Efficient Modeling of Concurrent Systems in BMC

Malay K. Ganai and Aarti Gupta
NEC Labs America, Princeton, NJ, USA

Abstract. We present an efficient method for modeling multi-threaded concurrent systems with shared variables and locks in Bounded Model Checking (BMC), and use it to improve the detection of safety properties such as data races. Previous approaches based on synchronous modeling of interleaving semantics do not scale up well due to the inherent asynchronism in those models. Instead, in our approach, we first create independent (uncoupled) models for each individual thread in the system, then explicitly add additional synchronization variables and constraints, incrementally, and only where such synchronization is needed to guarantee the (chosen) concurrency semantics (based on sequential consistency). We describe our modeling in detail and report verification results to demonstrate the efficacy of our approach on a complex case study.

1 Introduction

The growth of cheap and ubiquitous multi-processor systems and concurrent library support are making concurrency programming very attractive. On the other hand, verification of concurrent systems remains a daunting task especially due to complex and unexpected interactions between asynchronous threads, and various architecture-specific memory consistency models [1]. In this work, we focus on concurrency semantics based on sequential consistency [2]. In this semantics, the observer has a view of only the local history of the individual threads where the operations respect the program order. Further, all the memory operations exhibit a common *total order* that respect the *program order* and has the *read value property*, i.e., the read of a variable returns the last write on the same variable in that total order. In the presence of synchronization primitives such as locks/unlocks, the concurrency semantics also respects the mutual exclusion of operations that are guarded by matching locks. Sequential consistency is the most commonly used concurrency semantics for software development due to ease of programming, especially to obtain race-free, i.e, correctly synchronized threads. A *data race* corresponds to a global state where two different threads access the same shared variable, and at least one of them is a write.

Bounded Model Checking (BMC) [3] has been successfully applied to verify real-world designs. Strengths of BMC are manifold: First, expensive quantification used in symbolic model checking [4] is avoided. Second, reachable states are not stored, avoiding blow-up of intermediate state representation. Third, modern SAT solvers are able to search through the relevant paths of the problem even though the paths get longer with the each BMC unrolling. We focus on verifying concurrent systems through efficient modeling in BMC.

1.1 Related Work

We discuss various model checking efforts, both explicit and symbolic, for verifying concurrent systems with shared memory. The general problem of verifying a concurrent system with even two threads with unbounded stacks is undecidable [5]. In practice, these verification efforts use *incomplete* methods, or *imprecise* models, or sometimes

both, to address the scalability of the problem. The verification model is typically obtained by composing individual thread models using interleaving semantics, and model checkers are applied to systematically explore the global state space. Model checkers such as Verisoft [6], Zing [7] explore states and transitions of the concurrent system using explicit enumeration. Although several state space reduction techniques based on partial order methods [8] and transactions-based methods [9–12] have been proposed, these techniques do not scale well due to both state explosion and explicit enumeration.

Symbolic model checkers such as BDD-based SMV [4], and SAT-based Bounded Model Checking (BMC) [3] use symbolic representation and traversal of state space, and have been shown to be effective for verifying synchronous hardware designs. There have been some efforts [13–16] to combine symbolic model checking with the above mentioned state-reduction methods for verifying concurrent software. However, they still suffer from lack of scalability. To overcome this limitation, some researchers have employed sound abstraction [7] with bounded number of context switches [17], while some others have used finite-state model [15, 18] or Boolean program abstractions with bounded depth analysis [19]. This is also combined with a bounded number of context switches known *a priori* [15] or a proof-guided method to discover them [18]. To the best of our knowledge, all these model checking methods use synchronous modeling of interleaving semantics. As we see later, our focus is to move away from such synchronous modeling in BMC in order to obtain significant reduction in the size of the BMC instances.

Another development is the growing popularity of Satisfiability-Modulo Theory (SMT)-solvers such as [20]. Due to their support for richer expressive theories beyond Boolean logic, and several latest advancements, SMT-based methods are providing more scalable alternatives than BDD-based or SAT-based methods. In SMT-based BMC, a BMC problem is translated typically into a quantifier-free formula in a decidable subset of first order logic, instead of translating it into a propositional formula, and the formula is then checked for satisfiability using an SMT solver. Specifically, with several acceleration techniques, SMT-based BMC has been shown [21] to scale better than SAT-based BMC for finding bugs.

There have been parallel efforts [22–24] to detect bugs for weaker memory models. As shown in [25], one can check these models using axiomatic memory style specifications combined with constraint solvers. Note, though these methods support various memory models, they check for bugs using given test programs. There has been no effort so far, to our knowledge, to integrate such specifications in a model checking framework that does not require test programs. There have been other efforts using static analysis [26, 27] to detect static races. Unlike these methods, our goal is to find true bugs and to not report false warnings.

1.2 Our Approach: Overview

We present an efficient modeling for multi-threaded concurrent systems with shared variables and locks in BMC. We consider C threads under the assumption of a bounded heap and bounded stack. Using this modeling, we augment SMT-based BMC to detect violations of safety properties such as data races. *The main novelty of our approach is that it provides a sound and complete modeling with respect to the considered concurrency semantics, without the expensive synchronous modeling of interleaving semantics.* Specifically, we do not introduce *wait-cycles* to model interleaving of the individual threads, and do not model a scheduler explicitly. As we see later, these wait-cycles are detrimental to the performance of BMC. Instead, we first create *independent* (decoupled) individual thread models, and add memory consistency constraints

lazily, *incrementally*, and *on-the-fly* during BMC unrolling to capture the considered concurrency semantics. Our modeling preserves with respect to a property the set of all possible executions up to a bounded depth that satisfy the sequential consistency and synchronization semantics, without requiring an *a priori* bound on the number of context switches. We have implemented our techniques in a prototype SMT-based BMC framework, and demonstrate its effectiveness through controlled experiments on a complex concurrency benchmark. For experiments, we contrast our *lazy modeling* approach with an *eager modeling* [13–16] of the concurrent system, i.e., a monolithic model synchronously composed with interleaving semantics (and possibly, with state-reduction constraints) enforced by an explicit scheduler, capturing all concurrent behaviors of the system eagerly.

Outline: We provide a short background in Section 2; motivation in Section 3; illustrate our basic approach with an example in Section 4; formal description of our modeling in Section 5; correctness theorems and discussion on size complexity in Section 6; BMC size-reduction techniques in Section 7; followed by experiments in Section 8, and conclusions in Section 9.

2 Preliminaries

2.1 Concurrent System: Model and Semantics

We consider a concurrent system comprising a finite number of deterministic bounded-stack threads communicating with shared variables, some of which are used as synchronization objects such as locks. Each thread has a finite set of control states and can be modeled as an extended finite state machine (EFSM). An EFSM model is a 5-tuple (s_0, C, I, D, T) where, s_0 is an initial state, C is a set of control states (or blocks), I is a set of inputs, D is a set of data state variables (with possibly infinite range), and T is a set of 4-tuple (c, g, u, c') transitions where $c, c' \in C$, g is a Boolean-valued enabling condition (or *guard*) on state and input variables, u is an update function on state and input variables.

We define a concurrent system model $\mathcal{CS}$ as a 4-tuple $(\mathcal{M}, \mathcal{V}, \mathcal{T}, s_0)$, where $\mathcal{M}$ denotes a finite set of EFSM models, i.e., $\mathcal{M} = \{M_1, \cdots, M_n\}$ with $M_i = (s_{0i}, C_i, I_i, D_i \cup \mathcal{V}, T_i)$, $\mathcal{V}$ denotes a finite set of shared(or global) variables i.e., $\mathcal{V} = \{g_1, \cdots, g_m\}$, $\mathcal{T}$ denotes a finite set of transitions, i.e., $\mathcal{T} = \bigcup_i T_i$, s_0 denotes the initial global state. Note, for $i \neq j$, $C_i \cap C_j = \emptyset$, $I_i \cap I_j = \emptyset$, $D_i \cap D_j = \emptyset$, and $T_i \cap T_j = \emptyset$, i.e., except for shared variables $\mathcal{V}$, each M_i is disjoint. Let VL_i denote a set of tuples values for local data state variables in D_i, and VG denote a set of tuple values for shared variables in $\mathcal{V}$. A global state s of $\mathcal{CS}$ is a tuple $(s_1, \cdots, s_n, v) \in \mathcal{S} = (C_1 \times VL_1) \cdots \times (C_n \times VL_n) \times VG$ where $s_i \in C_i \times VL_i$ and $v \in VG$ denotes the values of the shared global variables. Note, s_i denotes the local state tuple (c_i, x_i) where $c_i \in C_i$ represents the local control state, and $x_i \in VL_i$ represents the local data state. A global transition system for $\mathcal{CS}$ is an interleaved composition of the individual EFSM models, M_i. Each global transition consists of firing of a local transition $t_i = (a_i, g_i, u_i, b_i) \in \mathcal{T}$. In a given global state s, the local transition t_i of model M_i is said to be *scheduled* if $c_i = a_i$, where c_i is the local control state component of s_i. Further, if enabling predicate g_i evaluates to true in s, we say that t_i is *enabled*. Note, in general, more than one local transition of model M_i can be *scheduled* but exactly one of them can be *enabled* (M_i is a deterministic EFSM). The set of all transitions that are enabled in a state s is denoted by *enabled*(s).

We can obtain a synchronous execution model for $\mathcal{CS}$ by defining a scheduling function $E : \mathcal{M} \times \mathcal{S} \mapsto \{0, 1\}$ such that t is said to be *executed* at global state s, *iff*

$t \in enabled(s) \cap T_i$ and $E(M_i, s) = 1$. Note, in interleaved semantics, at most one enabled transition can be executed at a global state s. In this synchronous execution model, each thread local state s_i (with shared access) has a *wait-cycle*, i.e., a self-loop to allow all possible interleavings.

