Parameterized Model Checking of Fine Grained Concurrency

Divjyot Sethi[1], Muralidhar Talupur[2], Daniel Schwartz-Narbonne[1], and Sharad Malik[1]

[1] Princeton University
[2] Strategic CAD Labs, Intel Corporation

Abstract. Concurrent data structures are provided in libraries such as Intel Thread Building Blocks and Java.util.concurrent to enable efficient implementation of multi-threaded programs. Their efficiency is achieved by using fine grained synchronization which creates less constrained interaction between the threads. This leads to a large number of possible interleavings and makes concurrent data structures hard to verify. In this paper, we describe our key insights from Murphi based parameterized model checking of these data structures. In particular, we describe the first model checking based framework to handle an unbounded number of threads for these data structures. This framework uses the CMP (CoMPositional) method which has been used in verifying cache coherence protocols. The CMP method requires the user to supply lemmas for abstraction refinement. A further contribution of our work is to show how a significant subset of these lemmas can be generated automatically.

1 Introduction

Concurrent data structures are provided in libraries such as Intel Thread Building Blocks [18] and Java.util.concurrent (JSR-166 [12]) to enable efficient implementation of multi-threaded programs. The efficiency of these data structures is achieved by either making them lock free or by using fine grained synchronization between the threads, i.e., by locking parts of the data structure instead of locking the entire data structure while making updates. This results in less constrained interaction between the threads and thus increases the number of possible interleavings. The large number of possible interleavings make these data structures highly error prone, as exemplified by bugs witnessed in published algorithms [16].

Given the large number of possible interleavings, model checking these data structures is non-trivial. Consequently, existing model checking based efforts [3, 23–25] address the verification of concurrent data structures for only a small number of threads. This motivated our work on model checking these data structures for an unbounded number of threads by applying parameterized model checking techniques. In particular, we leverage the CMP (CoMPositional) method [4], a parameterized verification technique which enables verifying correctness of a system with unbounded number of threads by constructing an abstract model

which consists of only a small number of threads and an environment thread. This technique has been successfully used in verifying message passing systems like cache coherence protocols [17, 19]. The CMP Method is discussed below:

1.1 The CMP Method

The CMP method is used to verify symmetric parameterized systems. A symmetric parameterized system, $P(N)$, consists of N identical threads with ids $1..N$. The properties which we verify on these systems using the CMP method are of the form $\forall i \in [1..N].\Phi(i)$, where $\Phi(i)$ is a propositional logic formula on the variables of thread i and shared variables. The CMP method first constructs an abstract model which consists of one thread from the original system (say thread 1 since the system is symmetric) and an abstract thread (named *Other*) that over-approximates the remaining threads (the environment). Then, if the property $\Phi(1)$ holds on thread 1 in the abstract model, $\forall i \in [1..N].\Phi(i)$ holds for $P(N)$ by symmetry. The verification of the abstract model is done by using a model checker (in our case Murphi).

Since the constructed abstract model is an over approximation, the model checker may report spurious counterexamples. In order to remove these counterexamples, the user needs to refine the model by coming up with candidate lemmas which constrain the abstraction.

While the CMP method requires human guidance in the form of supplying candidate lemmas, it essentially uses a model checker as an assistant and transfers most of the proof burden to it. The key difference between the CMP method and theorem proving based techniques then is that the former leverages a model checker. Thus, the lemmas supplied to the CMP method need not be inductive. Theorem proving techniques on the other hand require that all the invariants together with the properties under check be inductive. Further, in practice far fewer lemmas have to be supplied while using the CMP method [19]. The CMP method has been successfully used to verify large scale industrial cache coherence protocols which are significantly larger than the distributed protocols handled by pure theorem proving methods [17].

Mining candidate lemmas To reduce the manual effort of supplying candidate lemmas on the user and to accelerate the proof convergence, we show how several of the candidate lemmas required to refine the abstract model can be mined from the execution trace of a single thread. We use Daikon [6] for this purpose. Daikon takes in program execution traces and infers a collection of candidate invariants for the program that are based on invariant templates. Note that while the candidate lemmas generated using Daikon might be false, the CMP method also checks the lemmas used. Any false candidates are eliminated without affecting the soundness of the proof [11]. It is this feature of the CMP method that allows us to use an off-the-shelf invariant generator without compromising soundness.

1.2 Framework Description

The experimental framework which we use is shown in Figure 1. The input data structure encoded in Murphi (discussed in Section 2) is first augmented by Daikon generated candidate lemmas. These candidate lemmas are generated by analyzing the trace of a single thread (discussed in Section 4). The augmented model then enters the CMP loop (discussed in Section 3). In the CMP loop, the model is automatically abstracted by using the tool *Abster* [19]. Then, in case a proof of correctness or a real counterexample is found, the loop finishes. In case a spurious counterexample is found, the model is further refined by first adding more lemmas to the original program and then re-abstracting it, as shown in the figure. The new abstract model after re-abstraction is more refined, as discussed in Section 3. In practice the number of extra lemmas required to be added were reduced due to the initial addition of Daikon generated lemmas (detailed in Section 5).

