Succinct Representation of Concurrent Trace Sets [*]

Ashutosh Gupta
IST Austria
agupta@ist.ac.at

Thomas A. Henzinger
IST Austria
tah@ist.ac.at

Arjun Radhakrishna
IST Austria, University of Pennsylvania
arjunrad@seas.upenn.edu

Roopsha Samanta
IST Austria
rsamanta@ist.ac.at

Thorsten Tarrach
IST Austria
ttarrach@ist.ac.at

Abstract

We present a method and a tool for generating succinct representations of sets of concurrent traces. We focus on trace sets that contain all correct or all incorrect permutations of events from a given trace. We represent trace sets as *HB-formulas* that are Boolean combinations of *happens-before* constraints between events. To generate a representation of incorrect interleavings, our method iteratively explores interleavings that violate the specification and gathers generalizations of the discovered interleavings into an HB-formula; its complement yields a representation of correct interleavings.

We claim that our trace set representations can drive diverse verification, fault localization, repair, and synthesis techniques for concurrent programs. We demonstrate this by using our tool in three case studies involving synchronization synthesis, bug summarization, and abstraction refinement based verification. In each case study, our initial experimental results have been promising.

In the first case study, we present an algorithm for inferring missing synchronization from an HB-formula representing correct interleavings of a given trace. The algorithm applies rules to rewrite specific patterns in the HB-formula into locks, barriers, and wait-notify constructs. In the second case study, we use an HB-formula representing incorrect interleavings for bug summarization. While the HB-formula itself is a concise counterexample summary, we present additional inference rules to help identify specific concurrency bugs such as data races, define-use order violations, and two-stage access bugs. In the final case study, we present a novel predicate learning procedure that uses HB-formulas representing abstract counterexamples to accelerate counterexample-guided abstraction refinement (CEGAR). In each iteration of the CEGAR loop, the procedure refines the abstraction to eliminate multiple spurious abstract counterexamples drawn from the HB-formula.

Categories and Subject Descriptors D [2]: 4—Formal methods

Keywords Trace Generalization; Concurrent Programs; Synchronization Synthesis; Bug Summarization; CEGAR

POPL '15, January 15–17, 2015, Mumbai, India.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676726.2677008

1. Introduction

Sets of concurrent traces containing permutations of events from a given concurrent trace are useful for predictive analysis (e.g., [24, 34, 35, 41]) and synchronization synthesis (e.g., [8, 9]) of shared-memory concurrent programs. Most approaches using such trace sets are restricted to specific aspects of reasoning about concurrent programs such as data race detection [24, 34], detection of safety violations [35, 41] and fixing assertion failures [8, 9]. Moreover, the representations of trace sets and exploration strategies used in some of these approaches [8, 9, 35]) *underapproximate* the target trace sets. In this paper, we present a succinct, *complete* representation of such concurrent trace sets, which can drive diverse verification, fault localization, 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.

Concurrent trace sets. First, we fix some terminology. An *execution* π of a concurrent program $\mathcal{P}$ is an alternating sequence of variable valuations and events corresponding to a feasible interleaving of instructions from the threads of $\mathcal{P}$. An execution is *good* if it satisfies a given specification, and *bad* otherwise. A *trace* is a sequence of events corresponding to an interleaving of instructions from the threads of $\mathcal{P}$. The trace of an execution π is the sequence of events within π. The language $\mathcal{L}(\tau)$ of a trace τ is the set of all executions with trace τ. A trace τ is feasible if $\mathcal{L}(\tau)$ is non-empty, and infeasible otherwise. A feasible trace τ is good if all executions in $\mathcal{L}(\tau)$ are good, and bad otherwise.

We group traces into *neighbourhoods*. The neighbourhood $\mathcal{N}_\tau$ of a trace τ contains all permutations of τ that preserve τ's intra-thread event order. The *good neighbourhood* $\mathcal{N}_\tau^g$ of a trace τ is the set containing all the good traces in $\mathcal{N}_\tau$. The *bad neighbourhood* $\mathcal{N}_\tau^b$ of a trace τ is a set containing all the bad traces in $\mathcal{N}_\tau$. The languages $\mathcal{L}(\mathcal{N}_\tau)$, $\mathcal{L}(\mathcal{N}_\tau^g)$ and $\mathcal{L}(\mathcal{N}_\tau^b)$ are the unions of the languages of all traces in $\mathcal{N}_\tau$, $\mathcal{N}_\tau^g$ and $\mathcal{N}_\tau^b$, respectively.

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 [8, 9, 41]. 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. For instance, for the

[*]This research was supported in part by the European Research Council (ERC) under grant agreement 267989 (QUAREM), by the Austrian Science Fund (FWF) NFN project S11402-N23 (RiSE), and the NSF Expeditions award CCF 1138996.

Figure 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.[a]

```
globalvars: int x, withdrawal, deposit, balance,
         deposited, withdrawn;
init: x = balance; deposited = 0; withdrawn = 0;
         withdrawal > 0; deposit > 0;

thread_withdraw:
localvars: int temp;
TW[1]: temp := balance;
TW[2]: balance := temp - withdrawal;
TW[3]: withdrawn := 1;

thread_deposit:
localvars: int temp;
TD[1]: temp := balance;
TD[2]: balance := temp + deposit;
TD[3]: deposited := 1;

thread_checkresult:
TC[1]: assume (deposited = 1 and withdrawn = 1);
TC[2]: assert (balance = x + deposit - withdrawal);
```

Exact representation of $\mathcal{N}_\tau^b$:
$hb(\mathtt{T_W}[1], \mathtt{T_D}[2]) \wedge hb(\mathtt{T_D}[1], \mathtt{T_W}[2]) \wedge hb(\mathtt{T_W}[3], \mathtt{T_C}[1]) \wedge hb(\mathtt{T_D}[3], \mathtt{T_C}[1])$
Exact representation of $\mathcal{N}_\tau^g$:
$(hb(\mathtt{T_D}[2], \mathtt{T_W}[1]) \vee hb(\mathtt{T_W}[2], \mathtt{T_D}[1])) \wedge hb(\mathtt{T_W}[3], \mathtt{T_C}[1]) \wedge hb(\mathtt{T_D}[3], \mathtt{T_C}[1])$
Representation of sound overapproximation of $\mathcal{N}_\tau^b$:
$hb(\mathtt{T_W}[1], \mathtt{T_D}[2]) \wedge hb(\mathtt{T_D}[1], \mathtt{T_W}[2])$
Representation of sound overapproximation of $\mathcal{N}_\tau^g$:
$hb(\mathtt{T_D}[2], \mathtt{T_W}[1]) \vee hb(\mathtt{T_W}[2], \mathtt{T_D}[1])$

[a]In all the examples in this paper, we represent traces using typed global variable declarations/initializations, followed by each thread's typed local variable declarations and instructions. Note that this representation depicts a trace and not a program.

trace τ in Fig. 1, $\mathcal{N}_\tau^g$ is not a partial order, but is a disjunction (i.e., union) of partial orders. In our work, we represent trace sets as *HB-formulas*. An HB-formula is a Boolean combination of *happens-before* causality constraints between events. HB-formulas can represent arbitrary finite sets of finite traces, and in particular, good and bad neighbourhoods (see Fig. 1). As we will see later, HB-formulas are not only expressive, but also versatile enough to be usable for diverse objectives.

Given a trace τ and a correctness specification, we present a method to generate an HB-formula φ_B representing the bad neighbourhood of τ. To generate φ_B, we first encode all the bad executions in $\mathcal{L}(\mathcal{N}_\tau)$ 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 $\mathcal{L}(\mathcal{N}_\tau^b)$. 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 $\mathcal{N}_\tau^b$ represented by the current φ_B, (2) *generalize* 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 generalization used in each iteration has the following properties: (a) the model obtained in the iteration satisfies φ, and (b) any trace in $\mathcal{N}_\tau$ that satisfies φ is bad. The final HB-formula obtained is an *exact* representation of $\mathcal{N}_\tau^b$.

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 $\mathcal{N}_\tau^b$. The overapproximation of $\mathcal{N}_\tau^b$ 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 procedure. We use data-flow analysis and minimal unsatisfiability core (unsat core) computation for generalizing the trace of the model into an HB-formula φ in step (2) of each iteration. This new trace generalization step has the following properties: (a) the model obtained in the iteration satisfies φ, and (b) any trace in $\mathcal{N}_\tau$ satisfying φ is either bad or infeasible.

