Noname manuscript No.
(will be inserted by the editor)

Counterexample-Guided Abstraction Refinement for Symmetric Concurrent Programs

Alastair F. Donaldson · Alexander Kaiser · Daniel Kroening · Michael Tautschnig · Thomas Wahl

Received: date / Accepted: date

Abstract Predicate abstraction and counterexample-guided abstraction refinement (CEGAR) have enabled finite-state model checking of software written in mainstream programming languages. This combination of techniques has been successful in analysing system-level sequential `C` code. In contrast, there is little evidence of fruitful applications of CEGAR to shared-variable *concurrent* software. We attribute this gap to the lack of abstraction strategies that permit a scalable analysis of the resulting multi-threaded Boolean programs. The goal of this paper is to close this gap. We have developed a *symmetry-aware* CEGAR technique: it takes into account the replicated structure of programs that consist of many threads executing the same procedure, and generates a Boolean program template whose multi-threaded execution soundly overapproximates the original concurrent program. State explosion during model checking parallel instantiations of this template can now be absorbed by exploiting symmetry. We have implemented our method in a tool, SYMMPA, and demonstrate its superior performance over alternative approaches on a range of synchronisation programs.

Supported by EPSRC projects EP/G026254/1 and EP/G051100/1 and ERC project 280053.

A.F. Donaldson
Department of Computing, Imperial College London
E-mail: alastair.donaldson@imperial.ac.uk

A. Kaiser, D. Kroening, M. Tautschnig
Department of Computer Science, University of Oxford
E-mail: {alexander.kaiser, daniel.kroening, michael.tautschnig}@cs.ox.ac.uk

Th. Wahl
College of Computer and Information Science, Northeastern University
E-mail: wahl@ccs.neu.edu

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* [15] 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 an imperative-language program (e.g., written in C) into a finite-state *Boolean* program [2], 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 [7]. 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 flow-related errors in low-level operating system code [3].

The majority of concurrent software is written using mainstream APIs such as POSIX threads (pthreads) in C/C++. Multiple threads are spawned (up front or dynamically) to execute a given procedure in parallel, communicating via shared global variables. For a fixed number n of threads, this concurrency model turns a sequential program $\mathbb{P}$ into an n-thread concurrent program $\mathbb{P}^n$ in which global program variables are *shared* (readable and writeable) by all threads. Procedure-local variables become thread-*local*: each of the n threads executing the procedure owns a distinct copy, which is inaccessible to other threads. To execute $\mathbb{P}^n$, a thread is chosen by the scheduler to execute a statement of $\mathbb{P}$, which may modify shared variables and the thread's local variables. For such shared-variable concurrent programs, predicate abstraction success stories similar to that of SLAM are few and far between (see Section 6 for a detailed discussion). 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, as experience has shown [11].

The key to obtaining scalability is to exploit the *symmetry* naturally featured by these programs, namely the invariance of $\mathbb{P}^n$ under permutations of the involved threads. Fortunately, much progress has recently been made on analysing *replicated* non-recursive Boolean programs executed concurrently by many threads [4, 21, 29]. In this paper, we present a CEGAR technique for concurrent programs that leverages this recent progress. Our technique translates a non-recursive program $\mathbb{P}$ in a C-like language, 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 program $\mathbb{P}^n$. Our approach permits predicates in the form of arbitrary expressions over the variables a thread has access to, local or global. The abstract program $\mathbb{B}$ is refined in response to spurious counterexamples discovered in $\mathbb{B}^n$, in a way that preserves symmetry. Operating at the template level, our abstraction and refinement methods make exploiting symmetry during model checking of $\mathbb{B}^n$ straightforward. We thus refer to our novel CEGAR technique as *symmetry-aware* counterexample guided abstraction refinement.

We first show that sequential predicate abstraction techniques cannot directly be applied to symmetric concurrent programs at the template level if the abstraction relies on *mixed* predicates: predicate expressions that contain both shared and thread-local variables. We then outline our novel technique for soundly handling mixed predicates (Section 2). We define a simple imperative language for writing concurrent programs, and an extended Boolean programming language to be used as a target for abstracting such programs (Section 3). This allows us to formally present our symmetry-aware CEGAR technique (Section 4). We then describe an implementation of our technique as a tool, SYMMPA, and experimental results showing the effectiveness of SYMMPA in comparison to a recent CEGAR-based model checker for concurrent programs that does *not* exploit symmetry (Section 5).

We present our results for a language without pointers and aliasing. For an extension that covers these language features, we refer the reader to our prior work [11]. Concurrent threads are assumed to interleave with statement-level granularity; see the discussion in Section 7 on this subject.

2 Overview: Symmetry-Aware Predicate Abstraction

In this section we illustrate the basic ideas of our approach. We present programs as code fragments consisting of a procedure to be executed by multiple concurrent threads. A program declares *shared* global variables, visible to all threads, and *local* variables, of which each thread has a private copy.

Shared, Local and Mixed Predicates. As in traditional predicate abstraction for imperative programs [2], the Boolean program $\mathbb{B}$ to be built from the program $\mathbb{P}$ will be defined over Boolean variables, one for each predicate being used during abstraction. 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 variables of $\mathbb{P}$, the "shared" and "local" attributes clearly depend on the attributes of the variables in $\mathbb{P}$ a predicate is formulated over. We therefore classify predicates as follows.

Definition 1 *A **local** predicate refers solely to local variables. A **shared** predicate refers solely to shared variables. Any other predicate is **mixed**.*

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 variables; a change that is not visible to any other thread. This locality is inherited by the Boolean program: 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 less obvious: Consider the following program $\mathbb{P}$ and a corresponding Boolean program $\mathbb{B}$ which tracks the mixed predicate $s \ != \ l$ in a **local** Boolean variable b ($\mathbb{B}$ is the Boolean program obtained from $\mathbb{P}$ via the Cartesian abstraction [2]):

$\mathbb{P}$:

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

$\mathbb{B}$:

```
0: local bool b = 1;

1: assert b;
2: b = b ? ⋆ : 1;
```

Let $\mathbb{P}^2$ be 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; the assertion fails. On the other hand, $(\mathbb{B}')^2$ is safe. We conclude that $(\mathbb{B}')^2$ is **unsound** for $(\mathbb{P}')^2$.

These examples show that a mixed predicate cannot soundly be tracked by a standard shared or a local variable. It is also fairly easy to see that simply prohibiting mixed predicates is not an option, as it renders some trivial bug-free programs **unverifiable** using predicate abstraction [12]. As a result, the abstraction process needs to be amended to accommodate mixed predicates explicitly.

A technically simple but naive solution is to "resolve" mixed predicates by disambiguating local variables: we 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. 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 will be 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 a *symmetry-oblivious* approach does not scale beyond a very small number of threads [11]. The goal of this paper is a CEGAR technique that soundly handles mixed predicates without compromising the efficiency afforded by symmetry reduction.