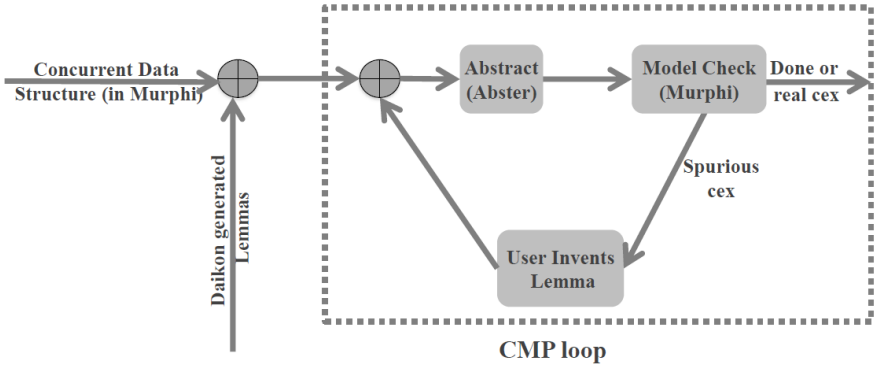

Fig. 1: The Experimental Framework

1.3 Main Contributions

Through this case study of applying the CMP method to verify concurrent data structures, we make the following contributions:

- *Unbounded threads*: We show how concurrent list based set data structures with an unbounded number of threads accessing them can be model checked by the CMP method. Further, these data structures allocate memory dynamically which leads to subtle issues causing a memory usage blowup as the number of threads accessing the data structure increase. We show how the CMP method addresses this.
- *Mining candidate lemmas*: We show how the burden of supplying candidate lemmas to the CMP method can be reduced by mining a significant subset

of these lemmas automatically by analyzing the execution trace of a single thread. We use the invariant generator Daikon for this.
- *Experiments*: We describe our insights in verifying several well known concurrent data structures: *Fine-grained*, *Lock Free* [7,15], *Optimistic* and *Lazy* [8] list based sets. We were also able to find the error in a known buggy version of the *Lock Free* list based set algorithm. Finally, we also describe the particular candidate lemmas we added in detail.

Limitations (1)We assume that the size of the data structure is bounded. We believe that this CMP method based approach, which handles unbounded number of threads, can be extended to handle unbounded sized data structures as well, by using an appropriate finite abstraction for the data structure. (2) Daikon generates invariants by using invariant templates to determine the structure of the invariant. These templates are provided by the user. While user-supplied, these templates were the same for the class of algorithms we verified, thus reducing the burden of creating them.

1.4 Related Work

Verification of concurrent data structures has received significant attention lately [1–3, 5, 20, 21, 23–25]. These verification efforts can be broadly classified as manual, model checking based and separation logic based, as detailed below:

Manual: Vafeiadis et al. [21] verified concurrent data structures using a mechanical proof assistant. They specified the interference between concurrent threads using rely and guarantee conditions. They were not able to handle the *Lazy* algorithm. In [5] Colvin et al. proposed using a forward-backward simulation based approach for the *Lazy* algorithm. These approaches allow both an arbitrary number of accessing threads and an arbitrary number of elements but they are manual effort intensive. Our approach, on the other hand, aims for increased automation by leveraging model checkers. Our framework is also able to handle the *Lazy* algorithm.

Model checking based: Vechev et al. [23] describe their experience with verifying Linearizability using the SPIN model checker. While this approach handles algorithms with unknown linearization points, they do not scale to more than a small number of threads and a small list size. Similarly, Zhang et al. [24, 25] and Liu et al. [13] used model checking to verify concurrent data structures. They used a naive refinement based approach which does not scale to more than a small number of threads and list size. A different approach is taken by Alur et al. [3]: they treat a list as a string and the thread as an automaton and then derive conditions on the automaton to prove decidability of Linearizability for lists of arbitrary length. The key focus of their work is on decidability instead of scalability; hence they too do not scale to more than 2-3 threads. In contrast, our method is able to handle an unbounded number of threads.

Separation logic based: Verification approaches have been developed for programs specified in RGSep, a logic which combines separation logic with rely-guarantee reasoning [2, 20, 22]. While their approach is automatic for a subset of RGSep, the designer still needs to specify concurrent actions which model the inter-thread interference, just like in rely-guarantee reasoning, for proving assertions. Further, this approach requires the designer to have an understanding of RGSep. In contrast, our tool requires the user to specify candidate lemmas, which are propositional logic formulas. Further, the CMP method automatically checks the candidate lemmas as well, so any false lemmas will be weeded out.

2 Encoding Concurrent Data Structures as Paramaterized Systems

In this section, we describe how to encode a concurrent data structure as a Murphi model.

We model a concurrent data structure as a program P with N identical threads with ids $1..N$, where N is arbitrary but fixed. Each thread i has a set of l finite local variables L_i. There is also a set of s finite shared variables S.

The action (in this paper we assume that every line in the pseudocode corresponds to an action) of each thread i is modeled as a *rule* of the form $\rho \Rightarrow a$, where ρ is the guard over variables in $L_i \cup S$ and the action a is a set of assignments to variables in $L_i \cup S$. Further, instead of writing out rules that are identical modulo thread ids or list node ids, we use quantified rules (called *Rulesets* in Murphi language). For instance, we can write
Ruleset $i : Thread; p : node$
Rule $\rho \Rightarrow$ Begin
  a;
End;

to represent the set of rules obtained by plugging in all possible values of i and p in the rule $\rho \Rightarrow a$.

