Pattern-based Synthesis of Synchronization for the C++ Memory Model

Yuri Meshman
Technion

Noam Rinetzky
Tel Aviv University

Eran Yahav
Technion

***Abstract*—We address the problem of synthesizing efficient and correct synchronization for programs running under the C++ relaxed memory model. Given a finite-state program P and a safety property S such that P satisfies S under a sequentially consistent (SC) memory model, our approach automatically eliminates concurrency errors in P due to the relaxed memory model, by creating a new program P with additional synchronization. Our approach works by automatically exploring the space of programs that can be created from P by adding synchronization operations. To explore this (vast) space, our algorithm: (i) explores bounded error traces to detect memory access patterns that can occur under the C++ memory model but not under SC, and (ii) eliminates these error traces by adding appropriate synchronization operations.**

We implemented our approach using CDSCHECKER as an oracle for detecting error traces and Z3 to symbolically explore the space of possible solutions. Our tool successfully synthesized synchronization operations for several challenging concurrent algorithms, including a state of the art Read-Copy-Update (RCU) algorithm.

I. INTRODUCTION

We address the problem of synthesizing efficient and correct synchronization for programs running under the *C++ relaxed memory model* (C++ RMM) [13]. Writing correct and efficient low-level concurrent programs in C++ is known to be very challenging. One of the challenging aspects is dealing with the details of the relaxed memory model; the complexity of the model is such that it eludes even veteran systems programmers and requires the attention of formal semantics experts [7], [8], [23], [26].

From a programmer's perspective, the main question is how to synchronize operations in her program to guarantee correctness and efficiency under the relaxed memory model. Under the C++ RMM, each operation on an atomic object is annotated with a *memory order*. The memory order ranges from being fully *relaxed* all the way to being fully *sequentially consistent* (for that atomic object), with a few more subtle modes in between these two extremes. Finding a correct and *efficient* synchronization is critical for many algorithms and concurrent data structures. In particular, to maintain efficiency, the programmer wants the most relaxed synchronization required to preserve correctness, and not more (even when it simplifies reasoning). Unfortunately, manually finding the right synchronization is extremely difficult, as it requires the programmer to reason about subtle interaction of the memory model. Our goal is to assist the programmer by automatically synthesizing the required synchronization.

A. The Problem

Given a finite-state program P and a safety property S such that $P \models S$ under a sequentially consistent (SC) memory model, we aim to automatically synthesize a program P', whose behaviors are a subset of P's behaviors, s.t. $P' \models S$ under C++ RMM in bounded executions.

B. Our Approach: Pattern Based Synthesis of Synchronization

Our synthesis algorithm automatically explores the space of programs that can be created from P by modifying memory access synchronization. To explore this (vast) space, our algorithm: (i) inspects P's (bounded) error traces to detect memory access patterns that can occur under C++ RMM but not under SC, and (ii) eliminates these error traces by preventing the occurrence of the detected violation patterns using as little synchronization as possible.

More specifically, our algorithm exhaustively explores the traces of P under *C++ RMM*, and looks for *error traces*—traces which do not satisfy the specification S. In case it finds an error trace, it inspects it looking for instances of *violation patterns*, behaviors that may occur under C++ RMM but not under SC *and* that we know how to avoid. (Recall that P satisfies S under *SC*. Hence, the violation of S must be due to behaviors introduced by the weak memory.) The algorithm then constructs a constraint which encodes all possible *avoidance templates* that can be used to eliminate that particular error trace. Where, *avoidance templates* are strategies to synthesize *memory order annotations* of memory instruction such as `load`, `store`, and `cas`. The algorithm accumulates the constraints required to eliminate the error traces and passes them to a SAT solver in the form of a CNF formula φ. Every satisfying assignment of φ represents a different way to synthesize the desired *memory order synchronization*.

The algorithm then checks which of the resulting programs satisfies S. The check is required because our set of violation patterns and avoidance templates is not complete. (In fact, we believe that coming up with a complete set is nontrivial, if at all possible). This means a program P' which contains no violation patterns may still violate the original specification S.

C. Main Contributions

The contributions of this paper are as follows:

- A novel approach for detecting missing synchronization using violation patterns, patterns of memory accesses that can happen under C++ RMM but not under SC.

- A technique for synthesizing synchronization by eliminating violation patterns using *avoidance patterns*, a set of predefined synchronization strategies.
- An algorithm which given a program P and a specification S synthesizes synchronization to ensure that P respects S in bounded executions.
- An implementation of our approach and empirical evaluation in which we successfully synthesized synchronizations for several challenging concurrent programs, including a program using a state of the art Read-Copy-Update (RCU) algorithm.

II. Overview

In this section, we provide an informal overview of our approach using our running example, Dekker's mutual exclusion algorithm for two threads [12].

A. Running example

Fig. 1 shows one of the many variants of Dekker's algorithm. The `load` (read) and `store` (write) commands are subscripted with *memory order annotations*. For now, these annotations can be ignored. The algorithm is comprised of two parts: An *entry* section (lines 1–7) and an *exit* section (lines 9–10). The critical section itself (line 8) is irrelevant, and elided. The algorithm enforces mutual exclusion using variables `flag`[0] and `flag`[1], and ensures deadlock and starvation freedom using variable `turn`.

To enter the critical section, thread i, where i is either 0 or 1, needs to execute its entry section: First, it sets the value of variable `flag`[i] to 1 (line 1), thus signaling its intentions to the other thread. Then, it inspects the value of `flag`$[1-$`i`$]$ to check if the other thread is also trying to enter the critical section or if it is already in it (line 2). If this is not the case, then it proceeds to the critical section. Otherwise, it sets its own flag to 0 (line 4), thus letting the other thread proceed, and waits for its turn to enter the critical section (line 5). Upon leaving the critical section, thread i executes the exit section, where it gallantly gives precedence to the other thread by setting `turn` to $1 - i$ and signals that it left the critical section by setting the value of its flag to 0.

