Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Semantics of RDMA Remote Read-Modify-Write Operations

Author:
Max Stupple

Supervisors:
Dr. Azalea Raad
Dr. Guillaume Ambal

Second Marker:
Prof. Alastair F. Donaldson

June 13, 2025

Abstract

Remote direct memory access (RDMA) allows each computer in a network to directly access the memory of every other machine, without involving the operating system of either side. This improves data transfer speeds, but reduces synchronisation, making it harder to reason about program behaviours. Formal memory models allow us to precisely describe which program behaviours are allowed, and to verify that a particular program only observes behaviours that the programmer expects.

The RDMA$^{\text{TSO}}$ memory model describes the semantics of RDMA, but it lacks support for remote read-modify-write operations. Read-modify-write (RMW) operations are the foundation for all synchronisation in single-machine concurrent programming, and RDMA remote RMWs allow the same techniques to be lifted to the network level, facilitating high-speed, fine-grained concurrent programming.

In this report, we extend RDMA$^{\text{TSO}}$ with support for remote RMWs. We provide an intuitive account of remote RMW behaviours, which has been reviewed and confirmed by an expert from NVIDIA, a leading vendor of RDMA systems. We formalise these behaviours through two models, one operational and the other declarative, and prove that the two models are equivalent. This work expands the range of programs for which RDMA$^{\text{TSO}}$ can describe a formal semantics, providing a robust mechanism to understand and reason about RDMA programs involving remote RMWs.

Acknowledgements

I owe great thanks to my supervisors, Azalea Raad and Guillaume Ambal, who have kindly and patiently supported me throughout this project. This work has been not only deeply interesting in itself, but also personally enlightening, in that it has played a significant role in my decision to pursue further research. I am very grateful to have had the opportunity to work on this topic.

I would also like to thank my second marker, Alastair Donaldson, for his diligent review of my interim report, which supported the final introduction and background sections seen here. Thanks also to Haggai Eran, whose expertise in RDMA has been invaluable in deepening and clarifying my understanding of this technology, and to Dan Iorga, for helping me better understand the Alloy modelling language.

Finally, I want to thank all of my friends and family who have supported me throughout this project and my degree. I am particularly grateful to my parents, who instilled in me the curiosity and ambition that have driven me at Imperial, and to my partner Laura, whose continual encouragement and support has helped me immeasurably.

Contents

1 Introduction

Concurrency refers to the ability of a system to execute multiple tasks (threads) at once, either by executing them *simultaneously* (in parallel) on multiple processor cores, or by time-sharing a single core and frequently context switching between tasks. All modern CPUs feature a multi-core design, so to maximise performance a program must distribute its work across multiple threads, which can then run concurrently: if a program can fully utilise all cores on a four-core multi-processor, then it would run in roughly one quarter of the time compared to if it executed the same tasks sequentially on a single core. Since the processor, not the programmer, decides how to distribute threads across cores and when to interrupt a thread to allow another to execute, the programmer must assume that the instructions of different threads may be executed in an arbitrary order. We call the rules which govern the order in which program instructions may executed the *memory model*, or concurrent semantics, of that processor.

Remote direct memory access (RDMA) technologies enable a machine to directly read and write the memory of another machine within the network, bypassing the operating system of each and thereby reducing the number of CPU cycles consumed in transferring data [1]. This is particularly useful in massively parallel computer clusters (e.g. cloud computing, big data and scientific computation), where this enables high-throughput, low-latency networking [2]. While older implementations of RDMA, such as *InfiniBand* [3], required specialised hardware, newer implementations of RDMA, such as *RDMA over Converged Ethernet* [4], operate on top of ordinary networking infrastructure. This has brought costs in line with traditional networking [5], leading to widespread adoption of RDMA as of 2018 [6].

Remote read (get) and write (put) operations are dispatched to the network interface card (NIC), which executes transfers in parallel as the program continues on the CPU. Due to the independent operation of CPU and NIC for local and remote operations respectively, instructions may not be executed in the order the programmer intuitively expects, leading to unintended behaviours; this is comparable to the operation of shared memory in a multi-processor. However, there is the additional caveat that the concurrent behaviour is multi-tiered: a program consists of several threads, with those threads distributed across a number of machines (nodes) in the network (inter-node concurrency), and each node executing several threads (intra-node concurrency). To write *correct* programs for RDMA networks, we need to know the order in which local (CPU) and remote (NIC) operations are executed, how they can be reordered, and when the effects of those operations become visible, both to local and remote threads.

To this end, similarly to how the concurrent semantics of Intel-x86 [7] or ARM [8] have been formalised, there has been some progress in formally describing the semantics of RDMA. Namely, two models exist in the literature: coreRMA [9] and RDMA$^{\text{TSO}}$ [10]. The coreRMA model is subject to several significant limitations, most notably that it models intra-node concurrency as following sequential consistency: an unrealistic and simplified memory model which no modern processor adheres to by default. RDMA$^{\text{TSO}}$ is instead based on the memory model of x86 processors, a reasonable assumption as Intel and AMD processors that use this architecture are widespread. However, RDMA$^{\text{TSO}}$ does not model the behaviour of every remote operation. While it does describe get and put operations, as well as remote *fences* useful for synchronisation, it omits remote *read-modify-write* (RMW, also known as atomic) operations.

Read-modify-write operations are an essential tool for concurrent programming [11],

and RDMA specifies remote RMWs to allow similar techniques to be applied at the network level. These operations read, modify, and write a remote memory location, while guaranteeing that no other remote RMW acts on that memory location between the read and write. This enables the implementation of shared higher-level synchronisation tools, such as locks and barriers. We propose extensions to the formal models described by RDMA^{TSO}, to also specify the behaviour of such remote atomic operations.

Contributions and Outline

In §2, we review the existing literature on memory models and RDMA. In §3, we overview the RDMA^{TSO} memory model and present our intuitive account of the weak behaviours of remote RMWs, which we illustrate through a number of examples. In §4, we develop the operational semantics of RDMA^{TSO} with new rules for remote RMWs, and in §5, we extend the declarative model. In §6, we discuss how the two models are proven equivalent. Finally in §7, we conclude with a discussion of the limitations of this project and future work on this topic.

2 Background

2.1 Concurrency

Since the end of the "free ride" for processor speed enjoyed during the era of Moore's law, the microprocessor landscape has become dominated by multi-core processors (MCPs) [12], which provide faster (effective) speeds than uniprocessors due to the distribution of work across multiple cores, which execute in parallel. In order to maximise the utilisation of these MCPs, and to minimise program execution time, programs must be written concurrently so that their work can be efficiently distributed across processor cores.

However, concurrent programming is widely considered to be challenging [13]. This is largely due to the non-deterministic nature of concurrent program execution: instructions from each program thread may interleave arbitrarily, which can lead to "Heisenbugs" [14] – errors which are only present in a subset of the possible interleavings, and hence appear seemingly at random while evading reliable reproduction. A typical cause of these types of bugs is a *data race*, where two program threads access the same memory location without synchronisation, with at least one of the accesses being a write. Depending on which thread happens to execute first, different program behaviours may be observed.

For example, consider the litmus test in Fig. 2.1a, where two threads attempt to increment the counter x concurrently: only if the two threads execute sequentially – that is, one thread executes to completion before the other commences – will the expected result of $x = 2$ be achieved; any other interleaving will cause one thread to overwrite the other, resulting in $x = 1$. This is equivalent to each thread executing $x := x + 1$, since that instruction would also be separated into a read and a write step; we have explicitly used two operations in our example for clarity.

2.2 Synchronisation

So, how can we avoid data races in our programs? We need a way to access data that guarantees *mutual exclusion*: the guarantee that so long as one thread is performing an operation on a shared memory location, no other thread may access it until the operation is complete. Modern processors provide *Read-Modify-Write* (RMW) instructions that facilitate this. An example of an RMW is *Compare-and-Swap* (CAS), which accepts three parameters and returns a value: $a := \texttt{CAS}(x, e, u)$. Given a memory location x containing a value v, and values e (expected) and u (update), it checks if $v = e$, and if so assigns u to location x; otherwise, it does not alter memory. It returns v – so in our example $a = v$ after the command returns – which allows the caller to check whether or not the swap was performed. Crucially, the hardware guarantees that this happens *atomically*, without any other thread accessing location x during the operation. In fact, CAS is sufficiently powerful to implement all other RMWs [11], and can be used to implement higher-level synchronisation tools such as locks and barriers [15, Chs. 7.2 and 17.2].

Another example of an RMW is *Fetch-and-Add* (FAA): $a := \texttt{FAA}(x, i)$, which given a memory location x containing value v, atomically assigns $v + i$ to location x, returning v. Unlike CAS, we might not know what value is stored at x before writing to it, but for use cases such as counters it allows a simpler implementation than CAS. However, if we needed to enforce a strict guarantee that the counter value stays within some bound, for example, FAA would not be suitable since the update occurs unconditionally.

$x = 0$	
$a := x$	$b := x$
$x := a + 1$	$x := b + 1$

(a) $x = 1, x = 2$ ✓

$x = 0$	
$a := \texttt{FAA}(x, 1)$	$b := \texttt{FAA}(x, 1)$

(b) $x = 1$ ✗ $x = 2$ ✓

Figure 2.1: Left: unsafe incrementation of the counter results in a data race — whichever thread reads first must write before the other reads or writes, or else its effect will be lost. Right: safe implementation using FAA guarantees the effects of both threads will be visible.

Notation. Columns (separated by |) denote distinct threads, variables x, y denote shared memory locations, a, b are thread-local variables (assumed to be initially 0), and the first line indicates the initial values of memory locations. Captions indicate allowed outcomes for variables with ✓, and disallowed outcomes with ✗.

Fig. 2.1b shows a safe implementation for concurrently implementing a counter using FAA. Since FAA guarantees that no other operation may occur on x between the read and write, we can be sure that both incrementations will be visible, resulting in a final state of $x = 2$.

2.3 Memory Models

Given the ability to synchronise between threads and ensure mutual exclusion where necessary, it seems that it would not be too challenging for programmers to reason about potential interleavings of instructions from each program thread, then use the tools available to write correct programs. However, the real world is significantly more complicated. The interleaving model, called *Sequential Consistency* (SC), is not realistic: no modern computer processor provides the guarantee that instructions within a thread are executed in order [16]. Instead, they only provide *weak* memory models.

To be precise, under sequential consistency, it is guaranteed that instructions within a thread execute in order, while instructions from different threads may interleave in an arbitrary execution order. A weak memory model, then, is one which, in addition to permitting any execution ordering allowed by SC, also permits some other orders – which is to say that the guarantee that instructions within a thread execute in order is *relaxed* in some way. It may also be, as we will see in the case of RDMA, that we cannot simply define one order over the instructions, as the order in which they are issued and that in which they become observable may be different.

There are numerous weak memory models, however we will focus on *Total Store Ordering* (TSO). TSO is a description of program execution on multiprocessors with store buffers, originally formulated to describe SPARC V8/9 processors [17]. It permits certain reorderings of instructions that are not allowed in SC, namely:

Write-read reordering (on different locations): a later read may be reordered *before* an earlier write, provided it is on a different memory location. This leads to weak behaviour known as *store-buffering*.

Write-to-read forwarding (on the same location): the value of an earlier write can be *copied* into a later read.

$x = y = 0$		$x = y = 0$		$x = y = 0$		$x = y = 0$	
$x := 1$ $a := y$	$y := 1$ $b := x$	$x := 1$ `mfence` $a := y$	$y := 1$ `mfence` $b := x$	$a := y$ $x := 1$	$b := x$ $y := 1$	$x := 1$ $y := 1$	$a := y$ $b := x$
(a) $a = b = 0$ ✓		(b) $a = b = 0$ ✗		(c) $a = b = 1$ ✗		(d) $(a, b) = (1, 0)$ ✗	

Figure 2.2: TSO litmus tests. Reproduced from [10] with permission.

Write-assignment reordering: a later assignment (which does not access memory) may be reordered *before* an earlier write.

Store-buffering is so-called because it can be modelled by assuming a buffer between a thread's private registers and shared memory, which behaves as a FIFO queue for write (store) instructions. Write instructions do not immediately affect memory, but rather enter the store buffer and execute at some later point, once they have reached the head of the queue. This facilitates write-read reordering, as a later read may not see an earlier write because it is still in the buffer; similarly, an assignment may not observe a buffered write. Write-to-read forwarding can be interpreted as a read viewing the latest value from the thread's store buffer, rather than reading from memory.

The TSO model has also been adapted to describe x86 processors, by Owens et al. [7]. In this paper, the authors define the concurrent semantics of the x86 architecture, which they call x86-TSO due to its similarity to the SPARC memory model. Notably, x86-TSO supports a *memory fence* instruction, which flushes a thread's store buffer. Henceforth, we use TSO to refer to the x86 memory model, rather than the original SPARC definition.

Inevitably, it is more challenging for programmers to reason about weak memory models than SC; indeed, TSO is a relatively simple model, yet there may be far more possible execution orders to consider due to the reordering within threads. In order to make these weak models easier to understand, it is useful to provide a *formal semantics* that programmers may use to build their intuition, or which may be used to write tools to automatically check programs for bugs.

Operational semantics are well established in the literature for defining the step-by-step execution of a program by providing a set of *transition rules*. These rules specify how the program's state can be updated at any given execution step, depending on whether particular logical properties hold at that point. For any valid program execution, one may write down a logical trace of the execution using the transition rules, called a derivation tree. Hence, an operational semantics gives a low-level description of program execution, which may be used for formal reasoning, and can help programmers understand program behaviour.

Declarative semantics, on the other hand, describe program executions as a graph: we use the program instructions (events) as nodes in the graph, and define several relations over the events, representing dependencies. For example, we might define the *reads-from* (rf) relation: if a read r reads the value written by an earlier write w, then we add an rf dependency from w to r. We might also impose certain restrictions on the relations in our graph, for example that our graph should not contain any cycles. Such execution graphs are particularly useful for understanding the properties and correctness of a program, and are much more concise than operational semantics; however they may be less useful for gaining an intuitive understanding of how programs execute.

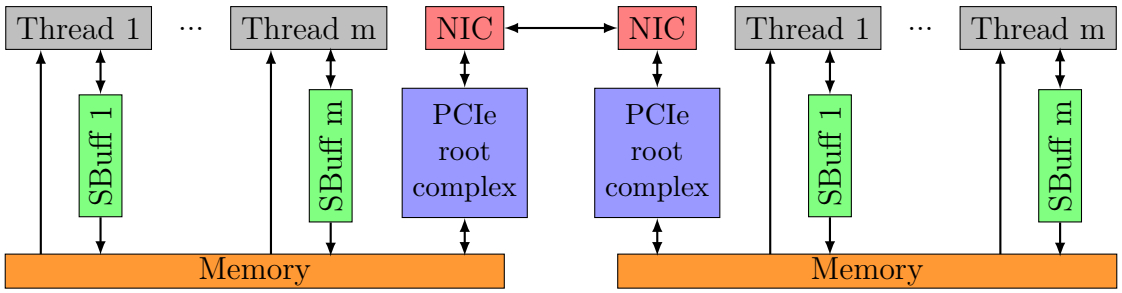

Figure 2.3: RDMA network overview [10, page 10]. Communication between NICs occurs in parallel to ongoing execution of local threads.

2.4 RDMA

Conventionally, communication over the network between separate machines requires the involvement of the operating system, which is responsible for the network and transport layers of the network stack. In common usage, the ~1ms latency introduced due to encoding/decoding the TCP/IP stack is a negligible factor in the overall latency, however in modern computer clusters the propagation delay is far lower. Thus, it is desirable to use a thinner network stack which bypasses the OS, so that remote reads and writes use fewer CPU cycles. This has led to widespread adoption of *Remote Direct Memory Access* (RDMA), which allows computers in the network to directly transfer data without the need for caches, switches, or any involvement by the CPU – the operation may be handled solely by the Network Interface Card (NIC), which transfers messages directly between main memory and the wire. Transfers are executed in parallel with system operations, reducing latency and increasing throughput [6].

In particular, as these network operations are executed in parallel to other operations on the CPU, they do not observe sequential consistency; hence it is necessary to define a memory model for RDMA so that programmers may understand the weak behaviours that can be observed, and use that knowledge to write correct programs and libraries for RDMA networks.

The first attempt to provide a formal semantics of RDMA programs was coreRMA [9]. However, this work is subject to a number of significant shortcomings, as described by Ambal et al. [10]. They identify four limitations of the coreRMA model: (1) the coreRMA authors assume that the concurrency on each node follows *sequential consistency* – this assumption is unrealistic as all modern CPU architectures (in particular, x86 and ARM, which are ubiquitous) offer a weaker model by default; (2) they provide only a *declarative* model, while operational models are required for certain applications, such as invariant synthesis [18], and may provide better intuition than declarative models, as they are explicitly intended to be easy to understand [19]; (3) the model is not validated against existing implementations, and they could not observe any of the weak behaviours permitted by coreRMA; (4) coreRMA is neither stronger nor weaker than the RDMA specification, admitting certain behaviours not allowed by the specification, while prohibiting behaviours which the specification permits.

Due to these limitations, Ambal et al. define RDMA$^{\text{TSO}}$, a memory model for RDMA which improves on coreRMA in a number of ways: it assumes an x86 architecture, hence modelling individual nodes as observing TSO; they provide both operational and declarative models; and they have validated that all weak behaviours permitted by the model

$x = 0$
$a := \texttt{CAS}(x, 0, 7) \mid b := \texttt{FAA}(x, 2)$

(a) $x = 2, x = 9$ ✓ $x = 7$ ✗

		$x = 0$
$a := \texttt{nCAS}(x^3, 0, 7)$	$b := \texttt{nFAA}(x^3, 2)$	

(b) $x = 2, x = 9$ ✓ $x = 7$ ✗

	$x = 0$
$a := \texttt{nCAS}(x^2, 0, 7)$	$b := \texttt{FAA}(x, 2)$

(c) $x = 2, x = 7, x = 9$ ✓

Figure 2.4: Remote RMWs are non-atomic with respect to local operations: this results in behaviours not visible when only using local RMWs. Remote RMWs are still atomic with respect to each other. We note remote CAS (resp. FAA) by `nCAS` (resp. `nFAA`), where `n` stands for NIC. Note that for RDMA litmus tests we use a double vertical line to indicate threads on separate nodes, and x^n to denote the variable x on foreign node n.

are either observed on actual hardware, or permitted by the specification but not implemented in current systems. Thus, due to the multiple models being proven equivalent, extensive hardware testing, and confirmation from hardware experts, RDMA$^{\text{TSO}}$ may be observed to be a very accurate model of concurrency possible on RDMA systems running on the x86 architecture.

2.5 Remote RMWs

Despite the advantages over coreRMA, the RDMA$^{\text{TSO}}$ model is arguably incomplete, as it does not describe all operations provided by RDMA. It provides the semantics of all local operations (as in TSO), and the semantics of remote read (get) and remote write (put) operations, as well as the semantics of remote fences instructions used for synchronisation. It does not provide, however, the semantics of remote read-modify-write (rRMW) instructions defined in the RDMA specification. Remote RMWs have the potential to be a very useful tool for synchronisation in distributed algorithms over RDMA networks, however they are subject to a peculiar set of properties.

Under TSO, (local) RMWs are not subject to any reordering: this makes them easy to reason about, since we know precisely what value the memory location in question held immediately before and immediately after the execution of the RMW, and, since the RMW executes as an atomic unit, we know that there was no intermediary value held or operation executed on that location. RDMA remote RMWs are much more challenging to understand and use by comparison.

Firstly, rRMWs have complex reordering rules with respect to other operations in the same thread, as we will detail later. Secondly, and more importantly, rRMWs only guarantee atomicity with respect to *other rRMWs* operations, *not* local operations on the target node, nor other types of remote operations towards that node. Although an option is available on some hardware to execute rRMWs with a higher level of atomicity, this is not globally supported, nor enabled by default. Thus, RDMA programmers will need to operate under the default assumption that other operations (in particular, local or remote writes) may occur between the read and write of a remote RMW.

Consider the examples in Fig. 2.4. Fig. 2.4a shows how the strong atomicity guarantees

of local RMWs require the effects of both operations to be visible: either the CAS succeeds first, setting x to 7, after which the FAA adds two to it to achieve $x = 9$; or the FAA occurs first, setting $x = 2$, after which the CAS fails. No other outcome is possible, as the execution of the two RMWs may not overlap – we certainly cannot observe $x = 7$, as this would mean we have somehow lost or overwritten the FAA. When using multiple remote RMWs as in Fig. 2.4b, we observe similar behaviour due to the guarantee that an rRMW is atomic with respect to other rRMWs operations. However in Fig. 2.4c, we see that such an outcome is possible when using a mixture of local and remote RMWs: the $x = 7$ result is achieved when the (local) FAA is executed between the read and write of the remote CAS. Such an execution is permitted due to the weak atomicity guarantees of remote RMWs.

For these reasons, a formal definition of the semantics of RDMA remote RMWs will be particularly helpful to RDMA programmers, and to that end we will present suitable extensions to RDMA^{TSO} such that programmers may confidently reason about all aspects of RDMA programs.

3 Overview

3.1 RDMA Concurrency

An interesting consequence of RDMA operations being handled by the NIC separately to the CPU is that it creates two distinct tiers of concurrency. In general, the CPU does not wait for NIC operations to complete – this would greatly slow down program execution, as despite network communication being relatively fast (compared to TCP/IP), it is still much slower than local operations on the CPU. Therefore, we can consider CPU (intra-node) concurrency (governed by TSO) separately to network-level (inter-node) concurrency. The reordering rules between CPU and NIC operations are very simple: an earlier NIC operations can *always* be reordered *after* a later CPU operation.

The CPU reordering rules have already been discussed in §2.3. Reordering rules for RDMA operations on a single thread (except rRMWs) are described in [10] and shown in Fig. 3.1. To summarise:

- CPU operations always complete before later RDMA operations (Fig. 3.1a)
- RDMA operations may be reordered after later CPU operations, as if executing concurrently in a separate thread (Fig. 3.1b)
- Poll awaits the earliest (unpolled) RDMA operation towards a given node (Figs. 3.1c and 3.1d)
- RDMA operations towards distinct nodes may be reordered (Figs. 3.1e and 3.1f)
- An earlier put always completes before a later get towards the same node (Fig. 3.1g)
- An earlier get may be reordered after a later put towards the same node (Fig. 3.1h)
- RDMA operations always complete before a later remote fence towards the same node

We are more interested in RDMA programs running on multiple nodes. Fig. 3.2 shows a number of litmus tests for inter-node concurrency. Store buffering, as seen in TSO, is similarly possible under RDMA (Fig. 3.2a). Additional weak behaviours are also possible, including load buffering (Fig. 3.2b); load buffering is analogous to store buffering, but concerning unobserved reads rather than writes (in our case, the read is a get operation).

Furthermore, whilst polling can prevent most weak behaviours, including load buffering (Fig. 3.2c), store buffering cannot be prevented due to a difference in how a poll awaits the completion of a put as opposed to a get. In the case of a get, the poll only completes once the value read on the remote node has been written to local memory. A put, however, may be considered complete while the write of the value to remote memory is still pending: this is because the confirmation of a put is returned when the operation is buffered, not when it is complete. In practice, the write is almost always de-buffered before the confirmation has travelled across the network, but the behaviour of Fig. 3.2d is allowed by the architecture specification.

However, it is still possible to prevent store buffering. To our knowledge, RDMA is implemented on hardware which meets the PCIe standard [20]. In certain instances, PCIe provides stronger guarantees than the RDMA specification, which we will always assume

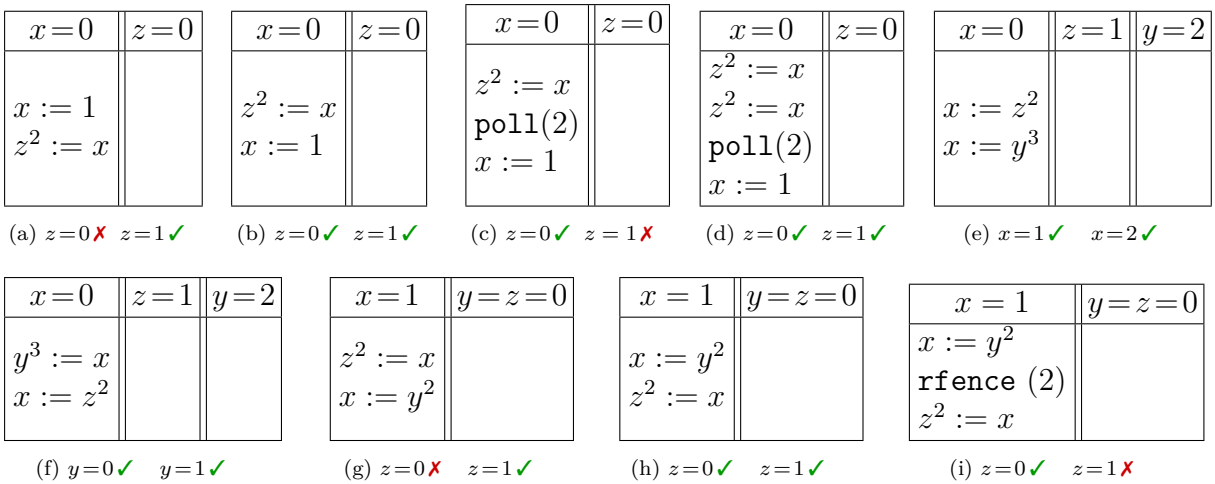

Figure 3.1: RDMA litmus tests showing reordering rules for a single node. Column (separated by ||) denote distinct nodes, and the top line of each column gives the initial values for shared memory locations on that node. Reproduced from [10] with permission.

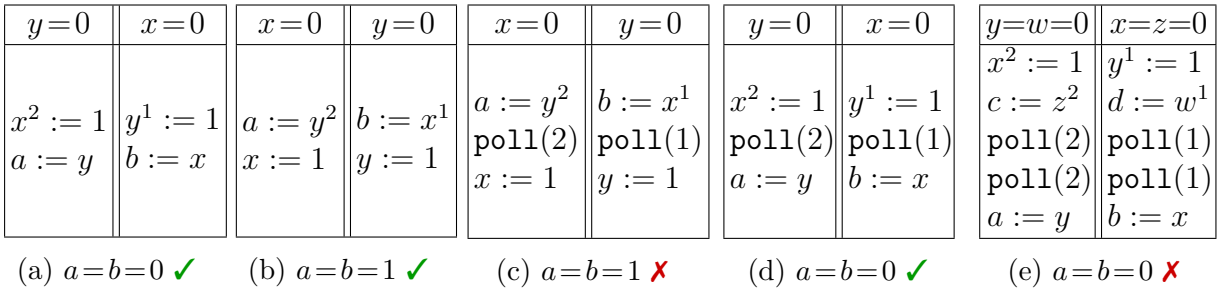

Figure 3.2: RDMA litmus tests for concurrency across multiple nodes. Reproduced from [10] with permission.

are available due to its ubiquity. The standard guarantees that a read cannot complete before a write which was issued earlier. Therefore, a NIC remote read (get) must flush all pending NIC remote writes (puts) to memory[1], so by polling a get which follows a put, we guarantee that the NIC remote write of the put has been committed to memory (Fig. 3.2e).