Mixed Predicates and Notify-All Updates. We now describe informally our solution to soundly handling mixed predicates. Let $E = \{\phi_1, \ldots, \phi_m\}$ be a set of predicates over $\mathbb{P}$, i.e., a set of Boolean expressions over the shared and local variables declared in $\mathbb{P}$. 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 above 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 [2]; 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. If the assignment does not affect ϕ, i.e., if v does not appear in ϕ, b does not change. Otherwise a statement needs to be generated to update b. This statement needs to take into account the "flavors" of v and ϕ, which give rise to three different forms 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 a statement 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 to the active (executing) thread, hence 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 $\mathbb{B}$: as stipulated 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, to allow them to update their local copy of b.

Note that the cases "v shared, ϕ local" and "v local, ϕ shared" cannot occur.

In order to illustrate how **Notify-All** updates can be realised in practice, let us consider the concrete example of the assignment $s = l$, for shared and local variables s and l, and the mixed predicate $\phi :: (s == l)$. The assignment causes the active thread's copy of this predicate to evaluate to $true$; what it does to other threads is not clear without inspecting them. A reasonable modular translation of this assignment into the Boolean program might therefore be:

$$\textbf{havoc } \{\textbf{all}(b)\} \textbf{ constrain } \mathtt{a}.b' \tag{1}$$

The new statement **havoc**s (assigns nondeterministically) **all** threads' copies of variable b, under the constraint that the active thread $\mathtt{a}$'s copy is forced to evaluate to true: **constrain** $\mathtt{a}.b'$ forces the post-havoc (primed) value of $\mathtt{a}$'s copy of b to be *true*.

We can increase the precision of statement (1) by inferring knowledge about the other threads' copies of b. Consider some passive thread p, and suppose ϕ holds for

both $\mathtt{a}$ and p before the assignment $s = l$, thus $\mathtt{a}.l == s == p.l$. In this case, ϕ holds for both $\mathtt{a}$ and p after the assignment, too. On the other hand, if $\mathtt{a}$ and p *disagree* on ϕ initially, then $\mathtt{a}.l != p.l$. In this case, after the assignment $s = l$ we will have $s != p.l$. This reasoning allows us to strengthen statement (1), as follows:

$$\begin{aligned}\textbf{havoc}\ \{\textbf{all}(b)\}\ \textbf{constrain}\ \forall\, p \neq \mathtt{a}\ .\ \mathtt{a}.b' \wedge (\mathtt{a}.b \wedge p.b \Rightarrow p.b') \\ \wedge (\mathtt{a}.b \oplus p.b \Rightarrow \neg p.b')\end{aligned} \tag{2}$$

where $\oplus$ denotes exclusive-or. This statement restricts the states that can be reached after havocking b across all threads to those where b holds for the active thread, and such that for any other (i.e., passive) thread p, the value of b for thread p after havocking is consistent with the state of the active thread.

Abstracting over multiple threads. Boolean program statements such as (2) that may explicitly refer to variables of passive threads are not only useful for **Notify-All** updates. Consider the mixed predicate $\phi :: (s == l)$ and the local predicate $\psi :: (l == 4)$, tracked by local Boolean variables b and c, respectively, and the assignment $l = s$ (which only affects the active thread's state). The statement:

$$\textbf{havoc}\ \{\mathtt{a}.b, \mathtt{a}.c\}\ \textbf{constrain}\ \mathtt{a}.b' \wedge (\mathtt{a}.b \wedge \mathtt{a}.c \Rightarrow \mathtt{a}.c') \wedge (\mathtt{a}.b \oplus \mathtt{a}.c \Rightarrow \neg \mathtt{a}.c') \tag{3}$$

soundly over-approximates the assignment. Here, we use the notation $\{\mathtt{a}.b, \mathtt{a}.c\}$ to indicate that local variables b and c belonging to the *active* thread should be havocked; the values of these variables in other threads are not changed.

However, statement (3) does not take into account information available from other threads in the Boolean program. Suppose there is some thread p different from the active thread such that $p.b$ and $p.c$ both hold. In a corresponding concrete state, this means that $p.l == 4$ and $s == p.l$, implying $s == 4$. Similarly, if exactly one of $p.b$ and $p.c$ hold, we can conclude $s != 4$. This reasoning allows us to translate the assignment $l = s$ with more precision than in (3), as follows:

$$\begin{aligned}\textbf{havoc}\ \{\mathtt{a}.b, \mathtt{a}.c\}\ \textbf{constrain}\ \forall\, p \neq \mathtt{a}\ .\ \mathtt{a}.b' \wedge (((\mathtt{a}.b \wedge \mathtt{a}.c) \vee (p.b \wedge p.c)) \Rightarrow \mathtt{a}.c') \\ \wedge (((\mathtt{a}.b \oplus \mathtt{a}.c) \vee (p.b \oplus p.c)) \Rightarrow \neg \mathtt{a}.c')\end{aligned}$$

It is easy to refine these examples to cases where the precision benefits from using predicates over *several* non-active threads; we omit the details. In practice, as demonstrated in Section 5, we have found that predicates involving the active thread and a *single* passive thread are a good compromise between increased precision and increased predicate complexity. This finding is intuitive since the update of a passive thread p's local variable $p.b$ is due to an assignment performed by some active thread. Nevertheless, because our benchmark set does not rule out the utility of predicate abstraction over multiple passive threads in general, and because this form of predicate abstraction is of theoretical interest, we present a fully general CEGAR technique that allows constraints over multiple passive threads.

3 Concrete and Abstract Programming Languages

Before formalising our CEGAR procedure (Section 4), we define a concrete C-like programming language, and a Boolean programming language with **Notify-All**.

prog	::=	**shared** shared_name = lit; . . . ; shared_name = lit; **local** local_name = lit; . . . ; local_name = lit; 1: stmt; . . . ; k: stmt;
stmt	::=	shared_name = expr | local_name = expr
	|	**assume** expr | **goto** pc, . . . , pc
expr	::=	lit | shared_name | local_name | compound expression
lit	::=	integer literal
pc	::=	integer in range $1 \ldots k$

shared_name, local_name denote elements of SharedNames and LocalNames respectively

Fig. 1 Syntax for Symmetric Concurrent Programming Language (SCPL)

3.1 Symmetric Concurrent Programming Language

Let SharedNames and LocalNames be disjoint sets of names. Figure 1 provides the syntax for a simple language for writing symmetric concurrent programs, which we call Symmetric Concurrent Programming Language (SCPL), defined over these sets.

An SCPL program is a template to be executed by some number of concurrent threads. The number of threads is not part of the program description: an SCPL program can be instantiated with any positive number of threads. Each program declares a set of *shared* integer variables with distinct names drawn from SharedNames, followed by a set of *local* integer variables with distinct names drawn from LocalNames. When the program is instantiated, there will be *one* instance of each shared variable, shared among all threads, while each thread will own a separate copy of every local variable. The names of shared and local variables occurring in an SCPL program $\mathbb{P}$ are denoted $V_S^{\mathbb{P}} \subseteq$ SharedNames and $V_L^{\mathbb{P}} \subseteq$ LocalNames, respectively.

Program behaviour is defined by a sequence of statements numbered 1 through k (for some $k \geq 1$). During execution, each thread has an associated program counter. An execution step consists of selecting a thread, and having this thread execute the statement associated with its program counter.