It is important to note that as long as a thread executes the critical section its flag is set to 1. Furthermore, a thread enters the critical section only after it discovers that the flag of the other thread is set to 0 at a time in which its own flag is set to 1. The above observation suffices to ensure mutual exclusion under SC: In this memory model, there is a total order between all the `load` and `store` commands and reading the value of a variable x returns the last value written to x. As a result, if two threads compete on entering the critical section, then at least one thread must notice in line 2 that the flag of the other thread is set to 1. Unfortunately, under C++ RMM this is no longer the case. The reason for this unintuitive behavior can be understood from the following simple program involving only two `store` and two `load` commands.

Example 1: Consider the following program and assume that both `flag`[0] and `flag`[1] are initialized to 0, that $\texttt{r}_0$ and $\texttt{r}_1$ are initialized to 2, and that each of $\texttt{r}_0$ and $\texttt{r}_1$ is local to the respective thread.

$$\texttt{store}_W(\texttt{flag}[0], 1);\ \texttt{r}_0 = \texttt{load}_X(\texttt{flag}[1]) \parallel \texttt{store}_Y(\texttt{flag}[1], 1);\ \texttt{r}_1 = \texttt{load}_Z(\texttt{flag}[0])\ .$$

Under *SC*, at the end of the program the only possible values of $\texttt{r}_0$ and $\texttt{r}_1$ are 0 and 1. Furthermore, at most one of them can have value 0. Under *C++ RMM*, their values can be either 0, 1 or 2, depending on the memory order annotations W, X, Y, and Z. Intuitively, this can happen because under C++ RMM a `store` operation can behave as if it writes its value to a "thread-local store buffer", leaving the other threads to read the value stored in the global memory.(C++ RMM exhibits x86-TSO behaviours)

As indicated by the above example, to ensure mutual exclusion the program must add synchronization to the program. One way to do it in C++ RMM is to explicitly annotate the `load` and `store` operations with the type of the required synchronization. Using strong synchronization primitives (e.g., requiring all `load` and `store` operations to be sequentially consistent) is expensive. Too weak synchronization primitives, on the other hand, leads to unexpected behaviors. Thus, determining correct and efficient annotations is challenging. Our tool, however, determined that the program shown in Fig. 1 is safe if the memory operations in lines 1, 2, 3, and 9 are sequentially consistent; and the `store` in line 10 is memory order release [1]. (See Section III.)

Note 1: The `load` in line 5 is not synchronized (i.e., it is annotated with RLX). However, as we show in Section V, our result is still verified by our underline model checker.

B. Synthesizing synchronization

The main insight behind our approach for synthesizing synchronization is that we can turn a program which is safe under SC to be safe under a weak memory model (C++ RMM in our case) by removing behaviors that cannot occur under SC. This approach faces three main challenges: (i) detecting such behaviors, (ii) determining a (cheap) way to remove them, and (iii) verifying that the resulting program is safe.

Addressing the first challenge We overcome the first challenge by exhaustively searching the program state space for an error trace, developing all the concrete traces which are possible under C++ RMM. We allow to specify safety properties as (a) assertions on the final state, (b) properties of thread-local variables, and (c) races on non-atomic location (see Section III). The search is guaranteed to terminate because we only follow bounded traces of finite state programs.

Addressing the second challenge If we find an error trace, we look for instances of violation patterns, memory behaviors involving a small number of `load` and `store` actions which can happen under C++ RMM but not under SC and which we know how to prevent. Once we discover such an instance, we add synchronization annotations to the relevant memory operations based on a predefined avoidance template which

[1] To the best of our knowledge, our solution is the only one to use memory order synchronizations and not fences.

```
i.1 store_SC(flag[0], 0);
i.2 store_SC(flag[1], 0);
i.3 store_SC(turn, 0);
```

Thread 0:

```
1  store_SC(flag[0], 1);
2  while( load_SC(flag[1])==1 ){
3   if( load_SC(turn)==1 ){
4    store_RLX(flag[0], 0);
5    while( load_RLX(turn)==1 )yield();
6    store_RLX(flag[0], 1);
7  } }
8      ... // critical section
9  store_SC(turn, 1);
10 store_REL(flag[0], 0);
```

Thread 1:

```
1  store_SC(flag[1], 1);
2  while( load_SC(flag[0])==1 ){
3   if( load_SC(turn)==0 ){
4    store_RLX(flag[1], 0);
5    while( load_RLX(turn)==0 )yield();
6    store_RLX(flag[1], 1);
7  } }
8      ... // critical section
9  store_SC(turn, 0);
10 store_REL(flag[1], 0);
```

Fig. 1. Dekker's mutual exclusion algorithm. Variables `flag[0]`, `flag[1]` and `turn` are declared as atomic locations and initialized to 0. The subscripts indicate the synchronization (consistency) annotations synthesized by our tool.

blocks the violation pattern, thus eliminating the error trace.

We describe the inferred synchronization annotations using a propositional formula and ask a SAT solver to find the sets of minimal satisfying assignments. (Note that a trace might contain several instances of violation patterns and thus can be eliminated using different avoidance patterns.) From each assignment, we generate a program and repeat the process until no bad trace is found. We use the verified solutions as a starting point in a new round of synthesis in which we raise the bound on the explored traces.

