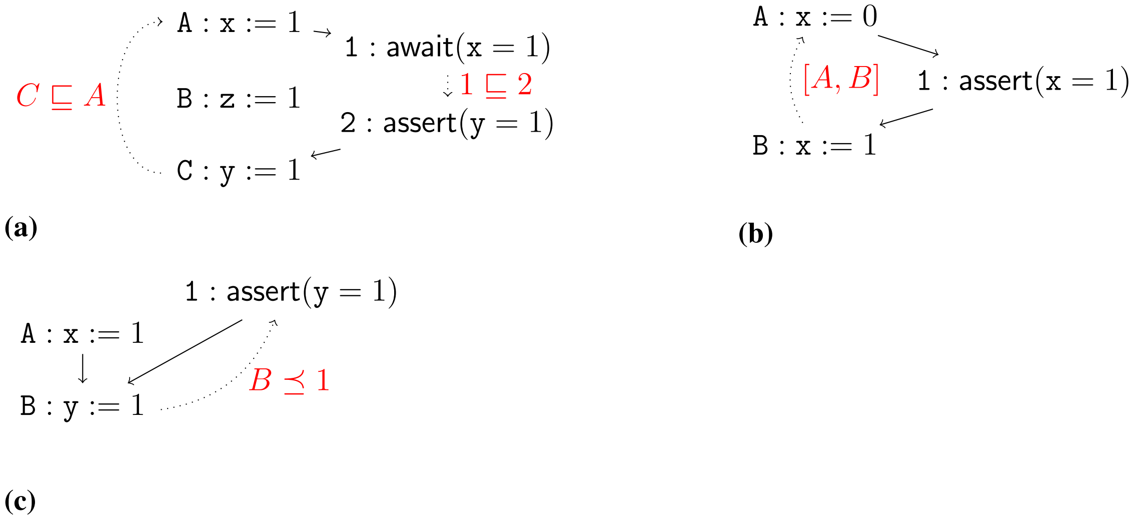
Given a bad trace π of $\mathcal{C}$, a program constraint Φ is a *error eliminating constraint* if for all transformations $\theta_1, \ldots, \theta_k$ and $\mathcal{C}'$ such that $\mathcal{C} \xrightarrow{\theta_1 \ldots \theta_k} \mathcal{C}'$ and $\mathcal{C}' \models \Phi$, each bad trace π' in $\theta_k^f \circ \ldots \circ \theta_1^f(\pi)$ is not a trace of $\mathcal{C}'$. In Chapter 3, we presented an algorithm to fix bad traces using reordering and atomic sections. The main idea behind the algorithm is as follows. Given a bad trace π, we (a) first, generalise the trace into a partial order neighbourhood; and (b) then, compute a program constraint that violates some essential part of the ordering necessary for the bug.

More precisely, the algorithm builds a trace elimination graph which contain edges corresponding to the orderings necessary for the bug to occur, as well as the edges corresponding program constraints. Fixes are found by finding cycles in this graph – the conjunction of the program constraints in a cycle form an error elimination constraint. Intuitively, the program constraints in the cycle will enforce a happens-before conflicting with the orderings necessary for the bug.

Example 4.2.6. *Consider the program in Figure 4.3a and the trace elimination graph for the trace* $\mathtt{A}; \mathtt{B}; \mathtt{1}; \mathtt{2}; \mathtt{C}$. *The orderings* $\mathtt{A}$ *happens-before* $\mathtt{1}$ *and* $\mathtt{2}$ *happens-before* $\mathtt{C}$ *are necessary for the error to happen. The cycle* $\mathtt{C} \rightarrow \mathtt{A} \rightarrow \mathtt{1} \rightarrow \mathtt{2} \rightarrow \mathtt{C}$ *is the elimination cycle. The corresponding error eliminating constraint is* $\mathtt{C} \sqsubseteq \mathtt{A} \wedge \mathtt{1} \sqsubseteq \mathtt{2}$*, and one possible fix is to move* $\mathtt{C}$ *ahead of* $\mathtt{A}$. *For the bad trace* $\mathtt{A}; \mathtt{1}; \mathtt{B}$ *in Figure 4.3b, the elimination cycle is* $\mathtt{A} \rightarrow \mathtt{1} \rightarrow \mathtt{B} \rightarrow \mathtt{A}$ *giving us the constraint* $[\mathtt{A}; \mathtt{B}]$ *and an atomic section around* $\mathtt{A}; \mathtt{B}$ *as the fix.*

The FixBad **algorithm.** The FixBad algorithm takes as input a program $\mathcal{C}$, a constraint Φ and a bad trace π. It outputs a program constraint Φ', sequence of program transformations $\theta_1, \ldots, \theta_k$, and a new program $\mathcal{C}'$, such that $\mathcal{C} \xrightarrow{\theta_1 \ldots \theta_k} \mathcal{C}'$. The algorithm guarantees that (a) Φ' is

Figure 4.3 Eliminating bad traces



an error eliminating constraint; (b) $\mathcal{C}' \models \Phi \wedge \mathcal{C}' \models \Phi'$; and (c) if there is no preemption-free trace π' of $\mathcal{C}$ such that π freely transforms to π' (i.e., $\pi' \in f(\pi)$), then none of the transformations $\theta \in \{\theta_1, \ldots, \theta_k\}$ acts across preemption points. The fact that $\theta_1 \ldots \theta_k$ and $\mathcal{C}'$ can be chosen to satisfy (c) is a consequence of the algorithm from Chapter 3.

Fixes using wait/signal statements. Some programs cannot be fixed by statement reordering or atomic section insertion. These programs are in general outside our definition of the program repair problem as they have bad sequential traces. However, they can be fixed by the insertion of wait/signal statements. One such example is depicted in Figure 4.3c where the trace `1`; `A`; `B` causes an assertion failure. A possible fix is to add a wait statement before `1` and a corresponding signal statement after `B`. The algorithm *FixBad* can be modified to insert such wait-signal statements by also considering constraints of the form $X \preceq Y$ to represent that X is scheduled before Y – the corresponding program transformation is to add a wait statement before Y and a signal statement after X. In Figure 4.3c, the edge $\texttt{B} \rightarrow \texttt{1}$ represents such a constraint $\texttt{B} \preceq \texttt{1}$ – the elimination cycle $\texttt{1} \rightarrow \texttt{B} \rightarrow \texttt{1}$ corresponds to the above described fix.

4.3 The Regression-free Synthesis Procedure

Algorithm 4.2 is a program-repair algorithm to fix concurrency bugs while avoiding regressions. The algorithm maintains the current program $\mathcal{C}$, and a constraint Φ that restricts possible reorderings. In each iteration, the algorithm tests if $\mathcal{C}$ is correct and if so returns $\mathcal{C}$. If not it picks a trace

Algorithm 4.2 Program-Repair Procedure for Concurrency

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

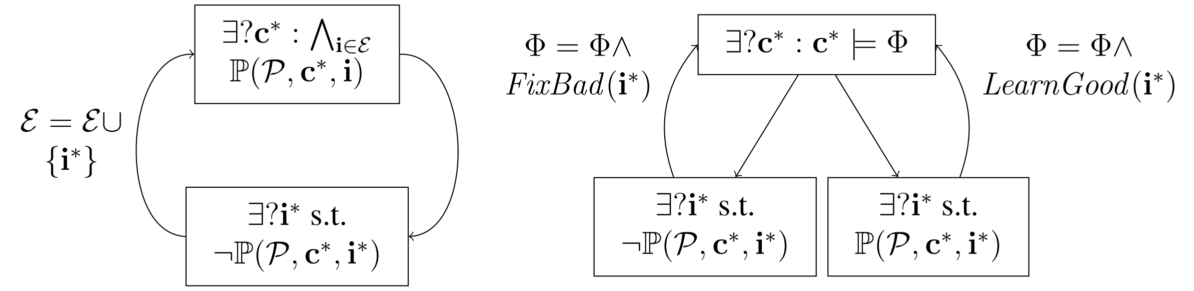
π in $\mathcal{C}$ (line 4). If the trace is good it learns the regression-preventing constraint Φ for π and the trace π is added to the set of good traces T_G (T_G is required only for the correctness proof). If π is bad it calls $FixBad$ to generate a new program that excludes π while respecting Φ, and Φ is strengthened by conjunction with the error elimination constraint Φ' produced by $FixBad$. The algorithm terminates with a valid solution for all choices of $\mathcal{C}'$ in line 10 as the constraint Φ is strengthened in each $FixBad$ iteration. Eventually, the strongest program-constraint will restrict the possible program $\mathcal{C}'$ to one with large enough atomic sections such that it will have only preemption-free or sequential traces.

Theorem 4.3.1 (Soundness)**.** *Given a program $\mathcal{C}$, Algorithm 4.2 returns a program $\mathcal{C}'$ with no bad traces that preserves the sequential and correct preemption-free behaviour of $\mathcal{C}$. Further, each iteration of the* **while** *loop where a bad trace π is chosen performs a regression-free error fix with respect to the good traces T_G.*

The extension of the $FixBad$ algorithm to wait/signal fixes in Algorithm 4.2 may lead to $\mathcal{C}'$ not preserving the good preemption-free and sequential behaviours of $\mathcal{C}$. However, in this case, the input $\mathcal{C}$ violates the pre-conditions of the algorithm.

Theorem 4.3.2 (Fair Termination)**.** *Assuming that a bad trace will eventually be chosen in line 4 if one exists in $\mathcal{C}$, Algorithm 4.2 terminates for any instantiation of $FixBad$.*

Figure 4.4 The CEGIS and PACES spectrum


$\exists ?\mathbf{c}^* : \bigwedge_{\mathbf{i} \in \mathcal{E}} \mathbb{P}(\mathcal{P}, \mathbf{c}^*, \mathbf{i})$
$\mathcal{E} = \mathcal{E} \cup \{\mathbf{i}^*\}$
$\exists ?\mathbf{i}^*$ s.t. $\neg\mathbb{P}(\mathcal{P}, \mathbf{c}^*, \mathbf{i}^*)$
$\Phi = \Phi \wedge$ $\mathit{FixBad}(\mathbf{i}^*)$
$\exists ?\mathbf{c}^* : \mathbf{c}^* \models \Phi$
$\Phi = \Phi \wedge$ $\mathit{LearnGood}(\mathbf{i}^*)$
$\exists ?\mathbf{i}^*$ s.t. $\neg\mathbb{P}(\mathcal{P}, \mathbf{c}^*, \mathbf{i}^*)$
$\exists ?\mathbf{i}^*$ s.t. $\mathbb{P}(\mathcal{P}, \mathbf{c}^*, \mathbf{i}^*)$

4.3.1 A Generic Program-Repair Procedure

We now explain how our program-repair algorithm relates to generic synthesis algorithms based on *counter-example guided inductive synthesis* (CEGIS) [Solar-Lezama *et al.*, 2006]. In the CEGIS approach, the input is a *partial-program* $\mathcal{P}$, i.e., a non-deterministic program and the goal is to specialise $\mathcal{P}$ to a program $\mathcal{C}$ so that all behaviours of $\mathcal{C}$ satisfy a specification. In our case, the partial-program would non-deterministically choose between various reorderings and atomic sections. Let C be the set of choices (e.g., statement orderings) available in $\mathcal{P}$. For a given $\mathbf{c} \in C$, let $\mathbb{P}(\mathcal{P}, \mathbf{c}, \mathbf{i})$ be the predicate that program obtained by specialising $\mathcal{P}$ with $\mathbf{c}$ behaves correctly on the input $\mathbf{i}$.

The CEGIS algorithm maintains a set $\mathcal{E}$ of inputs called experiments. In each iteration, it finds $\mathbf{c}^* \in C$ such that the $\forall \mathbf{i} \in \mathcal{E} : \mathbb{P}(\mathcal{P}, \mathbf{c}^*, \mathbf{i})$. Then, it attempts to find an input $\mathbf{i}^*$ such that $\mathbb{P}(\mathcal{P}, \mathbf{c}^*, \mathbf{i}^*)$ does not hold. If there is no such input, then $\mathbf{c}^*$ is the correct specialisation. Otherwise, $\mathbf{i}^*$ is added to $\mathcal{E}$. This algorithm is illustrated in Figure 4.4(left). Alternatively, CEGIS can be rewritten in terms of constraints on C. For each input $\mathbf{i}$, we associate the constraint $\phi_{\mathbf{i}}$ where $\phi_{\mathbf{i}}(\mathbf{c}) \Leftrightarrow \mathbb{P}(\mathcal{P}, \mathbf{c}, \mathbf{i})$. Now, instead of $\mathcal{E}$, the algorithm maintains the constraint $\Phi = \bigwedge_{\mathbf{i} \in \mathcal{E}} \phi_{\mathbf{i}}$. Every iteration, the algorithm picks a $\mathbf{c}$ such that $\mathbf{c} \models \Phi$; tries to find an input $\mathbf{i}^*$ such that $\neg\mathbb{P}(\mathcal{P}, \mathbf{c}, \mathbf{i})$ holds, and then strengthens Φ by $\phi_{\mathbf{i}^*}$.

This algorithm is exactly the else branch (i.e., FixBad algorithm) of an iteration in Algorithm 4.2 where $\mathbf{i}^*$ and $\phi_{\mathbf{i}^*}$ correspond to π and $\mathit{FixBad}(\pi)$. Intuitively, the initial variable values in π and the scheduler choices are the inputs to our concurrent programs. This suggests that the then branch in Algorithm 4.2 could also be incorporated into the standard CEGIS approach. This extension (dubbed PACES for *Positive and Counter-Examples in Synthesis*) to the CEGIS approach is shown in Figure 4.4(right). Here, the algorithm in each iteration may choose to find an input for which the program is correct and use the constraints arising from it. We discuss the advantages and disadvantages of this approach below.

Constraints vs. Inputs. A major advantage of using constraints instead of sample inputs is the possibility of using over- and under-approximations. As seen in Section 4.2.1, it is sometimes easier to work with approximations of constraints due to simplicity of representation at the cost of potentially missing good solutions. Another advantage is that the sample inputs may have no simple representations in some domains. The scheduler decisions are one such example – the scheduler choices for one program are hard to translate into the scheduler choices for another. For example, the original CEGIS for concurrency work [Solar-Lezama *et al.*, 2008] uses ad-hoc trace projection to translate the scheduler choices between programs.

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

4.4 Implementation and Experiments

We implemented Algorithm 4.2 in our tool ConRepair[2] consisting of 3300 lines of Scala code. The model checker CBMC [Clarke *et al.*, 2004] is used for generating both good and bad traces, and on an average more than 95% of the total execution time is spent in CBMC. Model checking is far from optimal to obtain good traces, and we expect that techniques from [Sen, 2008] can be used to generate good traces much faster. Our tool can operate in two modes: In "mixed" mode it first analyses good traces and then proceeds to fixing the program. The baseline "badOnly" mode skips the analysis of good traces (corresponds to the algorithm in Chapter 3).

In practice the analysis of bad traces usually generates a large number of potential reorderings that could fix the bug. Our previous algorithm from Chapter 3 (badOnly `ce1`) prefers reorderings over atomic sections, but in examples where an atomic section is the only fix, this algorithm has

[2] available as open-source software along with benchmarks: `https://github.com/thorstent/ConRepair`

Table 4.1 Results in iterations and time needed.

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

poor performance. To address this we implemented a heuristic (`ce2`) that places atomic sections before having tried all possible reorderings, but this can result in solutions having unnecessary atomic sections.

The fall back case in Algorithm 4.1 severely limits further fixes – it forces further fixes involving the same statements to be atomic sections. Hence, in our implementation, we omit this step and prefer an unsound algorithm (i.e., not necessarily regression-free) that can fix more programs with reorderings. While the implemented algorithm is unsound, our experiments show that even without the fallback, in our examples, there is no regression except for one artificial example (`ex-regr.c`) constructed precisely for that purpose.

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

Table 4.1 shows our experimental results: the iterations and the wall-clock time needed to find a valid fix for our mixed algorithm and the two heuristics of the badOnly algorithm. For the mixed algorithm the time is split into the time needed to generate and analyse good traces (first number) and the time needed for the fixing afterwards. All measurements were done on an Intel core i5-3320M laptop with 8GB of RAM.

Detailed analysis. The artificial examples `ex1.c` to `ex5.c` are used for testing and take only a few seconds; example `paper1.c` is the one in Figure 4.1a. Example `ex-regr.c` was constructed to show unsoundness of the implementation. Example `usb-serial.c` models the USB-to-serial adapter driver. Here, from the good traces the tool learns that two statements should not be reordered as it will trigger another bug. This prompts them to be reordered above a third statement together, while the badOnly analysis would first move one, find a new bug, and then fix that by moving the other statement. Thus, the good trace analysis saves us two rounds of bug fixing and reduces bug fixing time by 18 minutes.

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

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

Chapter 5

Synthesis of Locks and Other Synchronisation Primitives

5.1 Problem Statement and Illustrative Examples

In Chapters 3 and 4 we introduced a technique to repair concurrent programs using statement reordering and atomic sections. However, statement reordering only fixes a certain type of bugs and atomic sections are not directly implementable. In this chapter we introduce a technique that is able to place locks, wait-signal and barriers in the code. These are common synchronisation primitives supported in many programming languages.

In the two previous chapters we generalised a concurrent trace into a set of traces that *under-approximates* the target trace sets. In this chapter, we present a succinct, *complete* representation of such concurrent trace sets, which can drive diverse verification, fault localisation, repair, and synthesis techniques for concurrent programs. The representation is complete in the sense that it encodes every trace in the trace set of interest.

Neighbourhood computation. In Chapters 3 and 4 we used partial-order neighbourhoods, which are not able to represent exactly all bad interleavings of a trace. In this chapter we move to neighbourhoods that allow disjunctions in the HB-formula. Given a trace π and a correctness specification, we present a method to generate an HB-formula φ_B representing the bad neighbourhood of π (see Section 2.2). To generate φ_B, we first encode all the bad executions in $\mathcal{L}(\mathcal{N}_\pi)$ in a quantifier-free first-order formula Φ such that an execution π is a model of Φ iff π

is a bad execution in $\mathcal{L}(\mathcal{N}_\pi^b)$. We then *incrementally* construct φ_B. Initially, φ_B is set to `false`. In each step: (1) we invoke an SMT solver to obtain a model for Φ that does not belong to the language of the subset of $\mathcal{N}_\pi^b$ represented by the current φ_B, (2) *generalise* the trace of the model into an HB-formula φ, and (3) update φ_B by adding φ as a disjunct. We iterate until there is no new model of Φ. The trace generalisation used in each iteration has the following properties: (a) the model obtained in the iteration satisfies φ, and (b) any trace in $\mathcal{N}_\pi$ that satisfies φ is bad. The final HB-formula obtained is an *exact* representation of $\mathcal{N}_\pi^b$.

While an exact representation is a worthy goal, the corresponding φ_B may not be succinct. To gain succinctness and utility, we trade in exactness. In particular, we permit the inclusion of infeasible traces to obtain a succinct HB-formula representing a *sound overapproximation* of $\mathcal{N}_\pi^b$. The overapproximation of $\mathcal{N}_\pi^b$ is sound in the sense that it is guaranteed to not include any good traces. To generate such a succinct HB-formula, we enhance the above algorithm. We use data-flow analysis and minimal unsatisfiability core (unsat core) computation for generalising the trace of the model into an HB-formula φ in step (2) of each iteration. This new trace generalisation step has the following properties: (a) the model obtained in the iteration satisfies φ, and (b) any trace in $\mathcal{N}_\pi$ satisfying φ is either bad or infeasible.

Complementing φ_B, the succinct representation of a sound overapproximation of $\mathcal{N}_\pi^b$ yields φ_G, a succinct representation of a sound overapproximation of $\mathcal{N}_\pi^g$. Note that complementing the exact representation of $\mathcal{N}_\pi^b$ does not yield an exact representation of $\mathcal{N}_\pi^g$. In fact, our existing methodology cannot produce an exact representation of $\mathcal{N}_\pi^g$. Figure 5.1 shows the exact representation of $\mathcal{N}_\pi^b$ and the representations for sound overapproximations of $\mathcal{N}_\pi^g$ and $\mathcal{N}_\pi^b$ obtained by our method for the example trace shown.

We implemented the above algorithm as a tool TARA and used it to generate (succinct) representations of trace sets of programs drawn from the software verification competition (SV-Comp) [Beyer, 2014] and the regression suites of ESBMC [Morse *et al.*, 2014] and CONREPAIR (Chapter 4).

Synchronisation synthesis. We present a novel algorithm that uses φ_G to synthesise synchronisation for eliminating the bad neighbourhood of π. The algorithm proceeds by applying rewrite rules to derive synchronisation primitives such as mutex locks, barriers, shared exclusive locks and wait-signal statements from easily-identifiable patterns in φ_G. For example, a missing mutex lock in the example in Figure 5.1 that ensures the statements at `T`$_{\texttt{w}}$`[1]`

Figure 5.1 Online banking: This trace is drawn from a program consisting of three threads, one for withdrawing money, one for depositing money, and one for checking consistency of the bank account after completion of a withdrawal and a deposit. (In all the examples in this chapter, we represent traces using typed global variable declarations/initialisations, followed by each thread's typed local variable declarations and statements. Note that this representation depicts a trace and not a program.)

$\texttt{init} : \texttt{x} := \texttt{balance};\ \ \texttt{deposited} := 0;\ \ \texttt{withdrawn} := 0$

thread_withdraw

$\mathsf{T}_{\texttt{W}}[1] : \texttt{temp} := \texttt{balance};$

$\mathsf{T}_{\texttt{W}}[2] : \texttt{balance} := \texttt{temp} - \texttt{withdrawal};$

$\mathsf{T}_{\texttt{W}}[3] : \texttt{withdrawn} := 1$

thread_deposit

$\mathsf{T}_{\texttt{D}}[1] : \texttt{temp2} := \texttt{balance};$

$\mathsf{T}_{\texttt{D}}[2] : \texttt{balance} := \texttt{temp2} + \texttt{deposit};$

$\mathsf{T}_{\texttt{D}}[3] : \texttt{deposited} := 1$

thread_checkresult

$\mathsf{T}_{\texttt{C}}[1] : \mathsf{assume}(\texttt{deposited} = 1 \land \texttt{withdrawn} = 1);$

$\mathsf{T}_{\texttt{C}}[2] : \mathsf{assert}(\texttt{balance} = \texttt{x} + \texttt{deposit} - \texttt{withdrawal})$

Exact representation of $\mathcal{N}_\pi^b$:

$hb(\mathsf{T}_{\texttt{W}}[1], \mathsf{T}_{\texttt{D}}[2]) \land hb(\mathsf{T}_{\texttt{D}}[1], \mathsf{T}_{\texttt{W}}[2]) \land hb(\mathsf{T}_{\texttt{W}}[3], \mathsf{T}_{\texttt{C}}[1]) \land hb(\mathsf{T}_{\texttt{D}}[3], \mathsf{T}_{\texttt{C}}[1])$

Exact representation of $\mathcal{N}_\pi^g$:

$(hb(\mathsf{T}_{\texttt{D}}[2], \mathsf{T}_{\texttt{W}}[1]) \lor hb(\mathsf{T}_{\texttt{W}}[2], \mathsf{T}_{\texttt{D}}[1])) \land hb(\mathsf{T}_{\texttt{W}}[3], \mathsf{T}_{\texttt{C}}[1]) \land hb(\mathsf{T}_{\texttt{D}}[3], \mathsf{T}_{\texttt{C}}[1])$

Representation of sound overapproximation of $\mathcal{N}_\pi^b$:

$hb(\mathsf{T}_{\texttt{W}}[1], \mathsf{T}_{\texttt{D}}[2]) \land hb(\mathsf{T}_{\texttt{D}}[1], \mathsf{T}_{\texttt{W}}[2])$

Representation of sound overapproximation of $\mathcal{N}_\pi^g$:

$hb(\mathsf{T}_{\texttt{D}}[2], \mathsf{T}_{\texttt{W}}[1]) \lor hb(\mathsf{T}_{\texttt{W}}[2], \mathsf{T}_{\texttt{D}}[1])$

and $\mathtt{T_W}[2]$ in `thread_withdraw` do not interfere with the statements $\mathtt{T_D}[1]$ and $\mathtt{T_D}[2]$ in `thread_deposit` is identified by the pattern $hb(\mathtt{T_D}[2], \mathtt{T_W}[1]) \vee hb(\mathtt{T_W}[2], \mathtt{T_D}[1])$ in φ_G. We have implemented this algorithm as an extension of our tool TARA and used it to successfully synthesise synchronisation for our benchmarks.

