Form Methods Syst Des (2010) 36: 223–245
DOI 10.1007/s10703-010-0096-7

Context-aware counter abstraction

Gérard Basler · Michele Mazzucchi · Thomas Wahl · Daniel Kroening

Published online: 11 June 2010

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. We have carried out experiments on Boolean programs, an abstraction promoted by the success of the SLAM project. The experiments give evidence of the applicability of our method to realistic programs, and of the often huge savings obtained in comparison to plain symbolic state space exploration, and to exploration optimized by partial-order methods. To our knowledge, our tool marks the first implementation of counter abstraction to programs with non-trivial local state spaces, resulting in a Model Checker for concurrent Boolean programs that promises true scalability.

Keywords Symmetry reduction · Boolean programs · Symbolic Model Checking

1 Introduction

Software Model Checking has been a vibrant branch of research in formal methods for several years. *Predicate abstraction* [23, 27] is one of the most prominent approaches in this

This research is supported by the EU FP7 STREP MOGENTES (project ID ICT-216679), and by the EPSRC project EP/G026254/1. This paper is an extended version of a conference paper that appeared at CAV 2009 [6].

G. Basler (✉) · M. Mazzucchi · T. Wahl · D. Kroening
Computer Systems Institute, ETH Zurich, Zurich, Switzerland
e-mail: gerard.basler@gmail.com

G. Basler · M. Mazzucchi · T. Wahl · D. Kroening
Computing Laboratory, Oxford University, Oxford, UK

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* [1], which exclusively uses Boolean variables. Embedded in an automated abstraction-refinement framework [26], verifiers for Boolean programs have been used successfully to increase the reliability of system-level software such as Windows device drivers [4].

Recently, there have been attempts to extend these techniques to the verification of *concurrent* software [37]. 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 [37] conclude that none of the currently available tools is able to handle device drivers of realistic size in the presence of many threads.

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 for each local state, tracking how many of the n components currently reside in the local state. This technique turns a formal model of size exponential in n into one of size polynomial in n, promising a serious stab at state space explosion.

Emerson and Trefler proposed counter abstraction as a way of achieving *symmetry reduction* for fixed-size systems [17]. In their approach, the template program $\mathbb{P}$ is converted 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 [15]. For concurrent software, however, $\mathbb{P}$ 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 thread-local variables of a thread. As a result, there are exponentially many local states, measured in the number of thread-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 our algorithm is *context-aware*: the local-state context in which the statement is executed is known; the context determines which local-state counters need to be updated. If the translation is performed up-front, one has to embed each statement into *all* contexts in which the statement is enabled, which is infeasible for realistic programs. As a side-effect of the on-demand translation, only counters for *reachable* local states are ever introduced. Second, in a global state we keep counters only for those local states that at least one thread resides in. This idea leverages 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 orders of magnitude smaller than l, omitting the zero-valued counters results in huge savings: the complexity of counter abstraction 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 executed by a bounded number of possibly dynamically created parallel threads. This generalizes the traditional setting for symmetry reduction of systems with a fixed number of components known at modelling time. 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 [2]. Since symmetry reduction, of which finitary counter abstraction is an instance, has so far been implemented more successfully in explicit-state Model Checkers, we also include an experimental comparison of an explicit-state version of our method against explicit-state symmetry reduction, using the well-known Murφ Model Checker [30]. Finally, this paper provides an extensive comparison of our counter abstraction method with *partial-order reduction*, an alternative technique to curb the verification complexity for programs with interleaved concurrent threads.

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.

2 Related work

While the principal idea of using process counters already appeared in early work by Lubachevsky [28], *generic representatives* were suggested by Emerson and Trefler [17] as a means of addressing the complexity of symmetry-reducing symbolically represented systems. The term *counter abstraction* was coined by Pnueli, Xu, and Zuck in the context of parameterized verification of liveness properties [31]. The counters are cut off at some value c, indicating that *at least* c components currently reside in the corresponding local state. 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.

Local state-space explosion was identified by Emerson and Wahl as the major obstacle to using generic representatives with non-trivial symmetric programs [19]. 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: by Wei et al. [36], in the context of *virtual symmetry* [21], and by Donaldson and Miller [14], 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 [36]) 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, 36] include communication and mutual-exclusion protocols with at most a few dozen local states. Counter abstraction for infinite-state verification has been applied by Delzanno [12] and Pong and Dubois [32], in the context of high-level protocol verification, rather than software. The Beacon Model Checker [3] has been applied to a multi-threaded memory management system with 256 local states. In our benchmarks, threads have millions of local states (see Sect. 7).

Henzinger, Jhala, and Majumdar apply 0-1-∞ counter abstraction to predicate-abstracted concurrent C programs for race detection [24]. 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 the authors, 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 methods such as in MURφ [30] and SPIN [7, 13] (explicit-state) or SVISS [35] and RULEBASE [5] (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 [10, 34, and others]. None of these address the symmetry that concurrent Boolean programs exhibit, although some investigate partial-order methods [22]. The present paper includes an extensive experimental comparison of our proposed algorithm to partial-order reduction techniques; see Sect. 7.3.

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 $\mathbb{P}$. To enable sound verification of reachability properties, the Boolean program is constructed to overapproximate the behavior of $\mathbb{P}$. This may permit spurious paths, which need to be detected and eliminated, by refining the abstraction using additional predicates. This process is well-known as *counterexample-guided abstraction refinement* (CEGAR) [26].

In a preprocessing step, loops and `if` statements in the `C` program are replaced by nondeterministic `goto`s and `assume` statements. Function calls are inlined; the Boolean programs used in our experiments are assumed to be free of unbounded recursion.[1] The resulting "`goto` program" is then translated into a Boolean program. Figure 1 shows the result of this translation, applied to a fragment of the Apache webserver suite.

We roughly adopt the Boolean program syntax from [1]; Table 1 defines the valid statements and their semantics. The symbol pc represents the program counter, V the set of program variables. Primes represent the next-state value of variables, and $same(Z)$ abbreviates $\bigwedge_{v \in Z} v' = v$, for some set of variables Z. The set of well-formed expressions is the Boolean closure of constants 0, 1, $\star$, and variable identifiers. The symbol $\star$ nondeterministically evaluates to *false* or *true*. For example, an assignment of the form $v := expr$, where $expr$ contains an occurrence o of $\star$, is evaluated as $v' = expr|_0 \vee v' = expr|_1$, where $expr|_0$ and $expr|_1$ are obtained from $expr$ by replacing o by 0 and 1, respectively. Multiple occurrences of $\star$ in one expression may evaluate independently to 0 or 1. Moreover, a single occurrence of $\star$ may evaluate differently every time the containing expression is evaluated.

We use the constructs `assume` *expr* and `skip` as shorthands for $v := v$ `constrain` *expr* (using any variable v) and `assume 1`, respectively. The `start_thread` and

[1]Recursion renders the concurrent verification problem undecidable, even for Boolean programs.

```
while (1) {
  for (i = 0; i < ap_threads_per_child; i++) {

    int status = ap_scoreboard_image->servers[child_num_arg][i].status;
    if (status != SERVER_GRACEFUL && status != SERVER_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,
                   "apr_thread_create: cannot create worker thread");
      clean_child_exit(APEXIT_CHILDSICK);
    }
    threads_created++;
  }
  // handle service requests
}
```

(a) C program

```
main() {
  decl i_lt_ap_threads_per_child, status_eq_SRV_GRACEFUL,
       status_eq_SRV_DEAD, rv_eq_APR_SUCCESS;

       // predicate for "i < ap_threads_per_child":
  L0: i_lt_ap_threads_per_child := true;
  L1: goto L2, L9;                                // for-loop
  L2: assume (i_lt_ap_threads_per_child); // for-loop
      status_eq_SRV_GRACEFUL, status_eq_SRV_DEAD := ⋆, ⋆;
      goto L3, L4;
  L3: assume (status_eq_GRACEFUL || status_eq_SRV_DEAD);
      goto L8; // "continue"
  L4: assume (!status_eq_GRACEFUL && !status_eq_SRV_DEAD);
      start_thread Li; // Li: initial location in worker_thread function
  L5: rv_eq_APR_SUCCESS := ⋆;
      goto L6, L7;                    // "if (rv != APR_SUCCESS)"
  L6: assume (!rv_eq_APR_SUCCESS);   // "if (rv != APR_SUCCESS)"
      rv_eq_APR_SUCCESS := ⋆; // possible side effect of ap_log_error(...)
      goto L1; // end for
  L7: assume (rv_eq_APR_SUCCESS);
  L8: i_lt_ap_threads_per_child := ⋆; // "i++"
      goto L1; // end for
  L9: assume (!i_lt_ap_threads_per_child);
      // handle service requests
      ...
      goto L0; // end while
}
```

