On Interference Abstractions

Nishant Sinha
NEC Laboratories America
nishants@nec-labs.com

Chao Wang
NEC Laboratories America
chaowang@nec-labs.com

Abstract

Interference is the bane of both concurrent programming and analysis. To avoid considering all possible interferences between concurrent threads, most automated static analysis employ techniques to approximate interference, e.g., by restricting the thread scheduler choices or by approximating the transition relations or reachable states of the program. However, none of these methods are able to reason about interference directly. In this paper, we introduce the notion of *interference abstractions* (IAs), based on the models of shared memory consistency, to reason about interference efficiently. IAs differ from the known abstractions for concurrent programs and cannot be directly modeled by these abstractions. Concurrency bugs typically involve a small number of unexpected interferences and therefore can be captured by small IAs. We show how IAs, in the form of both over- and under-approximations of interference, can be obtained syntactically from the axioms of sequential consistency. Further, we present an automatic method to synthesize IAs suitable for checking safety properties. Our experimental results show that small IAs are often sufficient to check properties in realistic applications, and drastically improve the scalability of concurrent program analysis in these applications.

Categories, Subject Descriptors: D.2.4 [Software/Program Verification]: Model Checking, Formal Methods.

General Terms: Algorithms, Verification.

1. Introduction

Analyzing shared memory concurrent programs is difficult due to the fact that constituent program threads may *interfere* with each other via shared variables. Multiple formalisms have been developed to model and reason about interference, e.g., the Mazurkiewicz traces [1] model the program behaviors as a partial order over events while the *context-switching* model utilizes a scheduler to generate all possible thread interleavings. Because analyzing all possible interferences is intractable in practice, these models employ *reduction* techniques to focus on a subset of interferences, e.g., partial-order reduction [2–6] or context-bounding [7–9].

In this paper, we develop a new formalism based on *memory consistency* models [10] for analyzing shared memory concurrent programs efficiently. A memory consistency (MC) model prescribes rules on when a read to a shared location may observe some write to the same location, and hence determines the set of feasible executions of a concurrent program. Concurrent program analysis based on MC models has received considerable attention recently, both for the *relaxed* MC models which allow instruction reorderings [11, 12] by the compiler or hardware, as well as for high-level static analysis [13]. Using a MC model is particularly attractive because it allows us to reason about correlations between reads and writes of program threads directly, inside a reasoning framework for partially-ordered events, similar to Mazurkiewicz traces.

In this work, we focus on the well-known MC model of *sequential consistency* (SC) [10, 14], which is both intuitive as well as simpler to analyze. The SC model prescribes the following rules (more formally, axioms in first-order logic) of interference for correct program executions: (i) each read must observe some write to the same shared location (interfering write), and (ii) a read may only observe the *last* such write in the causal order. Note, however, that enforcing these rules for all reads and corresponding interfering writes in a program, again leads to an intractable analysis.

To obtain a scalable analysis, we introduce *interference abstraction* (IA), a new concurrency abstraction to reason about thread interference using the memory consistency models. One way to obtain an IA is by *weakening* the SC rules, e.g., by permitting a read to not observe any interfering write. In other words, we allow program executions where the read may obtain any value independent of the values of the interfering writes. This form of weakening leads to an *over*-approximation of interference (denoted as an OIA). Alternatively, we may *strengthen* the SC rules, e.g., by forcing a read to observe only a *subset* of all possible interfering writes. Consequently, executions where the read may observe writes outside the subset are ruled out. Such strengthening leads to an *under*-approximation of interference (denoted as an UIA). Employing either of these approximations (obtained by weakening or strengthening the SC rules) makes analysis more tractable. Intuitively, an IA enforces *fewer* dependency relationships among reads and writes than ordained by the SC rules. Many concurrent safety errors, e.g., data races, deadlocks and atomicity violations, typically occur due to a small amount of unexpected interference between threads [15]. This fact is captured formally by IAs, i.e., there often exist *small* IAs sufficient to detect these errors.

We present a formal framework to characterize IAs in an uniform manner by exploiting the axiomatic formulation of SC rules. More precisely, we show how to obtain a wide variety of IAs in a *syntactic* manner by selectively instantiating the SC axioms. The framework models not only OIA's and UIA's but also mixed IAs (MIA's) containing both over- and under-approximations of interference, in a seamless manner. Further, we show how several informal notions of concurrency abstractions for checking properties can be formally captured in the uniform framework of IAs.

Given our unified framework of IAs, the key problem is to synthesize IAs suitable for checking a property. To this goal, we present an iterative refinement scheme which starts from a coarse IA and gradually refines it in a property-driven manner. Most traditional abstraction/refinement schemes (e.g. [4, 16–19]) work with either over- or under-approximations to obtain proofs or witnesses respectively. In contrast, our algorithm works directly on a mixed approximation, i.e., mixed IAs, and iteratively steers the mixed IA to an OIA (if a proof exists) or an UIA (if a witness exists).

We implemented our approach in the FUSION platform, which is a collection of tools for concurrent program verification (e.g. [13, 20–23]). We evaluated the effectiveness of IAs for checking embedded assertions and data races on medium sized programs. Our results show that small IAs are sufficient to decide many of the properties, and our iterative refinement procedure enables drastically improved analysis of these benchmarks. Further, we have been able to check larger benchmarks that were intractable without using IAs.

To sum up, this paper makes the following contributions:

- We introduce the notion of *interference abstraction* to reason about interferences based on memory consistency models. These IAs may over- or under-approximate the thread interference, or represent their combination. (Sec. 5). We show that these IAs formally capture common concurrency bug patterns and as well as correctness proofs (Sec. 5.3).
- We present a unified framework for obtaining these IAs from the axioms of sequential consistency [10, 14] (Sec. 4, 5). The framework of IAs provides a flexible mechanism for approximating interference among reads and writes, guided by the memory consistency axioms.
- We formalize the set of IAs as a complete lattice and present an iterative approach to synthesize IAs for checking properties based on a symbiotic combination of over- and under-approximate IAs (Sec. 6). A set of focusing heuristics are also presented to make the iterative algorithm practical.

2. Preliminaries

We start with formalizing concurrent programs.

2.1 Concurrent Programs

A *concurrent program* consists of a finite set of *threads* $T_0, \ldots, T_k$ communicating via a set SV of *shared variables*. Each thread T_i has a set of local variables LV_i and is represented by a control flow graph (defined below). Threads are allowed to fork other threads in a bounded manner, i.e., the total number of threads is finite. Let T_0 denote the main thread and $V_i = SV \cup LV_i$ denote the set of variables accessible to thread T_i.

We represent a concurrent program using a concurrent control flow graph (CCFG), which may be viewed as an extension of control flow graphs (CFGs) for sequential programs. A CCFG C=(N, E) consists of a set of nodes N and a set of edges E. We use two special types of nodes $fork$ and $join$ to model thread creation and thread join respectively. A program thread T_i corresponds to a sub-graph (N_i, E_i) of the CCFG, where N_i consists of nodes representing program locations in thread T_i and E_i consists of edges representing the program statements. Assume that N_i contains unique *entry* and *exit* nodes of T_i. For each T_i ($i \neq 0$) the entry node has a single incoming edge from a $fork$ node and the exit node has a single outgoing edge to a $join$ node.

Each edge in E_i is labeled by one of the following actions:

- guarded assignment $(\mathsf{assume}(c), asgn)$, where c is a condition over V_i, and $asgn = \{w := exp\}$ is a set of parallel assignments, where $w \in V_i$ and exp is an expression over V_i. Intuitively, the assignments proceed iff condition c is true.
- $fork(j)$, where $0 < j \leq k$ and $j \neq i$, starts the execution of child thread T_j.
- $join(j)$, where $0 < j \leq k$ and $j \neq i$, waits for child thread T_j to terminate.
- $\mathsf{assert}(c)$, where c is a condition over V_i, asserts c.

```
Thread T0

     int x = 0;
     int y = 0;
     pthread_t t1, t2;
     main() {
t1       pthread_create(&t1,0,foo,0);
t2       pthread_create(&t2,0,bar,0);
t3       pthread_join(t2,0);
t4       pthread_join(t1,0);
t5       assert( x != y);
     }
```

```
Thread T1

       foo() {
           int a;
t11        a=y;
t12        if (a==0) {
t13            x=1;
t14            a=x+1;
t15            x=a;
t16        }else
t17            x=0;
t18  }
```

```
Thread T2

       bar() {
           int b;
t21        b=x;
t22        if (b==1) {
t23            y=1;
t24            b=y+1;
t25            y=b;
t26        }else
t27            y=0;
t28  }
```

Figure 1. A multi-threaded C program with an assertion.

By defining expressions suitably and using code transformations, the above formulation can model all statements in standard programming languages like Java and multi-threaded C. The details on modeling generic language constructs such as pointers and structures are omitted since they are not directly related to concurrency. For more information on language modeling, please refer to recent efforts including [24–26].

The guarded assignment action $(\mathsf{assume}(c), asgn)$ may have the following variants: (1) when $c = \mathtt{true}$, it represents normal assignments; (2) when the set $asgn$ is empty, $assume(c)$ itself can represent the `then`-branch of an `if(c)-then-else` statement, while $assume(\neg c)$ can represent the `else`-branch; and (3) with both guard and assignments, it represents an atomic *check-and-set*, which can be used as the foundation of all kinds of synchronization primitives. For example, acquiring the lock lk in thread T_i is modeled as $(\mathsf{assume}(lk = \bot), \{lk := i\})$ and releasing the lock is modeled as $(\mathsf{assume}(lk = i)\{lk := \bot\})$. Here the value of lk indicates the lock owner's thread index ($\bot$ means the lock is free). Similarly, acquiring the counting semaphore se is modeled as $(\mathsf{assume}(se > 0), \{se := se - 1\})$.

2.2 Bounded Concurrent Programs

Static analysis of concurrent programs with loops and/or recursion is known to be undecidable even with finite data. Our goal, however, is to analyze real-life programs where data structures are precisely modeled. We, therefore, focus on analyzing *bounded* concurrent programs, whose analysis is decidable. Intuitively, a *structurally* bounded program is obtained by finitely unwinding the loops and recursion in an arbitrary real-life concurrent program. Bounded programs are also obtained in the context of symbolic predictive analysis (e.g., [21, 22]), by generalizing from the sequence of program statements executed in a particular trace. This form of bounding finitizes the number of program threads and the heap. Further, if the underlying program theory is decidable, then the analysis becomes decidable. A bounded program under-approximates the sets of paths of the original program and hence the violations found by the analysis are real. However, the proofs (absence of violations) found may not generalize to the original program.

We represent bounded programs using CCFGs. For ease of presentation, we assume that all function calls in the program have been inlined. However, the presented technique can be directly extended to handle function calls modularly [13].

Example. Fig. 1 shows an example of a multi-threaded C program with two shared variables x and y. The main thread T_0 creates threads T_1 and T_2, which in turn execute functions `foo` and `bar`, respectively. Thread T_0 waits for T_1, T_2 to terminate and join back, before asserting $(x \neq y)$. Here `pthread_create` and `pthread_join` are routines in *PThreads* library, directly corresponding to *fork/join* in our model. (Since this particular example does not have loops and recursion, the bounded and the original programs are same.)

The assertion at t_5 defines the correctness property, which holds in some, but not in all, execution traces of the program. In particular, the execution trace $\rho = (t_1 t_2)(\{t_{11}–t_{15}\}t_{18})(t_{21} t_{26} t_{27} t_{28})\{t_3–t_5\}$ does not violate the assertion ($x = 2, y = 0$ at t_5), whereas the execution trace $\rho' = (t_1 t_2)(\{t_{11}\text{-}t_{14}\})(t_{21} - t_{25} t_{28})(t_{15} t_{18})\{t_3\text{-}t_5\}$ violates the assertion ($x = 2, y = 2$ at t_5).