The semantics of this guarded-command format are straight-forward: at every time step, the guards of all the rules are checked to see which of the rules are enabled. The Murphi model checker then non-deterministically picks an enabled rule and fires it. Since each thread i has a separate set of rules, the multi-threaded execution follows interleaving semantics.

2.1 Running Example: *Lazy* list based concurrent set

We demonstrate our approach on a concurrent data structure implementing a standard set interface, with *Add*, *Remove* and *Contains* methods. These methods have semantics expected of a standard sequential set *SeqSet* (Figure 2).

Our example follows the *Lazy* list based concurrent set data structure (or *Lazy* set for brevity) defined in [21]. The underlying representation of the *Lazy* set is a linked list, which stores the elements of the set in a (strictly) increasing order with each *node* of the linked list having the following fields: 1) a *key* holding

$$
\begin{aligned}
SeqContains(e):\ & \{SeqResult := e \in SeqSet; \\
& \quad return\ SeqResult; \} \\
SeqAdd(e):\ & \{SeqResult := e \notin SeqSet; \\
& \quad SeqSet := SeqSet \cup e \\
& \quad return\ SeqResult; \} \\
SeqRemove(e):\ & \{SeqResult := e \in SeqSet; \\
& \quad SeqSet := SeqSet \setminus e \\
& \quad return\ SeqResult; \}
\end{aligned}
$$

Fig. 2: Methods of the sequential set $SeqSet$.

the element's value, 2) a *next* pointer for accessing the next node in the list 3) a *marked* bit to indicate if the node is a part of the set or not, and 4) a *lock* field representing whether that node is currently locked. In addition, there are two special nodes, the first node *Head* and the last node *Tail*, that can neither be added nor be removed. All the nodes that are reachable from *Head* and do not have the *marked* bit set are considered part of the set, denoted by $ConcSet$.

As shown in Figure 3, the methods of this data structure traverse the linked list using pointers *curr* and *pred* which are local to the methods. For the *Add* method, the local pointer *entry* is used to store the address of the newly allocated node being added to the list. For the *Remove* method, this pointer is used to store the value of the *next* field of the *curr* pointer while removing the node pointed by *curr*.

Verifying Correctness: *Linearizability* [10] is the widely accepted correctness criterion for concurrent data structures. Intuitively, Linearizability implies that execution of every access method for the concurrent data structure appears to occur at some point, the *linearization point*, between the invocation and the response of the method. The linearization points for the *Lazy* set are marked with a $*$ in Figure 3.

As described in [21], Linearizability can be proved by using a refinement based approach. Thus, the Linearizability of the algorithm is established by comparing the results of the concurrent access methods against the results of the access methods of the sequential set, $SeqSet$, with access methods shown in Figure 2. This comparison is done by embedding the calls to $SeqSet$ in the implementation at the linearization point. For instance, the linearization point for *Remove* in case the call is successful is marked with [**SeqRemove(key)*] on Line 3 in Figure 3d. Similarly, [**SeqRemove(key)*] on Line 8 denotes the linearization point in the failing case. If the results returned by concurrent methods and the embedded sequential methods match, the concurrent data structure is Linearizable. Formally, for each $Method \in \{Add, Remove, Contains\}$, we check that $Method(key) \Leftrightarrow SeqMethod(key)$.

Further, to check if $ConcSet$ refines $SeqSet$, we also check if their contents match; i.e. $\forall v.v \in SeqSet \Leftrightarrow v \in ConcSet$ holds at all times.

The *Contains* method needs special treatment, because its linearization point depends on the return value of the method. If the element is found in the set,

```
1: node curr := Head;
2: while(curr.key < key)
3:     curr := curr.next;
4: if curr.key = key &
5:     !curr.marked then
           [*SeqContains(key)]
6:     return true;
7: else
8:     return false;
```

(a) Method *Contains(key)*

```
1: node pred, curr := locate(key);
2: if(curr.key != key) then
3:     entry := new node();
4:     entry.key := key;
5:     entry.next := curr;
6:     pred.next := entry;
           [*SeqAdd(key)]
7:     result := true;
8: else
9:     result := false;
           [*SeqAdd(key)]
10: pred.unlock();
11: curr.unlock();
12: return result;
```

(b) Method *Add(key)*

```
1: while(true)
2:     pred := Head;
3:     curr := pred.next;
4:     while(curr.key < key)
5:         pred := curr;
6:         curr := curr.next;
7:     pred.lock();
8:     curr.lock();
9:     if !pred.marked &
10:      !curr.marked &
11:      pred.next = curr then
12:        return pred, curr;
13:    else
14:        pred.unlock();
15:        curr.unlock();
```

(c) Method *locate(key)*

```
1: node pred, curr := locate(key);
2: if(curr.key = key) then
3:     curr.marked := true;
           [*SeqRemove(key)]
4:     entry := curr.next;
5:     pred.next := entry;
6:     result := true;
7: else
8:     result := false;
           [*SeqRemove(key)]
9: pred.unlock();
10: curr.unlock();
11: return result;
```

(d) Method *Remove(key)*

Fig. 3: Pseudo-code for linked list based *Lazy* set algorithm. The linearization points are marked with a $*$.

the *Contains* method is linearized at Line 5 in Figure 3c. If the item is not in the set, however, the linearization point is dependent on the history and cannot be statically determined. To prove Linearizability for this case, we proceed as follows: for every invocation of *Contains* such that it returns *false*, we need to determine a point between the invocation and response points such that a call to *SeqContains* also returns *false*. In other words, find a point where the abstract set *SeqSet* does not contain the element sought, say v. If for some invocation, no such point is found then the *Contains* method is not Linearizable.