(b) Boolean program

Fig. 1 A C program and a possible translation into a Boolean program

Table 1 Semantics of fundamental Boolean program statements

Syntax	Semantics
$v_1, \ldots, v_z := expr_1, \ldots, expr_z$ constrain $expr_c$	$(expr_c \Rightarrow (pc' = pc + 1 \wedge \forall i \in \{1, \ldots, z\} v'_i = expr_i \wedge same(V \setminus \{v_1, \ldots, v_z\}))) \wedge (\neg expr_c \Rightarrow \perp)$
goto $l_1, \ldots, l_z$	$(\bigvee_{l \in \{l_1, \ldots, l_z\}} pc' = l) \wedge same(V)$
start_thread Q	$pc' = pc + 1 \wedge same(V)$ (see main text)
end_thread	*true* (see main text)

end_thread commands are used in Boolean programs that are executed concurrently by multiple threads; these commands are discussed in the next section. The table only shows their effects on the executing thread; see the next paragraph for side effects.

We sketch how a Boolean program $\mathbb{P}$ induces a *concurrent system* $\mathbb{P}^{||}$; a full formalization is given by Cook et al. [11]. The set V of program variables is partitioned into two subsets V_s and V_l of *shared* and *thread-local* variables, respectively. A (global) state τ of $\mathbb{P}^{||}$ has the form (n, PC, Ω), where n is the number of threads running in state τ, function $PC\colon \{1, \dots, n\} \to \{1, \dots, pc_{\max}\}$ maps each thread identifier to the program counter location pointing to the next statement to be executed by that thread, and $\Omega\colon\ V_s \cup (\{1, \dots, n\} \times V_l) \to \mathbb{B} \cup \{\star\}$ is the valuation of the program variables.

The execution model of $\mathbb{P}^{||}$ is asynchronous. That is, a step of $\mathbb{P}^{||}$ is performed by a single thread, say with identifier $i \in \{1, \dots, n\}$, executing the statement of $\mathbb{P}$ at location $PC(i)$. Changes to the values of pc and the variables in V are reflected in updates to the state components PC and Ω, as indicated by symbolic constraints in Table 1. The value of n changes exactly in two circumstances:

- Thread i executes a start_thread Q command. In this case, the state is updated as follows. Let N be the bound on the number of threads that may be created. If $n < N$, then $n' = n + 1$, $PC'(i) = PC(i) + 1$, $PC'(n') = Q$, and for each thread-local variable $v_l \in V_l$, $\Omega((n', v_l)) = \Omega((i, v_l))$, i.e., thread i is *cloned*. All other values are unchanged. If $n = N$, then $n' = n$, $PC'(i) = PC(i) + 1$, and all other values are unchanged. That is, if the number of dynamically created threads is exhausted, start_thread behaves like skip for the executing thread, and is free of side-effects.
- Thread i executes an end_thread command. In this case, $n' = n - 1$, and PC and Ω are unchanged.

Let finally n_0 be a natural number with $1 \le n_0 \le N$, the initial number of threads. The set of initial states of $\mathbb{P}^{||}$ is given by $n = n_0$ and $PC(i) = 0$; the values $\Omega(v_s)$ and $\Omega((i, v_l))$ are defined according to the initial values of each shared variable v_s and each thread-local variable v_l of each thread $i \in \{1, \dots, n_0\}$. A classical concurrent system of a fixed number of threads is an instance of this formalization with $n_0 = n = N$ and a Boolean program without start_thread or end_thread commands.

When reasoning about the concurrent program $\mathbb{P}^{||}$, the notation of a state can be simplified by considering the PC simply as a thread-local variable. In that case, a state of $\mathbb{P}^{||}$ can be described in the form $(s, l_1, \dots, l_n)$, where vector s is a valuation of the shared variables V_s, and l_i stands for the *local state* of thread i, comprising the value of the program counter $PC(i)$ and the value $\Omega((i, v_l))$, for each thread-local variable $v_l \in V_l$. The translation between the notations (n, PC, Ω) and $(s, l_1, \dots, l_n)$ is straightforward. The *thread state* of thread i is the pair (s, l_i). Intuitively, for $i \in \{1, \dots, n\}$, thread i has full access to the shared variables and to the i-th copy of the thread-local variables. It has neither read nor write access to the thread-local variables of any other thread.

3.2 Symmetry reduction

Full symmetry is the property of a Kripke model of concurrent components to be invariant under permutations of these components. This invariance is traditionally formalized using permutations. A permutation π on $\{1, \dots, n\}$ is defined to act on a state $\sigma = (s, l_1, \dots, l_n)$ by acting on the thread indices, i.e. $\pi(\sigma) = (s, l_{\pi(1)}, \dots, l_{\pi(n)})$. We extend π to act on a transition (σ, τ) by acting point-wise on σ and τ.

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

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

From a symmetric structure 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 $\sigma \equiv \tau$ if there exists π such that $\pi(\sigma) = \tau$.

Theorem 2 [8, 16] *Let f be a μ-calculus formula with atomic propositions that are invariant under thread index permutations. Let further σ be state of M and $\overline{\sigma}$ be the equivalence class of σ under $\equiv$.*

$$M, \sigma \models f \quad \textit{iff} \quad \overline{M}, \overline{\sigma} \models f.$$

Thus, verification over M can be replaced by verification over $\overline{M}$, without loss of precision. This theorem can be proved by a bisimulation argument; the bisimulation relation between M and $\overline{M}$ relates states of M and their orbit equivalence classes. 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 often antagonistic features of abstractions: precision and compression.

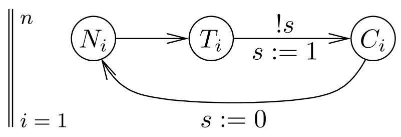
Symmetry reduction in the above formalization has been quite successful as an abstraction technique in model checkers based on explicit state enumeration; see Sect. 2 for examples. It has enjoyed much less popularity in BDD-based symbolic Model Checking. The reason is that the state canonization that is required by symmetry reduction can be expensive to perform using BDDs. In particular, it was shown in [8] that the orbit relation has no succinct BDD representation, for any variable order. Emerson and Wahl present a symbolic symmetry reduction technique that avoids the orbit relation but still relies on state canonization [18].

Counter abstraction, the topic of this paper, can be viewed as a form of symmetry reduction where state canonization is an automatic by-product of the state representation and thus does not have to be performed during Model Checking. The new representation, if used naively, has to be paid for with a blow-up of the program text, however, as we demonstrate in the next section.

4 Classical counter abstraction—merits and problems

Counter abstraction is an alternative formalization of symmetry reduction, 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 $\mathbb{P}$, *before* building a Kripke model. The resulting counter-abstracted program $\widehat{\mathbb{P}}$ 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.

```
shared boolean s := 0;
local enum {N,T,C} state := N;
```



```
boolean s := 0;
enum {0,...,n} n_N := n, n_T := 0, n_C := 0;
do
:: n_N > 0          → n_N--; n_T++
:: n_T > 0 && !s    → n_T--; n_C++; s := 1
:: n_C > 0          → n_C--; n_N++
od
```

Fig. 2 A model $\mathbb{P}$ of a semaphore-based Mutex algorithm (*left*); its counter-abstracted version $\widehat{\mathbb{P}}$ (*right*)

Let us look at an example. Classical counter abstraction assumes that the behavior of a single process is given as a local state transition diagram, as the one in Fig. 2 (left). This abstraction level is often used in descriptions of communication and cache-coherence protocols. The result of counter-abstracting this program is shown in the same figure on the right, in a guarded-command notation. We see that the reduction happens completely statically, i.e., on the program text. The new program is single-threaded; thus there are no notions of shared and thread-local variables. The Kripke structure corresponding to the program has shrunk from exponential size $\mathcal{O}(3^n)$ to low-degree polynomial size $\mathcal{O}(n^3)$. The reduced structure can be model-checked for hundreds if not thousands of processes.