Semantics of a sequentially consistent memory model [25, 28] are as follows:

- *Program Order Rule*: Shared accesses, i.e. read/write to shared variables, should follow individual program thread semantics.
- *Total Order Rule*: Shared accesses across all threads should have a total order.
- *Read Value Rule*: A read access of a shared variable should observe the effect of the last write access to the same variable in the total order.
- *Mutual Exclusion Rule*: Shared accesses in matched locks/unlock operations should be mutually exclusive.

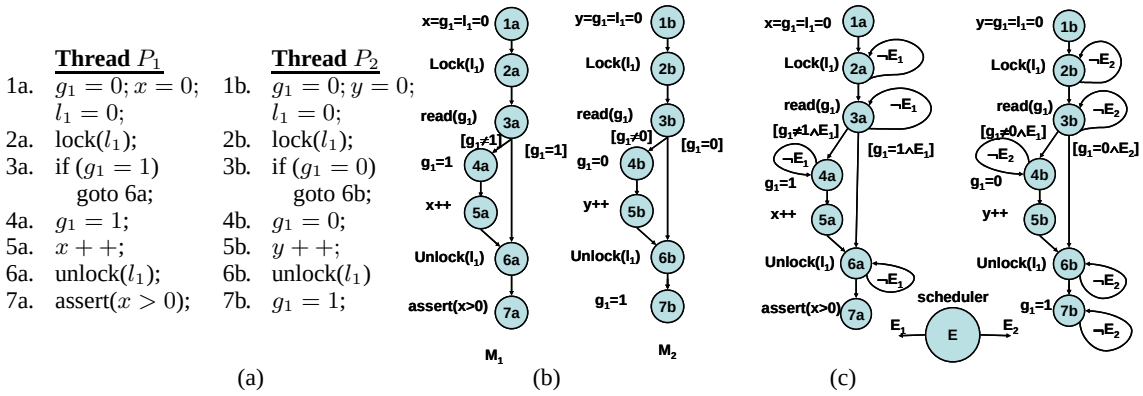


Fig. 1. (a) Concurrent system with threads P_1 and P_2 with local variables x and y respectively, communicating with lock l_1 and shared variable g_1. (b) CFG of P_1 and P_2, and (c) CFG of concurrent system with scheduler E.

Example: We illustrate a concurrent system comprising threads P_1 and P_2 with local variables x and y, respectively, interacting through lock l_1 and shared variable g_1, as shown in Figure 1(a). Each numbered statement is *atomic*, i.e., it cannot be interrupted. EFSM models M_1 and M_2 of the two threads P_1 and P_2 are shown as control flow graphs (CFG) in Figure 1(b). Note, M_1 is the tuple $(c_{01}, C_1, I_1, D_1, T_1)$ with $c_{01} = 1a$, $C_1 = \{1a, \cdots, 7a\}$, $I_1 = \{\}$, $D_1 = \{x\} \cup \{g_1, l_1\}$. The transitions are shown by directed edges with enabling predicates (if not a tautology) shown in square brackets and update functions are shown on the side of each control state. The model M_2 is similarly defined. An synchronously interleaved model for the concurrent system with threads P_1 and P_2, i.e., $\mathcal{CS}$= $(\{M_1, M_2\}, \{g_1, l_1\}, \{T_1, T_2\}, ((1a, x), (1b, y), (g_1, l_1)))$, with global shared variable g_1 and lock variable l_1, and a scheduler E is shown in Figure 1(c). It is obtained by inserting a wait-cycle, i.e., a self-loop at each control state of model M_i and associating the edge with a Boolean guard E_i such that $E_i = 1$ *iff* $E(M_i, s) = 1$. To understand the need for such wait-cycles, consider a global state s with thread control states at $2a$ and $6b$, respectively. To explore both the interleaving $2a \rightarrow 3a$ and $6b \rightarrow 7b$ from s, each thread needs to wait when the other makes the transition. By noting that the transitions at control states $5a$, $7a$, and $5b$ correspond to

non-shared memory accesses, one can remove the self-loops at these control states. In general, however, all self-loops can not be removed.

2.2 Building EFSMs from C threads

For brevity, we highlight the essentials in building a thread model (EFSM) from a C thread (using the F-Soft framework [29]) under the assumption of a bounded heap and a bounded stack. First we obtain a simplified CFG by creating an explicit memory model for (finite) data structures and heap memory, where indirect memory accesses through pointers are converted to direct accesses by using auxiliary variables. We model arrays and pointer arithmetic precisely using a sound pointer analysis. We model loops in the CFG without unrolling them. We handle non-recursive procedures by creating a single copy (i.e., not inlining) and using extra variables to encode the call/return sites. Recursive procedures are inlined up to some user-chosen depth. We perform merging of control nodes in CFG involving parallel assignments to local variables into a basic block, where possible, to reduce the number of such blocks. We, however, keep each shared access as a separate block to allow context-switches.

From the simplified CFG, we build an EFSM with each basic block identified with a unique *id* value, and a control state variable PC denoting the current block *id*. We construct a symbolic transition relation for PC, that represent the guarded transitions between the basic blocks. For each data variable, we add an update transition relation based on the expressions assigned to the variable in various basic blocks in the CFG. We use *Boolean* expressions and *arithmetic* expressions to represent the guarded and update transition functions, respectively.

2.3 Control State Reachability (CSR) and CSR-based BMC Simplification

Control state reachability (CSR) analysis is a breadth-first traversal of the CFG (corresponding to an EFSM model), where a control state b is one step reachable from a *iff* there is a transition edge $a \longrightarrow b$. At a given sequential depth d, let $R(d)$ represent the set of control states that can be reached *statically*, i.e., ignoring the guards, in one step from the states in $R(d-1)$, with $R(0) = s_0$. Computing CSR for the CFG of M_1 shown in Figure 1(b), we obtain the set $R(d)$ for the first six depths as follows: $R(0) = \{1a\}$, $R(1) = \{2a\}$, $R(2) = \{3a\}$, $R(3) = \{4a, 6a\}$, $R(4) = \{5a, 7a\}$, $R(5) = \{6a\}$, $R(6) = \{7a\}$. For some d, if $R(d-1) \neq R(d) = R(d+1)$, we say that the CSR *saturates* at depth d, and $R(t) = R(d)$ for $t > d$.

CSR can be used to reduce size of a BMC instance significantly [21]. Basically, if a control state $r \notin R(d)$, then the unrolled transition relation of variables that depend on r can be simplified. We define a Boolean predicate $B_r \equiv (PC = r)$, where PC is the program counter that tracks the current control state. Let v^d denote the unrolled variable v at depth d during BMC unrolling. Consider the thread model M_1, where the next state of variable g_1 is defined as *next*$(g_1) = B_{1a}? \ 0 : B_{4a}? \ 1 : g_1$ (using C language notation ?,: for *cascaded if-then-else*). At depths $k \notin \{0, 3\}$, $B^k_{1a} = B^k_{4a} = 0$ since $1a, 4a \notin R(k)$. Using this unreachability control state information, we can *hash* the expression representation for g_1^{k+1} to the existing expression g_1^k, i.e., $g_1^{k+1} = g_1^k$. This hashing, i.e., reusing of expressions, considerably reduces the size of the logic formula, i.e., the BMC instance.

3 Motivation: Why wait-cycles are bad?

The scope of CSR-based BMC simplification is reduced considerably by a large cardinality of the set $R(d)$, i.e., $|R(d)|$, and hence, the performance of BMC also gets

affected adversely. In general, re-converging paths of different lengths and different loop lengths are mainly responsible for enlarging set R, due to inclusion of all control states in a loop, and ultimately leading to saturation [21]. For example, computing *CSR* on the concurrent synchronous model (Figure 1(c)), we obtain $R(d)$ as follows:

$R(0)$= $\{1a, 1b\}$, $R(1)$= $\{2a, 2b\}$, $R(2)$= $\{2a, 3a, 2b, 3b\}$, $R(3)$= $\{2a, 3a, 4a, 6a, 2b, 3b, 4b, 6b\}$, $R(4)$= $\{2a, 3a, 4a, 5a, 6a, 7a, 2b, 3b, 4b, 5b, 6b, 7b\}$, $R(t)$= $R(4)$ for $t > 4$ (Saturates at 4)

Clearly, saturation is inevitable due to the presence of self-loops. At $t \geq 4$, the BMC unrolled transition relation cannot be simplified further using unreachable control states, i.e., not in $R(t)$. Thus, the scope of reusing the expression for next state logic expression is also reduced heavily. In general, saturation can also be caused by program loops. To overcome that we use a *Balancing Re-convergence* strategy [21] effectively to balance the lengths of the re-convergent paths and loops by inserting NOP states. An NOP state does not change the transition relation of any variable. However, this approach does not work well in the presence of self-loops.

In our experience, synchronous models with self-loops for modeling interleaving semantics are not directly suitable for verifying concurrent systems using BMC. Instead, we propose a modeling paradigm that eliminates self-loops with the goal of reducing the size of BMC instances. However, there are many challenges in doing so in a BMC framework.

- We would like to have soundness and completeness, i.e., neither to miss true witnesses nor to report spurious witnesses (up to some bounded depth). We do so by decoupling the individual thread models, and add the required memory consistency constraints on-the-fly to the unrolled BMC instances. This allows us to exploit advances in SMT-based BMC, without the expensive modeling of interleaving semantics. Note that for bounded analysis the number of shared memory accesses are also bounded, and therefore, these constraints are typically smaller than the model with constraints added eagerly, in practice.
- We also would like to formulate and solve iterative BMC problems incrementally, and integrate seamlessly state-of-the-art advancements in static analysis and BMC. We achieve our goals in a lazy modeling paradigm that simultaneously facilitates the use of several static techniques such as context-sensitive CSR and lockset analysis [9, 10, 14] and model transformations [21] to accelerate the performance of BMC.

4 Lazy Modeling Paradigm: Overview

We illustrate the main idea of our modeling using an example in Section 4.1, and highlight novelties in our approach in Section 4.2. A formal exposition of the modeling and its soundness is provided in Section 5.

