

# Towards the Synthesis of Coherence/Replication Protocols from Consistency Models via Real-Time Orderings

Vasilis Gavrielatos, Vijay Nagarajan, Panagiota Fatourou\*  
 The University of Edinburgh, \* University of Crete & FORTH-ICS  
 FirstName.LastName@ed.ac.uk, \* faturu@csd.uoc.gr

## Abstract

This work focuses on shared memory systems with a read-write interface (e.g., distributed datastores or multiprocessors). At the heart of such systems resides a protocol responsible for enforcing their consistency guarantees. Designing a protocol that correctly and efficiently enforces consistency is a very challenging task. Our overarching vision is to automate this task. In this work we take a step towards this vision by establishing the theoretical foundation necessary to automatically infer a protocol from a consistency specification. Specifically, we propose a set of mathematical abstractions, called *real-time orderings* (rt-orderings), that model the protocol. We then create a mapping from consistency guarantees to the minimal rt-orderings that enforce the guarantees. Finally, we informally relate the rt-orderings to protocol implementation techniques. Consequently, rt-orderings serve as an intermediate abstraction between consistency and protocol design, that enables the automatic translation of consistency guarantees into protocol implementations.

**CCS Concepts** • **Computer systems organization** → **Cloud computing; Multicore architectures; Software and its engineering** → **Consistency.**

**Keywords** Consistency; Coherence; Replication; Synthesis;

## 1 Introduction

This work focuses on “shared memory” systems that provide a read/write interface to the programmer. Such systems are ubiquitous in both computer architecture and distributed systems. Prominent examples include shared memory multiprocessors (SMPs), GPUs, NoSQL Databases [1], coordination services[9], and software-based DSMs [10].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PaPoC’21, April 26, 2021, Online, United Kingdom

© 2021 Association for Computing Machinery.

ACM ISBN 978-1-4503-8338-7/21/04...\$15.00

<https://doi.org/10.1145/3447865.3457964>

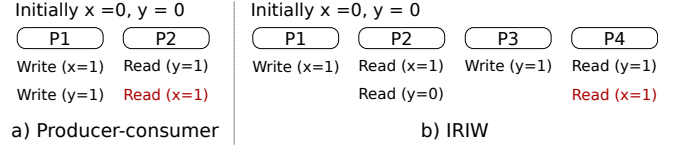


Figure 1: a) The producer-consumer synchronization pattern mandates that if P2 reads P1’s write to  $y$ , then it must also read P1’s write to  $x$ . b) The independent-reads independent-writes (IRIW) pattern mandates that if P2 sees P1’s writes but not P3’s and P4 sees P3’s write, then P4 must see P1’s write.

Such systems commonly replicate data – sometimes for performance (through caching), sometimes for fault tolerance and sometimes for both. To enable reasoning in the presence of replication, a *memory consistency model* (MCM) is specified as part of the system’s interface, providing the rules that dictate what values a read can return. In order to enforce the MCM, the system deploys a protocol which ensures that the replicas behave in accordance to the MCM. This protocol is called *coherence protocol* in computer architecture and *replication protocol* in distributed systems. We simply refer to it as *the protocol*.

The MCM specifies the behaviour of the system when executing parallel programs by enumerating all patterns through which parallel programs can synchronize. For example, an MCM  $CM_a$  can guarantee the synchronization pattern of Figure 1a (commonly known as “producer-consumer”). Specifically, in Figure 1a, process P1 writes object  $x$  and then  $y$ , while process P2 reads  $y$  and then  $x$ .  $CM_a$  guarantees that if P2’s read to  $y$  returns the write of P1, then P2’s read to  $x$  will also return the write of P1 (i.e.,  $x = 1$ ).

The MCM is thus a contract between the programmers and the designers. While programmers must understand the behaviour of the system, the designers must understand how to implement that behaviour. Problematically, enforcing the MCM is very challenging. For instance, consider two well known MCMs: TSO [18] and Causal Consistency (CC) [19]. TSO enforces both Figure 1a and b while CC enforces Figure 1a but not b. Given this information, how is the designer to implement a correct and efficient protocol for either MCM? Crucially, how is the designer to differentiate between the two models? E.g., how can one exploit that Figure 1b need not be enforced in the CC protocol?

Our overarching vision is to automate the above task: given a target MCM we envision a software tool that can produce an efficient protocol. We argue that to create this tool we need one additional level of abstraction which can act as an intermediary between the MCM and the protocol. That is: (1) we must be able to automatically translate any MCM to this abstraction; and (2) we must also be able to map the abstraction itself into protocol design choices. In this work, we take a step towards actualizing our vision, by proposing such an abstraction and, most crucially, presenting the mapping from MCMs to this abstraction. We also informally relate the abstraction to protocol design choices.

To create this abstraction, we observe that a shared memory system, be it a multiprocessor or a geo-replicated Key-Value Store (KVS), can be abstracted through the model of Figure 2, which depicts a set of processes executing a parallel program. Each process inserts its memory operations to a structure we call the *reorder buffer (ROB)*, allocating one ROB entry per operation. The *memory system* executes the operations it finds in the ROB and writes back the response of each operation in-place in its dedicated entry.

A process models a client of a KVS or a core of a multiprocessor. The ROB abstracts the core's load-store queue or a software queue that a KVS must maintain to keep track of incoming requests. The memory system is modelled as a distributed system comprising a set of *nodes*, where each node contains a controller and a memory. The nodes model the private caches of the multiprocessor or the geo-replicated memory servers of the KVS. Finally the network of the memory system can be thought of as the Network-on-Chip or as the WAN.

We observe that protocols of real systems enforce various consistency models using two widgets: 1) the ROB that allows for the reordering of operations; and 2) the memory system which determines how a read or a write executes, i.e., how it propagates to each of the replicas. We propose two sets of mathematical rules to abstract protocols, one that can abstract the reorderings of the ROB and one that can abstract the propagation rules of the memory system.

Specifically, we model the ROB using four rules named *program-order real-time orderings (prt-orderings)*. The  $prt_{wr}$  prt-ordering mandates that when write  $w$  precedes read  $r$  in a program execution, then  $r$  can begin executing in the memory system only after  $w$  has completed. Similarly, we define  $prt_{ww}$ ,  $prt_{rr}$ ,  $prt_{rw}$ , for the rest of the combinations between reads and writes. To summarize, we model an ROB by specifying the subset of the four prt-orderings it enforces.

We model the memory system using four rules named *synchronization real-time orderings (srt-orderings)*. The  $srt_{wr}$  srt-ordering mandates that if write  $w$  to object  $x$  completes before read  $r$  to object  $x$  in real time, then  $r$  must observe  $w$ . Similarly, we define  $srt_{ww}$ ,  $srt_{rr}$ ,  $srt_{rw}$ . To summarize we model a memory system by specifying the subset of the four srt-orderings it enforces.

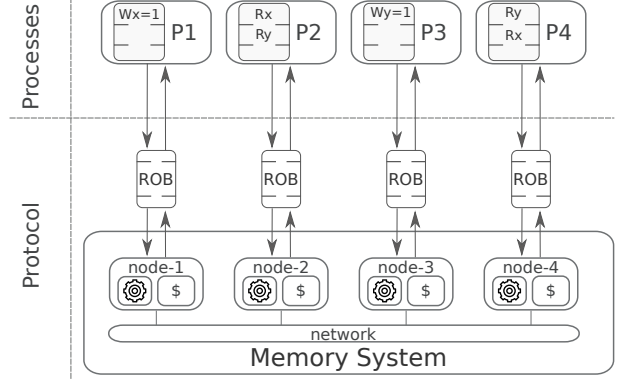


Figure 2: The model of a shared memory system. Processes P1-P4 execute the IRIW pattern of Figure 1b.

