Symbolic Counter Abstraction for Concurrent Software*

Gérard Basler[1], Michele Mazzucchi[1], Thomas Wahl[1,2], Daniel Kroening[1,2]

[1] Computer Systems Institute, ETH Zurich, Switzerland
[2] Computing Laboratory, Oxford University, United Kingdom

Abstract. The trend towards multi-core computing has made concurrent software an important target of computer-aided verification. Unfortunately, Model Checkers for such software suffer tremendously from combinatorial state space explosion. We show how to apply *counter abstraction* to real-world concurrent programs to factor out redundancy due to thread replication. The traditional global state representation as a vector of local states is replaced by a vector of thread counters, one per local state. In practice, straightforward implementations of this idea are unfavorably sensitive to the number of local states. We present a novel symbolic exploration algorithm that avoids this problem by carefully scheduling which counters to track at any moment during the search. Our experiments are carried out on Boolean programs, an abstraction promoted by the Slam project. To our knowledge, this marks the first application of counter abstraction to programs with non-trivial local state spaces, and results in the first scalable Model Checker for concurrent Boolean programs.

1 Introduction

Software Model Checking has been a vibrant branch of research in formal methods for many years. *Predicate abstraction* [1, 2] is one of the most prominent approaches in this area, promoted by the success of the Slam project at Microsoft Research. Instead of tracking the actual values of program variables, the abstraction monitors carefully selected predicates over these variables. Predicate abstraction results in a *Boolean program* [3], i.e., a program using exclusively Boolean variables. Embedded in an automated abstraction-refinement framework [4], verifiers for Boolean programs have been used successfully to increase the reliability of system-level software such as Windows device drivers [5].

Recently, there have been attempts to extend these techniques to the verification of *concurrent* software [6]. The challenge is the classical state space explosion problem: the number of reachable program states grows exponentially with the number of concurrent threads, which renders naive exploration impractical. The authors of [6] conclude that none of the currently available tools is able to handle device drivers of realistic size in the presence of many threads.

* This research is supported by the EU FP7 STREP MOGENTES (project ID ICT-216679), and by the EPSRC project EP/G026254/1.

One observation that comes to the rescue is that concurrent components of multi-threaded software are often simply *replications* of a template program describing the behavior of a component. The ensuant regularity in the induced system model can be exploited to reduce the verification complexity. One technique towards this goal is *counter abstraction.* The idea is to record the global state of a system as a vector of counters, one per local state, tracking how many of the n components currently reside in the local state. This idea was proposed as a way of achieving *symmetry reduction* for fixed-size systems [7], turning an n-process model of size exponential in n into one of size polynomial in n.

Counter abstraction as proposed in [7] requires conversion of the template program $\mathcal{P}$ into a local-state transition diagram, by identifying a set of local states a component can be in, and translating the program statements into local state changes. Such a conversion is straightforward if there are only few component configurations, such as with certain high-level communication protocols [8]. For concurrent software, however, $\mathcal{P}$ is given in a `C`-like language, with assignments to variables, branches, loops, etc. A local state is then defined as a valuation of all local variables of a thread. As a result, there are exponentially many local states, measured in the number of local variables. Introducing a counter variable for each local state is impractical but for tiny programs.

In this paper, we present a strategy to solve these complexity problems. Our solution is two-fold. First, we interleave the translation of individual program statements with the Model Checking phase. This has the advantage that the context in which the statement is executed is known; the counters for the source and target local states, which need to be updated, depend on this context. If the translation is performed up-front, one has to embed each statement into all local state contexts where the statement is enabled, which is infeasible for realistic programs. Second, in a global state we keep counters only for those local states that at least one thread resides in. This idea exploits a simple counting argument: given n threads with l conceivable local states each, at most n of the corresponding local state counters are non-zero at any time during execution. Since n is typically much smaller than l, omitting the zero-valued counters results in huge savings: the sensitivity of counter abstraction to the *local state space explosion problem* mentioned in the previous paragraph is reduced from exponential in l to exponential in $\min\{n, l\}$.

Contributions. We present an efficient algorithm for BDD-based symbolic state space exploration of Boolean programs run by a fixed number of parallel threads. The algorithm's primary accomplishment is to curb the local state space explosion problem, the classical bottleneck in implementations of counter abstraction. We demonstrate the effectiveness of our approach on a substantial set of Boolean program benchmarks, generated by two very different CEGAR-based toolkits, SatAbs [9] and Slam. Since symmetry reduction, of which finitary counter abstraction is an instance, has so far been implemented more successfully in explicit-state model checkers, we include an experimental comparison of an explicit-state version of our method against classical explicit-state symmetry reduction, using the well-known Murφ model checker [10].

We believe our algorithm marks a major step towards the solution of an exigent problem in verification today, namely that of Model Checking concurrent software. While the concepts underlying our solution are relatively straightforward, exploiting them in symbolic model checking is not. The succinctness of state space representations that BDDs often permit is paid for by rather rigid data manipulation mechanisms. To the best of our knowledge, our implementation is the first *scalable* approach to counter abstraction in symbolic verification of concurrent software with replicated threads.