4.1 Basic Approach

As a first step in our modeling, we construct abstract and independent (i.e. decoupled) thread models LM_1 and LM_2 corresponding to the threads P_1 and P_2 as shown in Figure 2. We introduce atomic thread-specific procedures `read_sync` and `write_sync` before and after every shared access. In Figure 2, the control states r_i and w_i correspond to calls to procedures `read_sync`$_i$ and `write_sync`$_i$, respectively. We also introduce global variables TK, CS_1, and CS_2 described later. For each thread, we make the global variables *localized* by renaming. As shown in Figure 2, for P_1 (and similarly for

P_2), we rename g_1, l_1, TK, CS_1, and CS_2 to local variables g_{11}, l_{11}, TK_1, CS_{11}, and CS_{21}, respectively. The localized shared variables get non-deterministic (ND) values in the control state r_i. (Refer Section 5.1 for detailed instrumentation.) The models LM_i obtained after annotations are *independent* since the update transition relation for each variable now depends only on the local state variables. Note, there are no self-loops in these models. However, due to ND read values for shared variables in r_i control state, these models have additional behaviors, which we eliminate by adding concurrency constraints as described below.

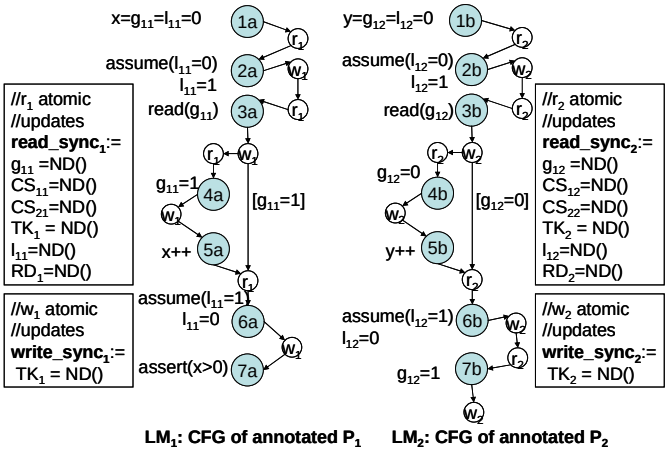
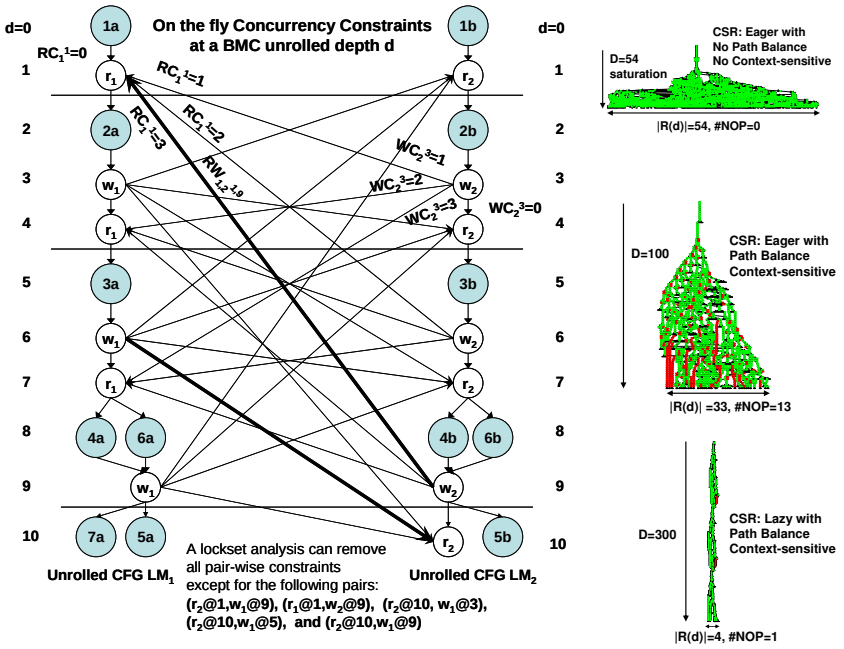


Fig. 2. CFG of threads P_1 and P_2 with annotations.

We unroll each model LM_i independently during BMC (i.e, with possibly different unroll depths). To each BMC instance, we add concurrency constraints on-the-fly between each pair of control states with accesses to shared variables, that are statically reachable in unrolled CFG at the corresponding thread-specific depths. For sequential consistency, these constraints allow sufficient context-switching to maintain the *read value property*, and sequentialize the context-switches to enforce a common *total order*. Note, in addition to these concurrency constraints, a BMC instance comprises transition relation of all thread models and the property constraints. The transition relation of each thread model ensures that memory accesses within the thread follow the *program order*. Specifically, to capture context-switching events, we added a Boolean shared variable referred to as *token* (TK). The semantics of a token asserted by a thread is equivalent to a guarantee that all *visible* operations, i.e., shared memory accesses, issued so far have been *committed* for other threads to see. Initially, only one thread (chosen non-deterministically) is allowed to assert its token. To track the sequentiality of the global execution and maintain *total order*, we also add two global clock variables (one per thread) i.e., CS_1 and CS_2 to timestamp [28] the token passing events. The pair-wise constraints, added between shared access states, allow *passing* of the token. Whenever the token is passed from thread LM_i (post-access shared state) to LM_j (pre-access shared state) the concurrency constraints ensure that each localized shared variable of LM_j gets the current state value of the corresponding localized shared variable of LM_i, and the clock variables get synchronized. We call these pair-wise constraints as *Read-Write Synchronization Constraints* as described in Section 5.2 with more details.



(a) (b)

Fig. 3. (a) Unrolled CFG of thread models LM_1 and LM_2 with concurrency constraints added on-the-fly during BMC unrolling. The dark arrows show the token passing events (i.e., context switches) leading to a data race between source lines 3a and 7b (in Figure 1(a)). (b) CSR graphs with path balancing (PB), context-sensitive (CXT) for eager/lazy on a thread-model (Green/Red dots denote non-NOP/NOP blocks, resp.)

We illustrate the concurrency constraints added in Figure 3(a). Let $c@k$ denote the control state c reached statically at depth k. Computing CSR, we obtain $R(0) = \{1a@0, 1b@0\}$, $R(1) = \{r_1@1, r_2@1\}$, $R(2) = \{2a@2, 2b@2\}$and so on. The concurrency constraints added between a pair $(r_i@k, w_j@h)$ are shown as an arrow from w_j (of LM_j) at depth h to r_i (of LM_i) at depth k. Note, r_i corresponds to pre-access shared state of LM_i, and w_j corresponds to post-access shared state of LM_j. These constraints also capture the exclusivity of the pair, i.e., constraint for pair $(r_i@k, w_j@h)$, excludes other pairs $(r_i@k$,*) and (*,$w_j@h)$. For BMC at depth $d < 3$ we do not add any pair-wise constraints. For BMC at $d = 3$, we add concurrency constraints corresponding to pairs $(r_1@1, w_2@3)$ and $(r_2@1, w_1@3)$. For BMC at $d = 4$, we add concurrency constraints corresponding to pairs $(r_1@1, w_2@3)$, $(r_2@1, w_1@3)$, $(r_1@4, w_2@3)$, and $(r_2@4, w_1@3)$. Note, in incremental formulation of BMC, we only need to add the constraints corresponding to the last two pairs. In general, the constraints grow quadratically with the analysis depth (ref. Section 6). In addition, we use various static analyses and model transformations to reduce the set of pair-wise constraints added that are redundant (ref. Section 7). For example, using the lockset analysis [9,10,14], we can remove constraints for all pairs other than $(r_2@1, w_1@9)$, $(r_1@1, w_2@9)$, $(r_2@10, w_1@3)$,

($r_2@10, w_1@5$) and ($r_2@10, w_1@9$). This is because other pairs are not reachable simultaneously in the mutually exclusive region, protected by the lock.

In our approach, a data race condition is detected, if there exists a witness trace where a token passing event occurs between the pair ($r_i@k, w_j@h$) with shared accesses on the same variable, with at least one access being a write. In Figure 3(a), we indicate the witness trace corresponding to the data race between source lines 3a and 7b (from Figure 1(c)) as a sequence of the token passing events highlighted in **bold arrows** between the pairs $(r_1@1, w_2@9)$ and $(r_2@10, w_1@5)$. Note, the control state pair $(r_2@10, w_1@5)$ corresponds to simultaneous accesses of shared variable g_1 with $r_2@10$ being a pre-write access state. We obtain the witness trace by unrolling each model in BMC up to depth 11. In a synchronously interleaved model (i.e., with wait-cycles), we would have obtained the trace at an unrolled depth of $11 + 7 = 18$. Note, we sum the two depths, as thread P_2 has to wait (self-loop) after context switch to P_1.

For a thread model (example described in Section 8), we compare the reachability graphs on the lazy and eager models, as shown in Figure 3(b), with and without using model transformation such as path/loop balancing (PB) [21] and context-sensitive CSR (CXT) [30]. Note, the width of the graph is proportional to $|R(d)|$. It is desirable that the width is small for greater BMC simplification. We observe that path/loop balancing is more effective on our lazy models (i.e., without wait-cycles) as $R(d)$ is reduced significantly compared to eager models (i.e., with wait-cycles).

4.2 Our Contributions

The main idea of our modeling paradigm for concurrent systems is to move away from expensive modeling based on synchronous interleaving semantics. We focus primarily on reducing the size of the BMC problem instances to enable deeper search within the limited resources, both time and memory. Features and merits of our approach are:

1. *Lazy modeling constraints:* By adding the constraints *lazily*, i.e., as needed for a bounded depth analysis, as opposed to adding them *eagerly*, we reduce the BMC problem size at that depth. The size of these concurrency-modeling constraints depends *quadratically* on the number of shared memory accesses at any given BMC depth in the worst case. Since the analysis depth of BMC bounds the number of shared memory accesses, these constraints are typically smaller than the model with constraints added eagerly, in practice.
2. *No wait-cycle:* We do not allow local wait cycles, i.e., there are no self-loops in read/write blocks with shared accesses. This enables us to obtain a reduced set of statically reachable blocks at a given BMC depth d, which dramatically reduces the set of pair-wise concurrency constraints that we need to add to the BMC problem.
3. *Deeper analysis:* For a given BMC depth D and n concurrent threads, we guarantee finding a witness trace (if it exists), i.e., a sequence of global interleaved transitions, of length $\leq n \cdot D$, where the number of local thread transitions is at most D. In contrast, an eager modeling approach using BMC [14], an unrolling depth of $n \cdot D$ is needed for such a guarantee. Thus, we gain in memory use by a factor of n.
4. *Using static analysis:* We use property preserving model transformations such as path/loop balancing, and context-sensitive control state reachability to reduce the set of blocks that are statically reachable at a given depth. Again, this potentially reduces the lazy modeling constraints. We also use *lockset* [9, 10, 14] analysis to reduce the set of constraints, by statically identifying which block pairs (with shared accesses) are simultaneously unreachable.