In general, counter abstraction can be viewed as a translation that turns a state space of potential size l^n (n local states over $\{1,\dots,l\}$) to one of potential size $(n+1)^l$ (l counters over $\{0,\dots,n\}$). The abstraction therefore reduces a problem of size exponential in n to one of size polynomial in n. Since, for any given Boolean program, l is a constant, we appear to have solved the state-space explosion problem.

This view does not, however, withstand a practical evaluation for concurrent software, where thread behavior is given in the form of a program that manipulates thread-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. The Boolean program in Fig. 1 declares only four thread-local Boolean variables and the PC with range $\{1,\dots,12\}$, but already gives rise to $2^4 * 12 = 192$ 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 optimization), resulting in many millions of local states. As a result of this *local state explosion* problem, the state space of the counter program is of size $\Omega(n^{2^{|V_l|}})$, *doubly-exponential* in the number of thread-local variables.

Let us apply these observations to the complexity analysis of counter abstraction. As we have seen, the abstract state space has size high-degree polynomial in n. This means that only for very large values of n, the classical counter abstraction approach will offer benefits over a Model Checking strategy that ignores symmetry and stores the local state for each thread. Even if model checkers of the future are able to explore systems with huge thread counts, it seems hard to conceive a scenario in which one would care to analyze the behavior of a system with, say, 2 million threads. The goal of this paper is therefore to reap the benefits of counter abstraction even for thread counts that are small compared to the number of local states. Our approach 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), we make the algorithm *context-aware*, by triggering the translation *on the fly*. This way we have to execute s only in the narrow context of a given (and, thus, reachable) local state.

2. Instead of storing the counter values for all local states in a global state, we 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 proportional to n^l, to proportional to $n^{\min\{n,l\}}$, completely *eliminating* the sensitivity to the local state space explosion problem. In the rest of this paper, we describe the state space exploration algorithms that implement this approach.

5 Explicit-state counter abstraction

In the following two sections, we present two algorithms for reachability analysis of concurrent Boolean programs, with on-the-fly symmetry reduction implemented via counter abstraction. This section covers an explicit-state version and serves mainly to introduce the fundamental characteristics of the exploration algorithm and the data structures it relies on. The next section presents the symbolic algorithm, which requires significant modifications to ensure scalability in practice.

Inputs to the algorithms are a template program $\mathbb{P}$ and the initial number of concurrent threads n_0. The algorithms compute, in a variable $\mathcal{R}$, the symmetry-reduced set of states reachable from a given set of initial states. The algorithms *interleave* the exploration with the process of performing counter abstraction, so as to avoid having to build a potentially unmanageable local state transition diagram of the input program.

5.1 Compact counter-abstracted state representation

Central to our counter abstraction exploration algorithm is the representation used for system states. In the previous section, we have argued that, for efficiency reasons, we should restrict the counters stored in a global state to the non-zero ones, representing only local states occupied by at least one thread. In practice, this means that we cannot adopt the *generic representatives* model used traditionally for counter abstraction, i.e., a vector of fixed length, where position i contains the counter associated with the ith local state. Instead, the information on what a counter counts needs to be explicitly attached to the counter. This is achieved using the following representation:

$$\tau := \langle s, \{(l_1, n_1), \ldots, (l_k, n_k)\}\rangle. \tag{1}$$

In this notation, s is a valuation of the shared variables V_s, each n_i a natural number, and each l_i is an element of the set L of (conceivable) local states. That is, l_i is a valuation of the program counter and the local variables V_l of a thread. During the exploration, we maintain the invariants (a) $i \neq j$ implies $l_i \neq l_j$, and (b) $n_i \geq 1$.

The intuition for the semantics of the representation (1) is that n_i is the number of threads in τ that reside in local state l_i; unoccupied local states are omitted ($n_i \geq 1$). The set of (l_i, n_i) pairs tracks the local state of all threads, but threads in identical local states are collapsed, and the order of the occurring local states is not interpreted. As a result, each abstract state τ represents precisely a *symmetry equivalence class* $[\tau]$ of states of the original model M:

$$[\tau] := \{(s, \ell_1, \ldots, \ell_n) : n = \sum_{i=1}^{k} n_i \wedge \forall j \in 1..k |\{i \in 1..n : \ell_i = l_j\}| = n_j\}. \tag{2}$$

Conversely, each symmetry equivalence class can be represented precisely in the form (1).

In practice, an implementation of the set $\{(l_1, n_1), \ldots, (l_k, n_k)\}$ will enforce the "unorderedness" of the pairs using a canonization routine. The need for canonization may appear peculiar, since the very purpose of counter abstraction is to remove the requirement to canonize states. To understand what is going on, let us view the canonization as a two-stage process: starting from a conventional vector of ordered local states, counter abstraction first collapses all threads with the same local state into a single one, and attaches a counter to it. The remaining local state/counter pairs must then still be explicitly canonized. In classical counter abstraction, this happens by storing the counter for local state number i into position i of the counter vector; the local state part of the pair can be dropped. In our approach, we use the same total order, but omit all zero-valued counters and condense the resulting vector; we thus have to retain the local state part of each pair.

5.2 Explicit State Space Exploration

Algorithm 1 Explicit-state counter abstraction

```
1:  $\mathcal{R} := \{\langle s_0, (l_0, n_0)\rangle\}$; insert $\langle s_0, (l_0, n_0)\rangle$ into $\mathcal{W}$        ▷ $n_0$ threads at local state $l_0$
2:  while $\mathcal{W} \neq \emptyset$ do
3:      remove $\tau = \langle s, F\rangle$, with $F = \{(l_1, n_1), \ldots, (l_k, n_k)\}$, from $\mathcal{W}$
4:      for $i \in \{1, \ldots, k\}$ do
5:          $t := (s, l_i)$
6:          $t' := (s', l') := \mathsf{Image}_{\mathbb{P}}(t)$        ▷ compute successor by executing $\mathbb{P}$
7:          $\tau' := \langle s', \textsc{UpdateCounters}(F, i, l')\rangle$        ▷ build new system state
8:          if $\tau' \notin \mathcal{R}$ then
9:              $\mathcal{R} := \mathcal{R} \cup \{\tau'\}$        ▷ store $\tau'$ as reachable state
10:             insert $\tau'$ into $\mathcal{W}$

11: procedure $\textsc{UpdateCounters}(F, i, l')$
12:     let $(l_i, n_i)$ be the $i$th pair in $F$
13:     $F' := F \setminus \{(l_i, n_i)\} \cup (n_i > 1 ? \{(l_i, n_i - 1)\} : \emptyset)$        ▷ update or eliminate pair $(l_i, n_i)$
14:     if $\exists j : (l', n_j) \in F$ then        ▷ update or add pair for $l'$
15:         $F' := F' \setminus \{(l', n_j)\} \cup \{(l', n_j + 1)\}$
16:     else
17:         $F' := F' \cup \{(l', 1)\}$
18:     return $F'$
```

Algorithm 1 expands unexplored system states from a worklist $\mathcal{W}$, initialized to contain the concrete state in which all threads reside in location 0, and all variables are initialized to 0. The loop in line 4 iterates over all pairs (l_i, n_i) contained in the popped state τ. To expand an individual pair, the algorithm first projects it to the ith thread state.

Recall that local state l_i contains the program counter of thread i. Program $\mathbb{P}$ is executed at that location, on thread state (s, l_i), to obtain the successor thread state t' (line 6). Notice that, since this expansion is performed only once per pair, irrespective of the value of n_i, this step corresponds to picking one representative thread for each local state l_i for expansion.

After the successor state has been computed, the algorithm constructs the respective system state for it (line 7). The $\textsc{UpdateCounters}$ function uses the local state part l' of the newly computed thread state to determine the new set F' of local state/counter pairs. If no more threads reside in the departed state l_i, the ith pair is eliminated (line 13). If the new local state l' was already present in the system state, its counter n_j is incremented, otherwise the state is inserted with counter value 1 (lines 14–17).