Counter abstraction has also been applied in parameterized system verification, using truncated counters, necessarily resulting in an incomplete method (see section on related work). We emphasize that, in this paper, we use the term *counter abstraction* in the sense of **exact** counters. The method we propose can be seen as an "exact abstraction", a notion that is common in symmetry reduction and other bisimulation-preserving reduction methods.

2 Related Work

While the principal idea of using process counters already appeared in early work by Lubachevsky [11], *generic representatives* were suggested by Emerson and Trefler [7] as a means of addressing the complexity of symmetry-reducing symbolically represented systems. The term *counter abstraction* was actually coined in the context of parameterized verification [12]. In contrast to the present work, the counters are cut off at some value c, indicating that *at least* c components currently reside in the corresponding local state.

Local state-space explosion was identified in [13] as the major obstacle to using generic representatives with non-trivial symmetric programs. The paper ameliorates this problem using a static live-variable analysis, and using an approximate but inexpensive local state reachability test. Being heuristic in nature, this work cannot guarantee a reduced complexity of the abstract program.

We are aware of a few significant works that resulted in tools using counter abstraction in symbolic Model Checking: [14], in the context of *virtual symmetry* [15], and [8], for probabilistic models. While valuable in their respective domains, both approaches suffer from a limitation that makes them unsuitable for general software: they are based on a system model (such as the *GSST* of [14]) that describes the process behavior by local state changes and thus require an up-front translation from whatever input language is used. The examples in [14, 8] include communication and mutual-exclusion protocols with at most a few dozen local states. The Beacon Model Checker [16] has been applied to a multi-threaded memory management system with 256 local states. In our benchmarks, threads have millions of local states (see section 5).

In [17], 0-1-∞ counter abstraction is applied to predicate-abstracted concurrent `C` programs for race detection. The counters monitor the states of *context threads*. To avoid local state space explosion, each context thread is simplified to an *abstract control flow automaton* (ACFA). According to [17], the ACFA has at most a few dozen vertices and can thus be explicitly constructed. In contrast, our goal is a general solution for arbitrary predicate abstractions, where we cannot rely on a small number of predicates and, thus, local states. Consequently, our work does not require first building a local state transition diagram.

Compared to canonization-based symmetry reduction approaches such as in Murφ [10] and Zing [18] (explicit-state) or Sviss [19] and RuleBase [20] (symbolic), the model checking overhead that counter abstraction incurs reduces to translating the program statements into local state counter updates. Sorting local state sequences, or other representative mapping techniques, are implicit in the translation.

Finally, the general problem of symbolically verifying multi-threaded programs has been tackled in many recent publications [21, 22, and others]. None of these address the symmetry that concurrent Boolean programs exhibit, although some investigate partial-order based methods [23].

3 Preliminaries

3.1 Boolean Programs

Boolean programs result from applying *predicate abstraction* to general software. All variables are of type Boolean, and track values of predicates over (possibly unbounded) variables of the original program $\mathcal{P}$. To enable sound verification of reachability properties, Boolean programs are constructed to over-approximate the behavior of $\mathcal{P}$. This may permit *spurious* paths, which need to be detected and eliminated, by refining the abstraction using additional predicates.

Several tools exist that translate `C` code into a Boolean program. Many of the Boolean programs used in our experiments were generated by a front-end of the SatAbs model checker, from Linux kernel `C` code. The code is assumed to be free of recursion and dynamic thread creation.[3] In a preprocessing step, loops and `if` statements are replaced by nondeterministic `goto`s and `assume` statements; function calls are inlined. Figure 1 shows a translation of a fragment of the Apache webserver suite into a Boolean program.

Table 1: Semantics of fundamental Boolean program statements

Syntax	**Semantics**
$v_1, \ldots, v_n := expr_1, \ldots, expr_n$ `constrain` $expr_c$	$(expr_c \Rightarrow pc' = pc + 1 \;\wedge\; \forall i \in \{1, \ldots, n\}, v'_i = expr_i \;\wedge$ $same(V \setminus \{v_1, \ldots, v_n\})) \wedge (\neg expr_c \Rightarrow \bot))$
`goto` $l_1 \ldots, l_n$	$\bigvee_{l \in \{l_1, \cdots, l_n\}} pc' = l \wedge same(V)$

We roughly adopt the Boolean program syntax from [3]; Table 1 defines the valid statements and their semantics. The symbol *pc* represents the program

[3] Recursion, in particular, renders the concurrent verification problem undecidable, even for Boolean programs. An option is to use overapproximations, as done in [24].