5. *SMT-based BMC:* We use an SMT solver instead of a traditional SAT solver, to exploit the richer expressiveness, in contrast to bit-blasting. We effectively capture the *exclusivity* of the pair-wise constraints, i.e., for a chosen shared access pair, other pairs with a common access are implied invalid immediately.

5 Lazy Modeling of Concurrent Systems

We present details of our modeling in Sections 5.1 and 5.2.

5.1 Sound Abstraction: Independent Thread Models

We perform source-to-source transformations to directly use a model builder (F-Soft [29]) for sequential programs and obtain sound abstraction.
Token: We introduce a global Boolean variable, a token TK, to signify that the thread with the token can execute a shared access operation and commit its current shared state to be *visible* to the future transitions. Initially, only one thread, chosen non-deterministically, is allowed to assert TK. Later, this token is *passed*, from one thread to another, i.e., de-asserted in one thread and asserted by the other thread, respectively.
Logical Clock: To obtain a total ordering on token *passing* events, we use the concept of *logical clocks* and *timestamp* [28]. We add a global clock variable CS_i for each thread P_i, so that the tuple $(CS_1 \cdots CS_n)$ represents the logical clock. These variables are initialized to 0. Whenever a token TK is acquired by a thread P_i, CS_i is incremented by 1 in P_i. The variable CS_i keeps track of the number of occurrences of token passing events wherein thread P_i acquires the token from another thread P_j, $j \neq i$.
Race Detector: We add a race detector local Boolean variable RD_i for each thread P_i. This variable is set to 1 (initially, 0) whenever a shared variable is accessed in thread P_i and written in another thread P_j while token is passed from P_j to P_i.
Localization: For each thread, we make the global variables *localized* by renaming.
Atomic Procedures: We add atomic thread-specific procedures `read_sync` and `write_sync` before and after every shared access. In the `read_sync` procedure, each *localized* shared and race detector variable get a non-deterministic value, (`ND`), while in the `write_sync` procedure only TK gets an `ND` value.
Synchronization primitives: Operations `lock(lk)` and `unlock(lk)` are modeled as atomic operations `assume`($lk = 1$) and `assume`($lk = 0$), respectively. To maintain synchronization semantics, we only consider wait-free execution [15] where the acquisition of the same lock twice is disallowed in a row without an intermediate unlock. Note, this consideration is sufficient to find all data races.

5.2 Concurrency Constraints

Given independent abstract models, obtained as above, we add concurrency constraints *incrementally*, and *on-the-fly* to each BMC instance, in addition to the transition constraints of the individual thread models and property constraints. The concurrency constraints capture *inter-* and *intra-* thread dependencies due to interleavings, and thereby, eliminate additional behaviors in the models up to a bounded depth. Specifically, these constraints comprise (a) pair-wise, i.e., inter-model constraints (shown in Table 1), (b) single-threaded, i.e., intra-model constraints (shown in Table 2), and (c) global constraints (shown in Table 3). In the following, we use $R_i(d)$, $0 \leq d \leq D$, to denote the set of control states reachable at depth d for each thread model LM_i, for a given BMC bound D. (Note, we compute *CSR* on each of the models LM_i separately before starting BMC.) Also, we use x_i^k to denote the expression for the variable x in the unrolled model LM_i at depth k.

Table 1. Pair-wise concurrency constraints added in each BMC instance

P1. Read-Write Synchronization Enabling Constraint: For every pair of *read_sync* control state in LM_i, $r_i \in R_i(k)$ and *write_sync* control state in $LM_j, j \neq i, w_j \in R_j(h)$, we introduce a Boolean variable RW_{ij}^{kh}, and add the following enabling constraint:

$$RW_{ij}^{kh} \iff (B_{r_i}^k \wedge \neg TK_i^k \wedge B_{w_j}^h \wedge TK_j^h \wedge CS_{ii}^k = CS_{ij}^h) \quad (1)$$

If $RW_{ij}^{kh} = 1$, we say, the *token passing condition* is enabled.

P2. Read-Write Synchronization Exclusivity Constraint: Let RS_i^k define the set $\{RW_{ij}^{kh} | i \neq j, 0 \leq h \leq d\}$ for a *read_sync* control state of LM_i at depth k. To allow at most one *write_sync* (from a different thread) to match with this *read_sync*, we assign a unique id $a_j^h \neq 0$ to each element of RS_i^k. We add a new variable RC_i^k for the *read_sync* control state of LM_i at depth k, *require* that it takes value $a_j^h \neq 0$ *iff* $RW_{ij}^{kh} = 1$. Similarly, we introduce a new variable WC_j^h for the *write_sync* control state of LM_j at depth h, and *require* that it takes value $b_i^k \neq 0$ *iff* $RW_{ij}^{kh} = 1$. The constraints added are:

$$RW_{ij}^{kh} \iff (RC_i^k = a_j^h), a_j^h \neq 0 \quad (2)$$

$$RW_{ij}^{kh} \iff (WC_j^h = b_i^k), b_i^k \neq 0 \quad (3)$$

Thus, if $RW_{ij}^{kh} = 1$, we require that both $RC_i^k \neq 0$ and $WC_j^h \neq 0$; and vice-versa.

P3. Read-Write Synchronization Update Constraint: For every RW_{ij}^{kh} variable introduced, we add the following update constraints:

$$RW_{ij}^{kh} \implies \bigwedge_{p=1}^{m} g_{pi}^{k+1} = g_{pj}^h \quad (4)$$

$$RW_{ij}^{kh} \implies (TK_i^{k+1} \wedge \neg TK_j^{h+1}) \quad (5)$$

$$RW_{ij}^{kh} \implies (CS_{ii}^{k+1} = CS_{ii}^k + 1) \wedge (\bigwedge_{q=1, q\neq i}^{n} CS_{qi}^{k+1} = CS_{qj}^h) \quad (6)$$

P4. Data Race Detection Property Constraint: We define two predicates *will_access* and *just_written* statically, where *will_access*$(r_i, g)= 1$ *iff* shared variable g is accessed in the next local control state reachable from r_i, and *just_written*$(w_i, g)= 1$ *iff* shared variable g was written in the previous local control state reachable to w_i. We add the following race detection constraint only if *will_access*$(r_i, g)= 1$ and *just_written*$(w_j, g)= 1$:

$$RW_{ij}^{kh} \implies RD_i^{k+1} \quad (7)$$

Table 2. Single threaded concurrency constraints added in each BMC instance

S1. No Sync Update Constraint: When none of the token passing events is triggered for a *read_sync* control state of LM_i at depth k, we force the next state values to be *unchanged* for each localized shared and race detector variable in LM_i by adding:

$$RC_i^k = 0 \Longrightarrow (\bigwedge_{p=1}^{m} g_{pi}^{k+1} = g_{pi}^k) \wedge (RD_i^{k+1} = RD_i^k) \tag{8}$$

$$RC_i^k = 0 \Longrightarrow TK_i^{k+1} = TK_i^k \tag{9}$$

$$RC_i^k = 0 \Longrightarrow \bigwedge_{q=1}^{n} CS_{qi}^{k+1} = CS_{qi}^k \tag{10}$$

Similarly, for every *write_sync* control state of LM_j at depth h, we force the next state token value to be *unchanged* by adding a similar constraint:

$$WC_j^h = 0 \Longrightarrow TK_j^{h+1} = TK_j^h \tag{11}$$

S2. Lock/Unlock Synchronization Constraint: To model `assume`$(lk = 0)$ in *lock* control state l_i of LM_i at depth k, and similarly, for *unlock* control state ul_i, we add

$$B_{l_i}^k \Longrightarrow (\neg lk_i^k); \quad B_{ul_i}^k \Longrightarrow (lk_i^k) \tag{12}$$

S3. Write Commit Constraint: We make only a write operation commit its current shared state to be *visible* to the future transitions by adding the following constraint in *write_sync* control state w_j of LM_j at depth h corresponding to write operation only, i.e.,

$$B_{w_j}^h \Longrightarrow TK_j^h \tag{13}$$

S4. Single Control State Reachability Property Constraint: For checking reachability of a local control state $a \in C_i$, we add constraint:

$$B_a^k \Longrightarrow TK_i^k \tag{14}$$

Table 3. Global concurrency constraints added in each BMC instance

Single Token Constraint: Initially, exactly one thread model has the token. We add,

$$(\bigvee_{1 \le i \le n} TK_i^0) \wedge (\bigwedge_{i \neq j} TK_i^0 \Longrightarrow \neg TK_j^0) \tag{15}$$

Multiple Race Detections: To check multiple data races *incrementally*, we add the following blocking clause corresponding to the *token passing events* seen in the last witness trace.

$$\neg(RW_{ij}^{kh} \wedge \cdots \wedge RW_{i'j'}^{k'h'}) \tag{16}$$

We add the following pair-wise (i.e., inter-model) constraints *P1-P4* between unrolled models LM_i and LM_j at depths k and h, respectively as shown in Table 1.