Finally, the algorithm adds any system states encountered for the first time to the set of reachable states, and to the worklist of states to expand (lines 8–10).

6 Symbolic counter abstraction

In this section, we present a symbolic algorithm for state space exploration of concurrent Boolean programs that achieves scalability through counter abstraction. This algorithm builds upon the explicit-state version presented in the previous section; it receives the same input and returns the counter-abstracted set of reachable states. We focus in the following on the differences of the two algorithms.

6.1 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 $\star$. An explicit-state representation of such states, that requires expansion of all possible values an expression involving $\star$ can stand for, is infeasible in practice. A better solution is to interpret the value $\star$ symbolically, as the set $\{0, 1\}$. Our approach to symbolic counter abstraction is therefore to count *sets* of local states, rather than individual ones. For example, consider a Boolean program with a single thread-local Boolean variable x. A counter value of 3 for the local state set $\mathbb{B} := \{0, 1\}$, containing valuations of x, represents three threads whose value for x is nondeterministic.

To this end, we change the representation of (1) into a symbolic version, by storing sets S and L_i of shared and local states, instead of individual shared and local states, as follows:

$$\tau := \langle S, \{(L_1, n_1), \ldots, (L_k, n_k)\}\rangle\,. \tag{3}$$

In the new notation, S is a set of valuations over V_s, and $L_i \subseteq L$ for $1 \le i \le k$. As an example (without shared variables), the abstract global state $\langle (x = 0, 3)\rangle$ represents all concrete states with 3 threads satisfying $x = 0$. Suppose now one thread executes the statement $x := \star$. This causes the global state to become $\langle (x = 0, 2), (x = \star, 1)\rangle$, indicating that x is nondeterministic for one of the threads. Since the symbolic local state $x = 0$ is subsumed by the symbolic local state $x = \star$, one might be tempted to "merge" the former into the latter, resulting in the global state $\langle (x = \star, 3)\rangle$. This is incorrect, however, as the global state after merging allows, for example, three threads to satisfy $x = 1$, while the global state before merging does not.

The formal semantics of this representation is given by the set of concrete states represented by expression (3), namely the states of the form $(s, l_1, \ldots, l_n)$ such that

$$\begin{array}{ll} \text{(a)} & s \in S, \\ \text{(b)} & n = \sum_{i=1}^{k} n_i, \text{ and} \\ \text{(c)} & \text{there exists a partition } \{I_1, \ldots, I_k\} \text{ of } \{1, \ldots, n\} \text{ such that} \\ & \text{for all } i \in \{1, \ldots, k\},\ |I_i| = n_i \text{, and for all } j \in I_i,\ l_j \in L_i. \end{array} \tag{4}$$

That is, an abstract state of the form (3) represents precisely the concrete states in the Cartesian product of valuations of the shared variables in S, and valuations of the thread-local variables satisfying the constraint (4) (c).

The semantics given in (4) defines a left-total function mapping abstract states of the form (3) to sets of concrete states of the form $(s, l_1, \ldots, l_n)$. It is key to notice that, differently to the explicit-state case, this function is now not right-total: there are sets of concrete states that are not representable by a single abstract state. Consider the set $\Delta := \{(0,0), (1,1)\}$ of concrete states (s,l), for a single thread, a shared variable s and a thread-local variable l. This set arises when, starting from state $(\star, \star)$, the thread executes the statement $s := l$. This statement introduces a constraint across the shared and thread-local variables. There is no single abstract state whose semantics (4) equals Δ. We therefore need to split the set Δ into the subsets $\{(0,0)\}$ and $\{(1,1)\}$, both of which are trivially representable by an abstract state. In this simple case, therefore, the abstraction provides no compression over the concrete domain. Section 6.2 explains why this does not materialize as a problem in practice. We substantiate this claim later in the experimental evaluation in Sect. 7. Section 7.4 presents alternative symbolic global state representations, together with a justification as to why they are inadequate in implementations of counter abstraction.

6.2 Symbolic state space exploration

Algorithm 2 performs reachability analysis of symmetric Boolean programs based on the symbolic state representation of (3). The exploration procedure follows that of Algorithm 1. Generally, Algorithm 2 operates on sets of (global, local, thread) states. For example, line 5 extracts from τ the set $T := (S, L_i) = \{(s, l_i) : s \in S \wedge l_i \in L_i\}$.

Algorithm 2 Symbolic counter abstraction

```
1: $\mathcal{R} := \{\langle S_0, (L_0, n_0)\rangle\}$; insert $\langle S_0, (L_0, n_0)\rangle$ into $\mathcal{W}$    ▷ $n_0$ threads with local state in $L_0$
2: while $\mathcal{W} \neq \emptyset$ do
3:     remove $\tau = \langle S, F\rangle$, with $F = \{(L_1, n_1), \ldots, (L_k, n_k)\}$, from $\mathcal{W}$
4:     for $i \in \{1, \ldots, k\}$ do
5:         $T := (S, L_i)$
6:         for $v \in$ valuations of $SpliceVariables(T)$ do
7:             $T' := (S', L') := \mathsf{Image}(T|_v)$    ▷ compute one image cofactor
8:             $\tau' := \langle S', \textsc{UpdateCounters}(F, i, L')\rangle$    ▷ build new system state
9:             if $\tau' \notin \mathcal{R}$ then
10:                $\mathcal{R} := \mathcal{R} \cup \{\tau'\}$    ▷ store $\tau'$ as reachable
11:                insert $\tau'$ into $\mathcal{W}$

12: procedure UpdateCounters($F, i, L'$)
13:     let $(L_i, n_i)$ be the $i$th pair in $F$
14:     $F' := F \setminus \{(L_i, n_i)\} \cup (n_i > 1 ? \{(L_i, n_i - 1)\} : \emptyset)$    ▷ update or eliminate pair $(L_i, n_i)$
15:     if $\exists j : (L', n_j) \in F$ then    ▷ update or add pair for $L'$
16:         $F' := F' \setminus \{(L', n_j)\} \cup \{(L', n_j + 1)\}$
17:     else
18:         $F' := F' \cup \{(L', 1)\}$
19:     return $F'$
```

The step of computing the successor thread states induced by the Boolean program (lines 6–7) requires special care in the symbolic case. Recall from Sect. 6.1 that constraints between shared and thread-local variables cannot be encoded in a single abstract state. The first step in dealing with this problem is to recognize statements that may introduce such constraints.

Definition 3 A *splice state* is a symbolic thread state given as a predicate f over the variables in $V_s \cup V_l \cup \{pc\}$ such that

$$(\exists V_s . f) \wedge (\exists V_l \exists pc . f) \not\equiv f.$$

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

A splice statement marks a point where a thread communicates data via the shared 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 shared variable and the right-hand side expression refers to thread-local variables, or vice versa, such as $s := l$ or $l := s$, and
– constraint assignments with a `constrain` clause whose expression refers to both shared and thread-local variables, such as $x := y$ `constrain` $s = l$.

The second case also covers statements of the form `assume` $s = l$.

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 (3). 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 observe in our experiments (see Sect. 7),

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 are actually *unsatisfiable* and do not contribute new states.

As a result, the potential combinatorial explosion does not materialize in our experiments.

After the image has been computed for each cofactor, the algorithm constructs the corresponding system state for it by means of the UPDATECOUNTERS function and adds it to the set of reachable states and the worklist in the usual way.

Theorem 4 *Let $\mathcal{R}$ be as computed by Algorithm 2 on termination, and let γ be the concretization function for abstract states defined in (4). The set $\gamma(\mathcal{R}) = \{\gamma(r) | r \in \mathcal{R}\}$ is the set of reachable states of the concurrent system induced by the Boolean program $\mathbb{P}$.*

Proof [sketch]: The proof of termination of Algorithm 2 follows since the explored state space is finite: the bound N on the number of threads that may be created, and of which the `start_thread` command is aware, ensures that there is only a finite number of global states, whether they are represented in the concrete or abstract form. Algorithm 2 performs a standard search over this finite state space and thus terminates. The correctness argument of the algorithm follows 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. □

6.3 Error detection and counterexample generation

Errors are program locations containing violated assertions, say of the form `assert`(Y). The predicate Y expresses a condition, over the current thread state, that is claimed to be an invariant. The violation of this invariance condition is checked in Algorithm 2 in line 7, by testing the new thread state (S', L') against the condition $\neg Y$: if the BDD for $S' \wedge L' \wedge \neg Y$ is non-empty, T' violates the assertion Y.