The algorithm is guaranteed to terminate because we consider only finite state programs, there is a finite number of memory annotations, and every change only increases the degree of synchronization (see Section III).

Example 2: Fig. 2 shows a trace of the Dekker algorithm which violates mutual exclusion. The trace contains two violation patterns, store buffering (SB) and load buffering (LB). The former, which we discussed in Example 1, is manifested here by the initialization `store` actions in lines 1 and 2, and the `load` actions in lines 5 and 8. (An rf-annotated arrow from a `store` action to a `load` action indicates that the latter read the value written by the former.) This instance of the SB pattern is blocked by synthesizing a SC annotation to the corresponding memory operations in the algorithms (lines $i.1$, $i.2$, 1, and 2 in Fig. 1.)

Note 2: The list of violation patterns and their corresponding synchronization templates is given as an input to the algorithm. Our algorithm is parametric in that list. The specific patterns and templates that we use in our implementation is given in Section IV-B.

Addressing the third challenge Our set of violation patterns and avoidance templates is not complete. Thus, after synthesizing the programs, we simply explore the state space again.

The synthesis procedure terminates if the offered solution contains only sequentially consistent memory accesses, and is thus correct by our assumption, or when no error trace is found. This ensures that the program satisfies the desired properties in executions in which every thread performs no more instructions than the explored bound.

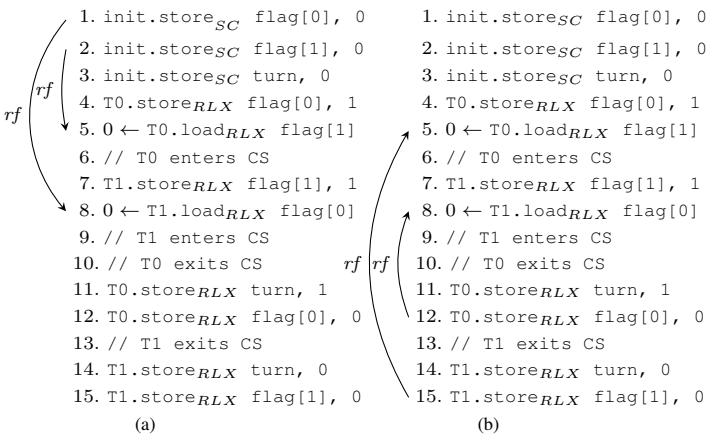

Fig. 2. An error trace containing two violation patterns: (a) store buffering (SB) and (b) load buffering pattern (LB). These patterns were detected by our tool when analyzing Dekker's algorithm.

III. C++ Relaxed Memory Model in a Nutshell

A memory model defines the possible behaviors of instructions such as `load` and `store` in the program. Arguably, the most intuitive (and restrictive) memory model is *Sequential Consistency* (SC) [19] in which there is a total order on the `load` and `store` instructions, and every `load` from location l reads the last value `stored` in l. (For simplicity, we treat the initial state as if it was produced by explicit `store` operations.)

C++ relaxed memory model relaxes the requirement of a total order on memory operations, thus allowing for a richer set of behaviors. The relaxation frees the hardware designers to implement various optimizations, but leads to unintuitive behaviors: The model distinguishes between *atomic* locations, where racy accesses are allowed, and *non atomic* locations, where the behavior of races is undefined. C++ RMM defines the behavior of memory operations using several partial orders, and allows `load` operations to return the value written by any `store` operation, *provided* certain requirements are respected. Below, we provide a (greatly simplified) overview of the part of C++ RMM regarding requirements on *relaxed* (RLX), *release/acquire* (REL/ACQ), and *sequential consistent* (SC) memory accesses.

C++ RMM defines the behavior of memory operations using several partial orders over the set of memory operations. The two main ones being a *read from* (rf) relation, denoted by $\rightarrow_{rf}$, and a *happens before* (hb) relation, denoted by $\rightarrow_{hb}$. rf relates every `load` to the `store` from which it takes its value and hb formalizes the notion of *dependency*.

The model ensures that the only possible executions are ones in which the rf and hb relations satisfy certain constraints. Firstly, hb must be acyclic. Secondly, rf and hb should not contradict each other: a `load` *cannot* read from a `store` that (i) depends on it, i.e., follows it in the hb relation, or (ii) is masked by another write, i.e., there exists a $\texttt{store}_2$ operation such that $\texttt{store} \rightarrow_{hb} \texttt{store}_2 \rightarrow_{hb} \texttt{load}$. Thirdly, the hb relation must respect the *program order*. Specifically, it must contains the *sequence before* (sb) relation, denoted by $\rightarrow_{sb}$, which places an irreflexive total order on the actions executed by the same thread. Fourthly, hb should be consistent with the *modification order* which defines a total order on all the `store` operations to the *same* location.

In addition, the hb relation must respect the *memory order* synchronization. Every memory operation is annotated with a *memory order annotation* which, intuitively, specifies its consistency level—the level of synchronization and ordering it requires. We consider three forms of annotations:

(i) SC-annotated memory actions must be totally ordered,
(ii) An ACQ-annotated `load` which gets its value from a REL-annotated `store` depends on it. Technically, these annotations induce a *synchronize with* (sw) order, denoted by $\rightarrow_{sw}$, which should be part of the hb relation. (We refer to this as adding rf edges to hb), and
(iii) RLX-annotated operations do not place additional restrictions on the hb relation.

IV. Synthesis of Synchronization

In this section we describe our synthesis algorithm (Section IV-A) and review the violation patterns with their respective avoidance templates (Section IV-B), that we implemented and experimented with. We also present two *abstract violation patterns* that go beyond concrete litmus tests: We identify patterns involving a small number of memory operations on a *single* location, and describe how to block them by placing a *chain* of dependencies going through an unbounded number of accesses to (possibly) different locations (Section IV-C).

