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]. The crucial task of writing correct and efficient low-level concurrent programs in C++ under this model is known to be very challenging: the model's complexity is such that it eludes even veteran systems programmers and requires the attention of formal semantics experts [7], [8], [23], [26].

Under C++ RMM, each operation on an atomic object is annotated with a *memory order*. The memory order ranges from being fully *relaxed* to being fully *sequentially consistent* (for that atomic object), with a few more subtle modes between these two extremes. To maintain efficiency, the programmer wants the most relaxed synchronization required to preserve correctness, and nothing more (even when it simplifies reasoning). Unfortunately, manually finding the right synchronization is extremely difficult, as it requires the programmer to reason about subtle interactions 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 (vast) space of programs that can be created from P by modifying memory access synchronization. It does so by: (i) inspecting P's (bounded) error traces to detect memory access patterns that can occur under C++ RMM but not under SC, and (ii) eliminating 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. If it finds an error trace, it searches it 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, violations of S must be due to behaviors introduced by the weak memory model.) The algorithm then constructs a constraint which encodes all possible *avoidance templates* that can be used to eliminate that particular error trace. (*Avoidance templates* are strategies to synthesize *memory order annotations* of memory instructions 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 devising a complete set is nontrivial, if at all possible). This means that a program P' with 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 occur 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 satisfies S in bounded executions.

- An implementation of our approach and an 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: an *entry* section (lines 1–7) and an *exit* section (lines 9–10). The critical section itself (line 8) is irrelevant, and thus 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 whether the other thread is also trying to enter the critical section or is already in it (line 2). If not, 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: (i) as long as a thread executes the critical section, its flag is set to 1; and (ii) a thread enters the critical section only after it ensures that the other thread's flag is set to 0 while its own flag is set to 1. The above observation suffices to ensure mutual exclusion under SC, since, 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. Thus, if two threads compete on entering the critical section, at least one must notice in line 2 that the flag of the other 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 $\mathtt{r}_0$ and $\mathtt{r}_1$ are initialized to 2, and that $\mathtt{r}_0$ and $\mathtt{r}_1$ are each local to the respective thread.

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

Under *SC*, at the end of the program the only possible values of $\mathtt{r}_0$ and $\mathtt{r}_1$ are 0 and 1. Furthermore, at most one of them can be 0. Under *C++ RMM*, $\mathtt{r}_0$ and $\mathtt{r}_1$ can be 0 simultaneously, for certain memory order annotations W, X, Y, and Z. This is 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 behaviors).

The above example shows that mutual exclusion can only be ensured by adding synchronization to the program. One way to do it in C++ RMM is to explicitly annotate the load and store operations with the required synchronization type. Using strong synchronization primitives (e.g., requiring all load and store operations to be sequentially consistent) is expensive. Using synchronization primitives that are too weak, however, leads to unexpected behaviors. Thus, determining correct and efficient annotations is challenging. In contrast, our tool was able to determine 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 underlying model checker.

B. Synthesizing synchronization

Our approach rests on the insight that we can turn a program that is safe under SC into one that is safe under a weak memory model (C++ RMM in our case) by removing behaviors that cannot occur under SC. We face three main challenges in implementing this approach: (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 possible under C++ RMM. We allow safety properties to be specified as: (a) assertions on the final state, (b) properties of thread-local variables, and (c) races on non-atomic locations (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 possible 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 using a predefined avoidance template that 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

[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.

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, the number of memory annotations is finite, and every change only increases the degree of synchronization (see Section III).

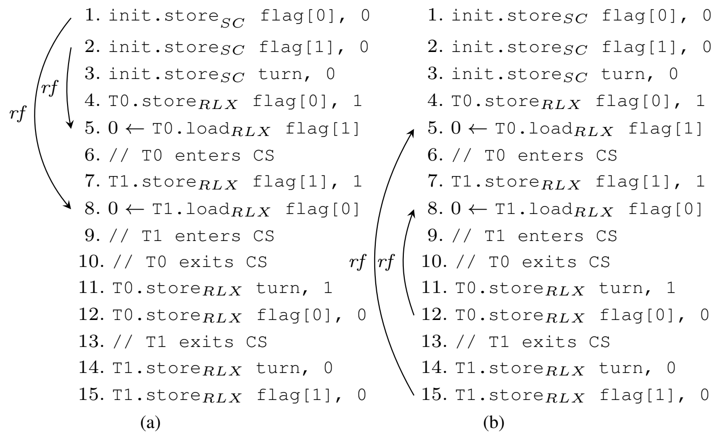
Example 2: Fig. 2 shows a trace of the Dekker algorithm that violates mutual exclusion. The trace contains two violation patterns, store buffering (SB) and load buffering (LB). The former, which we discussed in Example 1, is manifested here by the initialization `store` actions in lines 1 and 2, and 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 are 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.

