Symmetry-Aware Predicate Abstraction for Shared-Variable Concurrent Programs*

Alastair Donaldson, Alexander Kaiser, Daniel Kroening, and Thomas Wahl

Computer Science Department, Oxford University, United Kingdom

Abstract. *Predicate abstraction* is a key enabling technology for applying finite-state model checkers to programs written in mainstream languages. It has been used very successfully for debugging sequential system-level C code. Although model checking was originally designed for analyzing *concurrent* systems, there is little evidence of fruitful applications of predicate abstraction to shared-variable concurrent software. The goal of this paper is to close this gap. We have developed a *symmetry-aware* predicate abstraction strategy: it takes into account the replicated structure of C programs that consist of many threads executing the same procedure, and generates a Boolean program template whose multi-threaded execution soundly overapproximates the concurrent C program. State explosion during model checking parallel instantiations of this template can now be absorbed by exploiting symmetry. We have implemented our method in the SATABS predicate abstraction framework, and demonstrate its superior performance over alternative approaches on a large range of synchronization programs.

1 Introduction

Concurrent software model checking is one of the most challenging problems facing the verification community today. Not only does software generally suffer from *data state explosion*. Concurrent software in particular is susceptible to state explosion due to the need to track arbitrary thread interleavings, whose number grows exponentially with the number of executing threads.

Predicate abstraction [12] was introduced as a way of dealing with data state explosion: the program state is approximated via the values of a finite number of predicates over the program variables. Predicate abstraction turns C programs into finite-state *Boolean* programs [4], which can be model checked. Since insufficiently many predicates can cause spurious verification results, predicate abstraction is typically embedded into a counterexample-guided abstraction refinement (CEGAR) framework [9]. The feasibility of the overall approach was convincingly demonstrated for *sequential* software by the success of the SLAM project at Microsoft, which was able to discover numerous control-dominated errors in low-level operating system code [5].

The majority of concurrent software is written using mainstream APIs such as POSIX threads (pthreads) in C/C++, or using a combination of language and library support, such as the Thread class, Runnable interface and synchronized construct in Java. Typically, multiple threads are spawned — up front or dynamically, in response to

* This research is supported by EPSRC projects EP/G026254/1 and EP/G051100/1.

varying system load levels — to execute a given procedure in parallel, communicating via shared global variables. For such shared-variable concurrent programs, predicate abstraction success stories similar to that of SLAM are few and far between. The bottleneck is the exponential dependence of the generated state space on the number of running threads, which, if not addressed, permits exhaustive exploration of such programs only for trivial thread counts.

The key to obtaining scalability is to exploit the *symmetry* naturally exhibited by these programs, namely their invariance under permutations of the involved threads. Fortunately, much progress has recently been made on analyzing emphreplicated non-recursive Boolean programs executed concurrently by many threads [6]. In this paper, we present an approach to predicate-abstracting concurrent programs that leverages this recent progress. More precisely, our goal is a scheme that

- translates a non-recursive C program $\mathbb{P}$ with global-scope and procedure-scope variables into a Boolean program $\mathbb{B}$ such that the n-thread Boolean program, denoted $\mathbb{B}^n$, soundly overapproximates the n-thread C program, denoted $\mathbb{P}^n$. We call such an abstraction method *symmetry-aware*.
- permits predicates over arbitrary C program variables, local or global.

In the remainder of the Introduction, we illustrate why approaching this goal naïvely can render the abstraction **unsound**, creating the danger of missing bugs. In the main part of this paper, we present a sound abstraction method satisfying both of the above objectives. We go on to show how our approach can be implemented for C-like languages, complete with pointers and aliasing, and discuss the issues of spurious error detection and predicate refinement.

In the sequel, we present "programs" as code fragments that declare *shared* and *local* variables. Such code is to be understood as a procedure to be executed by any number of threads. The code can declare *shared* variables, assumed to be declared at the global scope of a (complete) program that contains this procedure. Code can also declare *local* variables, assumed to be declared locally within the procedure. We refer to such code fragments with shared and local variables as "programs". In program listings, we use == for the comparison operator, while = denotes assignment (as in C). Concurrent threads are assumed to interleave with statement-level granularity; see the discussion in the Conclusion on this subject.

1.1 Predicate Abstraction using Mixed Predicates

The Boolean program $\mathbb{B}$ to be built from the C program $\mathbb{P}$ will consist of Boolean variables, one per predicate as usual. Since $\mathbb{B}$ is to be executed by parallel threads, its variables have to be partitioned into "shared" and "local". As these variables track the values of various predicates over C program variables, the "shared" and "local" attributes clearly depend on the attributes of the C variables a predicate is formulated over. We therefore classify predicates as follows.

Definition 1 *A **local** predicate refers solely to local C program variables. A **shared** predicate refers solely to shared variables. A **mixed** predicate is neither local nor shared.*

We reasonably assume that each predicate refers to at least one program variable. A mixed predicate thus refers to both local and shared variables.

Given this classification, consider a local predicate ϕ, which can change only as a result of a thread changing one of its local C variables; a change that is not visible to any other thread. This locality is inherited by the Boolean program if predicate ϕ is tracked by a local Boolean variable. Similarly, shared predicates are naturally tracked by shared Boolean variables.

For a mixed predicate, the decision whether it should be tracked in the shared or local space of the Boolean program is non-obvious. Consider first the following program $\mathbb{P}$ and the corresponding generated Boolean program $\mathbb{B}$, which tracks the mixed predicate $s \ != \ l$ in a **local** Boolean variable b:

$\mathbb{P}$:

```
0: shared int s = 0;
   local  int l = 1;
1: assert s != l;
2: ++s;
```

$\mathbb{B}$:

```
0: local bool b = 1;

1: assert b;
2: b = b ? * : 1;
```

Consider the program $\mathbb{P}^2$, a two-thread instantiation of $\mathbb{P}$. It is easy to see that execution of $\mathbb{P}^2$ can lead to an assertion violation, while the corresponding concurrent Boolean program $\mathbb{B}^2$ is correct. (In fact, $\mathbb{B}^n$ is correct for any $n > 0$.) As a result, $\mathbb{B}^2$ is an **unsound** abstraction for $\mathbb{P}^2$. Consider now the following program $\mathbb{P}'$ and its abstraction $\mathbb{B}'$, which tracks the mixed predicate $s \ == \ l$ in a **shared** Boolean variable b:

$\mathbb{P}'$:

```
0: shared int s = 0;
   shared bool t = 0;
   local  int l = 0;
1: if * then
2:     if t then
3:         assert s != l;
4: l = s + 1;
5: t = 1;
```

$\mathbb{B}'$:

```
0: shared bool b = 1;
   shared bool t = 0;

1: if * then
2:     if t then
3:         assert !b;
4: b = 0;
5: t = 1;
```

Execution of $(\mathbb{P}')^2$ leads to an assertion violation if the first thread passes the first conditional, the second thread does not and sets t to 1, then the first thread passes the guard t. At this point, s is still 0, as is the first thread's local variable l. On the other hand, $(\mathbb{B}')^2$ is correct. We conclude that $(\mathbb{B}')^2$ is **unsound** for $(\mathbb{P}')^2$. The unsoundness can be eliminated by making b local in $\mathbb{B}'$; an analogous reasoning removes the unsoundness in $\mathbb{B}$ as an abstraction for $\mathbb{P}$. It is clear from these examples, however, that in general a predicate of the form $s \ == \ l$ that genuinely depends on s and l cannot be tracked by a shared or a local variable without further amendments to the abstraction process.

At this point it may be useful to consider whether, instead of designing solutions that deal with mixed predicates, we may not be better off by banning them, relying solely on shared and local predicates. Such restrictions on the choice of predicates render very simple bug-free programs **unverifiable** using predicate abstraction, including the following program $\mathbb{P}''$:

$\mathbb{P}''$:

```
0: shared int r = 0;
   shared int s = 0;
   local  int l = 0;
1: ++r;
2: if (r == 1) then
3:     f();
```

$f()$:

```
4: ++s, ++l;
5: assert s == l;
6: goto 4;
```

The assertion in $\mathbb{P}''$ cannot be violated, no matter how many threads execute $\mathbb{P}$, since no thread but the first will manage to execute f. It is easy to prove that, over a set of **non-mixed** predicates (i.e. no predicate refers to both l and one of $\{s, r\}$), no invariant is computable that is strong enough to prove $s == l$. We have included such a proof in the full version of this paper [11].

A technically simple solution to all these problems is to instantiate the template $\mathbb{P}$ n times, once for each thread, into programs $\{\mathbb{P}_1, \ldots, \mathbb{P}_n\}$, in which indices $1, \ldots, n$ are attached to the local variables of the template, indicating the variable's owner. Every predicate that refers to local variables is similarly instantiated n times. The new program has two features: (i) all its variables, having unambiguous names, can be declared at the global scope and are thus shared, including the original global program variables, and (ii) it is multi-threaded, but the threads no longer execute the same code. Feature (i) allows the new program to be predicate-abstracted in the conventional fashion: each predicate is stored in a shared Boolean variable. Feature (ii), however, entails that the new program is no longer symmetric. Model checking it will therefore have to bear the brunt of concurrency state explosion. Such an approach, which we refer to as *symmetry-oblivious*, will not scale beyond a very small number of threads.

To summarize our findings: Mixed predicates are necessary to prove properties for even very simple programs. They can, however, not be tracked using standard thread-local or shared variables. Disambiguating local variables avoids mixed predicates, but destroys symmetry. The goal of this paper is a solution without compromises.

2 Symmetry-Aware Predicate Abstraction

In order to illustrate our method, let $\mathbb{P}$ be a program defined over a set of variables V that is partitioned in the form $V = V_S \cup V_L$ into *shared* and *local* variables. The parallel execution of $\mathbb{P}$ by n threads is a program defined over the shared variables and n copies of the local variables, one copy for each thread. A thread is nondeterministically chosen to be *active*, i.e. to execute a statement of $\mathbb{P}$, potentially modifying the shared variables, and its own local variables, but nothing else. In this section, we ignore the specific syntax of statements, and we do not consider language features that introduce aliasing, such as pointers (these are the subject of Section 3). Therefore, an assignment to a variable v cannot modify a variable other than v, and an expression ϕ depends only on the variables occurring in it, which we refer to as $Loc(\phi) = \{v : v \text{ occurs in } \phi\}$.

2.1 Mixed Predicates and Notify-All Updates

Our goal is to translate the program $\mathbb{P}$ into a Boolean program $\mathbb{B}$ such that, for any n, a suitably defined parallel execution of $\mathbb{B}$ by n threads overapproximates the parallel

execution of $\mathbb{P}$ by n threads. Let $E = \{\phi_1, \ldots, \phi_m\}$ be a set of predicates over $\mathbb{P}$, i.e. a set of Boolean expressions over variables in V. We say ϕ_i is

shared if $Loc(\phi_i) \subseteq V_S$,
local if $Loc(\phi_i) \subseteq V_L$, and
mixed otherwise, i.e. $Loc(\phi_i) \cap V_L \neq \emptyset$ and $Loc(\phi_i) \cap V_S \neq \emptyset$.

We declare, in $\mathbb{B}$, Boolean variables $\{b_1, \ldots, b_m\}$; the intention is that b_i tracks the value of ϕ_i during abstract execution of $\mathbb{P}$. We partition these Boolean variables into *shared* and *local* by stipulating that b_i is shared if ϕ_i is shared; otherwise b_i is local. In particular, **mixed predicates are tracked in local variables**. Intuitively, the value of a mixed predicate ϕ_i depends on the thread it is evaluated over. Declaring b_i shared would thus necessarily lose information. Declaring it local does not lose information, but, as the example in the Introduction has shown, is insufficient to guarantee a sound abstraction. We will see shortly how to solve this problem.