A. Atomics memory access synchronization synthesis

Our synthesis procedure is comprised of two nested loops. The inner one synthesizes synchronization for a given program and the outer keeps refining the set of solutions by gradually increasing the bound on the length of the explored traces.

Algorithm 1 implements the inner loop of the synthesis procedure. It takes as input a program P, a specification S and produces a set of programs P' which satisfy S under C++ RMM using different forms of synchronization.

The algorithm first checks if P satisfies S, and if so returns it (line 2). Otherwise, it goes over the set of traces which violate the specification (line 5) and looks for violation

```
1  Procedure SynSync(P, S)
2      if P ⊨ S then return {P}
3      φ = true
4      𝒫 = ∅
5      foreach e ∈ errorTraces(P,S) do
6          β = blockOccurr(e, AcqRelFix())
7          if β then continue
8          β = blockOccurr(e, SCFix())
9          if ¬β then return allSC(P)
10         φ = φ ∧ β
11     φ = φ ∧ ⋀ impliedSync(φ)
12     avoidance = SAT(φ)
13     foreach annotation ∈ avoidance do
14         P' = addSync(P,annotation)
15         𝒫 = 𝒫 ∪ SynSync(P', S)
16     return 𝒫
17 blockOccurr(e,patterns)
18   β = false
19   foreach (p,c) ∈ patterns do
20       foreach i ∈ occurrence(p, e) do
21           β = β ∨ blockPattern(i,c)
22   return β
23 impliedSync(φ) = {a → b | a,b ∈ vars(φ)
24                 ∧ (SC ∈ annot(a))
25                 ∧ (REL ∈ annot(b) ∨ ACQ ∈ annot(b))
26                 ∧ (instr(a) == instr(b))}
```

Algorithm 1: The inner loop of the synthesis procedure.

```
1  Procedure PSynSync(P, S, N)
2      C_init, C_0, ..., C_m = getCmds(P)
3      𝒫_1 = SynSync(C_init ; (C_0 ‖ ... ‖ C_m) , S)
4      for n = 2 to N do
5          𝒫_n = ∅
6          foreach P' ∈ 𝒫_{n-1} do
7              C_init, C_0, ..., C_m = getCmds(P')
8              Loop_0 = "for i_0 = 1...n do C_0"
9              ...
10             Loop_m = "for i_m = 1...n do C_m"
11             P'' = "C_init; (Loop_0 ‖ ... ‖ Loop_m)"
12             𝒫_n = 𝒫_n ∪ SynSync(P'', S)
13     return 𝒫_N
```

Algorithm 2: The synthesis procedure. Program P is comprised of an initialization command C_{init} followed by a parallel composition of $m+1$ threads, where thread i executes command C_i for N times.

patterns in each trace. First, it searches for patterns which can be prevented using Acquire-Release synchronization (line 6), and only if no such patterns are found in the trace, it searches for patterns that can be prevented using the more expensive Sequential Consistency synchronization (line 8).

The search for instances of violation patterns and the corresponding avoidance template is done by the auxiliary procedure `blockOccur(·)` (Lines 19, 20). If there is an instance i of a pattern p in trace e, then the avoiding template

is instantiated according to the instance i and recorded in β as one way to eliminate trace e (line 21). Technically, an instance of an avoidance template is a conjunction of pairs (`instr`, `annot`), where `instr` is a `load` or a `store` in P and `annot` is the suggested synchronization for that instruction: either SC, REL, or ACQ. Intuitively, the conjunction records the memory order annotations pertaining to the actions forming the detected instance i, which suffice to prevent it. Formula β is constructed as a disjunction of correction application to the trace. The blocking formulae pertaining to all the error traces are accumulated as a conjunctive formula φ (line 10.)

Finally, we record in φ that every constraint enforced by a REL or ACQ synchronization is also enforced by an SC synchronization via adding the corresponding implications (line 11), thus increasing the set of possible solutions.

Every satisfying assignment to the program correction formula generates a different program P' which has more restrictive synchronization than P (line 13). We determine whether P' respects the specification S, or requires further synchronization, by calling `SynSync` recursively.

If `blockOccur(·)` does not find a way to eliminate an error trace, we annotate all memory operations as SC (line 9).

Algorithm 2 implements the outer loop of the synthesis procedure. For simplicity, we assume that the input program is comprised of an initialization command C_0 followed by a parallel composition of $m + 1$ loops, where loop i repeats executing a sequential command C_i for N times.

The algorithm takes the original program P, a specification S and the loop bound N, and generates a set of programs $\mathcal{P}_N$ which restrict the synchronization in P so it satisfies S. Because the number of behaviors rapidly grows as loop iteration is increased, we take an incremental approach: We iteratively construct a sequence of sets of programs $\mathcal{P}_n$, which satisfy the specification S when each loop performs only n iterations (lines 3 and 12). The programs in $\mathcal{P}_n$ are used as a starting point in synthesizing programs with $n + 1$ iterations (line 6). Upon termination we return a set of different programs which refines P using different memory order synchronization such that S is respected.

B. Patterns of weak memory behavior.

As mentioned previously, C++ RMM allows for a `load` certain behaviors which are not possible under SC. Below, we list some patterns of such behaviors and explain how they can be prevented using appropriated memory order annotations [7], [8]. The patterns can be seen in Fig. 3. We drew intuition for the patterns from what is often referred to as *litmus tests* [8].