Complementing φ_B, the succinct representation of a sound overapproximation of $\mathcal{N}_\tau^b$ yields φ_G, a succinct representation of a sound overapproximation of $\mathcal{N}_\tau^g$. Note that complementing the exact representation of $\mathcal{N}_\tau^b$ does not yield an exact representation of $\mathcal{N}_\tau^g$. In fact, our existing methodology cannot produce an exact representation of $\mathcal{N}_\tau^g$. Fig. 1 shows the exact representation of $\mathcal{N}_\tau^b$ and the representations for sound overapproximations of $\mathcal{N}_\tau^g$ and $\mathcal{N}_\tau^b$ obtained by our method for the example trace shown.

We implemented the above procedure as a tool TARA and used it to generate (succinct) representations of trace sets of programs drawn from the software verification competition (SV-Comp) [3] and the regression suites of ESBMC [31] and CONREPAIR [9].

We demonstrate the applicability of our representations of good and bad neighbourhoods of a trace to three case studies involving synchronization synthesis, bug summarization and verification based on counterexample-guided abstraction refinement (CEGAR).

Case Study: Synchronization synthesis. Shared-memory concurrent programs are excellent targets for automated *program completion*, in particular, for synthesis of missing synchronization [8, 9, 13, 30, 39]. We present a novel algorithm that uses φ_G to synthesize synchronization for eliminating the bad neighbourhood of τ. The algorithm proceeds by applying rewrite rules to derive synchronization primitives such as mutex locks, barriers, shared exclusive locks and wait-notify statements from easily-identifiable patterns in φ_G. For example, a missing mutex lock in the example in Fig. 1 that ensures the instructions at $\mathtt{T_W}[1]$ and $\mathtt{T_W}[2]$ in `thread_withdraw` do not interfere with the instructions $\mathtt{T_D}[1]$ 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 emphasize that most other synchronization synthesis techniques generate atomic sections rather than locks, wait-notify statements etc. Atomic sections are not directly implementable. Moreover, our synchronization primitives can potentially permit more correct concurrent behaviours than atomic sections. We have implemented this algorithm as an extension of our tool TARA and used it to successfully synthesize synchronization for our benchmarks.

Case Study: Bug summarization. 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. In our second case study, we use φ_B, the representation for a sound overapproximation of a trace's bad neighbourhood, for counterexample and bug summarization. The HB-formula φ_B encodes relevant *ordering* information about all counterexamples in the neighbourhood of τ 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 summarization and evaluated it on our benchmarks.

Case Study: Accelerating CEGAR. We also recognize 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. The choice of refinement procedure usually determines the number of iterations necessary. Many heuristics have been proposed to find relevant predicates quickly, e.g., [4]. 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* [1]. We have integrated our TARA-based refinement procedure within SATABS [12] 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-formulas (Sec. 2). HB-formulas 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-formulas are an intuitively appealing representation for trace sets. They can reveal specific patterns of causality relations between events that can drive diverse verification, fault localization, repair, and synthesis techniques for concurrent programs. We demonstrate the use of our tool in three applications — synchronization synthesis (Sec. 3), bug summarization (Sec. 4), and CEGAR acceleration (Sec. 5).

2. Trace Neighbourhoods and Representations

In this section, we formalize concurrent executions, traces and trace neighbourhoods. We also present algorithms and experimental results for computing good and bad neighbourhoods. The case studies in Sections 3, 4, and 5 are based on the techniques presented here.

2.1 Concurrent Programs and Traces

We consider shared-memory concurrent programs composed of a fixed number of sequential threads. In further discussion, we fix a concurrent program $\mathcal{P} = \langle V, \{T_1, \ldots, T_k\}, SV, \langle LV_1, \ldots, LV_k\rangle\rangle$ where $\{T_1, \ldots, T_k\}$ are a set of threads, SV is a 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. As the main objects of study in this paper are traces, we keep the exposition simple by not specifying syntactic and control flow details of threads at this stage. In this paper, we assume that variables range over integers and program instructions perform standard linear arithmetic operations. However, our techniques apply to a much wider variety of variable domains and operations.

Concurrent executions. A *concurrent execution* $\pi = \Gamma_0 e_1 \Gamma_1 \ldots \Gamma_{n-1} e_n \Gamma_n$ is an alternating sequence of valuations Γ_i of variables V and events e_i corresponding to some interleaving of instructions from the threads in $\mathcal{P}$—for each i, execution of e_i from valuation Γ_{i-1} leads to valuation Γ_i. Each event e is a labelled statement of the form $\texttt{T}[\ell] : stmt$, where $\texttt{T}$ is a thread identifier, ℓ is a location identifier*, and $stmt$ is an atomic instruction. We write $pid(e)$ for the thread identifier $\texttt{T}$. Without loss of generality, we assume that the location identifiers of events from each thread are sequential natural numbers, i.e., the first event from a thread gets location identifier 1, the next gets 2, and so on. Further, we abuse notation by often writing $\texttt{T}[\ell]$ instead of the event with label $\texttt{T}[\ell]$. We represent the sequence of events from thread T with location identifiers between ℓ and ℓ' (inclusive) by $\texttt{T}[\ell : \ell']$. We also use the symbol L to denote location identifier ranges such as $\ell : \ell'$.

*We assume that all location identifiers from one thread are unique. Thus, multiple occurrences of the same instruction (for example, in the body of a loop) are relabelled with unique identifiers.

We use two different formalisms to express atomic instructions.

- *Guarded actions.* Here, an instruction from thread T_i is either a guarded action $\texttt{assume}(G) \rightarrow \texttt{assign}$ or an assertion $\texttt{assert}(G)$, where G is a Boolean expression over V_i and $\texttt{assign}$ is a parallel assignment $\texttt{v}_1, \ldots, \texttt{v}_m :=$ $expr_1, \ldots, expr_m$ of expressions over V_i to variables in V_i.
- *Transition predicates.* Here, an instruction 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 instruction, respectively. For example, the assignment $\texttt{x} := \texttt{x} + \texttt{y}$ 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 Sec. 5. Assertions are represented as before, i.e., as $\texttt{assert}(G)$, where G is a Boolean expression over V_i.

An execution $\pi = \Gamma_0 e_1 \Gamma_1 \ldots e_n \Gamma_n$ is *good* if for each assertion $e_i = \texttt{T}[\ell] : \texttt{assert}(G)$, the Boolean expression G evaluates to $\texttt{true}$ under valuation Γ_{i-1}; the execution is *bad* otherwise.

Concurrent traces. A *concurrent trace* $\tau = e_1 \ldots e_n$ is a sequence of events that corresponds to some interleaving of instructions from threads in $\mathcal{P}$. The *language* $\mathcal{L}(\tau)$ of a trace $\tau = e_1 \ldots e_n$ is the set of all executions $\Gamma_0 e_1' \Gamma_1 \ldots e_n' \Gamma_{n+1}$ where $e_i = e_i'$ for $i \in [1, n]$. For a set of traces $\mathcal{N}$, we abuse notation and write $\mathcal{L}(\mathcal{N})$ instead of $\bigcup_{\sigma \in \mathcal{N}} \mathcal{L}(\sigma)$. We denote by $events(\tau)$ the set $\{e_1, \ldots, e_n\}$ of events in τ. For any two events $e_i, e_j \in events(\tau)$, we say $e_i <_\tau e_j$ if e_i occurs before e_j in τ.

A trace τ is *feasible* if its language has at least one execution (i.e., $\mathcal{L}(\tau) \neq \emptyset$), and is *infeasible* otherwise. A feasible trace τ is *good* if all executions in $\mathcal{L}(\tau)$ are good, and is *bad* otherwise.

2.2 Representing Trace Neighbourhoods

We reason about traces that differ only in the scheduling choices using trace neighbourhoods. The *neighbourhood* $\mathcal{N}_\tau$ of a trace τ is a set of traces $\mathcal{N}_\tau = \{\sigma \mid events(\sigma) = events(\tau) \wedge \forall e_i, e_j \in events(\tau) : pid(e_i) = pid(e_j) \wedge e_i <_\tau e_j \Rightarrow e_i <_\sigma e_j\}$. Intuitively, $\mathcal{N}_\tau$ contains all traces having the same events as τ and having the same order of events within each thread. A trace in $\mathcal{N}_\tau$ may be infeasible, good, or bad. We denote the subsets of good and bad traces in $\mathcal{N}_\tau$ by $\mathcal{N}_\tau^g$ and $\mathcal{N}_\tau^b$, respectively. We call $\mathcal{N}_\tau^b$ and $\mathcal{N}_\tau^g$ the *bad* and *good* neighbourhoods of τ.