There are four forms of statements: assignments to a shared or a local variable have the obvious meaning; **assume** ϕ causes thread execution (and therefore program execution) to halt if ϕ does not hold in the context of the executing thread; **goto** causes the executing thread's program counter to be set nondeterministically to one of a given set of values. Expressions occurring on the right-hand sides of assignments and as arguments to **assume** are defined over program variables and integer literals; compound expressions are formed in the usual way. Conditional flow of control and looping can be modeled using a combination of **goto** and **assume** statements.

We do not define a formal semantics for SCPL, since it is similar to many simple concurrent programming languages proposed in the literature. The crucial feature of SCPL is that threads are fully symmetric: nothing in the language allows a distinction to be made between one thread and another. This allows us to translate an SCPL program template into a Boolean program template, using the techniques described in the following section, and exploit symmetry for efficient model checking.

Property specification: We assume that the property of interest can be expressed as the unreachability of some location $e \in \{1, \ldots, k\}$ of $\mathbb{P}$. This characterisation

prog	::=	**shared** shared_name; . . . ; shared_name; **local** local_name; . . . ; local_name; 1: stmt; . . . ; k: stmt;
stmt	::=	**havoc** {havoc_var, . . . , havoc_var} **constrain** $\forall^{\neq}$ {thread_idx, . . . , thread_idx} . expr
	|	**goto** pc, . . . , pc
havoc_var	::=	shared_name | **a**.local_name | **all**(local_name)
expr	::=	0 | 1 | shared_name | shared_name′ | thread_idx.local_name
	|	thread_idx.local_name′ | compound expression
pc	::=	integer in range $1 \ldots k$

shared_name, local_name, thread_idx denote elements of SharedNamesBP, LocalNamesBP and ThreadIndices respectively

Fig. 2 Syntax for Boolean broadcast programs

captures assertion checking by redirecting control flow to a designated undesirable location after an assertion failure has been detected.

Predicates: A *predicate* over $\mathbb{P}$ is a Boolean expression of the form expr that contains at least one variable. As in Definition 1 we categorise a predicate as *shared* / *local* / *mixed* if it refers to variables in $V_S^{\mathbb{P}}$ only / $V_L^{\mathbb{P}}$ only / both $V_S^{\mathbb{P}}$ and $V_L^{\mathbb{P}}$, respectively.

3.2 Boolean Broadcast Programming Language

Our CEGAR technique for symmetric concurrent programs yields a form of concurrent Boolean programs. Although languages for such programs are known [8,4], they do not suit our purpose: as discussed earlier, the facility for a Boolean program thread to read and update variables of other threads is fundamental to our approach. We therefore now present syntax and semantics for this variant of concurrent Boolean programs, which we call *Boolean broadcast programs*.

Syntax. Let SharedNamesBP, LocalNamesBP and ThreadIndices be disjoint sets of names, and suppose that ThreadIndices contains a distinguished name **a**, which we call the *active thread index*. Figure 2 provides the syntax for Boolean broadcast programs, defined over these sets. Like an SCPL program, a Boolean broadcast program is a template to be executed by some number of concurrent threads. Below, we provide operational semantics describing the execution of a Boolean broadcast program by a specified number of threads. While discussing the language syntax, we informally describe the intended semantics in order to motivate the formal semantics.

A Boolean broadcast program declares a set of *shared* Boolean variables with distinct names drawn from SharedNamesBP, followed by a set of *local* Boolean variables with distinct names drawn from LocalNamesBP. When the program is instantiated, there will be *one* copy of each shared variable, shared among all threads, while each thread will own a separate copy of every local variable. All variables are initialised nondeterministically.

As in an SCPL program, behaviour is defined by a sequence of statements numbered 1 through k (for some $k \geq 1$). There are two forms of statements: **goto** and

constrained assignment. The latter causes the values of specified local and shared variables to be havocked and, if the resulting state satisfies a given *constraint*, the executing thread's program counter to be incremented. In a constrained assignment, the variables to be havocked are specified as a set following the **havoc** keyword. Havocking a shared variable is specified simply by providing the variable's name. Havocking the copy of a local variable b owned by the thread that executes the statement, which we call the *active thread*, is specified as **havoc** $\{\mathtt{a}.b\}$. Havocking local variable b in *all* threads is specified as **havoc** $\{\mathbf{all}(b)\}$.

The constraint that must be satisfied following havocking of variables is specified as an expression over shared variables, variables of the active thread, and variables of one or more symbolically represented additional threads. We refer to these additional threads as *passive* threads, since their state is modified and/or inspected even though they are not actively executing a statement. A constraint has the form:

$$\forall^{\neq}\ I\ .\ \phi$$

where I is a set of symbolic thread indices, which must contain the distinguished active thread index $\mathtt{a}$. Such a constraint should be read as follows: "expression ϕ holds under *every* assignment of distinct thread ids to the symbolic thread indices appearing in I such that $\mathtt{a}$ is assigned the id of the active thread". We call the symbol $\forall^{\neq}$ the "for all distinct" quantifier. In ϕ, unprimed and primed variables refer to the values before and after executing the **havoc** part of the statement, respectively.

A constraint $\forall^{\neq}\ I\ .\ \phi$ must be *closed*. For instance, if $t.b$ or $t.b'$ appear in ϕ (where b is a local variable), then I must contain t, so that it is bound under $\forall$.

Semantics. Let $\mathbb{B}$ be a Boolean broadcast program. Let $V_S^{\mathbb{B}}$ and $V_L^{\mathbb{B}}$ be the names of the shared and local variables declared in $\mathbb{B}$, respectively. Operational semantics for Boolean broadcast programs are presented in Figure 3, defined with respect to a fixed number of threads, n.

Definition 2 *A state Σ of a Boolean broadcast program over n threads is a tuple (S, L, PC) where*

- $S : V_S^{\mathbb{B}} \to \{0,1\}$ *maps each shared variable to a Boolean value,*
- L *is an n-tuple representing the local variables of each thread. For $1 \leq t \leq n$, $L(t) : V_L^{\mathbb{B}} \to \{0,1\}$ maps each local variable of thread t to a Boolean value,*
- $PC\colon \{1,\ldots,n\} \to \{1,\ldots,k+1\}$ *maps each thread to a program counter.*

We write $\Sigma.S$, $\Sigma.L$ and $\Sigma.PC$ to refer to the components of a state Σ.

The initial states are those in which every thread's program counter is set to 1 (shared and local variables are nondeterministic). Transitions are defined by two execution rules. For **goto** statements, the executing thread's program counter is set nondeterministically to one of the destinations; all other state components are left unchanged, as are all components of passive threads. A constrained assignment is executed by first setting all state components under the **havoc** clause nondeterministically, and incrementing the active thread's program counter. The $havoc$ function specifies the states that arise from performing this havoc and increment. A transition then exists to any such state where the constraint associated with the constrained assignment holds. A thread is permanently disabled if its program counter becomes $k+1$, since there is no program statement associated with this value.