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



```
 1. init.store_SC flag[0], 0
 2. init.store_SC flag[1], 0
 3. init.store_SC turn, 0
 4. T0.store_RLX flag[0], 1
 5. 0 ← T0.load_RLX flag[1]
 6. // T0 enters CS
 7. T1.store_RLX flag[1], 1
 8. 0 ← T1.load_RLX flag[0]
 9. // T1 enters CS
10. // T0 exits CS
11. T0.store_RLX turn, 1
12. T0.store_RLX flag[0], 0
13. // T1 exits CS
14. T1.store_RLX turn, 0
15. T1.store_RLX flag[1], 0
```

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

the initial state as if it were produced by explicit `store` operations.)

The C++ Relaxed Memory Model is relational: (i) without relations no order of executing instructions is guaranteed; and (ii) a load can read from arbitrary stores. In addition, the model distinguishes between *atomic* locations, where racy accesses are allowed, and *non-atomic* locations, where the behavior of races is undefined. The locations we discuss next will be *atomic*. Below, we provide a (greatly simplified) overview of the part of C++ RMM relevant to our work.

We shall use Fig. 3(SB) as a C++ Relaxed Memory execution trace example, though it was not intended as such and in Section IV-B will be referenced in a different context. Assume a two-threaded program where: variables `x` and `y` are initialized to zero; one thread sets the value of `x` to 1 and another sets the value of `y` to 1; finally, each thread reads the variable set by the other thread.

The first relation we consider is *read from* (rf), denoted by $\rightarrow_{rf}$, which relates `store` instructions to `load` instructions reading from them. The next relation we consider is *happens before* (hb), denoted by $\rightarrow_{hb}$. For our purpose it is a transitively-closed union of the following relations (in general, in C++RMM, hb can be non-transitive): (i) *sequence before* (sb), denoted by $\rightarrow_{sb}$, which places an irreflexive total order on the actions executed by the same thread; (ii) *additionally synchronized with* (asw), which relates instructions executed before thread creation to those executed by the thread, denoted by a dotted line separation; (iii) *synchronized with*(sw), which indicates instruction synchronizaion.

The model ensures that the only possible executions are ones in which these relations satisfy certain constraints. First, hb must be acyclic. Second, *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 $\texttt{store} \rightarrow_{hb} \texttt{store}_2 \rightarrow_{hb} \texttt{load}$. Third, the hb induced instruction order should not contradict the *modification order*, which defines a total order on all `store` operations to the *same* location.

Note 3: Note that in Fig. 3(SB) the aforementioned restrictions do not prevent reading values from initialization.

In addition, the hb relation should not contradict the *memory order annotation*. Every memory operation is annotated with a *memory order annotation* that specifies its consistency level: the level of synchronization and the ordering it requires. We consider three types of annotations:

(i) SC, whereby memory actions must be totally ordered;
(ii) ACQ/REL whereby a $\texttt{load}_{\text{ACQ}}$ that gets its value from a $\texttt{store}_{\text{REL}}$ imposes additional synchronization, and
(iii) RLX, whereby operations do not place additional restrictions on the hb relation.

Note 4: For item (ii) above, these annotations induce a sw relation, and for item (i) sc (total order) relation is induced.

IV. Synthesis of Synchronization

In this section we describe our synthesis algorithm (Section IV-A) and review the violation patterns and 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. Atomic 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 one 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 and 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 whether 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 $P \models S$ then return $\{P\}$
3      $\varphi$ = true
4      $\mathcal{P}$ = $\emptyset$
5      foreach $e \in$ errorTraces($P$,$S$) do
6          $\beta$ = blockOccurr($e$, AcqRelFix())
7          if $\beta$ then continue
8          $\beta$ = blockOccurr($e$, SCFix())
9          if $\neg\beta$ then return allSC($P$)
10         $\varphi = \varphi \wedge \beta$
11     $\varphi = \varphi \wedge \bigwedge impliedSync(\varphi)$
12     avoidance = SAT($\varphi$)
13     foreach $annotation \in avoidance$ do
14         $P'$ = addSync($P$,$annotation$)
15         $\mathcal{P} = \mathcal{P} \cup$ SynSync($P'$, $S$)
16     return $\mathcal{P}$