We further demonstrate the applicability of our representations of good and bad neighbourhoods of a trace to bug summarisation and verification based on counterexample-guided abstraction refinement (CEGAR).

Bug summarisation. Error detection tools based on model checking and static analyses typically provide counterexample traces to help with program debugging. However, these traces can be long and encumbered with unnecessary data, providing little insight about the actual bug. We use φ_B, the representation for a sound overapproximation of a trace's bad neighbourhood, for counterexample and bug summarisation. The HB-formula φ_B encodes relevant *ordering* information about all counterexamples in the neighbourhood of π and can be viewed as a stand-alone counterexample summary. While this can already be useful feedback for a human debugger, we present a set of rules to infer specific bugs such as data races, atomicity violations, two-stage access bugs and define-use order violations. These rules work by identifying particular patterns in φ_B and combining them with some lightweight data-flow information. We have extended TARA for bug summarisation and evaluated it on our benchmarks.

Accelerating CEGAR. We also recognise an application of our representation of bad neighbourhoods of abstract counterexamples in accelerating CEGAR for concurrent programs. CEGAR often takes many iterations to find the right predicates for proving correctness of a program.

There is a number of prior work to enhance the CEGAR loop by finding better predicates, e.g. [Beyer *et al.*, 2007; Sharma *et al.*, 2012]. In the setting of hardware model-checking (for circuits), Glusman et al. [Glusman *et al.*, 2003] extend the CEGAR loop by adding several predicates if a spurious counterexample is found; they generate all counterexamples of the same length and gather information about valuations crucial to the incorrectness of the counterexamples. In a similar setting, Wang et al. [Wang *et al.*, 2006] improve the CEGAR by introducing a technique to eliminate all spurious counterexamples for an invariant. Sakunkonchak et al. [Sakunkonchak *et al.*, 2007] apply CEGAR optimisations to software model checking and speed up the search for predicates that make the counterexample spurious. However, they do not use interpolants and instead search the counterexample for conflicting predicates. Bjesse et al. [Bjesse and Kukula, 2004] use predicates obtained from CEGAR to guide bounded-model checking (BMC) and extend its reach.

The choice of refinement procedure usually determines the number of iterations necessary. Many heuristics have been proposed to find relevant predicates quickly, e.g., [Beyer *et al.*, 2007]. This problem is compounded in concurrent program verification, where the existence of a large number of interleavings can delay the discovery of *interesting* spurious counterexamples that lead to relevant predicates. We present a new predicate learning procedure that uses the HB-formula φ_B representing the bad neighbourhood of a spurious counterexample of an abstract concurrent program. In each iteration of the CEGAR loop, our procedure refines the abstraction to eliminate multiple spurious abstract counterexamples drawn from φ_B, using a method similar to *beautiful interpolants* [Albarghouthi and McMillan, 2013]. We have integrated our TARA-based refinement procedure within SATABS [Clarke *et al.*, 2005] and have been able to reduce the number of iterations needed to verify various example programs.

Highlights. We introduce a novel representation for concurrent trace sets based on HB-formulæ (Section 5.2). HB-formulæ have several useful properties. They can express arbitrary finite trace sets. They enable efficient computation and concise expression of unions over trace sets. This is exploited by our tool TARA to compute succinct representations of sound overapproximations of good and bad neighbourhoods of a trace. HB-formulæ are an intuitively appealing representation for trace sets. They can reveal specific patterns of causality relations between events that can drive diverse verification, fault localisation, repair, and synthesis techniques for concurrent programs. We demonstrate the use of our tool in three applications — synchronisation synthesis (Section 5.3), bug summarisation (Section 5.4), and CEGAR acceleration (Section 5.5).

5.2 Computing Good and Bad Neighbourhoods

In this section, we present an algorithm for computing an exact representation for the bad neighbourhood of a trace. However, as this representation may be unwieldy and complex, we further provide an algorithm to produce sound overapproximations of $\mathcal{N}_\pi^b$ and $\mathcal{N}_\pi^g$, i.e., to find succinct HB-formulæ φ_G and φ_B such that $\mathcal{N}_\pi^g \subseteq [\![\varphi_G]\!]$, $\mathcal{N}_\pi^b \subseteq [\![\varphi_B]\!]$, and $[\![\varphi_G]\!] \cap \mathcal{N}_\pi^b =$

$[\![\varphi_B]\!] \cap \mathcal{N}_\pi^g = \emptyset$. Note that $\varphi_G \wedge \varphi_B$ is not necessarily `false`, because $\varphi_G \wedge \varphi_B$ may still represent infeasible traces.

We use two different formalisms to express statements.

- *Guarded actions.* Here, a statement from thread T_i is either a guarded action $\mathsf{assume}(G) \rightarrow$ assign or an assertion $\mathsf{assert}(G)$, where G is a Boolean expression over V_i and assign is a parallel assignment $var_1, \ldots, var_{\mathtt{m}} := expr_1, \ldots, expr_m$ of expressions over V_i to variables in V_i. We use just $:=$ to indicate no assignment is happening (i.e. the statement equals $\mathsf{assume}(G)$).
- *Transition predicates.* Here, a statement from thread T_i is a predicate over variables from $V_i \cup V_i'$ where V_i' contains primed versions of variables in V_i. Intuitively, variables from V_i and V_i' represent the values of program variables before and after the execution of the statement, respectively. For example, the assignment $\mathtt{x} := \mathtt{x} + \mathtt{y}$ is represented as $x' = x + y$. The advantage of this formalism is that it can express non-deterministic statements which we need to model abstract programs in Section 5.5. Assertions are represented as before, i.e., as $\mathsf{assert}(G)$, where G is a Boolean expression over V_i.

The transition predicate formalism is used exclusively for CEGAR accelaration in Section 5.5. For traces of programs we use the guarded actions formalism.

Translating traces to guarded actions. In Figure 5.2 we present a translation function that translates statements in traces into guarded actions. We do not consider havoc, input and output statements as they are used primarily for the implicit specification. To indicate what branch the trace took for loops and conditionals we use the words then, else, loop and exitloop. Statements inside an atomic section are merged into one translated statement.

Encoding bad executions. Given a trace π, our algorithm is based on constructing a quantifier free first-order formula that represents all bad executions in $\mathcal{L}(\mathcal{N}_\pi)$. We use the concurrent trace program encoding [Wang *et al.*, 2009] which is based on a concurrent single static assignment (CSSA) form of traces. We recall the encoding below to make the presentation self-contained. We present the encoding for the case where statements are expressed as guarded actions; the case where statements are expressed as transition predicates is similar. Given a trace π, we first rewrite it into the CSSA form.

Figure 5.2 Translation of statement with identifier ℓ to guarded action

$\mathit{ShVar} := \mathit{LoExp}$	$=$	$\mathtt{true} \rightarrow \mathit{ShVar} := \mathit{LoExp}$
$\mathit{LoVar} := \mathit{ShExp}$	$=$	$\mathtt{true} \rightarrow \mathit{LoVar} := \mathit{ShExp}$
$\mathsf{if}\ (\mathit{ShExp})\ \mathsf{then}$	$=$	$\mathit{ShExp} \rightarrow :=$
$\mathsf{if}\ (\mathit{ShExp})\ \mathsf{else}$	$=$	$\neg\mathit{ShExp} \rightarrow :=$
$\mathsf{while}\ \mathit{ShExp}\ \mathsf{loop}$	$=$	$\mathit{ShExp} \rightarrow :=$
$\mathsf{while}\ \mathit{ShExp}\ \mathsf{exitloop}$	$=$	$\neg\mathit{ShExp} \rightarrow :=$
$\mathsf{assert}(\mathit{ShExp})$	$=$	$\mathsf{assert}(\mathit{ShExp})$
$\mathsf{await}(\mathit{ShExp})$	$=$	$\mathit{ShExp} \rightarrow :=$
$\mathsf{lock}(\mathit{LkVar})$	$=$	$\mathit{LkVar} = 0 \rightarrow \mathit{LkVar} := \mathsf{tid}(\ell)$
$\mathsf{unlock}(\mathit{LkVar})$	$=$	$\mathit{LkVar} = \mathsf{tid}(\ell) \rightarrow \mathit{LkVar} := 0$
$\mathsf{wait}(\mathit{CondVar})$	$=$	$\mathit{CondVar} = 1 \rightarrow :=$
$\mathsf{wait\_not}(\mathit{CondVar})$	$=$	$\mathit{CondVar} = 0 \rightarrow :=$
$\mathsf{signal}(\mathit{CondVar})$	$=$	$\mathtt{true} \rightarrow \mathit{CondVar} := 1$
$\mathsf{reset}(\mathit{CondVar})$	$=$	$\mathtt{true} \rightarrow \mathit{CondVar} := 0$
$\mathsf{wait\_reset}(\mathit{CondVar})$	$=$	$\mathit{CondVar} = 1 \rightarrow \mathit{CondVar} := 0$
$\mathsf{assume}(\mathit{GrdVar})$	$=$	$\mathit{GrdVar} = \mathtt{true} \rightarrow :=$
$\mathsf{assume\_not}(\mathit{GrdVar})$	$=$	$\mathit{GrdVar} = \mathtt{false} \rightarrow :=$
$\mathit{GrdVar} \leftarrow \mathit{GrdExpr}$	$=$	$\mathtt{true} \rightarrow\ \mathit{GrdVar} := \mathit{GrdExpr}$

- For each variable v, we introduce a unique name $v_{w,\ell}$ for each event ℓ that may change the value of v (here, w stands for "write"). Further, for each variable v, we introduce a unique name v_ι to represent the value of v at the start of an execution.
- For each event ℓ that reads a variable v, we replace v as follows:
  - If v is a local variable, we replace v by $v_{w,\ell'}$ where ℓ' is the most recent event from the thread that writes to v; and
  - If v is a shared variable, we replace v by $v_{r,\ell}$ (where r stands for "read") and we store an additional constraint, where $v_{r,\ell} = \pi(v_\iota, v_{w,\ell_1}, v_{w,\ell_2},, \ldots, v_{w,\ell_\ell})$ where ℓ_i ranges over all events from other threads that write to v and the most recent event from the same thread that writes to v.

  The π-functions above are analogous to the ϕ-functions used to express joins in sequential single static assignment encodings, i.e., $v_{r,\ell} = \pi(v_\iota, v_{w,\ell_1}, \ldots, v_{w,\ell_\ell})$ expresses that ℓ reads either the initial value of v, or the value written by one of $\ell_1, \ldots, \ell_\ell$.
- Further, for each event ℓ, we define the condition that ℓ is feasibly reached. If ℓ is the first event in a thread, we set $\mathit{cond}(\ell) = \mathtt{true}$. Otherwise, $\mathit{cond}(\ell)$ depends on the previous event from the same thread in π (say ℓ'). If ℓ' is an assertion, we let $\mathit{cond}(\ell) = \mathit{cond}(\ell')$. Otherwise, ℓ' is a guarded action $\mathsf{assume}(G) \rightarrow \mathsf{assign}$, and we let $\mathit{cond}(\ell) = \mathit{cond}(\ell') \wedge G$.

Example 5.2.1. *In the running example from Figure 5.1, the statement* $\mathsf{T}_{\mathtt{W}}[1] : \mathtt{temp} := \mathtt{balance}$

would be encoded as $temp_{w,\mathtt{T_W[1]}} = balance_{r,\mathtt{T_W[1]}} \wedge balance_{r,\mathtt{T_W[1]}} = \pi(balance_\iota, balance_{w,\mathtt{T_D[2]}})$.

Given a trace π rewritten in the CSSA form, the following constraints encode executions in the neighbourhood $\mathcal{N}_\pi$ of π:

- *Thread orders.* In any execution in the neighbourhood of π, the order of events in each thread is the same as in the trace π. We define $\Phi_{PO} = \bigwedge\{hb(\ell_i, \ell_j) \mid \mathsf{tid}(\ell_i) = \mathsf{tid}(\ell_j) \wedge \ell_i <_\pi \ell_j\}$.
- *Variable assignments.* This part of the encoding is a direct translation of the assignments in each event into constraints. We have $\Phi_{VD} = \bigwedge_\ell \bigwedge_{i=1}^m v^i_{w,\ell} = expr^i$, where ℓ ranges over events of the form $\mathsf{T}[\ell]$: stmt with stmt being $\mathsf{assume}(G) \to v^1_{\mathtt{w},\ell}, \ldots, v^m_{\mathtt{w},\ell} := expr^1, \ldots, expr^m$.
- *π-constraints.* Each π-constraint chooses a value for a read of a shared variable from possible writes. Formally, each condition $v_{r,\ell} = \pi(v_\iota, v_{w,\ell_1}, \ldots, v_{w,\ell_\ell})$ is rewritten as $[v_{r,\ell} = v_\iota \wedge \bigwedge_i hb(\ell, \ell_i)] \vee \bigvee_{i=1}^{\ell}[v_{r,\ell} = v_{w,\ell_i} \wedge cond(\ell_i) \wedge hb(\ell_i, \ell) \wedge \bigwedge_{j \neq i}(hb(\ell_j, \ell_i) \vee hb(\ell, \ell_j))]$. Intuitively, the above formula states that: (a) the value of v read by ℓ is either the initial value of v (denoted as v_ι) or written by one of $\ell_1, \ldots, \ell_\ell$; (b) if the value is the initial value, all ℓ_i happen after ℓ; and (c) if the value is written by ℓ_i, then ℓ_i is feasibly reached and all conflicting writes either happen before ℓ_i or after ℓ. We denote by Φ_{PI} the conjunction of all such π-constraints. For example, for the π-function from Example 5.2.1, the corresponding constraint is $(balance_{r,\mathtt{T_W[1]}} = balance_\iota \wedge hb(\mathsf{T_W}[1], \mathsf{T_D}[2])) \vee (balance_{r,\mathtt{T_W[1]}} = balance_{w,\mathtt{T_D[2]}} \wedge hb(\mathsf{T_D}[2], \mathsf{T_W}[1]))$.
- *Correctness condition.* For correctness, if an assertion event $\ell = \mathsf{T}[\ell]$: $\mathtt{assert}(G_\ell)$ is feasibly reached, then G_ℓ must hold. Hence, the correctness condition is $\Phi_{COR} = \bigwedge_\ell(cond(\ell) \Rightarrow G_\ell)$ where ℓ ranges over assertion events.

The final encoding for bad executions is given by $\Phi_{CTP}(\pi) = \Phi_{PO} \wedge \Phi_{VD} \wedge \Phi_{PI} \wedge \neg\Phi_{COR}$. We also encode the complementary correctness condition as $\Phi_{\overline{CTP}}(\pi) = \Phi_{PO} \wedge \Phi_{VD} \wedge \Phi_{PI} \wedge \Phi_{COR}$.

For convenience, we use an auxiliary formula Φ_{FEA} to represent the condition that each assumption must hold. We have $\Phi_{FEA} = \bigwedge_\ell cond(\ell)$ where ℓ ranges over all events.

An execution π *corresponds* to a model $\mathcal{V}$ of Φ_{CTP} if: (a) the value of each v_ι in $\mathcal{V}$ is the initial value of v in π; (b) the value of each $v_{r,\ell}$ in $\mathcal{V}$ is the value of v read by ℓ in π; (c) the value of each $v_{w,\ell}$ in $\mathcal{V}$ is the value of v written by ℓ in π; and (d) the value of $hb(\ell_i, \ell_j)$ in $\mathcal{V}$ is true if and only if ℓ_i occurs before ℓ_j in π.

Theorem 5.2.2. *Given a trace π, (a) for every model $\mathcal{V}$ of $\Phi_{CTP}(\pi)$ there is a bad execution $\pi \in \mathcal{L}(\mathcal{N}^b_\pi)$ such that π corresponds to $\mathcal{V}$; and (b) for every $\pi \in \mathcal{L}(\mathcal{N}^b_\pi)$ there is a model $\mathcal{V}$ of $\Phi_{CTP}(\pi)$ such that π corresponds to $\mathcal{V}$.*

Bad neighbourhood computation. Armed with Φ_{CTP} — an SMT encoding of bad executions in the neighbourhood of a trace π — we now present an algorithm to compute a representation of $\mathcal{N}^b_\pi$. Algorithm 5.1 proceeds by repeatedly computing satisfying assignments to Φ_{CTP} using an SMT solver (lines 2 and 3), and accumulating the HB-formulæ in the models (lines 4 and 5). We conjoin Φ_{CTP} with additional constraints to ensure that the same satisfying assignments are not returned each time.

Algorithm 5.1 Computing the bad neighbourhood of a trace

```
Require: Trace $\pi$
Ensure: HB-formula $\varphi_B$ such that $\mathcal{N}^b_\pi = [\![\varphi_B]\!]$.
1: $\Phi \leftarrow \Phi_{CTP}(\pi)$; $\varphi_B \leftarrow$ false
2: while $\Phi \wedge \neg\varphi_B$ is satisfiable do
3:     $\mathcal{V} \leftarrow$ satisfying assignment for $\Phi \wedge \neg\varphi_B$
4:     $\varphi'_B \leftarrow \bigwedge\{hb(\ell, \ell') \mid \mathcal{V} \models hb(\ell, \ell')\}$
5:     $\varphi_B \leftarrow \varphi_B \vee \varphi'_B$
6: end while
7: return $\varphi_B$
```

Overapproximating bad neighbourhoods. While Algorithm 5.1 computes an exact representation of $\mathcal{N}^b_\pi$, it is inefficient in practice. Hence, we forgo the goal of an exact representation. Instead, we compute a *sound overapproximation* of $\mathcal{N}^b_\pi$, which may include infeasible traces, but not good traces. Given trace π, Algorithm 5.2 computes sound overapproximations of $\mathcal{N}^b_\pi$ and $\mathcal{N}^g_\pi$. Algorithm 5.2 performs several optimisations with respect to Algorithm 5.1 to accumulate weaker constraints from each model of Φ_{CTP}, i.e., Algorithm 5.2 attempts to accumulate larger subsets of $\mathcal{N}_\pi$ into φ_B in each iteration.

- **Data-flow analysis.** From the model $\mathcal{V}$ of $\Phi_{CTP}(\pi)$, the data-flow analysis retains those happens-before constraints (φ'_B) that are necessary to preserve the data-flow into the failing assertion in the corresponding execution. We use the function $DF_{\mathcal{V}}(\ell)$ (line 5) to compute constraints that ensure ℓ can be feasibly reached and can read the same variable values as in $\mathcal{V}$. Given the execution corresponding to $\mathcal{V}$, let $reads(\ell)$, $readsG(\ell)$, and $srcEvent(v, \ell)$ represent the variables read by ℓ, the variables read by ℓ in the guard (if ℓ is not a guarded assignment, $readsG(\ell) = \emptyset$), and the event that writes the value of v read by ℓ. We have $DF_{\mathcal{V}}(\ell) = DF^1_{\mathcal{V}}(\ell) \cup DF^2_{\mathcal{V}}(\ell)$ where:

- we let $DF^1_{\mathcal{V}}(\ell) = \bigcup_{v \in reads(\ell)} \big[\{(v, srcEvent(v, \ell), \ell)\} \cup DF_{\mathcal{V}}(srcEvent(v, \ell))\big]$; and
- $DF^2_{\mathcal{V}}(\ell) = \bigcup_{\ell' \in E, v \in readsG(\ell')} \big[\{(v, srcEvent(v, \ell'), \ell')\} \cup DF_{\mathcal{V}}(srcEvent(v, \ell'))\big]$ where event ℓ' ranges over $E = \{\ell' \mid \mathsf{tid}(\ell) = \mathsf{tid}(\ell') \wedge \mathcal{V} \models hb(\ell', \ell)\}$.

Intuitively, $DF^1_{\mathcal{V}}$ ensures that ℓ can read the same values as in $\mathcal{V}$ and $DF^2_{\mathcal{V}}$ ensures that ℓ is feasibly reached. We then get additional constraints ADF necessary to ensure conflicting writes do not affect the data-flow into the assertion (line 6).

- **Unsatisfiable core computation.** Next, we perform two rounds of generalisation on $\varphi_{B'}$ through unsatisfiable core computation. In the first round, we construct a formula $\varphi_{B'} \wedge Choices(\mathcal{V}) \wedge \Phi_{\overline{CTP}}(\pi)$ where $Choices(\mathcal{V})$ fixes the initial variable values to the ones from $\mathcal{V}$ (line 10). A satisfying assignment to this formula models executions where no failing assertion is feasibly reached. Therefore, if the formula is unsatisfiable, the happens-before constraints from the unsatisfiable core (line 12) ensure that all executions satisfying $Choices(\mathcal{V})$ are bad. Note that if all statements are deterministic, the above formula is always unsatisfiable. In the second round (line 10), we follow a similar procedure, but with the formula $\varphi_{B'} \wedge \Phi_{FEA} \wedge \Phi_{\overline{CTP}}$. Here, a model is a good execution and hence, the constraints from the unsatisfiable core (line 10) ensure that any feasible execution is necessarily bad.

  Roughly, the first round allows us to generalise the HB-formula in the case of data-dependent bugs. The second round lets us generalise further in the case of data-independent bugs.

The sound overapproximation, φ_G, of $\mathcal{N}^g_\pi$ is obtained by complementing φ_B (line 22). Note that φ_B returned is in disjunctive normal form (DNF), while φ_G is in conjunctive normal form (CNF).

Theorem 5.2.3. *For a trace π, if Algorithm 5.2 returns (φ_B, φ_G), then $\mathcal{N}^b_\pi \subseteq [\![\varphi_B]\!]$, $\mathcal{N}^g_\pi \subseteq [\![\varphi_G]\!]$, and $[\![\varphi_G]\!] \cap \mathcal{N}^b_\pi = [\![\varphi_B]\!] \cap \mathcal{N}^g_\pi = \emptyset$.*

The complexity of Algorithm 5.2 originates from SMT check in line 2, which is NP-complete. A comparison with Algorithm 3.1 is not meaningful, because Algorithm 5.2 basically solves the verification problem for a straight-line program.

Algorithm 5.2 Computing sound overapproximations of the bad and good neighbourhoods of a trace

```
Require: Trace $\pi$
Ensure: HB-formulæ $(\varphi_B, \varphi_G)$ such that $\mathcal{N}^g_\pi \subseteq [\![\varphi_G]\!]$, $\mathcal{N}^b_\pi \subseteq [\![\varphi_B]\!]$, and $[\![\varphi_G]\!] \cap [\![\varphi_B]\!] = \emptyset$.
$\Phi \leftarrow \Phi_{CTP}(\pi)$; $\varphi_B \leftarrow$ false
while $\Phi \wedge \neg\varphi_B$ is satisfiable do
    $\mathcal{V} \leftarrow$ satisfying assignment for $\Phi \wedge \neg\varphi_B$
                                                    ▷ Data-flow analysis
    $DF \leftarrow DF_{\mathcal{V}}(\ell^*)$ where $\ell^*$ is the failing assertion in $\mathcal{V}$
    $ADF \leftarrow \bigcup_{(v,\ell_i,\ell_j)\in DF} \bigcup_{\{\ell_k \mid \ell_k \text{ writes } v\}} \big(\{(v,\ell_k,\ell_i) \mid$
        $\mathcal{V} \models hb(\ell_k,\ell_i)\} \cup \{(v,\ell_j,\ell_k) \mid \mathcal{V} \models hb(\ell_j,\ell_k)\}\big)$
    $\varphi_{B'} \leftarrow \bigwedge_{(v,\ell_i,\ell_j)\in DF \cup ADF} hb(\ell_i,\ell_j)$
                                                    ▷ Unsat-core computation
    $Choices(\mathcal{V}) \leftarrow \bigwedge_{v \in V} v_\iota = \mathcal{V}[v_\iota]$
    if $\varphi_{B'} \wedge Choices(\mathcal{V}) \wedge \Phi_{\overline{CTP}}(\pi))$ is unsatisfiable then
        $\varphi_{B'} \leftarrow MinUNSATCore(Soft \leftarrow \varphi_{B'},$
                $Hard \leftarrow Choices(\mathcal{V}) \wedge \Phi_{\overline{CTP}}(\pi))$
    end if
    if $\varphi_{B'} \wedge \Phi_{FEA}(\pi) \wedge \Phi_{\overline{CTP}}(\pi))$ is unsatisfiable then
        $\varphi_{B'} \leftarrow MinUNSATCore(Soft \leftarrow \varphi_{B'},$
                $Hard \leftarrow \Phi_{FEA}(\pi) \wedge \Phi_{\overline{CTP}}(\pi))$
    end if
    $\varphi_B \leftarrow \varphi_B \vee \varphi_{B'}$
end while
$\varphi_G \leftarrow \neg\varphi_B$
return $(\varphi_B, \varphi_G)$
```

5.3 Synchronisation Synthesis

We use the representation of a sound overapproximation of the good neighbourhood of a trace π (returned as φ_G by Algorithm 5.2) to synthesise synchronisation that eliminates the bad neighbourhood of π. Missing synchronisation primitives such as locks, barriers, and wait-signal statements present themselves as easily identifiable HB-formula *patterns* in φ_G. Our algorithm derives the required synchronisation using rules that rewrite such patterns into the corresponding primitives.