3.2 Remote RMW Behaviours

We now present an intuitive account of rRMW behaviours.

Atomicity. Remote RMWs are atomic only with respect to each other – not with respect to any other memory operations. This is in contrast to standard RMWs, which are atomic with respect to all other operations. We will need to consider that either CPU operations on the remote side, or other remote operations (puts and gets) from any node can result in non-atomic behaviour for the rRMW.

Reordering. Due to the two-tiered concurrency of RDMA, all remote operations may be reordered after a later local operation – this equally applies to rRMWs. From [10],

[1]The converse is also true: the NIC local read of a put flushes pending NIC local writes due to gets. However this is not particularly useful, since poll is well behaved with respect to get.

we know that an earlier get can be reordered after a later put. We can say that get/put reordering is allowed. Other reorderings involving any combination of get and put are not allowed: get/get[1], put/get and put/put. Remote RMWs behave much like puts, so that get/rRMW is allowed, and all other reorderings are disallowed.

The litmus tests in Fig. 3.3 illustrate these behaviours. Figs. 3.3a to 3.3d show that if either a put or get reads or writes the location targeted by a remote CAS, non-atomic behaviour may be observed. In contrast, Fig. 3.3e shows that multiple rRMWs are atomic with respect to each other. Figs. 3.3f to 3.3i show that rRMWs never reorder with respect to puts or other rRMWs, and an earlier rRMW completes before a later get. Fig. 3.3j shows that an earlier get may be reordered after a later rRMW.

Discussion: Remote RMW Behaviours

Early in the project, it was believed that rRMWs had stronger atomicity guarantees, and weaker reordering behaviours.

Atomicity. We initially believed that rRMWs were atomic with respect to all other remote operations (i.e. puts and gets, in addition to other rRMWs). This confusion came about due to overloaded use of the term "atomic access" in the InfiniBand specification [21]. Section 9.4.5.1 reads:

> Atomicity of the read/modify/write on the responder's node by the ATOMIC Operation shall be assured in the presence of concurrent atomic accesses [by the NIC].

The correct interpretation of this quote is that rRMWs guarantee atomicity only with respect to rRMWs issued by other threads towards the same node – that is to say, we take "atomic access" to mean an RDMA remote read-modify-write. However, the term may also be used in the context of a simple read or write being atomic, meaning that while a memory location is being written to, it cannot be read – for example, see (non-)atomic accesses in C/C++ [22]. Without this guarantee, it is possible to read a memory location as containing a value which was never written to it, even after the memory location has been initialised. This "atomicity" is a far more ubiquitous guarantee, and puts and gets would both be considered "atomic accesses" in this sense. Fortunately, some other parts of the document help to clarify this: section 10.6.7.2 mentions that

> AtomicOps are not required by the architecture to be atomic with respect to RDMA writes

which reinforces that they are only atomic with respect to other rRMWs[2].

Reordering. We also believed that an earlier rRMW could be reordered *after* a later rRMW. This was due to Table 76, which shows which pairs of operations require a fence to guarantee in-order semantics. It shows that when the first and second operations are both RDMA Atomics, a fence is required. This seems to imply that without a fence, a pair of rRMWs may execute out of order. In fact, such behaviour is not possible – section 9.4.5.2 reads:

[1] Technically, a pair of gets may be partially reordered. They may read remote memory out of order, but must write to local memory in order.

[2] The term 'AtomicOp' in the quote refers to an RDMA remote RMW.

$x = 0$		$y = 1$
$x := 2$ $y^3 := x$	$a := \texttt{nCAS}(x^1, 0, 3)$	

(a) $(x, y) = (3, 2)$ ✓

		$x = 0$
$x^3 := 1$	$a := \texttt{nCAS}(x^3, 0, 2)$	

(b) $x = 2$ ✓

		$x = 0$
$a := x^3$	$b := \texttt{nCAS}(x^3, 0, 2)$	$x := 1$

(c) $(a, x) = (1, 2)$ ✓

$x = 0$		$y = 1$
$x := y^3$	$a := \texttt{nCAS}(x^1, 0, 2)$	

(d) $x = 2$ ✓

		$x = 0$
$a := \texttt{nFAA}(x^3, 1)$	$b := \texttt{nCAS}(x^3, 0, 2)$	

(e) $x = 1$ ✓ $x = 3$ ✓ $x = 2$ ✗

	$x = 0$
$a := \texttt{nCAS}(x^2, 1, 2)$ $b := \texttt{nCAS}(x^2, 0, 1)$	

(f) $x = 2$ ✗

	$x = 0$
$a := \texttt{nCAS}(x^2, 1, 2)$ $x^2 := 1$	

(g) $x = 2$ ✗

	$x = 0$
$x^2 := 2$ $a := \texttt{nCAS}(x^2, 0, 1)$	$b := x$

(h) $b = 1$ ✗

	$x = 0$
$a := \texttt{nCAS}(x^2, 0, 1)$ $b := x^2$	

(i) $b = 0$ ✗

	$x = 0$
$a := x^2$ $b := \texttt{nCAS}(x^2, 0, 1)$	

(j) $a = 1$ ✓

Figure 3.3: Litmus tests illustrating atomicity and reordering of rRMWs with respect to other remote operations.

> the ATOMIC Operation only accesses memory after prior (non-RDMA READ) requests access memory and before subsequent requests access memory

which clearly enforces that rRMWs are in-order with respect to each other.
It may be that the apparently erroneous table entry is due to completion of an rRMW being defined not with respect to the rRMW accessing memory, but rather the response packet it returns. This is described in section 9.4.5.2:

> For the requestor [the thread issuing the rRMW], an ATOMIC Operation is considered complete when the response packet returns.

So if we wish for a later rRMW to wait for the response of an earlier rRMW, we indeed require a fence, however the order of memory accesses is maintained regardless.

4 Operational Semantics

At its core, an operational semantics consists of a logical model of the state of a machine or system, and a set of rules describing how that system may transition from one state to the next. The transition rules can be composed into *derivations*, which describe the execution of a program. If we can build a derivation tree for a given execution, then we know the execution is possible for our system, because it is composed from allowed transitions.

In our case, the system is a network of machines (nodes) communicating using RDMA, and the states we are interested in are (a) the state of the program, i.e. which commands have been executed, and which remain to be executed, for each program thread; (b) the state of memory on each node; (c) the state of the TSO store buffer of each thread; and (d) the state of any network communications. Under RDMA, each thread has access to a *queue pair* for communicating with each other node in the network, so we are interested in the state of the queue pairs of each thread.

4.1 States of the Operational Semantics

Throughout, we follow and extend the notation of [10].

Nodes and Threads. We write $\mathsf{Node} = \{1..N\}$ for the set of node identifiers, and $\mathsf{Tid} = \{1..M\}$ for the set of thread identifiers. We write n (resp. t) to range over nodes (resp. threads), and given some node n we write $\overline{n}$ to range over the set of all other nodes $\mathsf{Node} \setminus \{n\}$. Each thread runs on a particular node, so we write $n(t)$ for the node the thread belongs to.

4.1.1 Memory

Although all nodes can directly access all memory locations, whether an operation is towards local or remote memory is pivotal to our semantics, so we are always careful to note the node to which a memory location belongs. We write Loc_n for the set of locations local to node n, and $\mathsf{Loc} = \biguplus_n \mathsf{Loc}_n$ for the set of all locations. We use $\mathsf{Loc}_{\overline{n}} = \mathsf{Loc} \setminus \mathsf{Loc}_n$ and write x^n, y^n, z^n for values in Loc_n, respectively $x^{\overline{n}}, y^{\overline{n}}, z^{\overline{n}}$ for $\mathsf{Loc}_{\overline{n}}$. When the node in question is sufficiently clear, we elide the superscript and instead simply write x or $\overline{x}$ for local or remote locations respectively.

4.1.2 Program State

Values and Expressions. The language of expressions is standard and elided. We write $v \in \mathsf{Val}$ for values, with $\mathbb{N} \subseteq \mathsf{Val}$, and $e \in \mathsf{Exp}$ for expressions. We write $\mathsf{elocs}(e)$ for the set of memory locations referenced in e, $e[v/x]$ for the expression obtained by substituting all references to location x in e with value v, and $[\![e]\!]$ for the evaluation of e given it is *closed*, that is, $\mathsf{elocs}(e) = \emptyset$. We use e^n for expressions where $\mathsf{elocs}(e^n) \subseteq \mathsf{Loc}_n$

Commands and Programs. Commands are described by the C^n grammar below. CPU operations (CComm) are assignment, assumption of the value of a location, memory fence, compare-and-set, and poll, which awaits the earliest completion notification of a remote operations towards $\overline{n}$.

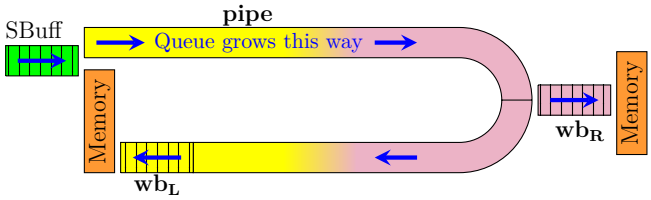

Figure 4.1: Simple queue-pair structure.

RDMA operations (RComm) are either a 'get' of the form $x := \overline{y}$ which reads a remote location $\overline{y}$ and writes its value to local location x, a 'put' ($\overline{y} := x$) which does the reverse, 'remote-CAS'[1] (resp. 'remote-FAA') which executes a *remote atomic* compare-and-set (resp. fetch-and-add), and 'remote fence' which ensures all prior RDMA operations towards $\overline{n}$ complete before any later RDMA operations towards $\overline{n}$ execute.

Primitive operations (PComm) are CPU or RDMA operations, and commands (Comm) are the no-op, primitive operations, sequential composition (executes the first command, then the second), non-deterministic choice (executes one command or the other), and non-deterministic loop (executes the command some finite, possibly zero number of times).

A program P consists of a map from threads to commands, such that each $t \in \mathsf{Tid}$ is mapped to a command on $n(t)$.

$$\mathsf{Comm} \ni \mathsf{C}^n ::= \texttt{skip} \mid \mathsf{c}^n \mid \mathsf{C}_1^n; \mathsf{C}_2^n \mid \mathsf{C}_1^n + \mathsf{C}_2^n \mid \mathsf{C}^{n*} \qquad \mathsf{PComm} \ni \mathsf{c}^n ::= \mathsf{cc}^n \mid \mathsf{rc}^n$$
$$\mathsf{CComm} \ni \mathsf{cc}^n ::= x := e^n \mid \texttt{assume}(x = v) \mid \texttt{assume}(x \neq v) \mid \texttt{mfence} \mid x := \texttt{CAS}(y, e_1, e_2) \mid \texttt{poll}(\overline{n})$$
$$\mathsf{RComm} \ni \mathsf{rc}^n ::= x := \overline{y} \mid \overline{y} := x \mid x := \texttt{nCAS}(\overline{y}, e_1, e_2) \mid x := \texttt{nFAA}(\overline{y}, e) \mid \texttt{rfence}\ (\overline{n})$$

4.1.3 Store Buffers

To implement the weak behaviour of TSO described in §2.3, we assign each thread a *store buffer* $\mathsf{B}(t)$, which is a FIFO queue containing pending writes to memory by that thread. When a thread performs a CPU read, it reads the most recent entry for that location in its store buffer, if there is one, instead of the value in memory. The write at the head of the queue may be flushed to memory at any time, and `mfence` and `CAS` wait until the store buffer is empty before executing.

4.1.4 RDMA Operations

Queue Pairs. We follow the *simplified* operational model described in [10], and therefore consider a queue-pair structure comprising three FIFO queues: **pipe**, which contains pending or in-progress RDMA operations; $\mathbf{wb_R}$, the remote write buffer, which contains pending writes to the memory of the remote node; and $\mathbf{wb_L}$, the local write buffer, which contains pending writes to the memory of the local node. The structure is shown in Fig. 4.1. Notice that under this simplified model, the transition between local and remote node in **pipe** is continuous – we do not explicitly model the transition between local (yellow) and remote (pink) sides.

[1] Note, we use the prefix `n` for 'NIC', rather than `r` for 'remote', for consistency with the notation used later in §5.

$$\frac{\mathsf{P} \xrightarrow{t:\varepsilon} \mathsf{P}'}{\mathsf{P},\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \Rightarrow \mathsf{P}',\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}} \qquad \frac{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{t:\varepsilon} \mathsf{M}',\mathsf{B}',\mathsf{A}',\mathsf{QP}'}{\mathsf{P},\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \Rightarrow \mathsf{P},\mathsf{M}',\mathsf{B}',\mathsf{A}',\mathsf{QP}'}$$

$$\frac{\mathsf{P} \xrightarrow{t:l} \mathsf{P}' \quad \mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{t:l} \mathsf{M}',\mathsf{B}',\mathsf{A}',\mathsf{QP}'}{\mathsf{P},\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \Rightarrow \mathsf{P}',\mathsf{M}',\mathsf{B}',\mathsf{A}',\mathsf{QP}'}$$

Figure 4.2: RDMA$^{\text{TSO}}$ operational semantics with the program and hardware transitions given in Fig. 4.3 and Fig. 4.4

Remote Atomics. To model the behaviour of RDMA atomic operations, we assign each node a *remote atomic lock* $\mathsf{A}(n)$, which is a boolean indicating whether an RDMA atomic is currently in progress *towards* that node.

> **Discussion: States of the Operational Semantics**
>
> Extending the language of commands to accommodate the new remote atomic operations is straightforward – we just need to include the new RDMA commands. We also need to introduce a remote atomic lock for each node, which will be used in the relevant transition rules to guarantee atomicity. Without this structure, it would be necessary to check all queue-pairs towards a particular $\overline{n}$ for in-progress rRMWs, which would be semantically equivalent but much more verbose. See §4.3.2 for more discussion of the structure A as it relates to the transition rules specifically.

4.2 Transitions of the Operational Semantics

We describe the rules governing the transitions between states, which comprise a program P, global memory M, store buffers B, queue-pairs QP and remote atomic locks A.

Transitions take the form shown on the right, which should be read as: if ϕ is true, then it is allowed for the system to transition from the state described by $\mathsf{P} \ldots \mathsf{QP}$ to the state described by $\mathsf{P}' \ldots \mathsf{QP}'$.

$$\frac{\phi}{\mathsf{P},\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \Rightarrow \mathsf{P}',\mathsf{M}',\mathsf{B}',\mathsf{A}',\mathsf{QP}'}$$

In practice, however, writing each transition rule in such a way would be verbose and hard to understand, as most transitions do not affect every part of the state. We can separate *program transitions* concerning P from *hardware transitions* concerning $\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}$. In order to synchronise the two where necessary, we assign labels to certain transitions and require that a labelled program transition only occur if it is matched by a hardware transition with the same label (or vice-versa). Labels are of the form $t : l$ where t is the thread executing at that step and $l \in \mathsf{Lab}$ is the label of the operation. *Silent transition*, which affect only the program (resp. only hardware) are written with the empty label, ϵ, and may be taken independently.

Fig. 4.2 shows the top-level rules of the operational semantics which govern this separation. We can henceforth consider the program and hardware transitions separately.

4.2.1 Program Transitions

Fig. 4.3 shows the program and command transitions (middle), labels (above) and expression rewriting rules (below). The transitions for non-remote commands are familiar from TSO. Notice that the transitions for get and put simply transition to `skip` with the relevant label; we know that this means there will be some relevant transition in the hardware. The transition to `skip` allows the program to continue executing, which we expect, as remote operations are handled asynchronously by the NIC.

This is similarly the case for the rules for remote-CAS and remote-FAA, highlighted in yellow. The expressions involved are required to be closed, similarly to the rules for local write and CAS; the expressions must be evaluated before the transition. It only makes sense to use a value, not an expression, in the label, since the corresponding hardware transition will only be concerned with values.

4.3 Hardware Domains

The upper section of Fig. 4.4 shows the hardware domains – that is, the states we are interested in *other* than the program. We have already described memory, store buffers, remote atomic locks and queue-pairs, but note that the structures B, A and QP are specifically maps from threads, nodes, and both, respectively, to the particular structures.

A remote atomic lock is a boolean, $\bot$ (available) or $\top$ (unavailable). A store buffer is a sequence of CPU writes and RDMA operations. A queue-pair is a tuple of three sequences **pipe**, $\mathbf{wb_R}$ and $\mathbf{wb_L}$, where **pipe** may contain any of the operations described below except for a confirmation notification, $\mathbf{wb_R}$ may contain NIC remote writes and NIC remote atomic writes, and $\mathbf{wb_L}$ may contain NIC local writes and confirmation notifications.

- $y^{\overline{n}} := x^n$ denotes a put operation where the value of local memory location x is yet to be read (NIC local read);
- $y^{\overline{n}} := v$ denotes a NIC remote write of value v to remote location y, which occurs as the latter part of a put;
- $\mathtt{ack_p}$ denotes the acknowledgement message returned by a put;
- $x^n := y^{\overline{n}}$ denotes a get operation where the value of the remote location y is yet to be read (pending NIC remote read)
- $x^n := v$ denotes a NIC local write of value v to local location x, which occurs as the latter part of a get or rRMW;
- $\mathtt{nCAS}(z^n, x^{\overline{n}}, v, v')$ denotes a remote (NIC) CAS towards remote location x, with expected value v, update value v', and returning to local location z;
- $\mathtt{nFAA}(z^n, x^{\overline{n}}, v)$ similarly denotes a NIC FAA towards x and returning to z, with increment value v;
- $y^{\overline{n}} :=_A v$ denotes a NIC remote write specifically in the case of an rRMW – it is necessary for this to be disambiguated from the NIC remote write of a put, as we will see in §4.3.2;

- `rfence` $\overline{n}$ denotes a remote fence towards node $\overline{n}$;
- `cn` denotes a confirmation of a successful NIC remote write.

4.3.1 Hardware Transitions

All remote commands enter the queue-pair pipe via the thread's store buffer. When the program takes a transition step labelled with a remote CAS or FAA, the hardware takes a transition with a matching label, which adds that operation to the store buffer – once again these are highlighted in yellow, in the lower part of Fig. 4.4. The seventh transition rule allows remote commands at the head of the store buffer to enter the pipe of a queue-pair, determined by their target node.

So far, we have seen that when an rRMW appears in the program, we can expect there to be a hardware transition which adds it to the store buffer, and later another hardware transition which removes it from the head of the store buffer and adds it to the suitable queue-pair.

The final hardware transition introduces the queue-pair transitions, indicated by $\rightarrow_{\texttt{sqp}}$[2]. When a particular queue-pair takes a transition step, involving memory and the global remote atomic lock, the hardware takes a suitable corresponding transition. The queue-pair transitions merely involve a particular subset of the hardware states, so the relationship is straightforward. This separation is purely made for clarity and simplification of the queue-pair transition rules.

4.3.2 Queue-Pair Transitions

The queue-pair transitions model most of the interesting behaviours discussed in §3.2. The new transition rules for rRMWs are highlighted in yellow. From Fig. 4.1, recall that remote operations enter the main **pipe** of the queue-pair, then are suitably processed until they exit the pipe, possibly adding a write to $\mathbf{wb_L}$ or $\mathbf{wb_R}$ (or both). Note that the pipe grows to the left, so throughout, α contains operations which are later in the program, while β contains earlier operations which have not yet completed.

The rules for remote fence, put, and get share a simple structure, where the premise for a transition either requires the operation to be at the head of the pipe $\mathbf{sqp.pipe} = \alpha \cdot (\text{operation})$, or allows it to be in the middle of the pipe $\mathbf{sqp.pipe} = \alpha \cdot (\text{operation}) \cdot \beta$, with some stipulation as to the operations allowed in β. In the prior case, the operation never executes before another, earlier operation; in the latter, it can execute before any operation in β which was issued before it. There may also be some requirement that buffers $\mathbf{wb_L}$ or $\mathbf{wb_R}$ contain no writes, due to PCIe guarantees: $\mathbf{wb_L} \in \{\texttt{cn}\}^*$ ($\mathbf{wb_L}$ contains only confirmation notifications) or $\mathbf{wb_R} = \epsilon$ (there are no operations in $\mathbf{wb_R}$). Consider, for example, the first step of a put, which is a NIC local read described by rule 2. The value of location x is read from memory, so long as $\mathbf{wb_L}$ has no pending writes and there are no other NIC local reads earlier in the pipe.

We can then describe the rules for rfence, put and get at a high level:

Remote fence (rule 1) An rfence may be removed from the pipe once it reaches the head (there are no earlier operations remaining to be processed). In combination

[2]SQP stands for simplified queue-pair. We only considered the simplified three-buffer queue-pair, so this disambiguation is technically unnecessary, but we maintain the notation for consistency with [10]

Program transitions: $\mathsf{Prog} \xrightarrow{\mathsf{Tid}:\mathsf{Lab}\uplus\{\varepsilon\}} \mathsf{Prog}$ **Command transitions:** $\mathsf{Comm} \xrightarrow{\mathsf{Lab}\uplus\{\varepsilon\}} \mathsf{Comm}$

$$\mathsf{Lab} \triangleq \bigcup_n \mathsf{Lab}_n \qquad l \in \mathsf{Lab}_n \triangleq \left\{ \begin{array}{l} \mathtt{lW}(x^n, v), \mathtt{lR}(x^n, v), \mathtt{CASS}(x^n, v_1, v_2), \mathtt{CASF}(x^n, v), \\ \mathtt{F}, \mathtt{P}(\overline{n}), \mathtt{Get}(x^n, y^{\overline{n}}), \mathtt{Put}(y^{\overline{n}}, x^n), \mathtt{rF}(\overline{n}), \\ \mathtt{nCAS}(y, x^{\overline{n}}, v_1, v_2), \mathtt{nFAA}(y, x^{\overline{n}}, v) \end{array} \middle| \begin{array}{l} x, y \in \mathsf{Loc}, \\ v, v_1, v_2 \in \mathsf{Val} \end{array} \right\}$$

$$\frac{\mathsf{C}_1 \xrightarrow{l} \mathsf{C}_1'}{\mathsf{C}_1; \mathsf{C}_2 \xrightarrow{l} \mathsf{C}_1'; \mathsf{C}_2} \qquad \frac{}{\mathtt{skip}; \mathsf{C} \xrightarrow{\varepsilon} \mathsf{C}} \qquad \frac{i \in \{1,2\}}{\mathsf{C}_1 + \mathsf{C}_2 \xrightarrow{\varepsilon} \mathsf{C}_i} \qquad \frac{}{\mathsf{C}^* \xrightarrow{\varepsilon} \mathtt{skip}} \qquad \frac{}{\mathsf{C}^* \xrightarrow{\varepsilon} \mathsf{C}; \mathsf{C}^*}$$

$$\frac{\mathsf{C} \rightsquigarrow \mathsf{C}'}{\mathsf{C} \xrightarrow{\varepsilon} \mathsf{C}'} \qquad \frac{\mathsf{elocs}(e) = \emptyset}{x := e \xrightarrow{\mathtt{lW}(x, [\![e]\!])} \mathtt{skip}} \qquad \frac{\mathsf{elocs}(e_{\mathrm{old}}) = \mathsf{elocs}(e_{\mathrm{new}}) = \emptyset \qquad v \neq [\![e_{\mathrm{old}}]\!]}{z := \mathtt{CAS}(x, e_{\mathrm{old}}, e_{\mathrm{new}}) \xrightarrow{\mathtt{CASF}(x,v)} z := v}$$

$$\frac{\mathsf{elocs}(e_{\mathrm{old}}) = \mathsf{elocs}(e_{\mathrm{new}}) = \emptyset}{z := \mathtt{CAS}(x, e_{\mathrm{old}}, e_{\mathrm{new}}) \xrightarrow{\mathtt{CASS}(x, [\![e_{\mathrm{old}}]\!], [\![e_{\mathrm{new}}]\!])} z := [\![e_{\mathrm{old}}]\!]} \qquad \frac{}{\mathtt{mfence} \xrightarrow{\mathtt{F}} \mathtt{skip}}$$

$$\frac{}{x := \overline{y} \xrightarrow{\mathtt{Get}(x, \overline{y})} \mathtt{skip}} \qquad \frac{}{\overline{y} := x \xrightarrow{\mathtt{Put}(\overline{y}, x)} \mathtt{skip}}$$

$$\frac{\mathsf{elocs}(e_{\mathrm{old}}) = \mathsf{elocs}(e_{\mathrm{new}}) = \emptyset \qquad v = [\![e_{\mathrm{old}}]\!] \qquad v' = [\![e_{\mathrm{new}}]\!]}{z := \mathtt{nCAS}(\overline{x}, e_{\mathrm{old}}, e_{\mathrm{new}}) \xrightarrow{\mathtt{nCAS}(z, \overline{x}, v, v')} \mathtt{skip}}$$

$$\frac{\mathsf{elocs}(e) = \emptyset \qquad v = [\![e]\!]}{z := \mathtt{nFAA}(\overline{x}, e) \xrightarrow{\mathtt{nFAA}(z, \overline{x}, v)} \mathtt{skip}} \qquad \frac{}{\mathtt{rfence}\ \overline{n} \xrightarrow{\mathtt{rF}(\overline{n})} \mathtt{skip}} \qquad \frac{}{\mathtt{poll}(\overline{n}) \xrightarrow{\mathtt{P}(\overline{n})} \mathtt{skip}}$$

$$\frac{}{\mathtt{assume}(x = v) \xrightarrow{\mathtt{lR}(x,v)} \mathtt{skip}} \qquad \frac{v \neq v'}{\mathtt{assume}(x \neq v') \xrightarrow{\mathtt{lR}(x,v)} \mathtt{skip}} \qquad \frac{\mathsf{P}(t) \xrightarrow{l} \mathsf{C}}{\mathsf{P} \xrightarrow{t:l} \mathsf{P}[t \mapsto \mathsf{C}]}$$

$$\begin{array}{rl}
x := e \rightsquigarrow \mathtt{assume}(y = v); x := e[v/y] & \text{for } y \in \mathsf{elocs}(e), v \in \mathsf{Val} \\
z := \mathtt{CAS}(x, e_{\mathrm{old}}, e_{\mathrm{new}}) \rightsquigarrow \mathtt{assume}(y = v); z := \mathtt{CAS}(x, e_{\mathrm{old}}[v/y], e_{\mathrm{new}}) & \text{for } y \in \mathsf{elocs}(e_{\mathrm{old}}), v \in \mathsf{Val} \\
z := \mathtt{CAS}(x, e_{\mathrm{old}}, e_{\mathrm{new}}) \rightsquigarrow \mathtt{assume}(y = v); z := \mathtt{CAS}(x, e_{\mathrm{old}}, e_{\mathrm{new}}[v/y]) & \text{for } y \in \mathsf{elocs}(e_{\mathrm{new}}), v \in \mathsf{Val} \\
z := \mathtt{nCAS}(\overline{x}, e_{\mathrm{old}}, e_{\mathrm{new}}) \rightsquigarrow \mathtt{assume}(y = v); z := \mathtt{nCAS}(\overline{x}, e_{\mathrm{old}}[v/y], e_{\mathrm{new}}) & \text{for } y \in \mathsf{elocs}(e_{\mathrm{old}}), v \in \mathsf{Val} \\
z := \mathtt{nCAS}(\overline{x}, e_{\mathrm{old}}, e_{\mathrm{new}}) \rightsquigarrow \mathtt{assume}(y = v); z := \mathtt{nCAS}(\overline{x}, e_{\mathrm{old}}, e_{\mathrm{new}}[v/y]) & \text{for } y \in \mathsf{elocs}(e_{\mathrm{new}}), v \in \mathsf{Val} \\
z := \mathtt{nFAA}(\overline{x}, e) \rightsquigarrow \mathtt{assume}(y = v); z := \mathtt{nFAA}(\overline{x}, e[v/y]) & \text{for } y \in \mathsf{elocs}(e), v \in \mathsf{Val}
\end{array}$$