(a)

```
for(i=0; i < apthr_per_child; i++) {
  int status = ap_scoreboard_image->servers[child_argno][i].status;

  if (status != SRV_GRACEFUL && status != SRV_DEAD) continue;
  apr_status_t rv = apr_thread_create(&threads[i], thread_attr,
                          worker_thread, my_info, pchild);
  if (rv != APR_SUCCESS) {
    ap_log_error(APLOG_MARK, APLOG_ALERT, rv,
                     ap_server_conf, "apthr_create:␣error");
    clean_child_exit(CHILDSICK);
  }
  threads_created++;
}
```

(b)

```
main() begin
  decl i_lt_apthr_per_child, status_eq_SRV_GRACEFUL,
       status_eq_SRV_DEAD, rv_eq_APR_SUCCESS;

  L1: goto L2, L5;
  L2: assume i_lt_apthr_per_child;
      status_eq_SRV_GRACEFUL, status_eq_SRV_DEAD := *, *;
      goto L3,L4;
  L3: assume (!status_eq_GRACEFUL) && (!status_eq_SRV_DEAD);
      i_lt_apthr_per_child := *; goto L1;
  L4: rv_eq_APR_SUCCESS := true;
      skip;
      i_lt_apthr_per_child := *; goto L1;
  L5: assume !i_lt_apthr_per_child;
end
```

Fig. 1: (a) A `C` program; (b) a possible translation into a Boolean program

counter, V the set of program variables. Primes represent the next-state value of variables, and $\mathit{same}(X)$ abbreviates $\bigwedge_{v \in X} v' = v$. Well-formed expressions are the Boolean closure of constants *true*, *false* and $\star$ (representing either value), and variable identifiers. The constructs `assume` *expr* and `skip` are shorthands for v `:=` v `constrain` *expr* and `assume` *true*, respectively.

A Boolean program $\mathcal{P}$ induces a *concurrent system* as follows. We define $\mathcal{P}_n := |||_{i=1}^{n} \mathcal{P}$ to be the interleaved parallel composition of n threads executing $\mathcal{P}$. Program $\mathcal{P}_n$ consists of each variable declared *global* in $\mathcal{P}$, and n copies of each variable declared *local* in $\mathcal{P}$. A *state* of $\mathcal{P}_n$ can therefore be described in the form $(g, l_1, \ldots, l_n)$, where vector g is a valuation of the global variables, and l_i stands for the *local state* of thread i; it comprises the value of the program counter pc_i and the valuation of the i-th copy of each local variable. The *thread state* of thread i is the pair (g, l_i). Intuitively, for $i \in \{1, \ldots, n\}$, thread i has full access to the global variables and to the i-th copy of the local variables. It has neither read nor write access to any other local variables. We assume a standard asynchronous execution model for $\mathcal{P}_n$: exactly one thread executes an instruction at any time. A full formalization is given in [24].

3.2 Counter Abstraction

Counter Abstraction can be viewed as a form of symmetry reduction, as follows. *Full symmetry* is the property of a Kripke model $M = (S, R)$ of concurrent components to be invariant under permutations of these components. This invariance is traditionally formalized using permutations acting on component indices. A permutation π on $\{1, \ldots, n\}$ is defined to act on a state $s = (g, l_1, \ldots, l_n)$ by acting on the thread indices, i.e. $\pi(s) = (g, l_{\pi(1)}, \ldots, l_{\pi(n)})$. We extend π to act on a transition (s, t) by acting point-wise on s and t.

Definition 1 *Structure M is* ***(fully) symmetric*** *if for all $r \in R$ and all permutations π on $\{1, \ldots, n\}$, $\pi(r) \in R$.*

We observe that a concurrent Boolean program built by *replicating* a template written in the syntax given in section 3.1 is (trivially) symmetric: the syntax does not allow thread identifiers in the program text, which could potentially break symmetry.

From a symmetric M, a reduced *quotient structure* $\overline{M}$ can be constructed using standard existential abstraction. The quotient is based on the *orbit relation* on states, defined as $s \equiv t$ if there exists π such that $\pi(s) = t$. Quotient $\overline{M}$ turns out to be bisimilar to the original model M [25, 26]. Thus, verification over M can be replaced by verification over the smaller $\overline{M}$, without loss of precision. In addition, $\overline{M}$ is roughly exponentially smaller than M: the equivalence classes of $\equiv$ collapse up to $n!$ many states of M. Symmetry reduction thus combines two otherwise antagonistic features of abstractions – precision and compression.

Counter abstraction is an alternative formalization of symmetry, namely using process counters. The idea is that two global states are identical up to permutations of the local states of the components exactly if, for every local state L, the same number of components reside in L. To implement this idea, we introduce a counter for each existing local state and translate a transition from local state A to local state B as a decrement of the counter for A and an increment of that for B. With some effort, this translation can actually be performed statically on the text of a symmetric program $\mathcal{P}$, *before* building a Kripke model. The resulting counter-abstracted program $\widehat{\mathcal{P}}$ gives rise to a Kripke structure $\widehat{M}$ whose reachable part is *isomorphic* to that of the traditional quotient $\overline{M}$ and that can be model-checked without further symmetry considerations.

Counter abstraction can be viewed as a translation that turns a state space of potential size l^n (n local states over $\{1, \ldots, l\}$) to one of potential size $(n+1)^l$ (l counters over $\{0, \ldots, n\}$). The technique is thus expected to yield a reduction whenever $n \gg l$. From a theoretical viewpoint, this is the case asymptotically if l is a constant and n is conceptually unbounded. This view does not, however, withstand a practical evaluation, as we shall see in the next section.