Initial states. All states Σ satisfying $\Sigma.PC(t) = 1$ $(1 \leq t \leq n)$

Execution rules.

$$\frac{d : \textbf{goto} \ldots, d', \ldots \text{ appears in } B \quad \Sigma.S = \Sigma'.S \quad \Sigma.L = \Sigma'.L \quad \Sigma.PC(t) = d \quad \Sigma'.PC(t) = d' \quad \forall t' \,.\, t' \neq t \Rightarrow \Sigma'.PC(t') = \Sigma.PC(t')}{\Sigma \rightarrow_t \Sigma'}$$

$$\frac{d : \textbf{havoc } H \textbf{ constrain } \forall^{\neq} I \,.\, \phi \text{ appears in } B \quad \Sigma' \in havoc(\Sigma, t, H) \quad \Sigma, \Sigma', t \models \forall^{\neq} I.\phi \quad \Sigma.PC(t) = d}{\Sigma \rightarrow_t \Sigma'}$$

Havocking variables. We define a function $havoc$ which generates the set of states arising from havocking a set of variables in state Σ, and incrementing the active thread t's program counter:

$$\begin{aligned} &havoc(\Sigma, t, \{x_1, \ldots, x_a, \mathtt{a}.y_1, \ldots, \mathtt{a}.y_b, \textbf{all}(z_1), \ldots, \textbf{all}(z_c)\}) = \\ &\{\Sigma' \mid \quad \Sigma'.PC(t) = \Sigma.PC(t) + 1 \\ &\qquad \wedge \forall t' \in \{1, \ldots, n\} \setminus \{t\} \,.\, \Sigma'.PC(t') = \Sigma.PC(t') \\ &\qquad \wedge \forall x \in V_S^{\mathbb{B}} \setminus \{x_1, \ldots, x_a\} \,.\, \Sigma.S(x) = \Sigma'.S(x) \\ &\qquad \wedge \forall y \in V_L^{\mathbb{B}} \setminus \{y_1, \ldots, y_b, z_1, \ldots, z_c\} \,.\, \Sigma.L(t)(y) = \Sigma'.L(t)(y) \\ &\qquad \wedge \forall t' \in \{1, \ldots, n\} \setminus \{t\} \,.\, \forall z \in V_L^{\mathbb{B}} \setminus \{z_1, \ldots, z_c\} \,.\, \Sigma.L(t')(z) = \Sigma'.L(t')(z)\} \end{aligned}$$

Constraint satisfaction. For sets A and B, $Inj(A, B)$ denotes the set of all injective functions from A to B. We define:

$$\Sigma, \Sigma', t \models \forall^{\neq} I \,.\, \phi \Leftrightarrow \forall \delta \in Inj(I, \{1, \ldots, n\}) \,.\, \delta(\mathtt{a}) = t \Rightarrow \Sigma, \Sigma', \delta \models \phi$$

$$\begin{array}{llll} \Sigma, \Sigma', \delta \models 1 & & \Sigma, \Sigma', \delta \not\models 0 & \\ \Sigma, \Sigma', \delta \models b \quad \Leftrightarrow \quad \Sigma.S(b) & & \Sigma, \Sigma', \delta \models b' \quad \Leftrightarrow \quad \Sigma'.S(b) & \\ \Sigma, \Sigma', \delta \models \tau.b \quad \Leftrightarrow \quad \Sigma.L(\delta(\tau))(b) & & \Sigma, \Sigma', \delta \models \tau.b' \quad \Leftrightarrow \quad \Sigma'.L(\delta(\tau))(b) & \end{array}$$

$\Sigma, \Sigma', \delta \models \phi$ is defined inductively for compound expressions

Fig. 3 Operational semantics for Boolean broadcast programs

4 CEGAR for Symmetric Concurrent Programs

Armed with the definitions of SCPL and Boolean broadcast programs, we can now formally present our CEGAR technique, which allows verification of a concurrent SCPL program through finite-state model checking of a series of successively refined Boolean broadcast programs. The key method we use to achieve scalability is performing abstraction and refinement generically at the *template* level. While our method is thread-aware, at no point does it distinguish explicitly between particular threads. This means that it is always possible to exploit symmetry during Boolean program model checking. We demonstrate the benefits of this in Section 5.

4.1 Initial Abstraction

Let $\mathbb{P}$ be an SCPL program, and E a set of predicates over $\mathbb{P}$. The CEGAR process starts with an almost trivial Boolean broadcast program $\mathbb{B}$ that overapproximates the behaviour of $\mathbb{P}$ in a very coarse way. We call this program the *initial abstraction*. Given a injective mapping $bool$ from shared predicates to SharedNamesBP, and from local and mixed predicates to LocalNamesBP, the declarations in $\mathbb{B}$ are as follows:

Statement in $\mathbb{P}$	**Corresponding statement in** $\mathbb{B}$
goto $d_1, \ldots, d_m$	**goto** $d_1, \ldots, d_m$
assume ϕ	**havoc** $\{\ \}$ **constrain** $\forall^{\neq}\ \{\mathtt{a}\}\ .\ 1$
$v = \phi\ (v \in V_L^{\mathbb{P}})$	**havoc** $\{\mathtt{a}.bool(\psi) \mid \psi \in E \wedge v$ appears in $\psi\}$ **constrain** $\forall^{\neq}\ \{\mathtt{a}\}\ .\ 1$
$v = \phi\ (v \in V_S^{\mathbb{P}})$	**havoc** $\{bool(\psi) \mid \psi \in E \wedge v$ appears in $\psi \wedge \psi$ is shared$\}\cup$ $\{$**all**$(bool(\psi)) \mid \psi \in E \wedge v$ appears in $\psi \wedge \psi$ is mixed$\}$ **constrain** $\forall^{\neq}\ \{\mathtt{a}\}\ .\ 1$

Fig. 4 Initial abstraction of SCPL statements into Boolean broadcast program statements

shared $bool(\phi_1); \ldots; bool(\phi_a);$
local $bool(\psi_1); \ldots; bool(\psi_b);$

where $\phi_1, \ldots, \phi_a$ are the shared predicates in E and $\psi_1, \ldots, \psi_b$ are the local and mixed predicates. Thus, as discussed in Section 2, shared predicates are tracked by shared Boolean variables, while local and mixed predicates are tracked by local Boolean variables. $\mathbb{B}$ contains the same number of statements as $\mathbb{P}$: each statement in $\mathbb{B}$ is derived from the corresponding statement in $\mathbb{P}$ according to the rules in Figure 4.

By construction, the initial abstraction yields a Boolean program $\mathbb{B}$ such that, for any $n \geq 1$, the n-thread instantiation of $\mathbb{B}$ is an existential abstraction of the n-thread instantiation of $\mathbb{P}$: whenever a statement in $\mathbb{P}$ might affect the truth of a predicate in E, the associated statement in $\mathbb{B}$ sets the Boolean variable associated with the predicate to a nondeterministic value. As argued above, mixed predicates are handled soundly by nondeterminising the relevant Boolean variables across all threads. A statement **assume** ϕ is translated into a **havoc** statement that havocs no variables and constrains the result to $true$, rendering it equivalent to **assume** $true$.