2.3 Gated Single Static Assignment

Recall that a sequential program can be encoded in a standard manner using the gated single static assignment (GSA) form [27], which combines the classic single static assignment (SSA) form (each variable is defined exactly once) with conditions under which a particular definition of a variable may reach a *join* node. For example, consider the following C code:

```
l1 : if (c1) { z = 1; }    // z1
l2 : else z = 2;           // z2
l3 : y = z + 3;            // y1
```

The SSA form renames the writes to z at l_1 and l_2 in terms of new *definitions* of z_1 and z_2, respectively. The *use* of z in l_3 is then rewritten using the ϕ function as $\phi(z_1, z_2)$, which, by definition, may evaluate to either z_1 or z_2. Consequently, l_3 is rewritten as $y_1 = \phi(z_1, z_2) + 3$, where y_1 is a fresh definition of y. Note that the ϕ operator does not contain information about the conditions under which definition z_1 or z_2 may be chosen, and therefore cannot be used for precise encoding of the bounded program. The GSA representation solves the problem by replacing $\phi(z_1, z_2)$ with $ite(c_1, z_1, z_2)$ where ite stands for the *if-then-else* operator. That is, $y_1 = z_1 + 3$ when condition c_1 is true; otherwise $y_1 = z_2 + 3$.

3. Symbolic Analysis of Bounded Programs

We say that two memory accesses interfere if both access the same memory location and at least one of them is a write. To avoid worrying about whether the accesses are concurrent or not, we use the term *interference* in a generic manner, both for access pairs occurring in the same thread or occurring concurrently.

In order to check properties of a bounded CCFG C, we encode it as a first-order logic formula in a step-wise manner. First the program statements in C are encoded in an *interference-modular* manner by ignoring the interference between all reads and writes (denoted Φ_C). The read-write interference in C is then encoded using sequential consistency axioms (denoted Π), which corresponds to composing the program threads. The property, e.g., existence of an assertion violation, data race, or atomicity violation, is encoded as a formula Φ_{PRP}. The combined formula

$$\Phi := \Phi_C \wedge \Pi \wedge \Phi_{PRP}$$

is then checked for satisfiability using an off-the-shelf constraint solver, e.g., an SMT solver [28, 29]. The formula Φ is satisfiable iff there exists an execution of the program that violates the property.

3.1 Interference-Modular Encoding

We show how to encode all the edges of the CCFG C without modeling the interference between the global reads and writes. The encoding in this section is similar to our previous works [13, 21]; we review the main details here. Although the GSA form can encode sequential programs (cf. Sec. 2.3), it cannot directly encode a program thread in a concurrent context. This is due to possibility of *interference* on shared variables by concurrent threads, i.e., a read of a shared variable (a global read, in short) must take into account all possible interferences from concurrent writes. To ignore modeling such interferences, each global read of variable z is assigned a fresh symbolic value rz (also called a *placeholder*). Using these fresh values, we can now encode the threads in the CCFG using the standard GSA encoding [1].

We say that a program edge is *global* if it accesses a shared variable; otherwise it is *local*. Both local and global edges participate in the data flow inside a thread; however, only global edges participate in data flow across the threads. Hence, we encode the local and global edges separately: this enables us to only consider global edges for encoding the thread interleavings, or more precisely, the interference between threads. Our encoding, denoted by Φ_C, consists of a local component Φ_L and a global component Φ_G:

$$\Phi_C := \Phi_L \wedge \Phi_G$$

We now discuss how Φ_L and Φ_G are obtained.

3.1.1 Encoding Local Edges (Φ_L)

Given the program in the GSA form together with placeholders for global reads, we can encode each program assignment of form $w := exp$ on a local edge as a formula ($w = exp$). The local encoding Φ_L is obtained by conjoining the formula obtained from local edges. Fig. 2 (left) shows the GSA encoding of the running example in Fig. 1. The local variable a is defined in t_{11} and t_{14}. At t_{18} the value of a is either a_1 (defined in t_{11}) or a_2 (defined in t_{14}), depending on the condition $(a_1 \neq 0)$.

3.1.2 Encoding Global Edges(Φ_G)

We first compute the enabling condition for each edge.

Path conditions. The path condition for an edge t_i in the CCFG C is denoted by $g(t_i)$: t_i executes iff $g(t_i)$ is satisfiable. Let t_{first} and t_{last} be the unique first and last edge in C respectively. Starting with $g(t_{first}) := true$, the path conditions are computed iteratively for each t_i via CCFG traversal as follows. We distinguish between CCFG nodes having multiple predecessors: if the predecessors are in the same thread, the node is said to be an *intra-thread* join; otherwise it is an *inter-thread* join.

- If the source of t_i is an *intra-thread join* node with incoming edges t_j and t_k, then $g(t_i) = g(t_j) \vee g(t_k)$.
- If the source of t_i is an *inter-thread join* node with incoming edges t_j and t_k, then $g(t_i) = g(t_j) \wedge g(t_k)$.
- If t_i is a branching statement with condition c and t_j precedes t_i, then $g(t_i) = g(t_j) \wedge c$.
- In all other cases, the source of t_i has a single incoming edge t_j and $g(t_i) = g(t_j)$.

Fig. 2 (center) shows the path conditions in the example CCFG.

Global accesses. Each global edge is encoded using the notion of a *global access*. The global access a for an edge e is a tuple $(Addr, Val, En)$, where $Addr(a)$ is the memory location accessed, $Val(a)$ is the value read or written, and $En(a)$ is the condition under which e is enabled. For example, edge t_{11} (cf. Fig. 2) is encoded as an access $r_{11} = (y, ry_1, g(t_{11}))$. Similarly, edge t_{15} is encoded as a global access $w_{15} = (x, a_1, g(t_{15}))$. The global access a captures all the information about the execution of the corresponding global edge e succinctly.

Interference Skeleton. Observe that to model all interferences in the CCFG C precisely, we not only need the values of global accesses, but also their relative order in C. Let $\sqsubset$ denote the partial order among global accesses induced by the CCFG (called *program order*). For accesses a_1 and a_2, if $a_1 \sqsubset a_2$ holds, then a_1 must happen before a_2 in all program executions. The set of global accesses in C, say RW, together with their program order, denoted

[1] We will later see that introducing such placeholders is, in fact, an instance of interference abstraction (Sec.5).

GSA Form:

$$\begin{array}{ll}
t_0 : x_0{=}0 \wedge y_0{=}0\ [\mathbf{w_{00}}, \mathbf{w_{01}}] & \\
t_1 : & \\
t_2 : & \\
t_{11} : a_1 = ry^1\ [\mathbf{r_{11}}] & t_{21} : b_1 = rx^2\ [\mathbf{r_{21}}] \\
t_{12} : & t_{22} : \\
t_{13} : x_1 = 1\ [\mathbf{w_{13}}] & t_{23} : y_1 = 1\ [\mathbf{w_{23}}] \\
t_{14} : a_2 = rx^1 + 1\ [\mathbf{r_{14}}] & t_{24} : b_2 = ry^2 + 1\ [\mathbf{r_{24}}] \\
t_{15} : x_2 = a_2\ [\mathbf{w_{15}}] & t_{25} : y_2 = b_2\ [\mathbf{w_{25}}] \\
t_{16} : & t_{26} : \\
t_{17} : x_3 = 0\ [\mathbf{w_{17}}] & t_{27} : y_3 = 0\ [\mathbf{w_{27}}] \\
t_{18} : a_3 = ite(a_1 \neq 0, a_1, a_2) & t_{28} : b_3 = ite(b_1 \neq 1, b_1, b_2) \\
t_3 : & \\
t_4 : & \\
t_5 : \text{assert}(rx^3 \neq ry^3)\ [\mathbf{r_{50}}, \mathbf{r_{51}}] &
\end{array}$$

Path Conditions of all Edges:

$$\begin{array}{ll}
g(t_1) = \mathsf{true} & \\
g(t_2) = g(t_1) & \\
g(t_{11}) = g(t_1) & g(t_{21}) = g(t_2) \\
g(t_{12}) = g(t_{11}) \wedge (a_1 = 0) & g(t_{22}) = g(t_{21} \wedge (b_1 = 1) \\
g(t_{13}) = g(t_{12}) & g(t_{23}) = g(t_{22}) \\
g(t_{14}) = g(t_{13}) & g(t_{24}) = g(t_{23}) \\
g(t_{15}) = g(t_{14}) & g(t_{25}) = g(t_{24}) \\
g(t_{16}) = g(t_{11}) \wedge (a_1 \neq 0) & g(t_{26}) = g(t_{21}) \wedge (b_1 \neq 1) \\
g(t_{17}) = g(t_{16}) & g(t_{27}) = g(t_{26}) \\
g(t_{18}) = g(t_{15}) \vee g(t_{17}) & g(t_{28}) = g(t_{25}) \vee g(t_{27}) \\
g(t_3) = g(t_2) \wedge g(t_{28}) & \\
g(t_4) = g(t_3) \wedge g(t_{18}) & \\
g(t_5) = g(t_4) &
\end{array}$$

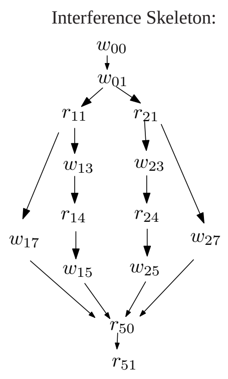


Figure 2. The symbolic encoding of the bounded program in Fig. 1. The global edges are labeled by the corresponding global accesses (e.g., t_{11} by $[\mathbf{r_{11}}]$). Edges t_0 is labeled by write accesses on x ($\mathbf{w_{00}}$)and y ($\mathbf{w_{01}}$) respectively. Edge t_5 is labeled similarly.

$(RW, \sqsubset)$, is called the *interference skeleton* (IS) of C. Fig. 2 (right) shows the IS (as a graph) for the running example: each node corresponds to an access $a = (Addr, Val, En)$ modeling the location, value and the enabling condition respectively, and the edges model the program order. Note that the IS models all the global accesses and their mutual ordering precisely.

To encode the IS $(RW, \sqsubset)$ in first-order logic, we introduce a new type called *Acc*, and the following operators over the type: a *must-happen-before* predicate HB over pairs of *Acc* elements and operators $addr$, val, and en which map an *Acc* element to its location, value and enabling condition, respectively. Now IS is encoded as Φ_G:

$$\Phi_G = \Phi_{Acc} \wedge \Phi_{PO}$$

where Φ_{Acc} encodes the set of accesses RW,

$$\Phi_{Acc} := \bigwedge_{a \in RW} \quad (addr(a) = Addr(a) \wedge en(a) = En(a) \wedge val(a) = Val(a))$$

and Φ_{PO} encodes $\sqsubset$:

$$\Phi_{PO} := \bigwedge_{(a_i, a_j) \in \sqsubset} (HB(a_i, a_j))$$

We also refer to Φ_{PO} as *program order constraints*.

3.2 Encoding Properties

Generic programming errors may be modeled as embedded assertions in the CCFG. The formula Φ_{PRP} then captures the condition under which a given assertion is violated. For an assertion $assert(c)$ in transition t, Φ_{PRP} is defined as

$$\Phi_{PRP} := g(t) \wedge \neg c$$

denoting that the condition c must hold if t is executed. In our running example in Fig. 2, fresh variables $rx^1, rx^2, rx^3, ry^1, ry^2, ry^3$ are added to denote the values of the six global reads, and the property sub-formula is defined as $\Phi_{PRP} := \ g(t_5) \wedge \neg(rx^3 \neq ry^3)$

Besides assertion violations, we can encode standard concurrency errors such as data races and atomicity violations directly as a set of happens-before constraints. Suppose we want to check the three-access atomicity violation [30, 31] involving global accesses c, c' and r, where c and c' are in the same thread and are intended to execute atomically, r is executed in another thread and interferes with both c and c'. (An example of such violation is given later in Fig. 7.) The property formula is defined as follows:

$$\Phi_{PRP} := \ en(c) \wedge en(r) \wedge en(c') \wedge HB(c, r) \wedge HB(r, c')$$

4. Axiomatic Composition

Given the skeleton IS = $(RW, \sqsubset)$ obtained from the interference-modular encoding of the CCFG C, we can now encode the interference (both intra- and inter-thread) using the axioms for sequential consistency (SC) over the set of global accesses RW in C. Intuitively, the SC axioms *link* the read accesses in RW to appropriate write accesses in RW to obtain feasible program executions. The basis of axiomatic composition is the *link* relation.