Each statement in $\mathbb{P}$ is now translated into a corresponding statement in $\mathbb{B}$. Statements related to flow of control are handled using techniques from standard predicate abstraction [4]; the distinction between shared, mixed and local predicates does not matter here. Consider an assignment to a variable v in $\mathbb{P}$ and a Boolean variable b of $\mathbb{B}$ with associated predicate ϕ. We first check whether variable v affects ϕ, written $\mathit{affects}(v, \phi)$. Given that in this section we assume no aliasing, this is the case exactly if $v \in Loc(\phi)$. If $\mathit{affects}(v, \phi)$ evaluates to false, b does not change. Otherwise, code needs to be generated to update b. This code needs to take into account the "flavors" of v and ϕ, which give rise to three different flavors of updates of b:

shared update: Suppose v and ϕ are both shared. An assignment to v is visible to all threads, so the truth of ϕ is modified for all threads. This is reflected in $\mathbb{B}$: by our stipulation above, the shared predicate ϕ is tracked by the *shared* variable b. Thus, we simply generate code to update b according to standard sequential predicate abstraction rules; the new value of b is shared among all threads.

local update: Suppose v is local and ϕ is local or mixed. An assignment to v is visible only by the active (executing) thread, so the truth of ϕ is modified only for the active thread. This also is reflected in $\mathbb{B}$: by our stipulation above, the local or mixed predicate ϕ is tracked by the *local* variable b. Again, sequential predicate abstraction rules suffice; the value of b changes only for the active thread.

notify-all update: Suppose v is shared and ϕ is mixed. An assignment to v is visible to all threads, so the truth of ϕ is modified for all threads. This is **not** reflected in $\mathbb{B}$: by our stipulation above, the mixed predicate ϕ is tracked by the *local* variable b, which will be updated only by the active thread. We solve this problem by (i) generating code to update b locally according to standard sequential predicate abstraction rules, and (ii) **notifying** all passive (non-active) threads of the modification of the shared variable v, so as to allow them to update their local copy of b.

We write $\mathit{must\_notify}(v, \phi)$ if the shared variable v affects the mixed predicate ϕ:

$$\mathit{must\_notify}(v, \phi) \quad = \quad \mathit{affects}(v, \phi) \ \wedge \ v \in V_S \ \wedge \ (Loc(\phi) \cap V_L \neq \emptyset) \, .$$

This formula evaluates to true exactly when it is necessary to notify passive threads of an update to v. What remains to be discussed in the rest of this section is how notifications are implemented in $\mathbb{B}$.

2.2 Implementing Notify-All Updates

We pause to recap some terminology and notation from sequential predicate abstraction [4]. We present our approach in terms of the Cartesian abstraction as used in [4], but our method in general is independent of the abstraction used. Given a set $E = \{\phi_1, \ldots, \phi_m\}$ of predicates tracked by variables $\{b_1, \ldots, b_m\}$, an assignment statement st is translated into the following code, in parallel for each $i \in \{1, \ldots, m\}$:

$$\begin{array}{lr} \textbf{if} \quad \mathcal{F}(\,WP(\ \ \phi_i, st)) \textbf{ then } b_i \ = \ \texttt{1} \\ \textbf{else if } \mathcal{F}(\,WP(\neg\phi_i, st)) \textbf{ then } b_i \ = \ \texttt{0} \\ \textbf{else} \qquad\qquad\qquad\qquad\quad b_i \ = \ \star\,. \end{array} \tag{1}$$

Here, $\star$ is the nondeterministic choice expression, WP the weakest precondition operator, and $\mathcal{F}$ the operator that strengthens an arbitrary predicate to a disjunction of cubes over the b_i. For example, with predicate ϕ :: $(l \ < \ \texttt{10})$ tracked by variable b, $E = \{\phi\}$ and statement st :: $++l$, we obtain $\mathcal{F}(\,WP(\phi, st)) = \mathcal{F}(l \ < \ \texttt{9}) = \mathit{false}$ and $\mathcal{F}(\,WP(\neg\phi, st)) = \mathcal{F}(l \ >= \ \texttt{9}) = (l \ >= \ \texttt{10}) = \neg\phi$, so that (1) reduces to

$$b \ = \ (\ b \ \texttt{?} \ \star \ \texttt{:} \ \texttt{0} \ )\,.$$

In general, (1) is often abbreviated using the assignment

$$b_i \ = \ \mathit{choose}(\mathcal{F}(\,WP(\phi_i, st)), \mathcal{F}(\,WP(\neg\phi_i, st)))\,,$$

where $\mathit{choose}(x, y)$ returns $\texttt{1}$ if x evaluates to true, $\texttt{0}$ if $(\neg x) \wedge y$ evaluates to true, and $\star$ otherwise. Abstraction of control flow guards uses the $\mathcal{G}$ operator, which is dual to $\mathcal{F}$: $\mathcal{G}(\phi) = \neg\mathcal{F}(\neg(\phi))$.

Returning to symmetry-aware predicate abstraction, if $\mathit{must\_notify}(v, \phi)$ evaluates to true for ϕ and v, predicate ϕ is mixed and thus tracked in $\mathbb{B}$ by some local Boolean variable, say b. Predicate-abstracting an assignment of the form $v \ = \ \chi$ requires updating the active thread's copy of b, as well as broadcasting an instruction to all passive threads to update their copy of b, in view of the new value of v. This is implemented using two assignments, which are executed in parallel. The first assignment is as follows:

$$b{=}\mathit{choose}(\mathcal{F}(\,WP(\phi, v = \chi)), \mathcal{F}(\,WP(\neg\phi, v = \chi)))\,. \tag{2}$$

This assignment has standard predicate abstraction semantics. Note that, since expression χ involves only local variables of the active thread and shared variables, only predicates over those variables are involved in the defining expression for b.

The second assignment looks similar, but introduces a new symbol:

$$[b]{=}\mathit{choose}(\mathcal{F}(\,WP([\phi], v = \chi)), \mathcal{F}(\,WP(\neg[\phi], v = \chi)))\,. \tag{3}$$

The notation $[b]$ stands for the copy of local variable b owned by some passive thread. Similarly, $[\phi]$ stands for the expression defining predicate ϕ, but with every local variable occurring in the expression replaced by the copy owned by the passive thread; this is the predicate ϕ in the context of the passive thread. Weakest precondition computation is with respect to $[\phi]$, while the assignment $v \ = \ \chi$, as an argument to WP, is

unchanged: v is shared, and local variables appearing in the defining expression χ must be interpreted as local variables of the *active* thread. Assignment (3) has the effect of updating variable b in every passive thread. We refer to Boolean programs involving assignments of the form $[b]$=... as *Boolean broadcast programs*; a formal syntax and semantics for such programs is given in [11].