Store Buffering (SB): This is the pattern from Fig. 2(a). In this pattern, two threads first write to two different locations and then try to determine the value of the location written by the other one. It might be the case that each thread would not observe the `store` made by the other. This behavior can occur when the stores of one thread are not made immediately visible to the other.

Pattern prevention. This patterns can be prevented only by making all the `load` and `store` instructions SC.

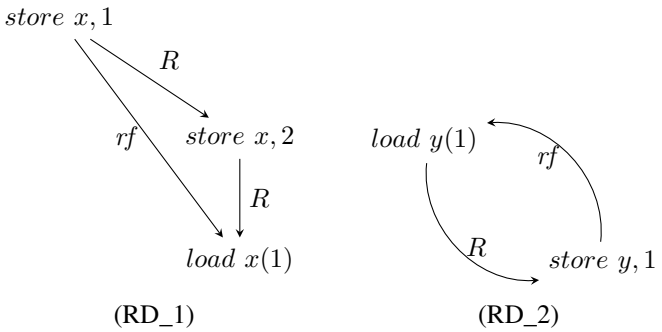

Fig. 4. Abstract Patterns of behaviors that are possible under C++ RMM but not under SC.

Independent Reads of Independent Writes (IRIW): Here two threads write to two different locations and the other two threads see those writes in different orders.

Pattern prevention. The above two patterns can be prevented only by making all the `load` and `store` instructions SC.

Load Buffering (LB): This is the pattern from Fig. 2(b). This patterns indicates that every thread can see later (according to the sb relation) writes of the other threads. Note that as the `store` might actually be dependent on the `load`, this pattern indicates that each thread can "magicaly" satisfy the needs of the other. Hence, this pattern is also called *satisfaction cycles* or reading values *out-of-thin-air*.

Pattern prevention. Adding one of the rf edges to hb would prevent this pattern. This can be done by annotating the `store` and `load` instructions of that edge with REL and ACQ, respectively.

Message Passing (MP): Here, one thread writes to two different locations, and the other thread sees the value written by the second `store` (to y), and miss the first `store` (to x).

Pattern prevention. Annotating the `store` to y with REL and the `load` from y with ACQ would add the rf edge to the hb relation and would prevent the pattern.

Write-to-Read Causality (WRC): This pattern is similar to the message passing pattern, but involves three threads. Here, the value written to x by the first thread is read by the second thread which then, according to the sb order, writes a value to y. The third thread sees the value written by the second thread but not by the first.

Pattern prevention. Annotating the `load` and `store` with REL and ACQ respectively, would prevent this pattern.

C. Abstracting the patterns

The presented patterns list captures several behaviours of C++RMM. Instances of those patterns were observed in almost all of our benchmarks but there are still C++RMM behaviours not captured by the previous list. What's more, the patterns share some similarities. In an attempt to brings us closer to completeness we drew on that resemblance and extracted the commonalities into abstract patterns.

using RD property in (RD_1, RD_2): The following patterns are motivated by the RD property defined in [7]. The relation R can be instantiated in two different ways. First as a transitive closure of rf and hb relations, and second as a

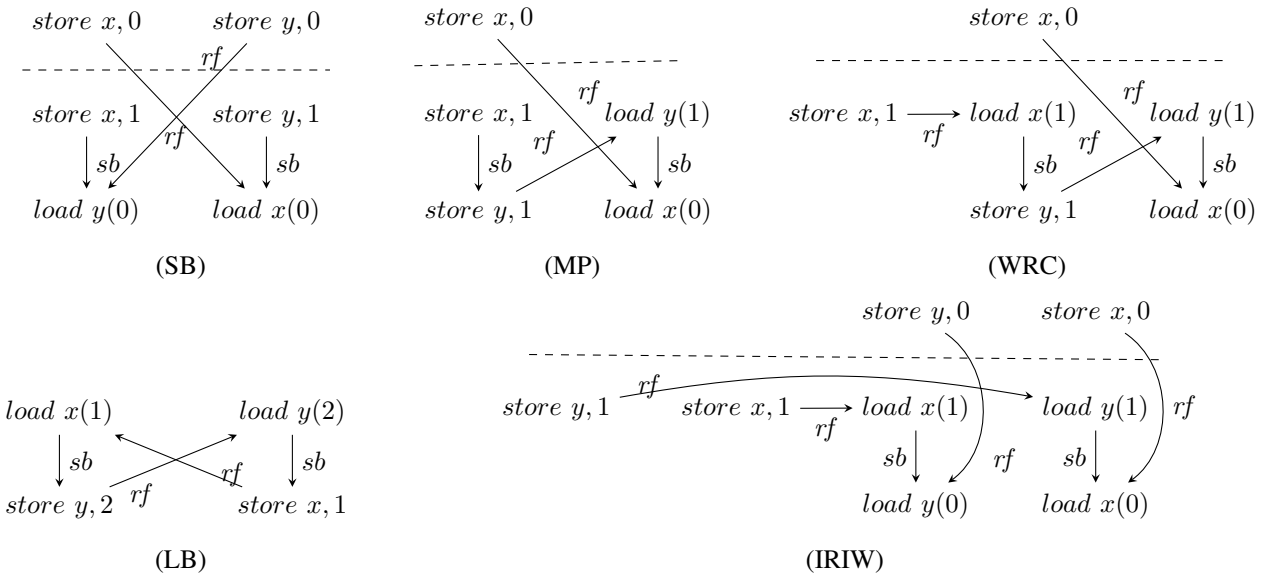

Fig. 3. Patterns of behaviors that are possible under C++ RMM but not under SC. Every column depicts the actions of one thread. We denote by *store x, 1* a write of value 1 to location x and by *load x(1)* a read of value 1 from x. We assume that the initial value of x and y is 0.

possible total order on the involved instructions.