Synchronisation primitives. We first describe various synchronisation primitives that we derive. Recall from Section 2.2.1 that we use the notation $\mathtt{T}[\ell]$ to refer to events labelled with $\mathtt{T}[\ell]$, and the notations $\mathtt{T}[\ell : \ell']$ and $\mathtt{T}[\mathtt{L}]$ to refer to corresponding event sequences.

1. *Wait-Signal.* A wait-signal $\mathsf{WaitSignal}\big(\mathtt{T}_2[\ell_2], \mathtt{T}_1[\ell_1]\big)$ denotes a wait to make $\mathtt{T}_2[\ell_2]$ wait for $\mathtt{T}_1[\ell_1]$ to complete, and a signal to make $\mathtt{T}_1[\ell_1]$ signal $\mathtt{T}_2[\ell_2]$ upon completion.
2. *Locks.* A lock $\mathsf{lock}(\mathtt{T}_1[\mathtt{L}_1], \ldots, \mathtt{T}_\mathtt{n}[\mathtt{L}_\mathtt{n}])$ denotes a common lock protecting each event sequence $\mathtt{T}_\mathtt{i}[\mathtt{L}_\mathtt{i}]$, $\mathtt{i} \in [1, \mathtt{n}]$, to ensure that these event sequences cannot execute concurrently.
3. *Barriers.* A barrier $\mathsf{Barrier}\big(\mathtt{T}_1[\ell_1], \ldots, \mathtt{T}_\mathtt{n}[\ell_\mathtt{n}]\big)$ at location $\ell_\mathtt{i}$ of thread $\mathtt{T}_\mathtt{i}$, $\mathtt{i} \in [1, \mathtt{n}]$, prevents each thread $\mathtt{T}_\mathtt{i}$ from proceeding beyond $\ell_\mathtt{i}$ until every other thread $\mathtt{T}_\mathtt{j}$ reaches $\ell_\mathtt{j}$. In other words, $\mathtt{T}_\mathtt{i}$ cannot execute the event at $\ell_\mathtt{i}$ until every other $\mathtt{T}_\mathtt{j}$ executes the event at $\ell_\mathtt{j} - 1$.
4. *Shared-exclusive locks.* A shared-exclusive lock (or, a readers-writers lock) $\mathsf{ShExLock}\big(\mathtt{Sh} : \mathtt{T}_{\mathtt{s1}}[\mathtt{L}_{\mathtt{s1}}], \ldots, \mathtt{T}_{\mathtt{sn}}[\mathtt{L}_{\mathtt{sn}}], \mathtt{Ex} : \mathtt{T}_{\mathtt{x1}}[\mathtt{L}_{\mathtt{x1}}], \ldots, \mathtt{T}_{\mathtt{xm}}[\mathtt{L}_{\mathtt{xm}}]\big)$ permits concurrent execution of all event sequences $\mathtt{T}_{\mathtt{si}}[\mathtt{L}_{\mathtt{si}}]$, $\mathtt{i} \in [1, \mathtt{n}]$, while preventing concurrent execution of (a) any two $\mathtt{T}_{\mathtt{xi}}[\mathtt{L}_{\mathtt{xi}}]$ and $\mathtt{T}_{\mathtt{xj}}[\mathtt{L}_{\mathtt{xj}}]$ with $\mathtt{i} \neq \mathtt{j}$, and (b) any $\mathtt{T}_{\mathtt{xi}}[\mathtt{L}_{\mathtt{xi}}]$ and $\mathtt{T}_{\mathtt{sj}}[\mathtt{L}_{\mathtt{sj}}]$.

Rewriting φ_G to derive synchronisation. During the rewrite process below, we use disjunctive formulæ (denoted by ψ) where each disjunct is either an atomic *hb*-constraint of the form $hb(\mathtt{T}_\mathtt{i}[\ell_\mathtt{i}], \mathtt{T}_\mathtt{j}[\ell_\mathtt{j}])$, or a synchronisation primitive. For a trace π, we repeatedly apply the rewrite rules from Figure 5.3 on φ_G (in CNF, as returned from Algorithm 5.2) until no more rules are applicable. The ADD.WAITSIGNAL, ADD.LOCK and ADD.BARRIER rules *introduce* the wait-signal, locks, and barrier primitives. The MERGE.LOCKS rule *merges* locks across pairs of threads, while the MERGE.LOCKS.DEADLOCKS rule merges locks that can potentially lead to deadlocks. The MULTITHREAD.LOCK and MULTITHREAD.BARRIER rules inductively derive

Figure 5.3 Rewrite rules for synchronisation synthesis

$$\frac{hb(\mathsf{T}_1[\ell_1'],\mathsf{T}_2[\ell_2]) \vee hb(\mathsf{T}_2[\ell_2'],\mathsf{T}_1[\ell_1]) \vee \psi \quad \ell_1 \leq \ell_1' \quad \ell_2 \leq \ell_2'}{\mathsf{lock}(\mathsf{T}_1[\ell_1 : \ell_1'],\mathsf{T}_2[\ell_2 : \ell_2']) \vee \psi}\ \textsc{Add.Lock}$$

$$\frac{hb(\mathsf{T}_1[\ell_1],\mathsf{T}_2[\ell_2]) \vee \psi}{\mathsf{WaitSignal}\big(\mathsf{T}_2[\ell_2],\mathsf{T}_1[\ell_1]\big) \vee \psi}\ \textsc{Add.WaitSignal}$$

$$\frac{\big(hb(\mathsf{T}_1[\ell_1-1],\mathsf{T}_2[\ell_2]) \vee \psi\big) \wedge \big(hb(\mathsf{T}_2[\ell_2-1],\mathsf{T}_1[\ell_1]) \vee \psi\big)}{\mathsf{Barrier}\big(\mathsf{T}_1[\ell_1],\mathsf{T}_2[\ell_2]\big) \vee \psi}\quad \textsc{Add.Barrier}$$

$$\frac{\big(\mathsf{lock}(\mathsf{T}_1[\mathtt{L}_1],..,\mathsf{T}_\mathtt{n}[\mathtt{L}_\mathtt{n}]) \vee \psi\big) \wedge \bigwedge_{i=1}^{n} \mathsf{lock}(\mathsf{T}_\mathtt{i}[\mathtt{L}_\mathtt{i}],\mathsf{T}_{\mathtt{n+1}}[\mathtt{L}_{\mathtt{n+1}}]) \vee \psi}{\mathsf{lock}(\mathsf{T}_1[\mathtt{L}_1],..,\mathsf{T}_\mathtt{n}[\mathtt{L}_{\mathtt{n+1}}]) \vee \psi}\quad \textsc{Multithread.Lock}$$

$$\frac{\big(\mathsf{lock}(\mathsf{T}_1[\mathtt{L}_1],\mathsf{T}_2[\mathtt{L}_2]) \vee \psi\big) \wedge \big(\mathsf{lock}(\mathsf{T}_1[\mathtt{L}_1{}'],\mathsf{T}_2[\mathtt{L}_2{}']) \vee \psi\big) \qquad \mathsf{T}_1[\mathtt{L}_1{}'] \subseteq \mathsf{T}_1[\mathtt{L}_1] \qquad \mathsf{T}_2[\mathtt{L}_2{}'] \subseteq \mathsf{T}_2[\mathtt{L}_2]}{\mathsf{lock}(\mathsf{T}_1[\mathtt{L}_1],\mathsf{T}_2[\mathtt{L}_2]) \vee \psi}\ \textsc{Merge.Locks}$$

$$\frac{\begin{array}{c}\ell_1^{\mathsf{a}} \leq \ell_1^{\mathsf{b}} \leq \ell_1^{\mathsf{a}'} \qquad \ell_2^{\mathsf{b}} \leq \ell_2^{\mathsf{a}} \leq \ell_2^{\mathsf{b}'}\\ \big(\mathsf{lock}(\mathsf{T}_1[\ell_1^{\mathsf{a}},\ell_1^{\mathsf{a}'}],\mathsf{T}_2[\ell_2^{\mathsf{a}},\ell_2^{\mathsf{a}'}]) \vee \psi\big) \wedge \big(\mathsf{lock}(\mathsf{T}_1[\ell_1^{\mathsf{b}},\ell_1^{\mathsf{b}'}],\mathsf{T}_2[\ell_2^{\mathsf{b}},\ell_2^{\mathsf{b}'}]) \vee \psi\big)\end{array}}{\mathsf{lock}(\mathsf{T}_1[\ell_1^{\mathsf{a}},\max(\ell_1^{\mathsf{a}'},\ell_1^{\mathsf{b}'})]],\mathsf{T}_2[\ell_2^{\mathsf{b}},\max(\ell_2^{\mathsf{b}'},\ell_2^{\mathsf{a}'})]) \vee \psi}\ \textsc{Merge.Locks.Deadlocks}$$

$$\frac{\big(\mathsf{Barrier}\big(\mathsf{T}_1[\ell_1],\ldots,\mathsf{T}_\mathtt{n}[\ell_\mathtt{n}]\big) \vee \psi\big) \wedge \bigwedge_{i=1}^{n}\big(\mathsf{Barrier}\big(\mathsf{T}_\mathtt{i}[\ell_\mathtt{i}],\mathsf{T}_{\mathtt{n+1}}[\ell_{\mathtt{n+1}}]\big) \vee \psi\big)}{\mathsf{Barrier}\big(\mathsf{T}_1[\ell_1],\ldots,\mathsf{T}_{\mathtt{n+1}}[\ell_{\mathtt{n+1}}]\big) \vee \psi}\quad \textsc{Multithread.Barrier}$$

$$\frac{\bigwedge_{i=1}^{n}\bigwedge_{j=1}^{m}\big(\mathsf{lock}(\mathsf{T}_{\mathtt{s}_\mathtt{i}}[\mathtt{L}_{\mathtt{s}_\mathtt{i}}],\mathsf{T}_{\mathtt{x}_\mathtt{j}}[\mathtt{L}_{\mathtt{x}_\mathtt{j}}]) \vee \psi\big) \quad \bigwedge_{i=1}^{m}\bigwedge_{j=1}^{m}\big(\mathsf{lock}(\mathsf{T}_{\mathtt{x}_\mathtt{i}}[\mathtt{L}_{\mathtt{x}_\mathtt{i}}],\mathsf{T}_{\mathtt{x}_\mathtt{j}}[\mathtt{L}_{\mathtt{x}_\mathtt{j}}]) \vee \psi\big)}{\mathsf{ShExLock}\big(\mathtt{Sh}:\mathsf{T}_{\mathtt{s}_1}[\mathtt{L}_{\mathtt{s}_1}],\ldots,\mathsf{T}_{\mathtt{s}_\mathtt{n}}[\mathtt{L}_{\mathtt{s}_\mathtt{n}}],\mathtt{Ex}:\mathsf{T}_{\mathtt{x}_1}[\mathtt{L}_{\mathtt{x}_2}],\ldots,\mathsf{T}_{\mathtt{x}_\mathtt{m}}[\mathtt{L}_{\mathtt{x}_\mathtt{m}}]\big) \vee \psi}\ \textsc{Add.SharedExclusiveLock}$$

locks and barriers spanning multiple threads. The ADD.SHAREDEXCLUSIVELOCK rule derives a shared exclusive lock from already inferred locks. Since φ_G, as generated by Algorithm 5.2, is already optimised, we do not merge `WaitSignal` primitives.

We explain two of the above rules here. The premise of the ADD.LOCK rule asks for two event sequences $\mathsf{T}_1[\ell_1 : \ell_1']$ and $\mathsf{T}_2[\ell_2 : \ell_2']$ such that one of them has to finish execution before the other starts, i.e., $hb(\mathsf{T}_1[\ell_1'],\mathsf{T}_2[\ell_2]) \vee hb(\mathsf{T}_2[\ell_2'],\mathsf{T}_1[\ell_1])$. Equivalently, the two event sequences do not execute concurrently. This is enforced by the lock $\mathsf{lock}(\mathsf{T}_1[\ell_1 : \ell_1'],\mathsf{T}_2[\ell_2 : \ell_2'])$. The premise of the MERGE.LOCKS.DEADLOCKS rule looks for two already derived locks, acquired by two threads in different orders (which may lead to a deadlock), and merges these locks into one.

Note that the rewriting process always terminates. However, depending on the order of rules applied, we may obtain different formulæ. Upon termination, we get a CNF formula over synchronisation primitives. We pick a set $\mathcal{S}$ of synchronisation primitives, consisting of one primitive from each conjunct. Let $\mathcal{C}^{\mathcal{S}}$ be the program obtained by inserting each synchronisation primitive in $\mathcal{S}$ into the corresponding position in the original concurrent program $\mathcal{C}$.

Theorem 5.3.1 (Soundness of rewrite rules)**.** *Given a trace π, let $\mathcal{C}^{\mathcal{S}}$ be obtained as described above. Let $\pi \in \mathcal{L}(\mathcal{N}_\pi)$ be a deadlock-free execution of $\mathcal{C}^{\mathcal{S}}$. Then $\pi \notin \mathcal{L}(\mathcal{N}_\pi^b)$, i.e., π is not bad.*

While $\mathcal{C}^{\mathcal{S}}$ is not guaranteed deadlock-free, we perform simple consistency checks when choosing $\mathcal{S}$ to prevent obvious deadlocks. For example, we ensure that `WaitSignal` primitives in $\mathcal{S}$ do not introduce *ordering cycles* over $\mathsf{locs}(\pi)$.

Note that our rewrite rules are by no means complete. It may be possible to derive the above synchronisation primitives using different rules that represent other scenarios. Further, our rewrite system can also be extended to other synchronisation primitives. We now present examples illustrating the application of our rules.

Example 5.3.2. *For the example trace shown in Figure 5.1, φ_G is given by $hb(\mathsf{T}_{\mathtt{D}}[2], \mathsf{T}_{\mathtt{W}}[1]) \vee hb(\mathsf{T}_{\mathtt{W}}[2], \mathsf{T}_{\mathtt{D}}[1])$. Applying the* ADD.LOCK *rewrite rule yields* $\mathsf{lock}(\mathsf{T}_{\mathtt{W}}[1:2], \mathsf{T}_{\mathtt{D}}[1:2])$.

Example 5.3.3. *For the example trace shown in Figure 5.4(a), φ_G is given by $(hb(\mathsf{T}_{\mathtt{F}}[4], \mathsf{T}_{\mathtt{S}}[3]) \vee hb(\mathsf{T}_{\mathtt{S}}[4], \mathsf{T}_{\mathtt{F}}[3])) \wedge hb(\mathsf{T}_{\mathtt{S}}[4], \mathsf{T}_{\mathtt{F}}[5]) \wedge hb(\mathsf{T}_{\mathtt{F}}[4], \mathsf{T}_{\mathtt{S}}[5])$. Applying the* ADD.LOCK *rewrite rule yields:* $\mathsf{lock}(\mathsf{T}_{\mathtt{F}}[3:4], \mathsf{T}_{\mathtt{S}}[3:4]) \wedge hb(\mathsf{T}_{\mathtt{S}}[4], \mathsf{T}_{\mathtt{F}}[5]) \wedge hb(\mathsf{T}_{\mathtt{F}}[4], \mathsf{T}_{\mathtt{S}}[5])$. *Applying the* ADD.BARRIER *rule yields:* $\mathsf{lock}(\mathsf{T}_{\mathtt{F}}[3:4], \mathsf{T}_{\mathtt{S}}[3:4]) \wedge \mathsf{Barrier}\big(\mathsf{T}_{\mathtt{F}}[5], \mathsf{T}_{\mathtt{S}}[5]\big)$.

Example 5.3.4. *For the example trace shown in Figure 5.4(b), φ_G is as shown. The disjuncts ψ_1 and ψ_2 are not relevant for this example except for the fact that ψ_1 is common to the 3^{rd} and 4^{th} conjuncts, ψ_2 is common to the 5^{th} and 6^{th} conjuncts and $\psi_1 \neq \psi_2$.*

- *Applying* ADD.LOCK *yields:* $hb(\mathsf{T}_{\mathtt{I}}[2], \mathsf{T}_{\mathtt{F}}[2]) \wedge hb(\mathsf{T}_{\mathtt{I}}[2], \mathsf{T}_{\mathtt{S}}[2]) \wedge (\mathsf{lock}(\mathsf{T}_{\mathtt{F}}[4], \mathsf{T}_{\mathtt{S}}[3:4]) \vee \psi_1) \wedge (\mathsf{lock}(\mathsf{T}_{\mathtt{F}}[3:4], \mathsf{T}_{\mathtt{S}}[4]) \vee \psi_1) \wedge (\mathsf{lock}(\mathsf{T}_{\mathtt{F}}[4], \mathsf{T}_{\mathtt{S}}[3:4]) \vee \psi_2) \wedge (\mathsf{lock}(\mathsf{T}_{\mathtt{F}}[3:4], \mathsf{T}_{\mathtt{S}}[4]) \vee \psi_2)$.
- *Applying* MERGE.LOCKS *next yields:* $hb(\mathsf{T}_{\mathtt{I}}[2], \mathsf{T}_{\mathtt{F}}[2]) \wedge hb(\mathsf{T}_{\mathtt{I}}[2], \mathsf{T}_{\mathtt{S}}[2]) \wedge (\mathsf{lock}(\mathsf{T}_{\mathtt{F}}[3:4], \mathsf{T}_{\mathtt{S}}[3:4]) \vee \psi_1) \wedge (\mathsf{lock}(\mathsf{T}_{\mathtt{F}}[3:4], \mathsf{T}_{\mathtt{S}}[3:4]) \vee \psi_2)$.
- *Finally, applying the* ADD.WAITSIGNAL *rule yields:* $\mathsf{WaitSignal}\big(\mathsf{T}_{\mathtt{F}}[2], \mathsf{T}_{\mathtt{I}}[2]\big) \wedge \mathsf{WaitSignal}\big(\mathsf{T}_{\mathtt{S}}[2], \mathsf{T}_{\mathtt{I}}[2]\big) \wedge (\mathsf{lock}(\mathsf{T}_{\mathtt{F}}[3:4], \mathsf{T}_{\mathtt{S}}[3:4]) \vee \psi_1) \wedge (\mathsf{lock}(\mathsf{T}_{\mathtt{F}}[3:4], \mathsf{T}_{\mathtt{S}}[3:4]) \vee \psi_2)$.

Note that the MERGE.LOCKS *rule does not apply to the last two conjuncts as $\psi_1 \neq \psi_2$. One possible solution for $\mathcal{S}$ is* $\{\mathsf{WaitSignal}\big(\mathsf{T}_{\mathtt{F}}[2], \mathsf{T}_{\mathtt{I}}[2]\big), \mathsf{WaitSignal}\big(\mathsf{T}_{\mathtt{S}}[2], \mathsf{T}_{\mathtt{I}}[2]\big), \mathsf{lock}(\mathsf{T}_{\mathtt{F}}[3:4], \mathsf{T}_{\mathtt{S}}[3:4])\}$.

Figure 5.4 Example programs

```
init:  value1 := 1;  value2 := 2;
value3 := 4;  value4 := 8;  sum := 0;
flag1 := 0;  flag2 := 0

thread_firsthalf
$T_F[1]$ : localsum1 := value1;
$T_F[2]$ : localsum1 := localsum + value2;
$T_F[3]$ : temp1 := sum;
$T_F[4]$ : sum := temp1 + localsum1;
$T_F[5]$ : value1 := value1/sum;
$T_F[6]$ : value2 := value2/sum;
$T_F[7]$ : flag1 := 1

thread_secondhalf
$T_S[1]$ : localsum2:=value3;
$T_S[2]$ : localsum2:=localsum2 + value4;
$T_S[3]$ : temp2:=sum;
$T_S[4]$ : sum:=temp2 + localsum2;
$T_S[5]$ : value3:=value3/sum;
$T_S[6]$ : value4:=value4/sum;
$T_S[7]$ : flag2:=1

thread_checkresult
$T_C[1]$ : assume(flag1 = 1 $\wedge$ flag2 = 1);
$T_C[2]$ : assert(value1 + value2 + value3 + value4 = 1)

$\varphi_G$: $(hb(T_F[4], T_S[3]) \vee hb(T_S[4], T_F[3])) \wedge hb(T_S[4], T_F[5]) \wedge hb(T_F[4], T_S[5])$
```

(a) Normalisation. The goal of the program this trace is drawn from is to normalise a set of values such that their sum computes to 1. The program consists of three threads. The first and second thread process one half each of the set of values. Once the first and second thread run to completion, the third thread checks if the sum of the normalised values is 1.

```
init:  intrmask:=0;  initdone:=0;
workqueueitems:=0;  interrupts:=0

thread_interruptmaskset
$T_I[1]$ : intrmask:=1;
$T_I[2]$ : initdone:=1

thread_first_irqhandler
locals:  int temp;
$T_F[1]$ : assume(intrmask = 1);
$T_F[2]$ : assert(initdone = 1);
$T_F[3]$ : temp1:=workqueueitems;
$T_F[4]$ : workqueueitems:=temp1 + 1;
$T_F[5]$ : interrupts:=interrupts + 1

thread_second_irqhandler
$T_S[1]$ : assume(intrmask = 1);
$T_S[2]$ : assert(initdone = 1);
$T_S[3]$ : temp2:=workqueueitems;
$T_S[4]$ : workqueueitem:=temp2 + 1;
$T_S[5]$ : interrupts:=interrupts + 1

thread_checkworkqueue
$T_C[1]$ : assert(workqueueitems $\geq$ interrupts);

$\varphi_G$: $hb(T_I[2], T_F[2]) \wedge hb(T_I[2], T_S[2]) \wedge$
    $(hb(T_S[4], T_F[4]) \vee hb(T_F[4], T_S[3]) \vee \psi_1) \wedge$
$(hb(T_F[4], T_S[4]) \vee hb(T_S[4], T_F[3]) \vee \psi_1) \wedge$
    $(hb(T_S[4], T_F[4]) \vee hb(T_F[4], T_S[3]) \vee \psi_2) \wedge$
$(hb(T_F[4], T_S[4]) \vee hb(T_S[4], T_F[3]) \vee \psi_2)$
```

(b) Interrupt handler (simplified snippet of the Linux RealTek 8169 network driver). Once the `intrmask` variable is set by the `interruptmaskset` thread, the hardware starts producing interrupts. The handling of these interrupts, by the two `irqhandler` threads, is correct only if the driver initialisation is complete (captured by the `initdone` variable). The `irqhandlers` add items to a workqueue; the addition of items is modelled using a counter `workqueueitems`. The variable `interrupts` counts the total number of interrupts handled by the `irqhandler` threads and the thread `checkworkqueue` uses `interrupts` to check for inconsistencies in the workqueue.

Figure 5.4 Example programs

```
init:  registered:=0

pci_thread
T_P[1] : registered:=1;
T_P[2] : hw_start:=&drv_hw_start;

network_thread
T_N[1] : assume(registered ≠ 0);
T_N[2] : assert(*hw_start = drv_hw_start)
        /* pointer dereference */

void drv_hw_start() {
/* does something */
}
```

φ_B: $hb(\mathsf{T}_\mathtt{N}[2], \mathsf{T}_\mathtt{P}[2])$

(c) Network device initialiser. This trace is drawn from a simplified snippet of the Linux RealTek 8169 network driver. The `pci` thread signals that a network device is registered using the variable `registered` and sets `hw_start` to point to the `drv_hw_start` method. The `network` thread calls `drv_open` once the network device is registered. The `drv_open` method dereferences the `hw_start` pointer.

```
globals:  int[] pagetable, memory;
init:  pagetable[1] = 5, memory[5] =
10;

thread_pagetableaccess:
locals:  int loc, data, page;
T_P[1]:  page := 1;
T_P[2]:  loc := pagetable[page];
T_P[3]:  data := memory[loc];
T_P[4]:  assert (data = 10);

thread_datamove:
locals:  int page, newloc, loc;
T_D[1]:  page, newloc := 1, 20;
T_D[2]:  loc := pagetable[page];
T_D[3]:  pagetable[page] := newloc;
T_D[4]:  memory[newloc] :=
memory[loc];
```

φ_B: $hb(\mathsf{T}_\mathtt{D}[3], \mathsf{T}_\mathtt{P}[2]) \wedge hb(\mathsf{T}_\mathtt{P}[3], \mathsf{T}_\mathtt{D}[4])$

(d) Page-table. The `pagetableaccess` thread reads a memory location `loc` from `pagetable` and reads `data` from that memory location. The `datamove` thread reads the current memory location `loc` from `pagetable`, updates `pagetable` with a new memory location `newloc` and copies the data from the old memory location to the new memory location.