DEFINITION 1 (Link Relation). *The predicate $link(r, w)$ denotes that the read r observes write w, i.e., the value retrieved by the read access r is the same as the value set by the write access w. The $link$ relation is* exclusive, *i.e.,* $link(r, w) \Rightarrow \forall w' \neq w.\ \neg link(r, w')$.

The SC axioms [11, 13, 32], denoted as Π, can be modeled in typed first order logic using operators HB, $addr$, val and en and quantified variables r, w and w' over type *Acc* (cf. Sec. 3.1). The formula $\Pi := \Pi_1 \wedge \Pi_2 \wedge \Pi_3$, where

$$\begin{array}{l}
\Pi_1 := \forall r\ .\ \exists w\ . \quad en(r) \Leftrightarrow (en(w) \wedge link(r, w)) \\
\Pi_2 := \forall r\ .\ \forall w\ . \quad link(r, w) \Rightarrow HB(w, r) \wedge \\
\qquad (addr(r) = addr(w)) \wedge (val(r) = val(w)) \\
\Pi_3 := \forall r\ .\ \forall w\ .\ \forall w'\ .\ \ (link(r, w) \Rightarrow \\
\qquad (\ en(w') \wedge \neg HB(w', w) \wedge \neg HB(r, w') \Rightarrow \\
\qquad addr(r) \neq addr(w')\ )
\end{array}$$

Formula Π_1 models that if a read r is enabled, then r must be linked to some enabled write w, and vice versa. Formula Π_2 models the data flow and relative order between r and w when r links with w, i.e., both the value and address of r and w must be same and w executes before r. Formula Π_3 says that if r links with w then no other write w' to the *same address* as r should be executed between w and r, i.e., w' executes either before w or after r.

Fig. 3 shows the hierarchical encoding of Φ. When checking properties, the Π axioms interact subtly with the property violation condition Φ_{PRP}: the condition Φ_{PRP} identifies well-formed paths in the CCFG that lead to a property violation and enables the reads and writes along those paths; the axioms Π then make sure that reads and writes along those paths can be appropriately linked to obtain a feasible thread interleaving.

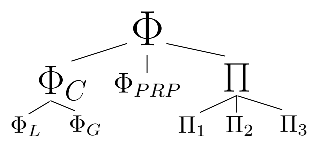


Figure 3. Hierarchical Encoding of the CCFG.

4.1 Full instantiation of SC axioms

Let $\mathbb{R}$ and $\mathbb{W}$ denote the set of all reads and writes in RW respectively. Given $\mathbb{R}$ and $\mathbb{W}$, Π can be encoded directly by instantiating the quantifiers for all reads in $\mathbb{R}$ and writes in $\mathbb{W}$. Recall from

Sec. 3.1 that the values of $addr(a)$, $val(a)$ and $en(a)$ for each access a are already encoded in Φ_G. Additional constraints are required to encode the exclusivity of $link$ and that HB is a strict partial order. Here, we exploit the theory of uninterpreted functions: rewrite $link(r, w)$ as $Id(r)$=$Id(w)$, where Id is an indexing function which maps each access to an unique integer. All writes w are initialized with unique $Id(w)$ values. Similarly, rewrite $HB(a, b)$ as $Clk(a) < Clk(b)$, where the Clk function assigns an integer time-stamp to each access and the operator $<$ encodes the partial order over integer time-stamps.

Example. The SC constraints for the running example (with substituted values for en, val, $addr$ and HB in Π_2 and Π_3) are

$$
\begin{aligned}
\Pi_1 :=\ & (en(r_{14}) \Leftrightarrow \bigvee_{j\in\{00,13,15,17\}}(en(w_j) \wedge link(r_{14}, w_j))\wedge\\
& (en(r_5) \Leftrightarrow \bigvee_{j\in\{01,23,25,27\}}(en(w_j) \wedge link(r'_5, w_j))\wedge\\
& \ldots\\
\Pi_2 :=\ & link(r_{14}, w_{00}) \Rightarrow Clk(w_{00}) < Clk(r_{14}) \wedge rx^1 = 0)\wedge\\
& link(r_{14}, w_{13}) \Rightarrow Clk(t_{13}) < Clk(t_{14}) \wedge rx^1 = 1)\wedge\\
& link(r_{14}, w_{15}) \Rightarrow Clk(t_{15}) < Clk(t_{14}) \wedge rx^1 = a_1)\wedge\\
& \ldots\\
\Pi_3 :=\ & link(r_{14}, w_{00}) \Rightarrow\\
& \qquad \neg(g(t_{13}) \wedge Clk(w_{00}) < Clk(w_{13}) < Clk(r_{14}))\wedge\\
& \qquad \neg(g(t_{15}) \wedge Clk(w_{00}) < Clk(w_{15}) < Clk(r_{14}))\wedge\\
& \qquad \neg(g(t_{17}) \wedge Clk(w_{00}) < Clk(w_{17}) < Clk(r_{14}))\wedge\\
& \ldots
\end{aligned}
$$

Let $\widehat{\Pi}$ denote the quantifier-free formula obtained by instantiating Π for all accesses in $\mathbb{R}$ and $\mathbb{W}$. Note that the number of constraints in the worst case is cubic in the sizes of $\mathbb{R}$ and $\mathbb{W}$.

The following theorem captures the idea that the solutions of $\widehat{\Pi}$ correspond to the feasible executions of the program that violate the checked property.

THEOREM 1. *[13] Suppose we have an encoding Φ_C of a bounded CCFG C and a property encoding Φ_{PRP}. The formula $\Phi := \Phi_C \wedge \widehat{\Pi} \wedge \Phi_{PRP}$ is satisfiable iff there exists a feasible execution of C which violates the property.*

For convenience, if Φ is satisfiable then we say that a witness exists; otherwise we say that a (unsatisfiability) proof exists. Instead of always referring to Φ for a given CCFG and a property, we say $\widehat{\Pi}$ is satisfiable (has a witness) or is unsatisfiable (has a proof).

We now characterize the satisfying models of Π using the notion of a *read-write match* and an *interference relation*.

DEFINITION 2 (Read-Write Match). *Given set of reads $\mathbb{R}$ and writes $\mathbb{W}$, a* read-write match *(match, in short) is a partial function $M : \mathbb{R} \to \mathbb{W}$.*

DEFINITION 3 (Interference Relation (IR)). *An* interference relation *is a tuple $(R_\mathcal{I}, W_\mathcal{I}, M, \sqsubset)$ where $R_\mathcal{I} \subseteq \mathbb{R}$, $W_\mathcal{I} \subseteq \mathbb{W}$, M is a match and $\sqsubset$ is a partial order over the set $\mathbb{R} \cup \mathbb{W}$.*

DEFINITION 4 (IR Satisfying $\widehat{\Pi}$). *If $\widehat{\Pi}$ is satisfiable with a model Θ, then there exists an unique IR $\mathcal{I} = (R_\mathcal{I}, W_\mathcal{I}, M, \sqsubset)$* satisfying *$\widehat{\Pi}$ defined as follows:*

- *$R_\mathcal{I}$ and $W_\mathcal{I}$ are respectively the enabled reads and writes in Θ, i.e., ($r \in R_\mathcal{I} \Leftrightarrow en(r)$) and ($w \in W_\mathcal{I} \Leftrightarrow en(w)$).*
- *$M : R_\mathcal{I} \to W_\mathcal{I}$ is well-defined for all $r \in R_\mathcal{I}$ with co-domain $W_\mathcal{I}$. Moreover, $M(r) = w$ iff $link(r, w)$ holds in Θ.*
- *$\sqsubset = \{(a, b) \mid a, b \in (R_\mathcal{I} \cup W_\mathcal{I}) \wedge HB(a, b)$ holds in Θ $\}$.*

Intuitively, if $\widehat{\Pi}$ is satisfiable, then we can extract a match M by recording which of the $link(r, w)$ predicates hold in the current solution and the partial order $\sqsubset$ induced by the HB predicate. Note that the exclusivity of the $link$ constraints ensures that M is a well-defined function.

For the running example in Fig. 2, $\widehat{\Pi}$ has a satisfying IR $\mathcal{I}$ = $(R_\mathcal{I}, W_\mathcal{I}, M, \sqsubset)$ where $R_\mathcal{I} = \{r_{11}, r_{14}, r_{50}, r_{51}, r_{21}, r_{24}\}$, $W_\mathcal{I} = \{w_{00}, w_{01}, w_{13}, w_{15}, w_{23}, w_{25}\}$, $M = \{(r_{11}, w_{01}), (r_{14}, w_{13}), (r_{50}, w_{15}), (r_{21}, w_{13}), (r_{24}, w_{23}), (r_{51}, w_{25})\}$ and $\sqsubset$ consists of program order constraints (Φ_{PO} in Sec. 3.1) together with $(w_{13} \sqsubset r_{21})$ and $(r_{21} \sqsubset w_{15})$. Note that w_{17} and w_{27} are disabled and hence not in $W_\mathcal{I}$. An execution trace $\rho = (t_1 t_2)(\{t_{11} - t_{14}\})(t_{21} - t_{25}\ t_{28})(t_{15} t_{18})\{t_3\text{-}t_5\}$ corresponds to $\mathcal{I}$ and violates the assertion.

4.2 Redundant Instantiation of SC axioms

As mentioned earlier, Π may give rise to a large number (cubic in the reads/writes) of constraints; in practice, many of these are redundant. For example, if a read r precedes write w in the program order, instantiating Π for $link(r, w)$ is wasteful and can be avoided. Redundant constraints also occur in a more obscure manner, e.g., suppose the given property can be violated by executing the program threads sequentially without interleaving them. In this case, most constraints linking reads in one thread to concurrent writes in another thread are unnecessary for checking the property. Pruning redundant constraints is the key to making our problem tractable.

Interestingly, the syntactic formulation of Π offers insights on how to prune redundant constraints. Let us consider a few pruning methods. Suppose, for example, we pick a subset of reads and writes, say $R \subseteq \mathbb{R}$ and $W \subseteq \mathbb{W}$ respectively, and instantiate Π_2 only for R and W to obtain, say, Π^+. Note that Π^+ is an *over-approximation* of $\widehat{\Pi}$ because $\widehat{\Pi} = \Pi^+ \wedge F$, where F corresponds to the pruned constraints. Hence, a witness to Π^+ may not correspond to any feasible program execution; however, a proof for Π^+ implies that the property is never violated in P. On pruning constraints as above, we are able to *over-approximate* the interference, and hence the program behaviors axiomatically. Clearly, such an over-approximation is cheaper to check if we can discover small R and W sets, which are sufficient for proving the absence of violation.

Consider another form of pruning, again based on the syntactic structure of the axioms. Observe that instantiating Π_1 leads to a disjunction of $link(r, w)$ formula for each read $r \in \mathbb{R}$ and write $w \in \mathbb{W}$. If $\mathbb{W}$ is large, we must explore a large number of choices to find an appropriate read-write match. This naturally leads to another approximation: pick a subset of writes $W \subseteq \mathbb{W}$ for each read r and instantiate Π_1 (similarly Π_2 and Π_3) only for pairs (r, w) where $w \in W$. By pruning the disjunctions, we obtain an *under-approximation* of the interference (denoted by Π^-) which is cheaper to analyze. This under-approximation preserves witnesses, i.e., a witness found using Π^- corresponds to a concrete witness in the program P. Again, the usefulness of Π^- depends on obtaining a small W sufficient for computing a witness.

Each of the approximations above relies on either weakening or strengthening the SC axioms by *decoupling* reads and writes; hence we refer to them as *interference abstractions* (IAs). The above two examples show how the syntactic structure of the SC axioms may be exploited to obtain approximations (under- or over-) of thread interference. In fact, the abundance of quantifiers in Π allows us to build a complex array of abstractions systematically, where the under- and over-approximations of interference are intricately combined. We now define interference abstractions obtained in this manner formally.

5. Interference Abstractions

To formally represent the interference abstractions, we first define the *link set* for a read.