Note that $\mathcal{N}_\tau$ corresponds to a partial order $(events(\tau), \sqsubseteq)$, with $e_i \sqsubseteq e_j$ iff $e_i <_\tau e_j$ and $pid(e_i) = pid(e_j)$. However, $\mathcal{N}_\tau^g$ and $\mathcal{N}_\tau^b$ do not, in general, correspond to a partial order (cf. the exact representation of $\mathcal{N}_\tau^g$ in Fig. 1).

Representing subsets of trace neighbourhoods. We represent subsets of trace neighbourhoods using *happens-before formulas*, or, *HB-formulas*. An HB-formula φ for a trace τ is either a: (a) *basic constraint* of the form $hb(e_i, e_j)$ where $e_i, e_j \in events(\tau)$; or (b) a Boolean combination of HB-formulas, 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-formulas.

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

Remark 2.1. *Our HB-formulas 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 instructions in a trace rather than the interaction between schedules and variable valuations [29].*

2.3 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 $\mathcal{N}_\tau^b$ and $\mathcal{N}_\tau^g$, i.e., to find succinct HB-formulas φ_G and φ_B such that $\mathcal{N}_\tau^g \subseteq [\![\varphi_G]\!]$, $\mathcal{N}_\tau^b \subseteq [\![\varphi_B]\!]$, and $[\![\varphi_G]\!] \cap \mathcal{N}_\tau^b = [\![\varphi_B]\!] \cap \mathcal{N}_\tau^g = \emptyset$.

Encoding bad executions. Given a trace τ, our algorithm is based on constructing a quantifier free first-order formula that represents all bad executions in $\mathcal{L}(\mathcal{N}_\tau)$. We use the concurrent trace program encoding [41] 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 instructions are expressed as guarded actions; the case where instructions are expressed as transition predicates is similar. Given a trace τ, we first rewrite it into the CSSA form.

- For each variable v, we introduce a unique name $v_{w,e}$ for each event e 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 e that reads a variable v, we replace v as follows:
  - If v is a local variable, we replace v by $v_{w,e'}$ where e' is the most recent event from the thread that writes to v; and
  - If v is a shared variable, we replace v by $v_{r,e}$ (where r stands for "read") and we store an additional constraint, where $v_{r,e} = \pi(v_\iota, v_{w,e_1}, v_{w,e_2}, \ldots, v_{w,e_\ell})$ where e_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., $v_{r,e} = \pi(v_\iota, v_{w,e_1}, \ldots, v_{w,e_\ell})$ expresses that e reads either the initial value of v, or the value written by one of $e_1, \ldots, e_\ell$.
- Further, for each event e, we define the condition that e is feasibly reached. If e is the first event in a thread, we set $cond(e) = \texttt{true}$. Otherwise, $cond(e)$ depends on the previous event from the same thread in τ (say e'). If e' is an assertion, we let $cond(e) = cond(e')$. Otherwise, e' is a guarded action $\texttt{assume}(G) \to \texttt{assign}$, and we let $cond(e) = cond(e') \wedge G$.

Example 2.2. *In the running example from Fig. 1, the statement* $\texttt{T}_\texttt{W}[1]$: $\texttt{temp} := \texttt{balance}$; *would be encoded as* $temp_{w,\texttt{T}_\texttt{W}[1]} = balance_{r,\texttt{T}_\texttt{W}[1]} \wedge balance_{r,\texttt{T}_\texttt{W}[1]} = \pi(balance_\iota, balance_{w,\texttt{T}_\texttt{D}[2]})$.

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

- *Thread orders.* In any execution in the neighbourhood of τ, the order of events in each thread is the same as in the trace τ. We define $\Phi_{PO} = \bigwedge\{hb(e_i, e_j) \mid pid(e_i) = pid(e_j) \wedge e_i <_\tau e_j\}$.
- *Variable assignments.* This part of the encoding is a direct translation of the assignments in each event into constraints. We have $\Phi_{VD} = \bigwedge_e \bigwedge_{i=1}^m v_{w,e}^i = expr^i$, where e ranges over events of the form $\texttt{T}[\ell]$: $stmt$ with $stmt$ being $\texttt{assume(G)} \to \texttt{v}^1_{\texttt{w,e}}, \ldots, \texttt{v}^\texttt{m}_{\texttt{w,e}} := expr^1, \ldots, expr^m$.
- *π-constraints.* Each π-constraint chooses a value for a read of a shared variable from possible writes. Formally, each condition $v_{r,e} = \pi(v_\iota, v_{w,e_1}, \ldots, v_{w,e_\ell})$ is rewritten as $[v_{r,e} = v_\iota \wedge \bigwedge_i hb(e, e_i)] \vee \bigvee_{i=1}^\ell [v_{r,e} = v_{w,e_i} \wedge cond(e_i) \wedge hb(e_i, e) \wedge \bigwedge_{j \neq i}(hb(e_j, e_i) \vee hb(e, e_j))]$. Intuitively, the above formula states that: (a) the value of v read by e is either the initial value of v or written by one of $e_1, \ldots, e_\ell$; (b) if the value is the initial value, all e_i happen after e; and (c) if the value is written by e_i, then e_i is feasibly reached and all conflicting writes either happen before e_i or after e. We denote by Φ_{PI} the conjunction of all such π-constraints. For example, for the π-function from Example 2.2, the corresponding constraint is $(balance_{r,\texttt{T}_\texttt{W}[1]} = balance_\iota \wedge hb(\texttt{T}_\texttt{W}[1], \texttt{T}_\texttt{D}[2])) \vee (balance_{r,\texttt{T}_\texttt{W}[1]} = balance_{w,\texttt{T}_\texttt{D}[2]} \wedge hb(\texttt{T}_\texttt{D}[2], \texttt{T}_\texttt{W}[1]))$.
- *Correctness condition.* For correctness, if an assertion event $e = \texttt{T}[\ell]$: $\texttt{assert}(G_e)$ is feasibly reached, then G_e must hold. Hence, the correctness condition is $\Phi_{COR} = \bigwedge_e(cond(e) \Rightarrow G_e)$ where e ranges over assertion events.

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

For convenience, we use an auxilliary formula Φ_{FEA} to represent the condition that each assumption must hold. We have $\Phi_{FEA} = \bigwedge_e cond(e)$ where e 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 $v_{r,e}$ in $\mathcal{V}$ is the value of v read by e in π; (c) the value of each $v_{w,e}$ in $\mathcal{V}$ is the value of v written by e in π; and (d) the value of $hb(e_i, e_j)$ in $\mathcal{V}$ is true if and only if e_i occurs before e_j in π.

Theorem 2.3 ([41]). *Given a trace τ, (a) for every model $\mathcal{V}$ of $\Phi_{CTP}(\tau)$ there is a bad execution $\pi \in \mathcal{L}(\mathcal{N}_\tau^b)$ such that π corresponds to $\mathcal{V}$; and (b) for every $\pi \in \mathcal{L}(\mathcal{N}_\tau^b)$ there is a model $\mathcal{V}$ of $\Phi_{CTP}(\tau)$ 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 τ — we now present an algorithm to compute a representation of $\mathcal{N}_\tau^b$. Algo. 1 proceeds by repeatedly computing satisfying assignments to Φ_{CTP} using an SMT solver (lines 2 and 3), and accumulating the HB-formulas 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 1 Computing the bad neighbourhood of a trace

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

Overapproximating bad neighbourhoods. While Algo. 1 computes an exact representation of $\mathcal{N}_\tau^b$, it is inefficient in practice. Hence, we forgo the goal of an exact representation. Instead, we compute a *sound overapproximation* of $\mathcal{N}_\tau^b$, which may include infeasible traces, but not good traces. Given trace τ, Algo. 2 computes sound overapproximations of $\mathcal{N}_\tau^b$ and $\mathcal{N}_\tau^g$. Algo. 2 performs several optimizations with respect to Algo. 1 to accumulate weaker constraints from each model of Φ_{CTP}, i.e., Algo. 2 attempts to accumulate larger subsets of $\mathcal{N}_\tau$ into φ_B in each iteration.