We refer to prt-orderings and srt-orderings as *real-time orderings (rt-orderings)*. The eight rt-orderings comprise our designer-centric intermediate abstraction of the protocol.

While the rt-orderings have been employed by other works in the past to model consistency models or protocols (§9), this work is the first to present a mapping from MCMs to the rt-orderings. That is, given any MCM, we provide the set of minimal rt-orderings that enforce it. Crucially, this mapping, along with relating the rt-orderings with protocol implementation techniques, pave the way for automating protocol design.

Using this mapping from MCMs to rt-orderings, we now revisit the questions we posed on Figure 1. Specifically, prt-orderings  $prt_{ww}$  and  $prt_{rr}$  and the srt-ordering  $srt_{wr}$  suffice to enforce the producer-consumer pattern (discussed in Section 5.4). Informally,  $prt_{ww}$  mandates that writes from the same process must be executed in the order intended by the program;  $prt_{rr}$  mandates the same for reads;  $srt_{wr}$  mandates that a read must be able to return the value of the latest completed write (from any process). To also enforce the IRIW pattern,  $srt_{rr}$  is required (discussed in Section 5.5). Informally,  $srt_{rr}$  mandates that if a read returns the value of write  $w$ , then any later read must also be able to return the value of write  $w$ .

**Contributions.** In summary, in this work we make the following contributions:

- We propose and formalize eight rt-orderings for mathematically modelling consistency enforcement protocols, serving as an intermediate abstraction between the MCM and the protocol implementation (§3).
- We create a mapping from MCMs to rt-orderings, specifying the minimal set of rt-orderings sufficient to enforce any MCM (§5) and vice versa, specifying the MCM enforced by any set of rt-orderings (§6).
- We informally map rt-orderings to protocol implementation techniques (§7).

## 2 Preliminaries

In this section, we first establish the system model, then we describe the notation we will use throughout the paper and finally we discuss how executions are modeled.

### 2.1 System Model

Figure 2 illustrates our system model. Specifically, there is a set of  $P$  processes, each of which executes a program and there is a *protocol* which enforces the MCM. The *protocol* is comprised of a set of *reorder buffers* (ROBs) and the *memory system*. The *memory system* stores a set  $X$  of shared *objects*, each of a unique name and value. A memory operation (or simply *operation*) can be either a read or a write of an object. A read returns a value and a write overwrites the value of the object and returns an ack.

**ROBs.** The ROBs are used to facilitate communication between the processes and the protocol. There are three events in the lifetime of an operation within the ROB: *issuing*, *begin* and *completion*. Specifically, we write that a process “issues an operation”, when the operation is first inserted in the ROB. A process pushes operations in the ROB in the order that they are encountered within its program. We refer to this order as the *program order*. Furthermore, we write that “an operation begins” when the memory system begins executing the operation. In this case the memory system marks the operation’s ROB entry as *beginned*. Similarly, we write that “an operation completes” when the memory system has finished executing the operation. In this case, the memory system marks the operation’s ROB entry as *completed* (and writes back the result if it is a read).

Notably, a process can use the value returned by a read, as soon as the read is marked *completed* by the memory system, irrespective of whether preceding operations are completed. An operation is removed from the ROB, if all three conditions hold: 1) it is the oldest operation, 2) it is completed and 3) the process has consumed its value, if it is a read.

**Memory system.** We model the memory system as a distributed system comprised of a set of *nodes*, where each node contains a controller and a memory. Furthermore, each node is connected to one ROB, from which it reads the operations that must be executed. The controller of each node is responsible for the execution of the operations. The memory of each node stores every object in  $X$ .

### 2.2 Notation

The notation used throughout this paper is the one used by Algave et al. [2]. We repeat it here for completeness. The notation is based on relations. Specifically, we will denote the transitive closure of  $r$  with  $r^*$  and the composition of  $r_1, r_2$  with  $r_1; r_2$ . We say that  $r$  is *acyclic* if its transitive closure is irreflexive and we denote by writing *acyclic*( $r$ ). Finally, we say that  $r$  is a *partial order*, when  $r$  is transitive and irreflexive.

We say that a partial order  $r$  is a *total order* over a set  $S$ , if for every  $x, y$  in  $S$ , it is that  $(x, y) \in r \vee (y, x) \in r$ .

We use small letters to refer to memory operations (e.g.,  $a, b, c$  etc.). In figures that show executions (e.g., Figure 3), we denote a write with  $Wx = val$  where  $x$  is the object to be written and  $val$  is the new value. Similarly, reads are represented as  $Rx = val$ , where  $val$  is the value returned. When required relations between operations are denoted with red arrows in figures.

### 2.3 Modelling executions

To model executions, we use the framework created by Algave et al. [2], introducing minor changes where necessary. Specifically, we model an execution as a *tuple*( $M, po, rf, hb, RL$ ).  $M$  is the set of memory operations included in the execution,  $po, rf$  and  $hb$  are relations over the operations and  $RL$  is a set of relations  $rl$  over the operations.

**Execution relations ( $po, rf, hb$  and  $rl$ ).** The *program order*  $po$  relation is a total order over the operations issued by a single process, specifying the order in which memory operations are ordered in the program executed by the process. Operations are issued by a process in this order. Only operations from the same process are ordered. For two operations  $a, b$  from process  $i \in P$ , if  $a$  is issued before  $b$  then  $(a, b) \in po$ .

The *reads-from*  $rf$  relation contains a pair for each read operation in  $M$ , relating it with a write on the same object. For the pair  $(a, b) \in rf$ , it must be that  $a$  is a read that returns the value created by the write  $b$ , i.e.,  $a$  *reads-from*  $b$ .

The *happens-before*  $hb$  relation is a partial order over all operations, specifying the real-time relation of operations. Specifically, for two operations  $a, b$ , if  $(a, b) \in hb$ , then that means that  $a$  completes before  $b$  begins.

Finally, the *read-legal*  $rl$  relation is a total order over all operations, with the restriction that *acyclic*( $rl \cup rf$ ). Given the  $rf$  of an execution  $E$ , it is often possible to construct more than one  $rl$ . We associate each execution  $E$ , with a set  $RL$  which contains every  $rl$  that can be constructed from  $E$ . For each  $rl \in RL$ , we derive the following relations.

**Relations derived from  $rl$  ( $ws, fr, syn$ ).** The *write-serialization*  $ws$  relation is a total order of writes to the same object that can be derived from  $rl$ , such that  $ws \subseteq rl$ . Intuitively,  $ws$  captures the order in which writes to the same object serialize.

The *from-reads*  $fr$  relation connects a read to writes from the same object that precede the read in  $rl$ . Specifically, for every read  $a$  and a write  $b$  to the same object where  $(a, b) \in rl$ , then  $(a, b) \in fr$ . It follows that  $fr$  is transitive and  $fr \subseteq rl$ .