DEFINITION 5 (Link Set). *Given a set of reads $R \subseteq \mathbb{R}$, let $\mathcal{W} : R \to 2^{\mathbb{W}}$ map each read $r \in R$ to a set of writes which r may link with. We say that $\mathcal{W}(r)$ is the link set of r.*

Let $\widehat{\mathcal{W}}$ denote the default link set such that $\widehat{\mathcal{W}}(r)$ contains all possible writes that r may link with statically. For example, in Fig. 2,

$\widehat{\mathcal{W}}(r_{14}) = \{w_{00}, w_{13}, w_{15}, w_{17}\}$ because r_{14} reads variable x. Let Λ and Σ denote the (r, w) pairs and (r, w, w') triplets for which Π_2 and Π_3 are instantiated respectively. Now, we can reformulate the SC axioms Π as follows.

$$\begin{aligned}
\Pi_1 &:= \forall r \in R \,.\, \exists w \in \mathcal{W}(r) \,.\ \ \phi_1(r, w)\\
\Pi_2 &:= \forall (r, w) \in \Lambda \,.\ \ \phi_2(r, w)\\
\Pi_3 &:= \forall (r, w, w') \in \Sigma \,.\ \ \phi_3(r, w, w')\\
\Pi &:= \Pi_1 \wedge \Pi_2 \wedge \Pi_3
\end{aligned}$$

where

$$\begin{aligned}
\phi_1(r, w) :=\ & en(r) \Leftrightarrow (en(w) \wedge link(r, w))\\
\phi_2(r, w) :=\ & link(r, w) \Rightarrow (HB(w, r) \wedge\\
& addr(r) = addr(w) \wedge val(r) = val(w))\\
\phi_3(r, w, w') :=\ & link(r, w) \Rightarrow\\
& (en(w') \wedge \neg HB(w', w) \wedge \neg HB(r, w')\\
& \Rightarrow addr(r) \neq addr(w'))
\end{aligned}$$

To model Λ and Σ, we introduce functions λ and σ as follows.

$$\lambda(R, \mathcal{W}) = \{(r, w) \mid r \in R, w \in \mathcal{W}(r)\}$$

$$\sigma(R, \mathcal{W}, \mathcal{W}') = \{(r, w, w') \mid r \in R, w \in \mathcal{W}(r), w' \in \mathcal{W}'(r)\}$$

Given R, $\mathcal{W}$ and $\mathcal{W}'$ the functions λ and σ model the set of (r, w) pairs and (r, w, w') triples that Π_2 and Π_3 are instantiated for, respectively. The complete instantiation $\widehat{\Pi}$ of Π corresponds to $R = \mathbb{R}$, $\mathcal{W} = \widehat{\mathcal{W}}$, $\Lambda = \lambda(R, \widehat{\mathcal{W}})$ and $\Sigma = \sigma(R, \widehat{\mathcal{W}}, \widehat{\mathcal{W}})$.

5.1 Syntactic IAs

An interference abstraction (IA) is characterized by an *incomplete* instantiation of Π axioms. We say that the IAs thus obtained are *syntactic* because they are obtained by restricting the instantiations of SC axioms syntactically.

DEFINITION 6 (Syntactic Interference Abstraction (IA)). *Given Π as above and the set of all global reads $\mathbb{R}$, an interference abstraction α is defined to be a tuple $\langle R, \mathcal{W}, \Lambda, \Sigma\rangle$ such that: (i) $R \subseteq \mathbb{R}$, (ii) $\mathcal{W} \subseteq \widehat{\mathcal{W}}$, (iii) $\Lambda \subseteq \lambda(R, \widehat{\mathcal{W}})$, and (iv) $\Sigma \subseteq \sigma(R, \widehat{\mathcal{W}}, \widehat{\mathcal{W}})$.*

We say that α is a *proper* IA if at least one of R, $\mathcal{W}$, Λ or Σ is a proper subset. We refer to R, $\mathcal{W}$, Λ and Σ as components or dimensions of α. The *size* of α is defined to be the sum of the sizes of components of α.

Each IA $\alpha = (R, \mathcal{W}, \Lambda, \Sigma)$ corresponds to an instantiation of Π, denoted by Π^α, consisting of sub-formulas Π_1^α, Π_2^α and Π_3^α. Intuitively, an IA corresponds to first fixing a set of reads R and the set of writes $\mathcal{W}(r)$ that each read $r \in R$ may link to, and then instantiating Π_1 for each (r, w) in $\lambda(R, \mathcal{W})$, and Π_2 and Π_3 for subsets of $\lambda(R, \widehat{\mathcal{W}})$ and $\sigma(R, \widehat{\mathcal{W}}, \widehat{\mathcal{W}})$ respectively. Note that replacing read values by placeholders (during interference-modular encoding, cf. Sec. 3.1) is the coarsest IA, where a read does not link with any write, i.e., may assume an arbitrary value.

An IA α is said to be *under*-approximate (UIA) iff $\Pi^\alpha \Rightarrow \widehat{\Pi}$. Similarly, α is said to be *over*-approximate (OIA) iff $\widehat{\Pi} \Rightarrow \Pi^\alpha$. Otherwise we say that α is a *mixed* IA (MIA). As expected, an UIA is useful for obtaining witnesses while an OIA helps obtain proofs. We now show how the OIA and UIA as defined above in a semantic manner can be obtained *syntactically*.

DEFINITION 7 (Syntactic OIA). *An IA $\alpha = (R, \mathcal{W}, \Lambda, \Sigma)$ is an* OIA *iff $\mathcal{W} = \widehat{\mathcal{W}}$.*

In other words, an OIA is obtained if Π_1 is instantiated for all writes in the default link set $\widehat{\mathcal{W}}(r)$ of each read $r \in R$. However, Π_1 need not be instantiated for all reads and Π_2/Π_3 may not be instantiated for all reads and writes. The following lemma shows that a syntactic OIA is also a *semantic* OIA and hence preserves proofs.

LEMMA 1. *Given a syntactic* OIA *α, $\widehat{\Pi} \Rightarrow \Pi^\alpha$.*

Syntactic UIA's are more tricky to define.

DEFINITION 8 (Syntactic UIA). *An IA $\alpha = (R, \mathcal{W}, \Lambda, \Sigma)$ is an* UIA *iff (i) $R = \mathbb{R}$, (ii) $\Lambda = \lambda(R, \mathcal{W})$ and (iii) $\Sigma = \sigma(R, \mathcal{W}, \widehat{\mathcal{W}})$.*

Intuitively, an UIA is obtained if Π_1 is instantiated for all reads $r \in \mathbb{R}$ but only for a subset $\mathcal{W}$ of the default link set of each read. Moreover, Π_2 must be instantiated for all pairs $(r, w) \in \lambda(R, \mathcal{W})$ and Σ consists of all triples (r, w, w') where w is drawn from $\mathcal{W}(r)$ but w' ranges over the default set $\widehat{\mathcal{W}}(r)$. All the above constraints are critical for constructing an UIA which preserves witnesses.

LEMMA 2. *Given a syntactic* UIA *$\alpha = (R, \mathcal{W}, \Lambda, \Sigma)$, $\Pi^\alpha \Rightarrow \widehat{\Pi}$.*

A subtle difference between an OIA $\alpha = (R, \mathcal{W}, \Lambda, \Sigma)$ and an UIA $\beta = (R', \mathcal{W}', \Lambda', \Sigma')$ must be noted. Suppose for some $r \in \mathbb{R}$, $r \notin R$ in α. In the UIA β, $R' = \mathbb{R}$ and hence $r \in R'$; however, let $\mathcal{W}'(r)$ be empty. Although these two cases appear similar, they correspond to different instantiations of Π. For the OIA α, Π_1 will not be instantiated for r at all; however, for the UIA β, Π_1 will be instantiated as $\neg en(r)$. In other words, not including a read r in an OIA allows r to be enabled in any witness of Π^α without linking to any write; in contrast, setting $\mathcal{W}'(r) = \emptyset$ in an UIA amounts to disabling r in all witnesses of Π^β.

A full instantiation $\widehat{\Pi}$ of Π corresponds to both an OIA and an UIA. Also, an IA $\alpha = (R, \mathcal{W}, \Lambda, \Sigma)$ is an mixed IA (MIA), if α is neither an OIA nor an UIA. In particular, if $R \neq \mathbb{R}$ and $\mathcal{W} \neq \widehat{\mathcal{W}}$, then α is an MIA. Intuitively, in an MIA, some reads may not be linked to any writes while other reads may link to a restricted set of writes. We next discuss how to visualize the set of syntactic IAs.

5.2 Visualizing Syntactic IAs

As mentioned above, an IA α corresponds to instantiating Π only for a subset of all possible reads and/or writes. Note that even though Π_1 is instantiated for $\lambda(R, \mathcal{W})$, Π_2 and Π_3 may be instantiated independent of Π_1, for different subsets of reads and writes from R and $\mathcal{W}$. The possible choice of read and write subsets gives rise to a complex space of IAs. To better visualize this space, consider Fig. 4 which shows the components as four independent dimensions, R, $\mathcal{W}$, Λ and Σ. Intuitively, the $\mathcal{W}$ dimension corresponds to an under-approximation, while the other dimensions correspond to over-approximations. Moving away from the center along any dimension corresponds to *reducing* the approximation corresponding to that dimension. Note also that the dimensions are somewhat inter-dependent, i.e., it makes sense to select values for Λ and Σ only after we finish selecting the values for R and $\mathcal{W}$.

The IA space shown in Fig. 4 is, in fact, even more complex, e.g., we may instantiate only the order constraints $HB(w, r)$ in Π_2 without instantiating the data flow constraints $val(r)$=$val(w)$. Consequently, we may obtain even more fine-grained OIA's, corresponding to sub-dimensions of the Λ-axis in the figure. We show later that the space of syntactic IAs directly corresponds to a complete lattice. We now show that these syntactic IAs indeed have a practical significance, i.e., they correspond to some common semantic notions useful for reasoning about concurrent programs.

5.3 Useful Semantic Interference Abstractions

Many popular techniques for reasoning about concurrent programs incorporate some form of *semantic* interference abstraction. Many of these abstractions can be readily modeled by our syntactic IAs.

Let us first examine the meaning of instantiating Π incompletely. Not instantiating Π_1 for some read r and write w disallows linking r with w in any execution. Formula Π_2 corresponds to *local* consistency: not instantiating Π_2 for some (r, w) implies that

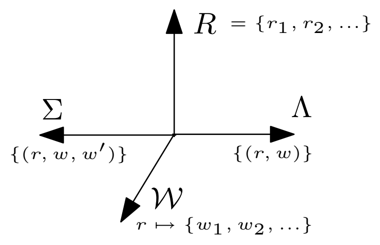
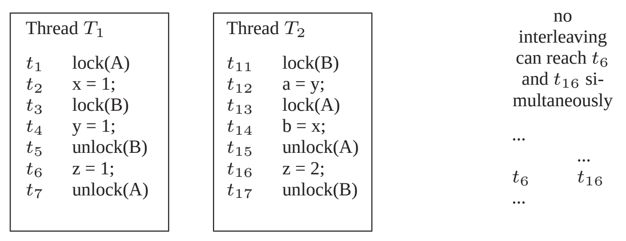


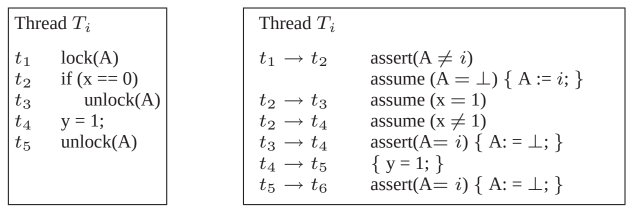
Figure 4. The Space of Syntactic Interference Abstractions.



```
Thread T1                 Thread T2
t1   lock(A)              t11   lock(B)
t2   x = 1;               t12   a = y;
t3   lock(B)              t13   lock(A)
t4   y = 1;               t14   b = x;
t5   unlock(B)            t15   unlock(A)
t6   z = 1;               t16   z = 2;
t7   unlock(A)            t17   unlock(B)
```

no interleaving can reach t_6 and t_{16} simultaneously
...
...
t_6 t_{16}
...