- **Data-flow analysis.** From the model $\mathcal{V}$ of $\Phi_{CTP}(\tau)$, 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 $DF_\mathcal{V}(e)$ (line 5) to compute constraints that ensure e can be feasibly reached and can read the same variable values as in $\mathcal{V}$. Given the execution corresponding to $\mathcal{V}$, let $reads(e)$, $readsG(e)$, and $srcEvent(v, e)$ represent the variables read by e, the variables read by e in the guard (if e is not a guarded assignment, $readsG(e) = \emptyset$), and the event that writes the value of v read by e. We have $DF_\mathcal{V}(e) = DF_\mathcal{V}^1(e) \cup DF_\mathcal{V}^2(e)$ where:

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

  Intuitively, $DF^1_{\mathcal{V}}$ ensures that e can read the same values as in $\mathcal{V}$ and $DF^2_{\mathcal{V}}$ ensures that e 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 generalization on $\varphi_{B'}$ through unsatisfiable core computation. In the first round, we construct a formula $\varphi_{B'} \wedge Choices(\mathcal{V}) \wedge \Phi_{\overline{CTP}}(\tau)$ where $Choices(\mathcal{V})$ fixes the initial variable values to the ones from $\mathcal{V}$ (line 9). 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 11) ensure that all executions satisfying $Choices(\mathcal{V})$ are bad. Note that if all instructions are deterministic, the above formula is always unsatisfiable. In the second round (line 13), 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 13) ensure that any feasible execution is necessarily bad.

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

The sound overapproximation, φ_G, of $\mathcal{N}^g_\tau$ is obtained by complementing φ_B (line 15). Note that φ_B returned is in disjunctive normal form (DNF), while φ_G is in conjunctive normal form (CNF).

Algorithm 2 Computing sound overapproximations of the bad and good neighbourhoods of a trace

```
Require: Trace $\tau$
Ensure: HB-formulas $(\varphi_B, \varphi_G)$ such that $\mathcal{N}^g_\tau \subseteq [\![\varphi_G]\!]$, $\mathcal{N}^b_\tau \subseteq [\![\varphi_B]\!]$, and $[\![\varphi_G]\!] \cap [\![\varphi_B]\!] = \emptyset$.
 1: $\Phi \leftarrow \Phi_{CTP}(\tau)$; $\varphi_B \leftarrow$ false
 2: while $\Phi \wedge \neg\varphi_B$ is satisfiable do
 3:     $\mathcal{V} \leftarrow$ satisfying assignment for $\Phi \wedge \neg\varphi_B$
 4:     {Data-flow analysis}
 5:     $DF \leftarrow DF_{\mathcal{V}}(e^*)$ where $e^*$ is the failing assertion in $\mathcal{V}$
 6:     $ADF \leftarrow \bigcup_{(v,e_i,e_j) \in DF} \bigcup_{\{e_k \mid e_k \text{ writes } v\}} \big(\{(v, e_k, e_i) \mid$
            $\mathcal{V} \models hb(e_k, e_i)\} \cup \{(v, e_j, e_k) \mid \mathcal{V} \models hb(e_j, e_k)\}\big)$
 7:     $\varphi_{B'} \leftarrow \bigwedge_{(v,e_i,e_j) \in DF \cup ADF} hb(e_i, e_j)$
 8:     {Unsat-core computation}
 9:     $Choices(\mathcal{V}) \leftarrow \bigwedge_{v \in V} v_\iota = \mathcal{V}[v_\iota]$
10:     if $\varphi_{B'} \wedge Choices(\mathcal{V}) \wedge \Phi_{\overline{CTP}}(\tau))$ is unsatisfiable then
11:         $\varphi_{B'} \leftarrow MinUNSATCore(Soft \leftarrow \varphi_{B'},$
                $Hard \leftarrow Choices(\mathcal{V}) \wedge \Phi_{\overline{CTP}}(\tau))$
12:     if $\varphi_{B'} \wedge \Phi_{FEA}(\tau) \wedge \Phi_{\overline{CTP}}(\tau))$ is unsatisfiable then
13:         $\varphi_{B'} \leftarrow MinUNSATCore(Soft \leftarrow \varphi_{B'},$
                $Hard \leftarrow \Phi_{FEA}(\tau) \wedge \Phi_{\overline{CTP}}(\tau))$
14:     $\varphi_B \leftarrow \varphi_B \vee \varphi_{B'}$
15: $\varphi_G \leftarrow \neg\varphi_B$; return $(\varphi_B, \varphi_G)$
```

Theorem 2.4. *For a trace τ, if Algo. 2 returns (φ_B, φ_G), then $\mathcal{N}^b_\tau \subseteq [\![\varphi_B]\!]$, $\mathcal{N}^g_\tau \subseteq [\![\varphi_G]\!]$, and $[\![\varphi_G]\!] \cap \mathcal{N}^b_\tau = [\![\varphi_B]\!] \cap \mathcal{N}^g_\tau = \emptyset$.*

2.4 Implementation and Evaluation

We have implemented Algorithms 1 and 2 in a tool TARA (accessible at `https://github.com/thorstent/TARA`). TARA consists of 4000 lines of C++ code and uses Z3 [15] 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 instructions (in SMT-LIB format) in each thread. This makes TARA independent and easy to use with any front-end that can translate instructions to the SMT-LIB syntax. We use a modified version of CONREPAIR [9] to generate CTRC files for bad traces. CONREPAIR, in turn, uses CBMC [11] to find bad traces in programs and CPACHECKER [5] to translate C statements into the SMT-LIB format.

TARA has a number of different output options. Algo. 1 generates an HB-formula in DNF, which is often large. Algo. 2 generates a succinct HB-formula in DNF, the sizes of whose disjuncts are locally minimized. 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 minimization technique—we use Z3 incrementally with triggers to activate and deactivate expressions for unsat core minimization. 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 [3] (suite sv) and the regression-test suites of CONREPAIR [9] (suite cr) and ESBMC [31] (suite es). We also use a set of small handmade examples with common bug patterns (suite hm). The cr suite contains simplifed 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.

We ran our experiments on a laptop with a 4-core Core i5 CPU and 8GB of RAM running Linux. Our results are presented in Table 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 1 indicates the complexity of the benchmarks in terms of the number of threads and instructions. 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 Algo. 1 and Algo. 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). Algo. 1 times out after 10 minutes in many cases—in such cases, we report the number of loop iterations completed before the timeout. With Algo. 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.

3. Case Study: Synchronization Synthesis

In our first case study, we use the representation of a sound overapproximation of the good neighbourhood of a trace τ (returned as φ_G by Algo. 2) to synthesize synchronization that eliminates the bad neighbourhood of τ. Missing synchronization primitives such as locks, barriers, and wait-notify statements present themselves as easily identifiable HB-formula *patterns* in φ_G. Our procedure derives the required synchronization using rules that rewrite such patterns into the corresponding primitives.

Table 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-4	N/A	2/32	16/5.0	>24.6k	8	TO	309ms	TO	8/2.0
loop-8	N/A	2/64	32/9.0	>15.3k	16	TO	3s	TO	16/2.0
loop-16	N/A	2/128	64/17.0	>4.4k	32	TO	1.1min	TO	32/2.0
loop-32	N/A	2/256	128/33.0	>674	64	TO	35.5min	TO	64/2.0

Synchronization primitives. We first describe various synchronization primitives that we derive. Recall from Sec. 2 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-Notify.* A wait-notify $\mathtt{WaitNotify}\big(\mathtt{T}_2[\ell_2], \mathtt{T}_1[\ell_1]\big)$ denotes a `wait` to make $\mathtt{T}_2[\ell_2]$ wait for $\mathtt{T}_1[\ell_1]$ to complete, and a `notify` to make $\mathtt{T}_1[\ell_1]$ signal $\mathtt{T}_2[\ell_2]$ upon completion.
2. *Locks.* A lock $\mathtt{Lk}\big(\mathtt{T}_1[\mathtt{L}_1], \ldots, \mathtt{T}_n[\mathtt{L}_n]\big)$ denotes a common lock protecting each event sequence $\mathtt{T}_i[\mathtt{L}_i]$, $\mathtt{i} \in [1, \mathtt{n}]$, to ensure that these event sequences cannot execute concurrently.
3. *Barriers.* A barrier $\mathtt{Barrier}\big(\mathtt{T}_1[\ell_1], \ldots, \mathtt{T}_n[\ell_n]\big)$ at location ℓ_i of thread $\mathtt{T}_i$, $\mathtt{i} \in [1, \mathtt{n}]$, prevents each thread $\mathtt{T}_i$ from proceeding beyond ℓ_i until every other thread $\mathtt{T}_j$ reaches ℓ_j. In other words, $\mathtt{T}_i$ cannot execute the event at ℓ_i until every other $\mathtt{T}_j$ executes the event at $\ell_j - 1$.
4. *Shared-exclusive locks.* A shared-exclusive lock (or, a readers-writers lock) $\mathtt{ShExLock}\big(\mathtt{Sh} : \mathtt{T}_{s1}[\mathtt{L}_{s1}], \ldots, \mathtt{T}_{sn}[\mathtt{L}_{sn}], \mathtt{Ex} : \mathtt{T}_{x1}[\mathtt{L}_{x1}], \ldots, \mathtt{T}_{xm}[\mathtt{L}_{xm}]\big)$ permits concurrent execution of all event sequences $\mathtt{T}_{si}[\mathtt{L}_{si}]$, $\mathtt{i} \in [1, \mathtt{n}]$, while preventing concurrent execution of (a) any two $\mathtt{T}_{xi}[\mathtt{L}_{xi}]$ and $\mathtt{T}_{xj}[\mathtt{L}_{xj}]$ with $\mathtt{i} \neq \mathtt{j}$, and (b) any $\mathtt{T}_{xi}[\mathtt{L}_{xi}]$ and $\mathtt{T}_{sj}[\mathtt{L}_{sj}]$.