5.4 Bug Summarisation

Next we show that the representation for a sound overapproximation of the bad neighbourhood of a trace π (returned as φ_B by Algorithm 5.2) is useful for counterexample summarisation and bug summarisation. The HB-formula φ_B encapsulates relevant ordering information about all counterexamples in the neighbourhood of π and can be viewed as a stand-alone counterexample summary. For instance, in Figure 5.4(c), one may view $\varphi_B = hb(\mathsf{T}_\mathtt{N}[2], \mathsf{T}_\mathtt{P}[2])$ as a counterexample summary that indicates a possible order violation. While such a bug report can already be useful to a human debugger, a cursory examination of the data-flow through the events in φ_B can enable formulation of a more precise bug summary. To this end, we present a set of rules to help infer specific bugs such as data races, define-use order violations and two-stage access bugs.

5.4.1 Inferring Bug Summaries from φ_B

We assume φ_B is in DNF. Our inference rules are presented in Figure 5.5. For a thread T, a location ℓ, and a global program variable v, (a) $read(\mathsf{T}[\ell], v)$ denotes that event $\mathsf{T}[\ell]$ reads from v, (b) $write(\mathsf{T}[\ell], v)$ denotes that event $\mathsf{T}[\ell]$ writes to v, and (c) $access(\mathsf{T}[\ell], v)$ denotes that

Figure 5.5 Inference rules for bug summarisation (In this figure, $\ell_1 < \ell'_1 < \ell''_1$ and $\ell_2 < \ell'_2 < \ell''_2$.)

$$\frac{\begin{array}{c} hb(\mathtt{T}_1[\ell'_1], \mathtt{T}_2[\ell_2]) \wedge hb(\mathtt{T}_2[\ell_2], \mathtt{T}_1[\ell''_1]) \wedge \psi \\ read(\mathtt{T}_1[\ell'_1], v) \quad write(\mathtt{T}_1[\ell''_1], v) \quad access(\mathtt{T}_2[\ell_2], v) \end{array}}{\mathtt{DataRace}(\{\mathtt{T}_1[\ell'_1], \mathtt{T}_1[\ell''_1]\}, \mathtt{T}_2[\ell_2])}\ \textsc{DataRace.1}$$

$$\frac{\begin{array}{c} hb(\mathtt{T}_1[\ell'_1], \mathtt{T}_2[\ell''_2]) \wedge hb(\mathtt{T}_2[\ell'_2], \mathtt{T}_1[\ell''_1]) \wedge \psi \\ read(\mathtt{T}_1[\ell'_1], v) \quad write(\mathtt{T}_1[\ell''_1], v) \quad read(\mathtt{T}_2[\ell'_2], v) \quad write(\mathtt{T}_2[\ell''_2], v) \end{array}}{\mathtt{DataRace}(\{\mathtt{T}_1[\ell'_1], \mathtt{T}_1[\ell''_1]\}, \{\mathtt{T}_2[\ell'_2], \mathtt{T}_2[\ell''_2]\})}\ \textsc{DataRace.2}$$

$$\frac{\begin{array}{c} hb(\mathtt{T}_1[\ell_1], \mathtt{T}_2[\ell_2]) \wedge hb(\mathtt{T}_2[\ell_2], \mathtt{T}_1[\ell'_1]) \wedge \psi \\ access(\mathtt{T}_1[\ell_1], v) \quad access(\mathtt{T}_2[\ell_2], v) \quad access(\mathtt{T}_1[\ell'_1], v) \end{array}}{\mathtt{AtomicityViolation}(\mathtt{T}_1[\ell_1 : \ell'_1], \mathtt{T}_2[\ell_2])}\ \textsc{AtomicityViolation.1}$$

$$\frac{\begin{array}{c} hb(\mathtt{T}_1[\ell_1], \mathtt{T}_2[\ell'_2]) \wedge hb(\mathtt{T}_2[\ell_2], \mathtt{T}_1[\ell'_1]) \wedge \psi \\ access(\mathtt{T}_1[\ell_1], v) \quad access(\mathtt{T}_2[\ell_2], v) \quad access(\mathtt{T}_1[\ell'_1], v) \quad access(\mathtt{T}_2[\ell'_2], v) \end{array}}{\mathtt{AtomicityViolation}(\mathtt{T}_1[\ell_1 : \ell'_1], \mathtt{T}_2[\ell_2 : \ell'_2])}\ \textsc{AtomicityViolation.2}$$

$$\frac{\begin{array}{c} hb(\mathtt{T}_1[\ell_1], \mathtt{T}_2[\ell_2]) \wedge hb(\mathtt{T}_2[\ell'_2], \mathtt{T}_1[\ell'_1]) \wedge \psi \\ write(\mathtt{T}_1[\ell_1], v) \quad write(\mathtt{T}_1[\ell'_1], w) \quad read(\mathtt{T}_2[\ell_2], v) \quad read(\mathtt{T}_2[\ell'_2], w) \end{array}}{\mathtt{TwoStageAccessBug}(\mathtt{T}_1[\ell_1 : \ell'_1], \mathtt{T}_2[\ell_2 : \ell'_2])}\ \textsc{TwoStageAccessBug.1}$$

$$\frac{\begin{array}{c} hb(\mathtt{T}_1[\ell_1], \mathtt{T}_2[\ell_2]) \wedge hb(\mathtt{T}_2[\ell'_2], \mathtt{T}_1[\ell'_1]) \wedge \psi \\ read(\mathtt{T}_1[\ell_1], v) \quad read(\mathtt{T}_1[\ell'_1], w) \quad write(\mathtt{T}_2[\ell_2], v) \quad write(\mathtt{T}_2[\ell'_2], w) \end{array}}{\mathtt{TwoStageAccessBug}(\mathtt{T}_1[\ell_1 : \ell'_1], \mathtt{T}_2[\ell_2 : \ell'_2])}\ \textsc{TwoStageAccessBug.2}$$

$$\frac{\begin{array}{c} hb(\mathtt{T}_1[\ell_1], \mathtt{T}_2[\ell_2]) \wedge \psi \quad read(\mathtt{T}_1[\ell_1], v) \quad write(\mathtt{T}_2[\ell_2], v) \\ \exists \sigma \in \mathcal{N}_\pi : \sigma \models hb(\mathtt{T}_1[\ell_1], \mathtt{T}_2[\ell_2]) \wedge \psi \wedge \bigwedge_{\mathtt{T}[\ell]} write(\mathtt{T}[\ell], v) \wedge \mathtt{T}[\ell] \neq \mathtt{T}_1[\ell_1] \Rightarrow hb(\mathtt{T}_1[\ell_1], \mathtt{T}[\ell]) \end{array}}{\mathtt{DefineUse}(\mathtt{T}_1[\ell_1], \mathtt{T}_2[\ell_2])}\ \textsc{DefineUse}$$

event $\mathtt{T}[\ell]$ reads from or writes to v. In the discussion below, we combine these with ordering constraints in a natural manner. For example, $read(\mathtt{T}_1[\ell_1], v) \rightarrow write(\mathtt{T}_2[\ell_2], v)$ says that event $\mathtt{T}_1[\ell_1]$ happens before $\mathtt{T}_2[\ell_2]$ and that $read(\mathtt{T}_1[\ell_1], v)$ and $write(\mathtt{T}_2[\ell_2], v)$ hold.

Data races. Take two events $\ell'_1; \ell''_1$, where event ℓ'_1 has statement $\mathtt{r}{:=}\mathtt{v} + 1$, event ℓ''_1 has statement $\mathtt{v}{:=}\mathtt{r}$, and r is a local variable modelling a register. In this case, a data race between events $\ell'_1; \ell''_1$ and some other event ℓ_2 writing to v in another thread manifests itself in a trace σ as the ordering pattern $\ell'_1 <_\sigma \ell_2 <_\sigma \ell''_1$. Hence, the DATARACE.1 rule infers a possible data race between events labelled $\mathtt{T}_1[\mathtt{l}'_1]$,$\mathtt{T}_1[\mathtt{l}''_1]$, and $\mathtt{T}_2[\mathtt{l}_2]$ if the pattern $read(\mathtt{T}_1[\mathtt{l}'_1], v) \rightarrow access(\mathtt{T}_2[\mathtt{l}_2], v) \rightarrow write(\mathtt{T}_1[\mathtt{l}''_1], v)$ is found in φ_B.

Further, if instead of ℓ_2 we have events $\ell'_2; \ell''_2$, where ℓ'_2 reads from v and ℓ''_2 writes to v, a data race can manifest in a trace σ as $\ell'_1 <_\sigma \ell''_2 \wedge \ell'_2 <_\sigma \ell''_1$. The DATARACE.2 rule infers a possible data race between $\mathtt{T}_1[\ell'_1]$, $\mathtt{T}_1[\ell''_1]$ and $\mathtt{T}_2[\ell'_2]$, $\mathtt{T}_2[\ell''_2]$, if the patterns $read(\mathtt{T}_1[\ell'_1], v) \rightarrow write(\mathtt{T}_2[\ell''_2], v)$ and $read(\mathtt{T}_2[\ell'_2], v) \rightarrow write(\mathtt{T}_1[\ell''_1], v)$ is found in the same disjunct of φ_B.

Atomicity violations. The ATOMICITYVIOLATION rules generalise the DATARACE rules. If the data-flow and ordering pattern $access(\mathsf{T}_1[\ell_1], v) \rightarrow access(\mathsf{T}_2[\ell_2], v) \rightarrow access(\mathsf{T}_1[\ell_1'], v)$ manifests in φ_B, the first rule infers a possible atomicity violation of the event sequence $\mathsf{T}_1[\ell_1 : \ell_1']$ via event $\mathsf{T}_2[\ell_2]$. If the patterns $access(\mathsf{T}_1[\ell_1], v) \rightarrow access(\mathsf{T}_2[\ell_2'], v)$ and $access(\mathsf{T}_2[\ell_2], v) \rightarrow access(\mathsf{T}_1[\ell_1'], v)$ manifest in the same disjunct of φ_B, the second rule infers a possible atomicity violation of the event sequence $\mathsf{T}_1[\ell_1 : \ell_1']$ and event sequence $\mathsf{T}_2[\ell_2 : \ell_2']$.

Two stage access. The TWOSTAGEACCESSBUG rules capture two classic scenarios of two-stage access bugs, indicating violations of some *consistency* requirement of accesses to v and w. In particular, the values of v and w read by a thread could be inconsistent if either of the following patterns manifest in φ_B: (a) $write(\mathsf{T}_1[\ell_1], v) \rightarrow read(\mathsf{T}_2[\ell_2], v) \rightarrow read(\mathsf{T}_2[\ell_2'], w) \rightarrow write(\mathsf{T}_1[\ell_1'], w)$; or (b) $read(\mathsf{T}_1[\ell_1], v) \rightarrow write(\mathsf{T}_2[\ell_2], v) \rightarrow write(\mathsf{T}_2[\ell_2'], w) \rightarrow read(\mathsf{T}_1[\ell_1'], w)$.

Define-use ordering. The DEFINEUSE rule infers a specific type of order violation indicating the use of a variable before it is defined. Given φ_B in DNF, if the ordering $read(\mathsf{T}_1[\ell_1], v) \rightarrow write(\mathsf{T}_2[\ell_2], v)$ manifests in a disjunct δ, the rule infers a define-use order violation if there exists a trace $\sigma \in \mathcal{N}_\pi$ such that σ satisfies δ and $\mathsf{T}_1[\ell_1]$ precedes all events that write to v in σ.

Starting from φ_B given in DNF, we repeatedly apply the inference rules from Figure 5.5 until no more rules are applicable. We report all inferred bugs as possible violations. Note that our goal here is only to assist the user in program debugging. Our inference rules are not complete. We do not claim that our inferred bugs will manifest in the program's executions, or that they will match a human debugger's intuition. We now present examples illustrating the application of some of our bug inference rules.

Example 5.4.1. *For the example trace shown in Figure 5.1,* φ_B *is given by* $hb(\mathtt{T_W}[1], \mathtt{T_D}[2]) \wedge hb(\mathtt{T_D}[1], \mathtt{T_W}[2])$. *Since* $read(\mathtt{T_W}[1], balance)$, $write(\mathtt{T_W}[2], balance)$, $read(\mathtt{T_D}[1], balance)$ *and* $write(\mathtt{T_D}[2], balance)$ *hold, we can apply the* DATARACE.2 *rule to infer a* $\mathtt{DataRace}(\mathtt{W}[1:2], \mathtt{Y}[1:2])$. *Note that this bug inference matches the synchronisation* $\mathsf{lock}(\mathtt{T_W}[1:2], \mathtt{T_D}[1:2])$ *synthesised in Example 5.3.2.*

Example 5.4.2. *Consider the example trace shown in Figure 5.4(c). In our encoding, the pointer* hw_start *is modelled as an integer variable* hw *that is initially* 0 *(since* hw_start *is uninitialised). The pointer dereference in* $\mathtt{T_N}[2]$ *is modelled as* $\mathsf{assert}(\mathtt{hw} > 0)$. *For this example,* φ_B *is given by* $hb(\mathtt{T_N}[2], \mathtt{T_P}[2])$. *Since* $read(\mathtt{T_N}[2], hw)$ *and* $write(\mathtt{T_P}[2], hw)$ *hold, and trace*

$\mathtt{T_P}[1], \mathtt{T_N}[1], \mathtt{T_N}[2], \mathtt{T_P}[2]$ *satisfies the last premise of the* DEFINEUSE *rule, we can apply the rule to infer a define-use order violation between* $\mathtt{T_N}[2]$ *and* $\mathtt{T_P}[2]$.

Example 5.4.3. *For the example trace shown in Figure 5.4(d),* φ_B *is given by* $hb(\mathtt{T_D}[3], \mathtt{T_P}[2]) \wedge hb(\mathtt{T_P}[3], \mathtt{T_D}[4])$. *Since* $write(\mathtt{T_D}[3], pagetable)$, $write(\mathtt{T_D}[4], memory)$, $read(\mathtt{T_P}[2], pagetable)$ *and* $read(\mathtt{T_P}[3], memory)$ *hold, we can apply the* TWOSTAGEACCESSBUG.1 *rule to infer* $\mathtt{TwoStageAccessBug}(\mathtt{T_D}[3:4], \mathtt{T_P}[2:3])$.

5.5 Accelerating CEGAR

Finally, we present a procedure for learning predicates for refinement in a CEGAR loop [Clarke *et al.*, 2000], with the help of TARA. An abstraction-refinement loop proceeds by building an abstract model of an input program and applying a model-checker on the abstract model. If the abstract model satisfies the correctness specification, then the input program is correct. Otherwise, the model-checker finds an abstract counterexample, i.e., an execution in the abstract model. The abstraction counterexample is spurious if there is no concrete execution that corresponds to the abstract counterexample. Given a spurious counterexample, the refinement procedure refines the abstract model. This is done by finding predicates that inform the abstraction procedure to construct the next abstract model by adding the relevant details to the current abstract model such that the spurious counterexample is absent from next abstract model. The process starts over with the newly refined abstraction. After a number of iterations, the abstract model may have no more counterexamples, which proves the correctness of the input program. For simpler presentation, we assume that the input program is correct and all the abstract counterexamples are spurious.

An abstraction-refinement loop often takes many iterations to find the right set of predicates to prove correctness of the input program. This usually depends on the design of the refinement procedure. Many heuristics have been proposed to find the relevant predicates in fewer iterations (see, for example, [Beyer *et al.*, 2007]). We aim to use TARA to accelerate the search for the right predicates, i.e., reduce the number of iterations of a CEGAR loop.

Our refinement procedure takes a concurrent abstract counterexample as input and returns refinement predicates. First, we analyse the counterexample using TARA and obtain an HB-formula φ_B that encodes a set of incorrect interleavings. We sample a number of interleavings from φ_B and attempt to compute refinement predicates that simultaneously remove all the sampled spurious inter-leavings using a method similar to *beautiful interpolants* [Albarghouthi and McMillan, 2013].

5.5.1 Abstraction and Refinement

An *abstract model* of a concurrent program $\mathcal{C} = \langle V, \{\mathsf{T}_1, \ldots, \mathsf{T}_k\}, SV, \langle LV_1, \ldots, LV_k \rangle \rangle$ is another concurrent program $\hat{\mathcal{C}} = \langle V, \{\hat{\mathsf{T}}_1, \ldots, \hat{\mathsf{T}}_k\}, SV, \langle LV_1, \ldots, LV_k \rangle \rangle$ such that, for each $i \in [1, k]$ and event ℓ in T_i, there is an event $\hat{\ell}$ in $\hat{\mathsf{T}}_i$ such that if $S_0 \ell S_1$ is feasible then $S_0 \hat{\ell} S_1$ is feasible.

In predicate abstraction, the abstract event $\hat{\ell}$ corresponding to an event ℓ is defined using a set of predicates as follows. Let us suppose predicates $\rho_1, \ldots, \rho_m$ are used for abstraction. Let $i \in [1, m]$. Let β_i be the weakest precondition of ℓ over ρ_i, and γ_i be the weakest precondition of ℓ over $\neg\rho_i$. Let $\hat{\beta}_i$ and $\hat{\gamma}_i$ be the weakest formulæ that are Boolean combinations of $\rho_1, \ldots, \rho_m$, and imply β_i and γ_i, respectively. $S_0 \hat{\ell} S_1$ is feasible iff $\forall i \in [1, m] : \ (S_0 \models \hat{\beta}_i \rightarrow S_1 \models \rho_i) \wedge (S_0 \models \hat{\gamma}_i \rightarrow S_1 \models \neg\rho_i)$.

Let $S_0 \hat{\ell}_1 S_1 \ldots \hat{\ell}_n S_n$ be a spurious counterexample, i.e., an execution in the abstract model that violates the specification. A refinement procedure computes additional predicates $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}, \alpha_n$ over program variables that satisfy the following constraint.

$$\alpha_0 = \mathit{true} \wedge \alpha_n = \mathit{false} \wedge \bigwedge_{i=1}^{n} \alpha_{i-1} \wedge \ell_i \rightarrow \alpha_i'$$

Note that the primed formula α_i' is the formula α_i where each variable v is replaced by its primed version v'. Recall that v' represents the value of v after the execution of a statement. An abstract model computed using predicates $\alpha_1, \ldots, \alpha_{n-1}$ is guaranteed to not exhibit the spurious counterexample [Henzinger *et al.*, 2004].

5.5.2 Sampling an HB-formula

We pass trace $\hat{\ell}_1 \ldots \hat{\ell}_n$ to TARA and obtain an HB-formula φ_B in DNF to represent bad abstract traces. φ_B is a formula over events $\hat{\ell}_1 \ldots \hat{\ell}_n$. With slight abuse of notation, we assume that φ_B is a formula over events $\ell_1 \ldots \ell_n$, which can be obtained by replacing abstract events by their corresponding concrete events in φ_B. We sample a few concrete infeasible traces that satisfy φ_B and try to compute the simultaneous refinement predicates, i.e., predicates that eliminate all the sampled traces from the abstract program. Intuitively, learning predicates simultaneously using multiple spurious counterexamples may allow us to find more *general* predicates. Both sampling and simultaneous refinement are heuristics choices. Here, we present one possible choice for the sampling. However, one can imagine a wide array of heuristics for these choices. In our sampling heuristic, we search for two disjuncts in φ_B of the form

$$\varphi_1 \wedge e_a < e_b \quad \text{and} \quad \varphi_2 \wedge e_b < e_a$$

such that negation of any HB-formula in φ_1 is not in φ_2. We generate traces π_1 and π_2 such that (a) they satisfy $\varphi_1 \wedge \varphi_2 \wedge e_a < e_b$ and $\varphi_1 \wedge \varphi_2 \wedge e_b < e_a$ respectively; and (b) they are of the following form with $e^1_{k_1} = e_a$ and $e^2_{k_2} = e_b$.

$$\begin{array}{rcccc}
\pi_1 = & \underbrace{e^0_1 \dots e^0_{k_0}}_{prefix} & \underbrace{e^1_1 \dots e^1_{k_1}}_{} & \underbrace{e^2_1 \dots e^2_{k_2}}_{} & \underbrace{e^3_1 \dots e^3_{k_3}}_{suffix} \\
 & & \multicolumn{2}{c}{\swarrow\!\!\!\!\searrow} & \\
\pi_2 = & \overbrace{e^0_1 \dots e^0_{k_0}} & \overbrace{e^2_1 \dots e^2_{k_2}} & \overbrace{e^1_1 \dots e^1_{k_1}} & \overbrace{e^3_1 \dots e^3_{k_3}}
\end{array}$$

If $\varphi_1 \wedge \varphi_2 \wedge e_a < e_b$ and $\varphi_1 \wedge \varphi_2 \wedge e_b < e_a$ are satisfiable, such traces always exist. Both the traces have a common prefix and suffix, and two middle segments $e^1_1 \dots e^1_{k_1}$ and $e^2_1 \dots e^2_{k_2}$ are swapped. From the traces, we obtain refinement predicates $\alpha_1 \dots \alpha_{k_0}$, $\beta_1 \dots \beta_{k_1+k_2}$, $\gamma_1 \dots \gamma_{k_1+k_2}$, and $\delta_1 \dots \delta_{k_3}$ by solving the following constraints.

$$\alpha_0 = true \wedge \bigwedge_{i=1}^{k_0} (\alpha_{i-1} \wedge \ell^0_i \to \alpha'_i) \wedge \alpha_{k_0} = \beta_0 = \gamma_0 \tag{prefix}$$

$$\bigwedge_{i=1}^{k_1} (\beta_{i-1} \wedge \ell^1_i \to \beta'_i) \wedge \bigwedge_{i=k_1+1}^{k_1+k_2} (\beta_{i-1} \wedge \ell^2_{i-k_1} \to \beta'_i) \tag{mid trace 1}$$

$$\bigwedge_{i=1}^{k_2} (\gamma_{i-1} \wedge \ell^2_i \to \gamma'_i) \wedge \bigwedge_{i=k_2+1}^{k_2+k_1} (\gamma_{i-1} \wedge \ell^1_{i-k_2} \to \gamma'_i) \tag{mid trace 2}$$

$$\delta_0 = \beta_{k_1+k_2} = \gamma_{k_1+k_2} \wedge \bigwedge_{i=1}^{k_3} (\delta_{i-1} \wedge \ell^3_i \to \delta'_i) \wedge \delta_{k_3} = false \tag{suffix}$$

In the above equations, the first and last constraints correspond to the prefix and suffix respectively. The second and third constraints correspond to the middle segments of the two traces.

5.5.3 Constraint Solving for Simultaneous Refinement

We discuss how to solve the above constraints for refinement. The above constraints are a set of non-recursive Horn clauses. Many techniques exist to solve such constraints (e.g. [Gupta *et al.*, 2011a; Bjørner *et al.*, 2013]). Since we are aiming for simultaneous refinement, we prefer the solutions for the unknown predicates to be simple atomic formulæ. If an unknown predicate appears as consequent of multiple implications (for example, α_{k_0+1}), then the solver may naturally give a solution that is a disjunction of two atomic formulæ. We use the method that is presented in Section 4 of [Albarghouthi and McMillan, 2013] for the theory of linear arithmetic that forces a solver to return solutions for the above constraints with single atomic formulæ if such a solution exists.

5.6 Implementation and Experiments

We have implemented Algorithms 5.1 and 5.2 in a tool TARA[1]. TARA consists of 4000 lines of C++ code and uses Z3 [de Moura and Bjørner, 2008] to discharge SMT queries. We use a new input format, CTRC, for specifying traces. The CTRC format consists of global and thread-local variables along with types and any initial valuations, and the statements (in SMT-LIB format) in each thread. This makes TARA independent and easy to use with any front-end that can translate statements to the SMT-LIB syntax. We use a modified version of CONREPAIR (Chapter 4) to generate CTRC files for bad traces. CONREPAIR, in turn, uses CBMC [Clarke *et al.*, 2004] to find bad traces in programs and CPACHECKER [Beyer and Keremoglu, 2011] to translate C statements into the SMT-LIB format.