4 Symbolic Counter Abstraction

In this section, we present the contribution of this paper, a symbolic algorithm for state space exploration of concurrent Boolean programs that achieves efficiency through counter abstraction. Before we illustrate the data structures used to store system states compactly, and describe our algorithm, we illustrate the problems a naive implementation of counter abstraction will inevitably entail if applied to practical software.

4.1 Classical Counter Abstraction – Merits and Problems

Classical counter abstraction assumes that the behavior of a single process is given as a local state transition diagram. This abstraction level is useful, for instance, in high-level communication protocols implementing some form of N-T-C mutual exclusion. In this case, counter abstraction reduces an n-process Kripke structure of exponential size $\mathcal{O}(3^n)$ to one of low-degree polynomial size $\mathcal{O}(n^3)$. The latter structure can be model-checked for hundreds if not thousands of processes.

This approach is problematic, however, for concurrent *software*, where thread behavior is given in the form of a program that manipulates local variables. The straightforward definition of a local state as a valuation of all thread-local variables is incompatible in practice with the idea of counter abstraction: the number of local states generated is simply too large. Consider again the Boolean program in Figure 1. It declares only four local Boolean variables and the PC with range $\{1, \ldots, 12\}$, giving rise to already $2^4 * 12 = 192$ local states. In applications of the magnitude we consider, concurrent Boolean programs routinely have several dozens of thread-local variables and many dozens of program lines (even after optimizations), resulting in many millions of local states.

Generally, as a result of this *local state explosion* problem, the state space of the counter program is of size **doubly-exponential** in the number v of local variables, namely $\Omega(n^{2^v})$. Our approach to tackling the problem is two-fold:

1. Instead of statically translating each statement s of the input program into counter updates (which would require enumerating the many possible local states in which s is enabled), do the translation *on the fly*. This way we have to execute s only in the narrow context of a *given* and *reachable* local state.
2. Instead of storing the counter values for all local states in a global state, store only the *non-zero* counters. This (obvious) idea exploits the observation that, if $l \gg n$, in every system state most counters are zero.

As a result, the worst-case size of the Kripke structure of the counter-abstracted program is reduced from n^l to $n^{\min\{n,l\}}$, completely **eliminating** the sensitivity to the local state space explosion problem. In the rest of this section, we describe the symbolic state-space exploration algorithm that implements this approach.

4.2 A Compact Symbolic Representation

Resulting from predicate abstractions of `C` code, Boolean programs make heavy use of data-nondeterminism, in the form of the nondeterministic Boolean value $\star$. Enumerating all possible values an expression involving $\star$ can stand for is infeasible in practice. A better approach is to interpret the value $\star$ symbolically, as the set $\{0, 1\}$. This interpretation is not only compatible with encodings of Boolean programs using BDDs, but can also be combined well with counter abstraction.

Our approach to counter abstraction is to count *sets* of local states represented by propositional formulas, rather than individual local states. Consider 3 threads executing a Boolean program with a single local Boolean variable x, and the global state $s = (\star, \star, \star)$: all threads satisfy $x = \star$. Defining the local state set $B := \{0, 1\}$, we can represent s compactly as the single abstract state characterized by $n_B = 3$, indicating that there are 3 threads whose value for x belongs to B (all other counters are zero).

To formalize our state representation, let L be the set of conceivable local states, i.e., $|L| = l$. An abstract global state takes the form of a set G of valuations of the global variables, followed by a list of pairs of a local state set and a counter:

$$\langle G, (L_1, n_1), \ldots, (L_k, n_k) \rangle \ . \tag{1}$$

In this notation, $L_i \subseteq L$ and $n_i \in \mathbb{N}$. We further maintain the invariants $\sum_{i=1}^{k} n_i = n$ and $n_i \geq 1$. The semantics of this representation is given by the set of concrete states that expression (1) represents, namely the states of the form $(g, l_1, \ldots, l_n)$ such that

$$\begin{array}{ll} \text{(a)} & g \in G, \text{ and} \\ \text{(b)} & \text{there exists a partition } \{I_1, \ldots, I_k\} \text{ of } \{1, \ldots, n\} \text{ such that} \\ & \text{for all } i \in \{1, \ldots, k\},\ |I_i| = n_i \text{ and for all } j \in I_i,\ l_j \in L_i. \end{array} \tag{2}$$

That is, an abstract state of the form (1) represents precisely the concrete states in the Cartesian product of valuations of the global variables in G, and valuations of the local variables satisfying the constraint (2) (b). We use separate BDDs to represent the sets G and L_i. Let V_g and V_l denote the sets of shared and local variables in $\mathcal{P}$, respectively, and *pc* the program counter. We represent the set G using a predicate f over the variables in V_g, and each set L_i using a predicate f_i over the variables in $V_l \cup \{pc\}$.