Improved initial abstractions. Our tool SYMMPA uses a less coarse abstraction by default. For an assignment $v = e$ and predicate ϕ, if the weakest precondition for ϕ to hold after the assignment is a constant or equivalent to another predicate, the new value for $bool(\phi)$ is captured precisely in the corresponding abstract assignment. SYMMPA also supports Cartesian abstraction with maximum cube length approximation [2]. When a standard assignment is generated, cube enumeration is performed over predicates of the active thread only. When a broadcast assignment is generated, cube enumeration is done with respect to predicates of both the active thread and a passive thread. Thus, when Cartesian abstraction is used, relatively precise abstract broadcast assignments can be derived. For example, abstract statement (2) on page 6 is generated directly. In Section 5 we experimentally evaluate the impact of using Cartesian abstraction vs. the more straightforward weakest-precondition abstraction.

4.2 Model Checking Boolean Broadcast Programs

During the CEGAR process, we must repeatedly model-check the n-thread Boolean program $\mathbb{B}^n$. As $\mathbb{B}^n$ is an existential abstraction of $\mathbb{P}^n$, correctness of $\mathbb{B}^n$ (i.e., unreachability of the error location e) implies correctness of $\mathbb{P}^n$. Any counterexample reported by the model checker is a trace of the form:

$$\Sigma_0 \rightarrow_{t_1} \Sigma_1 \rightarrow_{t_2} \cdots \rightarrow_{t_m} \Sigma_m$$

where Σ_0 is an initial state of $\mathbb{B}^n$; $t_i \in \{1, \ldots, n\}$ $(1 \leq i \leq m)$ is the identity of the thread that executed a statement to cause the i-th transition; for $i < m$ we have $\Sigma_i.PC(t) \neq e$ $(1 \leq t \leq n)$ and $\Sigma_m.PC(t_m) = e$.

The extended syntax and semantics for broadcasts mean that we cannot simply use existing concurrent Boolean program model checkers such as BOOM [4] for the model checking phase of the CEGAR loop. We have implemented an extension, B-BOOM, of BOOM, which adjusts the counter abstraction-based symmetry reduction capabilities of BOOM to support broadcast operations. Symbolic image computation for broadcast assignments is more expensive than 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 cost of image computation for broadcasts motivates our investment into determining tight conditions under which broadcasts are required. Checking **constrain** clauses of the form **constrain** $\forall^{\neq}\ I\ .\ \phi$ becomes more expensive as the set I of thread indices grows. Thus, during abstraction and refinement, it is desirable to avoid large index sets.

4.3 Simulation

Given a counterexample for $\mathbb{B}^n$, we reconstruct a multi-threaded trace through $\mathbb{P}^n$ that follows exactly the same control path. Each statement of the trace is labelled with the identity of the respective active thread. We are able to obtain a unique path as our abstraction yields one abstracted statement for each concrete statement. This trace through $\mathbb{P}^n$ can be used to construct an equation representing the constraints that must be satisfied for execution to follow this route. Such constraints arise from **assume** statements and assignments in $\mathbb{P}^n$ that are traversed by the trace.

If the resulting equation is satisfiable, a feasible error trace through $\mathbb{P}^n$ has been identified, and $\mathbb{P}$ can be reported as incorrect (along with the thread count n). Otherwise the counterexample in $\mathbb{B}^n$ is *spurious*: no corresponding counterexample exists in $\mathbb{P}^n$. To eliminate this counterexample (and possibly others), the abstract program $\mathbb{B}$ must be refined, either by improving the precision of individual statements in $\mathbb{B}$ (transition refinement), or by adding further predicates (predicate discovery).

4.4 Transition Refinement

Suppose that model checking and simulation have revealed a spurious counterexample trace in $\mathbb{B}^n$. It is possible that some transition $\Sigma \rightarrow_t \Sigma'$ occurring in the trace is spurious: no concrete state corresponding to Σ can lead via execution of the statement at $\Sigma.PC(t)$ by thread t to a concrete state corresponding to Σ'. In this case we can refine the **constrain** clause of the statement in $\mathbb{B}$ associated with the transition, such that this particular transition is eliminated *without* adding new predicates.

This type of refinement, which goes back to early work by Das and Dill [10] and has become known as *transition refinement*, is by now standard in CEGAR techniques for sequential software; it is, for example, implemented in SLAM [1]. We extend SLAM's transition refinement technique to our symmetric concurrent setting. In order to do so in a scalable manner we require the ability to add constraints over the variables of a *small* subset of threads: those threads actually responsible for the spuriousness of a transition. Furthermore, to retain symmetry, we cannot add constraints over specific threads; we must apply generalisation.

We first describe how to detect whether a transition is spurious with respect to a concrete set J of thread ids. For a Boolean variable b occurring in $\mathbb{B}$, let $pred(b)$ be the predicate in E that b tracks; $pred$ is the inverse of the injective mapping $bool$. For a predicate ϕ and $t \in \{1, \ldots, n\}$, we write ϕ_t to denote the expression identical to ϕ except that each local variable $v \in V_L^{\mathbb{P}}$ appearing in ϕ is replaced by v_t in ϕ_t.

Definition 3 ***Concretisation of a state.*** *Let $J \subseteq \{1, \ldots, n\}$ be a set of thread ids, and Σ a state of $\mathbb{B}$. The* concretisation *of Σ with respect to J is denoted $\gamma_J(\Sigma)$, and defined as follows:*

$$\gamma_J(\Sigma) = \big(\bigwedge_{\{b \in V_S^{\mathbb{B}} \mid \Sigma.S(b)\}} pred(b)\big) \wedge \big(\bigwedge_{\{b \in V_S^{\mathbb{B}} \mid \neg\Sigma.S(b)\}} \neg pred(b)\big) \wedge$$
$$\bigwedge_{t \in J} \big(\big(\bigwedge_{\{b \in V_L^{\mathbb{B}} \mid \Sigma.L(t)(b)\}} pred(b)_t\big) \wedge \big(\bigwedge_{\{b \in V_L^{\mathbb{B}} \mid \neg\Sigma.L(t)(b)\}} \neg pred(b)_t\big)\big)$$

Definition 4 ***Weakest precondition.*** *Let ϕ be an expression over integer literals, variables in $V_S^{\mathbb{P}}$, and variables of the form v_i where $v \in V_L^{\mathbb{P}}$ and $i \in \{1, \ldots, n\}$. Let $t \in \{1, \ldots, n\}$, and suppose s is a statement in $\mathbb{P}$. We define the* weakest precondition *of ϕ with respect to t executing s, denoted $WP(s, t, \phi)$, as follows:*

- $WP(\textbf{\textit{goto}} \ldots, t, \phi) = \phi$
- $WP(v = \psi, t, \phi) = \phi[v/\psi_t]$ *if* $v \in V_S^{\mathbb{P}}$
- $WP(v = \psi, t, \phi) = \phi[v_t/\psi_t]$ *if* $v \in V_L^{\mathbb{P}}$
- $WP(\textbf{\textit{assume}}\ \psi, t, \phi) = (\psi_t \Rightarrow \phi)$

Definition 5 *A transition $\Sigma \rightarrow_t \Sigma'$ is* spurious with respect to a set $J \subseteq \{1, \ldots, n\}$ of thread ids*, written $spurious(t, J, \Sigma, \Sigma')$, if*

$$\gamma_J(\Sigma) \Rightarrow \neg\, WP(\Sigma.PC(t), t, \gamma_J(\Sigma'))\,.$$

(Recall that $\Sigma.PC(t)$ denotes the statement to be executed by t in Σ.)