Let us illustrate the above technique using a canonical example: consider the assignment $s = l$, for shared and local variables s and l, and define the mixed predicate $\phi :: (s == l)$. The first part of the above parallel assignment simplifies to $b = true$. For the second part, we obtain:

$$[b] = choose(\mathcal{F}(\,WP(s{==}[l], s{=}l)), \mathcal{F}(\,WP(\neg(s{==}[l]), s{=}l))\,) .$$

Computing weakest preconditions, this reduces to:

$$[b] = choose(\mathcal{F}(l{==}[l]), \mathcal{F}(\neg(l{==}[l]))\,) .$$

Precision of the Abstraction. To evaluate this expression further, we have to decide on the set of predicates available to the $\mathcal{F}$ operator to express the preconditions. If this set includes only predicates over the shared variables and the local variables of the passive thread that owns $[b]$, the predicate $l == [l]$ is not expressible and must be strengthened to $false$. The above assignment then simplifies to $[b] = choose(false, false)$, i.e. $[b] = \star$. The mixed predicates owned by passive threads are essentially *invalidated* when the active thread modifies a shared variable occurring in such predicates, resulting in a very imprecise abstraction.

We can exploit information stored in predicates local to other threads, to increase the precision of the abstraction. For maximum precision one could make *all* other threads' predicates available to the strengthening operator $\mathcal{F}$. This happens in the symmetry-oblivious approach sketched in the Introduction, where local and mixed predicates are physically replicated and declared at the global scope and can thus be made available to $\mathcal{F}$. Not surprisingly, in practice, replicating predicates in this way renders the abstraction prohibitively expensive. We analyze this experimentally in Section 5.

A compromise which we have found to work well in practice (again, demonstrated in Section 5) is to equip operator $\mathcal{F}$ with all shared predicates, all predicates of the passive thread owning $[b]$, **and also** predicates of the active thread. This arrangement is intuitive since the update of a passive thread's local variable $[b]$ is due to an assignment performed by some active thread. Applying this compromise to our canonical example: if both $s == [l]$ and $s == l$ evaluate to true before the assignment $s{=}l$, we can conclude that $[l] == l$ before the assignment, and hence $s == [l]$ after the assignment. Using $\oplus$ to denote exclusive-or, the assignment to $[b]$ becomes:

$$[b] = choose([b] \wedge b, [b] \oplus b)\,) .$$

2.3 The Predicate Abstraction Algorithm

We now show how our technique for soundly handling mixed predicates is used in an algorithm for predicate abstracting C-like programs. To present the algorithm compactly,

Algorithm 1 Predicate abstraction

Input: Program template $\mathbb{P}$, set of predicates $\{\phi_1, \ldots, \phi_m\}$
Output: Boolean program $\mathbb{B}$ over variables $b_1, \ldots, b_m$

1: **for each** statement d: stmt of $\mathbb{P}$ **do**
2: **if** stmt is **goto** $d_1, \ldots, d_m$ **then**
3: output "d: **goto** $d_1, \ldots, d_m$"
4: **else if** stmt is **assume** ϕ **then**
5: output "d: **assume** $\mathcal{G}(\phi)$"
6: **else if** stmt is $v = \chi$ **then**
7: $\{i_1, \ldots, i_f\} \leftarrow \{i \mid 1 \leq i \leq m \wedge \mathit{affects}(v, \phi_i)\}$
8: $\{j_1, \ldots, j_g\} \leftarrow \{j \mid 1 \leq j \leq m \wedge \mathit{must\_notify}(v, \phi_j)\}$
9: output "d: $\begin{pmatrix} b_{i_1}, & \mathit{choose}(\mathcal{F}(WP(\phi_{i_1}, v{=}\chi)), \mathcal{F}(WP(\neg\phi_{i_1}, v{=}\chi))), \\ \vdots & \vdots \\ b_{i_f}, & \mathit{choose}(\mathcal{F}(WP(\phi_{i_f}, v{=}\chi)), \mathcal{F}(WP(\neg\phi_{i_f}, v{=}\chi))), \\ [b_{j_1}], & \mathit{choose}(\mathcal{F}(WP([\phi_{j_1}], v{=}\chi)), \mathcal{F}(WP(\neg[\phi_{j_1}], v{=}\chi))), \\ \vdots & \vdots \\ [b_{j_g}] & \mathit{choose}(\mathcal{F}(WP([\phi_{j_g}], v{=}\chi)), \mathcal{F}(WP(\neg[\phi_{j_g}], v{=}\chi))) \end{pmatrix}$"

we assume a language with three types of statement: assignments, nondeterministic gotos, and assumptions. Control-flow can be modelled via a combination of gotos and assumes, in the standard way.

Algorithm 1 processes an input program template of this form and outputs a corresponding Boolean broadcast program template. Statements **goto** and **assume** are handled as in standard predicate abstraction: the former are left unchanged, while the latter are translated directly except that the guard of an **assume** statement is expressed over Boolean program variables using the $\mathcal{G}$ operator (see Section 2.2).

The interesting part of the algorithm for us is the translation of assignment statements. For each assignment, a corresponding parallel assignment to Boolean program variables is generated. The $\mathit{affects}$ and $\mathit{must\_notify}$ predicates are used to decide for which Boolean variables regular and broadcast assignments are required, respectively.

3 Symmetry-Aware Predicate Abstraction with Aliasing

So far, we have ignored complications introduced by pointers and aliasing. We now explain how symmetry-aware predicate abstraction is realized in practice, for C programs that manipulate pointers. We impose one restriction: we do not consider programs where a shared pointer variable, or a pointer variable local to thread i, can point to a variable local to thread j (with $j \neq i$). This arises only when a thread copies the address of a stack or thread-local variable to the shared state. This unusual programming style allows thread i to directly modify the local state of thread j at the C program level, breaking the asynchronous model of computation assumed by our method.