This representation has a caveat: constraints between global and local variables, such as introduced by an assignment of the form *global* := *local*, cannot be expressed, since the defining predicates for G and L_i may not refer to variables from *both* V_g and V_l. Clearly, however, Boolean programs can introduce such constraints. Section 4.3 describes how our algorithm addresses this problem.

Intuitively, each pair (L_i, n_i) represents the n_i threads such that the *most precise* information on their local states is that they belong to L_i. For instance, the abstract global state $\langle (x = 0,\, 3),\, (x = \star,\, 4) \rangle$ represents those concrete states where 3 threads satisfy $x = 0$, whereas we have no information on x for the remaining 4 threads. This example also shows that we do not require the sets L_i to be disjoint: forcing the symbolic local $x = 0$ to be merged into the symbolic local state $x = \star$ would imply a loss of information, as the constraint $x = 0$ is more precise.

Traditional approaches that statically counter-abstract the entire input program often use a data structure that can be seen as a special case of (1), namely with $k = l = |L|$. Such implementations do not enforce the invariant $n_i \geq 1$ and thus suffer from the potential redundancy of n_i being 0 for most i.

4.3 Symbolic State Space Exploration

We present a symbolic algorithm for reachability analysis of symmetric Boolean programs for on-the-fly counter abstraction that employs the state representation described in section 4.2. The input consists of a template program $\mathcal{P}$ and the number n of concurrent threads; the algorithm computes the counter-abstracted set of states reachable from a given set of initial states.

Algorithm 1 Symbolic counter abstraction algorithm

```
 1: $\mathcal{R} := \{\langle G_0, (L_0, n)\rangle\}$, insert $\langle G_0, (L_0, n)\rangle$ into $\mathcal{W}$        ▷ $n$ threads at location 0
 2: while $\mathcal{W} \neq \emptyset$ do
 3:     remove $S = \langle G, F\rangle$, with $F = \{(L_1, n_1), \ldots, (L_k, n_k)\}$, from $\mathcal{W}$
 4:     for $i \in \{1, \ldots, k\}$ do
 5:         $T := \langle G, L_i\rangle$                                  ▷ extract thread state from $S$
 6:         for $v \in$ all valuations of $\mathit{SpliceVariables(T)}$ do
 7:             $T' = \langle G', L'\rangle := \mathsf{Image}(T|_v)$            ▷ compute one image cofactor of $T$
 8:             if $L' \neq L_i$ then                         ▷ build new system state $S'$ from $T'$
 9:                 $S' := \langle G', \textsc{UpdateCounters}(F,\ i,\ L')\rangle$
10:                 if $S' \notin \mathcal{R}$ then
11:                     $\mathcal{R} := \mathcal{R} \cup S'$                    ▷ store $S'$ as reachable, if new
12:                     insert $S'$ into $\mathcal{W}$

13: procedure $\textsc{UpdateCounters}(F,\ i,\ L')$
14:     let $(L_i, n_i)$ be the $i$-th pair in $F$
15:     $O := (n_i > 1\ ?\ \{(L_i, n_i - 1)\}\ :\ \emptyset)$              ▷ determine if $L_i$ is abandoned
16:     if $\exists j.\,(L_j, n_j) \in F \wedge L' = L_j$ then          ▷ update or add a state-counter pair
17:         $F' := F \setminus \{(L_i, n_i), (L_j, n_j)\} \cup O \cup \{(L', n_j + 1)\}$
18:     else
19:         $F' := F \setminus \{(L_i, n_i)\} \cup O \cup \{(L', 1)\}$
20:     return $F'$
```

Algorithm 1 expands unexplored system states from a worklist $\mathcal{W}$, initialized to contain the symbolic state that constrains all threads to location 0. The loop in line 4 iterates over all pairs (L_i, n_i) contained in the popped state S. To expand an individual pair, the algorithm first projects it to the ith symbolic thread state.

The next, and crucial, step is to compute the successor thread states induced by the Boolean program (lines 6–7). Recall from the previous section that our Cartesian state representation does not permit constraints between global and local variables, which can, however, be introduced by the program. Consider a global variable a and a local b, and the statement `a := b`. The concrete thread state (a', b') obtained after executing the statement is characterized by the constraint $a' \equiv b'$. In order to make this constraint expressible, we must treat certain assignments and related statements specially.

Definition 2 *A **splice state** is a symbolic thread state given as a predicate f over the variables in $V_g \cup V_l \cup \{pc\}$ such that*

$$(\exists V_g \, . \, f) \ \wedge \ (\exists V_l \, \exists pc \, . \, f) \quad \not\equiv \quad f \, .$$

*A **splice statement** is a statement s such that there exists a thread state u whose PC points to s and that, when executed on u, results in a splice state. A **splice variable** is a global variable dependent on $\exists V_g \, . f$.*

A splice statement marks a point where a thread communicates data via the global variables, in a way that constrains its local state with the values of some splice variables. Fortunately, statements with the *potential* to induce such communication can be identified syntactically:

- assignments whose left-hand side is a global variable and the right-hand side expression refers to local variables, or vice versa,
- assignments with a `constrain` clause whose expression refers to both global and local variables, and
- `assume` statements whose expression refers to both global and local variables.