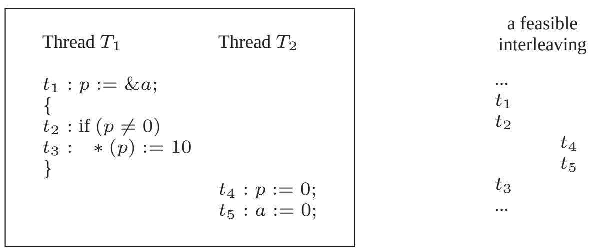
Figure 5. A control-state reachability analysis can prove the absence of data race (between t_6 and t_{16}).



```
Thread Ti
t1   lock(A)
t2   if (x == 0)
t3       unlock(A)
t4   y = 1;
t5   unlock(A)
```

```
Thread Ti
t1 → t2    assert(A ≠ i)
           assume (A = ⊥) { A := i; }
t2 → t3    assume (x = 1)
t2 → t4    assume (x ≠ 1)
t3 → t4    assert(A= i) { A: = ⊥; }
t4 → t5    { y = 1; }
t5 → t6    assert(A= i) { A: = ⊥; }
```

Figure 6. The assertion failure at t_5, caused by double unlock, can be detected in a serial execution.



```
Thread T1              Thread T2

t1 : p := &a;
{
t2 : if (p ≠ 0)
t3 :    *(p) := 10
}
                       t4 : p := 0;
                       t5 : a := 0;
```

a feasible interleaving
...
t_1
t_2
t_4
t_5
t_3
...

Figure 7. Any serial execution of block t_2, t_3 is non-erroneous.

in any execution, r may link with w irrespective of their values, addresses and execution order. The formula Π_3 corresponds to *global* consistency: not instantiating Π_3 for some (r, w, w') implies that in any execution where r links with w, any other interfering w' is allowed to interleave in between r and w.

5.3.1 Control-State Reachability

Sometimes we can prove a property using a control-state reachability analysis [23, 31] where the control flow structure and the synchronization operations (such as *lock-unlock* and *signal-wait*) are modeled precisely, whereas the other data flow is ignored. This reasoning corresponds to a *semantic* IA defined as follows: partition the set of all reads $\mathbb{R}$ on shared variables SV in the program into two disjoint subsets: R_{sync} and $(\mathbb{R} \setminus R_{sync})$. The subset R_{sync} consists of all the variables modeling the synchronization primitives. The subset $(\mathbb{R} \setminus R_{sync})$ consists of the remaining global variables, which will be ignored in the IA. Given a default write map $\widehat{\mathcal{W}}$ for the reads in the program, this semantic IA can be obtained syntactically as

$$\alpha = \langle R_{sync}, \widehat{\mathcal{W}}, \lambda(R_{sync}, \widehat{\mathcal{W}}), \sigma(R_{sync}, \widehat{\mathcal{W}}, \widehat{\mathcal{W}}) \rangle$$

The IA α is an OIA by definition ($\mathcal{W} = \widehat{\mathcal{W}}$ and $R_{sync} \subseteq \mathbb{R}$) and hence a proof obtained with α still holds when the SC axioms are fully enforced.

Consider Fig. 5 as an example. The two concurrent threads T_1, T_2 communicate through locks A, B and shared variables x, y, z. The property of interest is that whether the writes to z at t_6 and t_{16} cause a data race. We can show that no data race exists by using a control-state reachability analysis based on locks A and B only; the rest of the variables x, y and z may be ignored because of the following reason. Transitions t_6 and t_{16} cannot be enabled at the same time, because thread T_1 must acquire A in order to reach t_6, but if A is held by T_1, then thread T_2 cannot reach t_{16} because it cannot acquire A at t_{13}. We can capture this reasoning precisely by including reads on only A and B variables in R_{sync} in the IA α above and therefore can prove the absence of data race with α.

5.3.2 Serial or Largely Serial Execution

Sometimes program bugs are insensitive to thread scheduling, i.e., they appear even in executions where the threads execute serially or interleave sporadically [7, 9]. Thread-local bugs fall in this category, e.g., as in Fig. 6 (left), where $x = 0$ initially. Here lock A may be released twice. Recall (cf. Sec. 2) that we encode locks using guarded assignments to shared variables as in Fig. 6 (right). Additional assertions are added to t_1 for checking double-locking errors, and to t_3, t_5 for checking double-unlocking errors. The assertion at t_5 is violated due to double unlocking. We can detect this violation by only considering a serial execution of threads T_i.

This form of reasoning can be captured by an UIA $\alpha = (R, \mathcal{W}, \Lambda, \Sigma)$, where R contains reads on x which are only linked with writes inside the same thread or an initial write. More precisely, the read of x at t_2 links with the initial write $x = 0$, the read of A at t_5 links with the write at either t_1 or t_3, and so on. Note that restricting the set of writes to link makes $\mathcal{W}(r) \subset \widehat{\mathcal{W}}(r)$ for $r \in R$, and hence results in an UIA. Checking this UIA for satisfiability corresponds to checking only serial executions.

Sometimes all the serial executions are good, but an interleaved execution involving only a small amount of interference may lead to a bug. Fig. 7 shows one such bug due to atomicity violation[2]. The transitions t_2, t_3 in thread T_1 are intended to be executed atomically; however, the programmer fails to enforce it. If t_4 is interleaved in between t_2 and t_3, a NULL dereference occurs. Again, an UIA α is sufficient to detect such bugs. Note that if we force each global read to copy from the preceding intra-thread write, we will not be able to detect the bug. Therefore, the UIA should allow the read of p at t_3 to link with t_4 (besides t_1). Inferring such reduced set of writes ($\mathcal{W}$) automatically is, however, the prime challenge.

6. Exploring the IA Space

We now focus on finding an efficient exploration strategy over the IA space to discover IAs of small size, which are precise enough for checking the given property. Let $\mathcal{A}$ denote the set containing all possible IAs. We define an order relation on $\mathcal{A}$ as follows.

DEFINITION 9 (Order of IAs). *Given two IAs $\alpha = (R, \mathcal{W}, \Lambda, \Sigma)$ and $\beta = (R', \mathcal{W}', \Lambda', \Sigma')$, we say that $\alpha \prec \beta$ if $R \subseteq R'$, $\mathcal{W} \subseteq \mathcal{W}'$, $\Lambda \subseteq \Lambda'$, $\Sigma \subseteq \Sigma'$, and at least one of $R, \mathcal{W}, \Lambda$ is a proper subset of the corresponding $R, \mathcal{W}', \Lambda'$.*

[2] For modeling pointers/structures, please refer to our previous work [13].

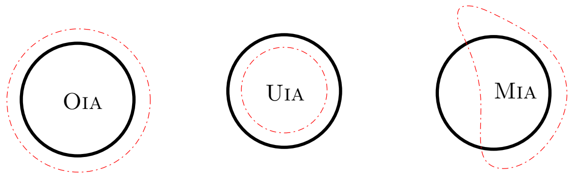


Figure 8. Semantic Interpretation of IAs. The bold circle denotes the full instantiation of Π.

If $\alpha \prec \beta$, then we say that β *refines* α. The poset $(\mathcal{A}, \prec)$ is a complete lattice with component-wise set union and intersection as the join and meet operators. The top element of the lattice $(\mathbb{R}, \widehat{\mathcal{W}}, \lambda(\mathbb{R}, \widehat{\mathcal{W}}), \sigma(\mathbb{R}, \widehat{\mathcal{W}}, \widehat{\mathcal{W}}))$ corresponds to a full instantiation $(\Pi^\alpha = \widehat{\Pi})$ while the bottom element $(\emptyset, \emptyset, \emptyset, \emptyset)$ corresponds to not instantiating Π at all $(\Pi^\alpha = true)$.

6.1 Exploring the Lattice

Given a property P, we say that an α is *minimal* for P if α is an OIA (UIA) which proves (falsifies) P, and there exists no $\beta \prec \alpha$ such that β proves(falsifies) P. Since computing a minimal IA is at least as hard as checking the property itself, we are only interested in practically efficient algorithms to compute *small* IAs.

The formulation of syntactic IAs suggests two naive strategies to obtain small IAs. Starting with a UIA α, one may iteratively augment the sets $\mathcal{W}$, Λ and Σ until an actual witness is obtained. Similarly, one may start with an OIA α and iteratively instantiate constraints until a proof is obtained. Both these refinement strategies introduce new constraints in a *lazy* manner. Semantically, this form of refinement corresponds to increasing coupling or interference between threads and checking if a witness or a proof persists as the coupling increases.

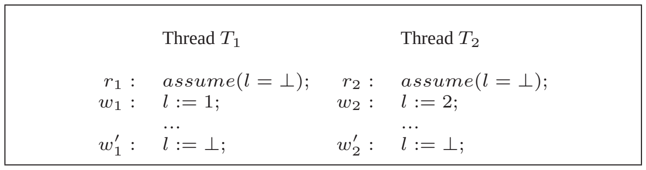
The two refinement strategies presented above have a number of issues. First, these strategies are suitable either for finding proofs (using OIA's) or witnesses (using UIA's), but not both. Second, since the two IAs are disjoint, the proof-directed strategy does not gain from the witness-directed strategy, and vice versa. Ideally, we desire of a refinement method where both OIA's and UIA's could work in unison and assist each other. A natural way to combine OIA's and UIA's is via an MIA.

Fig. 8 depicts and compares the semantics of OIA's, UIA's and MIA's in a visual manner. Recall that the full instantiation $\widehat{\Pi}$ models sequential consistency precisely and hence corresponds to all feasible thread interleavings. An OIA removes interference constraints from $\widehat{\Pi}$ and therefore leads to more interleavings (infeasible ones) than allowed by $\widehat{\Pi}$. In contrast, an UIA adds interference constraints to $\widehat{\Pi}$, leading to fewer interleavings than allowed by $\widehat{\Pi}$. An MIA contains both an OIA and an UIA, and therefore omits some interleavings from $\widehat{\Pi}$ while allowing some infeasible ones.

Although MIA's combine the advantages of both OIA's and UIA's, neither models nor proofs of MIA's may provide conclusive results because of combined over- and under-approximation in an MIA. We now examine the sufficient conditions under which a model (proof) of an MIA may be an actual model (proof) for $\widehat{\Pi}$.

6.2 Models and Proofs of (Mixed) IAs

Given an MIA α, we define the interference relation (IR) $\mathcal{I}$ satisfying Π^α (model of the MIA) in a manner similar to Defn. 4: in the solution of Π^α, $R_\mathcal{I}$ and $W_\mathcal{I}$ are enabled reads and writes respectively, $M(r) = w$ iff $link(r, w)$ holds and $\sqsubset$ consists of (r, w)-pairs satisfying HB relation. Each IR $\mathcal{I}$, in turn, corresponds to an induced IA defined as follows. Intuitively, if an IR $\mathcal{I}$ satisfies Π^α, then the induced IA, IA($\mathcal{I}$), models the subset of SC constraints relevant to $\mathcal{I}$. The hope is that IA($\mathcal{I}$) is an UIA; in that case, $\mathcal{I}$ is a true witness.



	Thread T_1		Thread T_2
r_1 :	$assume(l = \bot);$	r_2 :	$assume(l = \bot);$
w_1 :	$l := 1;$	w_2 :	$l := 2;$
	...		...
w_1' :	$l := \bot;$	w_2' :	$l := \bot;$

Figure 9. Example with concurrent lock/unlock.

DEFINITION 10 (IR-induced IA). *Given an interference relation* $\mathcal{I} = (R_\mathcal{I}, W_\mathcal{I}, M, \sqsubset)$, *the IA* $\beta = (R, \mathcal{W}, \Lambda, \Sigma)$ *induced by* $\mathcal{I}$, *denoted by IA*($\mathcal{I}$), *is defined as follows.*

- *(i)* $R = R_\mathcal{I}$,
- *(ii)* $\mathcal{W}(r) = \{M(r)\}$ *if* $M(r)$ *is defined, else* $\mathcal{W}(r) = \emptyset$,
- *(iii)* $\Lambda = M$,
- *(iv)* $\Sigma = \{(r, w, w') \mid (r, w) \in M \wedge w' \in W_\mathcal{I} \wedge w \sqsubset w' \sqsubset r\}$.