To check if such a point exists is simple: we keep a history variable aux_v that is set to *false* initially when *Contains(v)* is invoked. If at any point the *SeqSet* does not contain v then aux_v is updated to $true$. Then, when *Contains(v)* returns *false* we just have to check if aux_v is $true$ or not. Thus, by checking the invariant $(Contains(v) = false) \Rightarrow aux_v$, we can determine the Linearizability of *Contains*.

2.2 Encoding the *Lazy* set in Murphi

The state space of the *Lazy* set can be encoded in Murphi as follows. The shared state consists of the linked list and the *SeqSet* which can be encoded as the shared variables S. In particular, the linked list is stored as an array $list$ of *nodes* (which are essentially structs). Each *node* holds records with fields $next$, $marked$, key and $lock$, which correspond to the fields of the *node* discussed in Section 2.1. We encode locks such that they store both their state (locked/unlocked) as well as the thread id of the lock owner.

The local state of each thread which needs to be encoded consists of the variables *curr*, *pred* and *entry*, as shown in Figure 3. We encode these local variables as arrays with domain as thread id, $[1..N]$. Thus, $curr[i]$, $pred[i]$ and $entry[i]$ store the corresponding local variables for thread i. Then, $list[curr[i]].key$ (or $curr[i].key$ for brevity) is the value of the key of the node pointed by the *curr* pointer in thread i. Further, in order to do the refinement proof, we need a mechanism to store the result of the call to the access method of *SeqSet* embedded at the linearization point. We store this in $Seqresult[i]$ for thread i, as shown in Figure 4.

Figure 4 shows the Murphi encoding of the *Remove* method for thread i. (Due to space restrictions, we do not show the encoding of the entire algorithm.) Each rule in the figure corresponds to a statement of the *Remove* method in Figure 3d. In order to enforce the sequential order of the statements, we added the local variable $pc[i]$. This variable serves as a local program counter and is used in the guards of the rules. The code executes in sequential order starting from top left rule ($pc = 0$) to the bottom right rule ($pc = 10$).

Since the concurrent algorithm needs to be proved correct for an arbitrary set of calls to its interface functions, we encode the threads accessing the data structure such that it non-determinstically selects a key, and a method (*Add, Remove* or *Contains*) for execution. It then executes the method on the key and then restarts after finishing execution.

```
Ruleset i : Thread
Rule pc[i] = 0 ⇒ Begin
   pred[i], curr[i] :=
   locate(key); [1]
   pc[i] + +;
End;

Ruleset i : Thread
Rule pc[i] = 1 ⇒ Begin
   if (curr[i].key = key)
   then pc[i] + +;
   else pc[i] = 6;
End;

Ruleset i : Thread; Ncurr : node;
     Nkey : KeyType
Rule pc[i] = 2 & Ncurr = curr[i]
  & Nkey = key[i] ⇒ Begin
   Ncurr.marked := true;
   pc[i] + +;
   Seqresult[i] :=
   SeqRemove(Nkey);
End;

Ruleset i : Thread; Ncurr : node
Rule pc[i] = 3 ⇒ Begin
   entry[i] := Ncurr.next; pc[i] + +;
End;

Ruleset i : Thread; Npred : node
Rule pc[i] = 4 & Npred = pred[i] ⇒
Begin
   Npred.next := entry; pc[i] + +;
End;

Ruleset i : Thread
Rule pc[i] = 5 ⇒ Begin
   result[i] := true;
   pc[i] := pc[i] + 2;
End;

Ruleset i : Thread
Rule pc[i] = 6 ⇒ Begin
   [else] pc[i] + +;
End;

Ruleset i : Thread
Rule pc[i] = 7 ⇒ Begin
   result[i] := false; pc[i] + +;
End;

Ruleset i : Thread; Npred : node;
Rule pc[i] = 8 ⇒ Begin
   pred[i].unlock(i); pc[i] + +;
End;

Ruleset i : Thread; Ncurr : node
Rule pc[i] = 9 ⇒ Begin
   curr[i].unlock(i); pc[i] + +;
End;

Ruleset i : Thread
Rule pc[i] = 10 ⇒ Begin
   return result[i];
End;
```

Fig. 4: Murphi model for method *Remove*(i) for thread i: each rule corresponds to a line in the *Remove* method shown in Figure 3.

[1] The method locate is shown in pseudocode format in Figure 3.

Memory Allocation/Garbage Collection The *Lazy* set allocates new nodes when it adds elements and removes references when it removes elements. This requires allocation and potential deallocation of memory.

It is non-trivial to manually manage the memory for this algorithm by deleting nodes which are removed. This is because another thread might be accessing the node removed by the *Remove* method, making deletion of the node at the time of removal incorrect [23]. This makes the implementation of a garbage collector essential.