P1. Read-Write Synchronization Enabling Constraint: The constraints (Eqn 1) capture the enabling of *token passing* condition. This happens exactly when: a) thread model LM_i is in *read_sync* control state (i.e., pre-access shared state) at depth k and does not hold the token , b) thread model LM_j is in *write_sync* control state (i.e., post-access shared state) at depth h and holds the token, and c) thread model LM_j has the latest value of clock variable of LM_i, and both threads agree on that. Note, this constraint *per se* is not enough for *token passing*, and we require the following exclusivity constraint as well.

P2. Read-Write Synchronization Exclusivity Constraint: Exclusivity constraints (Eqn 2-3) ensure that for a chosen pair for token passing $(r_i@k, w_j@h)$ with $i \neq j$, other pairs $(r_i@k, w_{j'}@h')$ with $h \neq h'$ or $j \neq j'$ and $(r_{i'}^{k'}, w_j^h)$ with $i \neq i'$ or $k \neq k'$ are *implied invalid*. As shown in Figure 3(a), the pair $(r_1@1, w_2@3)$ excludes other pairs $(r_1@1, w_2@6)$, $(r_1@1, w_2@9)$, $(r_1@4, w_2@3)$, and $(r_1@7, w_2@3)$ due to specific values of variables RC_1^1 and WC_2^3 chosen. Note, Eqn 1, together with Eqn 2 and 3, define RW_{ij}^{kh}. We say a *token passing event* is triggered *iff* $RW_{ij}^{kh} = 1$.

P3. Read-Write Synchronization Update Constraint: If the token passing event is triggered, each localized shared variable of LM_i at depth k gets the current state value of the corresponding localized shared variable of LM_j at depth h (Eqn 4), the next state value of token of LM_i is constrained to 1, while it is constrained to 0 for LM_j, indicating a transfer of the token (Eqn 5), the next state value of the clock variable of LM_i is incremented by 1, while the remaining clock variables are *sync*-ed with that of LM_j (Eqn 6).

P4. Data Race or Pair-wise Reachability Property Constraints: A data race is detected, i.e., $RD_i^{k+1} = 1$ at thread model i at depth $k+1$ whenever for a shared variable g read-write synchronization enabling constraint RW_{ij}^{kh} holds (Eqn 7) with at least one write access on g. To check whether control states $a \in C_i$ (of LM_i) and $b \in C_j$ (of LM_j) are reachable simultaneously, we reduce the reachability problem to a token passing event detection, i.e., $RW_{ij}^{kh} = 1$ by adding control states *read_sync* and *write_sync* before and after control states a and b.

We add following single-threaded (i.e., intra-model) constraints *S1-S4* for each thread model LM_i (LM_j) at depth k (h) as shown in Table 2.

S1. No Sync Update Constraint: The constraints (Eqn 8-11) keep the states of thread-specific localized global variables, and newly introduced variables unchanged when there is no token passing event (i.e. no context switching) occurs.

S2. Lock/Unlock Synchronization Constraint: The constraints (Eqn 12) assert/de-assert the locking predicate variables in the respective lock/unlock states.

S3. Write Commit Constraint: The constraints (Eqn 13) make the write operation of a thread visible to others by asserting the token.

S4. Single Control State Reachability Property Constraint: Like *S3*, the constraints (Eqn 14) make the reachability of control state visible by asserting the token.

As shown in Table 3, we add a single token constraint (Eqn 15) needed for total order and blocking clauses (Eqn 16) to detect multiple data races.

6 Correctness and Size Complexity

Theorem 1 *(Correctness) (A) The lazy modeling constraints allow only those traces that respect the sequential consistency of memory model and synchronization semantics up to the bound D, i.e., our modeling is complete.*

(B) Further, if there exists a witness for a reachability property, such that the global trace length is $\leq n \cdot D$ and each local trace length $\leq D$, there exists an equivalent trace allowed by our model corresponding to the witness trace. In other words, our modeling is sound in that it does not miss any witness up to these bounds.

Proof: Here is an outline with details in Appendix A.

- *Completeness:* Our modeling captures the requirements for sequential consistency (a) *program order*: using the transition model of each thread, *No Sync Update Constraint* (Eqn 9-11), and *Write Commit Constraint* (Eqn 13), (b) *total order of shared accesses:* using logical clock along with *Read-Write Synchronization* (Eqn 5 and 6) and *No Sync Update Constraint* (Eqn 9 and 10), (c) *read value rule:* using *Read-Write Synchronization Constraint* (Eqn 4), and *No Sync Update Constraint* (Eqn 8), and (d) *mutual exclusion rule:* using *Lock/Unlock Synchronization Constraint* (Eqn 12).
- *Soundness:* We add pair-wise constraints for *all* pairs of shared accesses that are *statically reachable* up to the bounded depth. Thus, we capture all possible interleavings of shared accesses up to the bound, and hence, we cannot miss any witness up to the bound.□

6.1 Size Complexity

We now discuss the size of constraints and variables *incrementally* added at each depth d for n concurrent threads. We consider thread specific `read_sync` and `write_sync` procedure calls, without inlining. This implies that at each unrolled depth k of LM_i, at most one *read_sync* control state r_i and at most one *write_sync* control state w_i belong to the reachable set $R_i(k)$. Thus, at depth d, r_i (or w_i) block is paired with at most $(n \cdot d)$ w_j (or r_j) blocks. Since there are n threads, we have at most $(n^2 \cdot d)$ pairs. Thus, at depth d, the number of pair-wise constraints added, and variables introduced are $O(d)$, and number of non-pair constraints added is $O(1)$. Overall, the size of constraints and variables added up to depth d is $O(d^2)$. Thus, the concurrency constraints grow *quadratically* with unrolling in the worst case.

For X memory accesses up to depth d, the size complexity can be shown to be $O(X^2)$. To compare, the previous approaches [23, 24], incur a *cubic* cost, i.e., $O(X^3)$, for a given memory model and a test program.□.

7 Removal of Redundant Concurrency Constraints

We use various static analyses to reduce the number of context switches to consider. We identify and remove redundant concurrency constraints corresponding to them as follows:

- *CFG transformation* (PB): We use path/loop balancing transformations [21] on each thread model independently to obtain a reduced set of statically reachable blocks in *CSR*. This reduces concurrency and unrolled transition constraints.

– *Lockset-based analysis* (`MTX`): We determine statically pair-wise unreachability of *read_sync* and *write_sync* control states using *lockset* [9, 10, 14] analysis. For such pairs of *read_sync* and *write_sync* control states that are mutually exclusive (e.g., due to matching locks/unlocks), we do not add pair-wise constraints as the concurrency semantics forbids context-switching between those thread states.
– *Context-sensitive CSR* (`CXT`): In our modeling, we handle non-recursive procedures by creating a single copy and using extra variables to encode the call/return sites. (Recursive procedures are inlined upto some user-chosen depth). However, not inlining a procedure can cause *false* loops in the CFG of each thread, due to unmatched calls/returns of a procedure. We avoid saturation in CSR due to *false* loops in CSR by determining reachability in a context-sensitive manner, i.e., by matching the call/return sites for each procedure call. We observe in our experimental results that such analysis gives a dramatically reduced set of reachable *read_sync* and *write_sync* control states, and hence, a reduction in the set of pair-wise constraints added.

Discussion: The above mentioned static analysis techniques are not required to be precise as the imprecision does not affect completeness and soundness (Theorem 1) and size complexity of the overall analysis. However, less precise (but conservative) static analysis may result in less simplification and thereby poorer BMC performance. Furthermore, we can utilize transactions and/or partial order reductions [8, 14], to eliminate some pair-wise concurrency constraints.

8 Experiments

We experimented on the `Daisy file system` [31], a public benchmark used to evaluate and compare various concurrent verification techniques for concurrent threads communicating with shared memory and locks/unlocks. It is a 1KLOC Java implementation of a representative file system, where each file is allocated a unique inode that stores the file parameters, and a unique block which stores data. Each access to a file, inode or block is guarded by a dedicated lock. Since a thread does not guarantee exclusive operations on byte access, potential race conditions exist. This system has some known data races. For our experiments, we used a C version of the code [14] with a two-threaded concurrent system.

We conducted our experiments on an SMT-based BMC framework similar to [21]. We used the `yices-1.0` [32] SMT solver at the back-end. We compared our *lazy modeling* with an *eager modeling*. In eager modeling approach, we add the pair-wise constraints in the model itself between the states with shared access, that also have wait-cycles. We applied BMC simplification using *CSR* as discussed in Section 2.3 for all cases, referred to as the baseline strategy. We then combined this baseline strategy with other static analysis techniques such as path balancing/loop CFG transformations (`PB`) on CFG [21], context-sensitive analysis (`CXT`), and *lockset* analysis (`MTX`) [9, 10, 14]. We conducted controlled experiments with various combinations of these techniques.

8.1 Comparing BMC results

We now compare the performance of SMT-based BMC on detecting *multiple* data races, on both eager/lazy models, in various combinations of strategies. Note, multiple race detection is an optional feature. We detect *multiple data races* incrementally, by adding a blocking clause corresponding to the token passing events *seen* in the last witness trace to the satisfiable BMC instance, and then continuing the search. We conducted our experiments on a workstation with dual Intel 2.8 GHz Xeon Processors with 4GB physical

memory running Red Hat Linux 7.2, using a 6 hrs (≈20Ks) time limit and unroll bound limit of 300 for each BMC run. The results are summarized in Table 4(a). Column 1, shows the modeling approach (eager/lazy); Columns 2-6 show BMC results for various combinations of static analysis methods. Each *data point* $(d : t, m)$ corresponds to a performance summary of BMC up to depth d, with t and m representing the cumulative run time and memory used, respectively. Note, cumulative time includes the solve time incurred in the previous depths for the same run. We show a selected few *data points* for comparison. Specifically, Column 2 shows data for CSR with no `PB` and no `CXT`; Column 3 shows data for CSR with `PB` and no `CXT`; Column 4 shows data for CSR with `PB` and `CXT`; Column 5 shows data for CSR with `PB`, `CXT` and `MTX`; and Column 6 shows data for CSR with `PB` and `MTX`, but no `CXT`. For eager modeling, due to non-inlining of procedure calls, we did not obtain any useful lockset information to reduce the constraints statically, and therefore, results in the columns `(CSR+PB+CXT+MTX)` and `(CSR+PB+MTX)` are the same as `(CSR+PB+CXT)` and `(CSR+PB)`, respectively. As an example, consider BMC at unroll depth 64. BMC on eager model with `CSR` times out (`TO`) requiring 66 Mb, while on lazy model with `CSR` it takes 26s and 39Mb.