Rewriting φ_G to derive synchronization. During the rewrite process below, we use disjunctive formulae (denoted by ψ) where each disjunct is either an atomic *hb*-constraint of the form $hb(\mathtt{T}_i[\ell_i], \mathtt{T}_j[\ell_j])$, or a synchronization primitive. For a trace τ, we repeatedly apply the rewrite rules from Fig. 2 on φ_G (in CNF, as returned from Algo. 2) until no more rules are applicable. The ADD.WAITNOTIFY, ADD.LOCK and ADD.BARRIER rules *introduce* the wait-notify, 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 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 Algo. 2, is already optimized, we do not merge `WaitNotify` primitives.

We explain two of the above rules here. The premise of the ADD.LOCK rule asks for two event sequences $\mathtt{T}_1[\ell_1 : \ell_1']$ and $\mathtt{T}_2[\ell_2 : \ell_2']$ such that one of them has to finish execution before the other starts, i.e., $hb(\mathtt{T}_1[\ell_1'], \mathtt{T}_2[\ell_2]) \vee hb(\mathtt{T}_2[\ell_2'], \mathtt{T}_1[\ell_1])$. Equivalently, the two event sequences do not execute concurrently. This is enforced by the lock $\mathtt{Lk}\big(\mathtt{T}_1[\ell_1 : \ell_1'], \mathtt{T}_2[\ell_2 : \ell_2']\big)$. 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 formulae. Upon termination, we get a CNF formula over synchronization primitives. We pick a set $\mathcal{S}$ of synchronization primitives, consisting of one primitive from each conjunct. Let $\mathcal{P}^{\mathcal{S}}$ be the program obtained by inserting each synchronization primitive in $\mathcal{S}$ into the corresponding position in the original concurrent program $\mathcal{P}$.

Theorem 3.1 (Soundness of rewrite rules)**.** *Given a trace τ, let $\mathcal{P}^{\mathcal{S}}$ be obtained as described above. Let $\pi \in \mathcal{L}(\mathcal{N}_\tau)$ be a deadlock-free execution of $\mathcal{P}^{\mathcal{S}}$. Then $\pi \notin \mathcal{L}(\mathcal{N}_\tau^b)$, i.e., π is not bad.*

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

Note that our rewrite rules are by no means complete. It may be possible to derive the above synchronization primitives using different rules that represent other scenarios. Further, our rewrite

Figure 2 Rewrite rules for synchronization synthesis

$$\frac{hb(\texttt{T}_1[\ell_1'], \texttt{T}_2[\ell_2]) \vee hb(\texttt{T}_2[\ell_2'], \texttt{T}_1[\ell_1]) \vee \psi \quad \ell_1 \leq \ell_1' \quad \ell_2 \leq \ell_2'}{\texttt{Lk}\big(\texttt{T}_1[\ell_1 : \ell_1'], \texttt{T}_2[\ell_2 : \ell_2']\big) \vee \psi}\ \textsc{Add.Lock} \qquad \frac{hb(\texttt{T}_1[\ell_1], \texttt{T}_2[\ell_2]) \vee \psi}{\texttt{WaitNotify}\big(\texttt{T}_2[\ell_2], \texttt{T}_1[\ell_1]\big) \vee \psi}\ \textsc{Add.WaitNotify}$$

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

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

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

$$\frac{\big(\texttt{Lk}\big(\texttt{T}_1[\ell_1^{\texttt{a}}, \ell_1^{\texttt{a}'}], \texttt{T}_2[\ell_2^{\texttt{a}}, \ell_2^{\texttt{a}'}]\big) \vee \psi\big) \wedge \big(\texttt{Lk}\big(\texttt{T}_1[\ell_1^{\texttt{b}}, \ell_1^{\texttt{b}'}], \texttt{T}_2[\ell_2^{\texttt{b}}, \ell_2^{\texttt{b}'}]\big) \vee \psi\big) \qquad \ell_1^{\texttt{a}} \leq \ell_1^{\texttt{b}} \leq \ell_1^{\texttt{a}'} \qquad \ell_2^{\texttt{b}} \leq \ell_2^{\texttt{a}} \leq \ell_2^{\texttt{b}'}}{\texttt{Lk}\big(\texttt{T}_1[\ell_1^{\texttt{a}}, \max(\ell_1^{\texttt{a}'}, \ell_1^{\texttt{b}'})]], \texttt{T}_2[\ell_2^{\texttt{b}}, \max(\ell_2^{\texttt{b}'}, \ell_2^{\texttt{a}'})]\big) \vee \psi}\ \textsc{Merge.Locks.Deadlocks}$$

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

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

system can also be extended to other synchronization primitives. We now present examples illustrating the application of our rules.

Example 3.2. *For the example trace shown in Fig. 1, φ_G is given by* $hb(\texttt{T}_\texttt{D}[2], \texttt{T}_\texttt{W}[1]) \vee hb(\texttt{T}_\texttt{W}[2], \texttt{T}_\texttt{D}[1])$. *Applying the* ADD.LOCK *rewrite rule yields* $\texttt{Lk}\big(\texttt{T}_\texttt{W}[1 : 2], \texttt{T}_\texttt{D}[1 : 2]\big)$.

Example 3.3. *For the example trace shown in Fig. 3(a), φ_G is given by* $(hb(\texttt{T}_\texttt{F}[4], \texttt{T}_\texttt{S}[3]) \vee hb(\texttt{T}_\texttt{S}[4], \texttt{T}_\texttt{F}[3])) \wedge hb(\texttt{T}_\texttt{S}[4], \texttt{T}_\texttt{F}[5]) \wedge hb(\texttt{T}_\texttt{F}[4], \texttt{T}_\texttt{S}[5])$. *Applying the* ADD.LOCK *rewrite rule yields:* $\texttt{Lk}\big(\texttt{T}_\texttt{F}[3 : 4], \texttt{T}_\texttt{S}[3 : 4]\big) \wedge hb(\texttt{T}_\texttt{S}[4], \texttt{T}_\texttt{F}[5]) \wedge hb(\texttt{T}_\texttt{F}[4], \texttt{T}_\texttt{S}[5])$. *Applying the* ADD.BARRIER *rule yields:* $\texttt{Lk}\big(\texttt{T}_\texttt{F}[3 : 4], \texttt{T}_\texttt{S}[3 : 4]\big) \wedge \texttt{Barrier}\big(\texttt{T}_\texttt{F}[5], \texttt{T}_\texttt{S}[5]\big)$.

Example 3.4. *For the example trace shown in Fig. 3(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:* $hb(\texttt{T}_\texttt{I}[2], \texttt{T}_\texttt{F}[2]) \wedge hb(\texttt{T}_\texttt{I}[2], \texttt{T}_\texttt{S}[2]) \wedge (\texttt{Lk}\big(\texttt{T}_\texttt{F}[4], \texttt{T}_\texttt{S}[3 : 4]\big) \vee \psi_1) \wedge (\texttt{Lk}\big(\texttt{T}_\texttt{F}[3 : 4], \texttt{T}_\texttt{S}[4]\big) \vee \psi_1) \wedge (\texttt{Lk}\big(\texttt{T}_\texttt{F}[4], \texttt{T}_\texttt{S}[3 : 4]\big) \vee \psi_2) \wedge (\texttt{Lk}\big(\texttt{T}_\texttt{F}[3 : 4], \texttt{T}_\texttt{S}[4]\big) \vee \psi_2)$.
- *Applying* MERGE.LOCKS *next yields:* $hb(\texttt{T}_\texttt{I}[2], \texttt{T}_\texttt{F}[2]) \wedge hb(\texttt{T}_\texttt{I}[2], \texttt{T}_\texttt{S}[2]) \wedge (\texttt{Lk}\big(\texttt{T}_\texttt{F}[3 : 4], \texttt{T}_\texttt{S}[3 : 4]\big) \vee \psi_1) \wedge (\texttt{Lk}\big(\texttt{T}_\texttt{F}[3 : 4], \texttt{T}_\texttt{S}[3 : 4]\big) \vee \psi_2)$.
- *Finally, applying the* ADD.WAITNOTIFY *rule yields:* $\texttt{WaitNotify}\big(\texttt{T}_\texttt{F}[2], \texttt{T}_\texttt{I}[2]\big) \wedge \texttt{WaitNotify}\big(\texttt{T}_\texttt{S}[2], \texttt{T}_\texttt{I}[2]\big) \wedge (\texttt{Lk}\big(\texttt{T}_\texttt{F}[3 : 4], \texttt{T}_\texttt{S}[3 : 4]\big) \vee \psi_1) \wedge (\texttt{Lk}\big(\texttt{T}_\texttt{F}[3 : 4], \texttt{T}_\texttt{S}[3 : 4]\big) \vee \psi_2)$.