Figure 4.3: The RDMA$^{\mathrm{TSO}}$ program and command transitions

$$\mathsf{M}\in\mathsf{Mem}\triangleq\mathsf{Loc}\to\mathsf{Val}\qquad \mathsf{B}\in\mathsf{SBMap}\triangleq\lambda t\in\mathsf{Tid}.\mathsf{SBuff}_{n(t)}$$
$$\mathsf{A}\in\mathsf{RAMap}\triangleq\lambda n.\{\bot,\top\}\qquad \mathsf{QP}\in\mathsf{SQPMap}\triangleq\lambda t.\left(\lambda\overline{n(t)}.\mathsf{SQPair}_n^{\overline{n}}\right)$$
$$\mathsf{b}\in\mathsf{SBuff}_n\triangleq\left\{x^n{:=}v,y^{\overline{n}}{:=}x^n,x^n{:=}y^{\overline{n}},\mathtt{nCAS}(z^n,x^{\overline{n}},v,v'),\mathtt{nFAA}(z^n,x^{\overline{n}},v),\mathtt{rfence}\ \overline{n}\right\}^*$$
$$\mathbf{sqp}\in\mathsf{SQPair}_n^{\overline{n}}\triangleq\mathsf{Pipe}_n^{\overline{n}}\times\mathsf{WBR}_n^{\overline{n}}\times\mathsf{WBL}_n^{\overline{n}}$$
$$\mathbf{wb_L}\in\mathsf{WBL}_n^{\overline{n}}\triangleq\left\{\mathtt{cn},x^n:=v\right\}^*\qquad \mathbf{wb_R}\in\mathsf{WBR}_n^{\overline{n}}\triangleq\left\{y^{\overline{n}}:=v,\ y^{\overline{n}}:=_A v\right\}^*$$
$$\mathbf{pipe}\in\mathsf{Pipe}_n^{\overline{n}}\triangleq\left\{\begin{array}{l}y^{\overline{n}}:=x^n,y^{\overline{n}}:=v,\ y^{\overline{n}}:=_A v\ ,\mathtt{ack_p},x^n:=y^{\overline{n}},x^n:=v,\\ \mathtt{nCAS}(z^n,x^{\overline{n}},v,v'),\mathtt{nFAA}(z^n,x^{\overline{n}},v),\mathtt{rfence}\ \overline{n}\end{array}\right\}^*$$

$$\frac{\mathsf{B}'=\mathsf{B}[t\mapsto(x:=v)\cdot\mathsf{B}(t)]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{lW}(x,v)}\mathsf{M},\mathsf{B}',\mathsf{A},\mathsf{QP}}\qquad\frac{(\mathsf{M}\lhd\mathsf{B}(t))(x)=v}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{lR}(x,v)}\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}}\qquad\frac{\mathsf{B}(t)=\varepsilon\qquad\mathsf{M}(x)=v_1}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{CASS}(x,v_1,v_2)}\mathsf{M}[x\mapsto v_2],\mathsf{B},\mathsf{A},\mathsf{QP}}$$

$$\frac{\mathsf{B}(t)=\varepsilon\qquad\mathsf{M}(x)=v}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{CASF}(x,v)}\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}}\qquad\frac{\mathsf{B}(t)=\varepsilon}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{F}}\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}}\qquad\frac{\mathsf{B}(t)=\mathsf{b}\cdot(x:=v)}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\varepsilon}\mathsf{M}[x\mapsto v],\mathsf{B}[t\mapsto\mathsf{b}],\mathsf{A},\mathsf{QP}}$$

$$\frac{\begin{array}{c}\mathsf{B}(t)=\mathsf{b}\cdot\mathsf{rc}^n\qquad\mathsf{rc}^n\in\left\{x:=y^{\overline{n}},y^{\overline{n}}:=x,\mathtt{nCAS}(z,\overline{x},v,v'),\mathtt{nFAA}(z,\overline{x},v),\mathtt{rfence}\ \overline{n}\right\}\\ \mathsf{QP}(t)(\overline{n})=\mathbf{sqp}\qquad\mathbf{sqp}'=\mathbf{sqp}[\mathbf{pipe}\mapsto\mathsf{rc}^n\cdot\mathbf{sqp}.\mathbf{pipe}]\end{array}}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\varepsilon}\mathsf{M},\mathsf{B}[t\mapsto\mathsf{b}],\mathsf{A},\mathsf{QP}[t\mapsto\mathsf{QP}(t)[\overline{n}\mapsto\mathbf{sqp}']]}$$

$$\frac{\mathsf{B}'=\mathsf{B}[t\mapsto(x:=\overline{y})\cdot\mathsf{B}(t)]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{Get}(x,\overline{y})}\mathsf{M},\mathsf{B}',\mathsf{A},\mathsf{QP}}\qquad\frac{\mathsf{B}'=\mathsf{B}[t\mapsto(\overline{y}:=x)\cdot\mathsf{B}(t)]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{Put}(\overline{y},x)}\mathsf{M},\mathsf{B}',\mathsf{A},\mathsf{QP}}$$

$$\frac{\mathsf{B}'=\mathsf{B}[t\mapsto\mathtt{nCAS}(z,\overline{x},v,v')\cdot\mathsf{B}(t)]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{nCAS}(z,\overline{x},v,v')}\mathsf{M},\mathsf{B}',\mathsf{A},\mathsf{QP}}\qquad\frac{\mathsf{B}'=\mathsf{B}[t\mapsto\mathtt{nFAA}(z,\overline{x},v)\cdot\mathsf{B}(t)]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{nFAA}(z,\overline{x},v)}\mathsf{M},\mathsf{B}',\mathsf{A},\mathsf{QP}}$$

$$\frac{\mathsf{B}'=\mathsf{B}[t\mapsto(\mathtt{rfence}\ \overline{n})\cdot\mathsf{B}(t)]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{rF}(\overline{n})}\mathsf{M},\mathsf{B}',\mathsf{A},\mathsf{QP}}$$

$$\frac{\mathsf{QP}(t)(\overline{n})=\mathbf{sqp}\qquad\mathbf{sqp}.\mathbf{wb_L}=\alpha\cdot\mathtt{cn}\qquad\mathbf{sqp}'=\mathbf{sqp}[\mathbf{wb_L}\mapsto\alpha]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{P}(\overline{n})}\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}[t\mapsto\mathsf{QP}(t)[\overline{n}\mapsto\mathbf{sqp}']]}\qquad\frac{\mathsf{M},\mathsf{A},\mathsf{QP}(t)(\overline{n})\to_{\mathtt{sqp}}\mathsf{M}',\mathsf{A}',\mathbf{sqp}\quad(Fig.\ 4.5)}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\varepsilon}\mathsf{M}',\mathsf{B},\mathsf{A}',\mathsf{QP}[t\mapsto\mathsf{QP}(t)[\overline{n}\mapsto\mathbf{sqp}]]}$$

$$\text{with}\quad(\mathsf{M}\lhd\alpha)(x)\triangleq\begin{cases}v & \text{if }\alpha=\beta\cdot(x:=v)\cdot-\wedge\forall v'.\,x:=v'\notin\beta\\ \mathsf{M}(x) & \text{if }\forall v.\,x:=v\notin\alpha\end{cases}$$

Figure 4.4: RDMA$^{\text{TSO}}$ simplified hardware domains (above) and hardware transitions (below)

with the fact that no other transition rule allows a step to be taken when there is an rfence later in the pipe, this enforces the behaviour that all remote operations prior to an rfence complete before it, and all later ones after it.

Put (rules 2-5) Rule 2: a NIC local read is performed, replacing the location x with its value in memory. Rule 3: the NIC remote write is sent to $\mathbf{wb_R}$, and an acknowledgement created in the pipe. Rule 4: the remote write is committed to memory once it reaches the head of the queue. Rule 5: the acknowledgement in the pipe is converted to a confirmation notification in $\mathbf{wb_L}$, so that it can be polled.

Get (rules 6-8) Rule 6: a NIC remote read replaces the location $\overline{y}$ with its value in memory. Rule 7: the NIC local write is sent to $\mathbf{wb_L}$, with a confirmation notification for the purpose of polling. Rule 8: the local write is committed to memory once there are no pending earlier writes in the queue.

Now, consider the rules for rRMWs, highlighted in yellow. These rules are more complicated due to the need to check and update the remote atomic lock for the target node, which we see as $\mathsf{A}(n(\overline{x})) = \bot$ (the remote atomic lock for the target node is available), and $\mathsf{A}' = \mathsf{A}[n(\overline{x}) \mapsto \top]$ (update the remote atomic lock for the target node to indicate it is busy). We also have distinct rules for success and failure of nCAS, depending on whether the remote memory location holds the expected value ($\mathsf{M}(\overline{x}) = v$ or $\mathsf{M}(\overline{x}) \neq v$).

The rules can then be interpreted as follows:

(Rule 9) A failed nCAS read – the remote memory location does not hold the expected value. This read can only occur when the remote atomic lock is available, otherwise it would violate the atomicity guarantee. The value of $\overline{x}$ is read, and a NIC local write is added to the pipe to return that value to z. This is then handled by the same rules as for a get. The remote atomic lock is not obtained, since the remote location will not be written to.

(Rule 10) A successful nCAS read – the remote location contains the expected value. Once again, this requires that the remote atomic lock be available, and it is also obtained to ensure atomicity until the remote location is written to. A NIC local write is added to the pipe (similarly to 9), and a NIC remote atomic write to update the remote location is also added.

(Rule 11) The remote read of an nFAA – this is unconditionally successful. It is very similar to a successful nCAS, but the value for the NIC remote atomic write is calculated by adding v to the value of $\overline{x}$ in memory.

(Rule 12) A NIC remote atomic write in the pipe is processed into $\mathbf{wb_R}$ similarly to a regular NIC remote write.

(Rule 13) A NIC remote atomic write is committed to memory, and the remote atomic lock is released.

$$\frac{\mathbf{sqp.pipe} = \alpha \cdot (\texttt{rfence}\ \overline{n})}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \mathbf{sqp}[\mathbf{pipe} \mapsto \alpha]}$$

$$\frac{\begin{array}{c}\mathbf{sqp.pipe} = \alpha \cdot (\overline{y} := x) \cdot \beta \quad \mathbf{wb_L} \in \{\mathtt{cn}\}^* \\ \beta \in \{y' := v', y' :=_A v', \overline{y'} := v', x' := \overline{y'}, \mathtt{nCAS}(z, \overline{x}, v, v'), \mathtt{nFAA}(z, \overline{x}, v), \mathtt{ack_p}\}^*\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \mathbf{sqp}[\mathbf{pipe} \mapsto \alpha \cdot (\overline{y} := \mathsf{M}(x)) \cdot \beta]}$$

$$\frac{\begin{array}{c}\mathbf{sqp.pipe} = \alpha \cdot (\overline{y} := v) \cdot \beta \\ \beta \in \{x' := \overline{y'}, x' := v', \mathtt{ack_p}\}^* \\ \mathbf{sqp'} = \mathbf{sqp}[\mathbf{pipe} \mapsto \alpha \cdot \mathtt{ack_p} \cdot \beta][\mathbf{wb_R} \mapsto (\overline{y} := v) \cdot \mathbf{sqp.wb_R}]\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \mathbf{sqp'}}$$

$$\frac{\mathbf{sqp.wb_R} = \alpha \cdot (\overline{y} := v)}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\mathtt{sqp}} \mathsf{M}[\overline{y} \mapsto v], \mathsf{A}, \mathbf{sqp}[\mathbf{wb_R} \mapsto \alpha]} \qquad \frac{\begin{array}{c}\mathbf{sqp.pipe} = \alpha \cdot \mathtt{ack_p} \\ \mathbf{sqp'} = \mathbf{sqp}[\mathbf{pipe} \mapsto \alpha][\mathbf{wb_L} \mapsto \mathtt{cn} \cdot \mathbf{sqp.wb_L}]\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \mathbf{sqp'}}$$

$$\frac{\begin{array}{c}\mathbf{sqp.pipe} = \alpha \cdot (x := \overline{y}) \cdot \beta \qquad \beta \in \{x' := \overline{y'}, x' := v', \mathtt{ack_p}\}^* \\ \mathbf{sqp.wb_R} = \varepsilon \quad \mathbf{sqp'} = \mathbf{sqp}[\mathbf{pipe} \mapsto \alpha \cdot (x := \mathsf{M}(\overline{y})) \cdot \beta]\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \mathbf{sqp'}}$$

$$\frac{\begin{array}{c}\mathbf{sqp.pipe} = \alpha \cdot (x := v) \\ \mathbf{sqp'} = \mathbf{sqp}[\mathbf{pipe} \mapsto \alpha][\mathbf{wb_L} \mapsto \mathtt{cn} \cdot (x := v) \cdot \mathbf{sqp.wb_L}]\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \mathbf{sqp'}} \qquad \frac{\begin{array}{c}\mathbf{sqp.wb_L} = \alpha \cdot (x := v) \cdot \beta \quad \beta \in \{\mathtt{cn}\}^* \\ \mathbf{sqp'} = \mathbf{sqp}[\mathbf{wb_L} \mapsto \alpha \cdot \beta]\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\mathtt{sqp}} \mathsf{M}[x \mapsto v], \mathsf{A}, \mathbf{sqp'}}$$

$$\frac{\begin{array}{c}\mathbf{sqp.pipe} = \alpha \cdot \mathtt{nCAS}(z, \overline{x}, v, v') \cdot \beta \quad \mathbf{sqp.wb_R} = \epsilon \\ \mathsf{A}(n(\overline{x})) = \bot \quad \mathsf{M}(\overline{x}) \neq v \quad \beta \in \{x' := \overline{y'}, x' := v', \mathtt{ack_p}\}^* \\ \mathbf{sqp'} = \mathbf{sqp}[\mathbf{pipe} \mapsto \alpha \cdot (z := \mathsf{M}(\overline{x})) \cdot \beta]\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \mathbf{sqp'}}$$

$$\frac{\begin{array}{c}\mathbf{sqp.pipe} = \alpha \cdot \mathtt{nCAS}(z, \overline{x}, v, v') \cdot \beta \quad \mathbf{sqp.wb_R} = \epsilon \\ \mathsf{M}(\overline{x}) = v \quad \beta \in \{x' := \overline{y'}, x' := v', \mathtt{ack_p}\}^* \\ \mathsf{A}(n(\overline{x})) = \bot \quad \mathsf{A}' = \mathsf{A}[n(\overline{x}) \mapsto \top]\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}', \mathbf{sqp}[\mathbf{pipe} \mapsto \alpha \cdot (z := v) \cdot (\overline{x} :=_A v') \cdot \beta]}$$

$$\frac{\begin{array}{c}\mathbf{sqp.pipe} = \alpha \cdot \mathtt{nFAA}(z, \overline{x}, v) \cdot \beta \quad \mathbf{sqp.wb_R} = \epsilon \\ \mathsf{M}(\overline{x}) + v = v' \quad \beta \in \{x' := \overline{y'}, x' := v', \mathtt{ack_p}\}^* \\ \mathsf{A}(n(\overline{x})) = \bot \quad \mathsf{A}' = \mathsf{A}[n(\overline{x}) \mapsto \top]\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}', \mathbf{sqp}[\mathbf{pipe} \mapsto \alpha \cdot (z := v) \cdot (\overline{x} :=_A v') \cdot \beta]}$$

$$\frac{\begin{array}{c}\mathbf{sqp.pipe} = \alpha \cdot (\overline{x} :=_A v) \cdot \beta \\ \mathbf{wb_R}' = (\overline{x} :=_A v) \cdot \mathbf{sqp.wb_R} \\ \beta \in \{x' := \overline{y'}, x' := v', \mathtt{ack_p}\}^* \\ \mathbf{sqp'} = \mathbf{sqp}[\mathbf{pipe} \mapsto \alpha \cdot \beta][\mathbf{wb_R} \mapsto \mathbf{wb_R}']\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \mathbf{sqp'}} \qquad \frac{\begin{array}{c}\mathbf{sqp.wb_R} = \alpha \cdot (\overline{x} :=_A v) \\ \mathsf{A}' = \mathsf{A}[n(\overline{x}) \mapsto \bot] \\ \mathbf{sqp'} = \mathbf{sqp}[\mathbf{wb_R} \mapsto \alpha]\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\mathtt{sqp}} \mathsf{M}[\overline{x} \mapsto v], \mathsf{A}', \mathbf{sqp'}}$$

Figure 4.5: Queue-pair transitions of the simplified RDMA$^{\text{TSO}}$ operational semantics

$y=0$	$x=0$
$x^2 := 1$ $\texttt{poll}(2)$ $a := y$	$y^1 := 1$ $\texttt{poll}(1)$ $b := x$

(a) $(a, b) = (0, 0)$ ✓

$y=0$	$x=0$
$c := \texttt{nFAA}(x^2, 1)$ $\texttt{poll}(2)$ $a := y$	$d := \texttt{nFAA}(y^1, 1)$ $\texttt{poll}(1)$ $b := x$

(b) $(a, b) = (0, 0)$ ✓

Figure 4.6: Remote RMWs exhibit the same weak behaviours as puts, including store buffering even in the presence of polls.

Discussion: Queue-Pair Transitions

The atomicity requirement motivated the addition of A, however it previously took on a different form. Initially, rather than extending **pipe** and introducing A, only A was present. When an rRMW was in progress, the description of it would be held in A, rather than a boolean. This simplified the atomicity checks, however made reordering much more complicated, as other rules would need to check A as well as β for reordering purposes. Using a boolean instead, and maintaining the in-progress rRMW in the pipe, made the semantics much clearer.
It is also noteworthy that the local and remote write of an rRMW may be processed in either order. In particular, this means that local write may reach memory while the remote write is still in $\mathbf{wb_R}$. The local write indicates the completion of the rRMW for the local thread, so this seems problematic: the rRMW is considered complete for the local side before it has finished on the remote side. In fact, this is because rRMWs allow all the same weak behaviours as puts, other than those excluded by the atomicity guarantee. As discussed in §3.1, some of these cannot be prevented by a poll as we would expect. See Fig. 4.6 which replicates the store buffering example from Fig. 3.2d, and shows the same behaviour with rRMWs.

5 Declarative Semantics

A declarative semantics, in contrast to an operational one, describes only the events that occur in a system, not the state of the system itself. An execution is represented by a *graph*, with various relations over events. For example, given an event r, we say that it "reads-from" event w if r reads the value written to memory by w. We write $(w, r) \in \mathsf{rf}$ in this case. We then constrain these relations suitably to only allow execution graphs which make sense in the context of a program: considering rf again, we would naturally only allow $(w, r) \in \mathsf{rf}$ if the values of the read and write match.

Then, we know that an execution of a program is allowed if the graph of the execution is consistent. Contrast this with the operational semantics: there, our guarantee comes from the individual transition rules; here, it is due to the overall structure of the graph.

5.1 Events and Executions

An *execution* is a graph comprising a set of events and several relations over events; events are represented as graph nodes, and the relations are edges. An event has a unique identifier ι, is created by a thread $t \in \mathsf{Tid}$, and has an event label $l \in \mathsf{ELab}$ which describes the event.

Definition 1 (Labels and events)**.** Each event label is associated to a node n. The set of *event labels of node n* is denoted by $l \in \mathsf{ELab}_n$, where l is a tuple with one of the following forms:

- (CPU) local read: $l = \mathtt{lR}(x^n, v_\mathrm{r})$
- (CPU) local write: $l = \mathtt{lW}(x^n, v_\mathrm{w})$
- (CPU) CAS: $l = \mathtt{CAS}(x^n, v_\mathrm{r}, v_\mathrm{w})$
- (CPU) memory fence: $l = \mathtt{F}$
- (CPU) poll: $l = \mathtt{P}(\overline{n})$
- NIC local read: $l = \mathtt{nlR}(x^n, v_\mathrm{r}, \overline{n})$
- NIC remote write: $l = \mathtt{nrW}(y^{\overline{n}}, v_\mathrm{w})$
- NIC remote read: $l = \mathtt{nrR}(y^{\overline{n}}, v_\mathrm{r})$
- NIC local write: $l = \mathtt{nlW}(x^n, v_\mathrm{w}, \overline{n})$
- NIC fence: $l = \mathtt{nF}(\overline{n})$
- NIC atomic remote read: $l = \mathtt{narR}(y^{\overline{n}}, v_\mathrm{r})$
- NIC atomic remote write: $l = \mathtt{narW}(y^{\overline{n}}, v_\mathrm{w})$

The set of event labels are defined $\mathsf{ELab} \triangleq \bigcup_n \mathsf{ELab}_n$.
An *event*, $\mathsf{e} \in \mathsf{Event}$, is a triple (ι, t, l), where $\iota \in \mathbb{N}$, $t \in \mathsf{Tid}$ and $l \in \mathsf{ELab}_{n(t)}$.

We distinguish between events associated with the CPU (left) or NIC (right), with the prefix `n` used for all NIC event labels. Note that a put, get, or rRMW is modelled by multiple events: a put $\overline{x} := y$ comprises a NIC local read event of type `nlR` (on y) followed by a NIC remote write event `nrW` (on $\overline{x}$); conversely a get $x := \overline{y}$ comprises events of type `nrR` (on $\overline{y}$) and `nlW` (on x). A successful rRMW (either successful nCAS or nFAA) is modelled by three events of type `narR`, `narW` and `nlW`, while a failed rRMW (nCAS only) is modelled by only `narR` and `nlW`.

For a given label l, we write $\mathsf{type}(l)$, $\mathtt{loc}(l)$, $v_{\mathrm{r}}(l)$, $v_{\mathrm{w}}(l)$, $n(l)$ and $\overline{n}(l)$ for the type, location, value read or written, and local or remote node, where applicable. For example, consider $l = \mathtt{nlR}(x^n, v_{\mathrm{r}}, \overline{n})$:

- $\mathsf{type}(\mathtt{nlR}(x^n, v_{\mathrm{r}}, \overline{n})) = \mathtt{nlR}$
- $\mathtt{loc}(\mathtt{nlR}(x^n, v_{\mathrm{r}}, \overline{n})) = x$
- $v_{\mathrm{r}}(\mathtt{nlR}(x^n, v_{\mathrm{r}}, \overline{n})) = v_{\mathrm{r}}$
- $v_{\mathrm{w}}(\mathtt{nlR}(x^n, v_{\mathrm{r}}, \overline{n}))$ is undefined
- $n(\mathtt{nlR}(x^n, v_{\mathrm{r}}, \overline{n})) = n$
- $\overline{n}(\mathtt{nlR}(x^n, v_{\mathrm{r}}, \overline{n})) = \overline{n}$

We write $\iota(\mathsf{e})$, $t(\mathsf{e})$, $l(\mathsf{e})$ for the relevant constituents of an event tuple $\mathsf{e} = (\iota, t, l)$. We lift the functions on event labels to functions on events, for example $\mathsf{type}(\mathsf{e}) \triangleq \mathsf{type}(l(\mathsf{e}))$.

Issue and Observation Points. Some types of events do not occur instantaneously: for example, a local write event `lW` first enters the store before, before later being committed to memory. We therefore distinguish between the point at which an event is *issued* by the CPU or NIC, and the point at which it is *observed*, when its effect becomes visible in memory. An event is *instantaneous* if it either has no visible effect on memory, or if it affects memory immediately, as is the case for a local CAS operation. For instantaneous events, the issue and observation points coincide.

Notation. Once again, we follow and extend the notation of [10]. For a set A and relations r, r_1, r_2, we write:

$\boldsymbol{r^+}$ for the transitive closure of r;

$\boldsymbol{r^{-1}}$ for the inverse of r;

$\boldsymbol{r|_A} \triangleq r \cap (A \times A)$ for the restriction of r to set A;

$\boldsymbol{[A]} \triangleq \{(a, a) \mid a \in A\}$ for the identity relation

$\boldsymbol{r_1; r_2} \triangleq \{(a, b) \mid \exists c.(a, c) \in r_1 \wedge (c, b) \in r_2\}$ for relational composition;

$\boldsymbol{r|_{\mathbf{imm}}} \triangleq r \setminus (r; r)$ for the immediate edges in r, when r is a strict partial order.

For a set of events E, location x and label type `X`, we also define:

$E_x \triangleq \{\mathsf{e} \in E \mid \mathtt{loc}(\mathsf{e}) = x\}$, the events towards x;

$E.\mathtt{X} \triangleq \{\mathsf{e} \in E \mid \mathsf{type}(\mathsf{e}) = \mathtt{X}\}$, the events of type `X`;

$E.\mathcal{R} \triangleq E.\mathtt{lR} \cup E.\mathtt{CAS} \cup E.\mathtt{nlR} \cup E.\mathtt{nrR} \cup E.\mathtt{narR}$, the set of reads;

$E.\mathcal{W} \triangleq E.\mathtt{lW} \cup E.\mathtt{CAS} \cup E.\mathtt{nlW} \cup E.\mathtt{nrW} \cup E.\mathtt{narW}$, the set of writes;

$E.\mathtt{Inst} \triangleq E \setminus (E.\mathtt{lW} \cup E.\mathtt{nlW} \cup E.\mathtt{nrW} \cup E.\mathtt{narW})$, the set of instantaneous events.

Finally, we define the following relations:

$$\begin{array}{lrcl}
\textbf{Same-location:} & \mathsf{sloc} & \triangleq & \left\{(\mathsf{e}, \mathsf{e}') \in \mathsf{Event}^2 \mid \mathtt{loc}(\mathsf{e}) = \mathtt{loc}(\mathsf{e}')\right\} \\
\textbf{Same-thread:} & \mathsf{sthd} & \triangleq & \left\{(\mathsf{e}, \mathsf{e}') \in \mathsf{Event}^2 \mid t(\mathsf{e}) = t(\mathsf{e}')\right\} \\
\textbf{Same-queue-pair:} & \mathtt{sqp} & \triangleq & \left\{(\mathsf{e}, \mathsf{e}') \in \mathsf{Event}^2 \mid t(\mathsf{e}) = t(\mathsf{e}') \wedge \overline{n}(\mathsf{e}) = \overline{n}(\mathsf{e}')\right\}
\end{array}$$

Note that these relations are all symmetric, and $\mathtt{sqp} \subseteq \mathsf{sthd}$. Given events E, we write $E.\mathsf{sloc}$ for $\mathsf{sloc}|_E$, likewise for $E.\mathsf{sthd}$ and $E.\mathtt{sqp}$.

		$x = y = 0$
$a := \texttt{nCAS}(x^3, 0, 7)$	$b := \texttt{nCAS}(y^3, 0, 42)$	$x := 1$ $y := 2$ $y := 3$ $x := 4$

(a) $(x, b) = (7, 2)$ ✗

Figure 5.1: Remote RMWs are guaranteed to be mutually exclusive whenever they are towards the same node, even if they target different locations.