Suppose a transition $\Sigma \rightarrow_t \Sigma$ is spurious with respect to a set of thread ids J, and that $t \in J$.[1] Suppose that the statement at program point $\Sigma.PC(t)$ in $\mathbb{B}$ is

$$\textbf{havoc}\ H\ \textbf{constrain}\ \forall^{\neq}\ I\,.\,\phi$$

We wish to add a conjunct to ϕ that will rule out the spurious transition. In order to do so in a manner that preserves symmetry, we require the notion of *generalising* a concrete state to refer to a set of symbolic thread indices.

[1] It is easy to generalise what follows to handle the case where $t \notin J$. Because in practice it usually makes sense to refine transitions with respect to a set of threads that includes the active thread (indeed often the set J will simply be $\{t\}$) we do not present this generalisation, which would add tests on $t \notin J$ in several occurrences of J.

Definition 6 ***Generalisation of a state.*** *Let J and Σ be as in Definition 3, and let $\delta : J \to$ ThreadIndices be an injective function. The* generalisation *of Σ with respect to J and δ, denoted $\beta^{\delta}_{J}(\Sigma)$, is a formula representing the constraints satisfied in Σ by the threads in J, but with thread identities abstracted to symbolic names drawn from* ThreadIndices*. Formally:*

$$\beta^{\delta}_{J}(\Sigma) = \big(\bigwedge_{\{b \in V^{\mathbb{B}}_{S} | \Sigma.S(b)\}} b\big) \wedge \big(\bigwedge_{\{b \in V^{\mathbb{B}}_{S} | \neg\Sigma.S(b)\}} \neg b\big) \wedge$$
$$\bigwedge_{t \in J} \big(\big(\bigwedge_{\{b \in V^{\mathbb{B}}_{L} | \Sigma.L(t)(b)\}} \delta(t).b\big) \wedge \big(\bigwedge_{\{b \in V^{\mathbb{B}}_{L} | \neg\Sigma.L(t)(b)\}} \neg\delta(t).b\big)\big)$$

Adding the required conjunct to ϕ can be achieved as follows. First, if $|J| \geq |I|$, i.e., the spurious transition involves more concrete threads than the number of threads currently represented symbolically in the constrained assignment, we add $|J| - |I|$ distinct fresh elements to I taken from ThreadIndices. Second, we choose an arbitrary injection $\delta : J \to I$ satisfying $\delta(t) = \mathbf{a}$. (Note that $\mathbf{a} \in I$, according to the definition of a Boolean broadcast program.) Finally, we add the following conjunct to ϕ:

$$\neg(\beta^{\delta}_{J}(\Sigma) \wedge primed(\beta^{\delta}_{J}(\Sigma')))$$

where $primed(\phi)$ denotes ϕ with every Boolean variable name b replaced by b'.

Our implementation, SYMMPA, follows the strategy of *always* performing transition refinement if it is possible to do so, only adding additional predicates (see Section 4.5) as a last resort. This reflects a design decision in SATABS, on which SYMMPA is based, which has been shown to work well for verification of sequential software. As discussed above, it is expensive to model check Boolean broadcast programs that involve constraints over many threads. The strategy for applying transition refinement is thus to attempt to refine a spurious transition with respect to a single thread: the active thread. If this fails, SYMMPA attempts to refine with respect to two threads: the active thread and one passive thread. If this also fails, three threads are considered, and so on. In other words, the set J of concrete thread indices used for transition refinement is first taken to have size one, which is subsequently increased by one in each iteration. In practice, as we show in Section 5, for a large set of benchmarks we find that it is never necessary to consider $|J| > 2$.

4.5 Predicate discovery

Transition refinement may fail to add precision to $\mathbb{B}$ if all transitions in a given trace belong to the most precise Boolean abstraction with respect to the current set of predicates. In this case, the abstraction can only be refined using additional predicates.

New predicates can be extracted from counterexamples using thread-aware variants of standard techniques based on weakest preconditions, strongest postconditions, or interpolation. These standard methods require a small modification in our context: they 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 CEGAR technique operates at the template program level, we cannot add this predicate directly. Instead, we generalise such predicates by removing thread indices. Hence, in the above ex-

ample, we add the predicate $l < s$, at the template level. Since it is mixed, adding it is tantamount to adding such a predicate for *all* threads.

5 Experimental Evaluation

We evaluate SYMMPA in two steps. First, we analyse in detail the relative merits of various options offered by the tool. Second, we compare it to a recent verifier, THREADER [17], which exploits the compositional nature of many programs [16] but ignores any thread symmetry. This allows us to measure the impact of our reduction mechanism. In both steps we consider thread counts ranging from 2 to 4; the impact of larger thread counts was considered in prior work [11].

Benchmarks. The evaluation is based on a set of 24 concurrent C programs. In these benchmarks, threads synchronise via locks, or in a lock-free manner via atomic *compare-and-swap* instructions. Fourteen of the benchmarks are *parametric*, i.e., they contain procedures to be executed by any number of replicated threads. In contrast to the (idealised) syntax presented in Section 3, where the number of running threads is determined on program entry, most of our benchmarks consist of an initial thread, while further threads are created dynamically. In such cases, we use a parameter n to bound the number of threads that can be created: once this bound is reached, thread creation statements have no effect. The remaining $24 - 14 = 10$ benchmarks are non-parametric, with a built-in thread count.

We compare SYMMPA and THREADER on benchmarks that were previously used in the evaluation reported in our prior work [11], or in [17]:

Parametric benchmarks

1–4 a counter, concurrently incremented, or incremented and decremented, by multiple threads [27];
5,6 algorithms to establish mutual exclusion for an arbitrary number of threads [24];
7,8 concurrent pseudo-random number generator [27];
9–12 implementations of parallel reduction operation to find the maximum element in an array (as "simple" and "optimised" version; the latter reduces communication by locally computing a partial maximum value);
13 a program used in [12] to illustrate the need for mixed predicates;
14 a simple program used in [14] to illustrate thread-modular model checking;

Non-parametric benchmarks

15–22 algorithms to establish mutual exclusion for fixed thread counts; QRCU is a variant of the Linux read-copy-update algorithm [23];
23 a vulnerability fix from the Mozilla repository described in [22];
24 a Linux character driver related to global memory area access [9].

For each benchmark, we consider verification of a safety property, specified via an assertion. All experiments are performed on a 3GHz Intel Xeon machine with 8 GB RAM, running 64-bit Linux, with a timeout of 30 minutes.