We encode the memory as an array of shared variables in the Murphi model. Since Murphi does not provide support for dynamic memory management, we implemented the garbage collector as a function which is called when a new list node is allocated. The function returns *true* if it finds a new node which has no references to it. Otherwise, it returns an error.

Specifying Linearizability As discussed in Section 2.1, we need to check $Method(key) \Leftrightarrow SeqMethod(key)$, where $Method$ is in $\{Add, Remove, Contains\}$. Then, the condition to check Linearizability in the Murphi encoding is: $\forall i \in [1..N].Seqresult[i] = result[i]$.

The above condition involves variables of a single thread i, universally quantified over all threads. Similarly, the other properties which need to be checked can be specified either as a formula of the above form or as a propositional logic formula on the shared state. As discussed in Section 3, this structure is essential for constructing the CMP abstraction.

3 Parameterized Model Checking using the CMP Method

In this section we describe the CMP method and then show how it applies to the *Lazy* set.

Given a symmetric parameterized system $P(N)$ with threads $1..N$ and a property involving a small constant number c of threads, the CMP method first constructs a finite abstract model that can be checked using a model checker. The abstraction typically used, called *data type reduction* [14], keeps c threads unchanged and creates one abstract thread *Other* representing threads $[c+1..N]$. The abstraction operation involved in constructing *Other* is syntactic and fast: it essentially involves throwing away all the state variables of threads $[c+1..N]$ and over-approximating expressions involving them. To do the abstraction syntactically, the first step is to reduce the domains of all state variables holding array indices from $[1..N]$ to $[1..c]$. Next, in the Murphi program, thread indices $c+1..N$ are replaced with o (which is the id we use for the *Other* thread) wherever they appear. For instance an expression $pc[i] = 8$ for $i \in [c+1..N]$ reduces to $pc[o] = 8$. Consequently, since the *Other* thread has no state, $pc[o]$ becomes meaningless and so $pc[o] = 8$ is replaced by *true* or *false* (depending on which replacement leads to an over-abstraction). This syntactic abstraction operation is implemented in *Abster* [19].

Since the abstract thread *Other* is completely unconstrained, model checking the abstract model may lead to spurious counterexamples. If this happens, the user refines the system by adding candidate lemmas which are conjoined to the guards of the rules. Formally, suppose that the candidate lemma L is used. Now consider a rule r of the program P defined as: $\rho \Rightarrow a$. Then, refining P with L involves changing this rule to $\rho \wedge L \Rightarrow a$ and then re-abstracting the new program with the new rule obtained. This abstraction refinement procedure is also explained through an example in Section 3.1.

If this refined system passes the model checker, the property is proven. If, on the other hand, there is another counterexample for the refined system, the user must distinguish between three possible cases by examining the counterexample. 1) The counterexample is valid. 2) The counterexample is not valid and the candidate lemma is correct, in which case further refinement is required. 3) The counterexample is not valid and a candidate lemma is incorrect, in which case the incorrect candidate lemma must be removed or modified.

A useful heuristic for determining if the counterexample is due to an incorrect candidate lemma is that it fails for the (concrete) system with a small number of threads (one or two for the data structures we verified). This can either be detected by manual examination of the system (for concurrent data structures we verified, manual examination was sufficient due to relatively short counterexamples). Or, this may also be done by model checking the candidate lemma on the system for a small number of threads. (Before manual examination, we model checked all the lemmas for a model with a single thread.)

One key advantage of the CMP method is that the candidate lemmas used for strengthening are also checked during the process and any false candidate lemma that is added will be detected by the model checker [11]. This guarantees that the refinement step does not affect the soundness of the proof.

3.1 Verifying the *Lazy* set algorithm

Linearizability for the *Lazy* set can be specified as a propositional formula over a single thread (and so has $c = 1$), as discussed in Section 2.2. The abstracted system therefore contains the rules from the concrete thread shown in Figure 4 as well as the rules from the abstract thread as shown in Figure 5.

To understand the abstraction in detail, consider the rule from the concrete thread i for $pc = 4$:

Ruleset $i : Thread$; $Npred, Nentry : node$;
Rule $pc[i] = 4$ & $NPred = pred[i]$ & $Nentry = entry[i] \Rightarrow$ Begin
  $Npred.next := Nentry$; $pc[i] + +$;
End;

For thread 1 this ruleset will be preserved as it is. But for the *Other* thread the ruleset will be abstracted to

Ruleset $Npred$, $Nentry : node$;
Rule $true \Rightarrow$ Begin

% For pc = 2
Ruleset $Ncurr : node$
Rule $true \Rightarrow$ Begin
  $Ncurr.marked := true$;
  SeqRemove(Nkey);
End;

% For pc = 4
Ruleset $Npred : node$;
  $Nentry : node$
Rule $true \Rightarrow$ Begin
  $Npred.next := Nentry$;
End;

% For pc = 8
Ruleset $Npred : node$
Rule $true \Rightarrow$ Begin
  $Npred.unlock(o)$;
End;

% For pc = 9
Ruleset $Ncurr : node$
Rule $true \Rightarrow$ Begin
  $Ncurr.unlock(o)$;
End;

% For pc = 0,1,3,5,6,7,10
Ruleset
Rule $true \Rightarrow$ Begin
  no-op
End;

Fig. 5: Abstracted thread for *Remove* method.