For ease of presentation we consider the scenario where program variables either have a base type (e.g. **int** or **float**), or pointer type (e.g. **int**$\star$ or **float**$\star\star$). Our method can be extended to handle records, arrays and heap-allocated memory. As in [4], we

assume that input programs have been processed so that l-values involve at most one pointer dereference.

Alias information is important in deciding, once and for all, whether predicates should be classed as local, mixed or shared. For example, let p be a local variable of type **int**$\star$, and consider predicate $\phi :: (\star p == 1)$. Clearly ϕ is not shared since it depends on local variable p. Whether ϕ should be regarded as a local or mixed predicate depends on whether p may point to the shared state: we regard ϕ as local if p can never point to a shared variable, otherwise ϕ is classed as mixed. Alias information also lets us determine whether a variable update may affect the truth of a given predicate, and whether it is necessary to notify other threads of this update. We now show how these intuitions can be formally integrated with our predicate abstraction technique. This involves suitably refining the notions of local, shared and mixed predicates, and the definitions of $affects$ and $must\_notify$ introduced in Section 2.

3.1 Aliasing, Locations of Expressions, and Targets of l-values

We assume the existence of a sound pointer alias analysis for concurrent programs, e.g. [21], which we treat as a black box. This procedure conservatively tells us whether a shared variable with pointer type may point to a local variable. As discussed at the start of Section 3, we reject programs where this is the case.[1] Otherwise, for a program template $\mathbb{P}$ over variables V, alias analysis yields a relation $\mapsto_d \subseteq V \times V$ for each program location d. For $v, w \in V$, if $v \not\mapsto_d w$ then v provably does not point to w at d.

For an expression ϕ and program point d, we write $loc(\phi, d)$ for the set of variables that it may be necessary to access in order to evaluate ϕ at d, during an arbitrary program run. We have $loc(z, d) = \emptyset$ for a constant value z, $loc(v, d) = \{v\}$, and $loc(\&v, d) = \emptyset$ for $v \in V$: evaluating an "address-of" expression requires no variable access, as addresses of variables are fixed at compile time. Finally, for any $k > 0$:

$$loc(\underbrace{\star \ldots \star}_{k} v, d) = \{v\} \cup \bigcup_{w \in V} \{loc(\underbrace{\star \ldots \star}_{k-1} w, d) \mid v \mapsto_d w\}.$$

Evaluating a pointer dereference $\star v$ involves reading both v and the variable to which v points. For other compound expressions, $loc(\phi, d)$ is defined recursively in the obvious way. The precision of $loc(\phi, d)$ is directly related to the precision of alias analysis.

For an expression ϕ, we define $Loc(\phi) = \cup_{1 \leq d \leq k} loc(\phi, d)$ as the set of variables that may need to be accessed to evaluate ϕ at an *arbitrary* program point during an arbitrary program run. Note how this definition of Loc generalizes that used in Section 2.

We finally write $targets(x, d)$ for the set of variables that may be modified by writing to l-value x at program point d. Formally, we have $targets(v, d) = \{v\}$ and $targets(\star v, d) = \{w \in V \mid v \mapsto_d w\}$. Note that $targets(\star v, d) \neq loc(\star v, d)$: ***Writing*** through $\star v$ modifies only the variable to which v points, while ***reading*** the value of $\star v$ involves reading the value of v, to determine which variable w is pointed to by v, and then reading the value of w.

[1] This also eliminates the possibility of thread i pointing to variables in thread $j \neq i$: the address of a variable in thread j would have to be communicated to thread i via a shared variable.

3.2 Shared, Local and Mixed Predicates in the Presence of Aliasing

In the presence of pointers, we define the notion of a predicate ϕ being shared, local, or mixed exactly as in Section 2.1, but with the generalization of Loc presented in Section 3.1. In Section 2.1, without pointers, we could classify ϕ purely syntactically, based on whether any shared variables appear in ϕ. In the presence of pointers, we must classify ϕ with respect to alias information; our definition of Loc takes care of this.

Recall from Section 2.1 that we defined $\mathit{affects}(v, \phi) = (v \in \mathit{Loc}(\phi))$ to indicate that updating variable v may affect the truth of predicate ϕ. In the presence of pointers, this definition no longer suffices. The truth of ϕ may be affected by assigning to l-value x if x may alias some variable on which ϕ depends. Whether this is the case depends on the program point at which the update occurs. Our definitions of loc and $targets$ allow us to express this:

$$\mathit{affects}(x, \phi, d) = (\mathit{targets}(x, d) \cap \mathit{loc}(\phi, d) \neq \emptyset).$$

We also need to determine whether an update affects the truth of a predicate only for the thread executing the update, or for all threads. The definition of $must\_notify$ presented in Section 2.1 needs to be adapted to take aliasing into account. At first sight, it seems that we must simply parameterise $\mathit{affects}$ according to program location, and replace the conjunct $v \in V_S$ with the condition that x may target some shared variable:

$$\begin{aligned} must\_notify(x, \phi, d) = \mathit{affects}(x, \phi, d) \ \wedge \ & (\mathit{Loc}(\phi) \cap V_L \neq \emptyset) \\ & \wedge (\mathit{targets}(x, d) \cap V_S \neq \emptyset) \ . \end{aligned}$$

However, this is unnecessarily strict. We can refine the above definition to minimise the extent to which notifications are required, as follows:

$$must\_notify(x, \phi, d) = (\mathit{targets}(x, d) \cap \mathit{Loc}(\phi) \cap V_S \neq \emptyset) \wedge (\mathit{Loc}(\phi) \cap V_L \neq \emptyset) \ .$$

The refined definition avoids the need for thread notification in the following scenario. Suppose we have shared variables s and t, local variable l, local pointer variable p, and predicate $\phi :: (s > l)$. Consider an assignment to $\star p$ at program point d. Suppose that alias analysis tells us exactly $p \mapsto_d t$ and $p \mapsto_d l$. The only shared variable that can be modified by assigning through $\star p$ at program point d is t, and the truth of ϕ does not depend on t. Thus the assignment does *not* require a "notify-all" with respect to ϕ. Working through the definitions, we find that our refinement of $must\_notify$ correctly determines this, while the naïve extension of $must\_notify$ from Section 2.1 would lead to an unnecessary "notify-all".