17 blockOccurr($e$,$patterns$)
18     $\beta$ = false
19     foreach $(p,c) \in patterns$ do
20         foreach $i \in$ occurrence($p$, $e$) do
21             $\beta = \beta \vee$ blockPattern($i$,$c$)
22     return $\beta$

23 $impliedSync(\varphi) = \{a \rightarrow b \mid a,b \in \texttt{vars}(\varphi)$
24                $\wedge\,(\texttt{SC} \in \texttt{annot}(a))$
25                $\wedge\,(\texttt{REL} \in \texttt{annot}(b) \vee \texttt{ACQ} \in \texttt{annot}(b))$
26                $\wedge\,(\texttt{instr}(a) == \texttt{instr}(b))\}$
```

Algorithm 1: The inner loop of the synthesis procedure.

```
1  Procedure PSynSync($P$, $S$, $N$)
2      $C_{init}, C_0, \ldots, C_m$ = getCmds($P$)
3      $\mathcal{P}_1$ = SynSync($C_{\text{init}}$ ; ($C_0 \parallel \ldots \parallel C_m$) , $S$)
4      for $n = 2$ to $N$ do
5          $\mathcal{P}_n$ = $\emptyset$
6          foreach $P' \in \mathcal{P}_{n-1}$ do
7              $C_{init}, C_0, \ldots, C_m$ = getCmds($P'$)
8              $Loop_0$ = "for $i_0 = 1 \ldots n$ do $C_0$"
9              ...
10             $Loop_m$ = "for $i_m = 1 \ldots n$ do $C_m$"
11             $P''$ = "$C_{init}$;($Loop_0 \parallel \ldots \parallel Loop_m$)"
12             $\mathcal{P}_n = \mathcal{P}_n \cup$ SynSync($P''$, $S$)
13     return $\mathcal{P}_N$
```

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 does it search 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 avoidance 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. 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 ways to eliminate the trace e. 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 by 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' complies with 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).

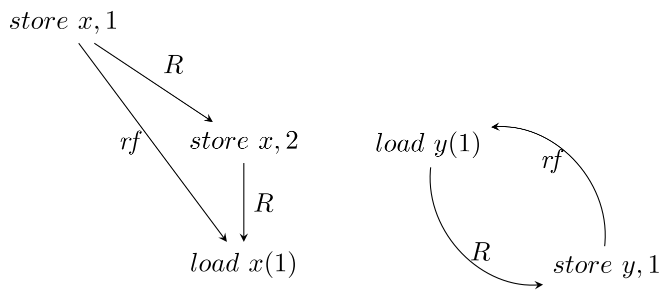
Algorithm 2 implements the outer loop of the synthesis procedure. For simplicity, we assume that the input program is comprised of an initialization command C_0 followed by a parallel composition of $m+1$ loops, where loop i repeats executing a sequential command C_i N times.

The algorithm takes the original program P, a specification S, and the loop bound N, and generates a set of programs $\mathcal{P}_N$ that restrict the synchronization in P so that 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 that refine P using different memory order synchronization such that P is compliant with S.

B. Patterns of weak memory behavior.

As mentioned previously, C++ RMM allows certain behaviors for a `load` that are not possible under SC. Below, we list some patterns of such behaviors and explain how they can be prevented using appropriate memory order annotations [7], [8]. The patterns can be seen in Fig. 3. We intuited the patterns from what are 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 is possible that each thread will not observe the `store` executed by the other. This behavior can occur when the stores of one thread are not immediately visible to the other.
Pattern prevention. This pattern can be prevented only by making all the `load` and `store` instructions SC.



(RD_1) (RD_2)

Fig. 4. Abstract patterns of behaviors 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 pattern 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 pattern 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 "magically" 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), but misses 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 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 pattern list captures several behaviors of C++RMM. Instances of those patterns were observed in almost all of our benchmarks but there are still C++RMM behaviors not captured by the previous list. What's more, the patterns share some similarities. In an attempt to bring us closer to completeness, we drew on that resemblance and extracted the commonalities into abstract patterns.

Using the 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

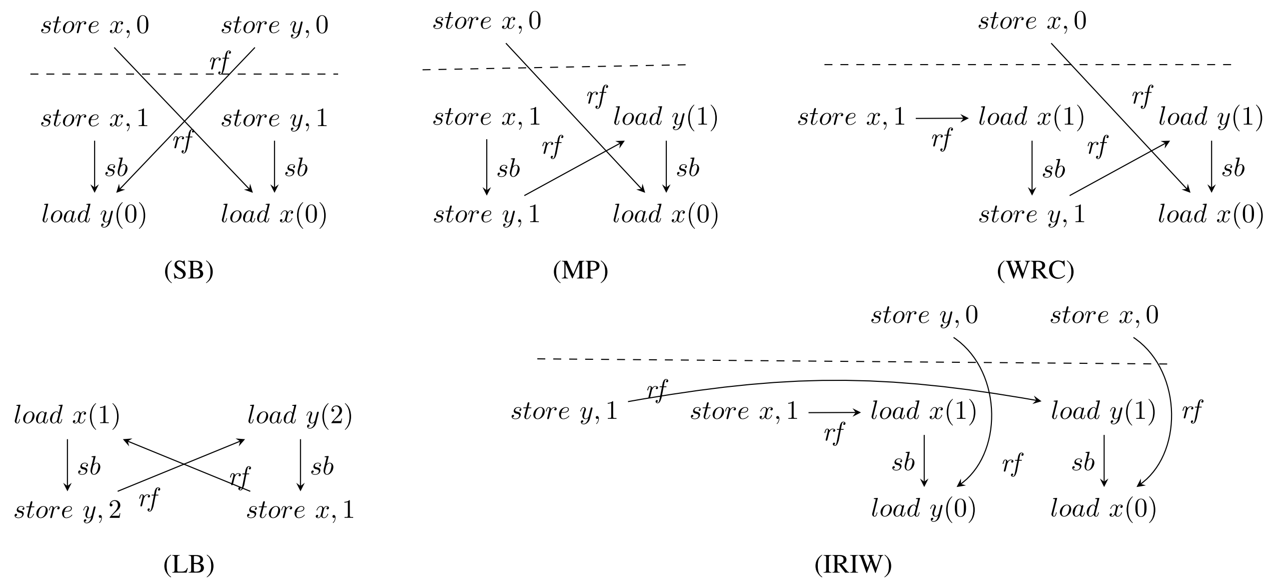


Fig. 3. Patterns of behaviors possible under C++ RMM but not under SC. Every column depicts the actions of one thread. We denote by *store x, 1* a write of value 1 to location x and by *load x(1)* a read of value 1 from x. We assume that the initial value of x and y is 0.