In order to embed our exploration method into the CEGAR loop, we now need to obtain a concrete path from an initial global state to a global state that contains thread state T'. Such a path can be presented in the form of a sequence of global states over Boolean program variables, including the PC. We have omitted a description of the standard back edges and the mechanisms to trace back a reached state to the initial state. Non-standard is the shape of the resulting path, namely a sequence U of states of the form $\langle S, (L_1, n_1), \ldots, (L_k, n_k)\rangle$, ending in τ' (from line 8). This abstract path can be mapped to a concrete path over the concurrent Boolean program using (4) and standard techniques in symmetry reduction.

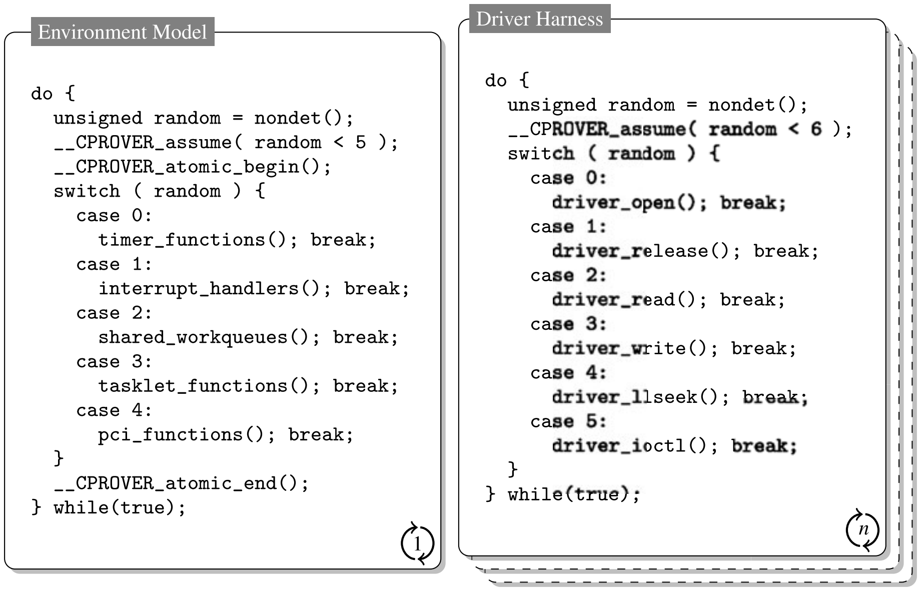
7 Experimental evaluation

We have implemented the algorithm presented in this paper in a tool called BOOM. While our main goal is the symbolic analysis of Boolean programs (Sect. 7.1), we have also built an *explicit-state* version of our method. The reason is that symmetry reduction has so far proven to be more successful in explicit-state than symbolic Model Checking, which begs a comparison against explicit-state symmetry reduction. As a competitor, we chose the well-known MURφ Model Checker [30] (Sect. 7.2).

We applied BOOM to 444 examples from two sources: a set of 208 Boolean programs generated by SATABS that abstract part of the Linux kernel components, and a set of 236 Boolean programs generated at Microsoft Research using SLAM. Both the tool and our benchmark set are available on our website at http://www.cprover.org/boom.

Before we discuss our experiments, we describe in detail how we obtained *concurrent* benchmarks from the Boolean program source. This step differs for the SLAM-generated programs and those generated with SATABS. For SLAM, we simply instantiate a sequential Boolean program once per thread; each thread executes the program's `main` procedure. Variables with global scope become shared variables of the concurrent programs, while variables with local scope become thread-local variables. Note that this does not restrict the computational model. Synchronization primitives such as locks and semaphores can be simulated using shared variables (e.g. `lock`$(s) \triangleq s := \mathit{true}$ `constrain` $!s$).

In contrast, the concurrent benchmarks produced by SATABS were generated using DDVERIFY [37], a harness for Linux device drivers. The resulting concurrent model supports both synchronization primitives (such as semaphores and spinlocks) and memory-mapped IO-registers for communication with the underlying hardware. Figure 3 illustrates how parallel execution is handled. An environment thread models the operating system



```
do {
  unsigned random = nondet();
  __CPROVER_assume( random < 5 );
  __CPROVER_atomic_begin();
  switch ( random ) {
    case 0:
      timer_functions(); break;
    case 1:
      interrupt_handlers(); break;
    case 2:
      shared_workqueues(); break;
    case 3:
      tasklet_functions(); break;
    case 4:
      pci_functions(); break;
  }
  __CPROVER_atomic_end();
} while(true);
```

```
do {
  unsigned random = nondet();
  __CPROVER_assume( random < 6 );
  switch ( random ) {
    case 0:
      driver_open(); break;
    case 1:
      driver_release(); break;
    case 2:
      driver_read(); break;
    case 3:
      driver_write(); break;
    case 4:
      driver_llseek(); break;
    case 5:
      driver_ioctl(); break;
  }
} while(true);
```

Fig. 3 Concurrent DDVERIFY execution model (`'ddverify -model con2'`)

threads and parallelism caused by hardware events, e.g., interrupts. These functions are non-preemptive since interrupt service routines cannot be switched out during execution by the operating system kernel. The interaction between the driver and a client application is simulated in an infinite loop that nondeterministically calls the driver's functions. This loop is executed by multiple threads, since access to a driver can be shared among clients.

The Boolean program that results from the predicate abstraction process exhibits the same control structure as the original code. In order to simplify our experiments, we create only one client thread in the original program and instantiate N copies of this thread in the Model Checker (the DDVERIFY harness does not exploit the possibility of dynamic thread creation). We remark that our method does allow different threads to execute different procedures. Because we do not allow recursion, all procedures can be merged together into one big procedure that is passed to the Model Checker. There is no message-passing communication between the threads, but they access the same shared data structures within the driver's code. The benchmarks feature, on average, 123 program locations, 21 thread-local variables, and 12 shared variables.

The experimental setup is as follows. For each tool and benchmark, we run full reachability analysis with $n_0 = 1$ initial threads and a bound of $N = 2$ on thread creation. We then increase the bound N until the tool times out. The timeout is set to 720 s and the memory limit to 12 GB. The experiments are performed on a 3 GHz Intel Xeon machine running the 64-bit variant of Linux 2.6.

7.1 Symbolic experiments

Since other symbolic Model Checkers did not scale to interesting thread counts (including the few tools with built-in support for symmetry, see Sect. 2), we compare the symbolic algorithm to a "plain" symbolic reference implementation in BOOM that ignores the sym-

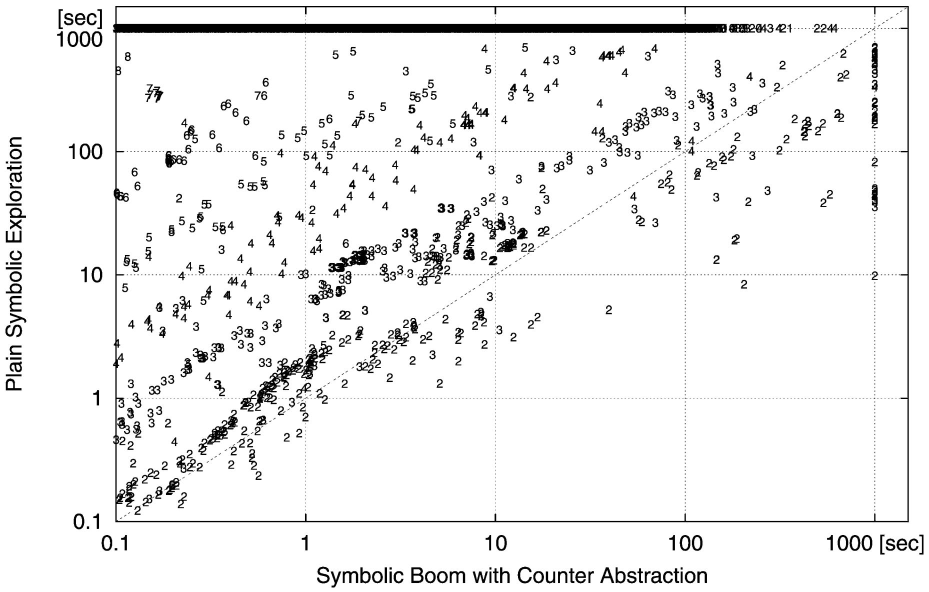


Fig. 4 Running time of symbolic BOOM vs. plain exploration, for various thread counts

metry. On sequential programs, the performance of the reference implementation is similar to that of the Model Checker that ships with SLAM.