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* $\{\texttt{WaitNotify}\big(\texttt{T}_\texttt{F}[2], \texttt{T}_\texttt{I}[2]\big), \texttt{WaitNotify}\big(\texttt{T}_\texttt{S}[2], \texttt{T}_\texttt{I}[2]\big), \texttt{Lk}\big(\texttt{T}_\texttt{F}[3 : 4], \texttt{T}_\texttt{S}[3 : 4]\big)\}$.

3.1 Experiments

We implemented the above procedure as an extension to TARA. Given a trace τ, TARA supports synchronization synthesis as an optional step after generating succinct representations of $\mathcal{N}_\tau^g$ and $\mathcal{N}_\tau^b$. The implementation first attempts to apply the rules ADD.BARRIER, ADD.LOCK and ADD.WAITNOTIFY (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 synchronization synthesis experiments in Table 2. In each case, we report the numbers of locks (#L), barriers (#B) and wait-notify (#WN) primitives synthesized. The synthesized synchronization matched our (human) intuition about the repairs needed. Since TARA generates fairly small φ_G formulae, the synthesis takes less than 50 microseconds in every case.

Table 2 Experiments: synchronization synthesis

Name	#L	#B	#WN	Name	#L	#B	#WN
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				

4. Case Study: Bug Summarization

In our second case study, we use the representation for a sound overapproximation of the bad neighbourhood of a trace τ (returned as φ_B by Algo. 2) for counterexample summarization and bug summarization. The HB-formula φ_B encapsulates relevant ordering information about all counterexamples in the neighbourhood of τ and can be viewed as a stand-alone counterexample summary. For instance, in Fig. 3(c), one may view $\varphi_B = hb(\texttt{T}_\texttt{N}[2], \texttt{T}_\texttt{P}[2])$ 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.

4.1 Inferring Bug Summaries from φ_B

We assume φ_B is in DNF. Our inference rules are presented in Fig. 4. For a thread T, a location ℓ, and a global program vari-

Figure 3 Example programs

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

thread_firsthalf:
locals: float temp, localsum;
init: localsum = 0;
T_F[1]: localsum := localsum + value1;
T_F[2]: localsum := localsum + value2;
T_F[3]: temp := sum;
T_F[4]: sum := temp + localsum;
T_F[5]: value1 := value1/sum;
T_F[6]: value2 := value2/sum;
T_F[7]: flag1 := 1;

thread_secondhalf:
locals: temp, localsum;
init: localsum = 0;
T_S[1]: localsum := localsum + value3;
T_S[2]: localsum := localsum + value4;
T_S[3]: temp := sum;
T_S[4]: sum := temp + localsum;
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 and flag2 = 1);
T_C[2]: assert (value1 + value2 + value3 + value4 = 1);
```

φ_G: $(hb(\mathrm{T_F}[4],\mathrm{T_S}[3]) \vee hb(\mathrm{T_S}[4],\mathrm{T_F}[3])) \wedge hb(\mathrm{T_S}[4],\mathrm{T_F}[5]) \wedge hb(\mathrm{T_F}[4],\mathrm{T_S}[5])$

(a) Normalization. The goal of the program this trace is drawn from is to normalize 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 normalized values is 1.

```
globals: pointer hw_start;
         int registered;
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: $hb(\mathrm{T_N}[2],\mathrm{T_P}[2])$

(c) Network device initializer. 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 intrmask, initdone, workqueueitems,
         interrupts;
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]: temp := workqueueitems;
T_F[4]: workqueueitems := temp + 1;
T_F[5]: interrupts := interrupts + 1;

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

thread_checkworkqueue:
T_C[1]: assert (workqueueitems ≥ interrupts);
```

φ_G: $hb(\mathrm{T_I}[2],\mathrm{T_F}[2]) \wedge hb(\mathrm{T_I}[2],\mathrm{T_S}[2]) \wedge$
$(hb(\mathrm{T_S}[4],\mathrm{T_F}[4]) \vee hb(\mathrm{T_F}[4],\mathrm{T_S}[3]) \vee \psi_1) \wedge (hb(\mathrm{T_F}[4],\mathrm{T_S}[4]) \vee hb(\mathrm{T_S}[4],\mathrm{T_F}[3]) \vee \psi_1) \wedge$
$(hb(\mathrm{T_S}[4],\mathrm{T_F}[4]) \vee hb(\mathrm{T_F}[4],\mathrm{T_S}[3]) \vee \psi_2) \wedge (hb(\mathrm{T_F}[4],\mathrm{T_S}[4]) \vee hb(\mathrm{T_S}[4],\mathrm{T_F}[3]) \vee \psi_2)$

(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 initialization is complete (captured by the `initdone` variable). The `irqhandlers` add items to a workqueue; the addition of items is modeled 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.

```
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: $hb(\mathrm{T_D}[3],\mathrm{T_P}[2]) \wedge hb(\mathrm{T_P}[3],\mathrm{T_D}[4])$

(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.

Figure 4 Inference rules for bug summarization[a]

$$\frac{hb(\mathtt{T}_1[\ell_1'], \mathtt{T}_2[\ell_2]) \wedge hb(\mathtt{T}_2[\ell_2], \mathtt{T}_1[\ell_1'']) \wedge \psi \quad read(\mathtt{T}_1[\ell_1'], v) \quad write(\mathtt{T}_1[\ell_1''], v) \quad access(\mathtt{T}_2[\ell_2], v)}{\mathtt{DataRace}(\{\mathtt{T}_1[\ell_1'], \mathtt{T}_1[\ell_1'']\}, \mathtt{T}_2[\ell_2])} \textsc{ DataRace.1}$$

$$\frac{hb(\mathtt{T}_1[\ell_1'], \mathtt{T}_2[\ell_2'']) \wedge hb(\mathtt{T}_2[\ell_2'], \mathtt{T}_1[\ell_1'']) \wedge \psi \quad read(\mathtt{T}_1[\ell_1'], v) \quad write(\mathtt{T}_1[\ell_1''], v) \quad read(\mathtt{T}_2[\ell_2'], v) \quad write(\mathtt{T}_2[\ell_2''], v)}{\mathtt{DataRace}(\{\mathtt{T}_1[\ell_1'], \mathtt{T}_1[\ell_1'']\}, \{\mathtt{T}_2[\ell_2'], \mathtt{T}_2[\ell_2'']\})} \textsc{ DataRace.2}$$

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

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

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

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

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

[a]In this figure, $\ell_1 < \ell_1' < \ell_1''$ and $\ell_2 < \ell_2' < \ell_2''$.

able v, (a) $read(\mathtt{T}[\ell], v)$ denotes that event $\mathtt{T}[\ell]$ reads from v, (b) $write(\mathtt{T}[\ell], v)$ denotes that event $\mathtt{T}[\ell]$ writes to v, and (c) $access(\mathtt{T}[\ell], v)$ denotes that 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, $read(\mathtt{T}_1[\ell_1], v) \to write(\mathtt{T}_2[\ell_2], v)$ 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. Recall that in our framework, every instruction is assumed to execute atomically. This includes statements of the form $\mathtt{v} := \mathtt{v} + \mathtt{1}$, which may execute non-atomically at a low-level. Hence, to infer data races corresponding to *concurrent* accesses of a shared variable v, we need to model statements at a lower level, i.e., by separating events into several low-level atomic events. In most cases, these low-level atomic events either read or write variables, but not both. For instance, a decomposition of an event e_1 with instruction $\mathtt{v} := \mathtt{v} + \mathtt{1}$ is given by $e_1'; e_1''$, where event e_1' has instruction $\mathtt{r} := \mathtt{v} + \mathtt{1}$, event e_1'' has instruction $\mathtt{v} := \mathtt{r}$, and r is a local variable modelling a register. In this case, a data race between event e_1 and some other event e_2 accessing v in another thread manifests itself in a trace σ as the ordering pattern $e_1' <_\sigma e_2 <_\sigma e_1''$. 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 $read(\mathtt{T}_1[\mathtt{l}_1'], v) \to access(\mathtt{T}_2[\mathtt{l}_2], v) \to write(\mathtt{T}_1[\mathtt{l}_1''], v)$ is found in φ_B.