TARA has a number of different output options. Algorithm 5.1 generates an HB-formula in DNF, which is often large. Algorithm 5.2 generates a succinct HB-formula in DNF, the sizes of whose disjuncts are locally minimised. In our experience, the unsat core provided by Z3 is often far from minimal. Hence, we first use Z3 to compute an unsat core and then use a custom minimisation technique—we use Z3 incrementally with triggers to activate and deactivate expressions for unsat core minimisation. TARA can also generate an HB-formula in CNF representing bad neighbourhoods. However, this is computationally more expensive.

Experiments. Our benchmarks are from a diverse set of sources, namely, the concurrency track of the 2014 software verification competition SV-COMP [Beyer, 2014] (suite sv) and the regression-test suites of CONREPAIR (Chapter 4) (suite cr) and ESBMC [Morse *et al.*, 2014] (suite es). We also use a set of small handmade examples with common bug patterns (suite hm). The cr suite contains simplified versions of real buggy code from the Linux kernel. To test the limits of TARA, we use the `loop-x` examples that have two threads each executing a loop of x iterations. For correct behaviour, each iteration should execute atomically with respect to iterations of the other thread. However, the locks required to ensure this are missing.

[1] available as open-source software along with benchmarks: `https://github.com/thorstent/TARA`

All measurements were done on an Intel core i5-3320M laptop with 8GB of RAM. Our results are presented in Table 5.1. The time reported only includes the time taken by TARA, and not the time needed to find a bad trace in the benchmark program. The #Threads/#Instrs column in Table 5.1 indicates the complexity of the benchmarks in terms of the number of threads and statements. The performance of SMT queries involving Φ_{CTP} is mostly influenced by the number and size of π-functions. The #π-functions/#Disjuncts column indicates the number of π-functions and average number of arguments per π-function.

The performance of TARA using Algorithm 5.1 and Algorithm 5.2 are in columns marked Algo.1 and Algo.2, respectively. For each algorithm, we report the number of iterations, the total time taken and the size of the generated φ_B (as the number of disjuncts and the average number of terms in each disjunct). Algorithm 5.1 times out after 10 minutes in many cases—in such cases, we report the number of loop iterations completed before the timeout. With Algorithm 5.2, TARA terminates within 5 seconds for each benchmark. This time is negligible compared to the time taken to find the initial counterexample trace. For example, CBMC took 2 minutes to find the trace `usb-serial-1`, while our analysis completed exploration of its bad neighbourhood in 2 seconds. We tested the limits of our tool in the `loop-x` examples. With 32 iterations per thread, we exceeded the timeout and hit the limit of our current implementation.

Table 5.1 Experiments: φ_B generation

				Iterations		Total time		Size of φ_B	
Name	Suite	#Threads/#Instrs	#π-functions/#Disjuncts	Algo.1	Algo.2	Algo.1	Algo.2	Algo.1	Algo.2
reorder_2	sv	2/3	2/2.0	1	1	18ms	28ms	1/2.0	1/2.0
define_use	cr	2/4	2/2.0	1	1	15ms	22ms	1/2.0	1/1.0
em28xx	cr	2/8	4/2.0	1	1	16ms	25ms	1/2.0	1/1.0
locks	es	3/8	10/1.6	12	2	27ms	37ms	12/5.5	2/4.0
2stage	hm	2/8	5/1.4	8	1	26ms	32ms	8/3.8	1/2.0
drbd_receiver	cr	2/9	5/1.6	40	1	42ms	28ms	40/3.9	1/1.0
md	cr	3/11	4/1.8	40	1	76ms	33ms	40/6.1	1/1.0
lazy01	sv	3/12	6/3.7	2	2	31ms	57ms	2/3.0	2/2.0
locks_hb	hm	4/13	10/2.2	>29.0k	7	TO	119ms	TO	6/3.0
lc_rc	cr	4/14	8/2.0	4.6k	1	21.4s	37ms	4.6k/16.7	1/1.0
barrier_locks	hm	3/18	17/2.6	10.6k	6	1.4min	521ms	10.6k/10.0	4/1.5
stateful01	sv	3/19	10/3.4	2.3k	2	10.5s	84ms	2.3k/9.4	2/1.0
read_write_lock	sv	4/22	16/3.4	9.2k	4	1.6min	319ms	9.2k/16.1	4/3.0
loop	hm	2/38	14/2.7	2	1	38ms	72ms	2/3.0	1/2.0
fib_bench	sv	3/39	24/3.6	>20.5k	2	TO	2.3s	TO	2/10.0
i2c_hid	cr	2/42	26/4.5	>23.4k	3	TO	615ms	TO	3/1.3
rtl8169-1	cr	7/71	22/2.7	>20.4k	1	TO	111ms	TO	1/2.0
rtl8169-2	cr	7/116	41/2.3	>7.3k	1	TO	463ms	TO	1/1.0
rtl8169-5	cr	7/134	48/3.1	>5.5k	1	TO	1.5s	TO	1/1.0
rtl8169-4	cr	7/142	48/3.0	>8.4k	9	TO	3.8s	TO	2/1.0
rtl8169-6	cr	7/144	52/2.9	>8.1k	1	TO	887ms	TO	1/1.0
usb_serial-1	cr	7/151	87/3.7	>5.5k	1	TO	1.9s	TO	1/1.0
usb_serial-2	cr	7/163	93/3.6	>4.4k	3	TO	4.4s	TO	1/1.0
rtl8169-3	cr	8/174	61/3.6	>4.2k	2	TO	2.7s	TO	1/1.0
usb_serial-3	cr	7/178	100/3.7	>4.3k	1	TO	2.1s	TO	1/1.0
loop-2	N/A	2/16	8/3.0	>4.0k	4	11.6s	135ms	4.0k/8.9	4/2.0
loop-8	N/A	2/64	32/9.0	>15.3k	16	TO	3s	TO	16/2.0
loop-32	N/A	2/256	128/33.0	>674	64	TO	35.5min	TO	64/2.0

5.6.1 Synchronisation Synthesis

We implemented the synthesis algorithm as an extension to TARA. Given a trace π, TARA supports synchronisation synthesis as an optional step after generating succinct representations of $\mathcal{N}_\pi^g$ and $\mathcal{N}_\pi^b$. The implementation first attempts to apply the rules ADD.BARRIER, ADD.LOCK and ADD.WAITSIGNAL (in that order). Then, the merging rules are applied, first merging locks across thread pairs, and then merging barriers and locks spanning multiple threads.

We report the results of synchronisation synthesis experiments in Table 5.2. In each case, we report the numbers of locks (#L), barriers (#B) and wait-signal (#WS) primitives synthesised. The synthesised synchronisation matched our (human) intuition about the repairs needed. Since TARA generates fairly small φ_G formulæ, the synthesis takes less than 50 microseconds in every case.

Table 5.2 Experiments: synchronisation synthesis
L : Locks WS : Wait-notifies B : Barriers.

Name	#L	#B	#WS	Name	#L	#B	#WS
reorder_2	1	0	0	loop	1	0	0
define_use	0	0	1	fib_bench	1	0	0
em28xx	0	0	1	i2c_hid	1	0	2
locks	1	0	0	rtl8169-1	0	0	1
2stage	0	0	1	rtl8169-2	0	0	1
drbd_receiver	0	0	1	rtl8169-5	0	0	1
md	0	0	1	rtl8169-4	0	0	2
lazy01	0	0	2	rtl8169-6	0	0	1
locks_hb	1	0	2	usb_serial-1	0	0	1
lc_rc	0	0	1	usb_serial-2	0	0	1
barrier_locks	1	1	0	rtl8169-3	0	0	1
stateful01	0	0	2	usb_serial-3	0	0	1
read_write_lock	4	0	0				

5.6.2 Bug Summarisation

Given a trace π, TARA supports bug summarisation as an optional step after generating φ_B. Starting from φ_B in DNF, the implementation attempts to apply the DATARACE, ATOMICITYVIOLATION, TWOSTAGEACCESSBUG and DEFINEUSE inference rules (in that order). Identical bug reports are merged to avoid duplicates.

The experimental results of using our TARA-based bug summarisation on our test-suite are presented in Table 5.3. We report the numbers of data races, atomicity violations, two-stage access bugs and define-use bugs inferred. The Human column in the table presents a classification of the bugs present in the benchmarks, as reported by an expert user. The last column indicates if

TARA's bug summary matched the human classification. For the majority of benchmarks, TARA summarised the bug correctly (Yes). In some cases, TARA did not infer a bug summary (–). For the usb_serial-1 benchmark, TARA's bug summary contradicted the human classification. For each example, the implementation takes at most 12 milliseconds.

Table 5.3 Experiments: bug summarisation
2S : 2-stage access bug DR : Data-race bug OAV : Other atomicity violation
DU : Define-use bugs OV : Order-violation bug (only human inspection)
Human : Human interpretation of the bug

Name	#2S	#DR	#AV	#DU	Human	Bug summary right?
reorder_2	0	0	0	1	DU	Yes
define_use	0	0	0	1	DU	Yes
em28xx	0	0	0	1	DU	Yes
locks	0	2	0	0	DR	Yes
2stage	1	0	0	0	2S	Yes
drbd_receiver	0	0	0	0	OV	–
md	0	0	0	1	DU	Yes
lazy01	0	0	0	0	OV	–
locks_hb	0	2	0	2	DR, DU	Yes
lc_rc	0	0	0	0	OV	–
barrier_locks	0	2	0	0	DR, OV	Yes
stateful01	0	0	0	0	OV	–
read_write_lock	0	0	4	0	AV	Yes
hm-loop	0	1	0	0	DR	Yes
fib_bench	0	0	2	0	AV	Yes
i2c_hid	0	0	1	0	AV, OV	Yes
rtl8169-1	0	0	0	1	DU	Yes
rtl8169-2	0	0	0	1	DU	Yes
rtl8169-5	0	0	0	0	OV	–
rtl8169-4	0	0	0	0	OV	–
rtl8169-6	0	0	0	0	OV	–
usb_serial-1	0	0	0	1	OV	No
usb_serial-2	0	0	0	0	OV	–
rtl8169-3	0	0	0	0	OV	–
usb_serial-3	0	0	0	0	OV	–

5.6.3 Accelerating CEGAR

We have implemented the abstraction refinement procedure in SATABS [Donaldson *et al.*, 2011] and refer to the modified version as SATABS[TARA]. In Table 5.4 we present the result of running SATABS and SATABS[TARA] on three hand crafted examples. Each of these examples contain two threads and 15-20 lines of code. Our method reduces the number of iterations in all the

Table 5.4 Experiments: CEGAR acceleration

Example	SATABS		SATABS[TARA]	
	Iterations	Time(s)	Iterations	Time(s)
example1	55	35.4	45	33.5
example2	65	45.7	60	47.9
example3	45	23.0	41	23.9

examples. However, the total time of verification increases for two examples due to the fact that our refinement procedure is not well optimised.

Chapter 6

Synthesis using an Implicit Specification

6.1 Problem Statement and Illustrative Example

In the past three chapters we introduced synthesis techniques for programs using an explicit specification, such as assertions. Explicit specifications 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 synthesises synchronisation code to ensure that every behaviour of the program under preemptive scheduling is included in the set of behaviours 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* [Yi and Flanagan, 2010]) can simplify the development of concurrent software, including operating system (OS) kernels, network servers, database systems, etc. [Sadowski and Yi, 2010; Ryzhyk *et al.*, 2009]. In the non-preemptive model, a thread can only be descheduled by voluntarily yielding control, e.g., by invoking a blocking operation. Synchronisation 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 [Sadowski and Yi, 2010] 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 synthesiser must then generate a concurrent program that is guaranteed to produce the same output as the sequential version [Bloem *et al.*, 2014]. 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 [Flanagan and Qadeer, 2003]. In the synthesised 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 statement 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 synchronisation 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 statement 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, synchronisation primitives can be used freely under non-preemptive scheduling. Internally, they are modelled using yields: for instance, a semaphore acquisition statement is modelled 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 behavioural equivalence between preemptive and non-preemptive executions, we focus on externally observable program behaviours: 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 behaviour is characterised in terms of its interaction with other modules. Given a multi-threaded program $\mathcal{C}$ and a synthesised program $\mathcal{C}'$ obtained by adding synchronisation to $\mathcal{C}$, $\mathcal{C}'$ is *preemption-safe* w.r.t. $\mathcal{C}$ if for each execution of $\mathcal{C}'$ under a preemptive scheduler, there is an observationally equivalent non-preemptive execution of $\mathcal{C}$. Our synthesis goal is to automatically generate a preemption-safe version of the input program.

We rely on abstraction to achieve efficient synthesis of multi-threaded programs. We propose a simple, *data-oblivious* abstraction inspired by an analysis of synchronisation 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 modelled as writes to a special memory location, with independent interfaces modelled 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 $\mathcal{C}_{abs}$.

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.

Our abstraction is reminiscent of previously used abstractions that track reads and writes to individual locations (e.g., [Vechev *et al.*, 2010b; Alglave *et al.*, 2014]). 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.

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 $\mathcal{C}$ under non-preemptive scheduling, and P_{abs}, representing the abstraction of the synthesised program

$\mathcal{C}'$ under preemptive scheduling. We show that preemption-safety of $\mathcal{C}'$ w.r.t. $\mathcal{C}$ is implied by preemption-safety of the abstract synthesised program $\mathcal{C}'_{abs}$ w.r.t. the abstract original program $\mathcal{C}_{abs}$, 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 [Bertoni *et al.*, 1982], we show that the antichain-based algorithm for standard language inclusion [de Wulf *et al.*, 2006] 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 $\mathcal{C}_{abs}$, which when executed in order 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 paper [Černý *et al.*, 2015b], 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 minimising the number of lock statements (leading to coarse-grained locking) and maximising 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 synthesised 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.

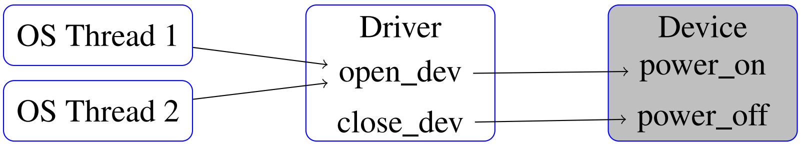
We implemented our synthesis algorithm in a new prototype tool called LISS (Language Inclusion-based Synchronisation Synthesis) and evaluated it on a series of device driver benchmarks, including an Ethernet driver for Linux and the synchronisation skeleton of a USB-to-serial

Figure 6.1 Running example

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

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

Figure 6.2 Interaction of the device driver with the OS and the device



controller driver, as well as an in-memory key-value store server. First, LISS was able to detect and eliminate all but two known race conditions in our examples; these included one race condition that we previously missed when synthesising from explicit specifications (Chapter 4), due to a missing assertion. Second, our abstraction proved highly efficient: LISS runs an order of magnitude faster on the more complicated examples than our previous synthesis tool based on the CBMC model checker. Third, our coarse abstraction proved surprisingly precise in practice: across all our benchmarks, we only encountered three program locations where manual abstraction refinement was needed to avoid the generation of unnecessary synchronisation. 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.

6.1.1 Illustrative Example

Figure 6.1 contains our running example, a part of a device driver. A driver interfaces the operating system with the hardware device (as illustrated in Figure 6.2) 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

Figure 6.3 Abstraction of the running example

```
        procedure OPEN_DEV_ABS
$\ell_{1a}$:    read(open)
$\ell_{1b}$:    if (*) then
$\ell_2$:           write(dev)
                end if
$\ell_{3a}$:    read(open)
$\ell_{3b}$:    write(open)
$\ell_4$:       yield
        end procedure
```

```
        procedure CLOSE_DEV_ABS
$\ell_{5a}$:    read(open)
$\ell_{5b}$:    if (*) then
$\ell_{6a}$:        read(open)
$\ell_{6b}$:        write(open)
$\ell_{7a}$:        read(open)
$\ell_{7b}$:        if (*) then
$\ell_8$:               write(dev)
                    end if
                end if
$\ell_9$:       yield
        end procedure
```

interact with the device. The first thread to call OPEN_DEV will cause the driver to power up the device (line ℓ_2), the last thread to call CLOSE_DEV will cause the driver to power down the device (line ℓ_8). 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 Figure 6.1 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 synchronisation, 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 6.3 contains the abstracted versions of the two procedures, OPEN_DEV_ABS and CLOSE_DEV_ABS. For instance, the statement $\mathtt{open} := \mathtt{open} + 1$ is abstracted to the two statements labelled 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

Figure 6.4 Running example with the synthesised locks

```
        procedure OPEN_DEV
            lock($LkVar$)
$\ell_1$:   if (open == 0) then
$\ell_2$:       POWER_UP
            end if
$\ell_3$:   open := open + 1
            unlock($LkVar$)
$\ell_4$:   yield
        end procedure
```

```
        procedure CLOSE_DEV
            lock($LkVar$)
$\ell_5$:   if (open > 0) then
$\ell_6$:       open := open − 1
$\ell_7$:       if (open == 0) then
$\ell_8$:           POWER_DOWN
                end if
            end if
            unlock($LkVar$)
$\ell_9$:   yield
        end procedure
```

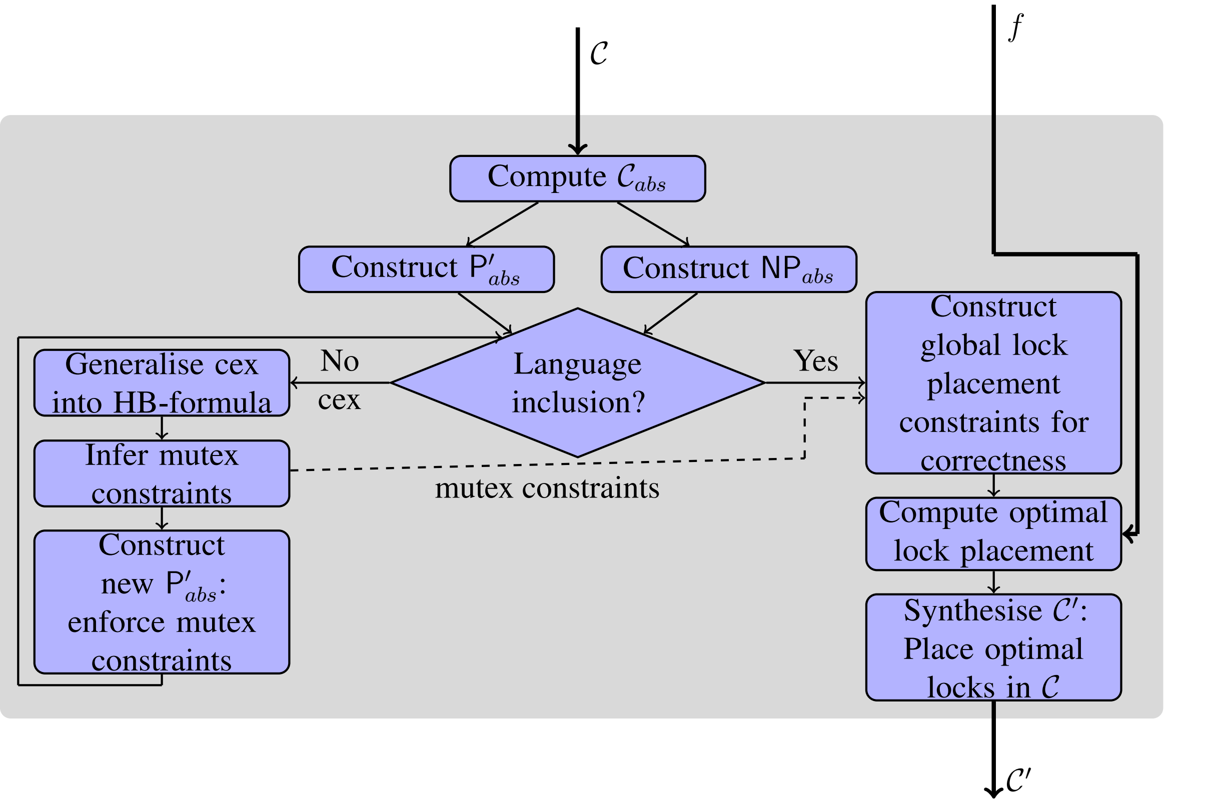
trace cannot be transformed by swapping independent events into any trace possible under a non-preemptive scheduler. This is because statements ℓ_{3b} : $\mathsf{write}(\mathsf{open})$ and ℓ_{1a} : $\mathsf{read}(\mathsf{open})$ are not independent. Further, ℓ_2 : $\mathsf{write}(\mathsf{dev})$ is not 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 algorithm finds this problem, and stores it as a mutex constraint: $\mathrm{mtx}([\ell_{1a} : \ell_{3b}], [\ell_2 : \ell_{3b}])$. 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 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 $\mathrm{mtx}([\ell_{5a} : \ell_8], [\ell_{6b} : \ell_8])$. With both mutex constraints the program is correct. Our lock placement procedure then encodes these constraints in SMT and the models of the SMT formula are all the correct lock placements. In Figure 6.4 we show OPEN_DEV and CLOSE_DEV with the inserted locks.

6.2 Solution Overview

Reduction of preemption-safety to language inclusion. To ensure tractability of checking preemption-safety, we build the abstract program $\mathcal{C}_{abs}$ from $\mathcal{C}$ using the abstraction function described in Section 6.3. Under abstraction, we model each thread as a nondeterministic finite

Figure 6.5 Solution Overview


$\mathcal{C}$
f
Compute $\mathcal{C}_{abs}$
Construct P'_{abs}
Construct NP_{abs}
Language inclusion?
No
cex
Yes
Generalise cex into HB-formula
Infer mutex constraints
Construct new P'_{abs}: enforce mutex constraints
mutex constraints
Construct global lock placement constraints for correctness
Compute optimal lock placement
Synthesise $\mathcal{C}'$: Place optimal locks in $\mathcal{C}$
$\mathcal{C}'$

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 $[\![\mathcal{C}_{abs}]\!]^{NP}$ and $[\![\mathcal{C}'_{abs}]\!]^{P}$, 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 algorithm in Section 6.4.

Inference of mutex constraints from generalised 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 which when executed in order produce an observation sequence that is in $[\![\mathcal{C}_{abs}]\!]^{P}$, but not in $[\![\mathcal{C}'_{abs}]\!]^{NP}$. We analyse cex to infer constraints on $\mathcal{L}(\mathsf{P}'_{abs})$ 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 *HB-formula* ϕ — a Boolean combination of *happens-before* ordering constraints between events — representing all counterexamples in $nhood(cex)$. Thus cex is generalised into a larger set of counterexamples represented as ϕ. From ϕ, we infer possible *mutual exclusion (mutex) constraints* on $\mathcal{L}(\mathsf{P}'_{abs})$ that can eliminate all counterexamples satisfying ϕ. We describe the procedure for finding constraints from cex in Section 6.5.1.

Automaton modification for enforcing mutex constraints. Once we have the mutex constraints inferred from a generalised 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 [Černý *et al.*, 2015b] that modifies $\mathcal{C}'_{abs}$, constructs a new automaton P'_{abs} from $\mathcal{C}'_{abs}$ and then restarts the language inclusion check. The greedy approach inserts locks into $\mathcal{C}'_{abs}$ that are never removed in a future iteration, which can lead to inefficient lock placement. For example a larger lock may be placed that completely surrounds an earlier placed lock.

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 Section 6.6.

Given a cost function f, we compute a lock placement that satisfies the global constraints and is optimal w.r.t. f. We then synthesise the final output $\mathcal{C}'$ by inserting the computed lock placement in $\mathcal{C}$. We present various objective functions and describe the computation of their respective optimal solutions in Section 6.7.

6.3 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 $\mathcal{W}_{abs}$ for concurrent programs written in $\mathcal{W}$ 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 modelled as writes to a special memory location (`dev`). Even inputs are modelled 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

Figure 6.6 Syntax of $\mathcal{W}_{abs}$

$var ::=$	Variables
$ShVar$	Shared variable
`dev`	Variable for interaction with channels
$GrdExpr ::=$	Expression over guard variables
`true/false`	Boolean constant
$GrdVar$	Guard variable
$boolop(GrdExpr_1, \ldots, GrdExpr_n)$	Boolean operation
$LbStmt ::=$	Labelled Statement
$\ell : stmt$	Statement annotated with a location
$LbStmt_1; LbStmt_2$	Sequence of statements
$stmt ::=$	Statement
skip	marks the end of the thread
$\mathsf{read}(var)$	Read a shared variable var
$\mathsf{write}(var)$	Write to shared variable var
$\mathsf{if}\ (*)\ \mathsf{then}\ LbStmt_1\ \mathsf{else}\ LbStmt_2$	conditional
$\mathsf{while}\ (*)\ LbStmt$	while loop
$\mathsf{lock}(LkVar)$	Locks the mutex lock
$\ldots$	remaining statements as in Figure 2.1

non-deterministically and tracked. Given $\mathcal{C}$ written in $\mathcal{W}$, we denote by $\mathcal{C}_{abs}$ the corresponding abstract program written in $\mathcal{W}_{abs}$.