BOOM uses the CUDD BDD library by Fabio Somenzi ([33, version 2.4.1]) as the decision diagram package. Our implementation stores the sets S and L_i of shared and thread-local variable valuations as separate BDDs; the conjunction of S and 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.

Figure 4 summarizes the running times of the symbolic counter abstraction implementation in BOOM and the plain symbolic exploration algorithm. The concentration of entries in the upper triangle signifies the improvement in scalability due to counter abstraction. Those runs where traditional Model Checking is faster contain a small number of threads; in fact, our algorithm can verify many instances for 7 or more threads. Overall, BOOM is faster on 83% of all tests, and on 96% of those running three or more threads. Among those, the speed-up is five orders of magnitude and more.

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

State merging Our implementation of the last step of Algorithm 2 (lines 20–21) uses state merging, an important optimization to compress sets of symbolic states. Two distinct symbolic states can be merged if they are identical except for *either* (i) the valuation of the shared variables, *or* (ii) the local state of exactly *one* thread. To illustrate the second condition, suppose the states $\langle(\{B\}, 2)\rangle$ and $\langle(\{C\}, 2)\rangle$ (no shared variables) are encountered in the worklist. Merging them into the state $\langle(\{B, C\}, 2)\rangle$ is incorrect: the merged state illegally represents, e.g., the concrete state (B, C), which is not a represented by either of the original two abstract states. The application of our merging rules provided an average speed-up of 83% over exploration without merging.

7.2 Explicit-state experiments

We compare our explicit-state implementation to MURφ [30], a mature and popular Model Checker with long-standing support for symmetry reduction, using the benchmarks described above. Since MURφ does not allow data nondeterminism, we first tried to simulate this feature using two rules per occurrence of a ⋆ symbol in a statement: one rule for each of the two data values. This leads to some blow-up in the program text, especially if there are several occurrences of ⋆ in a statement. More critically, however, this lead to an enormous blow-up in the number of states being explored, rendering meaningful experimentation impossible. This once again confirmed the need for a symbolic analysis of programs with data nondeterminism.

In order to nevertheless be able to compare counter abstraction to the symmetry implementation in MURφ, in a syntactic preprocessing step we replace every occurrence of ⋆ in the input programs randomly by 0 or 1. The resulting programs are converted into MURφ's input language using one MURφ rule per statement, guarded by the program counter value. In the explicit-state experiments, we compare the performance of explicit-state BOOM on each determinized Boolean program against MURφ on the guarded-rule version of the same program.

Figure 5 is a scatter plot of the running times of BOOM and of MURφ with symmetry reduction. BOOM 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.

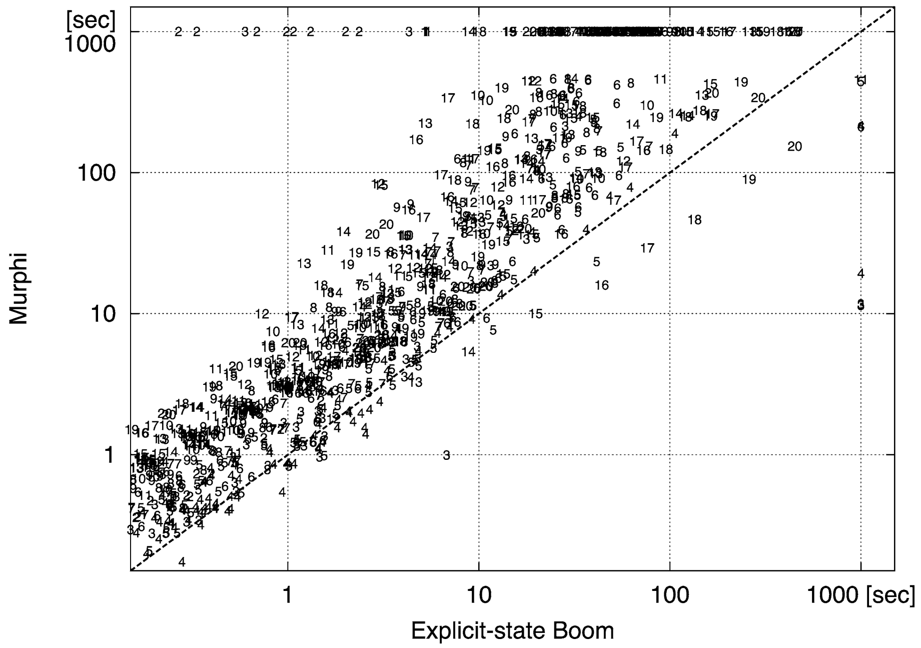


Fig. 5 Running time of explicit-state BOOM vs. MURφ, for various thread counts

Note that removing the data nondeterminism simplifies the programs, which is why the explicit-state explorations can often handle larger thread counts than the symbolic ones, reported in the previous subsection.

7.3 Comparison with partial-order methods

In contrast to symmetry reduction, which is based on the interchangeability of states, *partial-order reduction* (POR) exploits redundancy of program traces that are identical up to different interleavings of threads [25]. The idea is to identify *commutable* transitions, and pick only a representative schedule of those transitions. Deciding what constitutes a representative schedule, however, may be as hard as the Model Checking problem itself. Instead, static analysis techniques can be used to approximate the dependencies between transitions, resulting in suboptimal but affordable POR methods.

We have implemented a POR strategy in our Model Checker. Our approach is to look for a thread making an *invisible* transition: one that is independent of *any possible future transitions* made by *any other thread*. Contrast this to the work by Cook et al. [10], which considers only finite futures to identify collisions of transitions.

Our strategy is based on the set W_t of variables written by thread t, and the set R_t of variables read by t, in the current state. Analogously, let R_t^∞ and W_t^∞ denote the set of variables read or written at some time in the future (present included). As common in static analysis, these sets are computed using data-flow equations based on conservative assumptions on what constitutes a read or write. In particular, the set W_t contains all variables that appear in an instruction that could restrict the state space. Such an instruction may disable some instructions of other threads and must thus be considered a write. Specific to Boolean programs, the `assume` statement and constraint assignments belong to this category; see Example 1 below.

We use these sets in our POR algorithm as follows. If a thread t is found satisfying $W_t \cap (\bigcup_{i \neq t} R_i^\infty \cup W_i^\infty) = \emptyset$ (no variable written by t is ever used by another thread) and $R_t \cap \bigcup_{i \neq t} W_i^\infty = \emptyset$ (no variable read by t is ever written by another thread), we only explore the successors generated by executing t, but not by any other thread. Intuitively, t does not communicate with other threads during this transition. All other transitions are discarded at the current state. We illustrate this technique with a few examples.

Example 1 Consider the Boolean program in Listing 1. Suppose the `assume` instruction counts as a pure *read* access to the variables in its expression argument (only s in this case). Then, after the second thread has been created, both threads are about to execute invisible statements only. If POR picks the thread that continues execution at `P1` and runs it until the end, the second thread cannot reach the `assert` instruction anymore, and the path that violates the property will remain undiscovered.

Example 2 Consider the Boolean program in Listing 2, and a state where the threads are at location `P1` and `P2`, respectively. One thread is about to execute the invisible statement `l := T`. The other thread is reading from `s` and thus does not interfere with the instruction of the first thread. One might therefore be tempted to regard the `assert` statement as invisible, and omit other possible interleavings at this point. The consequence would again be that the violation of the assertion in the other thread is goes undetected.