Definition 2 (Pre-executions)**.** A *pre-execution* is a tuple $G = \langle E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{pf}, \mathsf{nfo}, \mathsf{rao} \rangle$, where:

- $E \subseteq \mathsf{Event}$ is the set of events and includes a set of *initialisation* events, $E^0 \subseteq E$, comprising a single write with label $\texttt{lW}(x, 0)$ for each $x \in \mathsf{Loc}$.
- $\mathsf{po} \subseteq E \times E$ is the '*program order*' relation defined as a disjoint union of strict total orders, each ordering the events of one thread, with $E^0 \times (E \setminus E^0) \subseteq \mathsf{po}$.
- $\mathsf{rf} \subseteq E.\mathcal{W} \times E.\mathcal{R}$ is the '*reads-from*' relation on events of the same location with matching values; i.e. $(a, b) \in \mathsf{rf} \Rightarrow (a, b) \in \mathsf{sloc} \wedge v_{\mathrm{w}}(a) = v_{\mathrm{r}}(b)$. Moreover, rf is total and functional on its range: every read in $E.\mathcal{R}$ is related to exactly one write in $E.\mathcal{W}$.
- $\mathsf{mo} \triangleq \bigcup_{x \in \mathsf{Loc}} \mathsf{mo}_x$ is the '*modification-order*', where each mo_x is a strict total order on $E.\mathcal{W}_x$ with $E^0_x \times (E.\mathcal{W}_x \setminus E^0_x) \subseteq \mathsf{mo}_x$ describing the order in which writes on x reach the memory.
- $\mathsf{pf} \subseteq (E.\texttt{nlW} \cup E.\texttt{nrW}) \times E.\texttt{P}$ is the '*polls-from*' relation, relating earlier (in program-order) NIC writes to later poll operations on the *same queue pair*; i.e. $\mathsf{pf} \subseteq \mathsf{po} \cap \texttt{sqp}$. Moreover, pf is functional on its domain (every NIC write can be be polled at most once), and pf is total and functional on its range (every poll in $E.\texttt{P}$ polls from exactly one NIC write).
- $\mathsf{nfo} \subseteq E.\texttt{sqp}$ is the '*NIC flush order*', such that for all $(a, b) \in E.\texttt{sqp}$, if $a \in E.\texttt{nlR}, b \in E.\texttt{nlW}$, then $(a, b) \in \mathsf{nfo} \cup \mathsf{nfo}^{-1}$, and if $a \in (E.\texttt{nrR} \cup E.\texttt{narR}), b \in (E.\texttt{nrW} \cup E.\texttt{narW})$, then $(a, b) \in \mathsf{nfo} \cup \mathsf{nfo}^{-1}$.
- $\mathsf{rao} \triangleq \bigcup_{n \in \mathsf{Node}} \mathsf{rao}_n$ is the '*remote-atomic-order*', where each rao_n is a strict total order on $\{e \mid e \in E.\texttt{narR} \wedge \overline{n}(e) = n\}$ describing the order in which remote atomics towards n are executed.

The definitions of po, rf and mo are familiar from TSO, while pf and nfo are introduced in [10]. The reason nfo is important is subtle: recall from §3.1 that we model the requirement from PCIe that a NIC local read flushes pending NIC remote writes on the same queue pair, and likewise for NIC local reads/writes. We will therefore need to maintain an order over these events to check that the property holds.

We introduce rao, which totally orders NIC remote atomic reads towards a given node. This will help us to check the atomicity guarantee.

Discussion: Pre-executions

At first glance, it might appear that rao should be a total order *per-location*, rather than per-thread; indeed, it was so in an earlier draft of the semantics. However, this results in an atomicity guarantee which is too weak. Consider the example shown in Fig. 5.1. When we observe the outcome $x = 7$, this means that every local write on node 3 occurred between the read and write of the nCAS towards x: in particular, $y = 2$ is only ever true within this critical period, not outside it. Thus if $x = 7$ then we cannot have $b = 2$, as this would imply the two rRMWs were not mutually exclusive. If atomicity were only guaranteed per-location, then this would be allowed, but this is forbidden by the specification.

Derived Relations. Given a pre-execution $\langle E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{pf}, \mathsf{nfo}, \mathsf{rao}\rangle$, we define the following *derived* relations:

- $\mathsf{rb} \triangleq (\mathsf{rf}^{-1}; \mathsf{mo}) \setminus [E]$ is the *reads-before* relation, relating each read r to writes that are mo-after the write from which r reads.
- $\mathsf{rf_b} \triangleq [\mathtt{lW}]; (\mathsf{rf} \cap \mathsf{sthd}); [\mathtt{lR}]$ is the rf-*buffer* relation, which includes rf edges only for CPU operations on the same thread, which thus share a store buffer; therefore when $w \xrightarrow{\mathsf{rf_b}} r$, it may be that the write w is not yet visible (committed to memory) when it is read by r, since CPU reads check the store buffer.
- $\mathsf{rf_{\overline{b}}} \triangleq \mathsf{rf} \setminus \mathsf{rf_b}$ is the $\mathsf{rf_b}$-complement: if $w \xrightarrow{\mathsf{rf_{\overline{b}}}} r$, then r only occurs after w is observable.
- $\mathsf{rb_b} \triangleq [\mathtt{lR}]; (\mathsf{rb} \cap \mathsf{sthd}); [\mathtt{lW}]$ is the rb-*buffer* relation, analogously.
- $\mathsf{ar} \triangleq [\mathtt{narW}]; (\mathsf{po}|_{\mathrm{imm}}{}^{-1})$ is the *atomic-write-to-read* relation, connecting the remote write of an rRMW to their corresponding read.

Note that these derived relations contain no additional information. We introduce them for ease and brevity of notation.

Preserved Program Order. In order to model the reordering rules of §3.1, we must identify which events in po are *issued* in order, and which are *observed* in order. The observation point of an event is no earlier than its issue point, so two events in po are only observed in order if they are issued in order. Furthermore, when the po-earlier event is instantaneous, the events are observed in order if and only if they are issued in order.

We therefore define two relations: ippo, the *issue-preserved-program-order* relation, and oppo, the *observation-preserved-program-order* relation, where $\mathsf{oppo} \subseteq \mathsf{ippo} \subseteq \mathsf{po}$. The tables in Fig. 5.2 show these relations, with the new cells due to rRMW events highlighted. Each row indicates the po-earlier event, while each column indicates that which is po-later. A cell labelled ✓ indicates the event pair is in ippo (resp. oppo) and must be issued (resp. observed) in program order, while ✗ indicates they are not in ippo/oppo and may be issued/observed out of program order. The label `sqp` indicates that the events are in ippo/oppo if they are events on the same queue-pair.

We can observe high-level reordering rules by looking at each quadrant of the two tables, which partition the event pairs by their categorisation as CPU or NIC events. The top left quadrant contains pairs of CPU events. Observe that CPU events are always issued in program order, and only an earlier CPU write may be observed out of

Later in Program Order

ippo			CPU					NIC						
			1	2	3	4	5	6	7	8	9	10	11	12
			lR	lW	CAS	F	P	nlR	nrW	narR	narW	nrR	nlW	nF
Earlier in Program Order: CPU	A	lR	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	B	lW	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	C	CAS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	D	F	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	E	P	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Earlier in Program Order: NIC	F	nlR	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp	sqp	sqp
	G	nrW	✗	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp	sqp
	H	narR	✗	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp	sqp
	I	narW	✗	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp	sqp
	J	nrR	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	sqp	sqp
	K	nlW	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	sqp	sqp
	L	nF	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp	sqp	sqp

Later in Program Order

oppo			CPU					NIC						
			1	2	3	4	5	6	7	8	9	10	11	12
			lR	lW	CAS	F	P	nlR	nrW	narR	narW	nrR	nlW	nF
Earlier in Program Order: CPU	A	lR	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	B	lW	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
	C	CAS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	D	F	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	E	P	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Earlier in Program Order: NIC	F	nlR	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp	sqp	sqp
	G	nrW	✗	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp	✗
	H	narR	✗	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp	sqp
	I	narW	✗	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	✗	✗
	J	nrR	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	sqp	sqp
	K	nlW	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	sqp	✗
	L	nF	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp	sqp	sqp

Figure 5.2: The RDMA^{TSO} ordering constraints on ippo (above) and oppo (below), where ✓ denotes that instructions are ordered (and cannot be reordered), ✗ denotes they are not ordered (and may be reordered), and `sqp` denotes they are ordered iff they are on the same queue pair. Highlighted: new cells due to rRMWs.

order, as all other CPU events are instantaneous. The bottom left quadrants shows that an earlier NIC event may always be issued or observed after a later CPU event, matching our intuition that NIC events execute concurrently, as if in a separate thread; conversely the top right shows that earlier CPU events always complete before later NIC events. In the bottom right quadrant, we can see that a pair of NIC events are only ordered if they are on the same queue-pair.

The relations ippo and oppo differ in only six cells. A CPU write may be buffered and hence not observed by a later CPU read or poll (B1 and B5). Other CPU writes and CAS or fence operations go via the store buffer, so earlier writes will be observed first. Similarly, a remote fence may be observed before an earlier NIC remote (atomic) write (resp. local), if that write is buffered in $\mathbf{wb_R}$ (resp. $\mathbf{wb_L}$) (G12 and I12, resp. K12). Finally, a po-later `nlW` may be observed before a po-earlier `narW` (I11). This occurs specifically in the case where both are created by the same rRMW, because the writes are sent to $\mathbf{wb_L}$ and $\mathbf{wb_R}$ respectively and may be committed in either order.

Definition 3 (Executions)**.** A pre-execution $G = \langle E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{pf}, \mathsf{nfo}, \mathsf{rao}\rangle$ is *well-formed* if the following hold for all w, r, w_1, w_2, p_2:

1. Poll events poll-from the oldest non-polled remote operation on the same queue pair:
   if $w_1 \in G.\mathtt{nlW} \cup G.\mathtt{nrW}$ and $w_1 \xrightarrow{\mathsf{po}\cap\mathsf{sqp}} w_2 \xrightarrow{\mathsf{pf}} p_2$, then there exists p_1 such that $w_1 \xrightarrow{\mathsf{pf}} p_1 \xrightarrow{\mathsf{po}} p_2$.

2. Each put (resp. get) operation corresponds to two events: a read and a write with the read immediately preceding the write in po: (a) if $r \in G.\mathtt{nlR}$ (resp. $r \in G.\mathtt{nrR}$), then $(r, w) \in \mathsf{po}|_{\mathrm{imm}}$ for some $w \in G.\mathtt{nrW}$ ($w \in G.\mathtt{nlW}$); and (b) if $w \in G.\mathtt{nrW}$ then $(r, w) \in \mathsf{po}|_{\mathrm{imm}}$ for some $r \in G.\mathtt{nlR}$. The case $w \in G.\mathtt{nlW}$ is handled by (6) below.

3. Read and write events of a put (resp. get) have matching values:
   if $(r, w) \in G.\mathsf{po}|_{\mathrm{imm}}$, $\mathsf{type}(r) \in \{\mathtt{nlR}, \mathtt{nrR}\}$ and $\mathsf{type}(w) \in \{\mathtt{nlW}, \mathtt{nrW}\}$, then $v_{\mathrm{r}}(r) = v_{\mathrm{w}}(w)$.

4. Each rRMW operation corresponds to either an atomic remote read followed by a local write, or an atomic remote read, followed by an atomic remote write, followed by a local write: (a) if $r \in G.\mathtt{narR}$ then $(r, w_1) \in \mathsf{po}|_{\mathrm{imm}}$ for some $w_1 \in G.\mathtt{narW} \cup G.\mathtt{nlW}$, and if $w_1 \in G.\mathtt{narW}$ then $(w_1, w_2) \in \mathsf{po}|_{\mathrm{imm}}$ for some $w_2 \in G.\mathtt{nlW}$, and (b) if $w_1 \in G.\mathtt{narW}$ then $(r, w_1) \in \mathsf{po}|_{\mathrm{imm}}$ for some $r \in \mathtt{narR}$, and $(w_1, w_2) \in \mathsf{po}|_{\mathrm{imm}}$ for some $w_2 \in \mathtt{nlW}$. The case for $w_2 \in \mathtt{nlW}$ is handled by (6) below.

5. Remote atomic read and local write events of an rRMW have matching values: if $(r, w) \in G.\mathsf{po}|_{\mathrm{imm}}$, $\mathsf{type}(r) = \mathtt{narR}$ and $\mathsf{type}(w) = \mathtt{nlW}$, then $v_{\mathrm{r}}(r) = v_{\mathrm{w}}(w)$; and if $(r, w_1), (w_1, w_2) \in G.po|_{\mathrm{imm}}$, $\mathsf{type}(r) = \mathtt{narR}$, $\mathsf{type}(w_1) = \mathtt{narW}$ and $\mathsf{type}(w_2) = \mathtt{nlW}$, then $v_{\mathrm{r}}(r) = v_{\mathrm{w}}(w_2)$.

6. (2) and (4) auxiliary in the case of $w \in \mathtt{nlW}$. If $w \in G.\mathtt{nlW}$ then either:
   (a) $(r, w) \in \mathsf{po}|_{\mathrm{imm}}$ for some $r \in G.\mathtt{nrR}$ or
   (b) $(r, w) \in \mathsf{po}|_{\mathrm{imm}}$ for some $r \in G.\mathtt{narR}$ or
   (c) $(r, w'), (w', w) \in \mathsf{po}|_{\mathrm{imm}}$ for some $r \in G.\mathtt{narR}$ and $w' \in G.\mathtt{narW}$.

An *execution* is a pre-execution (Def. 2) that is well-formed.

Given an execution G, we write $G.E$, $G.\mathsf{mo}$, $G.\mathsf{ippo}$ and so forth to project the components and derived relations of G. When the execution is question is clear, we simply write E, mo or similar.

Definition 4 (RDMA$^{\text{TSO}}$-consistency)**.** An execution $\langle E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{pf}, \mathsf{nfo}, \mathsf{rao}\rangle$ is *RDMA^{TSO}-consistent* iff 1. ib is irreflexive; and 2. ob is irreflexive, where:

$$\mathsf{ib} \triangleq \left(\mathsf{ippo} \cup \mathsf{rf} \cup \mathsf{pf} \cup \mathsf{nfo} \cup \mathsf{rb}_{\mathsf{b}} \cup \mathsf{rao} \cup (\mathsf{ob}; [\mathtt{Inst}])\right)^{+} \qquad \text{('issued-before')}$$

$$\mathsf{ob} \triangleq \left(\mathsf{oppo} \cup \mathsf{rf}_{\overline{\mathsf{b}}} \cup ([\mathtt{nlW}]; \mathsf{pf}) \cup \mathsf{nfo} \cup \mathsf{rb} \cup \mathsf{mo} \cup (\mathsf{ar}; \mathsf{rao}) \cup ([\mathtt{Inst}]; \mathsf{ib})\right)^{+} \qquad \text{('observed-before')}$$

These relations extend ippo and oppo respectively to describe the issue and observation orders across threads and nodes. They are required to be irreflexive, i.e. an event cannot be issued or observed before itself.

The remaining components of ib are (a) rf: if $w \xrightarrow{\mathsf{rf}} r$ then w was at least issued (if not observed) before r – recall that if the read and write are both CPU events on the same thread, w may not be observable; (b) pf: similarly $w \xrightarrow{\mathsf{pf}} p$ only if w was issued before p; (c) nfo: NIC events arrive in $\mathbf{wb_L}/\mathbf{wb_R}$ in the order they are issued; (d) rb_{b}: if $r \xrightarrow{\mathsf{rb}_{\mathsf{b}}} w$, then r must be issued before w, otherwise r would read from w or an mo-later w'; (e) rao: remote atomic reads are issued in the defined order; (f) $\mathsf{ob}; [\mathtt{Inst}]$: in general, an event is observed no earlier than it is issued, and for an instantaneous event, the two points coincide. Thus $e \xrightarrow{\mathsf{ob}} e'$ implies $e \xrightarrow{\mathsf{ib}} e'$ when e' is instantaneous.

On the other hand, for ob we have (a) $\mathsf{rf}_{\overline{\mathsf{b}}}$: if $w \xrightarrow{\mathsf{rf}_{\overline{\mathsf{b}}}} r$ then w was committed to memory before r, since r cannot read from the store buffer of another thread; (b) $[\mathtt{nlW}]; \mathsf{pf}$: NIC local writes cannot be polled until they are committed to memory; (c) nfo: NIC events are observed in the same order they arrive in $\mathbf{wb_L}/\mathbf{wb_R}$; (d) rb: if $r \xrightarrow{\mathsf{rb}} w$, then w was not observed before r, otherwise it would have been committed to memory before r; (e) mo: if $w \xrightarrow{\mathsf{mo}} w'$, then w was observed in memory before w'; (f) $\mathsf{ar}; \mathsf{rao}$: if $w \xrightarrow{\mathsf{ar}} r \xrightarrow{\mathsf{rao}} r'$, then we have that r and w are the read and write of the same rRMW operation, thus w must be observed before the rao-later r' to ensure atomicity. (g) $[\mathtt{Inst}]; \mathsf{ib}$: by a similar logic to above, we know that the ib-earlier instantaneous event is also observed earlier, since its issue and observation points coincide.

5.2 Semantics of a Program

Given a program P, we can generate an event graph (E, po), by a standard process, which we describe below. We then choose any $\mathsf{rf}, \mathsf{mo}, \mathsf{pf}, \mathsf{nfo}, \mathsf{rao}$ such that the execution is consistent. The semantics of P are the set of consistent executions of P.

Thread to Event Graph. Given a thread identifier $t \in \mathsf{Tid}$ and a sequence of labels $l_1, \ldots, l_n \in \mathsf{ELab}$, we define the *event graphs of* t as $(\{e_1, \ldots, e_n\}, \mathsf{po}) \in G^t(l_1, \ldots, l_n)$ where: (a) $l(e_i) = l_i$ for all $1 \leq i \leq n$; (b) $\iota(e_i) \neq \iota(e_j)$ for all $1 \leq i < j \leq n$; (c) $t(e_i) = t$ for all $1 \leq i \leq n$; (d) $\mathsf{po} = \{(e_i, e_j) \mid 1 \leq i < j \leq n\}$.

Initial Event Graph. Given a set of locations Loc, we define $G_{init} = (E_0, \emptyset)$, such that for each $x \in \mathsf{Loc}$ there is exactly one $e \in E_0$ with $l(e) = \mathtt{lW}(x, 0)$, and every event in E_0 has a unique identifier. We call E_0 the set of *initialisation events*.

Sequential Composition. For two event graphs G_1 and G_2, we define their *sequential* composition $G_1; G_2 = (E, \mathsf{po})$ where

$$E \triangleq G_1.E \uplus G_2.E$$
$$\mathsf{po} \triangleq G_1.\mathsf{po} \cup G_2.\mathsf{po} \cup (G_1.E \times G_2.E)$$

Note that all events in G_2 are ordered po-after every event in G_1. Sequential composition is defined only where the set of events of each graph are disjoint, i.e. $G_1.E \cap G_2.E = \emptyset$.

Parallel Composition. We define *parallel* composition by $G_1 \parallel G_2 = (E, \mathsf{po})$ where

$$E \triangleq G_1.E \uplus G_2.E$$
$$\mathsf{po} \triangleq G_1.\mathsf{po} \cup G_2.\mathsf{po}$$

Note that the events of each graph are not po-ordered with respect to one another. We also require that the event sets be disjoint. As this operation is commutative and associative, it is straightforward to lift it to sets of graphs, which we denote by $\parallel \mathcal{G}$, where $\mathcal{G}$ is a set of event graphs.

Program to Event Graph. A program P *generates* G if $G = G_{init}; (\|_{t \in \mathsf{Tid}}\ G_t)$ and there is a set of sequences $s_t \in S$ such that $\mathsf{P}(t) \rightarrowtail s_t$ and $G_t \in G^t(s_t)$ for all $t \in \mathsf{Tid}$.

The operation $C \rightarrowtail s$ relates a sequential program C to a sequence of labels s it generates. The definition is standard and show in Fig. 5.3. Note that RDMA operations generate multiple events, and for local and remote CAS operations, we distinguish between success and failure cases.

Theorem 1. The operational and declarative semantics of RDMA^{TSO} are equivalent.

Proof. See §6 □

$$\frac{C \rightsquigarrow C' \qquad C' \rightarrowtail s}{C \rightarrowtail s} \qquad \frac{C_1 \rightarrowtail s_1 \qquad C_2 \rightarrowtail s_2}{C_1; C_2 \rightarrowtail s_1, s_2} \qquad \frac{\mathsf{elocs}(e) = \emptyset}{x := e \rightarrowtail \mathtt{lW}(x, [\![e]\!])}$$

$$\frac{\begin{array}{c}\mathsf{elocs}(e_{\text{old}}) = \mathsf{elocs}(e_{\text{new}}) = \emptyset \\ z := [\![e_{\text{old}}]\!] \rightarrowtail s\end{array}}{z := \mathtt{CAS}(x, e_{\text{old}}, e_{\text{new}}) \rightarrowtail \mathtt{CAS}(x, [\![e_{\text{old}}]\!], [\![e_{\text{new}}]\!]), s}$$

$$\frac{\begin{array}{c}\mathsf{elocs}(e_{\text{old}}) = \mathsf{elocs}(e_{\text{new}}) = \emptyset \\ v \neq [\![e_{\text{old}}]\!] \qquad z := v \rightarrowtail s\end{array}}{z := \mathtt{CAS}(x, e_{\text{old}}, e_{\text{new}}) \rightarrowtail \mathtt{F}, \mathtt{lR}(x, v), s} \qquad \frac{}{\mathtt{mfence} \rightarrowtail \mathtt{F}} \qquad \frac{}{\mathtt{assume}(x = v) \rightarrowtail \mathtt{lR}(x, v)}$$

$$\frac{v' \neq v}{\mathtt{assume}(x \neq v) \rightarrowtail \mathtt{lR}(x, v')} \qquad \frac{}{x := y^{\overline{n}} \rightarrowtail \mathtt{nrR}(y^{\overline{n}}, v), \mathtt{nlW}(x, v, \overline{n})}$$

$$\frac{}{y^{\overline{n}} := x \rightarrowtail \mathtt{nlR}(x, v, \overline{n}), \mathtt{nrW}(y^{\overline{n}}, v)} \qquad \frac{}{\mathtt{rfence}\ (\overline{n}) \rightarrowtail \mathtt{nF}(\overline{n})} \qquad \frac{}{\mathtt{poll}(\overline{n}) \rightarrowtail \mathtt{P}(\overline{n})}$$

$$\frac{}{\mathtt{skip} \rightarrowtail \epsilon} \qquad \frac{\mathsf{elocs}(e_{old}) = \mathsf{elocs}(e_{new}) = \emptyset \qquad v \neq [\![e_{old}]\!]}{z := \mathtt{nCAS}(x^{\overline{n}}, e_{old}, e_{new}) \rightarrowtail \mathtt{narR}(x^{\overline{n}}, v), \mathtt{nlW}(z, v, \overline{n})}$$

$$\frac{\mathsf{elocs}(e_{old}) = \mathsf{elocs}(e_{new}) = \emptyset}{z := \mathtt{nCAS}(x^{\overline{n}}, e_{old}, e_{new}) \rightarrowtail \mathtt{narR}(x^{\overline{n}}, [\![e_{old}]\!]), \mathtt{narW}(x^{\overline{n}}, [\![e_{new}]\!]), \mathtt{nlW}(z, [\![e_{old}]\!], \overline{n})}$$

$$\frac{\mathsf{elocs}(e) = \emptyset \qquad v' = v + [\![e]\!]}{z := \mathtt{nFAA}(x^{\overline{n}}, e) \rightarrowtail \mathtt{narR}(x^{\overline{n}}, v), \mathtt{narW}(x^{\overline{n}}, v'), \mathtt{nlW}(z, v, \overline{n})}$$

Figure 5.3: Label Sequences Construction

6 Equivalence

The proof of Theorem 1 can be found in Appendix B, which directly adapts the proof given in [10]. However, this proof is necessarily verbose, so this section provides an overview of the structure, highlights, and notable additions to the proof.

6.1 Structure of the Proof

The goal of the proof is to show that the operational and declarative models, given in §4 and §5 respectively, are equivalent.

By 'equivalent', we mean that given a program P, the set of executions of P given by either semantics each describe the same program behaviours, such that each execution derived by the operational semantics has a corresponding execution graph given by the declarative semantics, and vice-versa.

Therefore, we aim to show that (a) given an execution graph G generated by a program P, there is an equivalent derivation D of an execution of P; and (b) given a derivation D of an execution of P, there is an equivalent execution graph G generated by P.

We note that a derivation D of an execution takes the form

$$\mathsf{P}, \mathsf{M}_0, \mathsf{B}_0, \mathsf{A}_0, \mathsf{QP}_0 \Rightarrow^* \lambda t.\mathtt{skip}, \mathsf{M}, \mathsf{B}_0, \mathsf{A}_0, \lambda t.\lambda \overline{n}.\langle \epsilon, \epsilon, \mathtt{cn}^* \rangle$$

where $\Rightarrow^*$ is the reflexive transitive closure of $\Rightarrow$ defined in Fig. 4.2.

Note that M_0 and similar denote initial states. A derivation must finish with the empty program (all threads contain only `skip`), all store buffers empty, all remote atomic locks released, and queue pairs containing no pending operations except for confirmation notifications, as it is allowed for these not to be polled.

However, it will be very challenging to convert between operational and declarative semantics directly, because the prior is concerned primarily with states while the latter only represents events. For the proof in the direction from operational to declarative (resp. declarative to operational), we would first need to derive the events that are implicit in the transitions between states (resp. the states implied by the events), then remove information about states (resp. events) that is no longer relevant.

In other words, for the proof in either direction, we first need to build a view that contains all the information of both semantics, then remove information to arrive at the appropriate form. There would be no reason not to use the same intermediary form for both directions. Therefore, we define an *annotated* semantics, as a direct extension of the operational semantics, where states are extended with event labels.

We then aim to prove that 1. the operational semantics is equivalent to the annotated semantics and 2. the declarative semantics is equivalent to the annotated semantics. Naturally, each point will require a proof in either direction.

Ensuring that the annotated semantics is defined as a direct extension of the operational one will make the proof of the two directions for the first point straightforward, while the latter point will be more complex.

6.2 Annotated Semantics

The annotated semantics is defined in §B.1. The states of the operational semantics are extended to contain events, rather than values. For example, a memory location

will contain a *particular* write event, rather than a mere value. This will help us to associate related events, in order to build the relations of the declarative semantics. The set of labels used is an extension of those defined for the declarative semantics, with the introduction of labels representing the operations put, get, nCAS and nFAA, and RDMA acknowledgement messages, as these are not explicitly modelled by labels in an execution graph, only by relations.

We then define annotated labels, which will be the labels for the transitions of the annotated semantics; each comprises a label type and tuple of events. The annotated labels explicitly connect transitions that concern the same event. Recall that for RDMA operations in the operational semantics, we expect to see a number of related transitions as it is processed into, through, and out of a queue-pair. In the annotated semantics, we directly connect those transitions by generating an event in the relevant program transition, then including that event in the label of each related hardware and queue-pair transition.