In general, `PB` and `CXT` help BMC go deeper in both the eager and lazy models. However, `CXT` has a pronounced effect on the BMC performance. We also observed that the *lockset* analysis helps in improving the BMC performance, but not significantly. BMC on eager model, in general, does not go very deep, and times out in all cases without detecting any data races. In contrast, BMC on lazy model with (`CSR+PB+CXT`) or (`CSR+PB+CXT+MTX`) is able to find 50 data races in a single BMC run. Note, less precise static analysis `CSR` and `CSR+PB` show poorer performance compared to `CSR+PB+CXT` and `CSR+PB+CXT+MTX`.

In Table 4(b), we provide details of BMC performance on lazy models using `CSR+PB+CXT+MTX` on the first five data races. Column 1 shows the data races listed in the order of detection; Columns 2-4 show the BMC depth, cumulative time, and memory used, respectively. Column 5 shows the context-switches in the trace, each denoted as $(P_i : k_i, l_i) \rightarrow (P_j : k_j, l_j)$ where model P_i executes *uninterrupted* from depth k_i to l_i, and then switches the context to P_j at depth k_j. The Daisy example intentionally included many data races as a benchmark. Each race reported here corresponds to a unique set of context-switches in the witness trace. As an example, the first data race is detected at depth 143 taking 12s and 10Mb. There are 3 context switches: a P_1 run from depth 0 to 127, followed by a P_2 run from depth 0 to 127, followed by another P_1 run from 128 to 142, followed by a data race detection when P_2 accesses the same variable at depth 128. Note, the length of the trace is $271 (= 143 + 128)$.

8.2 Comparison with related work

In a related effort [14], two write-write data races were detected for the same benchmark, i.e. `Daisy file system` [31], using 1283s and 122Mb, and 5925s and 902Mb, respectively. Note, our eager modeling (used in experiments) differs from them mainly in the back-end solver, i.e., SMT vs SAT, and in use of static reduction methods, which was the crux of their approach. We believe that the orders of magnitude improvement in BMC performance (as reported in Table 4(b)), are attributed mainly to our lazy modeling paradigm that facilitates dramatic size-reduction of BMC instances.

In contrast to a closely related work TCMBC [15], we do not bound the number of context switches. Further, we add constraints lazily and incrementally during BMC unrolling. TCBMC, built over CBMC [33], translates concurrent C threads into static single assignment (SSA) form, and adds constraints for a bounded number of context-switches for a bounded depth. TCMBC approach requires full inlining of functions and

unwinding of loops like CBMC. This CBMC-based approach, therefore, is not scalable to large piece of code or code with reactive behavior, as shown previously [29].

We also contrast our work with [23, 24], where weaker memory models are considered. However, these approaches only check given test programs. Further, the number of concurrency constraints they add are *cubic* [24] in the number of shared accesses, while we add a *quadratic* number of constraints.

Table 4. SMT-based BMC (a) Comparison Results (b) Sample Witness Traces

(a) Comparing SMT-based BMC on Lazy/Eager Models

Model	Static Analysis Strategies				
	CSR (1)	CSR+PB (2)	CSR+PB +CXT(3)	CSR+PB +CXT+MTX(4)	CSR+PB +MTX(5)
	d: t, m with $d \equiv$BMC Depth, $t \equiv$Cum. Time(s), $m \equiv$Mem(Mb)				
Eager	64: TO,66	64: 132,21 95: TO,59	64: 10,14 95: 2K,31 124: TO,49	same as (3)	same as (2)
race?	N	N	N	N	N
Lazy	64: 26,39 73: 8K,101	64: 6,10 95: 35,10	64: 2,6 95: 5,8	64: 1,6 95: 4,8	64: 3,10 95: 17,28
	Yices aborted	118: TO,114	118: 8,11 124: 16, 10 287: 2.7K,34	118: 8,11 124: 9, 10 287: 2.4K,32	118: 15K,108 119: TO,112
race?	N	N	50 races	50 races	N

Note: N≡No Race Detected, *≡Yices Aborted, TO≡Time Out

(b) First 5 Data Race Traces using CSR+PB+CXT+MTX on lazy model

#	BMC depth	Time sec	Mem Mb	$(P_i : k_i, l_i) \rightarrow (P_j : k_j, l_j)$ P_i executes from depths k_i to l_i *uninterrupted* and context-switches to P_j at depth k_j.
1	143	12	10	(1: 0,127)→(2: 0,127)→(1: 128,142)→(2: 128,-)
2	174	25	13	(1: 0,127)→(2: 0,127)→(1: 128,173)→(2: 128,-)
3	180	30	15	(1: 0,14)→(2: 0,179)→(1: 15,-)
4	211	96	17	(1:0,127)→(2:0,158)→(1:128,158)→(2:159,210)→(1:159,-)
5	211	99	18	(1: 0, 127)→(2: 0,210)→(1: 128,-)

9 Conclusions and Future Work

We described a novel lazy modeling paradigm for shared memory multi-threaded concurrent systems, that is more suitable for BMC compared to synchronous modeling of interleaving semantics proposed previously. Such direct modeling of concurrency semantics in BMC is geared toward reducing the size of BMC instances, and thereby, improving the performance of BMC for deeper analysis. We add concurrency constraints lazily, incrementally and on-the-fly during BMC unrolling that preserve the concurrency semantics up to the bounded depth. We demonstrated several benefits of such an approach without incurring the cost of modeling interleaving semantics. By avoiding wait-cycles, our modeling allows greater scope for reduction in size of a BMC instance.. In addition, we use various static analyses to reduce the number of context-switches to consider, which further reduces the size of the constraints. We demonstrated the efficacy of our approach on a complex example.

In future, we would like to combine *partial-order reduction* methods [8, 14], add deadlock detection, and analyze weaker memory models.

References

1. S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 1996.
2. L. Lamport. How to make multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 1979.
3. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of TACAS*, 1999.
4. Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
5. G. Ramalingam. Context sensitive synchronization sensitive analysis is undecidable. In *ACM Transactions on Programming Languages and Systems*, 2000.

6. P. Godefroid. Model checking for programming languages using verisoft. In *Proc. of POPL*, 1997.
7. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. ZING: Exploiting program structure for model checking concurrent software. In *Proc. of CONCUR*, 2004.
8. P. Godefroid. *Partial-order Methods for the Verification of Concurrent Systems: An Approach to the State-explosion Problem*. PhD thesis, 1995.
9. C. Flanagan and S. Qadeer. Transactions for software model checking. In *Proc. of TACAS*, 2003.
10. S. D. Stoller. Model-checking multi-threaded distributed Java programs. *Journal on STTT*, 2002.
11. S. D. Stoller and E. Cohen. Optimistic synchronization-based state-space reduction. In *Proc. of TACAS*, 2003.
12. V. Levin, R. Palmer, S. Qadeer, and S. K. Rajamani. Sound transaction-based reduction without cycle detection. In *Proc. of SPIN Workshop*, 2003.
13. R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Proc. of CAV*, pages 340–351, 1997.
14. V. Kahlon, A. Gupta, and N. Sinha. Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In *Proc. of CAV*, 2006.
15. I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *Proc. of CAV*, 2005.
16. F. Lerda, N. Sinha, and M. Theobald. Symbolic model checking of software. In *Electronic Notes Theoretical Computer Science*, 2003.
17. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Proc. of TACAS*, 2005.
18. O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided Underapproximation-Widening for Multi-process Sytems. In *Proc. of POPL*, 2005.
19. B. Cook, D. Kroening, and N. Sharygina. Symbolic Model Checking for Asynchronous Boolean Programs. In *Proc. of SPIN Workshop*, 2005.
20. B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proc. of CAV*, 2006.
21. M. K. Ganai and A. Gupta. Accelerating high-level bounded model checking. In *Proc. of ICCAD*, 2006.
22. S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *Proc. of ISCA*, 1991.
23. Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Memory-model-sensitive data race analysis. In *Proc. of ICFEM*, 2004.
24. S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *Proc. of PLDI*, 2007.
25. Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *Proc. of IPDPS*, 2004.
26. Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. In *ESEC/SIGSOFT FSE*, pages 205–214, 2007.
27. Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *Proc. of POPL*, pages 327–338, 2007.
28. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 1978.
29. F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based Bounded Model Checking for Software Verification. In *Proceedings of ISOLA*, 2004.
30. M. K. Ganai and A. Gupta. Completeness in SMT-based BMC for software programs. In *Proc. of DATE*, 2008.
31. Joint CAV/ISSTA Special Event. Specification, Verification, and Testing of Concurrent Software. *http://research.microsoft.com/ quadeer/cav-issta.htm*, 2004.
32. SRI. Yices: An SMT solver. *http://fm.csl.sri.com/yices*.
33. D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and verilog programs using bounded model checking. In *Proc. of DAC*, 2003.

The appendix should not be considered as a part of the submission.

Appendix: Proofs

A Proof of Correctness

We present the proof of soundness and completeness of our modeling approach. We start with a few definitions and notations.

Let there be n thread models with m shared variables. For each model LM_i, let c_i^k denote its local state at depth k, and $P_i^{0,D}$ denote the set of finite sequences of its local state transitions, $c_i^{0,d(i)} \equiv c_i^0, \cdots, c_i^{d(i)}$, allowed by the transition relation and concurrency modeling constraints added up to D for some $d(i) \leq D$, respectively. As a note, each model LM_i has different $d(i)$. To simplify notation, we use d instead of $d(i)$ without changing its meaning in the following.