The effect of our POR on the plain symbolic algorithm (*without* using counter abstraction) is an average speedup of 4.2 and combined runtime improvements of 85% for all

Listing 1: An `assume` stmt. disabling other threads

```
decl s;

void main() begin
     s := ⋆;

     start_thread P2;

P1:  assume(s);
     end_thread;

P2:  assume(!s);
     assert(F);

end
```

Listing 2: Effect of future transitions on POR

```
decl s;

void main() begin
     decl l;

     s := T;
     start_thread P2;

P1:  l := T;
     s := F;
     end_thread;

P2:  assert(s);
end
```

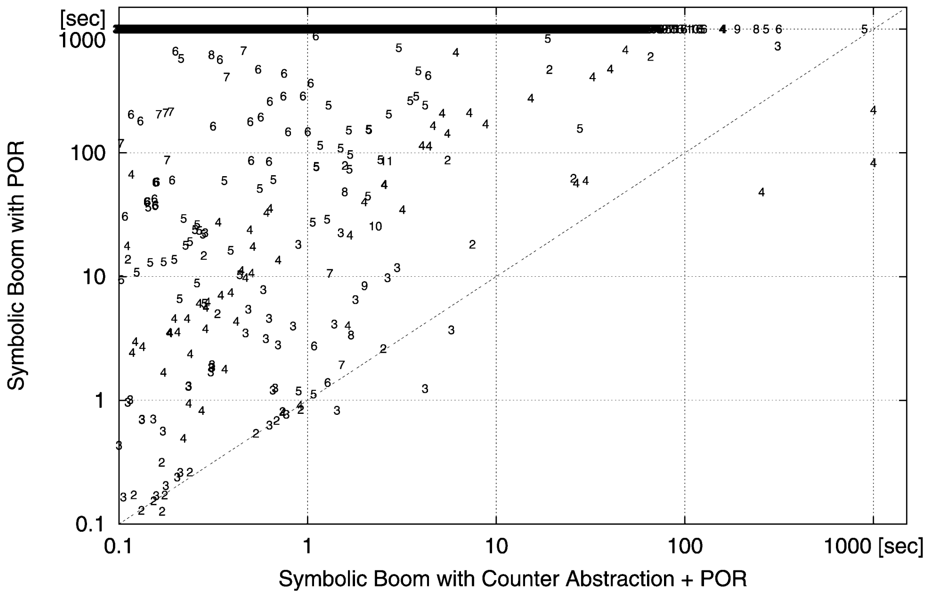


Fig. 6 Running time of symbolic BOOM with partial-order reduction vs. symbolic BOOM with counter abstraction and partial-order reduction, for various thread counts

benchmarks. We now compare the POR method sketched above to counter abstraction in various ways. We use the same sets of benchmarks and timing constraints as before.

Figure 6 shows the additional improvement of counter abstraction on an implementation that is solely based on POR. The combination of both techniques is on average 156 times faster than POR alone. This indicates that counter abstraction is by no means "subsumed" by POR.

The scatterplot in Fig. 7 depicts the speedup of employing POR and counter abstraction versus counter abstraction alone. In total, an average speedup of 140 could be measured. This indicates conversely that POR is not subsumed by counter abstraction either. Figures 6 and 7 witness once again the often observed complementariness of symmetry based and partial-order based methods (see, for instance, the work by Emerson et al. [20]).

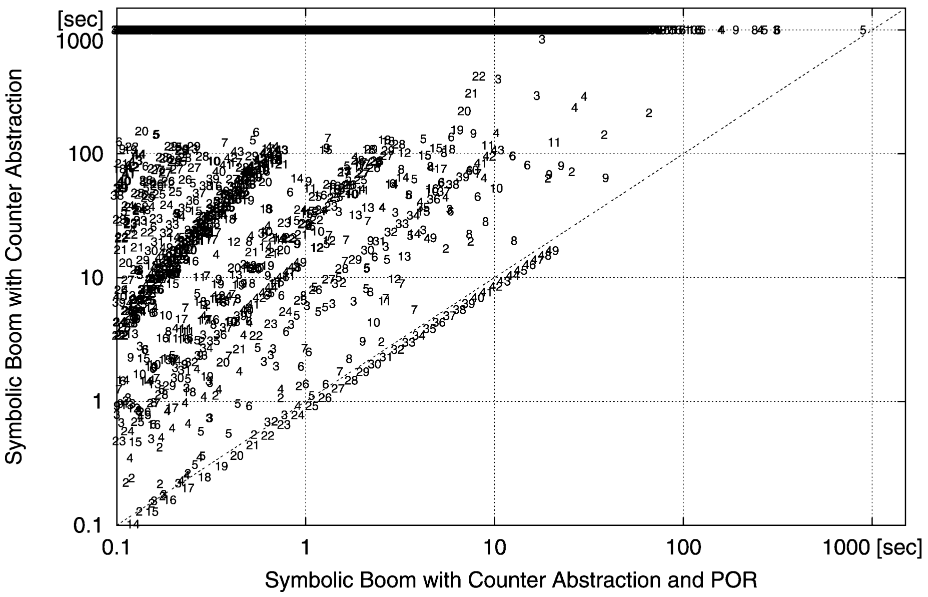


Fig. 7 Running time of symbolic BOOM with counter abstraction vs. symbolic BOOM with counter abstraction and partial-order reduction, for various thread counts

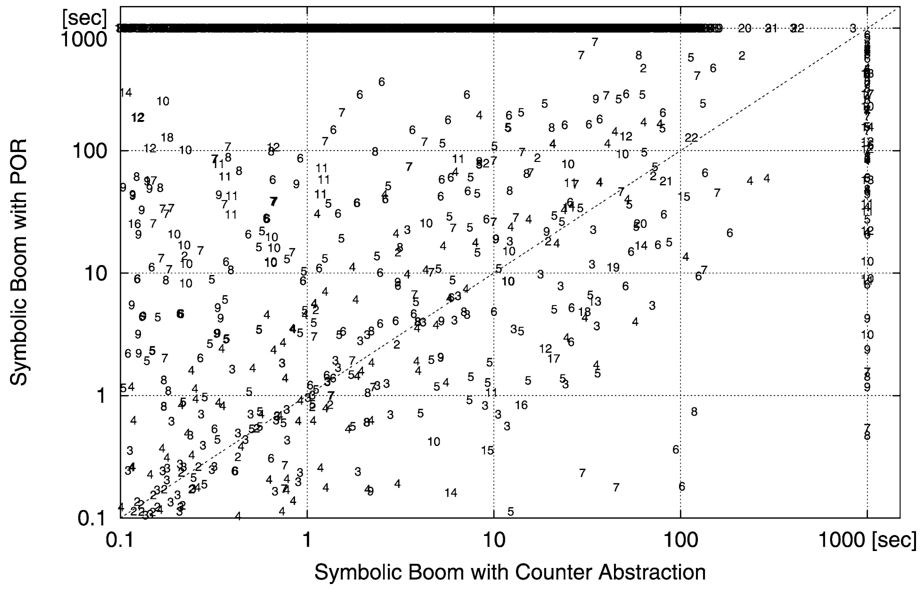


Fig. 8 Running time of symbolic BOOM with partial-order reduction vs. symbolic BOOM with counter abstraction, for various thread counts

Finally, Fig. 8 compares the methods directly against each other: counter abstraction alone outperforms partial-order reduction alone; the former gave an average speedup of 4.2.

7.4 Summary of the experimental evaluation

We have conducted extensive experiments to compare our variant of counter abstraction to alternative means of state space exploration, to traditional symmetry reduction, and to partial order reduction. Since the reachability problem for finite state spaces is decidable, progress in verification technology is bound to come from improvements in Model Checker performance, rather than theoretical advances. Our goal here was therefore to estimate the performance benefit of counter abstraction when exhaustively exploring system models.

We summarize our findings as follows. Counter abstraction provides tremendous benefits in both explicit-state and symbolic implementations of state space exploration, albeit for different reasons. In the former case, the technique is easy to implement, since the problem of splice statements does not arise. In fact, counting has long been known as one way to canonize local state vectors under full symmetry. Our implementation provides the added benefit that states are in fact stored in counter form; the latter is not only used for canonization purposes.