Further, if e_2 is also decomposed into $e_2'; e_2''$, where e_2' reads from v and e_2'' writes to v, a data race can manifest in a trace σ as $e_1 <_\sigma e_2' \wedge e_1' <_\sigma e_2$. 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 $read(\mathtt{T}_1[\ell_1'], v) \to write(\mathtt{T}_2[\ell_2''], v)$ and $read(\mathtt{T}_2[\ell_2'], v) \to write(\mathtt{T}_1[\ell_1''], v)$ is found in the same disjunct of φ_B.

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

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) $write(\mathtt{T}_1[\ell_1], v) \to read(\mathtt{T}_2[\ell_2], v) \to read(\mathtt{T}_2[\ell_2'], w) \to write(\mathtt{T}_1[\ell_1'], w)$; or (b) $read(\mathtt{T}_1[\ell_1], v) \to write(\mathtt{T}_2[\ell_2], v) \to write(\mathtt{T}_2[\ell_2'], w) \to read(\mathtt{T}_1[\ell_1'], w)$.

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 $read(\mathtt{T}_1[\ell_1], v) \to write(\mathtt{T}_2[\ell_2], v)$ manifests in a disjunct δ, the rule infers a define-use order violation if there exists a trace $\sigma \in \mathcal{N}_\tau$ such that σ satisfies δ and $\mathtt{T}_1[\ell_1]$ precedes all events that write to v in σ.

Starting from φ_B given in DNF, we repeatedly apply the inference rules from Fig. 4 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 4.1. *For the example trace shown in Fig. 1, φ_B is given by $hb(\mathtt{T}_\mathtt{W}[1], \mathtt{T}_\mathtt{D}[2]) \wedge hb(\mathtt{T}_\mathtt{D}[1], \mathtt{T}_\mathtt{W}[2])$. Since $read(\mathtt{T}_\mathtt{W}[1], balance)$, $write(\mathtt{T}_\mathtt{W}[2], balance)$, $read(\mathtt{T}_\mathtt{D}[1], balance)$ and $write(\mathtt{T}_\mathtt{D}[2], balance)$ 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 synchronization* $\mathtt{Lk}(\mathtt{T}_\mathtt{W}[1:2], \mathtt{T}_\mathtt{D}[1:2])$ *synthesized in Example 3.2.*

Example 4.2. *Consider the example trace shown in Fig. 3(c). In our encoding, the pointer hw_start is modelled as an integer variable hw that is initially 0 (since hw_start is uninitialized). The pointer dereference in $\mathtt{T}_\mathtt{N}[2]$ is modelled as* $\mathtt{assert}(\mathtt{hw} > 0)$. *For this example, φ_B is given by $hb(\mathtt{T}_\mathtt{N}[2], \mathtt{T}_\mathtt{P}[2])$. Since $read(\mathtt{T}_\mathtt{N}[2], hw)$ and $write(\mathtt{T}_\mathtt{P}[2], hw)$ hold, and trace $\mathtt{T}_\mathtt{P}[1], \mathtt{T}_\mathtt{N}[1], \mathtt{T}_\mathtt{N}[2], \mathtt{T}_\mathtt{P}[2]$ satisfies the last premise of the* DefineUse *rule, we can apply the rule to infer a define-use order violation between $\mathtt{T}_\mathtt{N}[2]$ and $\mathtt{T}_\mathtt{P}[2]$.*

Example 4.3. *For the example trace shown in Fig. 3(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* $\mathtt{TwoStageAccessBug}(\mathtt{T_D}[3:4], \mathtt{T_P}[2:3])$.

4.2 Experiments

Given a trace τ, Tara supports bug summarization 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 summarization on our test-suite are presented in Table 3. We report the numbers of data races (#DR), atomicity violations (#AV), two-stage access bugs (#2S) and define-use bugs (#DU) inferred. The Human column in the table presents a classification of the bugs present in the benchmarks, as reported by an expert user (OV stands for order violation). The last column indicates if Tara's bug summary matched the human classification. For the majority of benchmarks, Tara summarized 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 3 Experiments: bug summarization

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. Case Study: Accelerating CEGAR

In the final case study, we present a procedure for learning predicates for refinement in a CEGAR loop [14], 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, [4]). 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* [1].

5.1 Abstraction and Refinement

An *abstract model* of a concurrent program $\mathcal{P} = \langle V, \{T_1, \dots, T_k\}, SV, \langle LV_1, \dots, LV_k\rangle\rangle$ is another concurrent program $\hat{\mathcal{P}} = \langle V, \{\hat{T}_1, \dots, \hat{T}_k\}, SV, \langle LV_1, \dots, LV_k\rangle\rangle$ such that, for each $i \in [1, k]$ and event e in T_i, there is an event $\hat{e}$ in $\hat{T}_i$ such that if $\Gamma_0 e \Gamma_1$ is feasible then $\Gamma_0 \hat{e} \Gamma_1$ is feasible.

In predicate abstraction, the abstract event $\hat{e}$ corresponding to an event e is defined using a set of predicates as follows. Let us suppose predicates $\rho_1, \dots, \rho_m$ are used for abstraction. Let $i \in [1, m]$. Let β_i be the weakest precondition of e over ρ_i, and γ_i be the weakest precondition of e over $\neg\rho_i$. Let $\hat{\beta_i}$ and $\hat{\gamma_i}$ be the weakest formulas that are Boolean combinations of $\rho_1, \dots, \rho_m$, and imply β_i and γ_i, respectively. $\Gamma_0 \hat{e} \Gamma_1$ is feasible iff $\forall i \in [1, m] : (\Gamma_0 \models \hat{\beta_i} \rightarrow \Gamma_1 \models \rho_i) \wedge (\Gamma_0 \models \hat{\gamma_i} \rightarrow \Gamma_1 \models \neg\rho_i)$.

Let $\Gamma_0 \hat{e}_1 \Gamma_1 \dots \hat{e}_n \Gamma_n$ be a spurious counterexample, i.e., a trace in the abstract model that violates the specification. A refinement procedure computes additional predicates $\alpha_0, \alpha_1, \dots, \alpha_{n-1}, \alpha_n$ over program variables that satisfy the following constraint.

$$\alpha_0 = true \wedge \alpha_n = false \wedge \bigwedge_{i=1}^{n} \alpha_{i-1} \wedge e_i \rightarrow \alpha'_i$$

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 the execution of an instruction. An abstract model computed using predicates $\alpha_1, \dots, \alpha_{n-1}$ is guaranteed to not exhibit the spurious counterexample [23].

5.2 Sampling an HB-formula

We pass trace $\hat{e}_1 \dots \hat{e}_n$ to Tara and obtain an HB-formula φ_B in DNF to represent bad abstract traces. φ_B is a formula over events $\hat{e}_1 \dots \hat{e}_n$. With slight abuse of notation, we assume that φ_B is a formula over events $e_1 \dots e_n$, 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 τ_1 and τ_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 $e^2_{k_2} = e_b$.

$$\begin{aligned}\tau_1 &= \underbrace{e^0_1 \ldots e^0_{k_0}}_{prefix}\underbrace{e^1_1 \ldots e^1_{k_1}}\underset{\bowtie}{}\underbrace{e^2_1 \ldots e^2_{k_2}}\underbrace{e^3_1 \ldots e^3_{k_3}}_{suffix}\\ \tau_2 &= \overbrace{e^0_1 \ldots e^0_{k_0}}\overbrace{e^2_1 \ldots e^2_{k_2}}\overbrace{e^1_1 \ldots e^1_{k_1}}\overbrace{e^3_1 \ldots e^3_{k_3}}\end{aligned}$$

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 $e^1_1 \ldots e^1_{k_1}$ and $e^2_1 \ldots e^2_{k_2}$ are swapped. From the traces, we obtain refinement predicates $\alpha_1 \ldots \alpha_{k_0}$, $\beta_1 \ldots \beta_{k_1+k_2}$, $\gamma_1 \ldots \gamma_{k_1+k_2}$, and $\delta_1 \ldots \delta_{k_3}$ by solving the following constraints.

$$\alpha_0 = true \wedge \bigwedge_{i=1}^{k_0} (\alpha_{i-1} \wedge e^0_i \rightarrow \alpha'_i) \wedge \alpha_{k_0} = \beta_0 = \gamma_0 \qquad \text{(prefix)}$$

$$\bigwedge_{i=1}^{k_1} (\beta_{i-1} \wedge e^1_i \rightarrow \beta'_i) \wedge \bigwedge_{i=k_1+1}^{k_1+k_2} (\beta_{i-1} \wedge e^2_{i-k_1} \rightarrow \beta'_i) \qquad \text{(mid trace 1)}$$

$$\bigwedge_{i=1}^{k_2} (\gamma_{i-1} \wedge e^2_i \rightarrow \gamma'_i) \wedge \bigwedge_{i=k_2+1}^{k_2+k_1} (\gamma_{i-1} \wedge e^1_{i-k_2} \rightarrow \gamma'_i) \qquad \text{(mid trace 2)}$$

$$\delta_0 = \beta_{k_1+k_2} = \gamma_{k_1+k_2} \wedge \bigwedge_{i=1}^{k_3} (\delta_{i-1} \wedge e^3_i \rightarrow \delta'_i) \wedge \delta_{k_3} = false \qquad \text{(suffix)}$$

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.