$Npred.next := Nentry$;
End;

This is because expressions $pc[i] = 4$, $Npred = pred[i].next$ and $Nentry = entry[i]$ will be conservatively over-approximated to $true$. Similarly, the assignment $pc[i]++$ becomes a *no-op*. Since i no longer appears in the body of the ruleset, it can be dropped from the ruleset quantifiers as well. A model constructed in this way is an abstraction of the original system.

The above abstract rule is highly unconstrained and so may lead to spurious counterexamples. It can, for example, pick an arbitrary node in the linked list and set its *next* field to another arbitrary node. The next step in the CMP method is refinement of the abstract model using lemmas. One lemma that the user can add is that $entry[i]$ is the successor of $curr[i]$ which is the successor of $pred[i]$. This lemma can be expressed as $pred[i].next = curr[i]$ & $curr[i].next = entry[i]$, or in terms of ruleset $Npred.next = Ncurr$ & $Ncurr.next = Nentry$. Thus, the *strengthened* ruleset is now:

Ruleset $i : Thread$; $Npred, Nentry, Ncurr : node$
Rule $pc[i] = 4$ & $Npred = pred[i]$ & $Nentry = entry[i]$ & $Ncurr = curr[i]$ & $Npred.next = Ncurr$ & $Ncurr.next = Nentry \Rightarrow$ Begin
  $Npred.next := Nentry$; $pc[i]++$;
End;

Re-abstracting this strengthened rule leads to the abstract rule below which is more constrained than before, thus ruling out some spurious behaviors.

Ruleset $Npred$, $Nentry$, $Ncurr : node$
Rule $true$ & $Npred.next = Ncurr$ & $Ncurr.next = Nentry \Rightarrow$ Begin

$Npred.next := Nentry$;
End;

Observe that the above candidate lemma reflects the sequential behavior of the *Remove* method: it expresses the relationship between *pred*, *curr* and *entry* variables which is easily inferred if the sequential behavior of the thread is analyzed.

Finally, note that in the above rules, $Ncurr.next$ is used instead of $curr[i].next$. This is because using $curr[i].next$ hides the fact that it is a shared variable (accessed by local pointer $curr[i]$) and not a local variable. This can cause the abstraction tool *Abster* to go astray as it operates purely syntactically.

Bounding Memory Usage Blowup: In the concrete model, as the number of threads increase, the number of memory nodes which may have been removed by some *Remove* method but are still referenced by some other *Contains* method may also increase rapidly. These nodes cannot be garbage collected as they are not unreferenced. This is a challenge for model checking since the system state for such algorithms can grow quickly with increasing number of threads. A model with only 2 threads accessing set with at most 2 elements can require as many as 10 nodes [23].

In the CMP abstraction, since the local pointers of the abstract thread are thrown away, a node in the abstract model can only have references from the concrete thread. This naturally bounds the number of nodes which may have been removed from the list and cannot be garbage collected to three (one potentially for *curr*, one for *pred* and one for *entry*), solving the memory usage blowup problem.

4 Automatic Generation of Candidate Lemmas

As discussed in Section 3, a paramaterized model may initially be too abstract to prove the desired properties. In this case, candidate lemmas must be added to constrain the model. This step often requires human effort and expertise.

However, since the CMP method is sound and checks candidate lemmas for correctness, candidate lemmas can be mined from the execution traces. We used Daikon [6] for this purpose. Daikon learns invariants from execution traces, which we generated by sequentially executing one thread. The form of the lemmas learnt by Daikon depends on the templates it uses. For example, we discovered that templates which compared variables with constants tended to produce false candidate lemmas. This is because these lemmas are learnt from a single trace and did not generalize to other traces that well. On the other hand, templates which included comparison of integer variables and checking for equality, tended to produce good candidate lemmas, and were therefore used in all following experiments. The templates can be tuned to infer additional lemmas which had to be manually added. *Tuning of templates is a way of adding domain knowledge and thus speeding up verification of similar algorithms.*

Since the front end of Daikon does not accept Murphi, we used Java implementations of the code available at the accompanying website of [9]. While this

adds to the mechanical burden of running a tool on a different front end, this does not affect the soundness of our approach: if Daikon finds wrong lemmas or we make a mistake while translating them back to Murphi, the CMP method will still catch this. We plan to automate this step in the future.

5 Experiments

We verified the *Fine-grained*, *Optimistic*, *Lazy* and *Lock Free* algorithms presented in [9] using our framework. These algorithms are briefly described below:

***Fine-grained* Set:** The *Fine-grained* set uses hand-in-hand locking for traversing the list: the thread traversing the list releases the lock on a node only when it has locked the successor. Finally, during addition or removal of a node to the list, the thread keeps the two successive vertices locked.

***Optimistic* Set:** Since hand-in-hand locking uses locks to traverse the list, it creates contention between threads. The *Optimistic* algorithm reduces this contention by traversing the list without locking. For correctness, on finding the location to add or remove a node, the thread locks the nodes and then validates that the node it locked is reachable from the *Head* node by re-traversing the list.