The transitions of the annotated semantics correspond *directly* to those of the operational semantics, but with events instead of values checked in most places. These transitions have been suitably extended to match the corresponding extensions to the operational semantics.

Paths. We extend the annotated semantics with a path π, which is a sequence of annotated labels. This serves as a history of transitions taken, as each time a labelled transition occurs, the label is added to π. This means that, given a particular state of the operational semantics, we are able to use π to determine every transition taken up to that point. We maintain a number of invariants to ensure that a path is well-formed, and to identify if it corresponds to a complete execution. A number of these invariants need to be updated due to the introduction of rRMWs, and we also introduce a new invariant to model the atomicity guarantee.

Path Conditions. A path is *complete* if there are no pending operations. We can check if an operation is complete by looking for all the relevant labels in the path. For example, if there was a put operation in the program, there should be labels in the path corresponding to the operation being pushed to the store-buffer, pushed to the queue-pair, a NIC local read, NIC remote write, acknowledgement, and committing the NIC remote write to memory.

A path is *backwards-complete* if labels do not appear out of thin air. For example, if there is a NIC remote write, there must be a corresponding prior NIC local read, since a NIC remote write can only appear as part of a put operation. Likewise there must be an earlier label pushing the operation to the queue pair, and so on.

A path is *well-formed* if it is backwards-complete and observes a number of ordering rules.

- Poll order: for any two writes on the same queue-pair, if the later write is polled, then the earlier write must be polled first;
- Flush order: any pair of writes that go via the same buffer (i.e. store buffer or queue-pair write buffer) must be committed to memory in the order they were buffered;
- NIC order: for a pair of operations which are pushed to the same queue-pair, describes the minimum progress achieved by the first operation, given the progress of the second operation;

- NIC atomicity: an earlier successful rRMW must have had its remote write committed to remote memory before a later rRMW reads;

- Read order: a read should observe the latest write that is visible to it, i.e. for NIC reads, the latest write in memory, and for CPU reads, the latest write in the store buffer, if there is one, or else the latest write in memory.

We then define well-formedness for M, B, A, and QP with respect to π, which requires that events in π are reflected in those states. For example, the write event at a location in M should match the latest write event to that location which has been flushed in π.

We then observe the result that well-formedness is preserved by the transitions of the annotated semantics.

6.3 Annotated Semantics to Declarative Semantics

For this part of the proof, we show that a well-formed and complete path π yields a consistent execution. We define $\mathsf{getEG}(\pi)$ which finds E, po, rf, pf, mo, nfo and rao. This is straightforward, as by construction π provides an explicit total order for every event.

We then prove that $\mathsf{getEG}(\pi)$ is a pre-execution and meets the conditions for well-formedness and consistency. Showing that it is a well-formed pre-execution is straightforward, while consistency is more interesting. The conceit of the proof is to show that each component of ib and ob is a subset of the total orders on issue and observation points given by π. The notable new cases are due to the extensions to ippo and oppo, and the new component of ob, ar; rao. The proofs for the new cases of ippo and oppo rely heavily on the extended definition of NIC order, the cases of which broadly correspond to cells in the ippo/oppo tables. In the case of ar; rao, the new invariant regarding NIC atomicity is used.

6.4 Declarative Semantics to Annotated Semantics

Now, we need to do the converse: given a consistent execution $G = \langle E, \mathsf{po}, \mathsf{rf}, \mathsf{pf}, \mathsf{mo}, \mathsf{nfo}, \mathsf{rao} \rangle$, we aim to construct a well-formed path π. Using π, it is then straightforward to reconstruct the rest of the annotated semantics derivation.

To build π, we extend ib and ob into total orders. We update this process to accommodate rRMWs, which requires some additional work not present in the prior version of the proof. The event graph does not contain information about failed nCAS operations, nor does it distinguish between a successful nCAS and an nFAA. Therefore, we need to look at the derivation from P to G, as described in §5.2. By checking the derivation, we can retrieve the information that was lost during the creation of the event graph.

We then need to show that each path invariant holds. The most interesting case is the proof of NIC atomicity. We argue that when a successful rRMW with read r_1 and write w_1 appears in the path, there is no other rRMW with read r_1 towards the same node which reads between r_1 and w_1. Such a read would force $r_1 \xrightarrow{\mathsf{rao}} r_2$, and thus $w_1 \xrightarrow{\mathsf{ar};\mathsf{rao}} r_2$, which make ib and ob reflexive. This is not allowed by the consistency conditions.

6.5 Operational and Annotated Semantics

The annotated semantics was deliberately designed as a direct extension of the operational semantics in order to make this side of the proof much easier. The rules extending a derivation of the operational semantics to one of the annotated semantics are straightforward, as is the respective process for restricting an annotated semantics derivation to one of the operational semantics. We extend these definitions with rules in the case of rRMWs, which leads to the result we require.

7 Discussion

7.1 Conclusion

This project gives the first intuitive and formal descriptions of the behaviour of RDMA remote RMWs, outside of a hardware manual. We extend the RDMA^{TSO} memory model to provide operational and declarative semantics, enabling formal reasoning and verification of programs reliant on these operations. By formalising remote RMW behaviours, we have clarified ambiguities in the hardware specification, enabling programmers to understand them in simple terms. In the future, we hope this will lead to more reliable RDMA program and libraries, and potentially to the development of automated verification tools for RDMA.

We are confident that the models given in this report align with intended behaviour for rRMWs given by the InfiniBand technical specification [21]. The behaviours given by these models have been checked against the specification, and confirmed by an expert from NVIDIA, a leading vendor of RDMA hardware.

7.2 Limitations

Although we have sought to confirm the accuracy of our model, we have not conducted independent validation on RDMA hardware. Only an extensive testing campaign would reveal whether real networks implement the specification faithfully. As is the case with all memory models, there may be some behaviours which are permitted but not observed. It was noted in [10] that store buffering with puts was not observed when the earlier put was polled. NVIDIA have confirmed that their current hardware implements a stronger guarantee for polled NIC remote writes, which means the same behaviour is also not possible when rRMWs are used. Therefore, the examples shown in Fig. 4.6 are not possible under current NVIDIA hardware, but this may change in future implementations. In general, we cannot be sure the weak behaviours we have described are possible on current hardware, unless we observe it through empirical validation.

Our work is also limited to RDMA implementations where individual machines adhere to x86-TSO. While x86 is the most common architecture used for RDMA, there are other possibilities; recent research has shown the viability of ARM processors for high-performance computing workloads [23, 24, 25]. This could lead to wider use of RDMA on ARM, which does not observe TSO.

7.3 Future Work

Conformance Testing. The models presented here could be used to develop conformance tests, to confirm that implementation of RDMA faithfully adhere to the specification. One approach to this would be to follow the method shown in [26], by encoding the declarative model in Alloy, a tool for bounded model checking. This is then used to find program executions which are on the boundary of being allowed or disallowed. Appendix C contains a prototype encoding of the declarative semantics, and an example encoding of a litmus test. The next step would be to extend this with a framework for encoding litmus tests, as the process is too tedious and verbose to be done manually.

It is equally possible to encode the operational semantics using other bounded model checkers, such as CMBC [27].

Other Architectures. It would be interesting to investigate the semantics of RDMA in the presence of other architectures such as ARMv8. As the semantics of RDMA operations is independent from the semantics of local operations, it may be possible to develop a memory model for RDMA which is *parametric* in the semantics of the underlying architecture. This would make the semantics durable with respect to changes in the hardware with which RDMA is used.

Verifying Programs and Libraries. The main utility of these models is to verify RDMA libraries and programs. This may be performed manually using the event graph construction we have described, but developing a tool to automate this would enable faster and more extensive verification. One approach is to extend an existing verification tool such as Memalloy [28], which is extensible to arbitrary memory models, such as RDMA^{TSO}.

References

[1] M. Su et al. RFP: A remote fetching paradigm for RDMA-accelerated systems. *CoRR*, abs/1512.07805, 2015. arXiv: 1512.07805. URL: http://arxiv.org/abs/1512.07805.

[2] N. Boscia and H. S. Sidhu. Comparison of 40G RDMA and Traditional Ethernet Technologies. Technical report NAS-2014-01, NASA Advanced Supercomputing Division, NASA Ames Research Center, Moffett Field, CA, 2014. URL: https://www.nas.nasa.gov/assets/nas/pdf/papers/NAS_Technical_Report_NAS-2014-01.pdf.

[3] Introduction to InfiniBand. White Paper 2003WP, Mellanox Technologies Inc., 2003. URL: https://network.nvidia.com/pdf/whitepapers/IB_Intro_WP_190.pdf.

[4] Annex A16: RDMA over Converged Ethernet (RoCE). Technical report, InfiniBand Trade Association, Beaverton, OR, 2010. Available upon request.

[5] R. Gerstenberger et al. Enabling highly scalable remote memory access programming with MPI-3 one sided. *Commun. ACM*, 61(10):106–113, 2018. DOI: 10.1145/3264413.

[6] A. Shpiner et al. RoCE rocks without PFC: detailed evaluation. In M. Alizadeh and Y. Zhu, editors, *Proceedings of the Workshop on Kernel-Bypass Networks, KBNets@SIGCOMM 2017, Los Angeles, CA, USA, August 21, 2017*, pages 25–30. ACM, 2017. DOI: 10.1145/3098583.3098588.

[7] S. Owens et al. A better x86 memory model: x86-tso. In S. Berghofer et al., editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer, 2009. DOI: 10.1007/978-3-642-03359-9_27.

[8] C. Pulte et al. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.*, 2(POPL):19:1–19:29, 2018. DOI: 10.1145/3158107.

[9] A. M. Dan et al. Modeling and analysis of remote memory access programming. In E. Visser and Y. Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 129–144. ACM, 2016. DOI: 10.1145/2983990.2984033.

[10] G. Ambal et al. Semantics of remote direct memory access: operational and declarative models of RDMA on TSO architectures. *Proc. ACM Program. Lang.*, 8(OOPSLA2):1982–2009, 2024. DOI: 10.1145/3689781.

[11] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. DOI: 10.1145/114005.102808.

[12] A. McMenamin. The end of dennard scaling, 2013. URL: https://cartesianproduct.wordpress.com/2013/04/15/the-end-of-dennard-scaling/.

[13] J. Alglave et al. Frightening small children and disconcerting grown-ups: concurrency in the linux kernel. In X. Shen et al., editors, *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 405–418. ACM, 2018. DOI: 10.1145/3173162.3177156.

[14] Heisenbug - Wikipedia. URL: https://en.wikipedia.org/wiki/Heisenbug.

[15] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012. ISBN: 9780123973375.

[16] P. Sewell et al. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010. DOI: 10.1145/1785414.1785443.

[17] SPARC International, editor. *The SPARC architecture manual: version 8*. Prentice Hall, Englewood Cliffs, N.J, 1992. ISBN: 978-0-13-825001-0.

[18] A. Godbole et al. Automated conversion of axiomatic to operational models: theory and practice. *CoRR*, abs/2208.06733, 2022. DOI: 10.48550/ARXIV.2208.06733. arXiv: 2208.06733.

[19] G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.*, 60-61:17–139, 2004.

[20] PCI-SIG. PCI Express Base Specification Revision 6.0 Version 1.0. Technical report, PCI-SIG, 2022. URL: https://pcisig.com/pci-express-6.0-specification.

[21] *InfiniBand Architecture Specification Volume 1, Release 1.2.1*. Available upon request. InfiniBand Trade Association. Beaverton, OR, 2007.

[22] J. Preshing. Atomic vs. Non-Atomic Operations, 2013-06-18. URL: https://preshing.com/20130618/atomic-vs-non-atomic-operations/.

[23] F. Mantovani et al. Performance and energy consumption of HPC workloads on a cluster based on arm thunderx2 CPU. *CoRR*, abs/2007.04868, 2020. arXiv: 2007.04868. URL: https://arxiv.org/abs/2007.04868.

[24] N. A. Simakov et al. Are we ready for broader adoption of ARM in the HPC community: performance and energy efficiency analysis of benchmarks and applications executed on high-end ARM systems. In *Proceedings of the HPC Asia 2023 Workshops, HPC Asia 2023, Singapore, 27 February 2023 - 2 March 2023*, pages 78–86. ACM, 2023. DOI: 10.1145/3581576.3581618.

[25] A. Jackson et al. Evaluating the ARM ecosystem for high performance computing. *CoRR*, abs/1904.04250, 2019. arXiv: 1904.04250. URL: http://arxiv.org/abs/1904.04250.

[26] V. Klimis et al. Taking back control in an intermediate representation for GPU computing. *Proc. ACM Program. Lang.*, 7(POPL), 2023-01. DOI: 10.1145/3571253.

[27] D. Iorga et al. Simulating operational memory models using off-the-shelf program analysis tools. *IEEE Trans. Software Eng.*, 49(12):5084–5102, 2023. DOI: 10.1109/TSE.2023.3326056.

[28] J. Wickerson et al. Automatically comparing memory consistency models. In G. Castagna and A. D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 190–204. ACM, 2017. DOI: 10.1145/3009837.3009838.

A Declarations

A.1 Use of Generative AI

No generative AI models have been used in the production of this report.

A.2 Ethical Considerations

As with all research into the precise behaviour of computer systems, it is possible, however unlikely, that some behaviour identified may be abused by bad actors to achieve nefarious goals, such as unauthorised access to a system. However, RDMA is typically used on private network executing trusted code, so this is very unlikely in our case. As a primarily theoretical project, there are no other notable ethical considerations.

A.3 Sustainability

No resource intensive tasks were involved in this project. The computer used for the research and writing of this report was a Macbook with an energy efficient Apple M1 chip.

A.4 Availability of Materials

All materials are available within the text of this report, in particular the theoretical models are described in §4 and §5, and the Alloy encoding of the declarative model is provided in Appendix C.

B Annotated Semantics

B.1 Annotated Labels and Inference Rules

On top of the 12 labels presented in §5, we create six new labels: $\texttt{Put}(\overline{y}, x)$, $\texttt{Get}(x, \overline{y})$, $\texttt{nCAS}(z, \overline{x}, v, u)$, $\texttt{nFAA}(z, \overline{x}, u)$, $\texttt{nlEX}(\overline{n})$, and $\texttt{nrEX}(\overline{n})$. These labels can also be used to create events (when bundled with an event identifier and a thread identifier).

We note E^{ext} the extended set of all events, including the six new labels.

Recall that $\mathcal{R} = \texttt{lR} \cup \texttt{CAS} \cup \texttt{nlR} \cup \texttt{nrR} \cup \texttt{narR} \subseteq E^{\text{ext}}$ and $\mathcal{W} = \texttt{lW} \cup \texttt{CAS} \cup \texttt{nlW} \cup \texttt{nrW} \cup \texttt{narW} \subseteq E^{\text{ext}}$. We also note $\texttt{nEX} = \texttt{nlEX} \cup \texttt{nrEX}$ and $\texttt{nRMW} = \texttt{nCAS} \cup \texttt{nFAA}$.

For annotated labels, we reuse most names from labels, but they are different entities. For instance we note $r \in \texttt{lR}$ for an event with label lR, while $\lambda = \texttt{lR}\langle \ldots \rangle$ is an annotated label.

We use $\mathsf{type}(\lambda)$ to denote the type of the annotated label (lR, lW, CAS, F, Push, NIC, nlR, nrR, nlW, nrW, CN, P, nF, B, $\mathcal{E}$). We use $r(\lambda), w(\lambda), u(\lambda), a(\lambda), f(\lambda), p(\lambda), e(\lambda), \ldots$ to access the elements of a $\lambda \in \mathsf{ALabel}$ where applicable. Also, we note $t(\lambda)$ for the thread of the first argument of λ.

The annotated program transitions (Fig. B.2) use an additional annotated label $\texttt{CASF}\langle r, w \rangle$ with $r \in \texttt{lR}$ and $w \in \mathcal{W}$ to represent a failed CAS operation. This case is then translated into two labels (a memory fence and a local read) when creating a path in §B.2. Also, note that the annotated domains (e.g. the store buffers and the queue pairs) contain events, not annotated labels.

initialisation Given a program P, let

$\mathsf{M}_0 \in \mathsf{AMem}$	s.t. $\forall x \in \mathsf{Loc}.\ \mathsf{M}_0(x) = init_x$ with $l(init_x) \triangleq \texttt{lW}(x, 0)$
$\mathsf{b}_0 \in \mathsf{ASBuff}$	$\mathsf{b}_0 \triangleq \varepsilon$
$\mathsf{B}_0 \in \mathsf{ASBMap}$	$\mathsf{B}_0 \triangleq \lambda t.\mathsf{b}_0$
$\mathsf{A}_0 \in \mathsf{RAMap}$	$\mathsf{A}_0 \triangleq \lambda t.\bot$
$\mathsf{qp}_0 \in \mathsf{AQPair}$	$\mathsf{qp}_0 \triangleq \langle \varepsilon, \varepsilon, \varepsilon \rangle$
$\mathsf{QP}_0 \in \mathsf{AQPMap}$	$\mathsf{QP}_0 \triangleq \lambda t.\lambda n.\mathsf{qp}_0$

$\lambda \in \mathsf{ALabel}$

$$
\begin{array}{rll}
\lambda \triangleq & \mid \mathtt{lR}\langle r, w\rangle & \text{where } r \in \mathtt{lR}, w \in \mathcal{W}, \mathsf{eq}_{\mathtt{loc\&v}}(r, w) \\
& \mid \mathtt{lW}\langle w\rangle & \text{where } w \in \mathtt{lW} \\
& \mid \mathtt{CAS}\langle u, w\rangle & \text{where } u \in \mathtt{CAS}, w \in \mathcal{W}, \mathsf{eq}_{\mathtt{loc\&v}}(u, w) \\
& \mid \mathtt{F}\langle f\rangle & \text{where } f \in \mathtt{F} \\
& \mid \mathtt{Push}\langle a\rangle & \text{where } a \in (\mathtt{Put} \cup \mathtt{Get} \cup \mathtt{nCAS} \cup \mathtt{nFAA} \cup \mathtt{nF}) \\
& \mid \mathtt{NIC}\langle a\rangle & \text{where } a \in (\mathtt{Put} \cup \mathtt{Get} \cup \mathtt{nCAS} \cup \mathtt{nFAA} \cup \mathtt{nF}) \\
& \mid \mathtt{nlR}\langle r, w, a, w'\rangle & \text{where } r \in \mathtt{nlR}, w \in \mathcal{W}, a \in \mathtt{Put}, w' \in \mathtt{nrW}, \mathsf{eq}_{\mathtt{loc\&v}}(r, w), \\
& & \quad \mathtt{loc}_r(a) = \mathtt{loc}(r), \mathtt{loc}_w(a) = \mathtt{loc}(w'), v_{\mathrm{r}}(r) = v_{\mathrm{w}}(w') \\
& \mid \mathtt{nrR}\langle r, w, a, w'\rangle & \text{where } r \in \mathtt{nrR}, w \in \mathcal{W}, a \in \mathtt{Get}, w' \in \mathtt{nlW}, \mathsf{eq}_{\mathtt{loc\&v}}(r, w), \\
& & \quad \mathtt{loc}_r(a) = \mathtt{loc}(r), \mathtt{loc}_w(a) = \mathtt{loc}(w'), v_{\mathrm{r}}(r) = v_{\mathrm{w}}(w') \\
& \mid \mathtt{narR}\langle r, w, a, w', w''\rangle & \text{where } r \in \mathtt{narR}, w \in \mathcal{W}, a \in \mathtt{nRMW}, w' \in \mathtt{nlW}, w'' \in \mathtt{narW}, \\
& & \quad \mathsf{eq}_{\mathtt{loc\&v}}(r, w), \mathtt{loc}_r(a) = \mathtt{loc}(r) = \mathtt{loc}(w''), \\
& & \quad \mathtt{loc}_w(a) = \mathtt{loc}(w'), v_{\mathrm{r}}(r) = v_{\mathrm{w}}(w'), \\
& & \quad a \in \mathtt{nCAS} \implies v_{\mathrm{r}}(r) = v_{\mathrm{e}}(a) \wedge v_{\mathrm{w}}(w'') = v_{\mathrm{u}}(a) \\
& & \quad a \in \mathtt{nFAA} \implies v_{\mathrm{w}}(w'') = v_{\mathrm{r}}(r) + v(a) \\
& \mid \mathtt{naF}\langle r, w, a, w'\rangle & \text{where } r \in \mathtt{narR}, w \in \mathcal{W}, a \in \mathtt{nRMW}, w' \in \mathtt{nlW}, \mathsf{eq}_{\mathtt{loc\&v}}(r, w), \\
& & \quad \mathtt{loc}_r(a) = \mathtt{loc}(r), \mathtt{loc}_w(a) = \mathtt{loc}(w'), \\
& & \quad v_{\mathrm{r}}(r) = v_{\mathrm{w}}(w^l), v_{\mathrm{r}}(r) \neq v_{\mathrm{e}}(a) \\
& \mid \mathtt{nlW}\langle w, e\rangle & \text{where } w \in \mathtt{nlW}, e \in \mathtt{nlEX}, \mathsf{sameqp}(w, e) \\
& \mid \mathtt{nrW}\langle w, e\rangle & \text{where } w \in \mathtt{nrW}, e \in \mathtt{nrEX}, \mathsf{sameqp}(w, e) \\
& \mid \mathtt{narW}\langle w\rangle & \text{where } w \in \mathtt{narW} \\
& \mid \mathsf{CN}\langle e\rangle & \text{where } e \in \mathtt{nrEX} \\
& \mid \mathtt{P}\langle p, e\rangle & \text{where } p \in \mathtt{P}, e \in \mathtt{nEX}, \mathsf{sameqp}(p, e) \\
& \mid \mathtt{nF}\langle f\rangle & \text{where } f \in \mathtt{nF} \\
& \mid \mathsf{B}\langle w\rangle & \text{where } w \in \mathcal{W} \\
& \mid \mathcal{E}\langle t\rangle & \text{where } t \in \mathsf{Tid}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{eq}_{\mathtt{loc\&v}}(r, w) & \triangleq & \mathtt{loc}(r) = \mathtt{loc}(w) \wedge v_{\mathrm{r}}(r) = v_{\mathrm{w}}(w) \\
\mathsf{sameqp}(e, e') & \triangleq & t(e) = t(e') \wedge \overline{n}(e) = \overline{n}(e')
\end{array}
$$

Figure B.1: Annotated Labels

Program transitions: $\mathsf{Prog} \xrightarrow{\mathsf{ALabel} \uplus \{\mathtt{CASF}\}} \mathsf{Prog}$
Command transitions: $\mathsf{Comm} \xrightarrow{\mathsf{ALabel} \uplus \{\mathtt{CASF}\}} \mathsf{Comm}$

$$\frac{\mathsf{C}_1 \xrightarrow{\lambda} \mathsf{C}_1'}{\mathsf{C}_1; \mathsf{C}_2 \xrightarrow{\lambda} \mathsf{C}_1'; \mathsf{C}_2} \qquad \frac{}{\mathtt{skip}; \mathsf{C} \xrightarrow{\mathcal{E}\langle t \rangle} \mathsf{C}} \qquad \frac{i \in \{1,2\}}{\mathsf{C}_1 + \mathsf{C}_2 \xrightarrow{\mathcal{E}\langle t \rangle} \mathsf{C}_i} \qquad \frac{}{\mathsf{C}^* \xrightarrow{\mathcal{E}\langle t \rangle} \mathtt{skip}}$$

$$\frac{}{\mathsf{C}^* \xrightarrow{\mathcal{E}\langle t \rangle} \mathsf{C}; \mathsf{C}^*} \qquad \frac{\mathsf{C} \rightsquigarrow \mathsf{C}'}{\mathsf{C} \xrightarrow{\mathcal{E}\langle t \rangle} \mathsf{C}'} \qquad \frac{\mathsf{elocs}(e) = \emptyset \qquad w = (\iota, t, \mathtt{lW}(x, [\![e]\!]))}{x := e \xrightarrow{\mathtt{lW}\langle w \rangle} \mathtt{skip}}$$

$$\frac{\mathsf{elocs}(e_{\mathrm{old}}) = \mathsf{elocs}(e_{\mathrm{new}}) = \emptyset \qquad v \neq [\![e_{\mathrm{old}}]\!] \qquad r = (\iota, t, \mathtt{lR}(x, v))}{z := \mathtt{CAS}(x, e_{\mathrm{old}}, e_{\mathrm{new}}) \xrightarrow{\mathtt{CASF}\langle r, w \rangle} z := v}$$

$$\frac{\mathsf{elocs}(e_{\mathrm{old}}) = \mathsf{elocs}(e_{\mathrm{new}}) = \emptyset \qquad u = (\iota, t, \mathtt{CAS}(x, [\![e_{\mathrm{old}}]\!], [\![e_{\mathrm{new}}]\!]))}{z := \mathtt{CAS}(x, e_{\mathrm{old}}, e_{\mathrm{new}}) \xrightarrow{\mathtt{CAS}\langle u, w \rangle} z := [\![e_{\mathrm{old}}]\!]} \qquad \frac{f = (\iota, t, \mathtt{F})}{\mathtt{mfence} \xrightarrow{\mathtt{F}\langle f \rangle} \mathtt{skip}}$$

$$\frac{a = (\iota, t, \mathtt{Get}(x, \overline{y}))}{x := \overline{y} \xrightarrow{\mathtt{Push}\langle a \rangle} \mathtt{skip}} \qquad \frac{a = (\iota, t, \mathtt{Put}(\overline{y}, x))}{\overline{y} := x \xrightarrow{\mathtt{Push}\langle a \rangle} \mathtt{skip}} \qquad \frac{a = (\iota, t, \mathtt{nF}(\overline{n}))}{\mathtt{rfence}\ \overline{n} \xrightarrow{\mathtt{Push}\langle a \rangle} \mathtt{skip}}$$

$$\frac{\mathsf{elocs}(e_{\mathrm{old}}) = \mathsf{elocs}(e_{\mathrm{new}}) = \emptyset \qquad v = [\![e_{\mathrm{old}}]\!] \qquad u = [\![e_{\mathrm{new}}]\!] \qquad a = (\iota, t, \mathtt{nCAS}(z, \overline{x}, v, u))}{z := \mathtt{nCAS}(x, e_{\mathrm{old}}, e_{\mathrm{new}}) \xrightarrow{\mathtt{Push}\langle a \rangle} \mathtt{skip}}$$

$$\frac{\mathsf{elocs}(e) = \emptyset \qquad u = [\![e]\!] \qquad a = (\iota, t, \mathtt{nFAA}(z, \overline{x}, u))}{z := \mathtt{nFAA}(x, e) \xrightarrow{\mathtt{Push}\langle a \rangle} \mathtt{skip}} \qquad \frac{p = (\iota, t, \mathtt{P}(n))}{\mathtt{poll}(n) \xrightarrow{\mathtt{P}\langle p, e \rangle} \mathtt{skip}}$$

$$\frac{r = (\iota, t, \mathtt{lR}(x, v))}{\mathtt{assume}(x = v) \xrightarrow{\mathtt{lR}\langle r, w \rangle} \mathtt{skip}} \qquad \frac{v \neq v' \qquad r = (\iota, t, \mathtt{lR}(x, v'))}{\mathtt{assume}(x \neq v) \xrightarrow{\mathtt{lR}\langle r, w \rangle} \mathtt{skip}} \qquad \frac{\mathsf{P}(t(\lambda)) \xrightarrow{\lambda} \mathsf{C}}{\mathsf{P} \xrightarrow{\lambda} \mathsf{P}[t(\lambda) \mapsto \mathsf{C}]}$$