Finally, the *synchronization*  $syn$  relation combines the  $rf, fr$  and  $ws$  relations. Specifically,  $syn$  is a partial order over operations on the same object defined as the transitive closure of the union of  $rf, ws$  and  $fr$ , i.e.,  $syn \triangleq (rf \cup ws \cup fr)^*$ . Two operations on the same object are said to synchronize if

one of them is a write. Formally, for any pair of operations  $(a, b)$  on the same object, if at least one of  $a, b$  is a write then it must be that  $(a, b) \in \text{syn} \vee (b, a) \in \text{syn}$ . If both  $a, b$  are reads, then  $(a, b) \in \text{syn}$  iff there exists a write  $c$ , such that  $(a, c) \in \text{syn} \wedge (c, b) \in \text{syn}$ .

**Helper relations (po-type, syn-type, hb-type).** Given the set of operations  $M$ , we define four relations  $WW, WR, RR, RW$  over  $M$  that contain all write-write, write-read, read-read and read-write pairs found in  $M$ , respectively. Table 2 uses these four relations to define the *po-type*, *syn-type* and *hb-type* relations. Specifically, by taking the intersection of  $po$  with each of  $WW, WR, RR, RW$ , we define  $po_{ww}, po_{wr}, po_{rr}, po_{rw}$ . We write *po-type* as a placeholder that can be replaced by any of these four relations. Notably, every pair in  $po$  is also in one of the *po-type* relations. I.e.,

$$po \triangleq po_{ww} \cup po_{wr} \cup po_{rr} \cup po_{rw}$$

Similarly, we define  $\text{syn}_{wr}$  and  $\text{syn}_{rr}$ . Note that we do not need to define  $\text{syn}_{ww}$  and  $\text{syn}_{rw}$ , because they would be the same as  $ws$  and  $fr$ , respectively. We write *syn-type* as a placeholder for  $ws, \text{syn}_{wr}, \text{syn}_{rr}, fr$ . We note that every pair in  $\text{syn}$  is also in one of the *syn-type* relations and that  $rf$  is a subset of  $\text{syn}_{wr}$ .

$$\begin{aligned} \text{syn} &\triangleq ws \cup \text{syn}_{wr} \cup \text{syn}_{rr} \cup fr \\ rf &\subseteq \text{syn}_{wr} \end{aligned}$$

Finally, in the same spirit, we define the *hb-type* relations:  $hb_{ww}, hb_{wr}, hb_{rr}, hb_{rw}$ .

### 3 Real-time orderings

In this section we introduce and formally define the *real-time orderings* (rt-orderings), through which we model the protocol. There are two types of rt-orderings: the program-order real-time orderings (*prt-orderings*) through which we model the operation of the ROB and the synchronization real-time orderings (*srt-orderings*), through which we model the operation of the memory system. Table 1 provides the definition of each of the eight rt-orderings. Below we first describe the *prt-orderings* and then the *srt-orderings*.

#### 3.1 Program-order Real-time Orderings

An execution  $E(M, po, rf, hb, RL)$  is said to enforce the  $p_{rt_{wr}}$  if:

$$\forall w, r \in M \text{ s.t. } (w, r) \in po_{wr} \rightarrow (w, r) \in hb_{wr}$$

Plainly, a read  $r$  cannot begin before all writes that precede it in program order have completed. Table 1 extends this definition to write-write, read-read and read-write pairs. When all four *prt-orderings* are enforced then it follows that  $po \subseteq hb$ .

#### 3.2 Synchronization Real-time Orderings

The synchronization real-time orderings (*srt-orderings*) are constraints over operations to the same object. There are four *srt-orderings*:  $s_{rt_{ww}}, s_{rt_{wr}}, s_{rt_{rr}}, s_{rt_{rw}}$ . An execution  $E(M,$

prt-orderings		srt-orderings	
$p_{rt_{ww}}$	$po_{ww} \subseteq hb_{ww}$	$s_{rt_{ww}}$	$\text{acyclic}(hb_{ww} \cup \text{syn})$
$p_{rt_{wr}}$	$po_{wr} \subseteq hb_{wr}$	$s_{rt_{wr}}$	$\text{acyclic}(hb_{wr} \cup \text{syn})$
$p_{rt_{rr}}$	$po_{rr} \subseteq hb_{rr}$	$s_{rt_{rr}}$	$\text{acyclic}(hb_{rr} \cup \text{syn})$
$p_{rt_{rw}}$	$po_{rw} \subseteq hb_{rw}$	$s_{rt_{rw}}$	$\text{acyclic}(hb_{rw} \cup \text{syn})$

Table 1: The condition required to enforce each rt-ordering.

po-type	syn-type	hb-type
$po_{ww} \triangleq po \cap WW$	$ws \triangleq \text{syn} \cap WW$	$hb_{ww} \triangleq hb \cap WW$
$po_{wr} \triangleq po \cap WR$	$\text{syn}_{wr} \triangleq \text{syn} \cap WR$	$hb_{wr} \triangleq hb \cap WR$
$po_{rr} \triangleq po \cap RR$	$\text{syn}_{rr} \triangleq \text{syn} \cap RR$	$hb_{rr} \triangleq hb \cap RR$
$po_{rw} \triangleq po \cap RW$	$fr \triangleq \text{syn} \cap RW$	$hb_{rw} \triangleq hb \cap RW$

Table 2: The types of po, syn and hb relations

$po, rf, hb, RL$ ) enforces the *srt-ordering*  $s_{rt_{wr}}$  if there exists  $rl \in RL$  such that:

$$\text{acyclic}(hb_{wr} \cup \text{syn})$$

In other words the  $s_{rt_{wr}}$  mandates that for a write  $a$  and a read  $b$  if  $(a, b) \in hb_{wr}$ , then it must be that  $(b, a) \notin \text{syn}$ . Table 1 describes the condition required to enforce each of the *srt-orderings*. A memory system is said to enforce an *srt-ordering*, if that *srt-ordering* is enforced in every execution that can be generated by the memory system. Notably, a combination of the *srt-orderings* can be enforced, for example  $s_{rt_{ww}}, s_{rt_{wr}}$  are both enforced when  $\text{acyclic}(\text{syn} \cup hb_{ww} \cup hb_{wr})$ . When all four *srt-orderings* are enforced then linearizability is enforced [8] (we expand on this on Section 8.1).

### 4 Memory consistency models (MCMs)

In order to create a mapping from memory consistency models (MCMs) to the rt-orderings we need to establish a formalism for the MCMs. In this section, we use the formalism of Alglave et al. [2] to describe *synchronization patterns* (sync-pats) and assert that any MCM  $CM$  is defined as a set of sync-pats  $S_{CM}$ . An execution enforces  $CM$  iff it enforces every sync-pat in  $S_{CM}$ . Below we define what a sync-pat is and what its enforcement entails.

A sync-pat is a path between two operations to the same object. The path can be constructed through any composition of *po-type* and *syn-type* relations, with the only restriction being that at least one *po-type* must be included (explained in the remarks below). Table 2 describes which relations are denoted *po-type* and *syn-type*.