The predicate abstraction algorithm (Alg. 1) can now be adapted to handle pointers: parameter d is simply added to the uses of $\mathit{affects}$ and $must\_notify$. Handling of pointers in weakest preconditions works as in standard predicate abstraction [4], using Morris's general axiom of assignment [19].

4 Closing the CEGAR Loop

We have integrated our novel technique for predicate-abstracting symmetric concurrent programs with the SATABS CEGAR-based verifier [10], using the Cartesian abstraction method and the maximum cube length approximation [4]. We now sketch how we

have adapted the other phases of the CEGAR loop: model checking, simulation and refinement, to accurately handle concurrency.

Model checking Boolean broadcast programs. Our predicate abstraction technique generates a concurrent Boolean broadcast program. The extended syntax and semantics for broadcasts mean that we cannot simply use existing concurrent Boolean program model checkers such as BOOM [6] for the model checking phase of the CEGAR loop. We have implemented a prototype extension of BOOM, which we call B-BOOM. B-BOOM extends the counter abstraction-based symmetry reduction capabilities of BOOM to support broadcast operations. Symbolic image computation for broadcast assignments is significantly more expensive than image computation for standard assignments. In the context of BOOM it involves 1) converting states from counter representation to a form where the individual local states of threads are stored using distinct BDD variables, 2) computing the intersection of $n-1$ successor states, one for each passive thread paired with the active thread, and 3) transforming the resulting state representation back to counter form using Shannon expansion. The expense of image computation for broadcasts motivates the careful analysis we have presented in Sections 2 and 3 for determining tight conditions under which broadcasts are required.

Simulation. To determine the authenticity of abstract error traces reported by B-BOOM we have extended the SATABS simulator. The existing simulator extracts the control flow from the trace. This is mapped back to the original C program and translated into a propositional formula (using standard techniques such as single static assignment conversion and bitvector interpretation of variables). The error is spurious exactly if this formula is unsatisfiable. In the concurrent case, the control flow information of an abstract trace includes which thread executes actively in each step. We have extended the simulator so that each local variable involved in a step is replaced by a fresh indexed version, indicating the executing thread that owns the variable. The result is a trace over the replicated C program $\mathbb{P}^n$, which can be directly checked using a SAT/SMT solver.

Refinement. Our implementation performs refinement by extracting new predicates from counterexamples via weakest precondition calculations. This standard method requires a small modification in our context: weakest precondition calculations generate predicates over shared variables, and local variables of *specific* threads. For example, if thread 1 branches according to a condition such as $l < s$, where l and s are local and shared, respectively, weakest precondition calculations generate the predicate $l_1 < s$, where l_1 is thread 1's copy of l. Because our predicate abstraction technique works at the template program level, we cannot add this predicate directly. Instead, we generalize such predicates by removing thread indices. Hence in the above example, we add the mixed predicate $l < s$, for *all* threads.

An alternative approach is to refine the abstract transition relation associated with the Cartesian abstraction based on infeasible steps in the abstract counterexample [3]. We do not currently perform such refinement, as correctly refining abstract transitions involving broadcast assignments is challenging and requires further research.

5 Experimental Results

We evaluate the SATABS-based implementation of our techniques using a set of 14 concurrent C programs. We consider benchmarks where threads synchronize via locks (lock-based), or in a lock-free manner via atomic *compare-and-swap* (cas) or *test-and-set* (tas) instructions. The benchmarks are as follows:[2]

Increment, Inc./Dec. (lock-based and cas-based): a counter, concurrently incremented, or incremented and decremented, by multiple threads [20]

Prng (lock-based and cas-based concurrent pseudorandom number generator [20]

Stack (lock-based and cas-based thread-safe stack implementation, supporting concurrent pushes and pops, adapted from an Open Source IBM implementation[3] of an algorithm described in [20]

Tas Lock, Ticket Lock (tas-based) concurrent lock implementations [18]

FindMax, FindMaxOpt (lock-based and cas-based) implementations of parallel reduction operation [2] to find maximum element in array. **FindMax** is a basic implementation, and **FindMaxOpt** an optimized version where threads reduce communication by computing a partial maximum value locally.

Mixed predicates were required for verification to succeed in all but two benchmarks: lock-based *Prng*, and lock-based *Stack*. For each benchmark, we consider verification of a safety property, specified via an assertion. We have also prepared a buggy version of each benchmark, where an error is injected into the source code to make it possible for this assertion to fail. We refer to correct and buggy versions of our benchmarks as *safe* and *unsafe*, respectively.

All experiments are performed on a 3GHz Intel Xeon machine with 40 GB RAM, running 64-bit Linux, with separate timeouts of 1h for the abstraction and model checking phases of the CEGAR loop. Predicate abstraction uses a maximum cube length of 3 for all examples, and MiniSat 2 (compiled with full optimizations) is used for predicate abstraction and counterexample simulation.

Symmetry-aware vs. symmetry-oblivious method. We evaluate the scalability of our symmetry-aware predicate abstraction technique (SAPA) by comparing it against the symmetry-oblivious predicate abstraction (SOPA) approach sketched near the end of Section 1, for verification of correct versions of our benchmarks. Recall that in SOPA, an n-thread symmetric concurrent program is expanded so that variables for all threads are explicitly duplicated, and n copies of all non-shared predicate are generated. The expanded program is then abstracted over the expanded set of predicates, using standard predicate abstraction. This yields a Boolean program for each thread; the parallel composition of these n Boolean programs is explored by a model checker. Because symmetry is not exploited, and no broadcasts are required, any Boolean program model checker can be used. We have tried both standard BOOM [6] (without symmetry reduction) and Cadence SMV [17] to model check expanded Boolean programs. In all cases, we found BOOM to be faster than SMV, thus we present results only for BOOM.