However, in general, $\mathcal{I}$ may not correspond to a true witness of $\widehat{\Pi}$ because of approximation in MIA: all the constraints IA($\mathcal{I}$) relevant to the $\mathcal{I}$ may not be enforced by (contained in) the current MIA. On enforcing the missing constraints (from the set IA($\mathcal{I}$)), if $\mathcal{I}$ is no longer a witness, then we say that $\mathcal{I}$ is Π-inconsistent.

DEFINITION 11 (Π-inconsistent IR). *Suppose an IR* $\mathcal{I}$ *induces IA* β *= IA*($\mathcal{I}$). *We say that* $\mathcal{I}$ *is* Π*-inconsistent if* Π^β *is unsatisfiable.*

An IR $\mathcal{I} = (R_\mathcal{I}, W_\mathcal{I}, M, \sqsubset)$ may be Π-inconsistent (invalid witness) due to multiple reasons. For example, M may not be defined for some enabled $r \in R_\mathcal{I}$. Instantiating Π_1 for such r is unsatisfiable because antecedent $en(r)$ is true but the consequent is false. We say that $\mathcal{I}$ is Π_1-inconsistent here. Similarly, instantiating Π_2 for a subset of r-w pairs in M may be unsatisfiable. In this case, we say $\mathcal{I}$ is Π_2-inconsistent. We define Π_3-inconsistent similarly. Note that, in general, we may need to instantiate a combination of Π_1, Π_2 and Π_3 constraints for $\mathcal{I}$ to detect if $\mathcal{I}$ is Π-inconsistent.

The following lemma is crucial to finding actual witnesses using IAs: it shows that to find an actual witness, i.e., an IR satisfying $\widehat{\Pi}$, it is sufficient to compute a Π-consistent IR.

LEMMA 3. *If an IR* $\mathcal{I}$ *is* Π*-consistent, then* $\mathcal{I}$ *satisfies* $\widehat{\Pi}$.

Example. Consider the example in Fig. 9 with two threads T_1 and T_2, each containing a pair of lock/unlock statements on the lock l. The lock/unlock statements are transformed into guarded statements (cf. Sec. 2) and note that each lock/unlock pair is associated with a triple of lock accesses, e.g., (r_1, w_1, w_1') for T_1. Here, the link set of r_1 is $\mathcal{W}(r_1) = \{w_2, w_2'\}$. Similarly, $\mathcal{W}(r_2) = \{w_1, w_1'\}$. Consider an IR $\mathcal{I} = (R_\mathcal{I}, W_\mathcal{I}, M, \sqsubset)$, where $R_\mathcal{I}$ and $W_\mathcal{I}$ contain all reads and writes respectively, $M = \{(r_1, w_2'), (r_2, w_1')\}$ and $\sqsubset$ contains program order relation, i.e., $r_1 \sqsubset w_1 \sqsubset w_1'$ and $r_2 \sqsubset w_2 \sqsubset w_2'$. Clearly, match M violates lock semantics. More precisely, M is Π_2-inconsistent, i.e., on instantiating Π_2 for pairs in M, the IR $\mathcal{I}$ becomes unsatisfiable because the transitivity of HB relation is violated. Consider another IR $\mathcal{I}'$ with ordering: $r_1 \sqsubset w_1 \sqsubset w_1' \sqsubset r_2 \sqsubset w_2 \sqsubset w_2'$. Suppose the match M' links r_2 with w_1. The IR $\mathcal{I}'$ is Π_3-inconsistent: if r_2 links with w_1 then the interfering write w_2' should not occur in between, and hence $\mathcal{I}'$ violates Π_3.

Dual to the notion of a Π-consistent IR (a valid witness) is the idea of a *valid proof*, i.e. a subset of constraints of Π^α, which does not mention any reads whose link sets are under-approximated.

DEFINITION 12 (Valid Proof). *Suppose the instantiation* Π^α *for an IA* $\alpha = (R, \mathcal{W}, \Lambda, \Sigma)$ *is unsatisfiable and* P *is a proof of unsatisfiability. We say that* P *is valid if* P *contains no* r *such that* $\mathcal{W}(r) \subset \widehat{\mathcal{W}}(r)$.

We say that a read r *invalidates* a proof P, if P contains r and $\mathcal{W}(r) \subset \widehat{\mathcal{W}}(r)$. The following lemma shows that to prove absence of errors, it is sufficient to find an IA having a valid proof P.

LEMMA 4. *Let the instantiation Π^α of an IA $\alpha = (R, \mathcal{W}, \Lambda, \Sigma)$ have a valid proof P. Then P is also a proof for $\widehat{\Pi}$.*

6.3 Refinement of Mixed IA's

We now present our refinement procedure REF which tries to automatically compute an IA precise enough for either proving the property or finding a violating witness. REF works directly on an mixed IA (MIA) and iteratively refines the MIA to obtain either a valid proof or a Π-consistent IR (cf. Sec. 6.2). Fig. 10 provides an overview of our refinement procedure. The procedure starts with an initial IA $\alpha = (R, \mathcal{W}, \Lambda, \Sigma)$ by choosing sets R, $\mathcal{W}$, Λ and Σ. Then, REF checks if Π^α conjoined with the CCFG and property encodings, Φ_C and Φ_{PRP} respectively (cf. Sec. 3.1), is satisfiable. If Π^α is satisfiable and the IR $\mathcal{I}$ obtained from the solution is Π-consistent, the algorithm terminates with a valid witness. If $\mathcal{I}$ is not Π-consistent, then α must contain an over-approximation due to either R, Λ or Σ (cf. Sec. 6.2). Therefore, Π^α is refined to reduce the over-approximation (using procedure RED-O). Otherwise Π^α is unsatisfiable: if Π^α has a valid proof P, the algorithm terminates. If P is not valid, i.e., it contains an under-approximation due to $\mathcal{W}(r)$ for some r, then Π^α is refined to reduce the under-approximation (using procedure RED-U).

Note the advantage of working with an MIA: upon termination, the final IA may not be an OIA or an UIA; obtaining a Π-consistent IR or a valid proof is sufficient to obtain a conclusive result. We now describe the details of the RED-O and RED-U steps. The pseudo code is listed as follows.

$\boxed{\text{RED-O :}}$
Let $\alpha = (R, \mathcal{W}, \Lambda, \Sigma)$. Let the Π-inconsistent IR be $\mathcal{I} = (R_\mathcal{I}, W_\mathcal{I}, M, \sqsubset)$. Refine α by performing one of the following operations to reduce the over-approximation in α:

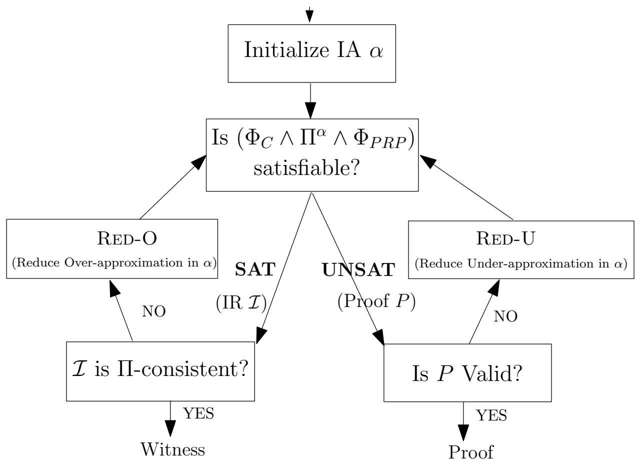
- Pick $r \in R_\mathcal{I} \setminus R$ and $W \subseteq \widehat{\mathcal{W}}(r)$. Set $R := R \cup \{r\}$ and $\mathcal{W}(r) := W$.
- Set $\Lambda := \Lambda \cup M$
- Set $\Sigma := \Sigma \cup \{(r, w, w') \mid (r, w) \in M \wedge w \sqsubset w' \sqsubset r\}$

$\boxed{\text{RED-U :}}$
Since Π^α is unsatisfiable, obtain the set of invalidating reads $R' \subseteq R$ in the proof of unsatisfiability of Π^α. Refine α by setting $\mathcal{W}(r) = \widehat{\mathcal{W}}(r)$ for each $r \in R'$ to reduce the under-approximation in α.

In words, RED-O analyzes the IR $\mathcal{I}$ and then chooses to refine α by updating one or more of R or Λ or Σ depending on the reason of Π-inconsistency (cf. Sec. 6.2). Similarly, RED-U checks if Π^α is unsatisfiable due to under-approximation in $\mathcal{W}(r)$ for some r; in that case, RED-U removes the approximation by setting $\mathcal{W}(r) := \widehat{\mathcal{W}}(r)$.

Checking if the IR $\mathcal{I}$ is Π-inconsistent is done by adding constraints IA($\mathcal{I}$) related to $\mathcal{I}$ incrementally and checking if the result is satisfiable. Note that both RED-O and checking Π-consistency involve adding constraints to reduce the over-approximation. Therefore, we combine them in practice using a layered instantiation strategy (Sec. 7.2). Besides disambiguating the choices in RED-O as presented above, the strategy also avoids adding irrelevant constraints.

Note how REF exploits the uniform representation of IAs in form of MIA's to check properties using a symbiotic combination of OIA's and UIA's. Also, the refinement is guided both by unsatisfiability proofs and witnesses obtained at intermediate iterations



Initialize IA α
Is $(\Phi_C \wedge \Pi^\alpha \wedge \Phi_{PRP})$ satisfiable?
RED-O
(Reduce Over-approximation in α)
SAT
(IR $\mathcal{I}$)
UNSAT
(Proof P)
RED-U
(Reduce Under-approximation in α)
NO
NO
$\mathcal{I}$ is Π-consistent?
Is P Valid?
YES
YES
Witness
Proof

Figure 10. Flow diagram for the refinement procedure REF.

and hence is property-directed. The algorithm REF terminates in finite number of steps because the height of the IA lattice is finite and each iteration of REF ascends the lattice by one or more steps.

The refinement procedure can be implemented efficiently using an incremental SMT solver [28, 29] by iteratively adding new constraints. Moreover, using an MIA during refinement enables sharing the learned information between the constituent OIA's and UIA's inside the MIA, thus allowing the OIA's and UIA's to assist each other for computing both proofs and witnesses.

For the example in Fig. 2, we initialize MIA $\alpha = (R, \mathcal{W}, \Lambda, \Sigma)$ as follows. R includes all reads $r_{11}, r_{14}, r_{21}, r_{24}, r_{50}, r_{51}$. $\mathcal{W}$ is initialized so that the link set of each read contains no concurrent writes: $\mathcal{W}(r_{11}) = \{w_{01}\}$, $\mathcal{W}(r_{14}) = \{w_{13}, w_{17}\}$, $\mathcal{W}(r_{50}) = \{w_{15}, w_{17}\}$ and so on. Suppose we instantiate $\Lambda = \lambda(R, \mathcal{W})$, i.e., for all writes in the link set of each read. Let $\Sigma = \emptyset$. The IA α is an MIA because, e.g., $\mathcal{W}(r_{14}) \subset \widehat{\mathcal{W}}(r_{14})$ (under-approximation) and $\Sigma = \emptyset$ (over-approximation).

On checking Π^α (with Φ_C and Φ_{PRP}), the result is unsatisfiable because r_{11} links with w_{01} and r_{21} links w_{00}. Hence, the *then* branch in T_1 executes while the *else* branch in T_2 executes. Therefore, r_{50} gets value 2 and r_{51} gets value 0 so that the assertion is never violated. The proof of unsatisfiability mentions r_{11} and r_{21}. Suppose the procedure RED-U then expands $\mathcal{W}$ for r_{21} to $\widehat{\mathcal{W}}(r_{21}) = \{w_{00}, w_{13}, w_{15}, w_{17}\}$ and Λ is updated with the corresponding pairs. In the next iteration, suppose we obtain an IR $\mathcal{I}$ which links r_{21} with w_{13}, but $\mathcal{I}$ has $w_{13} \sqsubset w_{15} \sqsubset r_{21}$ in $\mathcal{I}$. Here, $\mathcal{I}$ is not Π-consistent because it violates Π_3. Therefore, REF updates $\Sigma = \{(r_{21}, w_{15}, w_{13})\}$ using RED-O, which finally results in a Π-consistent IR with $w_{13} \sqsubset r_{21} \sqsubset w_{15}$.