For example, consider the sync-pat  $s$  that consists of a  $po_{wr}$  relation, then an  $fr$  relation and then a  $po_{wr}$  (depicted in Figure 3a). We can describe  $s$  by composing the three relations as follows:

$$s \triangleq po_{wr}; fr; po_{wr}$$



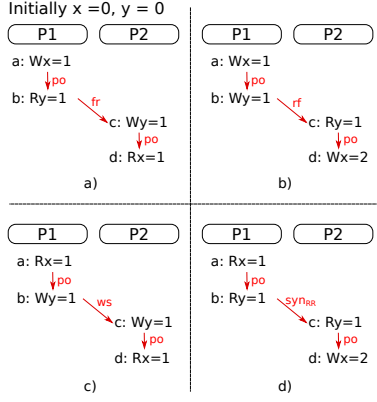


Figure 3: Four examples of sync-pats

Given an execution  $E(M, po, rf, hb, RL)$ , we assert that  $E$  enforces  $s$  if there exists  $rl \in RL$  from which we can derive a  $syn$  such that:

$$acyclic(s \cup syn)$$

In other words, a sync-pat that starts from operation  $a$  and ends on operation  $b$  is said to be *enforced* if  $(b, a) \notin syn$ .

Figure 3 depicts four sync-pats to serve as examples. In the execution of Figure 3a, the  $s$  sync-pat occurs between operations  $a$  and  $d$ . In this instance,  $s$  is enforced because  $d$  returns the value created by  $a$ . If  $d$  were to return 0, that would mean that there is a  $syn$  relation between  $d$  and  $a$  and thus  $s$  is not enforced.

**Remarks.** Note the following two remarks. First, we use  $syn$  rather than  $rl$  to test whether a sync-pat is enforced. This is because the sync-pat is a path between two operations to the same object, but  $rl$  is a total order of all operations across all objects. Second, note that a sync-pat must have at least one  $po$ -type, because otherwise it would just be a composition of  $syn$ -type relations. Any such composition is a subset of  $syn$ , which means that the execution enforces it by definition.

**Consistency models.** An MCM  $CM_i$  is defined by asserting that a set of sync-pats  $S_{CM}$  must be enforced. Plainly, an execution enforces  $CM_i$  iff it enforces every  $s \in S_{CM}$ . As an example, assume that  $CM_i$  is defined through the four sync-pats depicted in Figure 3, which are formalized as follows:

$$\begin{aligned} s_1 &\triangleq po_{wr}; fr; po_{wr}, \quad s_2 \triangleq po_{ww}; rf; po_{rw}, \\ s_3 &\triangleq po_{rw}; ws; po_{wr}, \quad s_4 \triangleq po_{rr}; syn_{rr}; po_{rw} \end{aligned}$$

We define  $CM_i$  to be the following rule:

$$\begin{aligned} &acyclic(s_1 \cup syn) \wedge acyclic(s_2 \cup syn) \\ &\wedge acyclic(s_3 \cup syn) \wedge acyclic(s_4 \cup syn) \end{aligned}$$

## 5 From MCMs to rt-orderings

In this section, we provide a mapping from MCMs to rt-orderings. Specifically, given an MCM  $CM_i$  that is specified

through a set  $S_{CM}$  of sync-pats, we automatically infer a set of rt-orderings, sufficient to enforce  $CM_i$ . In the rest of this section, we first differentiate between regular and irregular sync-pats (§5.1). Then we provide the mapping (§5.2), prove its correctness for an example sync-pat (§5.3), extend the proof to all regular sync-pats (§5.4) and finally extend the proof to irregular sync-pats (§5.5). In Table 3, we provide the mapping of 16 sync-pats to rt-orderings.

### 5.1 Regular and irregular sync-pats

We first categorize sync-pats into two classes: *regular* and *irregular*. A regular sync-pat is 1) composed by alternating  $po$ -type and  $syn$ -type relations and 2) starts and ends on a  $po$ -type. Thus, any regular sync-pat  $s$ , is of the following form:

$$s \triangleq po\text{-type}; syn\text{-type}; po\text{-type}; \dots syn\text{-type}; po\text{-type}$$

Any sync-pat that does not conform to regularity rules is *irregular*. The distinction between regular and irregular sync-pats will aid in the presentation of the mapping from sync-pats to rt-orderings. Specifically, we will first create a mapping from a regular sync-pat to the sufficient rt-orderings to enforce it. Then, in order to use the same mapping for irregular sync-pats, we will show that for every irregular sync-pat  $s_i$ , there exists a regular sync-pat  $s_r$ , such that enforcing  $s_r$  is sufficient to enforce  $s_i$ .

### 5.2 The mapping

We assert the following three conditions to enforce any regular sync-pat, assuming that an operation can be of type  $m$  or  $n$  where  $m, n \in \{read, write\}$ .

- **Cond-1:** for every  $po$ -type relation  $po_{mn}$  found in  $s$ , the corresponding prt-ordering  $prt_{mn}$  must be enforced. Plainly any  $po$ -type relation must also be an  $hb$  relation.
- **Cond-2:** for every  $syn$ -type relation  $syn_{mn}$ , if it is not an  $rf$ , then the reverse srt-ordering  $srt_{nm}$  must be enforced.
- **Cond-3:** if the first operation in  $s$  is of type  $m$  and the last of type  $n$ , the corresponding srt-ordering  $srt_{mn}$  must be enforced.

### 5.3 Proof for an example sync-pat

We will start by first proving that the three conditions are sufficient for the simple sync-pat, that is portrayed in Figure 3a. Then we will extend to any regular sync-pat. The simple sync-pat  $s$  is the following:

$$s \triangleq po_{wr}; fr; po_{wr}$$

For  $s$  the three conditions require the following rt-orderings:

- **Cond-1:** The  $prt_{wr}$  is required for both  $po_{wr}$  relations.
- **Cond-2:** The  $srt_{wr}$  for the  $fr$  (i.e.,  $syn_{rw}$ ) relation.
- **Cond-3:** The  $srt_{wr}$  because  $s$  starts with a write and ends on a read.

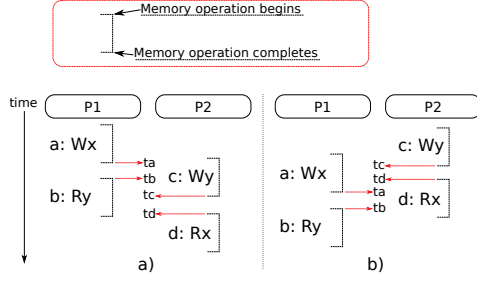


Figure 4: a)  $b$  begins before  $c$  completes b)  $c$  completes before  $b$  begins.

We will show that for any execution  $E(M, po, rf, hb, RL)$ , if  $p_{rt_{wr}}$  is enforced and there exists  $rl \in RL$  such that  $s_{rt_{wr}}$  is enforced then for the  $syn$  that is derived from that  $rl$  it holds that  $acyclic(s \cup syn)$ .

To prove this, we first establish a new relation named *begins-before-completes* ( $bbc$ ). For two operations  $a, b$  we assert that if  $b$  does not complete before  $a$  begins (i.e.,  $(b, a) \notin hb$ ), then  $a$  begins before  $b$  completes (i.e.,  $(a, b) \in bbc$ ). We establish the following rule:

$$\forall a, d \in M \text{ s.t. } (a, d) \in (hb; bbc; hb) \rightarrow (a, d) \in hb$$

We sketch a proof for this rule using Figure 4a. Figure 4a illustrates this through four operations  $(a, b, c, d)$ , each of which is associated with a timestamp  $(ta, tb, tc, td)$ . Operation  $b$  begins before  $c$  completes (i.e.,  $(b, c) \in bbc$ ), while also  $(a, b), (c, d) \in hb$ . Therefore,  $(a, d) \in (hb; bbc; hb)$ . Since  $a$  completes before  $b$  begins ( $ta < tb$ ) and  $b$  begins before  $c$  completes ( $tb < tc$ ), it follows that  $a$  completes before  $c$  completes ( $ta < tc$ ). Because  $c$  completes before  $d$  begins ( $tc < td$ ), it follows that  $a$  completes before  $d$  begins ( $ta < td$ ). Therefore,  $(a, d) \in hb$ .