***Lazy* Set:** The *Optimistic* algorithm re-traverses the list to validate the node it locked: the *Lazy* implementation eliminates this overhead by maintaining a *marked* field in each node. The algorithm maintains the invariant that an unmarked node is reachable from *Head*. Another advantage of having the *marked* field is that the *Contains* method no longer needs to acquire a lock.

***Lock Free* Set:** The *Lock Free* implementation eliminates the usage of lock and uses *compare and swap* instead for synchronization. This implementation also uses a *marked* field like the *Lazy* algorithm; we refer the reader to [9] for further details.

The linearization points for most of the methods considered in this paper are known. When they are not known, we use the *aux* variable approach described in Section 2.1. Further, the *ConcSet* for all these algorithms is the set of nodes reachable from *Head*, except for *Lazy* and *Lock Free* where the nodes have to be unmarked as well.

Performance For our experiments, we used a 4-core 2.4 GHz Intel processor. Figure 6 shows the model checking results. In the figure, the number of items represent the number of elements in the set and the number of list nodes represent the list size in the implementation. We model checked the correctness of the implementation for a maximum of two items in the set. Note that the number of items does not include the *Head*, *Tail* and memory leak nodes. For example, for the *Lock Free* algorithm, even for two items in the list, the total number of list nodes can go up to 10. In order to cut down the state space and speedup the verification of *Lazy* and *Lock Free* algorithms, we canonicalized the linked list in their implementation: all marked nodes were shifted to the end.

We were initially surprised by the better performance for *Lock Free* algorithm, especially since it is known to be the hardest for verification. We think that this

Algorithm	#Items	#List Nodes Required	Time(s)
Fine-grained	1	3	0.11
Fine-grained	2	4	3.73
Optimistic	1	3	0.83
Optimistic	2	4	114.39
Lazy	1	5	18
Lazy	2	8	29554
Lock Free	1	7	1.62
Lock Free	2	10	401.14
Lock Free (Buggy)	1	7	0.63

Fig. 6: Model Checking Results

is probably due to the fact that the *Other* thread of *Lock Free* algorithm does not have lock() and unlock() calls (which we observed significantly slowed down the *Lazy* algorithm); the abstract model then has only 5 statements. This leads to fewer interleavings and thus a speedup despite the 10 nodes in the list.

The *Lazy* algorithm was the slowest among all. We believe that this is because it uses a large number of list nodes (8 for the two item case) and also has significant interference from the *Other* thread due to locks.

5.1 Verification Experience: Candidate Lemmas

We now describe our experience in verifying these algorithms using Daikon generated candidate lemmas. Since Daikon learns candidate lemmas as clauses for each program location (each candidate lemma then is a conjunction of clauses), we compare the number of clauses in the manually added candidate lemmas with the number of clauses inferred automatically. Further, in what follows, we will consider only those clauses which are left after an initial pruning by model checking for a single thread.

***Fine-grained* Set** For the *Fine-grained* algorithm, 15 clauses were inferred by Daikon, 9 of which were correct. We manually had to add an additional 9 clauses. These 9 clauses were added as 4 candidate lemmas: of these 2 restrained the abstract statements from accessing the shared state when it was not locked by *Other* and one constrained that modifications be done to shared state only if the nodes pointed by the local pointers *curr* and *pred* are reachable from *Head*.

***Optimistic* Set** For the *Optimistic* set, 15 clauses were generated by Daikon, of which 2 were spurious. 13 clauses were added manually; these were added as 5 candidate lemmas. Similar to the *Fine-grained* algorithm case, two of these constrained accesses to happen only in locked region and two constrained the local pointers to be reachable from *Head*.

***Lazy* Set** For the *Lazy* set, Daikon generated 39 clauses out of which 1 was spurious. We manually added 9 clauses as 5 candidate lemmas. Three of these constrained accesses to happen only in the locked region and one constrained the local pointers to be reachable from *Head*.

***Lock Free* Set** In the *Lock Free* case, our candidate lemma generation scheme had limited success: we had a large number of spurious clauses. The total number of generated clauses which model check for single thread case is 29. Out of these, only 7 carried through and thus are of help in the proof. The rest of the clauses were falsified. The total number of manually added clauses was 23, from 5 manually added candidate lemmas.

Specific Candidate Lemmas for *Lazy Remove*: We now describe the specific lemmas which we added for the *Remove* method of the *Lazy* set algorithm shown in Figure 3d. As can be seen from the pseudocode, the *Remove* method modifies the shared state (list nodes) in lines 3 and 5. Since the CMP method discards all the local updates in constructing the environment abstraction (shown in Figure 5), lemmas are needed only for the rules corresponding to these lines for refinement.

The first set of candidate lemmas are the ones which describe the relationships between the thread variables while making updates. As discussed in the previous section, such lemmas are learned by using Daikon. For example, an important lemma in this category is that when the rule corresponding to line 3 of the *Remove* method is fired, the node pointed by *curr* is next to that pointed by *pred* and that by *entry* is next to *curr*. Other examples include lemmas which compare the values of the *key* fields of nodes pointed by *pred*, *curr* and *entry* variables.

The second set of candidate lemmas are concerned with synchronization. For the rules corresponding to line number 3 and 5, the variables *pred* and *curr* must be locked by the calling thread at the time of execution of the rules. While these lemmas had to be added manually, they can potentially be added automatically since the only information required is whether the variables should be locked or not. For concurrent data structures, the lock scopes are known statically.