Abstract Syntax (Figure 6.6). In the figure, var denotes all shared program variables and the `dev` variable. The syntax of all synchronisation 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. 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.

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

Single-thread semantics (Figure 6.7). The abstract state of a single thread tid is given simply by $\langle \mathcal{V}_o, \ell \rangle$ where $\mathcal{V}_o$ 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 $\theta \in \{(\mathsf{read}, sv), (\mathsf{write}, sv), \mathsf{then}, \mathsf{else}, \mathsf{loop}, \mathsf{exitloop}\}$. 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 $\{\mathsf{if}, \mathsf{else}, \mathsf{loop}, \mathsf{exitloop}\}$. Figure 6.7 presents the rules for

Figure 6.7 Partial set of rules for single-thread semantics of $\mathcal{W}_{abs}$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{read}(var) \qquad \ell' = \mathsf{succ}(\ell)}{\langle \mathcal{V}_o, \ell \rangle \xrightarrow{(tid,(\mathsf{read},var),\ell)} \langle \mathcal{V}_o, \ell' \rangle} \ \textsc{Read}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{write}(var) \qquad \ell' = \mathsf{succ}(\ell)}{\langle \mathcal{V}_o, \ell \rangle \xrightarrow{(tid,(\mathsf{write},var),\ell)} \langle \mathcal{V}_o, \ell' \rangle} \ \textsc{Write}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{if}\ (*)\ \mathsf{then}\ ls_1\ \mathsf{else}\ ls_2 \qquad \ell' = \mathsf{succ}_1(\ell)}{\langle \mathcal{V}_o, \ell \rangle \xrightarrow{(tid,\mathsf{then},\ell)} \langle \mathcal{V}_o, \ell' \rangle} \ \textsc{If1}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{if}\ (*)\ \mathsf{then}\ ls_1\ \mathsf{else}\ ls_2 \qquad \ell' = \mathsf{succ}_2(\ell)}{\langle \mathcal{V}_o, \ell \rangle \xrightarrow{(tid,\mathsf{else},\ell)} \langle \mathcal{V}_o, \ell' \rangle} \ \textsc{If2}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{while}\ (*)\ ls \qquad \ell' = \mathsf{succ}_1(\ell)}{\langle \mathcal{V}_o, \ell \rangle \xrightarrow{(tid,\mathsf{loop},\ell)} \langle \mathcal{V}_o, \ell' \rangle} \ \textsc{While1}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{while}\ (*)\ ls \qquad \ell' = \mathsf{succ}_2(\ell)}{\langle \mathcal{V}_o, \ell \rangle \xrightarrow{(tid,\mathsf{exitloop},\ell)} \langle \mathcal{V}_o, \ell' \rangle} \ \textsc{While2}$$

Figure 6.8 Example motivating branch-tagging

```
x := 0; y := 0
Thread T1                    Thread T2
$\ell_1$: x := 0                  $\ell_6$: x := 1
$\ell_2$: if (y) then
$\ell_3$:     yield
    end if
$\ell_4$: if (x) then
$\ell_5$:     output(ch, 10)
    end if
```

statements unique to $\mathcal{W}_{abs}$; the rules for statements common to $\mathcal{W}_{abs}$ and $\mathcal{W}$ are the same.

Concurrent semantics. A state of an abstract concurrent program is either $\langle \texttt{terminated} \rangle$, $\langle \texttt{invalid} \rangle$, or is given by $\langle \mathcal{V}_o, ctid, (\ell_1, \ldots, \ell_n) \rangle$ where $\mathcal{V}_o$ 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 T_1 to T_n, respectively. There is no $\langle \texttt{failed} \rangle$ state because $\mathcal{W}_{abs}$ does not have assertions. The non-preemptive and preemptive semantics of a concurrent program written in $\mathcal{W}_{abs}$ are defined in the same way as that of a concurrent program written in $\mathcal{W}$ minus the Assertion failure rule.

Figure 6.9 Abstraction function from $\mathcal{W}$ to $\mathcal{W}_{abs}$

$$
\begin{array}{rcl}
\langle\langle \mathit{LoExp} \rangle\rangle_\ell & = & \textit{(nothing)} \\
\langle\langle \mathit{ShVar} \rangle\rangle_\ell & = & \ell : \mathsf{read}(\mathit{ShVar}) \\
\langle\langle \mathit{op}(\mathit{ShVar}, \mathit{LoExp}_1, \ldots, \mathit{LoExp}_n) \rangle\rangle_\ell & = & \ell : \mathsf{read}(\mathit{ShVar}) \\
\langle\langle \mathit{LbStmt}_1; \mathit{LbStmt}_2 \rangle\rangle & = & \langle\langle \mathit{LbStmt}_1 \rangle\rangle; \langle\langle \mathit{LbStmt}_2 \rangle\rangle \\
\langle\langle \ell : \mathit{ShVar} := \mathit{LoExp} \rangle\rangle & = & \ell : \mathsf{write}(\mathit{ShVar}) \\
\langle\langle \ell : \mathit{LoVar} := \mathit{ShExp} \rangle\rangle & = & \langle\langle \mathit{ShExp} \rangle\rangle_\ell \\
\langle\langle \ell : \mathit{ShVar} := \mathsf{havoc} \rangle\rangle & = & \ell : \mathsf{write}(\mathit{ShVar}) \\
\langle\langle \ell : \mathit{ShVar} := \mathsf{input}(\mathit{ch}) \rangle\rangle & = & \ell : \mathsf{write}(\mathtt{dev}); \ell : \mathsf{write}(\mathit{ShVar}) \\
\langle\langle \ell : \mathsf{output}(\mathit{ShVar}, \mathit{ch}) \rangle\rangle & = & \ell : \mathsf{read}(\mathit{ShVar}); \ell : \mathsf{write}(\mathtt{dev}) \\
\langle\langle \ell : \mathsf{if}\ (\mathit{ShExp})\ \mathsf{then}\ \mathit{LbStmt}_1\ \mathsf{else}\ \mathit{LbStmt}_2 \rangle\rangle & = & \langle\langle \mathit{ShExp} \rangle\rangle_\ell; \ell : \mathsf{if}\ (*)\ \mathsf{then}\ \langle\langle \mathit{LbStmt}_1 \rangle\rangle\ \mathsf{else}\ \langle\langle \mathit{LbStmt}_2 \rangle\rangle \\
\langle\langle \ell : \mathsf{while}\ (\mathit{ShExp})\ \mathit{LbStmt} \rangle\rangle & = & \langle\langle \mathit{ShExp} \rangle\rangle_\ell; \ell : \mathsf{while}\ (*)\ \langle\langle \mathit{LbStmt} \rangle\rangle; \langle\langle \mathit{ShExp} \rangle\rangle_\ell \\
\langle\langle \ell : \mathsf{assert}(\mathit{ShExp}) \rangle\rangle & = & \textit{(nothing)} \\
\langle\langle \ell : \mathsf{await}(\mathit{ShExp}) \rangle\rangle & = & \langle\langle \mathit{ShExp} \rangle\rangle_\ell; \ell : \mathsf{while}\ (*)\ \langle\langle \mathit{LbStmt} \rangle\rangle \\
\langle\langle \ell : \mathsf{lock}(\mathit{LkVar}) \rangle\rangle & = & \ell : \mathsf{lock}(\mathit{LkVar}) \\
 & \ldots &
\end{array}
$$

6.3.1 Abstraction function (Figure 6.9)

A thread in $\mathcal{W}$ can be translated to $\mathcal{W}_{abs}$ using the abstraction function $\langle\langle\rangle\rangle$. The abstraction replaces all global variable access with $\mathsf{read}(\mathit{var})$ and $\mathsf{write}(\mathit{var})$ and replaces branching conditions with nondeterminism $(*)$. Since we will use the abstraction only for programs with implicit specification the assert statements are removed during the translation. All synchronisation primitives remain unaffected by the abstraction. The abstraction may result in duplicate labels ℓ, which are replaced by fresh labels. goto statements are rewritten accordingly. For example in our running example in Figure 6.1 the abstraction of ℓ_1 results in two abstract labels ℓ_{1a} and ℓ_{1b} in Figure 6.3.

6.4 Checking Preemption-safety

6.4.1 Reduction of Preemption-safety to Language Inclusion

Soundness of the abstraction. Formally, two observable behaviours $\omega_1 = \alpha_0 \ldots \alpha_k$ and $\omega_2 = \beta_0 \ldots \beta_k$ of an abstract program $\mathcal{C}_{abs}$ 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), with $a \in \{(\mathsf{read}, \mathit{var}), (\mathsf{write}, \mathit{var}), \mathsf{then}, \mathsf{else}, \mathsf{loop}, \mathsf{loopexit}\}$ are equal, (*A2*) For each variable var, the subsequences of $\alpha_0 \dots \alpha_k$ and $\beta_0 \dots \beta_k$ 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 $\mathcal{C}_{abs}$ and $\mathcal{C}'_{abs}$ in $\mathbb{W}_{abs}$ such that $\mathcal{C}'_{abs}$ is obtained by adding locks to $\mathcal{C}_{abs}$, $\mathcal{C}'_{abs}$ is *preemption-safe* w.r.t. $\mathcal{C}_{abs}$ if $[\![\mathcal{C}'_{abs}]\!]^P \Subset_{abs} [\![\mathcal{C}_{abs}]\!]^{NP}$.

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

To illustrate this we use the program in Figure 6.8, which is not preemption-safe. To see this consider the observation $(\mathtt{T1}, \mathsf{out}, 10, \mathtt{ch})$ 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 initialised 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 observation sequence that is observable under the abstract preemptive semantics, but not under the abstract non-preemptive semantics. This observation sequence is $(\mathsf{T1}, (\mathsf{write}, \mathtt{x}), \ell_1), (\mathsf{T2}, (\mathsf{write}, \mathtt{x}), \ell_6)$, $(\mathsf{T1}, (\mathsf{read}, \mathtt{y}), \ell_2), (\mathsf{T1}, \mathsf{else}, \ell_2), (\mathsf{T1}, (\mathsf{read}, \mathtt{x}), \ell_4), (\mathsf{T1}, \mathsf{then}, \ell_2), (\mathsf{T1}, (\mathsf{write}, \mathtt{dev}), \ell_5)$. 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: $(\mathsf{T1}, (\mathsf{write}, \mathtt{x}), \ell_1), (\mathsf{T2}, (\mathsf{write}, \mathtt{x}), \ell_6), (\mathsf{T1}, (\mathsf{read}, \mathtt{y}), \ell_2)$, $(\mathsf{T1}, (\mathsf{read}, \mathtt{x}), \ell_4), (\mathsf{T1}, (\mathsf{write}, \mathtt{dev}), \ell_5)$. 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 6.4.1 (soundness)**.** *Given concurrent program $\mathcal{C}$ and a synthesised program $\mathcal{C}'$ obtained by adding locks to $\mathcal{C}$, $[\![\mathcal{C}'_{abs}]\!]^P \Subset_{abs} [\![\mathcal{C}_{abs}]\!]^{NP} \implies [\![\mathcal{C}']\!]^P \Subset [\![\mathcal{C}]\!]^{NP}$.*

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

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

Towards contradiction we assume there is such an equivalent sequence ω_{abs} in $[\![\mathcal{C}_{abs}]\!]^{NP}$. 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 write(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. □

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 $\approx^t_I$ denote the reflexive transitive closure of $\approx_I$.[1] Given a language $\mathcal{L}$ over Σ, the closure of $\mathcal{L}$ w.r.t. I, denoted $\mathrm{Clo}_I(\mathcal{L})$, is the set $\{\sigma \in \Sigma^* \colon \exists \sigma' \in \mathcal{L} \text{ with } (\sigma, \sigma') \in \approx^t_I\}$. Thus, $\mathrm{Clo}_I(\mathcal{L})$ consists of all words that can be obtained from some word in $\mathcal{L}$ by repeatedly commuting adjacent independent symbol pairs from I.

[1] The equivalence classes of $\approx^t_I$ are Mazurkiewicz traces.

Definition 6.4.2 (Language inclusion modulo an independence relation)**.** Given NFAs A, B over a common alphabet Σ and an independence relation I over Σ, the language inclusion problem modulo I is: $\mathcal{L}(\mathrm{A}) \subseteq \mathrm{Clo}_I(\mathcal{L}(\mathrm{B}))$?

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 $var_\alpha \neq var_\beta$

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 $[\![\mathcal{C}_{abs}]\!]^{NP}$ and $[\![\mathcal{C}'_{abs}]\!]^{P}$, respectively. $\mathcal{C}_{abs}$ is the abstract program corresponding to the input program $\mathcal{C}$ and $\mathcal{C}'_{abs}$ is the program corresponding to the result of the synthesis $\mathcal{C}'$. It turns out that preemption-safety of $\mathcal{C}'$ w.r.t. $\mathcal{C}$ is implied by preemption-safety of $\mathcal{C}'_{abs}$ w.r.t. $\mathcal{C}_{abs}$, 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 6.4.3. *Given concurrent programs $\mathcal{C}$ and $\mathcal{C}'$, $[\![\mathcal{C}'_{abs}]\!]^{P} \Subset_{abs} [\![\mathcal{C}_{abs}]\!]^{NP}$ iff $\mathcal{L}(\mathsf{P}'_{abs}) \subseteq \mathrm{Clo}_{I_D}(\mathcal{L}(\mathsf{NP}_{abs}))$.*

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

6.4.2 Checking Language Inclusion

We first focus on the problem of language inclusion modulo an independence relation (Definition 6.4.2). This question corresponds to preemption-safety (Theorem 6.4.1, Proposition 6.4.3) and its solution drives our synchronisation synthesis.

Theorem 6.4.4. *For NFAs A, B over alphabet Σ and a symmetric, irreflexive independence relation $I \subseteq \Sigma \times \Sigma$, the problem $\mathcal{L}(A) \subseteq \mathrm{Clo}_I(\mathcal{L}(B))$ is undecidable [Bertoni* et al.*, 1982].*

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

Theorem 6.4.5. *For NFAs P'_{abs} and NP_{abs} constructed from $\mathcal{C}_{abs}$, the problem $\mathcal{L}(\mathsf{P}'_{abs}) \subseteq \mathrm{Clo}_{I_D}(\mathcal{L}(\mathsf{NP}_{abs}))$ is undecidable.*

Proof. Our proof is by reduction from the language inclusion modulo an independence relation problem (Definition 6.4.2). Theorem 6.4.5 follows from the undecidability of this problem (Theorem 6.4.4).

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

Figure 6.10 Simulator algorithm

Thread T1

```
$\ell_1$: while (*) do
$\ell_2$:     signal(ch-sym)        ▷ choose symbol
$\ell_3$:     wait_reset(ch-sym-compl)
$\ell_4$:     $sA^1 \leftarrow \Delta_A^1(sA^1, \ldots, sA^n, \tau^1, \ldots, \tau^p)$
$\ell_5$:     ...
$\ell_6$:     $sA^n \leftarrow \Delta_A^n(sA^1, \ldots, sA^n, \tau^1, \ldots, \tau^p)$
$\ell_7$:     $sB^1 \leftarrow \Delta_B^1(sB^1, \ldots, sB^m, \tau^1, \ldots, \tau^p)$
$\ell_8$:     ...
$\ell_9$:     $sB^m \leftarrow \Delta_B^m(sB^1, \ldots, sB^m, \tau^1, \ldots, \tau^p)$
$\ell_{10}$: end while
$\ell_{11}$: $final \leftarrow \big(simA \implies$
          $\bigvee_{q \in F_A}(sA^1 = q^1 \wedge \cdots \wedge sA^n = q^n)\big)$
            $\wedge\ \big(\neg simA \implies$
          $\bigvee_{q \in F_B}(sB^1 = q^1 \wedge \cdots \wedge sB^m = q^m)\big)$
$\ell_{12}$: assume(final)
```

Thread T2

```
$\ell_{12}$: $simA \leftarrow true$
$\ell_{13}$: $simA \leftarrow false$
```

Thread T$_\alpha$

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

transition for each symbol. We show that we can construct a program $\mathcal{C}_{abs}$ that is preemption-safe iff $\mathcal{L}(A) \subseteq \mathrm{Clo}_I(\mathcal{L}(B))$.

For our reduction we construct a program $\mathcal{C}_{abs}$ that simulates A or B if run with a preemptive scheduler and simulates only B if run with a non-preemptive scheduler. Note that $\mathcal{L}(A) \cup \mathcal{L}(B) \subseteq \mathrm{Clo}_I(\mathcal{L}(B))$ iff $\mathcal{L}(A) \subseteq \mathrm{Clo}_I(\mathcal{L}(B))$. 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) $\mathcal{C}_{abs}$ 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 Figure 6.10. States and symbols of A and B are mapped to natural numbers and represented as bitvectors to enable simulation using the language $\mathcal{W}_{abs}$. In particular we use Boolean guard variables from $\mathcal{W}_{abs}$ 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 $x^1 \ldots x^n$.

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 $\mathsf{T1}$ a symbol $\alpha \in \Sigma$ is chosen non-deterministically and applied to both automata (we discuss this step in the next paragraph). Whether thread $\mathsf{T1}$ 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 $\mathsf{T2}$ and can only be `true` if $\mathsf{T2}$ is preempted between locations ℓ_{12} and ℓ_{13}. With the non-preemptive scheduler the variable `simA` will always be `false` because thread $\mathsf{T2}$ cannot be preempted. The simulator can only reach the $\langle$`terminated`$\rangle$ state if all assumptions hold as otherwise it would end in the $\langle$`invalid`$\rangle$ state. The guard `final` will only be assigned `true` in ℓ_{11} 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_{11} executes atomically and the value of `simA` cannot change during its evaluation. This means that P'_{abs} simulates $\mathcal{L}(A) \cup \mathcal{L}(B)$ and NP_{abs} simulates $\mathcal{L}(B)$.

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 $\mathcal{W}_{abs}$. We therefore use the fact that the next thread to execute is chosen non-deterministically at a preemption point. We define a thread T_α for every $\alpha \in \Sigma$ that assigns to τ the number α maps to. Threads T_α 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 T_α 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 T_α, notifies thread $\mathsf{T1}$ using condition variable `ch-sym-compl`. Another symbol can only be produced in the next loop iteration of $\mathsf{T1}$.

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 T_α has to perform one loop iteration. This implicitly produces a loop tag $(\mathsf{T}_\alpha, \mathsf{loop}, \ell_{14})$.
- *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 $h(\alpha) = (\mathsf{T}_\alpha, \mathsf{loop}, \ell_{14}); (\mathsf{T}_\alpha, (\mathsf{write}, v_{\{\alpha,\alpha_1\}}), \ell_{o1}); \cdots; (\mathsf{T}_\alpha, (\mathsf{write}, v_{\{\alpha,\alpha_k\}}), \ell_{ok})$. 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 $(\mathsf{T}_{\alpha_{\{1,2\}}}, (\mathsf{write}, v_{\{\alpha_1,\alpha_2\}}), \ell_{oi})$ 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 $\mathcal{C}_{abs}$ is preemption-safe iff $\mathcal{L}(A) \subseteq \mathrm{Clo}_I(\mathcal{L}(B))$. By Proposition 6.4.3 it suffices to show that $\mathcal{L}(A) \subseteq \mathrm{Clo}_I(\mathcal{L}(B))$ iff $\mathcal{L}(\mathsf{P}'_{abs}) \subseteq \mathrm{Clo}_{I_D}(\mathcal{L}(\mathsf{NP}_{abs}))$.

1. We assume that $\mathcal{L}(A) \subseteq \mathrm{Clo}_I(\mathcal{L}(B))$. Then, for every word $\sigma \in \mathcal{L}(A)$ we have that $\sigma \in \mathrm{Clo}_I(\mathcal{L}(B))$. By construction $h(\sigma) \in \mathcal{L}(\mathsf{P}'_{abs})$. It remains to show that $h(\sigma) \in \mathrm{Clo}_{I_D}(\mathcal{L}(\mathsf{NP}_{abs}))$. By $\sigma \in \mathrm{Clo}_I(\mathcal{L}(B))$ we know there exists a word $\sigma' \in \mathcal{L}(B)$, such that $\sigma' \approx^t_I \sigma$. Therefore also $h(\sigma') \approx^t_{I_D} h(\sigma)$ and by construction $h(\sigma') \in \mathcal{L}(\mathsf{NP}_{abs})$.
2. We assume that $\mathcal{L}(A) \nsubseteq \mathrm{Clo}_I(\mathcal{L}(B))$. Then, there exists a word $\sigma \in \mathcal{L}(A)$ such that $\sigma \notin \mathrm{Clo}_I(\mathcal{L}(B))$. By construction $h(\sigma) \in \mathcal{L}(\mathsf{P}'_{abs})$. Let us assume towards contradiction that $h(\sigma) \in \mathrm{Clo}_{I_D}(\mathcal{L}(\mathsf{NP}_{abs}))$. Then there exists a word ω in $\mathcal{L}(\mathsf{NP}_{abs})$ such that $\omega \approx^t_{I_D} h(\sigma)$. By construction, this implies there exists some $\sigma' \in \mathcal{L}(B)$ such that $\omega = h(\sigma')$ and $h(\sigma') \approx^t_{I_D} h(\sigma)$. Thus, there exists $\sigma' \in \mathcal{L}(B)$ such that $\sigma' \approx^t_I \sigma$. This implies $\sigma \in \mathrm{Clo}_I(\mathcal{L}(B))$, 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 Section 6.4.1. 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 $\mathcal{L}$ over Σ, we use $\mathrm{Clo}_{k,I}(\mathcal{L})$ to denote the set $\{\sigma \in \Sigma^* : \exists \sigma' \in \mathcal{L} \text{ with } (\sigma, \sigma') \in \asymp^k\}$. In other words, $\mathrm{Clo}_{k,I}(\mathcal{L})$ consists of all words which can be generated from $\mathcal{L}$ using a finite-state transducer that remembers at most k symbols of its input words in its states. By definition we have $\mathrm{Clo}_{0,I}(\mathcal{L}) = \mathcal{L}$.

Definition 6.4.6 (Bounded language inclusion modulo an independence relation)**.** Given NFAs A, B over Σ, $I \subseteq \Sigma \times \Sigma$ and a constant $k > 0$, the k-bounded language inclusion problem modulo I is: $\mathcal{L}(\mathrm{A}) \subseteq \mathrm{Clo}_{k,I}(\mathcal{L}(\mathrm{B}))$?

We present an algorithm to check k-bounded language inclusion modulo I, based on the antichain algorithm for standard language inclusion [de Wulf *et al.*, 2006].

6.4.3 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 $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ for NFAs $A = (Q_A, \Sigma, \Delta_A, Q_{\iota,A}, F_A)$ and $B = (Q_B, \Sigma, \Delta_B, Q_{\iota,B}, F_B)$, the antichain algorithm proceeds by exploring A and B in lockstep. Without loss of generality we assume that A and B do not have ϵ-transitions. While A is explored nondeterministically, B is determinised 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 $p_1, \dots, p_n$ in the antichain with $p' \sqsubseteq p_1, \dots, p_n$, then $p_1, \dots, p_n$ are removed from the antichain (*Rule R2*). The algorithm terminates by either reporting a counterexample, or by declaring success when the frontier becomes empty.

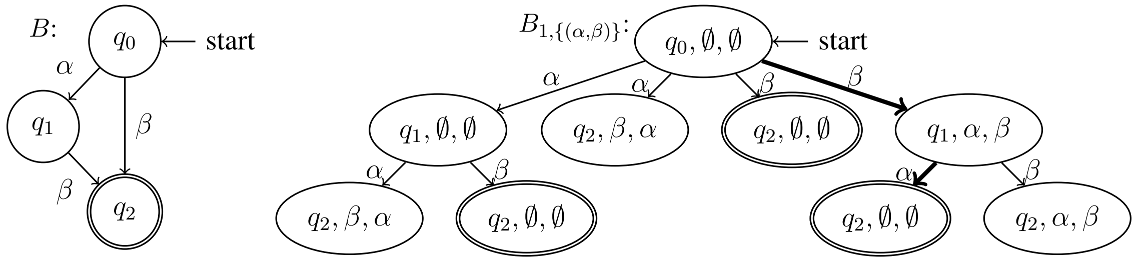
6.4.4 Antichain algorithm for k-bounded language inclusion modulo I

This algorithm is essentially the same as the standard antichain algorithm, with the automaton B above replaced by an automaton $B_{k,I}$ accepting $\mathrm{Clo}_{k,I}(\mathcal{L}(\mathrm{B}))$. The set $Q_{B_{k,I}}$ of states of $B_{k,I}$ consists of triples (s_B, η_1, η_2), where $s_B \in Q_B$ and η_1, η_2 are 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 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 i^{th} 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 $\eta''_1 := \eta'_1$

Figure 6.11 Example for illustrating construction of $B_{k,I}$ for $k = 1$ and $I = \{(\alpha, \beta)\}$.