Figure 4b illustrates the counter-example, where  $c$  completes before  $b$  begins; in this case it is possible for  $d$  to begin before  $a$  completes.

Consider an execution  $E(M, po, rf, hb, RL)$ , for which there exists an  $rl \in RL$  such that all three conditions we asserted are satisfied. Assume four operations  $a, b, c, d \in M$  such that  $(a, b) \in po_{wr} \wedge (b, c) \in syn_{wr} \wedge (c, d) \in po_{wr}$ . This implies that  $(a, d) \in s$ . Let us now assume that  $(a, d) \in (hb; bbc; hb)$  and thus  $(a, d) \in hb$ . Specifically,  $(a, d) \in hb_{wr}$ . This is illustrated in Figure 4a. From the cond-3 ( $s_{rt_{wr}}$ ), we can assert that since  $(a, d) \in hb_{wr}$  it follows that  $(d, a) \notin syn$  and thus it must be that  $acyclic(s \cup syn)$ . Plainly, cond-3 ensures that  $acyclic(s \cup syn)$  if the following condition holds:

$$\forall a, d \in M \text{ s.t. } (a, d) \in s \rightarrow (a, d) \in hb$$

Therefore it suffices to prove that cond-1 and cond-2 guarantee the above condition. We will do so by proving that  $(a, d) \in (hb; bbc; hb)$ .

Firstly, cond-1 ( $p_{rt_{wr}}$ ) mandates that  $po_{wr} \subseteq hb_{wr}$ . Therefore,  $(a, b), (c, d) \in hb$ . We need only prove that  $(b, c) \in bbc$ . Let's assume that  $(b, c) \notin bbc$ . It follows that  $(c, b) \in hb$ . Recall, that  $b$  is a write and  $c$  is a read. Cond-2 mandates

that  $s_{rt_{wr}}$  is enforced: since  $(c, b) \in hb$  then  $(b, c) \notin syn$ . We have reached a contradiction, and therefore it must be that  $(b, c) \in bbc$ .

Figure 4b provides the counter example where  $(b, c) \notin bbc$ . In this case it cannot be that  $(a, d) \in s$  because  $(b, c) \in hb$  and therefore from cond-2 it follows that  $(c, b) \in syn$ . Plainly, the sync-pat  $s$  will not occur here because  $P1$ 's read to  $y$  will observe  $P2$ 's write to  $y$ .

As we have shown above, from cond-1 and cond-2 it follows that  $(a, d) \in hb$  and thus from cond-3 it follows that  $(a, d) \notin syn$ . Therefore the three conditions are sufficient to enforce  $s$ .

#### 5.4 Extending to all regular sync-pats

Note the intuition behind the three conditions. Cond-1 ensures that every *po-type* relation of the sync-pat is also an *hb* relation. Similarly, cond-2 ensures that every *syn-type* relation is also a *bbc* relation. As a regular sync-pat is composed by alternating *po-type* and *syn-type* relations, enforcing cond-1 and cond-2 ensures that the sync-pat can be expressed as a composition of alternating *hb* and *bbc* relations.

As we saw  $hb; bbc; hb \subseteq hb$ . By induction we can extend this rule for any sequence of alternating *hb* and *bbc* relations. Specifically:

$$\forall a, b \in M \text{ s.t. } (a, b) \in hb; bbc; hb \dots hb; bbc; hb \rightarrow (a, b) \in hb$$

Therefore, cond-1 and cond-2 ensure that if  $(a, b) \in s$  then  $(a, b) \in hb$  for any regular sync-pat  $s$ . Finally, cond-3 ensures that the sync-pat is enforced, by ensuring that if  $(a, b) \in hb$  then  $(b, a) \notin syn$ . This means that our three conditions can be used to enforce any regular sync-pat.

**Examples – Table 3.** Table 3 depicts the sufficient rt-orderings for 16 sync-pats. Specifically, each cell represents a distinct sync-pat between operations  $a, b, c, d$ . For instance, the highlighted cell where  $a = Wx, b = Ry, c = Wy, d = Rx$ , corresponds to the sync-pat of Figure 3a.

**Cond-2 exception: rf.** Recall that cond-2 is not required for *rf* edges. This is because the purpose of cond-2 is to ensure that the *syn-type* relation is also a *bbc* relation. However, this is implied by the *rf* as it is impossible for a read  $r$  to read-from a write  $w$  if  $r$  completes before  $w$  begins. Plainly:  $\forall w, r \in M \text{ s.t. } (w, r) \in rf \rightarrow (w, r) \in bbc$ .

**Producer-consumer (Figure 1a).** Let  $s_{pc}$  be the producer-consumer sync-pat of Figure 1a (discussed in the Introduction). We assert that  $s_{pc} \triangleq po_{ww}; rf; po_{rr}$ . To enforce  $s_{pc}$ , cond-1 requires  $p_{rt_{ww}}, p_{rt_{rr}}$ , cond-2 does not require any  $s_{rt}$ -ordering because the only *syn-type* is an *rf* and cond-3 requires the  $s_{rt_{wr}}$ .

#### 5.5 Extending to irregular sync-pats

A sync-pat is deemed *irregular* if 1) it has consecutive *po-type* relations or 2) it has consecutive *syn-type* relations or 3) it does not start with a *po-type* relation (i.e., starts with

	c = Wy d = Ry	c = Wy d = Wy	c = Ry d = Ry	c = Ry d = Wy
a = Wx b = Rx	<i>srt<sub>wr</sub></i>	<i>srt<sub>ww</sub>, srt<sub>wr</sub></i>	<i>srt<sub>wr</sub>, srt<sub>rr</sub></i>	<i>srt<sub>ww</sub>, srt<sub>rr</sub></i>
	<i>prr<sub>wr</sub></i>	<i>prr<sub>wr</sub>, prr<sub>ww</sub></i>	<i>prr<sub>wr</sub>, prr<sub>rr</sub></i>	<i>prr<sub>wr</sub>, prr<sub>rw</sub></i>
a = Wx b = Wx	<i>srt<sub>wr</sub>, srt<sub>ww</sub></i>	<i>srt<sub>ww</sub></i>	<i>srt<sub>wr</sub>, srt<sub>rw</sub></i>	<i>srt<sub>ww</sub>, srt<sub>rw</sub></i>
	<i>prr<sub>ww</sub>, prr<sub>wr</sub></i>	<i>prr<sub>ww</sub></i>	<i>prr<sub>ww</sub>, prr<sub>rr</sub></i>	<i>prr<sub>ww</sub>, prr<sub>rw</sub></i>
a = Rx b = Rx	<i>srt<sub>rr</sub>, srt<sub>wr</sub></i>	<i>srt<sub>rw</sub>, srt<sub>wr</sub></i>	<i>srt<sub>rr</sub></i>	<i>srt<sub>rw</sub>, srt<sub>rr</sub></i>
	<i>prr<sub>rr</sub>, prr<sub>wr</sub></i>	<i>prr<sub>rr</sub>, prr<sub>ww</sub></i>	<i>prr<sub>rr</sub></i>	<i>prr<sub>rr</sub>, prr<sub>rw</sub></i>
a = Rx b = Wx	<i>srt<sub>rr</sub>, srt<sub>ww</sub></i>	<i>srt<sub>rw</sub>, srt<sub>ww</sub></i>	<i>srt<sub>rr</sub>, srt<sub>rw</sub></i>	<i>srt<sub>rw</sub></i>
	<i>prr<sub>rr</sub>, prr<sub>wr</sub></i>	<i>prr<sub>rw</sub>, prr<sub>ww</sub></i>	<i>prr<sub>rr</sub>, prr<sub>rr</sub></i>	<i>prr<sub>rw</sub></i>

Table 3: The mapping of 16 sync-pats to rt-orderings. Each cell represents a sync-pat of the form  $s \triangleq po\text{-type}; syn\text{-type}; po\text{-type}$ , where the first *po-type* includes  $(a, b)$ , the *syn-type* includes  $(b, c)$  and the second *po-type* includes  $(c, d)$ . The highlighted cell corresponds to Figure 3a.