Before executing a splice statement, the current thread state is *split* using Shannon decomposition. Executing the statement on the separate cofactors yields a symbolic successor that can be represented precisely in the form (1). That is, if variable v is the splice variable of the statement in T, denoted by $\mathit{SpliceVariables}(T) = \{v\}$, we decompose $\mathsf{Image}(T)$ as follows:

$$\mathsf{Image}(T) = \mathsf{Image}(T|_{v=0}) \vee \mathsf{Image}(T|_{v=1}) \, .$$

The price of this expansion is an explosion worst-case exponential in the number of splice variables. However, as we observed in our experiments (see section 5),

1. the percentage of splice statements is relatively small,
2. even within a splice statement, the number of splice variables involved is usually very small (1 or 2),
3. a significant fraction of cofactors encountered during the exploration is actually *unsatisfiable* and does not result in new states.

As a result, the combinatorial explosion never materialized in our experiments.

After the image has been computed for each cofactor, the algorithm constructs the respective system state for it (lines 8–9). The UpdateCounters function uses the local state part L' of the newly computed thread state to determine the new set of (state,counter) pairs F'. If no more threads reside in the "left" state L_i, its pair is expunged (line 15); if the new local state L' was already present in the system state, its counter n_j is incremented, otherwise the state is added with counter value 1 (lines 16–19).

Finally, the algorithm adds the states encountered for the first time to the set of reachable states, and to the worklist of states to expand (lines 10–12).

Theorem 3 *Let $\mathcal{R}$ be as computed by Algorithm 1 on termination, and let γ be the concretization function for abstract states defined in equation (2). The set $\gamma(\mathcal{R}) = \{\gamma(r) | \; r \in \mathcal{R}\}$ is the set of reachable states of the concurrent system induced by n threads executing the Boolean program $\mathcal{P}$ in parallel.*

The proof of this theorem, and that of the termination of Algorithm 1, follow from (i) the equivalent theorems for classical state space exploration under symmetry using canonical state representatives, and (ii) the isomorphism of the structures over such representatives and the counter representation.

Errors are detected by Algorithm 1 in the form of program locations that contain a violated assertion. We have omitted from the algorithm a description of the standard mechanisms to trace back a reached state to the initial state, in order to obtain error paths. Such a path is, in our case, a sequence of abstract states in the form (1). This abstract trace can be mapped to a concrete trace, by expanding each abstract state to a concrete one according to equation (2), and ensuring that, in the resulting sequence, any state differs from its successor in only one thread. Note that, as our method is exact with respect to the given Boolean program, no spurious reachability results occur.

5 Experimental Evaluation

In addition to the symbolic version of the algorithm, we have also implemented an explicit-state version, both in a tool called Boom (available at http://www.cprover.org/boom). The symbolic implementation stores the sets G and L_i of global and local variable valuations as individual BDDs, such that the conjunction of G with each L_i forms the thread-visible state T_i. As in most symbolic Model Checkers for software, the program counters are stored in explicit form: this permits partitioning the transition relation and ensures a minimum number of splice tests.

We applied Boom to over 400 examples from two sources: a set of 208 concurrent Boolean programs generated by SatAbs that abstract part of the Linux kernel components (available at http://www.cprover.org/boolean-programs), and a set of 236 Boolean programs generated at Microsoft Research using Slam. We build a concurrent out of a sequential Boolean program by instantiating n threads that execute the main procedure. All Boolean programs involve communication among threads through global variables. The 444 benchmarks feature, on average, 123 program locations, 21 local variables, and 12 global variables.

We compare the explicit-state implementation to Murφ [10], a mature and popular Model Checker with long-standing support for symmetry reduction. Since other symbolic model checkers did not scale to interesting thread counts (including the few tools with built-in support for symmetry), we compare the symbolic algorithm to a reference implementation in Boom that ignores the symmetry. On sequential programs, the performance of the reference implementation is similar to that of the Model Checker that ships with Slam.