(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 6.4.7. *In Figure 6.11, we have an NFA B with $\mathcal{L}(\boldsymbol{B}) = \{\alpha\beta, \beta\}$, $I = \{(\alpha, \beta)\}$ 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 $\{\alpha\beta, \beta\alpha, \beta\} = \mathrm{Clo}_{k,I}(\mathcal{L}(\boldsymbol{B}))$.*

Proposition 6.4.8. *Given $k \geq 0$, the automaton $B_{k,I}$ accepts at least $\mathrm{Clo}_{k,I}(\mathcal{L}(\boldsymbol{B}))$.*

Proof. The proof is by induction on k. The base case is trivially true, as $\mathcal{L}(B_{0,I}) = \mathcal{L}(B) = \mathrm{Clo}_{0,I}(\mathcal{L}(B))$. The induction case assumes that $B_{k,I}$ accepts at least $\mathrm{Clo}_{k,I}(\mathcal{L}(B))$ and we want to show that $B_{k+1,I}$ accepts at least $\mathrm{Clo}_{k+1,I}(\mathcal{L}(B))$. We take a word $\omega \in \mathrm{Clo}_{k+1,I}(\mathcal{L}(B))$. It must be derived from a word $\omega' \in \mathrm{Clo}_{k,I}(\mathcal{L}(B))$ 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 $\mathrm{Clo}_{k,I}(\mathcal{L}(\mathrm{B}))$. 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 6.4.9. *Given $k \geq 0$, the automaton $B_{k,I}$ accepts at most* $\mathrm{Clo}_I(\mathcal{L}(B))$.

Proof. We need to show that $\omega' \in B_{k,I} \implies \omega' \in \mathrm{Clo}_I(\mathcal{L}(B))$. For this we need to show that ω' is a permutation of a word $\omega \in \mathcal{L}(B)$ 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. □

6.4.5 Language inclusion check algorithm

We develop an algorithm to check language inclusion modulo I (Section 6.4.4) by iteratively increasing the bound k. The algorithm 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 6.1 (ignore Lines 25-29 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 20). If this check fails, the function reports language inclusion failure and returns the counterexample cex (Line 20). If this check succeeds, the successors are computed (Line 23). If a successor satisfies rule R1, it is ignored (Line 24), otherwise it is added to the frontier (Line 31) and the antichain (Line 32). When adding a successor to the frontier the symbol α it appended to the counterexample, denoted as

Algorithm 6.1 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 $\mathcal{L}(A) \subseteq \mathrm{Clo}_I(\mathcal{L}(B))$
$frontier \leftarrow \{(s_A, \{(Q_{\iota,B}, \emptyset, \emptyset)\}, \emptyset) : s_A \in Q_{\iota,A}\}$
No tuple in $frontier$ is dirty
$antichain \leftarrow frontier$
$overflow \leftarrow \emptyset$
$k \leftarrow 2$
while true do
    $cex \leftarrow$ INCLUSION$(k)$
    if $cex \neq$ true $\wedge$ $cex$ is spurious then
        $k \leftarrow k + 1$
        $frontier \leftarrow \{(s_A, S_{B_{k,I}}) \in frontier : S_{B_{k,I}}$ not dirty$\} \cup overflow$
        $antichain \leftarrow \{(s_A, S_{B_{k,I}}) \in antichain : S_{B_{k,I}}$ not dirty$\} \cup overflow$
        $overflow \leftarrow \emptyset$
    else
        return $cex$
    end if
end while

function INCLUSION$(k)$
    while $frontier \neq \emptyset$ do
        remove a tuple $(s_A, S_{B_{k,I}}, cex)$ from $frontier$
        if $s_A \in F_A \wedge (S_{B_{k,I}} \cap F_B) = \emptyset$ then return $cex$
        end if
        for all $\alpha \in \Sigma$ do
            $(s'_A, S'_{B_{k,I}}) \leftarrow \mathsf{succ}_\alpha(s_A, S_{B_{k,I}})$
            if $\nexists p \in antichain : p \sqsubseteq (s'_A, S'_{B_{k,I}})$ then                ▷ Rule R1
                if $\exists (s_B, \eta_1, \eta_2) \in S'_{B_{k,I}} : |\eta_1| > k \vee |\eta_2| > k$ then
                    if $S'_{B_{k,I}}$ not dirty then $overflow \leftarrow overflow \cup \{(s'_A, S'_{B_{k,I}})\}$
                    end if
                    $S'_{B_{k,I}} \leftarrow \{(s_B, \eta_1, \eta_2) \in S'_{B_{k,I}} : |\eta_1| \leq k \wedge |\eta_2| \leq k\}$
                    Mark $S'_{B_{k,I}}$ dirty
                end if
                $frontier \leftarrow frontier \cup \{(s'_A, S'_{B_{k,I}}, cex \cdot \alpha)\}$
                $antichain \leftarrow (antichain \setminus \{p : (s'_A, S'_{B_{k,I}}) \sqsubseteq p\}) \cup \{(s'_A, S'_{B_{k,I}})\}$ ▷ Rule R2
            end if
        end for
    end while
    return true
end function
```

$cex \cdot \alpha$. 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 honours 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 25), we remember the tuple $(s'_A, S'_{B_{k,I}})$ in the set $\mathit{overflow}$ (Line 27). 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 28) and mark $S'_{B_{k,I}}$ as *dirty* (line 28). 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 algorithm spurious if it is not a counterexample to the unbounded language inclusion, formally $cex \in \mathrm{Clo}_I(\mathcal{L}(B))$. 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 sufficently 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 algorithm with k being the length of cex. This is very fast because there is exactly one path through A.

Theorem 6.4.10 (bounded language inclusion check)**.** *The procedure* INCLUSION *of Algorithm 6.1 decides $\mathcal{L}(A) \subseteq \mathcal{L}(B_{k,I})$ 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 [de Wulf *et al.*, 2006] 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 23). The bound k is enforced separately in line 25. □

Theorem 6.4.11 (preemption-safety problem)**.** *If program $\mathcal{C}$ is not preemption-safe ($[\![\mathcal{C}]\!]^P \not\subseteq [\![\mathcal{C}]\!]^{NP}$), then Algorithm 6.1 will return* `false`.

Proof. By Theorem 6.4.1 we know $[\![\mathcal{C}_{abs}]\!]^P \not\subseteq_{abs} [\![\mathcal{C}_{abs}]\!]^{NP}$. From Proposition 6.4.3 we get $\mathcal{L}(\mathsf{P}_{abs}) \not\subseteq \mathrm{Clo}_{I_D}(\mathcal{L}(\mathsf{NP}_{abs}))$. From Proposition 6.4.9 we know that for any k this is equivalent to $\mathcal{L}(\mathsf{P}_{abs}) \not\subseteq \mathcal{L}(B_{k,I})$, where $B = \mathsf{NP}_{abs}$. Theorem 6.4.10 shows that Algorithm 6.1 decides this for any bound k. □

6.5 Finding and Enforcing Mutex Constraints

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}.

6.5.1 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 trace π: $tid_0.\ell_0; \ldots; tid_n.\ell_n$, where each ℓ_i is a location identifier from $\mathcal{C}_{abs}$.

Recall the definition of good, bad neighbourhoods and HB-formulæ from Section 2.2. In our setting good traces are those that are equivalent to a non-preemptive trace and all other feasible traces are bad.

Non-preemptive neighbourhood. 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: $\varphi_G = \bigvee_{\pi \in \mathrm{non-preemptive}} \Phi(\pi)$.

$\Phi(\pi)$ enforces the order between conflicting accesses in the abstract trace π:

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

Algorithm 6.2 Counterexample enumeration and generalisation algorithm

```
Require: Trace $\pi$, formula of good traces $\varphi_G$ in $nhood(\pi)$
Ensure: HB-formula of bad traces $\varphi_B$
$\Psi \leftarrow$ quantifier-free first-order formula representing all feasible traces in $nhood(\pi)$
$\Psi_B \leftarrow \Psi \wedge \neg\varphi_G$
$\varphi_B \leftarrow$ false
while $\Psi_B \wedge \neg\varphi_B$ is satisfiable do
    $\psi \leftarrow$ satisfying assignment for $\Psi_B \wedge \neg\varphi_B$
    $\sigma \leftarrow$ trace represented by $\psi$
                                                        ▷ Conflicting access analysis
    $\varphi_{B'} \leftarrow \Phi(\sigma)$
                                                        ▷ Unsat-core computation
    $\varphi_{B''} \leftarrow MinUNSATCore(Soft \leftarrow \varphi_{B'},$
            $Hard \leftarrow \Psi \wedge \varphi_G)$
    $\varphi_B \leftarrow \varphi_B \vee \varphi_{B''}$
end while
return $\varphi_B$
```

Example. Recall the counterexample trace from the running example in Section 6.1.1: $cex =$ $\mathtt{T1}.\ell_{1a}$; $\mathtt{T1}.\ell_{1b}$; $\mathtt{T2}.\ell_{1a}$; $\mathtt{T2}.\ell_{1b}$; $\mathtt{T1}.\ell_2$; $\mathtt{T2}.\ell_2$; $\mathtt{T2}.\ell_{3a}$; $\mathtt{T2}.\ell_{3b}$; $\mathtt{T2}.\ell_4$; $\mathtt{T1}.\ell_{3a}$; $\mathtt{T1}.\ell_{3b}$; $\mathtt{T1}.\ell_4$. There are two feasible traces in $\mathcal{N}^g_\pi$:

- $\pi_1 = \mathtt{T1}.\ell_{1a}$; $\mathtt{T1}.\ell_{1b}$; $\mathtt{T1}.\ell_2$; $\mathtt{T1}.\ell_{3a}$; $\mathtt{T1}.\ell_{3b}$; $\mathtt{T1}.\ell_4$; $\mathtt{T2}.\ell_{1a}$; $\mathtt{T2}.\ell_{1b}$; $\mathtt{T2}.\ell_2$; $\mathtt{T2}.\ell_{3a}$; $\mathtt{T2}.\ell_{3b}$; $\mathtt{T2}.\ell_4$ 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 $\Theta = \Phi(\pi_1) \vee \Phi(\pi_2)$.

Counterexample enumeration and generalisation. 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 synchronisation, 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 generalisation 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 generalising $\varphi_{B'}$ 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 6.2 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 generalisation 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 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.

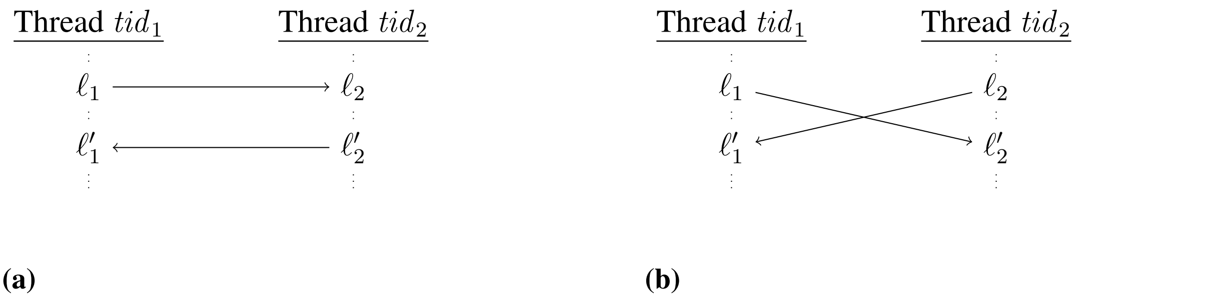
Example. Our trace π from Section 6.1.1 is generalised to $G(\Phi(\pi)) = \mathtt{T2}.\ell_{1a} < \mathtt{T1}.\ell_{3b} \wedge \mathtt{T1}.\ell_{3b} < \mathtt{T2}.\ell_{3b}$. This constraint captures the interleavings where T2 interrupts T1 between locations ℓ_{1a} and ℓ_{3b}. Any trace that fulfils this constraint is bad. All bad traces in $\mathcal{N}_\pi$ are represented as $P = (\mathtt{T2}.\ell_{1a} < \mathtt{T1}.\ell_{3b} \wedge \mathtt{T1}.\ell_{3b} < \mathtt{T2}.\ell_{3b}) \vee (\mathtt{T1}.\ell_{1a} < \mathtt{T2}.\ell_{3b} \wedge \mathtt{T2}.\ell_{3b} < \mathtt{T1}.\ell_{3b})$.

Inferring mutex constraints. The constraint inference uses the same patterns as are used to infer locks in Chapter 5. From each clause ρ in P described above, we infer mutex constraints to eliminate all bad traces satisfying ρ. The key observation we exploit is that atomicity violations show up in our formulæ as two simple patterns of ordering constraints between events.

1. The first pattern $tid_1.\ell_1 < tid_2.\ell_2 \ \wedge \ tid_2.\ell'_2 < tid_1.\ell'_1$ (visualised in Figure 6.12a) indicates an atomicity violation (thread tid_2 interrupts tid_1 at a critical moment).
2. The second pattern is $tid_1.\ell_1 < tid_2.\ell'_2 \ \wedge \ tid_2.\ell_2 < tid_1.\ell'_1$ (visualised in Figure 6.12b). This pattern is a generalisation 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 $\mathrm{mtx}(tid_1.[\ell_1 : \ell'_1], tid_2.[\ell_2 : \ell'_2])$.

Figure 6.12 Atomicity violation patterns



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

6.5.2 Enforcing Mutex Constraints

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

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) ℓ_i^{pre}, ℓ_i^{mid}, ℓ_i^{post} are adjacent locations in thread tid_i,

(b) ℓ_j^{cpre}, ℓ_j^{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 pre$(c) = \ell_i^{\text{pre}}$, mid$(c) = \ell_i^{\text{mid}}$, post$(c) = \ell_i^{\text{post}}$, cpre$(c) = \ell_j^{\text{cpre}}$ and 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 Figure 6.13 (we skip the statement labels in the nodes here). Suppose in some iteration we obtain $\text{mtx}(\mathsf{T1}.[\ell_{a1} : \ell_{a2}], \mathsf{T2}.[\ell_{b1} : \ell_{b4}])$. This yields 2 conflicts: c_1 given by $(\ell_{b1}, \ell_{b2}, \ell_{b3}, \ell_{a1}, \ell_{a2})$ and c_2 given by $(\ell_{b2}, \ell_{b3}, \ell_{b4}, \ell_{a1}, \ell_{a2})$.

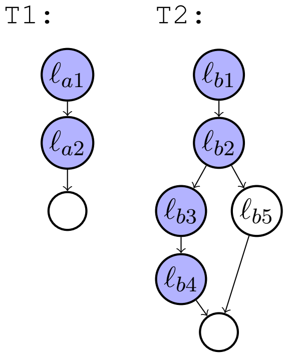
Figure 6.13 Example: Mutex constraints and conflicts

Thread T1