Let the pair (c_i^k, c_i^{k+1}) denote the local state transition between the corresponding states. Since the model LM_i does not have self-loops, there is *either* a transition such that $c_i^{k+1} \neq c_i^k$, *or* the associated `assume` constraint (Eqn 12-13) does not hold. In the latter case, transitions from c_i^k are terminated, i.e., $k = d$. Let $g_{i1}^k, \cdots, g_{im}^k$, TK_i^k, $CS_{i1}^k \cdots CS_{in}^k$ represent current state values of the *localized* shared variables of the model LM_i in state c_i^k. Let $c_i^{k,l}$ denote the subsequence $c_i^k, \cdots, c_i^l$ of $c_i^{0,d}$.

Definition 1 *A subsequence $c_i^{k,l}$ is called an initial subsequence (IS) iff the following hold*

- $k = 0, l < d$
- $\forall_{0 \leq h \leq l} TK_i^h = 0, TK_i^{l+1} = 1$

Note, each $c_i^{0,d}$ has at most one *IS*.

Definition 2 *A subsequence $c_i^{k,l}$ is called an uninterrupted subsequence (US) iff the following holds*

- $\exists_{k \leq h \leq l} TK_i^k = \cdots = TK_i^h = 1, TK_i^{h+1} = \cdots TK_i^l = 0$
- $TK_i^{k-1} = 0$, *if* $k \neq 0$
- $TK_i^{l+1} = 1$, *if* $l \neq d$

A *US*, denoted as $c_i^{k,h,l}$, is said to be *complete iff* $h < l$; otherwise, it is said to be *incomplete*, i.e., $h = l$. Note, an *incomplete US* has $TK_i^k = 1$ in all states in the subsequence.

Using Definitions 1 and 2, we chop the sequence $c_i^{0,d}$ into subsequences of *IS* and *US*. We can easily show by virtue of construction that for a subsequence $c_i^{k,h,l}$ with $l \neq d$, there is another subsequence $c_i^{l+1,h',l'}$. These subsequences can be concatenated (denoted by ::), i.e., $c_i^{k,h,l} :: c_i^{l+1,h',l'}$, to obtain the sequence $c_i^{0,d}$. We say these subsequences follow *concatenation order*.

Example: We illustrate above definitions using an example shown in Figure 4. Let there be three threads P_1, P_2 and P_3 with local sequences $c_1^{0,8}$, $c_2^{0,8}$, and $c_3^{0,6}$, respectively. Let each box represent the local state c_i^k of the thread P_i at step k. The value $1/0$ inside the shaded/unshaded box denotes the token value as *seen* at the thread state. The subsequences *US* are shown by bold rectangle, and *IS* are shown with dash outlines. Note, all *US* subsequences are *complete* except one, i.e., $c_2^{6,8,8}$ indicating P_2 is the current holder of the token. We will show later that given these subsequences, there exists a unique total order on the subsequences, denoted as *IUS* that satisfies the concurrency constraints up to depth 8.

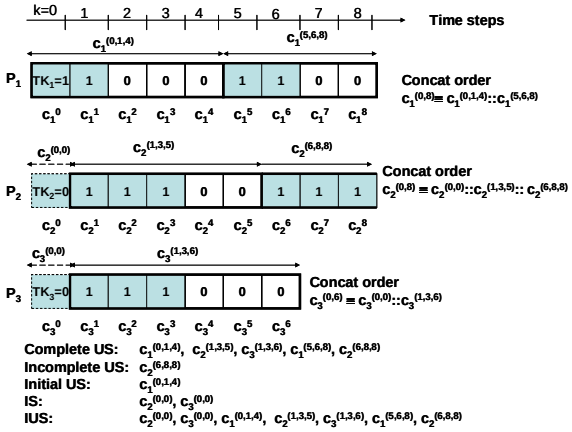


Fig. 4. Ilustrating various definition of subsequences.

Definition 3 *A local sequence $c_i^{0,d}$ can be one of the following types I, II, or III depending on the existence of IS and US in the sequence.*

- *Type I: IS does not exist and at least one US exists. One can concatenate multiple US such that $c_i^{0,d} \equiv c_i^{0,h,l} :: c_i^{l+1,h',l'} :: \cdots :: c_i^{l'',h'',d}$.*
- *Type II: IS exists (implying existence of at least one US) such that one can concatenate IS and US to obtain $c_i^{0,d}$, i.e., $c_i^{0,d} \equiv c_i^{0,l} :: c_i^{l+1,h',l'} :: \cdots :: c_i^{l'',h'',d}$.*
- *Type III: Neither an IS nor a US exists for the local sequence.*

One can show existence of such concatenations for types I and II using the definition of *US*. If *IS* exists, there exists at least one *US* in a sequence $c_i^{0,d}$ by construction.

Example: In Figure 4, $c_1^{0,8} \equiv c_1^{0,1,4} :: c_1^{5,6,8}$ is a *type I* sequence and $c_2^{0,8} \equiv c_2^{0,0} :: c_2^{1,3,5} :: c_2^{6,8,8}$ and $c_1^{0,6} \equiv c_3^{0,0} :: c_3^{1,3,6}$ are *type II* sequences.

Definition 4 *We say an ordered pair $(c_j^{k',h',l'}, c_i^{k,h,l})$ of US from different threads is valid iff $RW_{ij}^{(k-1)h'} = 1$, for $k > 0$, and $h' < l'$, i.e., token passing occurs.*

We have following *program order* property on each local sequence $c_i^{0,d}$.

Lemma 1. *Each local sequence $c_i^{0,d}$ follows program order.*

Proof: Note, the annotations (Section 5.1) do not change the relative order of the thread statements. Further, if all *Read-Write Synchronization Enabling Constraint* are disabled, i.e., $\forall_{j\neq i,0\leq h\leq d} RW_{ij}^{kh} = 0$, *No Sync Update Constraint* (Eqn 8-11) ensures that each thread model LM_i behaves identical to M_i. Therefore, the program order of memory access is also preserved in these sequences. □

Lemma 2. *Type III sequences, by Definition 3, cannot affect global state, and therefore, can be ignored while analyzing global state.*

Proof: Recall, type III sequence corresponds to a local execution from the initial state, where the write to shared variables have not been committed. Thus, such sequences cannot affect the global state, and therefore, can be ignored. Note, that the constraint *Single Control State Reachability Property Constraints* (Eqn 14) requires that $TK_i = 1$ when the local control state a is reachable, and therefore, witnesses are necessarily either *type I* or *III* sequences. □

Lemma 3. *Concatenation order of type I and II sequences follows program order and is unique for a local sequence.*

Proof: Follows from the definitions of *type I* and *II* sequences and Lemma 1. □

Lemma 4. *For a valid ordered pair $(c_j^{k',h',l'}, c_i^{k,h,l})$, the following hold:*

- $\forall_{q\neq i}^{n} CS_{qi}^{k} = CS_{qj}^{h'}$
- $CS_{ii}^{k} = CS_{ij}^{h'} + 1$
- $\forall_{p=1}^{m} g_{pi}^{k} = g_{pj}^{h'}$

Proof: For a *valid ordered pair*, $RW_{ij}^{(k-1)h'} = 1$ holds. Hence, from Eqn 1,$CS_{ii}^{k-1} = CS_{ij}^{h'}$. The proof follows from *Read-Write Synchronization Update Constraints* (Eqn 4 and 6). □

Lemma 5. *For an IS or US $c_i^{k,h,l}$ the following hold:*

- $\forall_{q=1}^{n} CS_{qi}^{k} = CS_{qi}^{l}$

Proof: Since in *IS* and *US*, there is no transition of token value from 0 to 1, $RC_i^k = \cdots = RC_i^l = 0$. Applying *No Sync Update Constraint* (Eqn 10), the proof follows. □

Lemma 6. *Given a concatenation, $c_i^{k,h,l} :: \cdots :: c_i^{k',h',l'}$, the following hold:*

- $CS_{ii}^{k} \leq CS_{ii}^{k'}$

Proof: Note, $k < k'$ as concatenation order is in *program order* (Lemma 3). Further, since the only update constraints (Eqn 6 and 10) do not decrease $CS_{ii}^{k'}$ for $k' > k$, the proof follows. □

Lemma 7. *There is a unique US $c_i^{k,h,l}$ with $k = 0$ for some i.*

Proof: It follows from the *Single Token Constraint* (Eqn 15), that allows only one model, say i, to have token initially. □

Lemma 8. *For an US $c_i^{k,h,l}$ with $k \neq 0$, there exists a unique complete US $c_j^{k',h',l'}$ with $h' < l'$ such that the ordered pair $(c_j^{k',h',l'}, c_i^{k,h,l})$ is valid.*

Proof: From the Definition 2, $c_i^{k,h,l}$ with $k \neq 0$ implies $TK_i^{k-1} \neq TK_i^k$; which in turn implies $RC_i^{k-1} \neq 0$, using Eqn 9. W.l.o.g. (Without loss of generality), let $RC_i^{k-1} = a_j^{h'}$, which with Eqn 2 implies $RW_{ij}^{(k-1)h'} = 1$. Thus, as per Definition 4, the ordered pair $(c_j^{k',h',l'}, c_i^{k,h,l})$ is valid. □

Lemma 9. *For a complete US $c_j^{(k',h',l')}$ with $h' < l'$, there exists a unique US $c_i^{k,h,l}$ such that the ordered pair $(c_j^{k',h',l'}, c_i^{k,h,l})$ is valid.*

Proof: From the Definition 2, $c_j^{k',h',l'}$ with $h' < l'$ implies $TK_j^{h'+1} = 0$ and $TK_j^{h'} = 1$; which in turn implies $WC_j^{h'} \neq 0$, using Eqn 11. W.l.o.g., let $WC_j^{h'} = a_i^k$, which with Eqn 3 implies $RW_{ij}^{(k-1)h'} = 1$. Thus, as per Definition 4, the ordered pair $(c_j^{k',h',l'}, c_i^{k,h,l})$ is valid. □

Lemma 10. *A global sequence of US (from different threads) can be constructed so that every consecutive pair of US is a valid ordered pair as per Definition 4. Further, such a sequence starts with a unique $c_i^{0,h,l}$ for some i.*

Proof: As per Lemma 7, there exists a unique *US* $c_i^{0,h,l}$. We start a sequence with the unique *US*. If $h = l$, we stop; otherwise using Lemma 9, we obtain a next *US* in the sequence, say, $c_j^{k',h',l'}$ and continue until $h' = l'$. Note, we stop at an *incomplete US* as per Definition 2. □

In the following Lemma 11- 13, we prove that the global sequence of *US*, so constructed, has the following properties: *finite*, *cycle-free*, *unique*, and a *total ordered*-interleaving sequence of *US* in the *program order*.