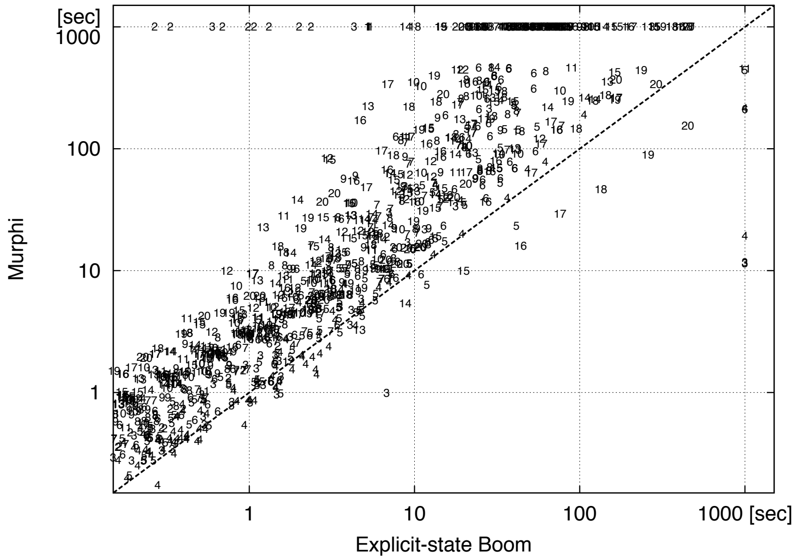

Fig. 2: Running time of explicit-state Boom vs Murφ, for various thread counts

The experimental setup is as follows. For each tool and benchmark, we start full reachability analysis with $n = 2$ threads, and increase the thread count until the tool times out; the timeout is set to 720 s and the memory limit to 12 GB[4].

Figure 2 is a scatter plot of the running times of the explicit-state version of Boom and of Murφ with symmetry reduction. Since Murφ does not allow data nondeterminism, we replace every occurrence of $\star$ in the input programs randomly by 0 or 1, before passing them to either tool. The resulting programs are converted into Murφ's input language using a Murφ *rule* per statement, guarded by the program counter value. Counter abstraction is faster than Murφ on 94 % of the tests; on 23 %, the improvement is better than one order of magnitude. It completes successfully on a significant number of problems where Murφ times out (19 %). In seven cases (1.2 %), our tool runs out of memory. Note that removing the data nondeterminism simplifies the programs, which is why the explicit-state explorations often terminate for larger thread counts than the symbolic ones.

Figure 3 summarizes the running times of the symbolic counter abstraction implementation in Boom and the plain symbolic exploration algorithm. The uniform distribution in the upper triangle signifies the improvement in scalability due to counter abstraction. Those tests where traditional Model Checking is faster feature a small number of threads, reflecting the polynomial complexity of counter abstraction; in fact, we can verify many instances for 7 or more threads. Overall, the symbolic counter abstraction tool is faster on 83 % of all tests, and

[4] The experiments are performed on a 3GHz Intel Xeon machine running the 64-bit variant of Linux 2.6.

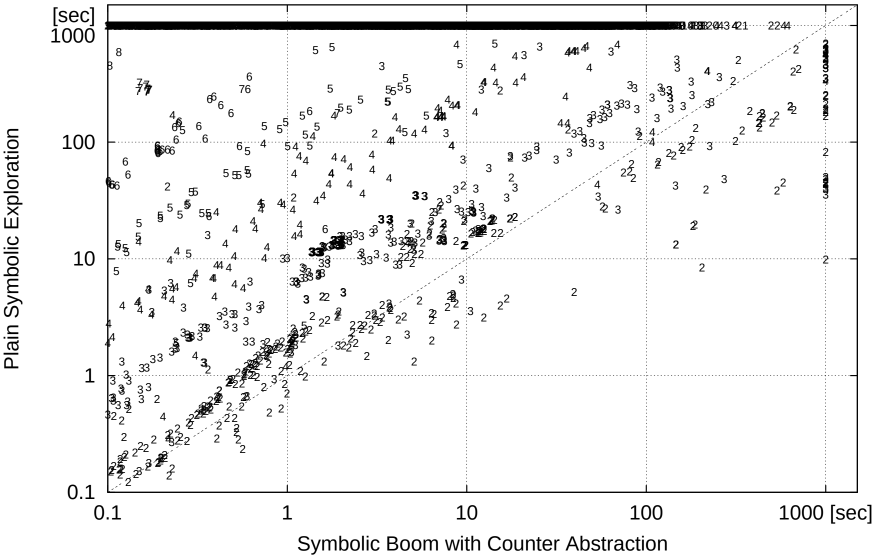

Fig. 3: Running time of symbolic Boom vs plain exploration, for various thread counts

on 96 % of those running three or more threads. Among those, the speed-up is five orders of magnitude and more.

Splice statements do not cause a blow-up in any of the benchmarks. In fact, they amount to less than 12 % of all statements, the average number of splice variables they involve is small (in our benchmarks, mean 2.1, median 1), and each such variable produces two valid cofactors in only 10 % of the cases.

Our implementation of the last step of Algorithm 1 (lines 20–21) uses state merging, a crucial optimization to compact sets of symbolic states. In counter abstraction, two symbolic states can be merged iff a) they differ only in the valuation of the global variables, or b) they differ in the local state of only *one* thread. These merging rules, albeit apparently strict, provide an average speed-up of 83 % over exploration without merging.

We have also considered global state representations alternative to equation (1). In one implementation, we use a monolithic BDD to represent the global variables and all thread states, along with their counters. In another, we separate the counters from the BDD and keep them explicit, but use a monolithic BDD for all other variables. Both allow us to retain the inter-thread constraints introduced by splice statements, and thus do not require the decomposition step. However, both do require complex manipulations for computing successor states, especially for updating the counters. Moreover, they give rise to very complex BDDs for the set of reachable states, which foils the scalability advantage inherent in counter abstraction. On our benchmarks, the algorithm proposed in Section 4 is at least 30 % faster than all alternatives.