Impact of Cartesian Abstraction. Table 1 presents the results for our approach without (SYMMPA-c0) and with Cartesian abstraction, using a maximum cube length of 1

Benchmark id/Name	n	Predicates			Maximum cube length approximation								
					SYMMPA-c0			SYMMPA-c1			SYMMPA-c2		
		L	*M*	*S*	*It.*	*T.r.*	*Time*	*It.*	*T.r.*	*Time*	*It.*	*T.r.*	*Time*
1/INC-L	4	2	2	5	7	12	**1**	6	12	2	6	12	2
2/INC-C	4	4	2	0	7	21	3	6	9	**2**	7	15	5
3/INC-DEC-L	4	4	3	9	16	43	25	10	17	**10**	10	17	17
4/INC-DEC-C	4	8	4	2	24	108	**908**	–	–	T.O.	–	–	T.O.
5/TAS-L	4	1	1	4	3	1	**1**	3	1	**1**	2	0	**1**
6/TICKET-L	3	2	5	9	9	44	**15**	8	7	18	6	5	75
7/PRNG-L	4	8	0	3	4	5	2	3	0	<**1**	3	0	5
8/PRNG-C	3	15	1	2	–	–	T.O.	15	60	**834**	–	–	T.O.
9/MAXSIMP-L	4	6	7	2	3	2	**8**	3	1	39	2	0	276
10/MAXSIMP-C	4	2	4	0	13	45	**1515**	–	–	T.O.	–	–	T.O.
11/MAXOPT-L	4	2	1	2	3	2	**5**	3	1	**5**	2	0	**5**
12/MAXOPT-C	4	3	4	0	–	–	T.O.	11	45	**1460**	–	–	T.O.
13/UNVEREX	4	0	3	5	8	30	3	4	2	**2**	3	0	**2**
14/SPIN	4	0	0	3	2	0	<**1**	2	0	<**1**	2	0	<**1**
15/DEKKER	2	0	0	5	2	0	<**1**	2	0	<**1**	2	0	<**1**
16/PETERSON	2	0	0	6	2	0	<**1**	2	0	<**1**	2	0	<**1**
17/SZYMANSKI	2	0	0	7	2	0	<**1**	2	0	<**1**	2	0	<**1**
18/READERS-WR.	2	0	0	7	3	0	<**1**	4	2	<**1**	4	2	<**1**
19/TIME-V.-MU	2	0	0	7	2	0	<**1**	2	0	<**1**	2	0	<**1**
20/LAMPORT	2	0	0	11	5	3	<**1**	5	3	1	5	3	<**1**
21/QRCU-2	2	10	2	13	11	40	**6**	11	40	12	11	40	80
22/QRCU-3	3	10	2	13	11	40	**6**	11	40	11	11	40	51
23/MOZILLA-FIX	2	0	0	5	3	0	<**1**	3	0	<**1**	3	0	<**1**
24/SCULL	3	9	0	5	6	6	**1**	6	6	2	6	6	2

Table 1 Results of SYMMPA without and with Cartesian abstraction; (joint) fastest run times per line in bold. The double line separates parametric and non-parametric instances

and 2 (SYMMPA-{c1,c2}). For benchmarks 1–12, the suffix indicates whether locks (L) or compare-and-swap (C) instructions were used for synchronisation. For each parametric instance (above the double line) we present statistics for the largest thread count that could be verified within the time limit for at least one of {c0,c1,c2}. The table shows this thread count (n), the number of local, mixed, and shared predicates (*L*, *M*, *S*) over the program that were required for verification to succeed, the number of CEGAR iterations (*It.*) and transition refinement operations performed (*T.r.*), and the total run time (*Time*). We find that the number of local, mixed and shared predicates required for verification varies slightly with n, but mostly remains unchanged for $n \geq 3$. As a result, the total run time increases for larger thread counts are mostly due to the increased complexity of model checking the Boolean abstractions.

With the exceptions of 6/TICKET-L and 8/PRNG-C, most parametric instances could be easily verified for four threads, and all non-parametric ones for the given fixed thread count. Generating more precise Cartesian-style predicate images whenever possible is more expensive but results in fewer iterations and transition refinements[2]. In our examples, the latter reduction causes the percentage of total run time spent in the abstraction phase to increase from 3 to 11 and 32% respectively, while reducing the model checking time from 80 to 72 and 55%, respectively. Each of SYMMPA-c0 and SYMMPA-c1 time out for two benchmarks. We find that the more precise Cartesian abstraction used in SYMMPA-c2 does not pay off: timeout occurs for four benchmarks that require many predicates to verify.

[2] Exceptions are 2/INC-C and 17/READERS-WR., which is due to different counterexamples reported by the model checker.

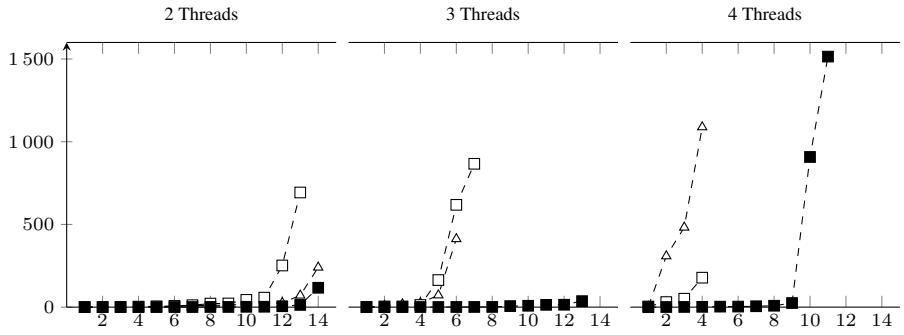

Fig. 5 Runtime comparison for various thread counts of THREADER-OG (△), THREADER-RG (□), and SYMMPA-c0 (■). The vertical axis shows the runtimes in seconds. The horizontal axis arranges the benchmark instances with increasing runtime

Mixed predicates were required for verification to succeed in 13 of the 23 cases. These require broadcast transitions for sound abstraction, which are more expensive to implement than traditional single-thread transitions. The table reflects that with higher run-times for benchmarks with large numbers of mixed predicates (M), such as the four examples where SYMMPA-c0 and SYMMPA-c1 time out for 4 threads, compared to similar examples with smaller numbers of mixed predicates.

In all cases, the precision of the abstraction did not benefit from considering more than one passive thread: it was always possible to eliminate a spurious transition by adding a constraint over the active thread and perhaps a single passive thread. In the notation of Section 4.4, we were always able to demonstrate spurious transitions using a set J of concrete threads with $|J| \leq 2$, and correspondingly for symbolic index sets I appearing in **constrain** $\forall^{\neq}\ I$ clauses, we always found $|I| \leq 2$ sufficed. We also considered a version of configuration SYMMPA-c0 with abstraction and refinement restricted to consider only variables of the *active* thread. This restriction removes the ability to precisely refine over mixed predicates, and significantly decreases the overall performance: for 9 benchmarks the CEGAR procedure times out already with just two threads.

Scalability. The effect of turning symmetry reduction features of SYMMPA on and off was demonstrated in prior work [11]; we do not repeat this evaluation here. We evaluate the scalability of SYMMPA by comparing it against two approaches implemented in the THREADER tool [17]:

THREADER-OG: Owicki-Gries proof system [26];
THREADER-RG: A compositional version of Owicki-Gries [20].

THREADER's front-end capabilities are similar to that of SYMMPA, facilitating comparison of the tools[3]. On the other hand, we emphasise that THREADER is not optimised for the analysis of *replicated* multi-threaded programs. Support in

[3] A notable difference is that THREADER relies on natural integers for representing numeric C types, whereas SYMMPA uses machine integers, hence is "bit-precise". For comparability we skipped three benchmarks from [17] where this difference affects the verification outcome.