For the first instantiation, the relation R is the transitive closure of $rf \cup hb$. Making all `load` instructions ACQ, and all `store` instructions REL across the path will add all the rf edges along the path in R to hb, forming a sequence violating [7]'s RD and preventing that behaviour.

When we cannot find such instantiation of the relation R in the error trace, we try to instantiate it as a possible total order of instructions, and prevent the error trace using SC. In our implementation we chose to attempt instantiation of R as the scheduler choice made by CDSCHECKER. One such scheduling choice is exemplified in Fig. 2(a) as the index of instructions 1-15, is a possible total order which the SB pattern violates. Forcing total order of instruction, involved in the pattern, (making the memory order access SC) will cause the load to violate [7]'s RD property.

In addition: 1) RD_1 with R as $rf \cup hb$ transitive closure is an abstraction of the message passing(MP) pattern. 2) RD_2 with R as $rf \cup hb$ transitive closure is an abstraction of the load buffering(LB) pattern. 3) RD_1 with R as a possible instruction total order is an abstraction of the Store Buffering(SB) pattern. 4) RD_2 with R as a possible instruction total order is a read from future C++ relaxed behaviour.

V. IMPLEMENTATION AND EXPERIMENTAL EVALUATION

We implemented our approach in a tool called PSYNSYN based on CDSCHECKER [20]. Our tool computes a symbolic formula that captures possible fixes, and uses Z3 to find minimal satisfying assignments. Then, we thoroughly evaluated the tool on a number of challenging concurrent algorithms. For all benchmarks our tool found a non trivial solution with non SC memory accesses. All experiments were conducted on an AMD Opteron Processor 6376 with 128GB RAM and 64 cores, but using only a single thread per benchmark execution. The synthesized solutions are available at [1].

The results of the experimental evaluation are summarized in Table V. The meaning of most table columns is self explanatory, but we elaborate on the following columns:

- *N* – is the maximal number of iterations for each thread we attempted.
- *patterns observed* – is the instances of patterns described in Section IV-B and C, we found for each algorithm.
- *# solutions* – is the number of solutions to the benchmark we found. Unless otherwise specified the solutions are for maximal N attempted.
- *# bad traces for N=1* – is the number of bad traces CDSCHECKER found in the original benchmark with each process doing 1 iteration.
- *inferred synch* – the number of memory access synchronization of every type suggested by our tool in every solution. Due to space restriction we present synchronization of up to 3 solutions per each benchmark and use "..." if more solutions exist.

All our benchmarks, when having an error trace, exhibited one of the searched patterns. For RD_1 and RD_2 patterns SC notation is used when the relation R was instantiated by a possible instruction scheduling and SC synchronisation was needed for the error trace prevention. In addition, the RD pattern occurrences are reported only if they could not be captured by patterns from Fig. 3. e.g. a pattern instance similar to MP but the path from *store x,1* to *load x(0)* involved more than three $sb \cup rf$ edges so it can only be classified as RD_1 and cannot be classified as MP.

For *abp* we can see that the original algorithm verified when each process performed 1 iteration. It was not until each process did 3 iterations that a violation of the checked property was encountered. At that point 1 error trace was found but it exhibited several patterns therefore several ways

Algorithm	N	time (s)	space (Mb)	# calls ToZ3	patterns observed	# solutions	# bad traces for N=1	inferred synch (SC, REL, ACQ, RLX)
Alternating Bit Protocol (abp)	5	20s.89	22	1	MP(SC), RD_1 RD_4	5	(N=1,2) 0, (N=3) 1	(5, 0, 0, 1) (4, 0, 0, 2) ...
dekker [12]	1	3m:22	22	3	MP, LB, SB, RD_1, RD_2, RD_1(SC), RD_4	13	631	(10, 1, 0, 8) (13, 0, 1, 5) ...
d-prcu-v1 [6]	3	3m:14	19	20	LB, SB, RD_2, RD_1(SC), RD_2(SC),	7	5	(7, 2, 1, 0)10 (7, 1, 0, 2) ...
d-prcu-v2 [6]	3	3h:53m	22	88	MP, LB, RD_2, RD_1(SC), RD_2(SC),	17	8	(9, 2, 1, 4) (12, 1, 1, 2) ...
kessel [15]	3	57m:16	22	5	MP, LB, SB, RD_1, RD_2, RD_1(SC), RD_2(SC)	2	85	(13, 1, 0, 0) (14, 0, 0, 0)
peterson [22]	3	26m:41	22	3	MP, LB,RD_2, RD_1(SC) , RD_2(SC)	(N=1) 2* (N=2,3) 2	37	(11, 1, 0, 1) *(12, 1, 0, 0) (13, 0, 0, 0)
bakery [18]	2	10m:21	33	3	MP, LB, RD_1,RD_2, RD_1(SC), RD_2(SC)	(N=1) 6 (N=2)4	974	(16, 1, 1, 0) (17, 0, 1, 0) ...
ticket [5]	4	1m:08	19	7	RD_1(SC), RD_2(SC)	4	8	(9, 0, 0, 1) (8, 0, 0, 2) ...
treiber stack [24]	1	1h:05	23	1	MP	1	160	(0, 5, 3, 4)

TABLE I
RESULTS OF SYNCHRONISATION SYNTHESIS

of preventing it were found. We found 5 solutions that verified for 3 iterations of *abp*, and those solutions verified for 4 and 5 iterations as well.