as a 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 the RD property in [7] and preventing that behavior.

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, 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 instructions involved in the pattern (making the memory order access SC) will cause the load to violate the RD property.

The following points should also be noted: 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 behavior.

V. IMPLEMENTATION AND EXPERIMENTAL EVALUATION

We implemented our approach in a tool called PSYNSYN, which is 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 nontrivial 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 and visual tools that explain our work are available at [1].

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

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

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

For *abp* we can see that the original algorithm was verified

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 SYNCHRONIZATION SYNTHESIS

when each process performed 1 iteration. It was not until each process performed 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 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, remained 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 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, thus turning it into the solution marked with "*". That solution later verified for 3 iterations.

For *bakery*, the 6 solutions 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 process and remained verified for the maximal number of iterations attempted.

Previous attempts (e.g. [26]) were made to verify RCU under C++RMM, but the version we verified was the first one where an update waits only for the reads whose consistency it affects, and does not wait for the completion of all existing reads.

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

For *Treiber's stack* algorithm, CDSCHECKER had a synchronized verified version. For it, our proposed solution was more restrictive than 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 have 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. They later [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] infer 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 synthesizes synchronization for the 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 intuited from the formal model when generalizing 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 technique that 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 fairly straightforward 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. The following works should therefore shed some light on the behaviors it allows. Joshi et al. [14] introduced Reorder Bounded Model Checking. Their approach is based on instruction reordering, and their tool synthesizes minimal fence placement. We, on the other hand, synthesize memory order synchronization. It would be interesting to see whether our technique can be combined with theirs. *Musketeer*, developed by Algave et al. [4], provides a flexible scheme for fence 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 makes it possible to verify user-provided safety properties.

CONCLUSION

We present the first synthesis procedure for inferring efficient memory order synchronizations for C++ RMM. Our procedure ensures that a program complies with a user-provided safety property in bounded executions. We introduce a novel approach for detecting missing synchronization by searching for violation patterns, behaviors 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 allow us to detect an infinite number of concrete patterns. We provide a technique to eliminate program executions that do not comply with the given safety property by blocking the violation patterns they contain using generic avoidance patterns. We successfully synthesized nontrivial 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, and 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 nontrivial, 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).