Finally, the last candidate lemma which we had to manually add was for the rule corresponding to line 5. This lemma stated that when the node pointed by *curr* is being removed from the list, it should be marked.

Discussion We observed that the manually added candidate lemmas, particularly for *Fine-grained*, *Lazy* and *Lock Free* algorithms, had a very similar structure, and had information which could be inferred from a single thread. This strongly motivates further research on exploring even better techniques for inferring lemmas from a single thread.

6 Conclusion and Future Work

In this paper we presented a case study of parameterized model checking of concurrent data structures. In particular, we have shown how concurrent list based set data structures can be model checked for an unbounded number of threads by leveraging an important parameterized model checking based technique. We have also shown how the manual effort involved in the parameterized model

checking using the CMP method can be reduced by mining invariants from the execution trace of a single thread.

Our work opens up interesting future research directions. First, it suggests experimental exploration for better invariant generation approaches which can capture most of the lemmas required by the CMP method based verification loop. In particular, we believe that better candidate lemmas can be generated by leveraging higher level specifications like those used for program sketching. Second, it highlights the need for generic data structure abstractions, which can be integrated with the CMP method to model check the data structures for an unbounded size and with an unbounded number of threads accessing them. This is essential for full formal verification.

References

1. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Thread quantification for concurrent shape analysis. In: Proceedings of the 20th international conference on Computer Aided Verification. pp. 399–413. CAV '08, Springer-Verlag, Berlin, Heidelberg (2008)
2. Calcagno, C., Parkinson, M., Vafeiadis, V.: Modular safety checking for fine-grained concurrency. In: In SAS. LNCS. The MIT Press (2007)
3. Cerný, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., Alur, R.: Model checking of linearizability of concurrent list implementations. In: Computer Aided Verification. pp. 465–479 (2010)
4. Chou, C.T., Mannava, P.K., Park, S.: A simple method for parameterized verification of cache coherence protocols. In: Proc. FMCAD) (2004)
5. Colvin, R., Groves, L., Luchangco, V., Moir, M.: Formal verification of a lazy concurrent list-based set. In: In 18th CAV. pp. 475–488. Springer (2006)
6. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming 69(1–3), 35–45 (Dec 2007)
7. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Proceedings of the 15th International Conference on Distributed Computing. pp. 300–314. DISC '01, Springer-Verlag, London, UK, UK (2001), `http://dl.acm.org/citation.cfm?id=645958.676105`
8. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer, W., Shavit, N.: A lazy concurrent list-based set algorithm. In: Proc. of the 9th International Conference On Principles Of Distributed Systems. pp. 3–16 (2005)
9. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2008)
10. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12, 463–492 (July 1990)
11. Kristic, S.: Parameterized system verification with guard strengthening and parameter abstraction. 4th Int. Workshop on Automatic Verification of Finite State Systems (2005)
12. Lea, D.: The java.util.concurrent synchronizer framework. Sci. Comput. Program. 58, 293–309 (December 2005), `http://dl.acm.org/citation.cfm?id=1127037.1127039`

13. Liu, Y., Chen, W., Liu, Y.A., Sun, J.: Model checking linearizability via refinement. In: Proceedings of the 2nd World Congress on Formal Methods. pp. 321–337. FM '09, Springer-Verlag, Berlin, Heidelberg (2009)
14. McMillan, K.L.: Verification of infinite state systems by compositional model checking. In: Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods. pp. 219–234. CHARME '99, Springer-Verlag, London, UK, UK (1999), http://dl.acm.org/citation.cfm?id=646704.702020
15. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures. pp. 73–82. SPAA '02, ACM, New York, NY, USA (2002), http://doi.acm.org/10.1145/564870.564881
16. Michael, M.M., Scott, M.L.: Correction of a memory management method for lock-free data structures. Tech. rep., Rochester, NY, USA (1995)
17. O'Leary, J., Talupur, M., Tuttle, M.: Protocol verification using flows: An industrial experience. In: Formal Methods in Computer-Aided Design, 2009. FMCAD 2009. pp. 172 –179 (nov 2009)
18. Reinders, J.: Intel threading building blocks. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edn. (2007)
19. Talupur, M., Tuttle, M.: Going with the flow: Parameterized verification using message flows. In: Formal Methods in Computer-Aided Design, 2008. FMCAD '08. pp. 1–8 (nov 2008)
20. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 335–348. VMCAI '09, Springer-Verlag, Berlin, Heidelberg (2009)
21. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 129–136. PPoPP '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1122971.1122992
22. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: IN 18TH CONCUR. pp. 256–271. Springer (2007)
23. Vechev, M., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: Proceedings of the 16th International SPIN Workshop on Model Checking Software. pp. 261–278. Springer-Verlag, Berlin, Heidelberg (2009)
24. Zhang, S.J.: Scalable automatic linearizability checking. In: Proceeding of the 33rd international conference on Software engineering. pp. 1185–1187. ICSE '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1985793.1986037
25. Zhang, S.J., Liu, Y.: Model checking a lazy concurrent list-based set algorithm. In: Proceedings of the 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement. pp. 43–52. SSIRI '10, IEEE Computer Society, Washington, DC, USA (2010), http://dx.doi.org/10.1109/SSIRI.2010.37