Note how a combination of over- and under-approximation was exploited by REF to arrive at a Π-consistent IR. Further, even though we check bounded programs (where paths in each thread or number of threads have been under-approximated), over-approximating interference is orthogonal and does not work against the previous under-approximation. In fact, it may assist in finding the bug quickly in many cases (cf. above example) or obtaining a proof early by avoiding redundant constraints.

7. Focused Refinement

The procedure REF proceeds iteratively by ascending the lattice of IAs based on the current satisfying solution or an infeasibility proof. Since the size of the lattice is exponential in the number of reads and writes, the basic refinement strategy may not converge quickly to a desirable small IA on its own. In particular, it may add redundant constraints guided by the model from the solver, thus making the intermediate IA larger and the subsequent iterations more expensive. We propose a set of heuristics to focus the refinement on relevant constraints.

7.1 Static Focusing

We first describe heuristics to guide REF by removing redundant constraints or adding useful lemmas statically.

(S0) Interference Pruning. For each read r, compute a small $\widehat{\mathcal{W}}(r)$, not containing any writes that r may never link with. For example, a r cannot link with w if $HB(r, w)$ holds statically, or if $HB(w, r)$ and there exist interfering writes w' along each path from w to r. Such writes may be detected by performing a static analysis on the interference skeleton (IS) (cf. Sec. 3.1).

(S1) Biased Initialization. To obtain IAs with lesser concurrent interference, we initialize $\mathcal{W}(r)$ for each r with only writes in the same thread or initial writes. This ensures that if a serial or largely serial execution (cf. Sec. 5.3) violates the property, then few refinement iterations will be sufficient. We also bias the initial IA to couple reads and writes on synchronization variables only. This forces REF to start with only those IAs where the above interference conditions must hold.

(S2) Lock Lemmas. A number of optimizations are possible for locks. **(S2a)** Consider the program shown in Fig. 9 again. The read in the initial assume statement (r_1) may link with either w_2 or w_2'. However note that if $link(r_1, w_2)$ holds then $val(r_1) = val(w_2) = 2$, making the guard $(l = \perp)$ as $false$, which in turn blocks the execution of T_1. Therefore, we reduce the link set $\widehat{\mathcal{W}}(r_1)$ by removing w_2 from it. **(S2b)** Let $\mathcal{L}$ denote the set of matching lock/unlock statements in the whole program and the accesses to the lock variable for each $L_i \in \mathcal{L}$ be (r_i, w_i, w_i'). For each $L_1, L_2 \in \mathcal{L}$, either the statement block denoted by L_1 executes before the block denoted by L_2 or vice-versa. This fact can be captured by the constraints $\forall i.\forall j.\ (HB(w_i', r_j) \vee HB(w_j', r_i))$, which are quadratic in the size of $\mathcal{L}$ (better than the original cubic size).

(S3) Although instantiating Π_2 eagerly for all pairs of reads/writes $\Lambda = \lambda(\mathbb{R}, \widehat{\mathcal{W}})$ is expensive, instantiating only a portion of Π_2 for Λ incurs lesser cost both in terms of constraints size and solving times. We can therefore instantiate only a portion of Π_2, e.g., $link(r, w) \Rightarrow HB(w, r)$ for all $(r, w) \in \Lambda$ eagerly. This is especially useful if the ordering constraints are sufficient for checking the property. Formally, this form of instantiation corresponds to further partitioning the components of an α and instantiating some of those partitions eagerly.

7.2 Layered Instantiation

Note that in RED-O, updating Σ adds far more (quadratic in number of reads and writes) constraints than updating Λ (linear in size of reads R). We, therefore, perform a layered refinement towards the goal of adding fewer redundant constraints. Given an IR $\mathcal{I} = (R, W, M, \sqsubset)$ satisfying Π^α, we first check if $\mathcal{I}$ is Π_2-consistent (cf. Sec. 6.2). If $\mathcal{I}$ is not Π_2-consistent, we update Λ and continue to the next iteration. Only when we find a Π_2-consistent IR, we check if $\mathcal{I}$ is Π_3-consistent. In this way, we bias the refinement towards Λ. Similarly, we update the set R only when the current match M is both Π_2- and Π_3-consistent.

8. Experiments

We implemented the algorithm REF and evaluated it on concurrent benchmarks in the FUSION framework [20, 21]. Although our method can be applied to arbitrary bounded concurrent programs, in these experiments, we focus on analyzing concurrent program slices obtained by run-time analysis of the FUSION tool. Our benchmarks consist of Java program slices obtained from the various publicly available programs [33–35]. We check for errors such as assertion violations and data races. Many of these program slices are difficult to analyze, because they contain multiple threads (with forks and joins), each having a large number of global accesses, together with *assume* statements containing guards for branches taken during run-time. Our goal was to investigate if the hardest subset of available benchmarks could be checked using small IAs.

We initialize the refinement scheme with an IA containing all reads in the CCFG and the link set of each read with only writes which are non-concurrent with the read. Lock lemmas were also added to improve refinement. We observed that lemma **S2a** had a much larger impact than lemma **S2b**, although **S2b** also helped reduce run-times in a few cases. Further, we used biased refinement (7.2) strategy for reducing over-approximation. We focus on showing the effectiveness of IAs and that MIA's can perform better than OIA's or UIA's. All experiments were conducted on 2.33GHz Intel Xeon machine with 16GB memory.

Fig. 11 compares the full instantiation with the refinement (REF) procedure on a set of properties of the hard benchmarks. In this table, Columns 1-3 show the name of each test case, the number of threads, and whether there is a bug (T) or a proof (F). Columns 4-6 show the number of reads, the number of writes, and the average size of the link sets. The remaining columns compare the run-times and the IA sizes obtained with and without interference abstraction. The IAs computed by REF are drastically smaller than full instantiation both for satisfiable and unsatisfiable benchmarks. This confirms the belief that only a small amount of interference need be considered for checking many properties. Moreover, small IAs lead to much lesser run-times than with the full instantiation. We observed that for obtaining proofs, REF needed to instantiate Π_2 only a few times for each r-w pair in the initial IA. Finding witnesses is more challenging than finding proofs: REF goes through a number of iterations for reducing both the under- and over-approximation present in the initial IA.

Fig. 12 shows the advantage of using combined MIA's in REF vs using only OIA's or only UIA's. These are scatter plots where the x-axis is the run-time with only OIA's or only UIA's, and the y-axis is the run-time with MIA's. The data points correspond to checking several data race properties in benchmark (syncBench.1119). We observe that MIA's clearly outperform OIA's implying that under-approximated IAs are essential for efficiency. In contrast, MIA's may not be always better than only UIA's: however, the over-approximation in MIA's improves the performance in most cases.

The procedure REF for finding small IAs may be viewed as an automated quantifier instantiation (AQI) technique specific to the domain of SC axioms where the variables are quantified over finite domains. Although a number of SMT solvers support asserting quantified formula and include generic AQI heuristics, we found that such approach did not perform well even on small benchmarks (we used Yices [28] with default settings). We believe it is due to the following reasons: first, the solver must be provided with large number (at least quadratic in the number of reads and writes) of non-interference constraints derived from static pruning (cf. Sec. 7.1), without which these benchmarks become intractable. Secondly, generic AQI heuristics often lead to eager redundant instantiation which severely hurts performance. Most importantly, it is difficult for generic heuristics to infer a biased initialization of the MIA's: the initialization exploits knowledge about the typical behaviors of concurrent programs, which is unavailable to the solver.

We are aware that a number of improved AQI heuristics have been implemented in Z3 [36]. We believe that these recent improvements are complementary, and, if exploited correctly, could assist REF in converging faster. Note that an SMT solver also uses a combination of under- and over-approximations internally during constraint solving, which is unfortunately ineffective in handling the full instantiation directly. In other words, the solver is unable

Bm	N	SAT	R	W	Avg. $\|\mathcal{W}\|$	Full Instantiation		With IAs (REF)	
						T(s)	IA Size	T (s)	IA Size
barrierB.653	13	F	285	269	9	29	2.7K / 2.7K / 91K	2	249 / 285 / 0
barrierB.653	13	T	285	269	9	30	2.7K / 2.7K / 91K	22	350 / 382 / 3.2K
syncBench.722	13	F	270	289	3	4	891 / 891 / 8.5K	2	259 / 270 / 0
syncBench.722	13	T	270	289	3	4	891 / 891 / 8.5K	2	254 / 270 / 55
syncBench.1119	16	F	496	457	24	902	12K / 12K / 1M	3	477 / 496 / 0
syncBench.1119	16	T	496	457	24	741	12K / 12K / 1M	149	533 / 498 / 12K
syncBench.1954	19	F	1012	856	48	>1hr	49K / 49K / 8M	15	989 / 1012 / 0
syncBench.1954	19	T	1012	856	48	>1hr	49K / 49K / 8M	258	1056 / 1023 / 20K
daisy1	3	F	496	798	19	117	10K / 10K / 0.4M	5	370 / 495 / 0
daisy1	3	T	496	798	19	681	10K / 10K / 0.4M	396	370 / 30 / 850
elevator1	4	F	829	615	3	202	3K / 3K / 29K	38	824 / 0 / 0
elevator1	4	T	829	615	3	82	3K / 3K / 29K	23	824 / 30 / 0
elevator2	4	F	2259	1491	10	>1hr	24K / 24K / 0.7M	15	2204 / 0 / 0
elevator2	4	T	2259	1491	10	>1hr	24K / 24K / 0.7M	14	2204 / 39 / 215

Figure 11. Experimental Results. R (W) = number of reads (writes). IA size denotes the size of the IA $\alpha = (R', \mathcal{W}, \Lambda, \Sigma)$ when the check terminates, in form (A/B/C) where A = $\sum_r |\mathcal{W}(r)|$, B = $|\Lambda|$ and C = $|\Sigma|$. nK and nM are shorthand for $n * 10^3$ and $n * 10^6$ respectively.

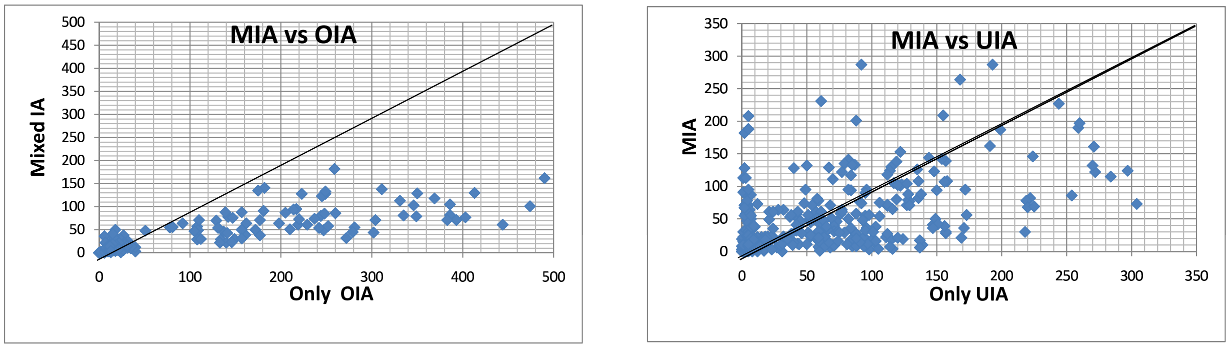


Figure 12. Run-times using MIA's vs. only OIA's or only UIA's on benchmark syncBench.1119.

to focus on the concurrent facts relevant to the property by itself; the presented refinement scheme, in contrast, has the concurrency-specific knowledge (e.g., to compute an initial biased IA) and is able to steer the solver towards the relevant facts.