6 Summary

We have presented an algorithm for BDD-based symbolic state space exploration of concurrent Boolean programs, a significant branch of the pressing problem of concurrent software verification. The algorithm draws its efficiency from counter abstraction as a reduction technique, without having to resort to approximation at any time. It is specifically designed to cope with large numbers of local states and thus addresses a classical bottleneck in implementations of counter abstraction. We showed how to avoid the *local state space explosion* problem using a combination of two techniques: 1) interleaving the translation with the state space exploration, and 2) ensuring that only non-zero counters and their corresponding local states are kept in memory.

We have shown experimental results both for an explicit-state and, more importantly, a symbolic implementation. While standard symmetry reduction is employed in tools like MURφ and RULEBASE, we are not aware of a prior implementation of counter abstraction that is efficient on programs other than abstract protocols with relatively few control states. We believe our model checker to be the first with a true potential for scalability in concurrent software verification, due to its polynomial dependence on the thread count n, while incurring little verification time overhead.

An interesting question for future work is the relationship between our work and *partial-order reduction*. The latter is generally viewed as combinable with symmetry reduction for yet better compression. We have preliminary experiments that indicate this is true for our symbolic counter abstraction algorithm as well. We also plan to investigate how our algorithms can be combined with other techniques to curb the local state space explosion problem, such as based on static analysis or approximation (see [13]). Finally, we want to extend our techniques to richer specification languages, especially Boolean programs with dynamic thread creation.

References

1. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Computer-Aided Verification (CAV). (1997)
2. Lahiri, S.K., Bryant, R., Cook, B.: A symbolic approach to predicate abstraction. In: Computer-Aided Verification (CAV). (2003)
3. Ball, T., Rajamani, S.: Bebop: A symbolic model checker for Boolean programs. In: Model Checking of Software (SPIN). (2000)
4. Kurshan, R.: Computer-Aided Verification of Coordinating Processes. Princeton University Press (1995)
5. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S., Ustuner, A.: Thorough static analysis of device drivers. In: EuroSys. (2006)
6. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent Linux device drivers. In: Automated Software Engineering (ASE). (2007)

7. Emerson, A., Trefler, R.: From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In: Correct Hardware Design and Verification Methods (CHARME). (1999)
8. Donaldson, A., Miller, A.: Symmetry reduction for probabilistic model checking using generic representatives. In: Automated Technology for Verification and Analysis (ATVA). (2006)
9. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). (2005)
10. Melton, R., Dill, D.: Murϕ Annotated Reference Manual, rel. 3.1, `http://verify.stanford.edu/dill/murphi.html`
11. Lubachevsky, B.: An approach to automating the verification of compact parallel coordination programs. Acta Informatica (1984)
12. Pnueli, A., Xu, J., Zuck, L.: Liveness with $(0, 1, \infty)$-counter abstraction. In: Computer-Aided Verification (CAV). (2002)
13. Emerson, A., Wahl, T.: Efficient reduction techniques for systems with many components. In: Brazilian Symposium on Formal Methods (SBMF). (2004)
14. Wei, O., Gurfinkel, A., Chechik, M.: Identification and counter abstraction for full virtual symmetry. In: Correct Hardware Design and Verification Methods (CHARME). (2005)
15. Emerson, A., Havlicek, J., Trefler, R.: Virtual symmetry reduction. In: Logic in Computer Science (LICS). (2000)
16. Ball, T., Chaki, S., Rajamani, S.: Parameterized verification of multithreaded software libraries. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). (2001)
17. Henzinger, T., Jhala, R., Majumdar, R.: Race checking by context inference. In: Programming Language Design and Implementation (PLDI). (2004)
18. Andrews, T., Qadeer, S., Rajamani, S., Rehof, J., Xie, Y.: Zing: A model checker for concurrent software. In: Computer-Aided Verification (CAV). (2004)
19. Blanc, N., Emerson, A., Wahl, T.: Sviss: Symbolic verification of symmetric systems. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). (2008)
20. Barner, S., Grumberg, O.: Combining symmetry reduction and under-approximation for symbolic model checking. Formal Methods in System Design (FMSD) (2005)
21. Cook, B., Kroening, D., Sharygina, N.: Symbolic model checking for asynchronous Boolean programs. In: Model Checking of Software (SPIN). (2005)
22. Suwimonteerabuth, D., Esparza, J., Schwoon, S.: Symbolic context-bounded analysis of multithreaded Java programs. In: Model Checking of Software (SPIN). (2008)
23. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Principles of Programming Languages (POPL). (2005)
24. Cook, B., Kroening, D., Sharygina, N.: Verification of Boolean programs with unbounded thread creation. Theoretical Computer Science (TCS) (2007)
25. Clarke, E., Enders, R., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. Formal Methods in System Design (FMSD) (1996)
26. Emerson, A., Sistla, P.: Symmetry and model checking. Formal Methods in System Design (FMSD) (1996)