5.3 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. [7, 22]). Since we are aiming for simultaneous refinement, we prefer the solutions for the unknown predicates to be simple atomic formulas. 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 formulas. We use the method that is presented in Sec. 4 of [1] for the theory of linear arithmetic that forces a solver to return solutions for the above constraints with single atomic formulas if such a solution exists.

Table 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

5.4 Experimental Results

We have implemented the above refinement procedure in SatAbs [16] and refer to the modified version as SatAbs[Tara]. In Table 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 examples. However, the total time of verification increases for two examples due to the fact that our refinement procedure is not well optimized.

6. Related Work

Representations of trace sets. The Φ_{CTP} encoding used in Sec. 2 was introduced in [41], and subsequently generalized in [27, 37]. We may find more applications and variations of our tool Tara from exploring other suitable happens-before constraint based encodings of "interesting" interleavings from the body of predictive analysis literature (cf. [35, 38]). Concurrent counterexample traces have been generalized into partial order (Mazurkiewicz) traces in [8, 9]. As demonstrated in this paper, partial orders may not be adequate to represent arbitrary trace sets. The work in [20] introduces a new data structure, an inductive data-flow graph (iDFG), to generalize proofs of programs. While iDFGs is also a representation of (concurrent) trace sets, it is not clear how one may apply iDFGs for program debugging or synthesis. In other work, interference scenarios have been proposed in [19] to represent concurrent executions that are behaviourally equivalent under the same input values. For sequential programs, the authors in [2] represent all counterexamples of recursive programs using pushdown automata.
Synchronization synthesis. In the formal methods and programming languages community, synchronization synthesis of concurrent programs has been an active area of research for a long time [8, 9, 13, 30, 39, 40]. In the past, the approaches in [13, 40] were based on inferring synchronization by constructing and exploring the entire product graph or tableaux corresponding to a concurrent program. Recent approaches infer synchronization incrementally from traces [39] or generalizations of bad traces [8, 9]. However, the techniques from [8, 9, 39] infer atomic sections rather than locks—atomic sections are not directly implementable and need to be translated into locks either manually, or using other automated techniques (see, for example, [10]). Jin et al. introduce CFIX [26], a tool that fixes bugs in concurrent C programs by matching bug patterns and proposing fixes. However, in their case, the bug patterns are simple (corresponding to only conjunctions of happens-before constraints) and hence, cannot infer synchronization such as barriers. On the other hand, the CFIX tool is very robust and practical, and has generated fixes and corresponding patches for real open-source libraries.
Bug summarization. While there have been various techniques for fault localization, error explanation, counterexample minimization and bug summarization for sequential programs, we restrict our attention to relevant works for concurrent programs. In [28], 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 [25], 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 [18, 29]. 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 [17, 32, 34, 43].
Accelerating CEGAR. There are several works to enhance the CEGAR loop by finding better predicates, e.g. [4, 36]. In the setting of hardware model-checking (for circuits), Glusman et al. [21] 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. [42] improve the CEGAR by introducing a technique to eliminate all spurious counterexamples for an invariant. Sakunkonchak et al. [33] apply CEGAR optimizations 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. [6] use predicates obtained from CEGAR to guide bounded-model checking (BMC) and extend its reach.

7. Concluding Remarks

We propose a representation for concurrent trace sets based on HB-formulas. We present a method and a tool TARA for generating succinct representations of sound overapproximations of good and bad neighbourhoods of a trace. We use TARA to successfully drive three applications in concurrent program reasoning — synchronization synthesis, bug summarization and CEGAR. We believe that our representation and algorithms can significantly boost the applicability and utility of trace-based techniques for concurrency.

While the initial experiments using TARA have been promising, there are several avenues for future work. We plan to extend TARA to infer synchronization from traces over different set of events. In the bug summarization domain, we plan to add a larger class of bug inference rules. For accelerating CEGAR based verification, we plan to implement a more efficient refinement procedure and explore other sampling rules for picking abstract counterexamples.

Acknowledgements. The authors would like to thank Jyotirmoy Deshmukh, Roderick Bloem, and Bettina Könighofer for enabling fruitful discussions. The authors would also like to thank the anonymous reviewers for their considerable, constructive feedback.

References

[1] A. Albarghouthi and K. McMillan. Beautiful interpolants. In *CAV*, pages 313–329, 2013.

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

[3] D. Beyer. Status report on software verification. In *TACAS*, pages 373–388. 2014. `http://sv-comp.sosy-lab.org/`.

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

[5] D. Beyer and E. Keremoglu. CPACHECKER: A tool for configurable software verification. In *CAV*, pages 184–190, 2011. `http://cpachecker.sosy-lab.org/`.

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

[7] N. Bjørner, K. McMillan, and A. Rybalchenko. On solving universally quantified horn clauses. In *SAS*, pages 105–125, 2013.

[8] P. Černý, T. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. Efficient synthesis for concurrency by semantics-preserving transformations. In *CAV*, pages 951–967, 2013.

[9] P. Černý, T. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. Regression-free synthesis for concurrency. In *CAV*, pages 568–584. 2014. `https://github.com/thorstent/ConRepair`.

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

[11] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, pages 168–176, 2004. `http://www.cprover.org/cbmc/`.

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

[13] E. M. Clarke and E. A. Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*. Springer, 1982.

[14] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[15] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *CACAS*, pages 337–340. 2008. `http://z3.codeplex.com/`.

[16] A. Donaldson, A. Kaiser, D. Kroening, and T. Wahl. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In *CAV*, pages 356–371, 2011.

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

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

[19] A. Farzan, A. Holzer, N. Razavi, and H. Veith. Con2colic testing. In *FSE*, pages 37–47, 2013.

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

[21] 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.

[22] A. Gupta, C. Popeea, and A. Rybalchenko. Solving recursion-free horn clauses over li+uif. In *APLAS*, pages 188–203, 2011.

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

[24] J. Huang, P. Meredith, and G. Roşu. Maximal sound predictive race detection with control flow abstraction. In *PLDI*, pages 337–348, 2014.

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

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

[27] 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.

[28] S. Kashyap and V. Garg. Producing short counterexamples using "crucial events". In *CAV*, pages 491–503. 2008.

[29] 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.

[30] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. In *TOPLAS*, pages 68–93, 1984.

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

[32] 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.

[33] T. Sakunkonchak, S. Komatsu, and M. Fujita. Using counterexample analysis to minimize the number of predicates for predicate abstraction. In *ATVA*, pages 553–563. 2007.

[34] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. on Computer Systems (TOCS)*, pages 391–411, 1997.

[35] K. Sen, G. Roşu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *FMOODS*, pages 211–226, 2005.

[36] R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *CAV*, pages 71–87, 2012.

[37] N. Sinha and C. Wang. On Interference Abstractions. In *POPL*, pages 423–434, 2011.

[38] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. *ACM SIGPLAN Notices*, pages 387–400, 2012.

[39] M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, pages 327–338, 2010.

[40] M. T. Vechev, E. Yahav, and G. Yorsh. Inferring synchronization under limited observability. In *TACAS*, pages 139–154, 2009.

[41] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *FM*, pages 256–272. 2009.

[42] 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.

[43] C. Wang, R. Limaye, M. Ganai, and A. Gupta. Trace-based symbolic analysis for atomicity violations. In *TACAS*, pages 328–342. 2010.