*syn-type*) or 4) it does not end with a *po-type* relation (i.e., ends with *syn-type*). For each such sync-pat  $s$ , we derive a sync-pat  $s'$  such that 1)  $s'$  is regular and 2) if  $s'$  is enforced then  $s$  must also be enforced.

**Consecutive po-type.** We start with sync-pats with consecutive *po-type* relations. We use the insight that any composition of *po-type* relations must be the subset of one of  $po_{ww}, po_{wr}, po_{rr}, po_{rw}$  relation. For instance  $po_{ww}; po_{wr}; po_{rr}; po_{rw}$  is a subset of  $po_{ww}$ . Therefore, for any  $s$  that has consecutive composed *po-type* relations, we derive a  $s'$  which replaces them with a single *po-type* relation, such that  $s \subseteq s'$  and we assert that enforcing  $s'$  is sufficient to also enforce  $s$ . Below is an example of  $s$  and the derived  $s'$  using the insight that  $(po_{wr}; po_{rw}) \subseteq po_{ww}$ .

$$s \triangleq po_{wr}; po_{rw}; syn_{wr}; po_{rr}$$

$$s' \triangleq po_{ww}; syn_{wr}; po_{rr}$$

Note that an alternative approach would have been to omit deriving  $s'$  and instead simply asserting that the first condition is applied for each *po-type* relation in  $s$ . That would still be correct and can be used.

**Consecutive syn-type.** Our approach is identical for sync-pats with consecutive composed *syn-type* relations. Specifically, a composition of *syn-type* relations is always a subset of one of  $fr, ws, syn_{wr}, syn_{rw}$ . This is because any pair in a composition of *syn-type* relations is also in *syn* and thus it must be in one of  $fr, ws, syn_{wr}, syn_{rw}$ , as *syn* is the union of these four relations. Therefore, for any  $s$  that has consecutive composed *syn-type* relations, we derive an  $s'$  which replaces them with a single *syn-type* relation, such that  $s \subseteq s'$ .

**Begin with syn-type.** When a sync-pat  $s$  begins with a *syn-type* relation, then we derive  $s'$  by simply removing it. Let us see why through an example. Assume the following  $s$  which starts with an *fr* and the derived  $s'$  which removes

the *fr*:

$$s \triangleq fr; po_{wr}fr; po_{wr}$$

$$s' \triangleq po_{wr}fr; po_{wr}$$

Assume now that  $(a, c) \in s$ ,  $(b, c) \in s'$  and  $(a, b) \in fr$ . Let us now prove that if  $s'$  is enforced,  $s$  is enforced too. Assume  $s'$  is enforced but  $s$  is not. Therefore,  $(c, b) \notin syn$  but  $(c, a) \in syn$ . We know that  $(a, b) \in fr$  and thus  $(a, b) \in syn$ . By the transitivity property of *syn* we assert that since  $(c, a) \in syn \wedge (a, b) \in syn$  it must be that  $(c, b) \in syn$ . This contradicts our assumption that  $s'$  is enforced. Therefore enforcing  $s'$  is sufficient to also enforce  $s$ .

**End on syn-type.** Similarly, if  $s$  ends on a *syn-type* relation, we remove it to derive  $s'$ . Assume the following example.

$$s \triangleq po_{wr}fr; po_{wr}fr$$

$$s' \triangleq po_{wr}fr; po_{wr}$$

Assume now that  $(a, c) \in s$ ,  $(a, b) \in s'$  and  $(b, c) \in fr$ . Using the same proof as above, we can infer that if  $s'$  is enforced then  $(b, a) \notin syn$  and thus it follows that  $(c, a) \notin syn$ . Therefore enforcing  $s'$  is sufficient to also enforce  $s$ .

**A combination of the above (IRIW – Figure 1b).** When a sync-pat falls into more than one of the above irregular categories, we combine the techniques discussed above. For instance, let  $s_{iriw}$  be the IRIW sync-pat of Figure 1b (discussed in the Introduction).

$$s_{iriw} \triangleq rf; po_{rr}; fr; rf; po_{rr}$$

This is an irregular sync-pat, which both starts with a *syn-type* and includes a composition of consecutive *syn-type* relations. We use the rules above to derive  $s'_{iriw} \triangleq po_{rr}; syn_{rr}po_{rr}$  and assert that if  $s'_{iriw}$  is enforced then  $s_{iriw}$  is also enforced. To enforce  $s'_{iriw}$ , cond-1 requires the prt-ordering  $prr_{rr}$  and cond-2 requires the srt-ordering  $srt_{rr}$  which is also required by cond-3.

## 6 From rt-orderings to MCMs

So far we have created a mapping from MCMs to rt-orderings, by establishing the sufficient rt-orderings to enforce any sync-pat. In this section, we use this result to obtain the reverse mapping from rt-orderings to MCMs, focusing our discussion solely on regular sync-pats, seeing as the enforcement of irregular sync-pats is done by mapping them to regular ones. To obtain the mapping from rt-orderings to sync-pats, we reverse each of the three conditions, assuming that an operation can be of type  $m$  or  $n$  where  $m, n \in \{read, write\}$ . Specifically, given a protocol that enforces a set of prt-orderings  $R_p$  and a set of srt-orderings  $R_s$ , a sync-pat  $s$  is enforced when it abides by the following conditions:

- **Cond-1'** :  $s$  can include a *po-type* relation  $po_{mn}$  if  $prr_{mn} \in R_p$ .

- *Cond-2'*:  $s$  can include a *syn-type* relation  $syn_{mn}$  if it is an *rf* relation or  $srt_{nm} \in R_s$ .
- *Cond-3'*:  $s$  can start with an operation of type  $m$  and end on an operation of type  $n$ , if  $srt_{mn} \in R_s$ .

**Example.** To showcase how these conditions can be used in practice, let us specify the MCM  $CM_i$  that is enforced by a protocol that enforces the rt-orderings  $prt_{ww}$ ,  $prt_{wr}$ ,  $srt_{rw}$  and  $srt_{wr}$ . To do so we must specify a set of sync-pats  $S_{CM}$  where each regular sync-pat  $s \in S_{CM}$  abides by the following rules:

- Any *po-type* relation in  $s$  is either a  $po_{ww}$  or a  $po_{wr}$
- Any *syn-type* relation in  $s$  is either a  $syn_{wr}$  or a  $syn_{rw}$ . (*rf* is included in  $syn_{wr}$ )
- $s$  must either start on a write and end on a read, or start on a read and end on a write.