[2] All benchmarks and tools are available online: `http://www.cprover.org/SAPA`

[3] `http://amino-cbbs.sourceforge.net`

		Pred.			SOPA		SAPA					Pred.			SOPA		SAPA		
Benchmark	n	S	L	M	Abs	MC	#Its	Abs	MC	**Benchmark**	n	S	L	M	Abs	MC	#Its	Abs	MC
Increment	6	2	1	1	13	5	2	**1**	**<1**	Prng	1	1	5	0	<1	<1	2	<1	<1
(lock-based)	8				29	152			**1**	(lock-based)	6	1	12	0	69	21	5	**26**	**<1**
	9				40	789			**1**		7				83	191			**1**
	10				56	T.O.			**2**		8				96	T.O.			**2**
	12								**7**		16								**142**
	14								**24**		26								**3023**
	16								**100**	Prng	1	1	5	0	<1	<1	2	<1	<1
	18								**559**	(cas-based)	3	1	14	2	**29**	**<1**	5	48	1
	20								**2882**		4				**40**	**12**		48	38
Increment	4	2	4	2	50	12	3	**6**	**1**		5				57	**1049**		**48**	1832
(cas-based)	5				94	358			**13**	FindMax	6	0	0	1	5	30	1	**<1**	**<1**
	6				159	T.O.			**116**	(lock-based)	7				9	244			**1**
	7								**997**		8				14	T.O.			**1**
Inc./Dec.	4	6	3	2	71	6	3	**11**	**2**		16								**125**
(lock-based)	5				132	656			**50**		25								**3005**
	6				231	T.O.			**1422**	FindMax	3	0	5	1	4	7	3	**1**	**2**
Inc./Dec.	2	6	10	4	125	<1	5	**78**	<1	(cas-based)	4				8	407			**368**
(cas-based)	3				372	6			**3**	FindMaxOpt	4	0	1	1	3	40	1	**<1**	**3**
	4				872	4043			**252**	(lock-based)	5				6	1356			**33**
Tas Lock	3	4	2	2	3	2	3	**1**	**<1**		6				11	T.O.			**269**
(tas-based)	4				9	114			**4**		7								**1773**
	5				14	T.O.			**72**	FindMaxOpt	3	0	6	1	9	11	3	**3**	**2**
	6								**725**	(cas-based)	4				15	1097			**61**
Ticket Lock	2	12	3	4	554	1	2	**251**	1		5				22	T.O.			**1240**
(tas-based)	3				1319	3			**1**	Stack	3	1	4	0	<1	14	2	<1	**8**
	4				T.O.	–			**2**	(lock-based)	4				<1	945			**374**
	6								**62**	Stack	3	1	4	1	2	29	2	**<1**	**14**
	8								**2839**	(cas-based)	4				8	3408			**813**

Table 1. Comparison of symmetry-aware and symmetry-oblivious predicate abstraction over our benchmarks. For each configuration, the fastest abstraction and model checking times are in bold.

Table 1 presents the results of the comparison. For each benchmark and each approach we show, for interesting thread counts (including the largest thread count that could be verified with each approach), the number of local, mixed, and shared predicates (L, M, S) over the template program that were needed to prove the program safe (which varies slightly with n), and the elapsed time for predicate abstraction and model checking. For each configuration, the fastest abstraction and model checking times are shown in bold. Model checking uses standard BOOM, without symmetry reduction (SOPA) and B-BOOM, our extension to BOOM discussed in Section 4 (SAPA), respectively. T.O. indicates a timeout; succeeding cells are then marked '–'.

The results show that in the vast majority of cases our novel SAPA technique significantly outperforms SOPA, both in terms of abstraction and model checking time. The former can be attributed to the fact that, with SOPA, the number of predicates grows according to the number of threads considered, while with SAPA, this is thread count-independent. The latter is due to the exploitation of template-level symmetry by B-BOOM. The exception to this is the cas-based *Prng* benchmark, for which SAPA yields slower verification. Profiling with respect to this benchmark shows that the inferior performance of the model checker with SAPA comes from the expense of performing broadcast operations. Note, however, that the ratio between model checking times for SOPA and SAPA on this benchmark decreases as the thread counts go up.

Benchmark	Symmetry-Aware				Mixed as local				Mixed as shared			
	Safe	n	Unsafe	n	Safe	n	Unsafe	n	Safe	n	Unsafe	n
Increment (lock-based)	safe	>10	unsafe	2	safe	>10	**error**	2	safe	10	**error**	2
Incr. (cas-based)	safe	7	unsafe	2	safe	8	**safe**	5	**error**	2	**error**	2
Incr./Dec. (lock-based)	safe	6	unsafe	3	safe	>10	**safe**	>10	safe	>10	unsafe	3
Incr./Dec. (cas-based)	safe	4	unsafe	3	safe	6	**safe**	8	**error**	2	**error**	3
Tas Lock (tas-based)	safe	7	unsafe	2	safe	8	**error**	2	**error**	2	**error**	2
Ticket Lock (tas-based)	safe	8	unsafe	3	safe	>10	unsafe	3	safe	5	unsafe	3
Prng (lock-based)	safe	>10	unsafe	2	safe	>10	unsafe	2	safe	>10	unsafe	2
Prng (cas-based)	safe	5	unsafe	3	safe	7	unsafe	3	safe	6	unsafe	3
FindMax (lock-based)	safe	>10	unsafe	2	safe	>10	**safe**	>10	safe	2	**error**	2
FindMax (cas-based)	safe	4	unsafe	2	safe	5	**safe**	4	safe	2	**safe**	1
FindMaxOpt (lock-based)	safe	7	unsafe	2	safe	7	**safe**	6	**error**	2	**error**	2
FindMaxOpt (cas-based)	safe	5	unsafe	1	safe	5	unsafe	1	**error**	2	unsafe	1
Stack (lock-based)	safe	4	unsafe	4	safe	4	unsafe	4	safe	4	unsafe	4
Stack (cas)	safe	4	unsafe	2	safe	4	**safe**	6	safe	4	**error**	2

Table 2. Comparison of sound and unsound approaches; incorrect results in bold.