Lemma 11. *The global sequence of US (from different thread) is finite, has no cycle and always ends with an incomplete US.*

Proof: As the number of *US* is finite for a local sequence $c_i^{0,d}$, and number of threads are finite, i.e., n, we only have to show that there is no cycle in the sequence of *US*. To prove by contradiction, we assume there exists a cycle, i.e., $c_i^{k,h,l}, \cdots, c_j^{k',h',l'}, c_i^{k,h,l}$. W.l.o.g., let there be no nested cycle in this cycle. Applying Lemma 4 on consecutive *US* in the sequence $c_i^{k,h,l}, \cdots, c_j^{k',h',l'}$, one can show that $CS_{ii}^{k} \leq CS_{ij}^{k'}$. Again, applying Lemma 4 and 5 on $(c_j^{k',h',l'}, c_i^{k,h,l})$, we show that $CS_{ij}^{k'} = CS_{ij}^{l'}$ and $CS_{ij}^{l'} + 1 = CS_{ii}^{k}$. Thus, we obtain a contradiction $CS_{ii}^{k} < CS_{ii}^{k}$. Therefore, we conclude that the sequence is finite with no cycle and always ends with an *incomplete US* as per construction using Lemma 10. □

Lemma 12. *The global sequence of US is unique.*

Proof: We show this by induction on the subsequence length.

- *Basis*: Sequence with one *US*.
  This is trivially true as there is a unique *US* $c_i^{0,h,l}$ as per Lemma 7.
- *Induction:* Given a unique global subsequence of *US* of length t starting with a unique *US* and ending with a *complete US*, we show that the subsequence of length $t+1$ is also unique. Note, if the last *US* is *incomplete*, the induction step is trivially true since there can be only one *incomplete* US in a global sequence of *US*.
  Let the t^{th} *US* in the global sequence be $c_j^{k',h',l'}$. We use Lemma 9 to obtain a unique next *US*. Thus, the subsequence of length $t+1$ is also unique. □

Lemma 13. *The global sequence of US is a total order-ed sequence and maintains program order.*

Proof: We show that the global sequence includes all the local sequences of *US* in the concatenation order.

- *Inclusion*: To show total inclusion, we show a given *US* $c_i^{k,h,l}$, is not left out, i.e., it is always included in the global sequence. If $k = 0$, we are done trivially, as per Lemma 7 this *US* is unique, and the global sequence starts with this *US* (Lemma 10); otherwise, we use Lemma 8 repeatedly, finitely often as per Lemma 11, and obtain another *US* $c_j^{k',h',l'}$ preceding it, until $k' = 0$. As per Lemma 12, there is a unique global sequence, and the sequence $c_j^{0,h',l'} \cdots c_i^{k,h,l}$ is a subsequence of the global sequence.
- *Concatenation order*: We show this by contradiction. Given a concatenated pair of *US* $c_i^{k,h,l} :: c_i^{l+1,h',l'}$, we assume that in the global sequence, we have a subsequence $c_i^{l+1,h',l'} \cdots c_i^{k,h,l}$, where the pair does not appear in *concatenation order*. Applying Lemma 4 on consecutive *US* in the global sequence $c_i^{l+1,h',l'}, \cdots, c_i^{k,h,l}$, one can show that $CS_{ii}^{l+1} < CS_{ii}^{k}$. However, using Lemma 6, we obtain, $CS_{ii}^{k} \leq CS_{ii}^{l+1}$; a contradiction.

Thus, *inclusion* and *concatenation order* rules give a total interleaving of the local sequences of *US* and using Lemma 3 we also obtain that the global sequence of *US* maintains the program order. □

We construct a global sequence, denoted as *IUS*, by sequencing all *IS* in some total order before the global sequence of *US*.
Example: In Figure 4, the global sequence $c_2^{0,0}$, $c_3^{0,0}$, $c_1^{0,1,4}$, $c_2^{1,3,5}$, $c_3^{1,3,6}$, $c_1^{5,6,8}$, $c_2^{6,8,8}$ represents an *IUS*.

Now, we prove that *IUS* holds the sequential consistency and synchronization semantics.

Lemma 14. *The sequence IUS maintains program order.*

Proof: This follows trivially from Lemma 1 and 13. □

Lemma 15. *The sequence IUS has a total order of memory access.*

Proof: As per Lemma 13, the global sequence of *US* is a total ordered sequence. Further, since we choose a total order on *IS*, the proof follows. □

Lemma 16. *Read value rule is maintained in IUS, i.e., a read access of a shared variable observes the effect of the last write access to the same variable in the sequence IUS.*

Proof: In each *US*, *Read Value Rule* is observed due to program semantics as per Lemma 1. Between consecutive *US* (from different threads), *Read Value Rule* is maintained by Lemma 4. In each *IS*, and in the starting unique *US*, the read of shared variable *sees* the initial state of the shared memory, which is the same for all thread models. □

Lemma 17. *Shared accesses by different threads in matched lock/unlock operations are mutually exclusive in IUS.*

Proof: It is is sufficient to show that the mutually exclusive shared access does not occur twice in the *IUS* sequence with same lock variable being set to 1 without setting it to 0 in between.

W.l.o.g., assume a shared variable is accessed simultaneously in a state a_i and b_j that are mutually exclusive using lock lk in the global sequence of *IUS*. Let $c_i^{k,h,l}$ denote a *US* such that lk_i^l has value 1. Let $c_j^{k',h',l'}$ denote *US* such that for some $k' \leq t \leq h'$, $lk_{1,j}^{t+1}$ gets value 1 in control state c_j^{t+1} due to an update from a lock control state c_j^t. Let $c_i^{k,h,l}$ occur before $c_j^{k',h',l'}$ in the *IUS*. We further assume that there exists no other sequence corresponding to lock/unlock of lock lk. A necessary condition for exclusive violation to occur, is to show that there exists in the *IUS* a transition from lock acquisition control state, i.e., c_j^t with $t < l'$. Note, $lk_j^{k'} = lk_i^l = 1$ as *Read Value Rule* is maintained (Lemma 16), which the *Mutual Exclusion Constraint* (Eqn 12), $B_{c_j^t} \implies (lk_j^t = 0)$, there is a clear conflict. Hence, such a global sequence *IUS* does not exist where lock variable is set to 1 twice without setting it to 0. Similarly, we can argue for unlock case, where lock is set to 0 twice without setting it to 1. □
We now present the following Theorem validating the soundness and completeness of our lazy modeling approach for BMC.

Theorem 1 *The lazy modeling constraints allow only those traces that respect the sequential consistency of memory model and synchronization schematics up to the bound D, i.e., our modeling is complete.*

Further, if there exists witness for a reachability property, such that the global trace length is $\leq n \cdot D$ and each local trace length $\leq D$, there exists an equivalent trace allowed by our model, i.e., IUS corresponding to the witness trace. In other words, our modeling is sound in that it does not miss any witness up to these bounds.

Proof:

- *Completeness:* Recall, for a given BMC bound D, $P_i^{0,D}$ denotes the set of sequences of local state transitions up to the bound. Given a tuple of local state transitions $(c_1^{0,d(1)}, \cdots, c_n^{0,d(n)}) \in P_1^{0,D} \times \cdots \times P_n^{0,D}$, we construct a unique global sequence *IUS*, that capture the requirements of sequential consistency of memory model and synchronization semantics up to the bound D (Lemmas 14-17). Thus, the *IUS* represents a valid trace in the concurrent system w.r.t sequential consistency and synchronization semantics.
- *Soundness:* We show that for a given witness trace that follows sequential consistency and synchronization semantics, there exists a tuple of local state transitions $(c_1^{0,d(1)}, \cdots, c_n^{0,d(n)})$. Note, from this tuple we construct a unique *IUS* (using Lemma 10 and 12).
  Let us take a total ordered t^{th}-length memory trace $A_i^0 \cdots A_j^t$ that is sequentially consistent and follows synchronization semantics and is a witness to the given reachability property, where A_j^t represents a shared access by thread P_j at t^{th} step. We chop this trace into subsequences of consecutive accesses of a thread P_i, i.e., $A_i^k, \cdots, A_i^l$. We then append a NOP operation N_i^{l+1} after A_i^l. We obtain an equivalent *US* from each such subsequence by associating a token TK_i such that $TK_i^k = \cdots = TK_i^h = 1$, and $TK_i^{h+1} = \cdots = TK_i^{l+1} = 0$, where we choose h from one of the following:
  - $h = k$, if there is no write in the subsequence
  - $h = r$, if subsequence is not the last subsequence, and A_i^r is the last write in the subsequence, $k \leq r \leq l$
  - $h = l$, if the subsequence is the last sequence (and there is a write operation in the subsequence).

  We then obtain a local sequence of *US* for each thread P_i, by taking the *US* corresponding to the thread P_i and concatenate them in the same order as they appear in the witness. Note, each local sequence follows the program order (guaranteed by the witness), which can be represented by the tuple of local state transition $c_1^{0,d(1)}, \cdots, c_n^{0,d(n)}$. Note, each local sequence is a *type I* or *II* sequences by Definition 3. □

Comments: The maximum length of an *IUS* can be $n \cdot D$ with the restriction that each local sequence, combining *IS* and *US*, is of maximum length D. Note, the construction of *IUS* is only conceptual. Indeed, we use BMC to explore all such possible *IUS* to find a violation or a witness for a reachability property.