9. Related Work

Automated reasoning about concurrent programs with shared memory has been traditionally done by systematically restricting the thread scheduler [2, 4, 5, 37, 38] based on partially-ordered traces [1] with dependency relation (Mazurkiewicz (M-) traces). In particular, the work in [4] uses iterative enlargements of scheduler under-approximations to find bugs based on proofs from a SAT solver. Automated compositional methods have also employed abstractions of both transition relations [39, 40] and state spaces [41] of individual threads. IAs, in contrast, build upon the axioms of memory consistency instead of M-traces (cf. [42]). Note that the notion of IAs is orthogonal to abstractions of transition relations: IAs abstract only the correlations between reads and writes without modifying the transition relations of individual threads. The notion of *field abstraction* introduced in [26] for removing reads and writes to selected structure fields may be viewed as a form of OIA. However, field refinement links each field read with *all* possible writes, thus hampering its scalability. As our experiments show, the combination of OIA's with UIA's is important.

Iterative abstraction-refinement methods [16, 17] for sequential software using predicate abstraction [18, 19] have been investigated widely. Mixed abstractions of transition systems containing both *may* and *must* transitions to preserve universal and existential properties respectively have also been studied and applied to sequential software (cf. [43] for a nice overview). Recent work has also combined may- and must-summaries of procedures to obtain a more scalable analysis of sequential software [44]. Decision procedures for bit-vectors [45] also employ mixed abstractions of formula.

Automatic quantifier instantiation (AQI) inside SMT solvers is an active research topic. The most prevalent AQI strategy [28, 29], introduced in Simplify [46], employs *triggers* [36, 46]: to enable QI, subterms (triggers) of quantified assertions are matched (unified) with the ground terms in the partial model of the solver. However, such heuristics are in general incomplete and often cause a large number of redundant instantiations. Leino *et al.* proposed to handle quantified assertions via a separate module [47] similar to a theory module in an SMT solver. However, lack of tight integration between the quantifier module and the main solver leads to duplicate theory reasoning as well as restrained learning.

10. Conclusions

We presented a new form of concurrency abstraction for shared memory programs called interference abstractions (IAs) based on the axioms of sequential consistency. The framework of IAs provides an automated and flexible mechanism for approximating interference. An iterative algorithm to synthesize IAs for checking concurrent properties was presented and shown to yield small IAs for practical benchmarks. IAs may be extended in multiple ways, e.g., we can *cluster* multiple reads and/or writes into a single access and reason about these access sets simultaneously. These extensions, in contrast to pure IAs, may also violate the program order. Extending the notion of IAs to relaxed memory models is also an interesting direction. We also plan to compare with automated quantifier instantiation inside constraint solvers, handle unbounded programs, and investigate rely-guarantee reasoning using interference abstractions.

Acknowledgments. We thank the anonymous reviewers for their careful reading and valuable suggestions. We are also indebted to Yeting Ge and Cesare Tinelli for useful discussions.

References

[1] Mazurkiewicz, A.W.: Trace theory. In: Advances in Petri Nets. (1986) 279–324

[2] Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1996)

[3] Peled, D.: Partial order reduction: Model-checking using representatives. In: MFCS. (1996) 93–112

[4] Grumberg, O., Lerda, F., Strichman, O., Theobald, M.: Proof-guided underapproximation-widening for multi-process systems. In: POPL. (2005) 122–131

[5] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: OSDI. (2008) 267–280

[6] Kahlon, V., Wang, C., Gupta, A.: Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In: CAV. (2009) 398–413

[7] Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS. (2005) 93–107

[8] Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI. (2007) 446–455

[9] Lal, A., Touili, T., Kidd, N., Reps, T.W.: Interprocedural analysis of concurrent programs under a context bound. In: TACAS. (2008) 282–298

[10] Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer **29**(12) (1996) 66–76

[11] Burckhardt, S., Alur, R., Martin, M.M.K.: Checkfence: checking consistency of concurrent data types on relaxed memory models. In: PLDI. (2007) 12–21

[12] Torlak, E., Vaziri, M., Dolby, J.: Memsat: checking axiomatic specifications of memory models. In: PLDI. (2010) 341–350

[13] Sinha, N., Wang, C.: Staged concurrent program analysis, FSE 2010

[14] Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers **28**(9) (1979) 690–691

[15] Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. SIGARCH Comput. Archit. News **36**(1) (2008) 329–339

[16] Kurshan, R.P.: Computer-aided verification of coordinating processes: the automata-theoretic approach. Princeton University Press (1994)

[17] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM (JACM) **50**(5) (2003) 752–794

[18] Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: PLDI. Volume 36(5)., ACM Press (June 2001) 203–213

[19] Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL. (2004) 232–244

[20] Wang, C., Chaudhuri, S., Gupta, A., Yang, Y.: Symbolic pruning of concurrent program executions. In: FSE 2009. 23–32

[21] Wang, C., Kundu, S., Ganai, M.K., Gupta, A.: Symbolic predictive analysis for concurrent programs. In: FM. (2009) 256–272

[22] Wang, C., Limaye, R., Ganai, M., Gupta, A.: Trace-based symbolic analysis for atomicity violations. In: TACAS, Springer (2010) 328–342

[23] Kahlon, V., Wang, C.: Universal Causality Graphs: A precise happens-before model for detecting bugs in concurrent programs. In: CAV, Springer (2010) 434–445

[24] Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In Jensen, K., Podelski, A., eds.: TACAS. Volume 2988 of LNCS., Springer (2004) 168–176

[25] Ivancic, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P.: F-soft: Software verification platform. In: CAV. (2005) 301–306

[26] Lahiri, S.K., Qadeer, S., Rakamaric, Z.: Static and precise detection of concurrency errors in systems code using smt solvers. In: CAV. (2009) 509–524

[27] Ballance, R.A., Maccabe, A.B., Ottenstein, K.J.: The program dependence web: A representation supporting control, data, and demand-driven interpretation of imperative languages. In: PLDI'90. 257–271

[28] Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: CAV. (2006) 81–94

[29] de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: TACAS. (2008) 337–340

[30] Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: detecting atomicity violations via access interleaving invariants. In: ASPLOS. (2006) 37–48

[31] Farzan, A., Madhusudan, P., Sorrentino, F.: Meta-analysis for atomicity violations under nested locking. In: CAV. (2009) 248–262

[32] Yang, Y., Gopalakrishnan, G., Lindstrom, G., Slind, K.: Nemos: A framework for axiomatic and executable specifications of memory consistency models. In: IPDPS. (2004)

[33] `http://www.javagrande.org/`: The Java Grande Forum Benchmark Suite.

[34] Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. International Journal on Software Tools for Technology Transfer (STTT) **2**(4) (2000)

[35] von Praun, C., Gross, T.R.: Static detection of atomicity violations in object-oriented programs. Object Technology **3**(6) (2004)

[36] de Moura, L.M., Bjørner, N.: Efficient e-matching for smt solvers. In: CADE. (2007) 183–198

[37] Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL. (2005) 110–121

[38] Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press.

[39] Flanagan, C., Qadeer, S.: Thread-modular model checking. In: SPIN. (2003) 213–224

[40] Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread-modular abstraction refinement. In: CAV. Volume 2725., Springer-Verlag (2003) 262–274

[41] Cohen, A., Namjoshi, K.S.: Local proofs for global safety properties. Formal Methods in System Design **34**(2) (2009) 104–125

[42] Şerbănuţă, T.F., Chen, F., Roşu, G.: Maximal causal models for sequentially consistent multithreaded systems. Technical report, University of Illinois (2010)

[43] Wei, O., Gurfinkel, A., Chechik, M.: Mixed transition systems revisited. In: VMCAI. (2009) 349–365

[44] Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional may-must program analysis: unleashing the power of alternation. In: POPL. (2010) 43–56

[45] Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.A.: An abstraction-based decision procedure for bit-vector arithmetic. STTT **11**(2) (2009) 95–104

[46] Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3) (2005) 365–473

[47] Leino, K.R.M., Musuvathi, M., Ou, X.: A two-tier technique for supporting quantifiers in a lazily proof-explicating theorem prover. In: TACAS. (2005) 334–348

Appendix

LEMMA 1. *Given a syntactic* OIA α, $\widehat{\Pi} \Rightarrow \Pi^\alpha$.

Proof. We can view $\widehat{\Pi}$ as Π^α conjoined with instantiations of Π for $r \in \mathbb{R} \setminus R$ or $(r, w) \in \lambda(\mathbb{R}, \widehat{\mathcal{W}}) \setminus \Lambda$ or $(r, w, w') \in \sigma(\mathbb{R}, \widehat{\mathcal{W}}, \widehat{\mathcal{W}}) \setminus \Sigma$. Therefore, $\widehat{\Pi} \Rightarrow \Pi^\alpha$. $\diamond$

LEMMA 2. *Given a syntactic* UIA $\alpha = (R, \mathcal{W}, \Lambda, \Sigma)$, $\Pi^\alpha \Rightarrow \widehat{\Pi}$.

Proof. Let the IR $\mathcal{I} = (M, \sqsubset)$ be the model for Π^α where $M : R \rightarrow \mathbb{W}$. Since Π^α is satisfiable, so are Π_1^α, Π_2^α and Π_3^α. We now show that all of $\widehat{\Pi_1}$, $\widehat{\Pi_2}$, $\widehat{\Pi_3}$ are individually satisfiable, and hence $\mathcal{I}$ satisfies $\widehat{\Pi}$. Note that $\widehat{\Pi_1}$ only contains more disjunctions than Π_1^α, hence $\widehat{\Pi_1}$ is satisfiable. We can partition the set of r-w pairs $\lambda(R, \mathcal{W})$ for $\widehat{\Pi_2}$ into $\Lambda_1 = \{(r, M(r)) \, | r \in R\}$ and the rest, say Λ_2. Note that since M contains (r, w) for each $(r, w) \in \Lambda_1$, so $link(r, w)$ must hold. Since $link$ is exclusive, so $(link(r, w) = false)$ for all $(r, w) \in \Lambda_2$. Hence $\phi_2(r, w)$ evaluates to true for all $(r, w) \in \Lambda_2$, and therefore $\widehat{\Pi_2}$ is satisfiable. Similar reasoning applies to $\widehat{\Pi_3}$. $\diamond$

LEMMA 3. *If an IR $\mathcal{I}$ is Π-consistent, then $\mathcal{I}$ satisfies $\widehat{\Pi}$.*

Proof. Let $\mathcal{I} = (R_\mathcal{I}, W_\mathcal{I}, M, \sqsubset)$. The induced IA $IA(\mathcal{I}) = (R_\mathcal{I}, \mathcal{W}', \Lambda', \Sigma')$ (cf. Defn. 10) can be extended to an UIA $\alpha = (R, \mathcal{W}, \Lambda, \Sigma)$ as follows. Let R consist of $R_\mathcal{I}$ together with all disabled reads. Let the $\mathcal{W} = \mathcal{W}'$. Let, $\Lambda = \lambda(R, \mathcal{W})$. Let $\Sigma = \Sigma' \cup \{(r, w, w') \mid (r, w) \in M \wedge (w' \sqsubset w \vee r \sqsubset w')\}$. Note that $\Sigma = \sigma(R, \mathcal{W}, \widehat{\mathcal{W}})$ and hence α is an UIA. Because $\mathcal{I}$ also satisfies UIA α, hence, by Lemma 2, $\mathcal{I}$ also satisfies $\widehat{\Pi}$.

LEMMA 4. *Let the instantiation Π^α of an IA $\alpha = (R, \mathcal{W}, \Lambda, \Sigma)$ have a valid proof P. Then P is also a proof for $\widehat{\Pi}$.*

Proof. Let $R' \subseteq R$ consist of reads r with under-approximated link set, i.e., $\mathcal{W}(r) \subset \widehat{\mathcal{W}}(r)$. We can write Π^α as $F \wedge G$ where $F = \forall r \in R'. \exists w \in \mathcal{W}(r). \; \phi_1(r, w)$ and G denotes the rest of the formula. Since P is an UNSAT proof for Π^α and does not mention R', therefore P is a proof for G. Now $\widehat{\Pi}$ may be written as $F' \wedge G \wedge G'$, where $F' = \forall r \in R' \exists w \in \widehat{\mathcal{W}}(r). \; \phi_1(r, w)$ and G' denotes the rest of the formula. Therefore, $\widehat{\Pi}$ is also has a proof P.