Notably, the interplay amongst the rules can be used to further simplify them. For instance the third rule asserts that from the available srt-orderings ( $srt_{rw}$  and  $srt_{wr}$ ) it follows that either the first operation must be a read and the last a write or the reverse. However, neither of the available *po-types* ( $po_{ww}$  and  $po_{wr}$ ) start with a read and since a regular sync-pat must start with a *po-type* relation, it cannot be that the first operation is a read. Consequently, the first operation can only be a write, and thus the last operation must be a read, to abide by the third rule.

Similarly, because a regular sync-pat, is a composition of alternating *po-type* and *syn-type* relations, it follows that the available *po-type* and *syn-type* relations can only be used if they can synergize. For instance, the  $syn_{wr}$  cannot be used at all because neither of the available *po-types* ( $po_{ww}$  and  $po_{wr}$ ) starts from a read. Similarly, because the  $syn_{wr}$  cannot be used, the  $po_{ww}$  cannot be used before any *syn-type*, nor can it be used as the last relation because the sync-pat must end on a read.

Therefore the rules for a sync-pat  $s \in S_{CM}$  are simplified as follows:

- Any *po-type* relation in  $s$  must be a  $po_{wr}$
- Any *syn-type* relation in  $s$  must be a  $syn_{rw}$ .
- $s$  must start on a write and end on a read.

As a result any regular sync-pat  $s \in S_{CM}$  is a composition of alternating  $po_{wr}$  and  $syn_{rw}$  relations. Below, we list a few examples sync-pats in  $S_{CM}$

$$\begin{aligned} s_1 &\triangleq po_{wr}; syn_{rw}; po_{wr} \\ s_2 &\triangleq po_{wr}; syn_{rw}; po_{wr}; syn_{rw}; po_{wr} \\ s_3 &\triangleq po_{wr}; syn_{rw}; po_{wr}; \dots syn_{rw}; po_{wr} \end{aligned}$$

Notably, enforcing the  $prt_{ww}$  and  $srt_{rw}$  are not contributing towards enforcing any  $s \in S_{CM}$ .

## 7 From rt-orderings to Protocols

In previous sections we established the mappings between the sync-pats and the rt-orderings. In this section, we relate rt-orderings to some well-known protocol design techniques. We start with a brief discussion on the prt-orderings and then we focus on the srt-orderings.

### 7.1 Enforcing rt-orderings

Prt-orderings model the operation of the ROB specifying when the memory system can begin executing an operation. Upholding the prt-orderings is as simple as inspecting the state of the ROB. For instance, enforcing  $prt_{wr}$  implies that the memory system cannot begin executing a read  $r$  from process  $p$ , until every preceding write in the ROB is completed.

Srt-orderings models how the memory system executes reads and writes. Below we discuss two common techniques that can be used to enforce srt-orderings 1) *overlap* and 2) *lockstep*.

**Overlap.** The srt-ordering  $srt_{mn}$  can be enforced simply by ensuring that operations of type  $m$  must overlap with operations of type  $n$  in a physical location. For instance, we can enforce  $srt_{wr}$ , by ensuring that a write is propagated to  $x$  nodes and a read queries  $y$  nodes, where  $x + y > N$  and  $N$  is the number of nodes. Alternatively, both types of operations can “meet” in some centralized physical location (e.g., the directory for multiprocessors). To ensure all four srt-orderings and thus linearizability, both reads and writes must query  $y$  nodes to learn about completed operations and must broadcast their results to  $x$  nodes. This is exactly how the multi-writer variant of ABD [14] operates.

**Lockstep.** Lockstep is a technique, where a memory system node first “grabs a lock” on the object before beginning an operation and releases it when the operation completes. Upon grabbing the lock, the node learns about the operation executed by the previous lock holder. The act of “grabbing the lock” is similar to getting a cache-line in  $M$  or  $S$  state in a coherence protocol [17], or becoming the leader of the next log entry in a state machine replication protocol, such as Paxos [11]. Notably, lockstep entails overlap as operations must meet in a physical location to exchange the lock, but it also precludes the operations from executing concurrently.

There are two aspects of lockstep that can enforce srt-orderings. First, the srt-ordering between two operations is enforced if a lock must be passed from one to the other. For example we can enforce  $srt_{ww}$  by mandating that a lock must be grabbed to perform a write. Second, locking also ensures that certain operations cannot overlap in real-time. When a write cannot overlap with a write and  $srt_{ww}$  is enforced then  $srt_{rw}$  is also enforced. This is because if a read  $r$  returns the value of write  $w$ , then a write  $k$  that begins after  $r$  has completed must also begin after  $w$  has completed. Similarly, when a write cannot overlap with a read and  $srt_{wr}$



is enforced, then  $srt_{rr}$  is enforced. This is because if a read  $r$  returns the value of write  $w$ , then it must be that  $w$  completes before  $r$  begins. Therefore, a read  $m$  that begins after  $r$  has completed, must also begin after  $w$  has completed and thus will observe  $w$ . Protocols often combine the two aspects of lock-step to enforce the single-writer multiple-reader invariant (SWMR) [17], where for any given object at any given time there is either a single write in progress or multiple reads. This ensures that writes must grab a lock from each other ( $srt_{ww}$ ), reads must grab a lock from writes ( $srt_{rw}$ ), writes cannot overlap in time ( $srt_{rw}$ ) and writes do not overlap in time with reads ( $srt_{rr}$ ).

## 8 Discussion

In this work, we have used the rt-orderings to model the protocol, allowing us to create a mapping from MCMs to protocols. In this section, we discuss other use-cases of the rt-orderings. We start by describing the relation of srt-orderings to linearizability and then we discuss how they can be used to describe the real-time guarantees of existing protocols and to decide the consistency guarantees provided when protocols are composed.

### 8.1 Relation of srt-orderings to linearizability

Formally, the lin criterion [8] can be written in our notation as follows. An execution  $E(M, po, rf, hb, RL)$  is linearizable if there exists  $rl \in RL$  such that  $hb \subseteq rl$ . Furthermore, because lin is a *local* property, it suffices that only operations to the same object abide by the rule. Plainly, for two operations  $a, b$  to the same object  $x$ , if  $(a, b) \in hb$  then it must be that  $(a, b) \in rl$ . This is equivalent to:  $acyclic(hb, syn)$ . This, in turn, is equivalent to enforcing all four srt-orderings. Therefore, the lin property is the property of enforcing all four srt-orderings.

**Intuition.** Informally, the lin property can be defined by writing that *each memory operation appears to take effect at some point between its invocation and completion* [7]. Notably, each of the srt-orderings is a specialization of this definition. For instance,  $srt_{wr}$  mandates that each write appears to take effect at some point between its invocation and completion w.r.t. every read. Combining all four srt-orderings mandates that each write or read appears to take effect at some point between its invocation and completion w.r.t. every read or write. This is equivalent to the lin definition.

### 8.2 Srt-orderings for compositionality

So far we have viewed the srt-orderings solely as a mathematical model of the protocol. However, in distributed systems, it is often necessary to describe the real-time guarantees offered by a system, in order to compose different systems. Consider the example of Zookeeper. Zookeeper does not offer linearizability, and thus multiple Zookeeper instances cannot be composed. However, researchers have found that

Zookeeper does offer some real-time guarantees that can be leveraged to achieve compositionality [12].