Figure B.2: RDMA$^{\mathrm{TSO}}$ program and command transitions for the annotated semantics

$$\mathsf{M} \in \mathsf{AMem} \triangleq \{m \in \mathsf{Loc} \to \mathcal{W} \mid \forall x \in \mathsf{Loc}.\mathtt{loc}(m[x]) = x\} \qquad \mathsf{B} \in \mathsf{ASBMap} \triangleq \mathsf{Tid} \to \mathsf{ASBuff}$$
$$\mathsf{A} \in \mathsf{RAMap} \triangleq \lambda n.\,\{\bot, \top\} \qquad \mathsf{QP} \in \mathsf{AQPMap} \triangleq \mathsf{Tid} \to (\mathsf{Node} \to \mathsf{AQPair})$$
$$\mathsf{b} \in \mathsf{ASBuff} \triangleq (\mathtt{lW} \cup \mathtt{Get} \cup \mathtt{Put} \cup \mathtt{nF} \cup \mathtt{nCAS} \cup \mathtt{nFAA})^* \qquad \mathbf{sqp} \in \mathsf{AQPair} \triangleq \mathsf{APipe} \times \mathsf{AWBR} \times \mathsf{AWBL}$$
$$\mathbf{pipe} \in \mathsf{APipe} \triangleq (\mathtt{Get} \cup \mathtt{Put} \cup \mathtt{nF} \cup \mathtt{nrW} \cup \mathtt{narW} \cup \mathtt{nrEX} \cup \mathtt{nlW} \cup \mathtt{nCAS} \cup \mathtt{nFAA})^*$$
$$\mathbf{wb_R} \in \mathsf{AWBR} \triangleq (\mathtt{nrW}, \mathtt{narW})^* \qquad \mathbf{wb_L} \in \mathsf{AWBL} \triangleq (\mathtt{nlW} \cup \mathtt{nlEX} \cup \mathtt{nrEX})^*$$

$$\frac{\mathsf{B}' = \mathsf{B}[t(w) \mapsto w \cdot \mathsf{B}(t(w))]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{\mathtt{lW}\langle w\rangle} \mathsf{M},\mathsf{B}',\mathsf{A},\mathsf{QP}} \qquad \frac{(\mathsf{M} \blacktriangleleft \mathsf{B}(t(r)))(\mathtt{loc}(r)) = w \qquad v_{\mathrm{r}}(r) = v_{\mathrm{w}}(w)}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{\mathtt{lR}\langle r,w\rangle} \mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}}$$

$$\frac{\mathsf{B}(t(u)) = \varepsilon \qquad \mathsf{M}(\mathtt{loc}(u)) = w \qquad v_{\mathrm{r}}(u) = v_{\mathrm{w}}(w)}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{\mathtt{CAS}\langle u,w\rangle} \mathsf{M}[x \mapsto u],\mathsf{B},\mathsf{A},\mathsf{QP}} \qquad \frac{\mathsf{B}(t(f)) = \varepsilon}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{\mathtt{F}\langle f\rangle} \mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}}$$

$$\frac{\mathsf{B}' = \mathsf{B}[t(a) \mapsto a \cdot \mathsf{B}(t(a))]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{\mathtt{Push}\langle a\rangle} \mathsf{M},\mathsf{B}',\mathsf{A},\mathsf{QP}} \qquad \frac{\mathsf{B}(t(w)) = \mathsf{b} \cdot w \qquad w \in \mathtt{lW}}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{\mathtt{B}\langle w\rangle} \mathsf{M}[x \mapsto w],\mathsf{B}[t(w) \mapsto \mathsf{b}],\mathsf{A},\mathsf{QP}}$$

$$\frac{\mathsf{B}(t(a)) = \mathsf{b} \cdot a \qquad a \notin \mathtt{lW} \qquad \mathsf{QP}(t(a))(n(a)) = \mathsf{qp} \qquad \mathsf{qp}' = \mathsf{qp}[\mathbf{pipe} \mapsto a \cdot \mathsf{qp}.\mathbf{pipe}]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{\mathtt{NIC}\langle a\rangle} \mathsf{M},\mathsf{B}[t(a) \mapsto \mathsf{b}],\mathsf{A},\mathsf{QP}[t(a) \mapsto \mathsf{QP}(t(a))[n(a) \mapsto \mathsf{qp}']]}$$

$$\frac{\mathsf{QP}(t(p))(n(p)) = \mathsf{qp} \qquad \mathsf{qp}.\mathbf{wb_L} = \alpha \cdot e \qquad e \in \mathtt{nEX} \qquad \mathsf{qp}' = \mathsf{qp}[\mathbf{wb_L} \mapsto \alpha]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{\mathtt{P}\langle p,e\rangle} \mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}[t(p) \mapsto \mathsf{QP}(t(p))[n(p) \mapsto \mathsf{qp}']]}$$

$$\frac{\mathsf{M},\mathsf{A},\mathsf{QP}(t(\lambda))(\overline{n}) \xrightarrow{\lambda}_{\mathtt{sqp}} \mathsf{M}',\mathsf{A}',\mathsf{qp}}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{\lambda} \mathsf{M}',\mathsf{B},\mathsf{A}',\mathsf{QP}[t(\lambda) \mapsto \mathsf{QP}(t(\lambda))[\overline{n} \mapsto \mathsf{qp}]]}$$

$$\text{with} \quad (\mathsf{M} \blacktriangleleft \alpha)(x) = \begin{cases} \mathsf{M}[x] & \alpha = \varepsilon \\ w & \alpha = w \cdot \beta \wedge w \in \mathcal{W} \wedge \mathtt{loc}(w) = x \\ (\mathsf{M} \blacktriangleleft \beta)(x) & \alpha = e \cdot \beta \wedge (e \notin \mathcal{W} \vee \mathtt{loc}(e) \neq x) \end{cases}$$

Figure B.3: RDMA$^{\mathrm{TSO}}$ hardware domains and hardware transitions for the annotated semantics

$$\frac{\mathbf{pipe} = \alpha \cdot f \qquad f = (\iota, t, \mathtt{nF}(n))}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathtt{nF}\langle f \rangle}_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \langle \alpha, \mathbf{wb_R}, \mathbf{wb_L} \rangle}$$

$$\frac{\begin{array}{c}\mathbf{pipe} = \alpha \cdot a \cdot \beta \qquad a = (\iota_a, t, \mathtt{Put}(\overline{y}, x)) \qquad \mathsf{M}(x) = w \qquad r = (\iota_r, t, \mathtt{nlR}(x, v_{\mathrm{w}}(w), n(\overline{y})))\\ w' = (\iota_{w'}, t, \mathtt{nrW}(\overline{y}, v_{\mathrm{w}}(w))) \qquad \beta \in (\mathtt{nrW} \cup \mathtt{narW} \cup \mathtt{Get} \cup \mathtt{nlW} \cup \mathtt{nCAS} \cup \mathtt{nFAA} \cup \mathtt{nrEX})^* \qquad \mathbf{wb_L} \in \mathtt{nEX}^*\end{array}}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathtt{nlR}\langle r, w, a, w' \rangle}_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \langle \alpha \cdot w' \cdot \beta, \mathbf{wb_R}, \mathbf{wb_L} \rangle}$$

$$\frac{\mathbf{pipe} = \alpha \cdot w \cdot \beta \qquad w = (\iota_w, t, \mathtt{nrW}(\overline{y}, v)) \qquad e = (\iota_e, t, \mathtt{nrEX}(n(\overline{y}))) \qquad \beta \in (\mathtt{Get} \cup \mathtt{nlW} \cup \mathtt{nrEX})^*}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathtt{nrW}\langle w, e \rangle}_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \langle \alpha \cdot e \cdot \beta, w \cdot \mathbf{wb_R}, \mathbf{wb_L} \rangle}$$

$$\frac{\mathbf{wb_R} = \alpha \cdot w \qquad w \in \mathtt{nrW}}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathsf{B}\langle w \rangle}_{\mathtt{sqp}} \mathsf{M}[\mathtt{loc}(w) \mapsto w], \mathsf{A}, \langle \mathbf{pipe}, \alpha, \mathbf{wb_L} \rangle}$$

$$\frac{\mathbf{pipe} = \alpha \cdot e \qquad e \in \mathtt{nrEX}}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathsf{CN}\langle e \rangle}_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \langle \alpha, \mathbf{wb_R}, e \cdot \mathbf{wb_L} \rangle}$$

$$\frac{\begin{array}{c}\mathbf{pipe} = \alpha \cdot a \cdot \beta \qquad a = (\iota_a, t, \mathtt{Get}(x, \overline{y})) \qquad \mathsf{M}(\overline{y}) = w \qquad r = (\iota_r, t, \mathtt{nrR}(\overline{y}, v_{\mathrm{w}}(w)))\\ w' = (\iota_{w'}, t, \mathtt{nlW}(x, v_{\mathrm{w}}(w), n(\overline{y}))) \qquad \beta \in (\mathtt{Get} \cup \mathtt{nlW} \cup \mathtt{nrEX})^* \qquad \mathbf{wb_R} = \varepsilon\end{array}}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathtt{nrR}\langle r, w, a, w' \rangle}_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \langle \alpha \cdot w' \cdot \beta, \mathbf{wb_R}, \mathbf{wb_L} \rangle}$$

$$\frac{\mathbf{pipe} = \alpha \cdot w \qquad w = (\iota_w, t, \mathtt{nlW}(x, v, n)) \qquad e = (\iota_e, t, \mathtt{nlEX}(n))}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathtt{nlW}\langle w, e \rangle}_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \langle \alpha, \mathbf{wb_R}, e \cdot w \cdot \mathbf{wb_L} \rangle}$$

$$\frac{\mathbf{wb_L} = \alpha \cdot w \cdot \beta \qquad w \in \mathtt{nlW} \qquad \beta \in \mathtt{nEX}^*}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathsf{B}\langle w \rangle}_{\mathtt{sqp}} \mathsf{M}[\mathtt{loc}(w) \mapsto w], \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \alpha \cdot \beta \rangle}$$

$$\frac{\begin{array}{c}\mathbf{pipe} = \alpha \cdot a \cdot \beta \quad \mathbf{wb_R} = \varepsilon \quad \mathsf{M}(\overline{x}) = w \qquad v_{\mathrm{w}}(w) \neq v \qquad \mathsf{A}(n(\overline{x})) = \bot \qquad a = (\iota_a, t, \mathtt{nCAS}(z, \overline{x}, v, u))\\ r = (\iota_r, t, \mathtt{narR}(\overline{x}, v_{\mathrm{w}}(w))) \qquad w' = (\iota_{w'}, t, \mathtt{nlW}(z, v_{\mathrm{w}}(w), n(\overline{x}))) \qquad \beta \in (\mathtt{Get} \cup \mathtt{nlW} \cup \mathtt{nrEX})^*\end{array}}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathtt{naF}\langle r, w, a, w' \rangle}_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \langle \alpha \cdot w' \cdot \beta, \mathbf{wb_R}, \mathbf{wb_L} \rangle}$$

$$\frac{\begin{array}{c}\mathbf{pipe} = \alpha \cdot a \cdot \beta \quad \mathbf{wb_R} = \varepsilon \quad \mathsf{M}(\overline{x}) = w\\ v_{\mathrm{w}}(w) = v \qquad \mathsf{A}(n(\overline{x})) = \bot \qquad a = (\iota_a, t, \mathtt{nCAS}(z, \overline{x}, v, u)) \qquad r = (\iota_r, t, \mathtt{narR}(\overline{x}, v_{\mathrm{w}}(w)))\\ w'' = (\iota_{w''}, t, \mathtt{narW}(\overline{x}, u)) \qquad w' = (\iota_{w'}, t, \mathtt{nlW}(z, v_{\mathrm{w}}(w), n(\overline{x}))) \qquad \beta \in (\mathtt{Get} \cup \mathtt{nlW} \cup \mathtt{nrEX})^*\end{array}}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathtt{narR}\langle r, w, a, w', w'' \rangle}_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}[n(\overline{x}) \mapsto \top], \langle \alpha \cdot w' \cdot w'' \cdot \beta, \mathbf{wb_R}, \mathbf{wb_L} \rangle}$$

$$\frac{\begin{array}{c}\mathbf{pipe} = \alpha \cdot a \cdot \beta \quad \mathbf{wb_R} = \varepsilon \quad \mathsf{M}(\overline{x}) = w\\ v_{\mathrm{w}}(w) + v = u \qquad \mathsf{A}(n(\overline{x})) = \bot \qquad a = (\iota_a, t, \mathtt{nFAA}(z, \overline{x}, v)) \qquad r = (\iota_r, t, \mathtt{narR}(\overline{x}, v_{\mathrm{w}}(w)))\\ w'' = (\iota_{w''}, t, \mathtt{narW}(\overline{x}, u))) \qquad w' = (\iota_{w'}, t, \mathtt{nlW}(z, v_{\mathrm{w}}(w))) \qquad \beta \in (\mathtt{Get} \cup \mathtt{nlW} \cup \mathtt{nrEX})^*\end{array}}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathtt{narR}\langle r, w, a, w', w'' \rangle}_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}[n(\overline{x}) \mapsto \top], \langle \alpha \cdot w' \cdot w'' \cdot \beta, \mathbf{wb_R}, \mathbf{wb_L} \rangle}$$

$$\frac{\mathbf{pipe} = \alpha \cdot w \cdot \beta \qquad \beta \in (\mathtt{Get} \cup \mathtt{nlW} \cup \mathtt{nrEX})^* \qquad w = (\iota_w, t, \mathtt{narW}(\overline{x}, v))}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathtt{narW}\langle w \rangle}_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \langle \alpha \cdot \beta, w \cdot \mathbf{wb_R}, \mathbf{wb_L} \rangle}$$

$$\frac{\mathbf{wb_R} = \alpha \cdot w \qquad w = (\iota_w, t, \mathtt{narW}(\overline{x}, v))}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathsf{B}\langle w \rangle}_{\mathtt{sqp}} \mathsf{M}[\mathtt{loc}(w) \mapsto w], \mathsf{A}[n(\overline{x}) \mapsto \bot], \langle \mathbf{pipe}, \alpha, \mathbf{wb_L} \rangle}$$

Figure B.4: Annotated 3 Buffers NIC Semantics

B.2 Paths, Gluing, and Other Definitions

We define a path as: $\pi \in \mathsf{Path} \triangleq (\mathsf{ALabel} \setminus \mathcal{E}\langle t \rangle)^*$

We define Annotated Operational Semantics Gluing with the following rules.

$$\frac{\mathsf{P} \xrightarrow{\mathcal{E}\langle t \rangle} \mathsf{P}'}{\mathsf{P}, \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi \Rightarrow \mathsf{P}', \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi}$$

$$\frac{\mathsf{P} \xrightarrow{\lambda} \mathsf{P}' \qquad \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP} \xrightarrow{\lambda} \mathsf{M}', \mathsf{B}', \mathsf{A}, \mathsf{QP}' \qquad \lambda \in (\mathtt{lR} \cup \mathtt{lW} \cup \mathtt{CAS} \cup \mathtt{F} \cup \mathtt{Push} \cup \mathtt{P}) \qquad \mathsf{fresh}(\lambda, \pi)}{\mathsf{P}, \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi \Rightarrow \mathsf{P}', \mathsf{M}', \mathsf{B}', \mathsf{A}, \mathsf{QP}', \lambda \cdot \pi}$$

$$\frac{\mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP} \xrightarrow{\lambda} \mathsf{M}', \mathsf{B}', \mathsf{A}', \mathsf{QP}' \qquad \lambda \in (\mathtt{NIC} \cup \mathtt{nlR} \cup \mathtt{nrR} \cup \mathtt{nlW} \cup \mathtt{nrW} \cup \mathtt{naF} \cup \mathtt{narR} \cup \mathtt{narW} \cup \mathsf{CN} \cup \mathtt{nF} \cup \mathsf{B}) \qquad \mathsf{fresh}(\lambda, \pi)}{\mathsf{P}, \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi \Rightarrow \mathsf{P}, \mathsf{M}', \mathsf{B}', \mathsf{A}', \mathsf{QP}', \lambda \cdot \pi}$$

$$\frac{\mathsf{P} \xrightarrow{\mathtt{CASF}\langle r, w \rangle} \mathsf{P}' \qquad \lambda_1 = \mathtt{F}\langle (\iota, t(r), \mathtt{F}) \rangle \qquad \lambda_2 = \mathtt{lR}\langle r, w \rangle \qquad \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP} \xrightarrow{\lambda_1} \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP} \xrightarrow{\lambda_2} \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP} \qquad \mathsf{fresh}(\lambda_1, \pi) \qquad \mathsf{fresh}(\lambda_2, \pi)}{\mathsf{P}, \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi \Rightarrow \mathsf{P}', \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \lambda_2 \cdot \lambda_1 \cdot \pi}$$

Two annotated labels are non-conflicting ($\lambda_1 \bowtie \lambda_2$) if they are of a different type or if their relevant arguments are disjoints. An annotated label is fresh if it does not conflict with any previous annotated label.

$$\mathbf{Relevant} : \mathsf{ALabel} \to 2^{E^{\text{ext}}}$$

$$\begin{aligned}
\mathbf{Relevant}(\mathtt{lR}\langle r, _ \rangle) &\triangleq \{r\} & \mathbf{Relevant}(\mathtt{narR}\langle r, _, a, w'', w' \rangle) &\triangleq \{r, a, w', w''\} \\
\mathbf{Relevant}(\mathtt{lW}\langle w \rangle) &\triangleq \{w\} & \mathbf{Relevant}(\mathtt{narW}\langle w \rangle) &\triangleq \{w\} \\
\mathbf{Relevant}(\mathtt{CAS}\langle u, _ \rangle) &\triangleq \{u\} & \mathbf{Relevant}(\mathtt{nlW}\langle w, e \rangle) &\triangleq \{w, e\} \\
\mathbf{Relevant}(\mathtt{F}\langle f \rangle) &\triangleq \{f\} & \mathbf{Relevant}(\mathtt{nrW}\langle w, e \rangle) &\triangleq \{w, e\} \\
\mathbf{Relevant}(\mathtt{Push}\langle a \rangle) &\triangleq \{a\} & \mathbf{Relevant}(\mathsf{CN}\langle e \rangle) &\triangleq \{e\} \\
\mathbf{Relevant}(\mathtt{NIC}\langle a \rangle) &\triangleq \{a\} & \mathbf{Relevant}(\mathtt{P}\langle p, e \rangle) &\triangleq \{p, e\} \\
\mathbf{Relevant}(\mathtt{nlR}\langle r, _, a, w' \rangle) &\triangleq \{r, a, w'\} & \mathbf{Relevant}(\mathtt{nF}\langle f \rangle) &\triangleq \{f\} \\
\mathbf{Relevant}(\mathtt{nrR}\langle r, _, a, w' \rangle) &\triangleq \{r, a, w'\} & \mathbf{Relevant}(\mathsf{B}\langle w \rangle) &\triangleq \{w\} \\
\mathbf{Relevant}(\mathtt{naF}\langle r, _, a, w' \rangle) &\triangleq \{r, a, w'\} & \mathbf{Relevant}(\mathcal{E}\langle _ \rangle) &\triangleq \{\}
\end{aligned}$$

$$\begin{aligned}
\lambda_1 \bowtie \lambda_2 &\triangleq \mathsf{type}(\lambda_1) \neq \mathsf{type}(\lambda_2) \ \vee \mathbf{Relevant}(\lambda_1) \cap \mathbf{Relevant}(\lambda_2) = \emptyset \\
\mathsf{fresh}(\lambda, \pi) &\triangleq \forall \lambda' \in \pi,\ \lambda \bowtie \lambda' \\
\mathsf{nodup}(\pi) &\triangleq \forall \pi_2, \lambda, \pi_1.\ \pi = \pi_2 \cdot \lambda \cdot \pi_1 \implies \mathsf{fresh}(\lambda, \pi_1)
\end{aligned}$$

Relevant(λ) are the arguments that are important to consider to avoid duplicating events. The excluded events are the write operations we lookup when reading. For instance:

- Having both $\texttt{lR}\langle r_1, w\rangle$ and $\texttt{lR}\langle r_2, w\rangle$ during an execution is fine, since w can be looked up any number of time.

- Having both $\texttt{nlR}\langle r_1, w_1, a, e_1\rangle$ and $\texttt{nlR}\langle r_2, w_2, a, e_2\rangle$ during an execution is problematic, since it means the put operation a is being run twice.

Completeness

$$
\begin{aligned}
\mathsf{complete}(\pi) \triangleq \forall a, w', e, r, w, f, w''.\ & \\
& \texttt{lW}\langle w\rangle \in \pi \implies \mathsf{B}\langle w\rangle \in \pi \\
\wedge\ & \texttt{Push}\langle a\rangle \in \pi \implies \texttt{NIC}\langle a\rangle \in \pi \\
\wedge\ & \texttt{NIC}\langle f\rangle \in \pi \wedge f \in \texttt{nF} \implies \texttt{nF}\langle f\rangle \in \pi \\
\wedge\ & \texttt{NIC}\langle a\rangle \in \pi \wedge a \in \texttt{Put} \implies \exists r, w, w'.\ \texttt{nlR}\langle r, w, a, w'\rangle \in \pi \\
\wedge\ & \texttt{NIC}\langle a\rangle \in \pi \wedge a \in \texttt{Get} \implies \exists r, w, w'.\ \texttt{nrR}\langle r, w, a, w'\rangle \in \pi \\
\wedge\ & \texttt{NIC}\langle a\rangle \in \pi \wedge a \in \texttt{nFAA} \implies \exists r, w, w'.\ \texttt{narR}\langle r, w, a, w', w''\rangle \in \pi \\
\wedge\ & \texttt{NIC}\langle a\rangle \in \pi \wedge a \in \texttt{nCAS} \implies \left(\begin{array}{c} \exists r, w, w'.\ \texttt{naF}\langle r, w, a, w'\rangle \in \pi \\ \vee\ \exists r, w, w', w''.\ \texttt{narR}\langle r, w, a, w', w''\rangle \in \pi \end{array}\right) \\
\wedge\ & \texttt{nlR}\langle r, w, a, w'\rangle \in \pi \implies \exists e.\ \texttt{nrW}\langle w', e\rangle \in \pi \\
\wedge\ & \texttt{nrR}\langle r, w, a, w'\rangle \in \pi \implies \exists e.\ \texttt{nlW}\langle w', e\rangle \in \pi \\
\wedge\ & \texttt{narR}\langle r, w, a, w', w''\rangle \in \pi \implies \texttt{narW}\langle w''\rangle \in \pi \\
\wedge\ & \texttt{narR}\langle r, w, a, w', w''\rangle \in \pi \implies \exists e.\ \texttt{nlW}\langle w', e\rangle \in \pi \\
\wedge\ & \texttt{naF}\langle r, w, a, w'\rangle \in \pi \implies \exists e.\ \texttt{nlW}\langle w', e\rangle \in \pi \\
\wedge\ & \texttt{nlW}\langle w, e\rangle \in \pi \implies \mathsf{B}\langle w\rangle \in \pi \\
\wedge\ & \texttt{nrW}\langle w, e\rangle \in \pi \implies \mathsf{B}\langle w\rangle \in \pi \wedge \mathsf{CN}\langle e\rangle \in \pi \\
\wedge\ & \texttt{narW}\langle w\rangle \in \pi \implies \mathsf{B}\langle w\rangle \in \pi
\end{aligned}
$$

Informal: every pending operation is done and (most) buffers are empty. Note that some `nEX` (i.e., completion notifications) might still be in $\mathbf{wb_L}$.