In fact for almost all our benchmarks, solutions once found stayed consistent as verified solutions when more iterations per thread were attempted. This was not the case for *peterson* as indicated by the "*" in the last column. Here the solution (11, 1, 0, 1) (which are (SC, REL, ACQ, RLX) respectively) found in the 1 iteration had a mutual exclusion breach when attempted with 2 iterations and the solution was further restricted by making the one relaxed memory access SC turning it into the solution marked with "*" beside it. That solution later verified for 3 iterations.

For *bakery* the 6 solution in iteration 1 reduced to a subset of 4. For all other benchmarks the solutions in the last column were found with 1 iteration per each process and remained verified for the maximal number of iterations attempted.

There were previous attempts to verify RCU under C++RMM such as [26] but the version we verified was the first one where an update waits only for the reads whose consistency it affects, and not waits for the completion of all existing reads.

For *Dekker*'s algorithm, we are not aware of any previous attempt to synthesise correct version of it using memory accesses instead of fences (one such fence solution is a benchmark of CDSCHECKER). Relative to fence based, the solution found by our tool seems more restrictive: for example in our solution the load of flags of the while condition creates fences (when translated to intermediate code) both for when exiting the loop and when entering it; while in the fenced suggestion we know a fence appears only after the loop exit and not at the entrance to its body. What's more, where the fence placements do correlate, ours are still more restrictive. Perhaps due to incompleteness of our set of patterns and corrections.

For *Treiber's stack* algorithm CDSCHECKER had a synchronized verified version. For it, our suggestion was more restrictive then the manual one provided by CDSCHECKER.

VI. RELATED WORK

In this section, we review some closely related work, including synthesis of synchronization, automatic verification, bounded model checking, and dynamic analysis.

Fence Synthesis for x86-TSO and PSO Existing techniques for synthesizing synchronization for relaxed memory models has focused on hardware memory models. Kuperstein et al. [16] presented a framework for fence inference in hardware memory models such as PSO and TSO. Their framework is based on a simple operational semantics that explicitly tracks store-buffers to capture effects of the relaxed memory model. In later work, Kuperstein et al. [17] extended their technique using abstractions of unbounded store buffers. This allowed them to scale their technique and handle a larger set of algorithms. Abdulla et al. [3] infers memory fences for infinite-state programs under x86-TSO by combining predicate abstraction with abstractions of store buffers. Dan et al. [11] used an analysis based on numerical domains to synthesize minimal fence placements under PSO and TSO, utilizing various heuristic search optimizations to minimize the solution space. Our technique synthesize synchronization for C++ relaxed memory model. We note that the memory behavior under TSO and PSO is captured by the SB violation pattern.

Formalizing C++ RMM Batty et al. [8], [9] formalized the C++ RMM and proved correctness of compilation onto TSO

and Power [2]. These works inspired our definition of violation patterns and avoidance templates. We also drew intuition from the formal model in our generalization of the concrete violation patterns into abstract ones. Their tool, *CPPMEM*, bears some similarity to CDSCHECKER, which we use in our implementation. Thus, we believe that it would be possible to incorporate *CPPMEM* in our synthesis procedure.

Program Logics for C++RMM Vafeiadis et al. [25], [27] developed a Hoare style program logic verification which extends separation logic [21], [28] to C++ RMM. Batty et al. [7] provided an extension of linearizability and verified that an implementation of Treiber's stack [24] corresponds to an abstract stack under C++ RMM. These works allow for *manual* verification. Our synthesis procedure is based on Bounded Model Checking. However, if these works pan out to automatic verification techniques, it should be relatively straight forward to combine them with our technique as a final stage in which we verify the synthesized solutions.

Fence Synthesis for x86-TSO, PSO and IBM Power C++ RMM was developed with underlying hardware in mind, therefore the following works shed some light on the behaviors it allows. Joshi et al. in [14] introduced Reorder Bounded Model Checking. Their approach is based on instruction reordering, and their tool synthesize minimal fence placement. We, on the other hand, synthesize memory order synchronization. It would be interesting to see if our technique can be combined with theirs. *Musketeer*, developed by Algave et al. [4], provides a flexible scheme for fences synthesis to ensure robustness, i.e., that every concurrent execution be observationally equivalent to a serial execution. CheckFence of Burckhardt et al. [10], also ensures robustness by converting a program into a form that can be checked against an axiomatic model specification. Our technique allows to verify user-provided safety properties.

CONCLUSION

We present the first synthesis procedure for inferring efficient memory order synchronizations for C++ RMM which ensures that a program respects a user-provided safety property in bounded executions. We introduce a novel approach for detecting missing synchronization by searching for violation patterns, behaviors that are possible under C++ RMM but not under SC. We generalize concrete patterns to abstract ones, thus significantly improving the applicability of our approach because the abstract patterns allows to detect an infinite number of concrete patterns. We provide a technique to eliminate program executions that do not respect the given safety property by blocking the violation patterns they contain using generic avoidance patterns. We successfully synthesized non trivial memory order synchronization for several challenging concurrent algorithms, including a state of the art Read-Copy-Update (RCU) algorithm.

Our set of violation patterns and avoidance templates is not complete, thus our algorithm might fail to find any solution except the trivial one, where all memory operations are sequentially consistent. In fact, we believe that coming up with a complete set is non-trivial, if at all possible. We plan to address this challenge in future work.