Srt-orderings can be used to capture these real-time guarantees. For example, the precise real-time guarantees of Zookeeper are the  $srt_{ww}$  and the  $srt_{rw}$  srt-orderings and all four prt-orderings. More importantly, given this knowledge we can specify the MCM provided by composing different Zookeeper instances: the composed system will enforce the same rt-orderings as a single Zookeeper instance and thus we can use the reverse mapping from rt-orderings to sync-pats to specify the resulting MCM. In fact, we can use srt-orderings in the same spirit, to specify the MCM of any combination of composed systems, by asserting that the composed system will enforce any rt-ordering that is enforced by all of its subsystems.

## 9 Related Work

This work is the first to provide a mapping from MCMs to the protocols that can enforce them. To present this mapping we have used an abstract system model and the formalism presented by Alglave et al. [2] in order to describe MCMs, executions and real-time guarantees. Several works [4–6, 13, 20] have also described similar system models and formalism, but differ from our work in that they do not provide a mapping from MCMs to the protocols. Specifically, Szekeres and Zhang [20] provide a system model and a formalism called *result visibility* to describe consistency guarantees, including real-time guarantees. Crooks et al. [5], focusing on databases, provide a state-based formalization of isolation guarantees. Burckhardt, in his book on Eventual Consistency [4], provides a formalism to describe consistency models and protocols, with a focus on weaker guarantees. Lev-ari et al. [13] define Ordered Sequential Consistency, OSC(A), in order to specify the real-time guarantees of a protocol (with a focus on Zookeeper [9]). Similarly, Gotsman and Burckhardt propose GSC [6], a generic operational model for systems that totally order all writes, which can capture all of the srt-orderings for such systems.

In the cache coherence literature, the four *program-orderings* are used to describe consistency guarantees [17]. In fact, researchers have shown that when the coherence protocol enforces the *single-writer multiple-reader* (SWMR) invariant, the MCM depends solely on the enforced program orderings [3, 16]. Program orderings are very similar to prt-orderings with the subtle difference that program orderings carry the implication that the memory system enforces SWMR. In contrast, prt-orderings make no such assumption, allowing us to explore all possible behaviours of the memory system.

Finally, CCICheck [15] provides a way to verify an existing coherence protocol against its target MCM using the notion of  $\mu hb$  orderings, which are related to real-time. Our work

provides a mapping from MCMs to protocols, enabling the design of minimal protocols for any MCM.

## 10 Conclusion

In this work, we took an important step towards actualizing our overarching vision, which is the automation of the design of protocols that are responsible to enforce the consistency model (MCM) in shared memory systems. Specifically, we argued for the need of an intermediate abstraction between the MCM and the protocol implementation, that will allow us to map the MCM to specific protocol implementation techniques. We proposed such an abstraction by mathematically modelling the protocol through eight rt-orderings. To do so, we observed that any such protocol is comprised by two widgets: the ROB and the memory system. We mathematically abstracted the ROB with the four prt-orderings and the memory system with the four srt-orderings. Crucially, we created a mapping from consistency guarantees to the rt-orderings, such that any MCM can be translated into the minimal set of rt-orderings that are required to enforce it. Finally, we completed the picture, by relating the rt-orderings to protocol implementation techniques, paving the way for automating protocol design.

## Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments and Dan Sorin for the insightful discussions and valuable feedback. This work was supported in part by EPSRC grant EP/L01503X/1 to The University of Edinburgh and ARM through its PhD Scholarship Program.

## References

- [1] 2014. Manhattan, our real-time, multi-tenant distributed database for Twitter scale. <https://bit.ly/2Vgu0wd>. (Accessed on 07/08/2019).
- [2] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats. *ACM TOPLAS* 36, 2 (jul 2014), 1–74. <https://doi.org/10.1145/2627752>
- [3] Arvind and Jan-Willem Maessen. 2006. Memory Model = Instruction Reordering + Store Atomicity. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA '06)*. IEEE Computer Society, USA, 29–40. <https://doi.org/10.1109/ISCA.2006.26>
- [4] Sebastian Burckhardt. 2014. Principles of Eventual Consistency. *Found. Trends Program. Lang.* 1, 1-2 (Oct. 2014), 1–150. <https://doi.org/10.1561/25000000011>
- [5] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC '17)*. Association for Computing Machinery, New York, NY, USA, 73–82. <https://doi.org/10.1145/3087801.3087802>
- [6] Alexey Gotsman and Sebastian Burckhardt. 2017. Consistency Models with Global Operation Sequencing and their Composition. In *Leibniz International Proceedings in Informatics (LIPIcs); 31st International Symposium on Distributed Computing (DISC 2017)* (leibniz international proceedings in informatics (lipics); 31st international symposium on distributed computing (disc 2017) ed.), Vol. 91. Dagstuhl Publishing, 0.959027778–0.969444444. <https://www.microsoft.com/en-us/research/publication/consistency-models-with-global-operation-sequencing-and-their-composition/>
- [7] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- [8] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. <https://doi.org/10.1145/78969.78972>
- [9] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10)*. USENIX Association, Berkeley, CA, USA, 11–11. <http://dl.acm.org/citation.cfm?id=1855840.1855851>
- [10] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. 1994. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference (WTEC'94)*. USENIX Association, Berkeley, CA, USA, 10–10. <http://dl.acm.org/citation.cfm?id=1267074.1267084>
- [11] Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)* 16, 2 (1998), 133–169.
- [12] Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. 2016. Modular Composition of Coordination Services. In *USENIX Annual Technical Conference*.
- [13] Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. 2017. Composing Ordered Sequential Consistency. *Inf. Process. Lett.* 123, C (July 2017), 47–50. <https://doi.org/10.1016/j.ipl.2017.03.004>
- [14] N. A. Lynch and A. A. Shvartsman. 1997. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*. 272–281. <https://doi.org/10.1109/FTCS.1997.614100>
- [15] Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2015. CCICheck: using  $\mu$ hb graphs to verify the coherence-consistency interface. In *MICRO'15*. 26–37. <https://doi.org/10.1145/2830772.2830782>
- [16] Albert Meixner and Daniel J. Sorin. 2005. Dynamic Verification of Sequential Consistency. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*. IEEE Computer Society, USA, 482–493. <https://doi.org/10.1109/ISCA.2005.25>
- [17] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2020. A Primer on Memory Consistency and Cache Coherence, Second Edition. *Synthesis Lectures on Computer Architecture* 15, 1 (2020), 1–294. <https://doi.org/10.2200/S00962ED2V01Y201910CAC049> arXiv:<https://doi.org/10.2200/S00962ED2V01Y201910CAC049>
- [18] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: X86-TSO. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs '09)*. Springer-Verlag, Berlin, Heidelberg, 391–407. [https://doi.org/10.1007/978-3-642-03359-9\\_27](https://doi.org/10.1007/978-3-642-03359-9_27)
- [19] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. 1997. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. Association for Computing Machinery, New York, NY, USA, 288–301. <https://doi.org/10.1145/268998.266711>
- [20] Adriana Szekeres and Irene Zhang. 2018. Making Consistency More Consistent: A Unified Model for Coherence, Consistency and Isolation. In *Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC '18)*. Association for Computing Machinery, New York, NY, USA, Article 7, 8 pages. <https://doi.org/10.1145/3194261.3194268>