For a path π without duplicate (e.g. if $\mathsf{nodup}(\pi)$ holds), we define the total ordering of its annotated labels as follows. Note that the early part of the path is on the right.

$$\lambda_1 \prec_\pi \lambda_2 \triangleq \exists \pi_1, \pi_2, \pi_3 \text{ s.t. } \pi = \pi_3 \cdot \lambda_2 \cdot \pi_2 \cdot \lambda_1 \cdot \pi_1$$

Backward Completeness (with ordering)

$$\begin{aligned}
\mathsf{backComp}(\pi) \triangleq \forall a, w', e, r, w, f, p, w''.\ & \\
& \mathsf{B}\langle w\rangle \in \pi \implies \left(\begin{array}{l} \mathtt{lW}\langle w\rangle \prec_\pi \mathsf{B}\langle w\rangle \\ \vee\ \exists e.\mathtt{nlW}\langle w, e\rangle \prec_\pi \mathsf{B}\langle w\rangle \\ \vee\ \exists e.\mathtt{nrW}\langle w, e\rangle \prec_\pi \mathsf{B}\langle w\rangle \\ \vee\ \exists e.\mathtt{narW}\langle w\rangle \prec_\pi \mathsf{B}\langle w\rangle \end{array}\right) \\
\wedge\ & \mathtt{NIC}\langle a\rangle \in \pi \implies \mathtt{Push}\langle a\rangle \prec_\pi \mathtt{NIC}\langle a\rangle \\
\wedge\ & \mathtt{nF}\langle f\rangle \in \pi \implies \mathtt{NIC}\langle f\rangle \prec_\pi \mathtt{nF}\langle f\rangle \\
\wedge\ & \mathtt{nlR}\langle r, w, a, w'\rangle \in \pi \implies \mathtt{NIC}\langle a\rangle \prec_\pi \mathtt{nlR}\langle r, w, a, w'\rangle \\
\wedge\ & \mathtt{nrR}\langle r, w, a, w'\rangle \in \pi \implies \mathtt{NIC}\langle a\rangle \prec_\pi \mathtt{nrR}\langle r, w, a, w'\rangle \\
\wedge\ & \mathtt{naF}\langle r, w, a, w'\rangle \in \pi \implies \mathtt{NIC}\langle a\rangle \prec_\pi \mathtt{naF}\langle r, w, a, w'\rangle \\
\wedge\ & \mathtt{narR}\langle r, w, a, w', w''\rangle \in \pi \implies \mathtt{NIC}\langle a\rangle \prec_\pi \mathtt{narR}\langle r, w, a, w', w''\rangle \\
\wedge\ & \mathtt{nrW}\langle w', e\rangle \in \pi \implies \exists r, w, a.\ \mathtt{nlR}\langle r, w, a, w'\rangle \prec_\pi \mathtt{nrW}\langle w', e\rangle \\
\wedge\ & \mathtt{nlW}\langle w', e\rangle \in \pi \implies \left(\begin{array}{l} \exists r, w, a.\ \mathtt{nrR}\langle r, w, a, w'\rangle \prec_\pi \mathtt{nlW}\langle w', e\rangle \\ \vee\ \exists r, w, a.\ \mathtt{naF}\langle r, w, a, w'\rangle \prec_\pi \mathtt{nlW}\langle w', e\rangle \\ \vee\ \exists r, w, a, w''.\left(\begin{array}{l} \mathtt{narR}\langle r, w, a, w', w''\rangle \\ \prec_\pi \mathtt{narW}\langle w''\rangle \prec_\pi \mathtt{nlW}\langle w', e\rangle \end{array}\right) \end{array}\right) \\
\wedge\ & \mathtt{narW}\langle w'\rangle \in \pi \implies \exists r, w, a, w''.\mathtt{narR}\langle r, w, a, w', w''\rangle \prec_\pi \mathtt{narW}\langle w'\rangle \\
\wedge\ & \mathsf{CN}\langle e\rangle \in \pi \implies \quad \exists w.\ \mathtt{nrW}\langle w, e\rangle \prec_\pi \mathsf{CN}\langle e\rangle \\
\wedge\ & \mathtt{P}\langle p, e\rangle \in \pi \implies \left(\begin{array}{l} \exists w.\ \mathtt{nlW}\langle w, e\rangle \prec_\pi \mathsf{B}\langle w\rangle \prec_\pi \mathtt{P}\langle p, e\rangle \\ \vee\ \mathsf{CN}\langle e\rangle \prec_\pi \mathtt{P}\langle p, e\rangle \end{array}\right)
\end{aligned}$$

Poll Order

$$\mathsf{pollOrder}(\pi) \triangleq \forall e_1, e_2.\left(\begin{array}{l} \mathsf{sameqp}(e_1, e_2) \\ \wedge\ \lambda_1 \in \{\mathtt{nlW}\langle \_, e_1\rangle, \mathsf{CN}\langle e_1\rangle\} \\ \wedge\ \lambda_2 \in \{\mathtt{nlW}\langle \_, e_2\rangle, \mathsf{CN}\langle e_2\rangle\} \\ \wedge\ \lambda_1 \prec_\pi \lambda_2 \\ \wedge\ \mathtt{P}\langle \_, e_2\rangle \in \pi \end{array}\right) \implies \mathtt{P}\langle \_, e_1\rangle \prec_\pi \mathtt{P}\langle \_, e_2\rangle$$

Flush Order

$$
\begin{aligned}
&\mathsf{bufFlushOrd}(\pi) \triangleq \\
&\quad \forall w_1, w_2 \in \mathtt{lW}.\ \left(\begin{array}{l} t(w_1) = t(w_2) \implies \\ (\mathsf{B}\langle w_2\rangle \in \pi \wedge \mathtt{lW}\langle w_1\rangle \prec_\pi \mathtt{lW}\langle w_2\rangle) \iff \mathsf{B}\langle w_1\rangle \prec_\pi \mathsf{B}\langle w_2\rangle \end{array}\right) \\
\wedge\ &\forall a_1, a_2 \in (\mathtt{Get} \cup \mathtt{Put} \cup \mathtt{nF} \cup \mathtt{nCAS} \cup \mathtt{nFAA}). \\
&\quad \left(\begin{array}{l} t(a_1) = t(a_2) \implies \\ (\mathtt{NIC}\langle a_2\rangle \in \pi \wedge \mathtt{Push}\langle a_1\rangle \prec_\pi \mathtt{Push}\langle a_2\rangle) \iff \mathtt{NIC}\langle a_1\rangle \prec_\pi \mathtt{NIC}\langle a_2\rangle \end{array}\right) \\
\wedge\ &\forall a_1 \in (\mathtt{Get} \cup \mathtt{Put} \cup \mathtt{nF} \cup \mathtt{nCAS} \cup \mathtt{nFAA}), w_2 \in \mathtt{lW}. \\
&\quad \left(\begin{array}{l} t(a_1) = t(w_2) \implies \\ \quad (\mathsf{B}\langle w_2\rangle \in \pi \wedge \mathtt{Push}\langle a_1\rangle \prec_\pi \mathtt{lW}\langle w_2\rangle) \iff \mathtt{NIC}\langle a_1\rangle \prec_\pi \mathsf{B}\langle w_2\rangle \\ \wedge\, (\mathtt{NIC}\langle a_1\rangle \in \pi \wedge \mathtt{lW}\langle w_2\rangle \prec_\pi \mathtt{Push}\langle a_1\rangle) \iff \mathsf{B}\langle w_2\rangle \prec_\pi \mathtt{NIC}\langle a_1\rangle \end{array}\right) \\
\wedge\ &\forall w_1, w_2 \in \mathtt{nlW}.\ \left(\begin{array}{l} \mathsf{sameqp}(w_1, w_2) \implies \\ (\mathsf{B}\langle w_2\rangle \in \pi \wedge \mathtt{nlW}\langle w_1\rangle \prec_\pi \mathtt{nlW}\langle w_2\rangle) \iff \mathsf{B}\langle w_1\rangle \prec_\pi \mathsf{B}\langle w_2\rangle \end{array}\right) \\
\wedge\ &\forall w_1, w_2 \in \mathtt{nrW}.\ \left(\begin{array}{l} \mathsf{sameqp}(w_1, w_2) \implies \\ (\mathsf{B}\langle w_2\rangle \in \pi \wedge \mathtt{nrW}\langle w_1\rangle \prec_\pi \mathtt{nrW}\langle w_2\rangle) \iff \mathsf{B}\langle w_1\rangle \prec_\pi \mathsf{B}\langle w_2\rangle \end{array}\right) \\
\wedge\ &\forall w_1, w_2 \in \mathtt{narW}.\ \left(\begin{array}{l} \mathsf{sameqp}(w_1, w_2) \implies \\ (\mathsf{B}\langle w_2\rangle \in \pi \wedge \mathtt{narW}\langle w_1\rangle \prec_\pi \mathtt{narW}\langle w_2\rangle) \iff \mathsf{B}\langle w_1\rangle \prec_\pi \mathsf{B}\langle w_2\rangle \end{array}\right) \\
\wedge\ &\forall w_1 \in \mathtt{nrW}, w_2 \in \mathtt{narW}.\ \left(\begin{array}{l} \mathsf{sameqp}(w_1, w_2) \implies \\ (\mathsf{B}\langle w_2\rangle \in \pi \wedge \mathtt{nrW}\langle w_1\rangle \prec_\pi \mathtt{narW}\langle w_2\rangle) \iff \mathsf{B}\langle w_1\rangle \prec_\pi \mathsf{B}\langle w_2\rangle \end{array}\right) \\
\wedge\ &\forall w_1 \in \mathtt{narW}, w_2 \in \mathtt{nrW}.\ \left(\begin{array}{l} \mathsf{sameqp}(w_1, w_2) \implies \\ (\mathsf{B}\langle w_2\rangle \in \pi \wedge \mathtt{narW}\langle w_1\rangle \prec_\pi \mathtt{nrW}\langle w_2\rangle) \iff \mathsf{B}\langle w_1\rangle \prec_\pi \mathsf{B}\langle w_2\rangle \end{array}\right) \\
\wedge\ &\forall w \in \mathtt{lW}, f \in \mathtt{F}.\ \mathtt{lW}\langle w\rangle \prec_\pi \mathtt{F}\langle f\rangle \wedge t(w) = t(f) \implies \mathsf{B}\langle w\rangle \prec_\pi \mathtt{F}\langle f\rangle \\
\wedge\ &\forall w \in \mathtt{lW}, u \in \mathtt{CAS}.\ \mathtt{lW}\langle w\rangle \prec_\pi \mathtt{CAS}\langle u, \_\rangle \wedge t(w) = t(u) \implies \mathsf{B}\langle w\rangle \prec_\pi \mathtt{CAS}\langle u, \_\rangle \\
\wedge\ &\forall w \in \mathtt{nlW}, r \in \mathtt{nlR}. \\
&\quad (\mathtt{nlW}\langle w, \_\rangle \prec_\pi \mathtt{nlR}\langle r, \_, \_, \_\rangle \wedge \mathsf{sameqp}(w, r)) \implies \mathsf{B}\langle w\rangle \prec_\pi \mathtt{nlR}\langle r, \_, \_, \_\rangle \\
\wedge\ &\forall w \in \mathtt{nrW}, r \in \mathtt{nrR}. \\
&\quad (\mathtt{nrW}\langle w, \_\rangle \prec_\pi \mathtt{nrR}\langle r, \_, \_, \_\rangle \wedge \mathsf{sameqp}(w, r)) \implies \mathsf{B}\langle w\rangle \prec_\pi \mathtt{nrR}\langle r, \_, \_, \_\rangle \\
\wedge\ &\forall w \in \mathtt{nrW}, r \in \mathtt{narR}. \\
&\quad (\mathtt{nrW}\langle w, \_\rangle \prec_\pi \mathtt{naF}\langle r, \_, \_, \_\rangle \wedge \mathsf{sameqp}(w, r)) \implies \mathsf{B}\langle w\rangle \prec_\pi \mathtt{naF}\langle r, \_, \_, \_\rangle \\
\wedge\ &\forall w \in \mathtt{nrW}, r \in \mathtt{narR}. \\
&\quad (\mathtt{nrW}\langle w, \_\rangle \prec_\pi \mathtt{narR}\langle r, \_, \_, \_, \_\rangle \wedge \mathsf{sameqp}(w, r)) \implies \mathsf{B}\langle w\rangle \prec_\pi \mathtt{narR}\langle r, \_, \_, \_, \_\rangle \\
\wedge\ &\forall w \in \mathtt{narW}, r \in \mathtt{nrR}. \\
&\quad (\mathtt{narW}\langle w\rangle \prec_\pi \mathtt{nrR}\langle r, \_, \_, \_\rangle \wedge \mathsf{sameqp}(w, r)) \implies \mathsf{B}\langle w\rangle \prec_\pi \mathtt{nrR}\langle r, \_, \_, \_\rangle \\
\wedge\ &\forall w \in \mathtt{narW}, r \in \mathtt{narR}. \\
&\quad (\mathtt{narW}\langle w\rangle \prec_\pi \mathtt{naF}\langle r, \_, \_, \_\rangle \wedge \mathsf{sameqp}(w, r)) \implies \mathsf{B}\langle w\rangle \prec_\pi \mathtt{naF}\langle r, \_, \_, \_\rangle \\
\wedge\ &\forall w \in \mathtt{narW}, r \in \mathtt{narR}. \\
&\quad (\mathtt{narW}\langle w\rangle \prec_\pi \mathtt{narR}\langle r, \_, \_, \_, \_\rangle \wedge \mathsf{sameqp}(w, r)) \implies \mathsf{B}\langle w\rangle \prec_\pi \mathtt{narR}\langle r, \_, \_, \_, \_\rangle
\end{aligned}
$$

NIC Order

$$
\begin{aligned}
&\mathsf{nicActOrder}(\pi) \triangleq \forall a_1, a_2.\ \mathtt{NIC}\langle a_1\rangle \prec_\pi \mathtt{NIC}\langle a_2\rangle \wedge \mathsf{sameqp}(a_1, a_2) \implies \\
&\quad (a_1 \in \mathtt{nF} \wedge a_2 \in \mathtt{Get} \wedge \mathtt{nrR}\langle \_, \_, a_2, \_\rangle \in \pi \implies \mathtt{nF}\langle a_1\rangle \prec_\pi \mathtt{nrR}\langle \_, \_, a_2, \_\rangle) \\
&\wedge\ (a_1 \in \mathtt{nF} \wedge a_2 \in \mathtt{Put} \wedge \mathtt{nlR}\langle \_, \_, a_2, \_\rangle \in \pi \implies \mathtt{nF}\langle a_1\rangle \prec_\pi \mathtt{nlR}\langle \_, \_, a_2, \_\rangle) \\
&\wedge\ (a_1 \in \mathtt{nF} \wedge a_2 \in \mathtt{nCAS} \wedge \mathtt{naF}\langle \_, \_, a_2, \_\rangle \in \pi \implies \mathtt{nF}\langle a_1\rangle \prec_\pi \mathtt{naF}\langle \_, \_, a_2, \_,\rangle) \\
&\wedge\ (a_1 \in \mathtt{nF} \wedge a_2 \in \mathtt{nRMW} \wedge \mathtt{narR}\langle \_, \_, a_2, \_, \_\rangle \in \pi \implies \mathtt{nF}\langle a_1\rangle \prec_\pi \mathtt{narR}\langle \_, \_, a_2, \_, \_\rangle) \\
&\wedge\ (a_1 \in \mathtt{nF} \wedge a_2 \in \mathtt{nF} \wedge \mathtt{nF}\langle a_2\rangle \in \pi \implies \mathtt{nF}\langle a_1\rangle \prec_\pi \mathtt{nF}\langle a_2\rangle) \\
&\wedge\ (a_1 \in \mathtt{Get} \wedge a_2 \in \mathtt{nF} \wedge \mathtt{nF}\langle a_2\rangle \in \pi \implies \mathtt{nrR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nlW}\langle w_1, \_\rangle \prec_\pi \mathtt{nF}\langle a_2\rangle) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{nF} \wedge \mathtt{nF}\langle a_2\rangle \in \pi \\ \implies \mathtt{nlR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nrW}\langle w_1, e_1\rangle \prec_\pi \mathsf{CN}\langle e_1\rangle \prec_\pi \mathtt{nF}\langle a_2\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{nCAS} \wedge a_2 \in \mathtt{nF} \wedge \mathtt{nF}\langle a_2\rangle \in \pi \\ \implies \mathtt{narR}\langle \_, \_, a_1, w_1, w_2\rangle \prec_\pi \mathtt{narW}\langle w_2\rangle \prec_\pi \mathtt{nlW}\langle w_1, \_\rangle \prec_\pi \mathtt{nF}\langle a_2\rangle \\ \quad \vee\ \mathtt{naF}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nlW}\langle w_1, \_\rangle \prec_\pi \mathtt{nF}\langle a_2\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{nFAA} \wedge a_2 \in \mathtt{nF} \wedge \mathtt{nF}\langle a_2\rangle \in \pi \\ \implies \mathtt{narR}\langle \_, \_, a_1, w_1, w_2\rangle \prec_\pi \mathtt{narW}\langle w_2\rangle \prec_\pi \mathtt{nlW}\langle w_1, \_\rangle \prec_\pi \mathtt{nF}\langle a_2\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Get} \wedge a_2 \in \mathtt{Get} \wedge \mathtt{nrR}\langle \_, \_, a_2, w_2\rangle \prec_\pi \mathtt{nlW}\langle w_2, \_\rangle \\ \implies \mathtt{nrR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nlW}\langle w_1, \_\rangle \prec_\pi \mathtt{nlW}\langle w_2, \_\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Get} \wedge a_2 \in \mathtt{Put} \wedge \mathtt{nlR}\langle \_, \_, a_2, w_2\rangle \prec_\pi \mathtt{nrW}\langle w_2, e_2\rangle \prec_\pi \mathsf{CN}\langle e_2\rangle \\ \implies \mathtt{nrR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nlW}\langle w_1, \_\rangle \prec_\pi \mathsf{CN}\langle e_2\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Get} \wedge a_2 \in \mathtt{nCAS} \wedge \left(\begin{array}{l} \mathtt{narR}\langle \_, \_, a_2, w_2, \_\rangle \prec_\pi \mathtt{nlW}\langle w_2, \_\rangle \\ \vee\ \mathtt{naF}\langle \_, \_, a_2, w_2\rangle \prec_\pi \mathtt{nlW}\langle w_2, \_\rangle \end{array}\right) \\ \implies \mathtt{nrR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nlW}\langle w_1, \_\rangle \prec_\pi \mathtt{nlW}\langle w_2, \_\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Get} \wedge a_2 \in \mathtt{nFAA} \wedge \mathtt{narR}\langle \_, \_, a_2, w_2, \_\rangle \prec_\pi \mathtt{nlW}\langle w_2, \_\rangle \\ \implies \mathtt{nrR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nlW}\langle w_1, \_\rangle \prec_\pi \mathtt{nlW}\langle w_2, \_\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{Get} \wedge \mathtt{nrR}\langle \_, \_, a_2, \_\rangle \in \pi \\ \implies \mathtt{nlR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nrW}\langle w_1, \_\rangle \prec_\pi \mathtt{nrR}\langle \_, \_, a_2, \_\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{Get} \wedge \mathtt{nrR}\langle \_, \_, a_2, w_2\rangle \prec_\pi \mathtt{nlW}\langle w_2, \_\rangle \\ \implies \mathtt{nlR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nrW}\langle w_1, e_1\rangle \prec_\pi \mathsf{CN}\langle e_1\rangle \prec_\pi \mathtt{nlW}\langle w_2, \_\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{nCAS} \wedge \mathtt{naF}\langle \_, \_, a_2, w_2\rangle \in \pi \\ \implies \mathtt{nlR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nrW}\langle w_1, e_1\rangle \prec_\pi \mathtt{naF}\langle \_, \_, a_2, w_2\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{nCAS} \cup \mathtt{nFAA} \wedge \mathtt{narR}\langle \_, \_, a_2, w_2, \_\rangle \in \pi \\ \implies \mathtt{nlR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nrW}\langle w_1, e_1\rangle \prec_\pi \mathtt{narR}\langle \_, \_, a_2, w_2, \_\rangle \end{array}\right) \\
&\wedge\ (a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{Put} \wedge \mathtt{nlR}\langle \_, \_, a_2, \_\rangle \in \pi \implies \mathtt{nlR}\langle \_, \_, a_1, \_\rangle \prec_\pi \mathtt{nlR}\langle \_, \_, a_2, \_\rangle) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{Put} \wedge \mathtt{nlR}\langle \_, \_, a_2, w_2\rangle \prec_\pi \mathtt{nrW}\langle w_2, \_\rangle \\ \implies \mathtt{nlR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nrW}\langle w_1, \_\rangle \prec_\pi \mathtt{nrW}\langle w_2, \_\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{Put} \wedge \mathtt{nlR}\langle \_, \_, a_2, w_2\rangle \prec_\pi \mathtt{nrW}\langle w_2, e_2\rangle \prec_\pi \mathsf{CN}\langle e_2\rangle \\ \implies \mathtt{nlR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nrW}\langle w_1, e_1\rangle \prec_\pi \mathsf{CN}\langle e_1\rangle \prec_\pi \mathsf{CN}\langle e_2\rangle \end{array}\right)
\end{aligned}
$$

$$
\begin{array}{l}
\wedge \left(\begin{array}{l} a_1 \in (\texttt{nCAS} \cup \texttt{nFAA}) \wedge a_2 \in \texttt{Get} \wedge \texttt{nrR}\langle \_, \_, a_2, \_ \rangle \in \pi \\ \implies \left(\begin{array}{l} a_1 \in \texttt{nCAS} \wedge \texttt{naF}\langle \_, \_, a_1, \_ \rangle \prec_\pi \texttt{nrR}\langle \_, \_, a_2, w_2 \rangle \\ \vee\ \texttt{narR}\langle \_, \_, a_1, w_1, \_ \rangle \prec_\pi \texttt{narW}\langle w_1 \rangle \prec_\pi \texttt{nrR}\langle \_, \_, a_2, w_2 \rangle \end{array} \right) \end{array} \right) \\
\wedge \left(\begin{array}{l} a_1 \in (\texttt{nCAS} \cup \texttt{nFAA}) \wedge a_2 \in \texttt{Get} \wedge \texttt{nrR}\langle \_, \_, a_2, w_2 \rangle \prec_\pi \texttt{nlW}\langle w_2, \_ \rangle \\ \implies \left(\begin{array}{l} a_1 \in \texttt{nCAS} \wedge \texttt{naF}\langle \_, \_, a_1, w_1 \rangle \prec_\pi \texttt{nlW}\langle w_1, \_ \rangle \prec_\pi \texttt{nlW}\langle w_2, \_ \rangle \\ \vee\ \texttt{narR}\langle \_, \_, a_1, w_1, \_ \rangle \prec_\pi \texttt{nlW}\langle w_1, \_ \rangle \prec_\pi \texttt{nlW}\langle w_2, \_ \rangle \end{array} \right) \end{array} \right) \\
\wedge \left(\begin{array}{l} a_1 \in (\texttt{nCAS} \cup \texttt{nFAA}) \wedge a_2 \in \texttt{Put} \wedge \texttt{nlR}\langle \_, \_, a_2, w_2 \rangle \prec_\pi \texttt{nrW}\langle w_2, \_ \rangle \\ \implies \left(\begin{array}{l} a_1 \in \texttt{nCAS} \wedge \texttt{naF}\langle \_, \_, a_1, w_1 \rangle \prec_\pi \texttt{nrW}\langle w_2, \_ \rangle \\ \vee\ \texttt{narR}\langle \_, \_, a_1, w_1, \_ \rangle \prec_\pi \texttt{narW}\langle w_1 \rangle \prec_\pi \texttt{nrW}\langle w_2, \_ \rangle \end{array} \right) \end{array} \right) \\
\wedge \left(\begin{array}{l} a_1 \in (\texttt{nCAS} \cup \texttt{nFAA}) \wedge a_2 \in \texttt{Put} \wedge \texttt{nlR}\langle \_, \_, a_2, w_2 \rangle \prec_\pi \texttt{nrW}\langle w_2, e_2 \rangle \prec_\pi \mathsf{CN}\langle e_2 \rangle \\ \implies \left(\begin{array}{l} a_1 \in \texttt{nCAS} \wedge \texttt{naF}\langle \_, \_, a_1, w_1 \rangle \prec_\pi \texttt{nlW}\langle w_1, \_ \rangle \prec_\pi \mathsf{CN}\langle e_2 \rangle \\ \vee\ \texttt{narR}\langle \_, \_, a_1, w_1, \_ \rangle \prec_\pi \texttt{nlW}\langle w_1, \_ \rangle \prec_\pi \mathsf{CN}\langle e_2 \rangle \end{array} \right) \end{array} \right) \\
\wedge \left(\begin{array}{l} a_1 \in (\texttt{nCAS} \cup \texttt{nFAA}) \wedge a_2 \in \texttt{nCAS} \wedge \texttt{naF}\langle \_, \_, a_2, \_ \rangle \in \pi \\ \implies \left(\begin{array}{l} a_1 \in \texttt{nCAS} \wedge \texttt{naF}\langle \_, \_, a_1, \_ \rangle \prec_\pi \texttt{naF}\langle \_, \_, a_2, \_ \rangle \\ \vee\ \texttt{narR}\langle \_, \_, a_1, \_, w_1 \rangle \prec_\pi \texttt{narW}\langle w_1 \rangle \prec_\pi \texttt{naF}\langle \_, \_, a_2, \_ \rangle \end{array} \right) \end{array} \right) \\
\wedge \left(\begin{array}{l} a_1, a_2 \in (\texttt{nCAS} \cup \texttt{nFAA}) \wedge \texttt{narR}\langle \_, \_, a_2, \_, \_ \rangle \in \pi \\ \implies \left(\begin{array}{l} a_1 \in \texttt{nCAS} \wedge \texttt{naF}\langle \_, \_, a_1, w_1 \rangle \prec_\pi \texttt{narR}\langle \_, \_, a_2, \_, \_ \rangle \\ \vee\ \texttt{narR}\langle \_, \_, a_1, \_, w_1 \rangle \prec_\pi \texttt{narW}\langle w_1 \rangle \prec_\pi \texttt{narR}\langle \_, \_, a_2, \_, \_ \rangle \end{array} \right) \end{array} \right) \\
\wedge \left(\begin{array}{l} a_1, a_2 \in (\texttt{nCAS} \cup \texttt{nFAA}) \wedge \left(\begin{array}{l} a_1 \in \texttt{nCAS} \wedge \texttt{naF}\langle \_, \_, a_2, w_2 \rangle \prec_\pi \texttt{nlW}\langle w_2, \_ \rangle \\ \vee\ \texttt{narR}\langle \_, \_, a_2, w_2, \_ \rangle \prec_\pi \texttt{nlW}\langle w_2, \_ \rangle \end{array} \right) \\ \implies \left(\begin{array}{l} a_1 \in \texttt{nCAS} \wedge \texttt{naF}\langle \_, \_, a_1, w_1 \rangle \prec_\pi \texttt{nlW}\langle w_1, \_ \rangle \prec_\pi \texttt{nlW}\langle w_2, \_ \rangle \\ \vee\ \texttt{narR}\langle \_, \_, a_1, w_1, \_ \rangle \prec_\pi \texttt{nlW}\langle w_1, \_ \rangle \prec_\pi \texttt{nlW}\langle w_2, \_ \rangle \end{array} \right) \end{array} \right)
\end{array}
$$

NIC Atomicity

$$
\mathsf{nicAtomicity}(\pi) \triangleq \forall a_1, a_2, r, w. \left(\begin{array}{l} \lambda_1 = \texttt{narR}\langle r_1, \_, a_1, \_, w \rangle \\ \wedge\ \lambda_2 \in \{\texttt{naF}\langle \_, \_, a_2, \_ \rangle, \texttt{narR}\langle \_, \_, a_2, \_, \_ \rangle\} \\ \wedge\ a_1, a_2 \in \texttt{nRMW} \ \wedge\ \overline{n}(a_1) = \overline{n}(a_2) \ \wedge\ \lambda_1 \prec_\pi \lambda_2 \end{array} \right) \implies \mathsf{B}\langle w \rangle \prec_\pi \lambda_2
$$