```
$\ell_{a1}$: write(v)
$\ell_{a2}$: read(x)
```

Thread T2

```
$\ell_{b1}$: read(v)
$\ell_{b2}$: if (*) then
$\ell_{b3}$:     write(v)
$\ell_{b4}$:     read(x)
         else
$\ell_{b5}$:     read(x)
         end if
```



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 $\texttt{lock}(\mathsf{T1}.[\ell_1 : \ell_2], \mathsf{T2}.[\ell_1 : \ell_4])$. This is not a valid lock placement; in executions executing the `else` branch, the lock is never released.

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 $\langle \mathcal{V}_o, \mathit{ctid}, (\ell_1, \ldots, \ell_n) \rangle$ of the abstract program $\mathcal{C}_{abs}$ corresponding to $\mathcal{C}$, as well as $\langle \texttt{terminated} \rangle$ and $\langle \texttt{invalid} \rangle$,

(b) Σ is the set of abstract observable symbols,

(c) $Q_{\iota,\text{old}}$ is the initial state of $\mathcal{C}_{abs}$,

(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.

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 Figure 6.13, conflict c_1 is activated when T2 moves from ℓ_{b1} to ℓ_{b2}; c_1 progresses if now T1 moves from ℓ_{a1} to ℓ_{a2} and is aborted if instead T2 moves from ℓ_{b2} to ℓ_{b3}; c_1 completes after progressing if T2 moves from ℓ_{b2} to ℓ_{b3} and is aborted if instead T2 moves from ℓ_{b2} to ℓ_{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) $Q_{\iota,\text{new}} = (Q_{\iota,\text{old}}, (0,\dots,0))$,

(d) $F_{\text{new}} = \{(Q,(p_1,\dots,p_K)) : Q \in F_{\text{old}} \wedge p_1,\dots,p_K \in \{0,1,2\}\}$ and

(e) Δ_{new} is constructed as follows:

add $((Q,(p_1,\dots,p_K)),\alpha,(Q',(p'_1,\dots,p'_K)))$ 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 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 $p'_k = 1$,

2. *Conflict progress:* (thread $tid_1(c_k)$ is interrupted by $tid_2(c_k)$ and the conflicting statement at location cpre(c_k) is executed)

   else if $p_k = 1$, $ctid = ctid' = tid_2(c_k)$, $\ell_{ctid} = \text{cpre}(c_k)$ and $\ell'_{ctid} = \text{cpost}(c_k)$, then $p'_k = 2$,

3. *Conflict completion and state pruning:* (the statement at location 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 $(Q',(p'_1,\dots,p'_K))$,

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 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 $\ell'_{ctid} = \text{post}(c_k)$, 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 $p_1,\dots,p_K$ 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 6.5.1. *We are given a program $\mathcal{C}_{abs}$ and a sequence of observable symbols ω that is a counterexample to preemption-safety, formally $\omega \in \mathcal{L}(\mathsf{P}'_{abs}) \wedge \omega \notin \text{Clo}_I(\mathcal{L}(\mathsf{NP}_{abs}))$. 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* $\omega \notin \mathcal{L}(\mathsf{P}'_{abs})$*.*

Proof. The pattern P eliminating ω represents a mutex constraint $\mathrm{mtx}(\mathit{tid}_i.[\ell_1{:}\ell''_1], \mathit{tid}_j.[\ell_2{:}\ell''_2])$, 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: $\ell_i^{\text{pre}} \to \ell_i^{\text{mid}} \to \ell_j^{\text{cpre}} \to \ell_j^{\text{cpost}} \to \ell_i^{\text{mid}} \to \ell_i^{\text{post}}$. Because P eliminates ω we know that ω has a context switch from $\mathit{tid}_i.\ell'_1$ to $\mathit{tid}_j.\ell'_2$, where $\ell_1 \le \ell'_1 \le \ell''_1$ and $\ell_2 \le \ell'_2 \le \ell''_2$. 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}. □

6.6 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[2] 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 $\mathit{LkVar}, \mathit{LkVar}', \mathit{LkVar}_1, \ldots$ 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 thread $\mathit{tid}(\ell)$ in $\mathcal{C}$. We use the following Boolean variables in the encoding.

[2] 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 optimisation (see Section 6.7).

$\mathsf{LockBefore}(\ell, \mathit{LkVar})$	$\mathsf{lock}(\mathit{LkVar})$ 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 statements 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.

$$\mathsf{InLock}(\ell, \mathit{LkVar}) = \mathsf{LockBefore}(\ell, \mathit{LkVar}) \lor \; (\neg\mathsf{UnlockBefore}(\ell, \mathit{LkVar}) \land \\ \mathsf{InLockEnd}(\ell', \mathit{LkVar}))$$

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

$$(\mathsf{InLock}(\ell, \mathit{LkVar}) \land \neg\mathsf{UnlockAfter}(\ell, \mathit{LkVar})) \lor \mathsf{LockAfter}(\ell, \mathit{LkVar})$$

(*D3*) $\mathsf{Order}(\mathit{LkVar}, \mathit{LkVar}')$: We give a fixed lock order that is transitive, asymmetric, and irreflexive.

$\mathsf{Order}(\mathit{LkVar}, \mathit{LkVar}') = \texttt{true}$ 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

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

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

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

(*C4*) Existing locks may not be nested inside synthesised locks. They are implicitly ordered before synthesised 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 synthesised 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

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

(*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 synthesise a lock across an existing lock or wait statement to ensure preemption-safety.

We have the following result.

Theorem 6.6.1. *Let concurrent program $\mathcal{C}'$ be obtained by inserting any lock placement satisfying* LkCons *into concurrent program $\mathcal{C}$. Then $\mathcal{C}'$ is guaranteed to be preemption-safe w.r.t. $\mathcal{C}$ and not to introduce new deadlocks (that were not already present in $\mathcal{C}$).*

Proof. To show preemption-safety we need to show that language inclusion holds (Proposition 6.4.3). 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 $s = \langle \mathcal{V}, ctid, (\ell_1, \ldots, \ell_n) \rangle$ is reachable in $\mathcal{C}'$. By definition this means that none of the rules of the preemptive semantics of $\mathcal{W}$ (Figures 2.3 and 2.4) is 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 (Figure 2.2) 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 synthesised 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 $\mathsf{signal}(CondVar)/\mathsf{reset}(CondVar)$. 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.

□

6.7 Optimising Lock Placement

The global lock placement constraint LkCons constructed in Section 6.6 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 algorithm 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.

Objective functions. We say that a statement $\ell : \mathit{stmt}$ in a concurrent program $\mathcal{C}$ 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 $\mathcal{C}_1$ over $\mathcal{C}_2$ if the number of lock statements in $\mathcal{C}_1$ is fewer than in $\mathcal{C}_2$. 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 $\mathsf{Coarse}(\mathcal{C}_i)$ to be $\lambda + \epsilon \cdot \mathsf{StmtInLock}(\mathcal{C}_i)$ where λ is the count of `lock` statements in $\mathcal{C}_i$, $\mathsf{StmtInLock}(\mathcal{C}_i)$ is the count of statements in $\mathcal{C}_i$ that are protected by any lock and ϵ is given by $\frac{1}{2k}$ where k is the total number of statements in $\mathcal{C}_i$. 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 $\mathcal{C}_1$ over $\mathcal{C}_2$ if $\mathcal{C}_1$ allows more concurrency than $\mathcal{C}_2$. Concurrency of a program is measured by the number of pairs of statements from different threads that cannot be executed together. Formally, we define $\mathsf{Fine}(\mathcal{C}_i)$ 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.

Optimisation procedure. The main idea behind the optimisation 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 formulæ 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 minimising the cost we maximise 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 6.7.1. *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.*

6.8 Implementation and Experiments

In order to evaluate our synthesis algorithm, 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[3]. The language inclusion algorithm is available separately as a library called LIMI[4]. LISS implements the synthesis method presented in this chapter with several optimisations. 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 algorithm; (3) Precision of the proposed coarse abstraction scheme on real-world programs; (4) Quality of the locks placed.

6.8.1 Benchmarks

We ran LISS on a number of benchmarks, summarised in Table 6.1. 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 (Figure 6.5) 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

[3] `https://github.com/thorstent/Liss`

[4] `https://github.com/thorstent/Limi`

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 8GB of RAM.

Implicit vs 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 (Chapter 4). In addition, we evaluate LISS and CONREPAIR on several *new* assertion-based benchmarks (Table 6.1). 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.

The set includes synthetic microbenchmarks modelling typical concurrency bug patterns in Linux drivers and the `usb-serial` macrobenchmark, which models a complete synchronisation 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 synthesise 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 race conditions that manifested under non-preemptive scheduling; LISS did not detect these race conditions. LISS was able to detect and fix all other known races without relying on assertions. Furthermore, LISS detected a new race in the `usb-serial` family of benchmarks, which was not detected by CONREPAIR due to a missing assertion. We compared the output of LISS (using coarse-grained locking as an objective function) with manually placed synchronisation (taken from real bug fixes) and found that the two versions were similar in most of our examples.

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

The loss of precision due to abstraction may cause the inclusion check to return a counter-

Table 6.1 Experiments

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

Th=Threads, It=Iterations, MB=Max bound, TT=Time for language incl. loop, CR=ConRepair time

[a] bug-free example

[b] timeout after three hours

[c] race not detected, as it was present under non-preemptive scheduling

Table 6.2 Lock placement statistics: the number of synthesised lock variables, lock and unlock statements, and the number of abstract statements protected by locks for different objective functions.

	No objective			Coarse			Fine		
Name	locks	locks/ un- locks	protected instr	locks	locks/ un- locks	protected instr	locks	locks/ un- locks	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

example that is spurious in the concrete program, leading to unnecessary synchronisation being synthesised. 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.

Real-world benchmarks. While these results are encouraging, synthetic benchmarks are not necessarily representative of real-world performance.

CPMAC benchmark. We therefore implemented another set of benchmarks based on a complete Linux driver for the TI AR7 CPMAC Ethernet controller. The benchmark was constructed as follows. We manually preprocessed driver source code to eliminate pointer aliasing. We combined the driver with a model of the OS API and the software interface of the device written in C. We modelled most OS API functions as writes to a special memory location. Groups of unrelated functions were modelled using separate locations. Slightly more complex models were required for API functions that affect thread synchronisation. 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 behaviour to avoid races. We introduced a condition variable to model this synchronisation. Similarly, most device accesses were modelled 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 behaviour 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 synchronisation synthesis case study, based on real-world code, reported in the literature.

The CPMAC driver used in this case study did not contain any known concurrency bugs, so we artificially simulated five typical race conditions that commonly occur in drivers of this type (see Section 3.5.1). LISS was able to detect and automatically fix each of these defects (bottom part of Table 6.1). The coarse abstraction may lead to unnecessary synchronisation, which may be solved by abstraction refinement using guard variables. We only encountered two program locations where manual abstraction refinement was necessary. This process could be automated; automatic abstraction refinement is a known technique [Vechev *et al.*, 2010a] we could implement in LISS.

Memcached benchmark. Finally, we evaluate LISS on *memcached*, an in-memory key-value store version 1.4.5 [memcached]. 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 synthesise lock placement for a fragment of the `thread.c` module. In contrast to our other case studies, here we would like to synthesise 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 synthesise both a coarse-grained and a fine-grained version and at deployment the appropriate version is selected.

Quality of synthesis. Next, we focus on the quality of synthesised solutions for the two real-world benchmarks from our benchmark set. Table 6.2 compares the implementation synthesised for these benchmarks using each objective functions in terms of (1) the number of locks used in synthesised code, (2) the number of lock and unlock statements generated, and (3) total number of program statements protected by synthesised locks.

We observe that different objective functions produce significantly different results in terms of the size of synthesised critical sections and the number of lock and unlock operations guarding them: the fine-grained objective synthesises smaller critical sections at the cost of introducing a larger number of lock and unlock operations. Implementations synthesised without an objective function are clearly of lower quality than the optimised 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 synchronisation already in place, as well as the memcached benchmark, where we synthesise synchronisation from scratch.

To summarise our experiments, we found that (1) our coarse abstraction is highly precise in practice; (2) manual effort involved in synchronisation synthesis can be further reduced via automatic abstraction refinement; (3) additional work is required to improve the performance of our method to be able to handle real-world systems without simplification; (4) the objective functions allow specialising synthesis to a particular locking scheme; (5) the time required to solve the MaxSMT problem is small compared to the analysis time.

Chapter 7

Conclusion

In this thesis we introduced a number of synchronisation techniques for concurrent programs. We started with a simple technique that can infer statement reorderings and atomic sections for concurrent programs with assertions (Chapters 3 and 4). Next, we improved the synthesis by considering additional synchronisation primitives: Instead of atomic sections, which are not directly implementable, we now place locks, barriers and wait-signal statements. Finally, we moved from an explicit specification to 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 optimised using a static objective function.

We developed a number of tools for this thesis. The most mature tool is LISS, which can parse a significant framework of the C programming language and automatically insert missing locks so that the program is correct w.r.t. our implicit specification. A number of key elements are missing to enable LISS to process real-world programs: For example to process pointers we would need an aliasing analysis. An abstraction refinement would be needed to increase precision of the synthesis. Further, for real-world programs there could be performance issues because the automata in the language inclusion step would grow too large.

An this point we have a number of research directions we can pursue. In ongoing work [Černý *et al.*, 2015a] we aim to optimise lock placement not merely using syntactic criteria, but by optimising the actual performance of the program running on a specific system. In this approach we start with a synthesised 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.

Another direction are weak memory models. In this thesis we assumed a sequentially consist memory model, which is an idealised model not found in computers today. To deal with weak memory models we could adapt our techniques to synthesise fences.

Bibliography

[Albarghouthi and McMillan, 2013] A. Albarghouthi and K. McMillan, “Beautiful Interpolants,” In *CAV*, pages 313–329, 2013.

[Alglave *et al.*, 2014] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl, “Don’t Sit on the Fence — A Static Analysis Approach to Automatic Fence Insertion,” In *CAV*, pages 508–524, 2014.

[Alglave *et al.*, 2013] J. Alglave, D. Kroening, and M. Tautschnig, “Partial Orders for Efficient Bounded Model Checking of Concurrent Software,” In *CAV*, pages 141–157, 2013.

[Alur *et al.*, 2013] R. Alur, R. Bodík, G. Juniwal, M. Martin, M. Raghothaman, S. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, “Syntax-guided synthesis,” In *FMCAD*, pages 1–17, 2013.

[Ball *et al.*, 2001] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, “Automatic predicate abstraction of C programs,” In *PLDI*, pages 203–213, 2001.

[Basu *et al.*, 2003] S. Basu, D. Saha, Y. Lin, and S. Smolka, “Generation of all counter-examples for push-down systems,” In *FORTE*, pages 79–94, 2003.

[Bertoni *et al.*, 1982] A. Bertoni, G. Mauri, and N. Sabadini, “Equivalence and membership problems for regular trace languages,” In *Automata, Languages and Programming*, pages 61–71, 1982.

[Beyer, 2014] D. Beyer, “Status Report on Software Verification,” In *TACAS*, pages 373–388, 2014, `http://sv-comp.sosy-lab.org/`.

[Beyer *et al.*, 2007] D. Beyer, T. Henzinger, R. Majumdar, and A. Rybalchenko, “Path invariants,” In *PLDI*, pages 300–309, 2007.

[Beyer and Keremoglu, 2011] D. Beyer and E. Keremoglu, “CPAchecker: A Tool for Configurable Software Verification,” In *CAV*, pages 184–190, 2011, `http://cpachecker.sosy-lab.org/`.

[Biere *et al.*, 2003] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, “Bounded model checking,” *Advances in computers*, 58:117–148, 2003.

[Bjesse and Kukula, 2004] P. Bjesse and J. Kukula, “Using counter example guided abstraction refinement to find complex bugs,” In *DATE*, pages 156–161, 2004.

[Bjørner *et al.*, 2013] N. Bjørner, K. McMillan, and A. Rybalchenko, “On Solving Universally Quantified Horn Clauses,” In *SAS*, pages 105–125, 2013.

[Bloem *et al.*, 2014] R. Bloem, G. Hofferek, B. Könighofer, R. Könighofer, S. Außerlechner, and R. Spörk, “Synthesis of Synchronization using Uninterpreted Functions,” In *FMCAD*, pages 35–42, 2014.

[Bryant, 1986] R. Bryant, “Graph-Based Algorithms for Boolean Function Manipulation,” *Computers, IEEE Transactions on*, C-35(8):677–691, 1986.

[Cadiou and Levy, 1973] J. M. Cadiou and J.-J. Levy, “Mechanizable proofs about parallel processes,” In *Switching and Automata Theory*, pages 34–48, 1973.

[Černý *et al.*, 2015a] P. Černý, E. M. Clarke, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, R. Samanta, and T. Tarrach, “Optimizing Solution Quality in Synchronization Synthesis,” *ArXiv e-prints*, November 2015, arXiv:1511.07163.

[Černý *et al.*, 2015b] P. Černý, E. M. Clarke, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, R. Samanta, and T. Tarrach, “From Non-preemptive to Preemptive Scheduling Using Synchronization Synthesis,” In *CAV*, pages 180–197, 2015.

[Černý *et al.*, 2013] P. Černý, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach, “Efficient Synthesis for Concurrency by Semantics-Preserving Transformations,” In *CAV*, pages 951–967, 2013.

[Černý *et al.*, 2014] P. Černý, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach, “Regression-Free Synthesis for Concurrency,” In *CAV*, pages 568–584, 2014.

[Chatterjee and Henzinger, 2007] K. Chatterjee and T. A. Henzinger, "Assume-guarantee synthesis," In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 261–275, 2007.

[Cherem *et al.*, 2008] S. Cherem, T. Chilimbi, and S. Gulwani, "Inferring locks for atomic sections," In *PLDI*, pages 304–315, 2008.

[Chou *et al.*, 2001] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating Systems Errors," In *SOSP*, pages 73–88, 2001.

[Church, 1962] A. Church, "Logic, arithmetic and automata," In *Proceedings of the international congress of mathematicians*, pages 23–35, 1962.

[Clarke *et al.*, 1986] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[Clarke *et al.*, 2000] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-Guided Abstraction Refinement," In *CAV*, pages 154–169, 2000.

[Clarke *et al.*, 2004] E. M. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," In *TACAS*, pages 168–176, 2004, `http://www.cprover.org/cbmc/`.

[Clarke *et al.*, 2005] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-based predicate abstraction for ANSI-C," In *TACAS*, pages 570–574, 2005.

[Clarke and Emerson, 1982] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," In *Logics of Programs*, pages 52–71, 1982.

[Clarke *et al.*, 1983] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach," In *POPL*, pages 117–126, 1983.

[Cousot and Cousot, 1977] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," In *POPL*, pages 238–252, 1977.

[de Moura and Bjørner, 2008] L. de Moura and N. Bjørner, “Z3: An efficient SMT solver,” In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[de Wulf *et al.*, 2006] M. de Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin, “Antichains: A new algorithm for checking universality of finite automata,” In *CAV*, pages 17–30, 2006.

[Deshmukh *et al.*, 2010] J. Deshmukh, G. Ramalingam, V. Ranganath, and K. Vaswani, “Logical Concurrency Control from Sequential Proofs,” In *Programming Languages and Systems*, pages 226–245, 2010.

[Dijkstra, 1968] E. W. Dijkstra, “Co-operating Sequential Processes,” *Programming Languages. Academic Press, New York*, 1968.

[Donaldson *et al.*, 2011] A. Donaldson, A. Kaiser, D. Kroening, and T. Wahl, “Symmetry-Aware Predicate Abstraction for Shared-Variable Concurrent Programs,” In *CAV*, pages 356–371, 2011.

[Elmas *et al.*, 2009] T. Elmas, S. Qadeer, and S. Tasiran, “A calculus of atomic actions,” In *POPL*, pages 2–15, 2009.

[Engler and Ashcraft, 2003] D. Engler and K. Ashcraft, “RacerX: effective, static detection of race conditions and deadlocks,” In *SOSP*, pages 237–252, 2003.

[Ermis *et al.*, 2012] E. Ermis, M. Schäf, and T. Wies, “Error Invariants,” In *FM*, pages 187–201, 2012.

[Esparza, 1997] J. Esparza, “Decidability of model checking for infinite-state concurrent systems,” *Acta Informatica*, 34(2):85–107, 1997.

[Eswaran *et al.*, 1976] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, “The notions of consistency and predicate locks in a database system,” *Communications of the ACM*, 19(11):624–633, 1976.

[Farchi *et al.*, 2003] E. Farchi, Y. Nir, and S. Ur, “Concurrent bug patterns and how to test them,” In *PDPS*, page 7 pp., 2003.

[Farzan *et al.*, 2013a] A. Farzan, A. Holzer, N. Razavi, and H. Veith, “Con2colic testing,” In *FSE*, pages 37–47, 2013.

[Farzan *et al.*, 2013b] A. Farzan, Z. Kincaid, and A. Podelski, "Inductive data flow graphs," In *POPL*, pages 129–142, 2013.

[Flanagan and Qadeer, 2003] C. Flanagan and S. Qadeer, "Types for atomicity," In *ACM SIGPLAN Notices*, pages 1–12, 2003.

[Floyd, 1967] R. W. Floyd, "Assigning meanings to programs," *Mathematical aspects of computer science*, 19(19-32):1, 1967.

[Glusman *et al.*, 2003] M. Glusman, G. Kamhi, S. Mador-Haim, R. Fraer, and M. Vardi, "Multiple-counterexample guided iterative abstraction refinement: An industrial evaluation," In *TACAS*, pages 176–191, 2003.

[Godefroid *et al.*, 1996] P. Godefroid, D. Peled, and M. Staskauskas, "Using partial-order methods in the formal validation of industrial concurrent programs," In *ISSTA*, pages 261–269, 1996.

[Graf and Saidi, 1997] S. Graf and H. Saidi, "Construction of abstract state graphs with PVS," In *Computer Aided Verification*, pages 72–83, 1997.

[Green, 1969] C. Green, "Application of theorem proving to problem solving," Technical report, DTIC Document, 1969.

[Griesmayer *et al.*, 2006] A. Griesmayer, R. Bloem, and B. Cook, "Repair of Boolean Programs with an Application to C," In *CAV*, pages 358–371, 2006.

[Gupta *et al.*, 2011a] A. Gupta, C. Popeea, and A. Rybalchenko, "Solving Recursion-Free Horn Clauses over LI+UIF," In *APLAS*, pages 188–203, 2011.

[Gupta *et al.*, 2011b] A. Gupta, C. Popeea, and A. Rybalchenko, "Threader: A Constraint-Based Verifier for Multi-threaded Programs," In *CAV*, pages 412–417, 2011.

[Gupta *et al.*, 2015] A. Gupta, T. A. Henzinger, A. Radhakrishna, R. Samanta, and T. Tarrach, "Succinct Representation of Concurrent Trace Sets," In *POPL*, pages 433–444, 2015.

[Henzinger *et al.*, 2004] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," In *POPL*, pages 232–244, 2004.

[Henzinger *et al.*, 2002] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," In *POPL*, pages 58–70, 2002.

[Henzinger *et al.*, 2000] T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "Decomposing refinement proofs using assume-guarantee reasoning," In *ICCAD*, pages 245–253, 2000.

[Herlihy and Wing, 1990] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," In *TOPLAS*, pages 463–492, 1990.

[Hoare, 1969] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, 12(10):576–580, 1969.

[Hoare, 1978] C. A. R. Hoare, "Communicating Sequential Processes," *Commun. ACM*, 21(8):666–677, August 1978.

[Jalbert and Sen, 2010] N. Jalbert and K. Sen, "A trace simplification technique for effective debugging of concurrent programs," In *FSE*, pages 57–66, 2010.

[Jin *et al.*, 2012] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, "Automated Concurrency-Bug Fixing," In *OSDI*, pages 221–236, 2012.

[Jobstmann *et al.*, 2005] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program Repair as a Game," In *CAV*, pages 226–238, 2005.

[Jones, 1983] C. B. Jones, "Specification and Design of (Parallel) Programs.," In *IFIP congress*, pages 321–332, 1983.

[Kahlon and Wang, 2010] V. Kahlon and C. Wang, "Universal causality graphs: a precise happens-before model for detecting bugs in concurrent programs," In *CAV*, pages 434–449, 2010.

[Kashyap and Garg, 2008] S. Kashyap and V. Garg, "Producing Short Counterexamples Using "Crucial Events"," In *CAV*, pages 491–503, 2008.

[Khoshnood *et al.*, 2015] S. Khoshnood, M. Kusano, and C. Wang, "ConcBugAssist: Constraint solving for diagnosis and repair of concurrency bugs," In *International Symposium on Software Testing and Analysis*, 2015.

[Kripke, 1963] S. Kripke, “Semantical Considerations on Modal Logic,” *Acta Philosophica Fennica*, pages 83–94, 1963.

[Lamport, 1979] L. Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs,” *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[Lamport, 1980] L. Lamport, ““Sometime is sometimes” not “never” – On the temporal logic of programs,” In *POPL*, pages 174–185, 1980.

[Lipton, 1975] R. J. Lipton, “Reduction: a new method of proving properties of systems of processes,” In *POPL*, pages 78–86, 1975.

[Lu *et al.*, 2008] S. Lu, S. Park, E. Seo, and Y. Zhou, “Learning from mistakes: a comprehensive study on real world concurrency bug characteristics,” In *ASPLOS*, pages 329–339, 2008.

[Manna and Wolper, 1984] Z. Manna and P. Wolper, “Synthesis of Communicating Processes from Temporal Logic Specifications,” In *TOPLAS*, pages 68–93, 1984.

[Mazurkiewicz, 1987] A. Mazurkiewicz, “Trace theory,” In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 278–324, 1987.

[McCarthy, 1960] J. McCarthy, “Recursive functions of symbolic expressions and their computation by machine, Part I,” *Communications of the ACM*, 3(4):184–195, 1960.

[McCarthy, 1962] J. McCarthy, “Towards a mathematical science of computation,” In *IFIP Congress*, pages 21–28, 1962.

[McMillan, 1993] K. L. McMillan, *Symbolic Model Checking*, Springer US, 1993.

[memcached] “Memcached distributed memory object caching system,” http://memcached.org.

[Milner, 1980] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science. Springer-Verlag, 1980.

[Milner, 1992] R. Milner, “Functions as processes,” *Mathematical Structures in Computer Science*, 2:119–141, 6 1992.

[Morse *et al.*, 2014] J. Morse, M. Ramalho, L. Cordeiro, D. Nicole, and B. Fischer, “ESBMC 1.22,” In *TACAS*, pages 405–407, 2014, `http://www.esbmc.org/`.

[Musuvathi and Qadeer, 2007] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," In *PLDI*, pages 446–455, 2007.

[Owicki and Gries, 1976] S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs I," *Acta Informatica*, 6(4):319–340, 1976.

[Papadimitriou, 1986] C. Papadimitriou, *The theory of database concurrency control*, Computer Science Press Inc., Rockville, MD, 1986.

[Petri, 1962] C. A. Petri, *Kommunikation mit Automaten*, PhD thesis, Universität Hamburg, 1962.

[Pnueli and Rosner, 1989] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," In *POPL*, pages 179–190, 1989.

[Pnueli, 1977] A. Pnueli, "The temporal logic of programs," In *Foundations of Computer Science, 18th Annual Symposium on*, pages 46–57, 1977.

[Q] "Q Program Verifier," `http://research.microsoft.com/en-us/projects/verifierq/`.

[Qadeer and Rehof, 2005] S. Qadeer and J. Rehof, "Context-bounded model checking of concurrent software," In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, 2005.

[Queille and Sifakis, 1982] J. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," In *International Symposium on Programming*, pages 337–351, 1982.

[Rabin, 1972] M. O. Rabin, "Automata on infinite objects and Church's problem," 1972.

[Ramadge and Wonham, 1987] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes," *SIAM journal on control and optimization*, 25(1):206–230, 1987.

[Ryzhyk *et al.*, 2009] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser, "Dingo: Taming Device Drivers," In *Eurosys*, 2009.

[Sadowski and Yi, 2010] C. Sadowski and J. Yi, “User Evaluation of Correctness Conditions: A Case Study of Cooperability,” In *PLATEAU*, pages 2:1–2:6, 2010.

[Said *et al.*, 2011] M. Said, C. Wang, Z. Yang, and K. Sakallah, “Generating data race witnesses by an SMT-based analysis,” In *NASA Formal Methods*, pages 313–327, 2011.

[Sakunkonchak *et al.*, 2007] T. Sakunkonchak, S. Komatsu, and M. Fujita, “Using counter-example analysis to minimize the number of predicates for predicate abstraction,” In *ATVA*, pages 553–563, 2007.

[Samanta *et al.*, 2008] R. Samanta, J. Deshmukh, and A. Emerson, “Automatic Generation of Local Repairs for Boolean Programs,” In *FMCAD*, pages 1–10, 2008.

[Savage *et al.*, 1997] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: A dynamic data race detector for multithreaded programs,” *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[Schnoebelen, 2002] P. Schnoebelen, “The Complexity of Temporal Logic Model Checking,” *Advances in modal logic*, 4(393-436):35, 2002.

[Sen, 2008] K. Sen, “Race Directed Random Testing of Concurrent Programs,” In *PLDI*, 2008.

[Sharma *et al.*, 2012] R. Sharma, A. V. Nori, and A. Aiken, “Interpolants as Classifiers,” In *CAV*, pages 71–87, 2012.

[Sinha and Wang, 2010] N. Sinha and C. Wang, “Staged concurrent program analysis,” In *FSE*, 2010.

[Sinha and Wang, 2011] N. Sinha and C. Wang, “On Interference Abstractions,” In *POPL*, 2011.

[Solar-Lezama *et al.*, 2008] A. Solar-Lezama, C. Jones, and R. Bodík, “Sketching concurrent data structures,” In *PLDI*, pages 136–148, 2008.

[Solar-Lezama *et al.*, 2006] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, “Combinatorial sketching for finite programs,” In *ASPLOS*, pages 404–415, 2006.

[Vechev and Yahav, 2008] M. Vechev and E. Yahav, “Deriving linearizable fine-grained concurrent objects,” In *PLDI*, pages 125–135, 2008.

[Vechev *et al.*, 2010a] M. Vechev, E. Yahav, and G. Yorsh, "Abstraction-guided synthesis of synchronization," In *POPL*, pages 327–338, 2010.

[Vechev *et al.*, 2009] M. T. Vechev, E. Yahav, and G. Yorsh, "Inferring Synchronization under Limited Observability," In *TACAS*, pages 139–154, 2009.

[Vechev *et al.*, 2010b] M. T. Vechev, E. Yahav, R. Raman, and V. Sarkar, "Automatic Verification of Determinism for Structured Parallel Programs," In *SAS*, pages 455–471, 2010.

[von Essen and Jobstmann, 2013] C. von Essen and B. Jobstmann, "Program Repair without Regret," In *CAV*, pages 896–911, 2013.

[Waldinger and Lee, 1969] R. J. Waldinger and R. C. T. Lee, "PROW: a step toward automatic program writing," In *Proceedings of the 1st international joint conference on Artificial intelligence*, pages 241–252, 1969.

[Wang *et al.*, 2009] C. Wang, S. Kundu, M. Ganai, and A. Gupta, "Symbolic predictive analysis for concurrent programs," In *FM*, pages 256–272, 2009.

[Wang *et al.*, 2006] C. Wang, B. Li, H. Jin, G. Hachtel, and F. Somenzi, "Improving Ariadne's bundle by following multiple threads in abstraction refinement," *IEEE TCAD*, pages 2297–2316, 2006.

[Wang *et al.*, 2010] C. Wang, R. Limaye, M. Ganai, and A. Gupta, "Trace-based symbolic analysis for atomicity violations," In *TACAS*, pages 328–342, 2010.

[Weeratunge *et al.*, 2011] D. Weeratunge, X. Zhang, and S. Jaganathan, "Accentuating the positive: atomicity inference and enforcement using correct executions," In *OOPSLA*, pages 19–34, 2011.

[Yi and Flanagan, 2010] J. Yi and C. Flanagan, "Effects for cooperable and serializable threads," In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, pages 3–14, 2010.

[Zhang and Wang, 2014] L. Zhang and C. Wang, "Runtime prevention of concurrency related type-state violations in multithreaded applications," In *ISSTA*, pages 1–12, 2014.

List of Publications

- Pavol Černý, Edmund M. Clarke, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, Roopsha Samanta and Thorsten Tarrach,
  "From Non-preemptive to Preemptive Scheduling Using Synchronization Synthesis,"
  In *Computer Aided Verification* (2015), pages 180–197.
  Implementation: `https://github.com/thorstent/Liss`

- Ashutosh Gupta, Thomas A. Henzinger, Arjun Radhakrishna, Roopsha Samanta and Thorsten Tarrach,
  "Succinct Representation of Concurrent Trace Sets,"
  In *Principles of Programming Languages* (2015), pages 433–444.
  Implementation: `https://github.com/thorstent/TARA`

- Pavol Černý, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk and Thorsten Tarrach,
  "Regression-Free Synthesis for Concurrency,"
  In *Computer Aided Verification* (2014), pages 568–584.
  Implementation: `https://github.com/thorstent/ConRepair`

- Pavol Černý, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk and Thorsten Tarrach,
  "Efficient Synthesis for Concurrency by Semantics-Preserving Transformations,"
  In *Computer Aided Verification* (2013), pages 951–967, 2013.
  Implementation: `https://github.com/thorstent/ConcurrencySwapper`

- Michael Backes, Cătălin Hriţcu and Thorsten Tarrach,
  "Automatically verifying typing constraints for a data processing language,"
  In *Certified Programs and Proofs* (2011), pages 296–313.