THREADER for replicated threads is limited to the predicate discovery phase: predicates over the variables of one thread are replicated to threads with identical functionality, so as to avoid rediscovery. The authors point out that supporting "symmetry reduction ... would be beneficial for [their] approach" [17].

Figure 5 presents a "cactus plot" showing verification times (in seconds) for the 14 parametric benchmarks with 2, 3 and 4 concurrent threads. For a given tool and thread count, a point (x, y) on the corresponding graph indicates that if the tool was launched on all 14 benchmarks in parallel, verification of x of the benchmarks would have completed within y seconds. As previously demonstrated, SYMMPA-c0 and SYMMPA-c1 perform similarly, hence we only show data for the former. In the vast majority of cases, our technique significantly outperforms both THREADER approaches. This can be attributed to the fact that, with THREADER, the number of predicates grows according to the number of threads considered, while with SYMMPA, this is thread-count independent. The Owicki-Gries method only outperforms our approach significantly for 8/PRNG-C and $n = 2$, and the Rely-Guarantee method in addition for 10/MAXSIMP-C and $n = 4$. For these, THREADER is able to discover very compact thread-modular proofs. For the non-parametric benchmarks our method is also superior: The Rely-Guarantee method times out for one instance and requires 160 seconds for the remaining ones. The Owicki-Gries method succeeds on all instances but requires 1800 seconds in total. In contrast, our configuration SYMMPA-c0 takes only 12 seconds for all instances (again no time out).

6 Related Work

Predicate abstraction for sequential programs [2] was implemented as part of the SLAM project, building on foundational work by Graf and Saïdi [15]. 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. 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 [4] has made progress in this direction, motivating us to revisit concurrent predicate abstraction.

We are not aware of other work that presents precise solutions to the problem of mixed predicates. Some approaches disallow such predicates, e.g. [28], whose authors do not discuss, however, the reasons for (or consequences of) doing so. Another approach nondeterministically resets global variables that may be affected by an operation [6], taking away the mixed flavour from certain predicates. The authors of [18], 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 [18]. In [5], 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 discussed in Section 2, 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 [5].

Other model checkers with some support for concurrent software include BLAST, which does not support general assertion checking when concurrency is enabled [19], and MAGIC [5], which does not support shared variable communication. These limitations prevent a meaningful experimental comparison of these tools with our work.

Symmetry reduction as a technique to handle state explosion in model checking has become well established (see [25,30] for a survey). The scalability of our approach hinges on a recent method that allows symmetry reduction to be combined with a BDD-based representation [4].

7 Open Problems

We have presented the ideas, a formalisation, and an implementation of a CEGAR strategy for symmetric concurrent programs. Our strategy is *symmetry-aware*, passing on the symmetry from the concurrent source program to the abstract Boolean program, enabling the application of powerful reduction methods to facilitate model checking. Our experiments have clearly demonstrated the impact that our strategy thus affords, resulting in significant efficiency gains.

In conclusion, we mention two open problems for future consideration. We have here assumed a very strict (and unrealistic) memory model that guarantees atomicity at the statement level. One can work soundly with the former assumption by preprocessing 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 is one aspect of future work.

Our technique assumes that the input program is known to replicate threads in a perfectly symmetric manner. To fully automate the application of our method to source code, we additionally need a technique for detecting the presence of thread symmetry. Existing approaches try to identify symmetries in PROMELA models [13] or in fact multi-threaded C code [31]. In practice, there are two options: one can focus on full symmetry, checking that the code of the replicated function does not use thread ids in a symmetry-breaking way. A more general approach is to permit symmetry violations and determine the largest symmetry group that still accommodates such violations. Classical examples include the rotation group for threads with a round-table communication pattern. As a consequence, the algorithmic methods used in BOOM, which are based on counter abstraction, no longer apply.

References

1. Thomas Ball, Byron Cook, Satyaki Das, and Sriram K. Rajamani. Refining approximations in software predicate abstraction. In *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 388–403. Springer, 2004.
2. Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.
3. Thomas Ball and Sriram Rajamani. The SLAM project: debugging system software via static analysis. In *Principles of Programming Languages (POPL)*, pages 1–3, 2002.

4. Gerard Basler, Michele Mazzucchi, Thomas Wahl, and Daniel Kroening. Context-aware counter abstraction. *Formal Methods in System Design (FMSD)*, 36(3):223–245, 2010.
5. S. Chaki, E.M. 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.
6. Alessandro Cimatti, Andrea Micheli, Iman Narasamdya, and Marco Roveri. Verifying SystemC: a software model checking approach. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2010.
7. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 2003.
8. Byron Cook, Daniel Kroening, and Natasha Sharygina. Verification of Boolean programs with unbounded thread creation. *Theoretical Computer Science (TCS)*, 2007.
9. J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.
10. Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *Logic in Computer Science (LICS)*, 2001.
11. Alastair F. Donaldson, Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 356–371. Springer, 2011.
12. Alastair F. Donaldson, Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Symmetry-aware predicate abstraction for shared-variable concurrent programs (extended technical report). *CoRR*, abs/1102.2330, 2011.
13. Alastair F. Donaldson and Alice Miller. Automatic symmetry detection for Promela. *J. Autom. Reasoning*, 41(3-4):251–293, 2008.
14. Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In *Model Checking of Software (SPIN)*, 2003.
15. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification (CAV)*, LNCS, pages 72–83. Springer, 1997.
16. A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, pages 331–344. ACM, 2011.
17. Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Threader: A constraint-based verifier for multi-threaded programs. In *Computer-Aided Verification (CAV)*, 2011.
18. T. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *CAV*, LNCS, pages 262–274. Springer, 2003.
19. Thomas Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *Programming Language Design and Implementation (PLDI)*, pages 1–13, 2004.
20. Cliff B. Jones. Tentative steps toward a development method for interfering programs. *Transactions on Programming Languages and Systems (TOPLAS)*, 1983.
21. Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *Computer-Aided Verification (CAV)*, 2010.
22. Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.
23. P. McKenney. *Using Promela and Spin to verify parallel algorithms*. LWN.net, weekly edition, 2007.
24. John Mellor-Crummey and Michael Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.
25. Alice Miller, Alastair Donaldson, and Muffy Calder. Symmetry in temporal logic model checking. *Computing Surveys*, 2006.
26. Susan Speer Owicki. *Axiomatic proof techniques for parallel programs*. PhD thesis, Cornell University, 1975.
27. Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.
28. N. Timm and H. Wehrheim. On symmetries and spotlights – verifying parameterised systems. In *ICFEM*, LNCS, pages 534–548. Springer, 2010.
29. Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. Model-checking parameterized concurrent programs using linear interfaces. In *Computer-Aided Verification (CAV)*, 2010.
30. Thomas Wahl and Alastair Donaldson. Replication and abstraction: Symmetry in automated formal verification. *Symmetry*, 2010.
31. Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Chao Wang. Automatic discovery of transition symmetry in multithreaded programs using dynamic analysis. In *SPIN*, 2009.