Comparison with unsound methods. In Section 1, we described two naïve solutions to the mixed predicate problem: uniformly using local or shared Boolean variables to represent mixed predicates, and then performing standard predicate abstraction. We denote these approaches *mixed as local* and *mixed as shared*, respectively. Although we demonstrated theoretically in Section 1 that both methods are unsound, it is interesting to see how they perform in practice. Table 2 shows the results of applying CEGAR-based model checking to safe and unsafe versions of our benchmarks, using our sound technique, and the unsound *mixed as local* and *mixed as shared* approaches. In all cases, B-BOOM is used for model checking. For the sound technique, we show the largest thread count for which we could prove correctness of each safe benchmark, and the smallest thread count for which a bug was revealed in each unsafe benchmark. The other columns illustrate how the unsound techniques differ from this, where "error" indicates a refinement failure: it was not possible to extract further predicates from spurious counterexamples. Bold entries indicate cases where the unsound approaches produce incorrect, or inconclusive results.[4] The number of cases where the unsound approaches produce false negatives, or lead to refinement failure, suggest that little confidence can be placed in these techniques, even for purposes of falsification. This justifies the more sophisticated and, crucially, sound techniques developed in this paper.

6 Related Work and Conclusion

There exists a large body of work on the different stages of CEGAR-based program analysis. We focus here on the abstraction stage, which is at the heart of this paper.

[4] We never expect the unsound techniques to report conclusively that a safe benchmark is unsafe: this would require demonstrating a concrete error trace in the original, safe, program.

Predicate abstraction goes back to the foundational work by Graf and Saïdi [12]. It was first presented for *sequential* programs in a mainstream language (C) by Ball, Majumdar, Millstein, Rajamani [4] and implemented as part of the SLAM project. We have found many of the optimizations suggested by [4] to be useful in our implementation as well. Although SLAM has had great success in finding real bugs in system-level code, we are not aware of any extensions of it to concurrent programs (although this option is mentioned by the authors of [4]). We attribute this to a large part to the infeasibility, at the time, to handle realistic multi-threaded Boolean programs. We believe our own work on BOOM [6] has made progress in this direction that has made it attractive again to address concurrent predicate abstraction.

We are not aware of other work that presents precise solutions to the problem of "mixed predicates". Some approaches avoid it by syntactically disallowing such predicates, e.g. [22], whose authors don't discuss, however, the reasons for (or, indeed, the consequences of) doing so. Another approach *havocks* (assigns nondeterministically) global variables that may be affected by an operation [8], thus taking away the mixed flavor from certain predicates. In yet other work, "algorithmic circumstances" may make the treatment of such predicates unnecessary. The authors of [15], for example, use predicate abstraction to finitely represent the environment of a thread in multi-threaded programs. The "environment" consists of assumptions on how threads may manipulate the *shared* state of the program, irrespective of their local state. Our case of *replicated* threads, in which mixed predicates would constitute a problem, is only briefly mentioned in [15]. In [7], an approach is presented that handles *recursive* concurrent C programs. The abstract transition system of a thread (a pushdown system) is formed over predicates that are projected to the global or the local program variables and thus cannot compare "global against local" directly. As we have discussed, some reachability problems cannot be solved using such restricted predicates. We conjecture this problem is one of the potential causes of non-termination in the algorithm of [7].

Other model checkers with some support for concurrency include BLAST, which does not allow general assertion checking [14], and MAGIC [7], which does not support shared variable communication, making a comparison to our work little meaningful.

In conclusion, we mention that building a CEGAR-based verification strategy is a tremendous effort, and our work so far can only be the beginning of such effort. We have assumed a very strict (and unrealistic) memory model that guarantees atomicity at the statement level. One can work soundly with the former assumption by pre-processing input programs so that the shared state is accessed only via word-length reads and writes, ensuring that all computation is performed using local variables. Extending our approach to weaker memory models, building on existing work in this area [16, 1], is future work. Our plans also include a more sophisticated refinement strategy, drawing upon recent results on abstraction refinement for concurrent programs [13], and a more detailed comparison with existing approaches that circumvent the mixed-predicates problem using other means.

Acknowledgments. We are grateful to Gérard Basler for assistance with BOOM, and Michael Tautschnig for his insightful comments on an earlier draft of this work.

References

1. J. Alglave. *A Shared Memory Poetics*. PhD thesis, Université Paris 7 and INRIA, 2010. http://moscova.inria.fr/~alglave/these.
2. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, San Francisco, CA, USA, 2002.
3. T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining approximations in software predicate abstraction. In *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 388–403. Springer, 2004.
4. T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.
5. T. Ball and S. Rajamani. The SLAM project: debugging system software via static analysis. In *Principles of Programming Languages (POPL)*, pages 1–3, 2002.
6. G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening. Context-aware counter abstraction. *Formal Methods in System Design (FMSD)*, 36(3):223–245, 2010.
7. S. Chaki, E. Clarke, N. Kidd, T. Reps, and T. Touili. Verifying concurrent message-passing C programs with recursive calls. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, pages 334–349. Springer, 2006.
8. A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri. Verifying SystemC: a software model checking approach. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2010.
9. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 2003.
10. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design (FMSD)*, pages 105–127, 2004.
11. A. F. Donaldson, A. Kaiser, D. Kroening, and T. Wahl. Symmetry-aware predicate abstraction for shared-variable concurrent programs (extended technical report). *CoRR*, abs/1102.2330, 2011.
12. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification (CAV)*, LNCS, pages 72–83. Springer, 1997.
13. A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, pages 331–344. ACM, 2011.
14. T. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *Programming Language Design and Implementation (PLDI)*, pages 1–13, 2004.
15. T. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *CAV*, LNCS, pages 262–274. Springer, 2003.
16. J. Lee and D. Padua. Hiding relaxed memory consistency with a compiler. *IEEE Transactions on Computers*, 50:824–833, 2001.
17. K. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.
18. J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.
19. J. Morris. A general axiom of assignment. In *Theoretical Foundations of Programming Methodology, Lecture Notes of an International Summer School*, pages 25–34. D. Reidel Publishing Company, 1982.
20. T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.
21. R. Rugina and M. C. Rinard. Pointer analysis for multithreaded programs. In *PLDI*, pages 77–90, 1999.
22. N. Timm and H. Wehrheim. On symmetries and spotlights – verifying parameterised systems. In *ICFEM*, LNCS, pages 534–548. Springer, 2010.