For symbolic implementations, the story is different. In serious concurrent software verification using predicate abstraction, symbolic algorithms are essential. Combining this with the (obvious) need for symmetry reduction has resulted in a mostly unsolved challenge, due to the orbit problem. Achieving an equivalent effect using counter abstraction is much more involved than in the explicit-state case. Our paper fills this gap by pinning down the details of how an efficient implementation can be realized. A crucial point is the balance between our state representation with deliberately limited expressiveness (only benign concrete state sets can be effectively compressed), and the occasional need to respond to this problem during the exploration, by splitting state sets.

We have also considered global state representations other than (3). In one implementation, we use a *monolithic* BDD to represent the shared variables and all thread states, along with their counters. In another, we keep the counters explicit, but use a monolithic BDD for all other variables. Both implementations allow us to retain the inter-thread constraints introduced by splice statements, and thus render the decomposition step unnecessary. The first implementation has the additional advantage of not requiring state merging techniques (Sect. 7.1): given a single BDD, merging happens automatically when adding new frontier states to the BDD.

A technical challenge with these alternative representations is that they require more complex manipulations for computing successor states, especially in order to update the counters. The more severe downside, however, is efficiency, as is often the case with monolithic symbolic data structures: the resulting BDD for the set of reachable states is complex, foiling the scalability advantage inherent in counter abstraction. In fact, the separation of a global state into thread states and associated counters suggests a natural way of partitioning the BDD for the reachable states set, which should not be given up lightly. On our benchmarks, the algorithm proposed in Sect. 6 is at least 30% faster than all alternatives. We finally remark that the idea to decompose state sets while abstracting away some correlations between the states has also been employed in shape analysis, where the decomposition is suggested by the heap structure [29].

8 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 resorting 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 have shown how to avoid the *local state space explosion* problem using a combination of two techniques: (1) achieving context-awareness by 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 presented experimental results both for an explicit-state and, more importantly, a symbolic implementation. While standard symmetry reduction is employed in tools like MURφ, we are not aware of a prior implementation of counter abstraction that is efficient on programs other than abstract protocols with very 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.

We have also investigated in detail the relationship between our implementation of counter abstraction and partial-order methods. Our experiments seem to confirm the folk wisdom that symmetry and partial-order reduction are, although not independent, certainly complementary and can be combined for yet more effective compression.

Symmetry reduction, no matter of what flavor, is limited in scope in that it considers only systems of identically replicated, concurrent components of a number that is a design-time constant. We have, in this work, extended the technique to software with bounded *dynamic* thread creation. An obvious direction for future work is to push the limits further by considering unbounded dynamic thread creation, or the parameterized version of the concurrent reachability problem.

Acknowledgements The authors are grateful to Alastair Donaldson for his rigorous review of this paper.

References

1. Ball T, Rajamani SK (2000) Bebop: a symbolic model checker for Boolean programs. In: SPIN, pp 113–130
2. Ball T, Rajamani SK (2002) The SLAM project: debugging system software via static analysis. In: POPL, pp 1–3
3. Ball T, Chaki S, Rajamani SK (2001) Parameterized verification of multithreaded software libraries. In: TACAS, pp 158–173
4. Ball T, Bounimova E, Cook B, Levin V, Lichtenberg J, McGarvey C, Ondrusek B, Rajamani SK, Ustuner A (2006) Thorough static analysis of device drivers. In: EuroSys, pp 73–85
5. Barner S, Grumberg O (2005) Combining symmetry reduction and under-approximation for symbolic Model Checking. Form Methods Syst Des 27(1–2):29–66
6. Basler G, Mazzucchi M, Wahl T, Kroening D (2009) Symbolic counter abstraction for concurrent software. In: CAV, pp 64–78
7. Bosnacki D, Dams D, Holenderski L (2002) Symmetric spin. Int J Softw Tools Technol Transf 4(1):92–106
8. Clarke EM, Jha S, Enders R, Filkorn T (1996) Exploiting symmetry in temporal logic Model Checking. Form Methods Syst Des 9(1/2):77–104
9. Clarke EM, Kroening D, Sharygina N, Yorav K (2005) SATABS: SAT-based predicate abstraction for ANSI-C. In: TACAS, pp 570–574
10. Cook B, Kroening D, Sharygina N (2005) Symbolic Model Checking for asynchronous Boolean programs. In: SPIN, pp 75–90
11. Cook B, Kroening D, Sharygina N (2007) Verification of Boolean programs with unbounded thread creation. Theor Comput Sci 388(13):227–242
12. Delzanno G (2000) Automatic verification of parameterized cache coherence protocols. In: CAV, pp 53–68

13. Donaldson AF, Miller A (2006) Exact and approximate strategies for symmetry reduction in model checking. In: FM, pp 541–556
14. Donaldson AF, Miller A (2006) Symmetry reduction for probabilistic Model Checking using generic representatives. In: ATVA, pp 9–23
15. Donaldson A, Miller A, Parker D (2009) Language-level symmetry reduction for probabilistic Model Checking. In: QEST, pp 289–298
16. Emerson EA, Sistla AP (1996) Symmetry and Model Checking. Form Methods Syst Des 9(1/2):105–131
17. Emerson EA, Trefler RJ (1999) From asymmetry to full symmetry: new techniques for symmetry reduction in Model Checking. In: CHARME, pp 142–156
18. Emerson EA, Wahl T (2005) Dynamic symmetry reduction. In: TACAS, pp 382–396
19. Emerson EA, Wahl T (2005) Efficient reduction techniques for systems with many components. Electron Notes Theor Comput Sci 130:379–399
20. Emerson EA, Jha S, Peled D (1997) Combining partial order and symmetry reductions. In: TACAS, pp 19–34
21. Emerson EA, Havlicek J, Trefler RJ (2000) Virtual symmetry reduction. In: LICS, pp 121–131
22. Flanagan C, Godefroid P (2005) Dynamic partial-order reduction for Model Checking software. In: POPL, pp 110–121
23. Graf S, Saïdi H (1997) Construction of abstract state graphs with PVS. In: CAV, pp 72–83
24. Henzinger TA, Jhala R, Majumdar R (2004) Race checking by context inference. In: PLDI, pp 1–13
25. Holzmann GJ, Peled D (1994) An improvement in formal verification. In: FORTE, pp 197–211
26. Kurshan RP (1994) Computer-aided verification of coordinating processes: the automata-theoretic approach. Princeton University Press, Princeton
27. Lahiri SK, Bryant RE, Cook B (2003) A symbolic approach to predicate abstraction. In: CAV, pp 141–153
28. Lubachevsky BD (1984) An approach to automating the verification of compact parallel coordination programs I. Acta Inform 21:125–169
29. Manevich R, Lev-Ami T, Sagiv M, Ramalingam G, Berdine J (2008) Heap decomposition for concurrent shape analysis. In: SAS, pp 363–377
30. Melton R, Dill D Murϕ annotated reference manual, rel. 3.1. http://verify.stanford.edu/dill/murphi.html
31. Pnueli A, Xu J, Zuck LD (2002) Liveness with $(0, 1, \infty)$-counter abstraction. In: CAV, pp 107–122
32. Pong F, Dubois M (1995) A new approach for the verification of cache coherence protocols. IEEE Trans Parallel Distrib Syst 6(8):773–787
33. Somenzi F The CU decision diagram package, release 2.3.1. University of Colorado at Boulder. http://vlsi.colorado.edu/~fabio/CUDD/
34. Suwimonteerabuth D, Esparza J, Schwoon S (2008) Symbolic context-bounded analysis of multithreaded java programs. In: SPIN, pp 270–287
35. Wahl T, Blanc N, Emerson EA (2008) SVISS: symbolic verification of symmetric systems. In: TACAS, pp 459–462
36. Wei O, Gurfinkel A, Chechik M (2005) Identification and counter abstraction for full virtual symmetry. In: CHARME, pp 285–300
37. Witkowski T, Blanc N, Kroening D, Weissenbacher G (2007) Model Checking concurrent Linux device drivers. In: ASE, pp 501–504