REFERENCES

[1] http://www.practicalsynthesis.org/PSynSyn.html.
[2] IBM Power ISA v.2.05. 2007.
[3] ABDULLA, P. A., ATIG, M. F., CHEN, Y.-F., LEONARDSSON, C., AND REZINE, A. Automatic fence insertion in integer programs via predicate abstraction. SAS'12.
[4] ALGLAVE, J., KROENING, D., NIMAL, V., AND POETZL, D. Don't sit on the fence - A static analysis approach to automatic fence insertion. In *CAV* (2014), pp. 508–524.
[5] ANDREWS, G. R. *Concurrent programming - principles and practice*. Benjamin/Cummings, 1991.
[6] ARBEL, M., AND MORRISON, A. Predicate RCU: an RCU for scalable concurrent updates. In *PPoPP* (2015), pp. 21–30.
[7] BATTY, M., DODDS, M., AND GOTSMAN, A. Library abstraction for C/C++ concurrency. In *POPL* (2013), pp. 235–248.
[8] BATTY, M., OWENS, S., SARKAR, S., SEWELL, P., AND WEBER, T. Mathematizing C++ concurrency. In *POPL* (2011), pp. 55–66.
[9] BATTY, M. J. *The C11 and C++11 Concurrency Model*. PhD thesis, Wolfson College University of Cambridge, November 2014.
[10] BURCKHARDT, S., ALUR, R., AND MARTIN, M. M. K. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *PLDI* (2007).
[11] DAN, A. M., MESHMAN, Y., VECHEV, M. T., AND YAHAV, E. Effective abstractions for verification under relaxed memory models. In *VMCAI* (2015), pp. 449–466.
[12] DIJKSTRA, E. Cooperating sequential processes, TR EWD-123. Tech. rep., 1965.
[13] ISO/IEC. Programming Languages – C, 9899:2011.
[14] JOSHI, S., AND KROENING., D. Property-driven fence insertion using reorder bounded model checking. In *FM* (2015).
[15] KESSELS, J. L. W. Arbitration without common modifiable variables. *Acta informatica 17*, 2 (1982), 135–141.
[16] KUPERSTEIN, M., VECHEV, M., AND YAHAV, E. Automatic inference of memory fences. In *FMCAD* (2010).
[17] KUPERSTEIN, M., VECHEV, M., AND YAHAV, E. Partial-coherence abstractions for relaxed memory models. PLDI '11.
[18] LAMPORT, L. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM* (1974).
[19] LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. 28*, 9 (Sept. 1979), 690–691.
[20] NORRIS, B., AND DEMSKY, B. CDSchecker: checking concurrent data structures written with c/c++ atomics. In *OOPSLA* (2013).
[21] O'HEARN, P. W., REYNOLDS, J. C., AND YANG, H. Local reasoning about programs that alter data structures. In *CSL* (2001), pp. 1–19.
[22] PETERSON, G. L. Myths about the mutual exclusion problem. *Inf. Process. Lett. 12*, 3 (1981).
[23] TASSAROTTI, J., DREYER, D., AND VAFEIADIS, V. Verifying read-copy-update in a logic for weak memory. In *PLDI* (2015).
[24] TREIBER, R. K. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.
[25] TURON, A., VAFEIADIS, V., AND DREYER, D. Gps: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA* (2014), pp. 691–707.
[26] VAFEIADIS, V., BALABONSKI, T., CHAKRABORTY, S., MORISSET, R., AND ZAPPA NARDELLI, F. Common compiler optimisations are invalid in the c11 memory model and what we can do about it. In *POPL* (2015), pp. 209–220.
[27] VAFEIADIS, V., AND NARAYAN, C. Relaxed separation logic: a program logic for c11 concurrency. In *OOPSLA* (2013).
[28] VAFEIADIS, V., AND PARKINSON, M. J. A marriage of rely/guarantee and separation logic. In *CONCUR* (2007).

APPENDIX

A. Working with CDSCHECKER.

Handling satisfaction cycles pattern in CDSCHECKER is rather challenging: CDSCHECKER creates run traces by scheduling at each schedule point the next possible instruction according to Sequential Consistency. Each encountered load is relaxed and can read from stores visible to it according to RMM. (so: 1) maybe not the last store writing to the read memory location; 2) conditions dependant on values read-relaxed account for non SC traces.) One aspect of this, is that CDSCHECKER allows rf relations of loads only with stores that appeared before it in the run trace. This means that for instance the satisfaction cycle pattern will not be captured looking at the rf relation, since in every run trace, one load will appear before the store it reads from.

CDSCHECKER stores an additional set of promises for loads. Meaning that an encountered load is promised to read its value from a future store (this is done via backtracking). If the promise is satisfied later in the trace – the load reads from the future. In CDSCHECKER implementation, read from future is a set distinct from read from.

Busy wait: To support busy waits we invoked CDSCHECKER with a fairness condition where it restricts the number of consecutive times a load can see the same value. This fairness condition was not sufficient for *peterson* and *kessel* which had simultaneous waits on two values. Therefore, to ensure termination, we had to replace the busy wait with a bounded loop.

B. Intuition for RD patterns from classic patterns

Fig. 3 patterns list captures several behaviours of C++RMM and instances of those patterns were observed in almost all of our benchmarks but there are still C++RMM behaviours not captured by that list. What's more, the patterns share some similarities. In an attempt to brings us closer to completeness we drew on that resemblance and extracted the commonalities into abstract patterns.

For instance, non formally, in SB, MP and IRIW the *load x(0)* is reading from *store x,0* where under a more restrictive model it would read from *store x,1* that "appears later" (or respectively *load y(0)* missing a store *store y,1*). The meaning of "appears later" differs between the cases. For MP we can find a path of *sb* and *rf* edges from *load x(0)* to *store x,1* which continues to the initialization *store x,0*. For SB under any execution schedule either *load y(0)* or *load x(0)* is missing a `store` of 1.