Read Order

$$
\begin{array}{rl}
\mathsf{wfrd}(\pi) \triangleq & \forall \pi_2, r, w, \pi_1.\ \pi = \pi_2 \cdot \texttt{lR}\langle r, w \rangle \cdot \pi_1 \implies \mathsf{wfrdCPU}(r, w, \pi_1) \\
\wedge & \forall \pi_2, u, w, \pi_1.\ \pi = \pi_2 \cdot \texttt{CAS}\langle u, w \rangle \cdot \pi_1 \implies \mathsf{wfrdCPU}(u, w, \pi_1) \\
\wedge & \forall \pi_2, r, w, \pi_1.\ \pi = \pi_2 \cdot \texttt{nlR}\langle r, w, \_, \_ \rangle \cdot \pi_1 \implies \mathsf{wfrdNIC}(r, w, \pi_1) \\
\wedge & \forall \pi_2, r, w, \pi_1.\ \pi = \pi_2 \cdot \texttt{nrR}\langle r, w, \_, \_ \rangle \cdot \pi_1 \implies \mathsf{wfrdNIC}(r, w, \pi_1) \\
\wedge & \forall \pi_2, r, w, \pi_1.\ \pi = \pi_2 \cdot \texttt{naF}\langle r, w, \_, \_ \rangle \cdot \pi_1 \implies \mathsf{wfrdNIC}(r, w, \pi_1) \\
\wedge & \forall \pi_2, r, w, \pi_1.\ \pi = \pi_2 \cdot \texttt{narR}\langle r, w, \_, \_, \_ \rangle \cdot \pi_1 \implies \mathsf{wfrdNIC}(r, w, \pi_1)
\end{array}
$$

$$\mathsf{wfrdCPU}(r,w,\pi) \triangleq \left(\begin{array}{l} \exists \pi_2, \lambda, \pi_1.\ \pi = \pi_2 \cdot \lambda \cdot \pi_1 \\ \wedge\ \lambda \in \{\mathsf{B}\langle w\rangle, \mathtt{CAS}\langle w, \_\rangle\} \\ \wedge\ \{\mathsf{B}\langle w'\rangle, \mathtt{CAS}\langle w', \_\rangle \in \pi_2 \mid \mathtt{loc}(w') = \mathtt{loc}(r)\} = \emptyset \\ \wedge\ \left\{ w' \;\middle|\; \begin{array}{l} \mathtt{lW}\langle w'\rangle \in \pi \wedge \mathsf{B}\langle w'\rangle \notin \pi\ \wedge \\ \mathtt{loc}(w') = \mathtt{loc}(r) \wedge t(w') = t(r) \end{array} \right\} = \emptyset \end{array}\right)$$
$$\vee \left(\begin{array}{l} \exists \pi_2, \lambda, \pi_1.\ \pi = \pi_2 \cdot \lambda \cdot \pi_1 \\ \wedge\ \lambda = \mathtt{lW}\langle w\rangle \wedge t(w) = t(r) \wedge \mathsf{B}\langle w\rangle \notin \pi_2 \\ \wedge\ \{\mathtt{lW}\langle w'\rangle \in \pi_2 \mid \mathtt{loc}(w') = \mathtt{loc}(r) \wedge t(w') = t(r)\} = \emptyset \end{array}\right)$$
$$\vee \left(\begin{array}{l} w = init_{\mathtt{loc}(w)} \wedge \\ \left\{ \begin{array}{l} \mathsf{B}\langle w'\rangle, \mathtt{CAS}\langle w', \_\rangle \in \pi, \\ \mathtt{lW}\langle w''\rangle \in \pi \end{array} \;\middle|\; \begin{array}{l} \mathtt{loc}(w') = \mathtt{loc}(r)\ \wedge \\ \mathtt{loc}(w'') = \mathtt{loc}(r) \wedge t(w'') = t(r) \end{array} \right\} = \emptyset \end{array}\right)$$

$$\mathsf{wfrdNIC}(r,w,\pi) \triangleq \left(\begin{array}{l} \exists \pi_2, \lambda, \pi_1.\ \pi = \pi_2 \cdot \lambda \cdot \pi_1 \\ \wedge\ \lambda \in \{\mathsf{B}\langle w\rangle, \mathtt{CAS}\langle w, \_\rangle\} \\ \wedge\ \{\mathsf{B}\langle w'\rangle, \mathtt{CAS}\langle w', \_\rangle \in \pi_2 \mid \mathtt{loc}(w') = \mathtt{loc}(r)\} = \emptyset \end{array}\right)$$
$$\vee \left(\begin{array}{l} w = init_{\mathtt{loc}(w)} \wedge \\ \{\mathsf{B}\langle w'\rangle, \mathtt{CAS}\langle w', \_\rangle \in \pi \mid \mathtt{loc}(w') = \mathtt{loc}(r)\} = \emptyset \end{array}\right)$$

Well-formed path

$$\begin{aligned} \mathsf{wfp}(\pi) \triangleq\ & \mathsf{nodup}(\pi) \\ \wedge\ & \mathsf{backComp}(\pi) \\ \wedge\ & \mathsf{bufFlushOrd}(\pi) \\ \wedge\ & \mathsf{pollOrder}(\pi) \\ \wedge\ & \mathsf{nicActOrder}(\pi) \\ \wedge\ & \mathsf{nicAtomicity}(\pi) \\ \wedge\ & \mathsf{wfrd}(\pi) \end{aligned}$$

Definition 5.

$$\begin{aligned} \mathsf{wf}(\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP},\pi) \triangleq\ & \mathsf{wfp}(\pi) \\ \wedge\ & \forall x \in \mathsf{Loc}.\ \mathsf{M}(x) = \mathsf{read}(\pi, x) \\ \wedge\ & \forall t \in \mathsf{Tid}.\mathsf{B}(t) = \mathsf{mksbuff}(\varepsilon, t, \pi) \\ \wedge\ & \forall n \in \mathsf{Node}.\mathsf{A}(n) = \mathsf{chkatm}(n, \pi) \\ \wedge\ & \forall t \in \mathsf{Tid}. \forall \overline{n} \in (\mathsf{Node} \setminus \{n(t)\}). \left(\begin{array}{l} \mathsf{QP}(t)(\overline{n}).\mathbf{pipe} = \mathsf{mkpipe}(\varepsilon, t, \overline{n}, \pi) \\ \mathsf{QP}(t)(\overline{n}).\mathbf{wb_R} = \mathsf{mkwbR}(\varepsilon, t, \overline{n}, \pi) \\ \mathsf{QP}(t)(\overline{n}).\mathbf{wb_L} = \mathsf{mkwbL}(\varepsilon, t, \overline{n}, \pi) \end{array}\right) \end{aligned}$$

Where, the functions read, mksbuff, chkatm, mkpipe, mkwbR, and mkwbL are defined below.

$$\mathsf{read}(\lambda{\cdot}\pi, x) \triangleq \begin{cases} w & \lambda \in \{\mathsf{B}\langle w\rangle, \mathtt{CAS}\langle w, \_\rangle\} \wedge \mathtt{loc}(w) = x \\ \mathsf{read}(\pi, x) & \text{otherwise} \end{cases}$$
$$\mathsf{read}(\varepsilon, x) \triangleq init_x$$

$$\mathsf{mksbuff}(\mathsf{b}, t, \varepsilon) \triangleq \mathsf{b}$$

$$\mathsf{mksbuff}(\mathsf{b}, t, \pi{\cdot}\lambda) \triangleq \begin{cases} \mathsf{mksbuff}(w{\cdot}\mathsf{b}, t, \pi) & \lambda = \mathtt{lW}\langle w\rangle \wedge t(w) = t \wedge \mathsf{B}\langle w\rangle \notin \pi \\ \mathsf{mksbuff}(a{\cdot}\mathsf{b}, t, \pi) & \lambda = \mathtt{Push}\langle a\rangle \wedge \mathtt{NIC}\langle a\rangle \notin \pi \wedge t(a) = t \\ \mathsf{mksbuff}(\mathsf{b}, t, \pi) & \text{otherwise} \end{cases}$$

$$\mathsf{chkatm}(n, \pi) \triangleq \begin{cases} \bot & \forall w. \left(\begin{array}{l} \mathtt{narR}\langle \_, \_, a, \_, w\rangle \in \pi \\ \wedge\ \overline{n}(a) = n \end{array} \right) \implies \mathsf{B}\langle w\rangle \in \pi \\ \top & \text{otherwise} \end{cases}$$

$$\mathsf{mkpipe}(\mathbf{pipe}, t, \overline{n}, \varepsilon) \triangleq \mathbf{pipe}$$

$$\mathsf{mkpipe}(\mathbf{pipe}, t, \overline{n}, \pi{\cdot}\lambda) \triangleq \begin{cases} \mathsf{mkpipe}(a{\cdot}\mathbf{pipe}, t, \overline{n}, \pi) & \text{if} \left(\begin{array}{l} t(\lambda) = t \wedge\ \overline{n}(\lambda) = \overline{n} \wedge\ \lambda = \mathtt{NIC}\langle a\rangle \\ \wedge\ \mathtt{nlR}\langle \_, \_, a, \_\rangle \notin \pi \wedge\ \mathtt{nrR}\langle \_, \_, a, \_\rangle \notin \pi \\ \wedge\ \mathtt{nF}\langle a\rangle \notin \pi \wedge\ \mathtt{naF}\langle \_, \_, a, \_\rangle \notin \pi \\ \wedge\ \mathtt{narR}\langle \_, \_, a, \_, \_\rangle \notin \pi \end{array} \right) \\ \mathsf{mkpipe}(w{\cdot}\mathbf{pipe}, t, \overline{n}, \pi) & \text{if} \left(\begin{array}{l} t(\lambda) = t \wedge\ \overline{n}(\lambda) = \overline{n} \wedge\ \lambda = \mathtt{NIC}\langle a\rangle \\ \wedge\ \mathtt{nlR}\langle \_, \_, a, w\rangle \in \pi \wedge\ \mathtt{nrW}\langle w, \_\rangle \notin \pi \end{array} \right) \\ \mathsf{mkpipe}(e{\cdot}\mathbf{pipe}, t, \overline{n}, \pi) & \text{if} \left(\begin{array}{l} t(\lambda) = t \wedge\ \overline{n}(\lambda) = \overline{n} \wedge\ \lambda = \mathtt{NIC}\langle a\rangle \\ \wedge\ \mathtt{nlR}\langle \_, \_, a, w\rangle \in \pi \wedge\ \mathtt{nrW}\langle w, e\rangle \in \pi \\ \wedge\ \mathsf{CN}\langle e\rangle \notin \pi \end{array} \right) \\ \mathsf{mkpipe}(w{\cdot}\mathbf{pipe}, t, \overline{n}, \pi) & \text{if} \left(\begin{array}{l} t(\lambda) = t \wedge\ \overline{n}(\lambda) = \overline{n} \wedge\ \lambda = \mathtt{NIC}\langle a\rangle \\ \wedge\ \mathtt{nrR}\langle \_, \_, a, w\rangle \in \pi \wedge\ \mathtt{nlW}\langle w, \_\rangle \notin \pi \end{array} \right) \\ \mathsf{mkpipe}(w{\cdot}\mathbf{pipe}, t, \overline{n}, \pi) & \text{if} \left(\begin{array}{l} t(\lambda) = t \wedge\ \overline{n}(\lambda) = \overline{n} \wedge\ \lambda = \mathtt{NIC}\langle a\rangle \\ \wedge\ \mathtt{naF}\langle \_, \_, a, w\rangle \in \pi \wedge\ \mathtt{nlW}\langle w, \_\rangle \notin \pi \end{array} \right) \\ \mathsf{mkpipe}(w{\cdot}w'{\cdot}\mathbf{pipe}, t, \overline{n}, \pi) & \text{if} \left(\begin{array}{l} t(\lambda) = t \wedge\ \overline{n}(\lambda) = \overline{n} \wedge\ \lambda = \mathtt{NIC}\langle a\rangle \\ \wedge\ \mathtt{narR}\langle \_, \_, a, w, w'\rangle \in \pi \wedge\ \mathtt{narW}\langle w'\rangle \notin \pi \end{array} \right) \\ \mathsf{mkpipe}(\mathbf{pipe}, t, \overline{n}, \pi) & \text{otherwise} \end{cases}$$

$$\mathsf{mkwbR}(\mathbf{wb_R}, t, \overline{n}, \varepsilon) \triangleq \mathbf{wb_R}$$

$$\mathsf{mkwbR}(\mathbf{wb_R}, t, \overline{n}, \pi{\cdot}\lambda) \triangleq \begin{cases} \mathsf{mkwbR}(w{\cdot}\mathbf{wb_R}, t, \overline{n}, \pi) & \text{if} \left(\begin{array}{l} t(\lambda) = t \wedge \overline{n}(\lambda) = \overline{n} \wedge \mathsf{B}\langle w\rangle \notin \pi \\ \wedge\ \lambda \in \{\mathtt{nrW}\langle w, \_\rangle, \mathtt{narW}\langle w\rangle\} \end{array} \right) \\ \mathsf{mkwbR}(\mathbf{wb_R}, t, \overline{n}, \pi) & \text{otherwise} \end{cases}$$

$$\mathsf{mkwbL}(\mathbf{wb_L}, t, \overline{n}, \varepsilon) \triangleq \mathbf{wb_L}$$

$$\mathsf{mkwbL}(\mathbf{wb_L}, t, \overline{n}, \pi{\cdot}\lambda) \triangleq \begin{cases} \mathsf{mkwbL}(e{\cdot}w{\cdot}\mathbf{wb_L}, t, \overline{n}, \pi) & \text{if} \left(\begin{array}{l} t(\lambda) = t \wedge\ \overline{n}(\lambda) = \overline{n} \wedge\ \lambda = \mathtt{nlW}\langle w, e\rangle \\ \wedge\ \mathsf{B}\langle w\rangle \notin \pi \wedge \mathtt{P}\langle \_, e\rangle \notin \pi \end{array} \right) \\ \mathsf{mkwbL}(e{\cdot}\mathbf{wb_L}, t, \overline{n}, \pi) & \text{if} \left(\begin{array}{l} t(\lambda) = t \wedge\ \overline{n}(\lambda) = \overline{n} \wedge\ \lambda = \mathtt{nlW}\langle w, e\rangle \\ \wedge\ \mathsf{B}\langle w\rangle \in \pi \wedge \mathtt{P}\langle \_, e\rangle \notin \pi \end{array} \right) \\ \mathsf{mkwbL}(e{\cdot}\mathbf{wb_L}, t, \overline{n}, \pi) & \text{if} \left(\begin{array}{l} t(\lambda) = t \wedge\ \overline{n}(\lambda) = \overline{n} \wedge\ \lambda = \mathsf{CN}\langle e\rangle \\ \wedge\ \mathtt{P}\langle \_, e\rangle \notin \pi \end{array} \right) \\ \mathsf{mkwbL}(\mathbf{wb_L}, t, \overline{n}, \pi) & \text{otherwise} \end{cases}$$

Theorem 2. For all $\mathsf{P}, \mathsf{P}', \mathsf{M}, \mathsf{M}', \mathsf{B}, \mathsf{B}', \mathsf{A}, \mathsf{A}', \mathsf{QP}, \mathsf{QP}', \pi, \pi'$:

- $\mathsf{wf}(\mathsf{M}_0, \mathsf{B}_0, \mathsf{A}_0, \mathsf{QP}_0, \varepsilon)$;
- if $\mathsf{P}, \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi \Rightarrow \mathsf{P}', \mathsf{M}', \mathsf{B}', \mathsf{A}', \mathsf{QP}', \pi'$ and $\mathsf{wf}(\mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi)$ then $\mathsf{wf}(\mathsf{M}', \mathsf{B}', \mathsf{A}', \mathsf{QP}', \pi')$;
- if $\mathsf{P}, \mathsf{M}_0, \mathsf{B}_0, \mathsf{A}_0, \mathsf{QP}_0, \varepsilon \Rightarrow^* (\lambda t.\mathtt{skip}), \mathsf{M}, \mathsf{B}_0, \mathsf{A}_0, \mathsf{QP}, \pi$ such that forall $t, \overline{n}$ we have $\mathsf{QP}(t)(\overline{n}) = \langle \varepsilon, \varepsilon, \mathtt{nEX}^*\rangle$, then $\mathsf{wf}(\mathsf{M}, \mathsf{B}_0, \mathsf{A}_0, \mathsf{QP}, \pi)$ and $\mathsf{complete}(\pi)$.

The proof of the first part follows trivially from the definitions of M_0, B_0, A_0, and QP_0. The second part is proved by induction on the structure of $\Rightarrow$. The last part follows from the previous two parts and induction on the length of $\Rightarrow^*$, as well as how the definition of wf on empty store buffers and queue pairs (regardless of $\mathtt{nEX}$ in $\mathbf{wb_L}$) implies $\mathsf{complete}(\pi)$.

B.3 From Annotated Semantics to Declarative Semantics

We define

$$\mathsf{getEG}(\pi) \triangleq \begin{cases} (\mathsf{Event}, \mathsf{po}, \mathsf{rf}, \mathsf{pf}, \mathsf{mo}, \mathsf{nfo}, \mathsf{rao}) & \text{if } \mathsf{wfp}(\pi) \wedge \mathsf{complete}(\pi) \\ \text{undefined} & \text{otherwise} \end{cases}$$

with

$$\mathsf{Event} \triangleq \mathsf{Event}_0 \cup \{\mathsf{getA}(\lambda) \mid \lambda \in \pi\}$$

Recall that Event_0 is the set of initialisation events $\{init_x \mid x \in \mathsf{Loc}\}$, where $l(init_x) = \mathtt{lW}(x, 0)$

$$\mathsf{getA} : \mathsf{ALabel} \rightharpoonup \mathsf{Event}$$

$$\begin{aligned}
\mathsf{getA}(\mathtt{lR}\langle r, _\rangle) &\triangleq r & \mathsf{getA}(\mathtt{narR}\langle r, _, _, _, _\rangle) &\triangleq r \\
\mathsf{getA}(\mathtt{lW}\langle w\rangle) &\triangleq w & \mathsf{getA}(\mathtt{narW}\langle w\rangle) &\triangleq w \\
\mathsf{getA}(\mathtt{CAS}\langle u, _\rangle) &\triangleq u & \mathsf{getA}(\mathtt{P}\langle p, _\rangle) &\triangleq p \\
\mathsf{getA}(\mathtt{F}\langle f\rangle) &\triangleq f & \mathsf{getA}(\mathtt{nF}\langle f\rangle) &\triangleq f \\
\mathsf{getA}(\mathtt{nlR}\langle r, _, _, _\rangle) &\triangleq r & \mathsf{getA}(\mathsf{B}\langle w\rangle) &\triangleq w \\
\mathsf{getA}(\mathtt{nrW}\langle w\rangle) &\triangleq w & \mathsf{getA}(\mathtt{Push}\langle _\rangle) &\text{ is undefined} \\
\mathsf{getA}(\mathtt{nrR}\langle r, _, _, _\rangle) &\triangleq r & \mathsf{getA}(\mathtt{NIC}\langle _\rangle) &\text{ is undefined} \\
\mathsf{getA}(\mathtt{nlW}\langle w, _\rangle) &\triangleq w & \mathsf{getA}(\mathsf{CN}\langle _\rangle) &\text{ is undefined} \\
\mathsf{getA}(\mathtt{naF}\langle r, _, _, _\rangle) &\triangleq r & \mathsf{getA}(\mathcal{E}\langle _\rangle) &\text{ is undefined}
\end{aligned}$$

We define $\mathsf{getI}\lambda(\_, \pi)$ and $\mathsf{getO}\lambda(\_, \pi)$ to perform the reverse operation of getA. In the case of write events, $\mathsf{getI}\lambda(\_, \pi)$ retrieves the first label sending the write to the buffer, while $\mathsf{getO}\lambda(\_, \pi)$ retrieves the second label committing the write to memory.

$$\mathsf{getI}\lambda(\_, \pi), \mathsf{getO}\lambda(\_, \pi) : \{\mathsf{getA}(\lambda) \mid \lambda \in \pi\} \to \mathsf{ALabel}$$

For all $\lambda \in \pi$:

- if $\mathsf{type}(\lambda) \in \{\mathtt{lR}, \mathtt{CAS}, \mathtt{F}, \mathtt{P}, \mathtt{nlR}, \mathtt{nrR}, \mathtt{narR}, \mathtt{naF}, \mathtt{nF}\}$,
  then $\mathsf{getI}\lambda(\mathsf{getA}(\lambda), \pi) \triangleq \mathsf{getO}\lambda(\mathsf{getA}(\lambda), \pi) \triangleq \lambda$;
- if $\mathsf{type}(\lambda) \in \{\mathtt{lW}, \mathtt{nlW}, \mathtt{nrW}, \mathtt{narW}\}$,
  then $\mathsf{getI}\lambda(\mathsf{getA}(\lambda), \pi) \triangleq \lambda$ while $\mathsf{getO}\lambda(\mathsf{getA}(\lambda), \pi) \triangleq \mathsf{B}\langle\lambda\rangle$.
- if $\lambda = \mathsf{B}\langle w\rangle$, then from $\mathsf{backComp}(\pi)$ there is $\lambda' \prec_\pi \lambda$ such that $\mathsf{type}(\lambda) \in \{\mathtt{lW}, \mathtt{nlW}, \mathtt{nrW}, \mathtt{narW}\}$ and $\mathsf{getA}(\lambda') = \mathsf{getA}(\lambda) = w$. From the previous case, we have $\mathsf{getI}\lambda(w, \pi) \triangleq \lambda'$ and $\mathsf{getO}\lambda(w, \pi) \triangleq \lambda$.

From this we define two relations IB and OB on Event total on all meaningful events by copying the ordering in π.

$$\mathsf{IB} \triangleq \{(e_1, e_2) \mid \mathsf{getI}\lambda(e_1, \pi) \prec_\pi \mathsf{getI}\lambda(e_2, \pi)\} \cup (\mathsf{Event}_0 \times (\mathsf{Event} \setminus \mathsf{Event}_0))$$

$$\mathsf{OB} \triangleq \{(e_1, e_2) \mid \mathsf{getO}\lambda(e_1, \pi) \prec_\pi \mathsf{getO}\lambda(e_2, \pi)\} \cup (\mathsf{Event}_0 \times (\mathsf{Event} \setminus \mathsf{Event}_0))$$

From $\mathsf{wfp}(\pi)$, IB and OB are transitive and irreflexive. Note: we could make IB and OB total by adding an arbitrary total order on Event_0.

$$\mathsf{rf} \triangleq \left\{ (w, r) \;\middle|\; \begin{array}{l} \mathtt{lR}\langle r, w\rangle \in \pi \vee \mathtt{nlR}\langle r, w, \_, \_\rangle \in \pi \vee \mathtt{nrR}\langle r, w, \_, \_\rangle \in \pi \vee \mathtt{CAS}\langle r, w\rangle \in \pi \\ \vee\ \mathtt{narR}\langle r, w, \_, \_, \_\rangle \in \pi \vee \mathtt{naF}\langle r, w, \_, \_\rangle \in \pi \end{array} \right\}$$

$$\mathsf{pf} \triangleq \left\{ (w, p) \;\middle|\; \begin{array}{l} \mathtt{nlW}\langle w, e\rangle \prec_\pi \mathtt{P}\langle p, e\rangle \\ \vee\ \mathtt{nrW}\langle w, e\rangle \prec_\pi \mathtt{P}\langle p, e\rangle \end{array} \right\}$$

$$\lambda \text{ generates } e \text{ in } \pi \triangleq \left(\begin{array}{l} \lambda \in \{\mathtt{lR}\langle e, \_\rangle, \mathtt{lW}\langle e\rangle, \mathtt{CAS}\langle e, \_\rangle, \mathtt{Push}\langle e\rangle, \mathtt{P}\langle e, \_\rangle, \mathtt{F}\langle e\rangle\} \\ \vee\ \lambda = \mathtt{Push}\langle a\rangle \wedge \left(\begin{array}{l} \lambda \prec_\pi \mathtt{nlR}\langle e, \_, a, \_\rangle \\ \vee\ \lambda \prec_\pi \mathtt{nlR}\langle \_, \_, a, e\rangle \\ \vee\ \lambda \prec_\pi \mathtt{nrR}\langle e, \_, a, \_\rangle \\ \vee\ \lambda \prec_\pi \mathtt{nrR}\langle \_, \_, a, e\rangle \\ \vee\ \lambda \prec_\pi \mathtt{naF}\langle e, \_, a, \_\rangle \\ \vee\ \lambda \prec_\pi \mathtt{naF}\langle \_, \_, a, e\rangle \\ \vee\ \lambda \prec_\pi \mathtt{narR}\langle e, \_, a, \_, \_\rangle \\ \vee\ \lambda \prec_\pi \mathtt{narR}\langle \_, \_, a, e, \_\rangle \\ \vee\ \lambda \prec_\pi \mathtt{narR}\langle \_, \_, a, \_, e\rangle \end{array} \right) \end{array} \right)$$

$$\mathsf{po} \triangleq \left(\begin{array}{l} \mathsf{Event}_0 \times (\mathsf{Event} \setminus \mathsf{Event}_0) \\ \cup \left\{ (e_1, e_2) \;\middle|\; \begin{array}{l} \lambda_1 \prec_\pi \lambda_2 \wedge t(\lambda_1) = t(\lambda_2) \\ \wedge\ \lambda_1 \text{ generates } e_1 \text{ in } \pi \\ \wedge\ \lambda_2 \text{ generates } e_2 \text{ in } \pi \end{array} \right\} \\ \cup \left\{ (r, w) \;\middle|\; \begin{array}{l} \mathtt{nlR}\langle r, \_, \_, w\rangle \in \pi \\ \vee\ \mathtt{nrR}\langle r, \_, \_, w\rangle \in \pi \\ \vee\ \mathtt{naF}\langle r, \_, \_, w\rangle \in \pi \\ \vee\ \mathtt{narR}\langle r, \_, \_, \_, w\rangle \in \pi \end{array} \right\} \\ \cup \{(w_1, w_2) \mid \mathtt{narR}\langle \_, \_, \_, w_2, w_1\rangle\} \in \pi \end{array} \right)$$

$$\mathsf{mo} \triangleq \left\{ (w_1, w_2) \;\middle|\; \begin{array}{l} w_1 = init_x \\ \wedge\ (\mathtt{B}\langle w_2\rangle \in \pi \vee \mathtt{CAS}\langle w_2, \_\rangle \in \pi) \\ \wedge\ \mathtt{loc}(w_1) = x = \mathtt{loc}(w_2) \end{array} \right\} \cup \left\{ (w_1, w_2) \;\middle|\; \begin{array}{l} \lambda_1 \prec_\pi \lambda_2 \\ \wedge\ \lambda_1 \in \{\mathtt{B}\langle w_1\rangle, \mathtt{CAS}\langle w_1, \_\rangle\} \\ \wedge\ \lambda_2 \in \{\mathtt{B}\langle w_2\rangle, \mathtt{CAS}\langle w_2, \_\rangle\} \\ \wedge\ \mathtt{loc}(w_1) = \mathtt{loc}(w_2) \end{array} \right\}$$

$$\mathsf{nfo} \triangleq \left(\begin{array}{l} \{(r, w) \mid \mathsf{sameqp}(r, w) \wedge \mathtt{nlR}\langle r, \_, \_, \_\rangle \prec_\pi \mathtt{nlW}\langle w, \_\rangle \prec_\pi \mathtt{B}\langle w\rangle\} \\ \cup\ \{(r, w) \mid \exists \lambda_r, \lambda_w. \mathsf{sameqp}(r, w) \wedge \lambda_r \prec_\pi \lambda_w \prec_\pi \mathtt{B}\langle w\rangle\} \\ \cup\ \{(w, r) \mid \mathsf{sameqp}(w, r) \wedge \mathtt{nlW}\langle w, \_\rangle \prec_\pi \mathtt{B}\langle w\rangle \prec_\pi \mathtt{nlR}\langle r, \_, \_, \_\rangle\} \\ \cup\ \{(w, r) \mid \exists \lambda_r, \lambda_w. \mathsf{sameqp}(w, r) \wedge \lambda_w \prec_\pi \mathtt{B}\langle w\rangle \prec_\pi \lambda_r\} \end{array} \right)$$

where $\lambda_r \in \{\mathtt{nrR}\langle r', \ldots\rangle, \mathtt{naF}\langle r', \ldots\rangle, \mathtt{narR}\langle r', \ldots\rangle\}$

$\lambda_w \in \{\mathtt{nrW}\langle w, \_\rangle, \mathtt{narW}\langle w\rangle\}$

$$\mathsf{rao} \triangleq \left(\left\{ (r_1, r_2) \;\middle|\; \overline{n}(a_1) = \overline{n}(a_2) \wedge \left(\begin{array}{c} \lambda_1 \prec_\pi \lambda_2 \\ \wedge\ \lambda_1 \in \{\mathtt{naF}\langle r_1, a_1, \ldots\rangle, \mathtt{narR}\langle r_1, a_1, \ldots\rangle\} \\ \wedge\ \lambda_2 \in \{\mathtt{naF}\langle r_2, a_2, \ldots\rangle, \mathtt{narR}\langle r_2, a_2, \ldots\rangle\} \end{array} \right) \right\} \right)$$

From an execution graph $E = \mathsf{getEG}(\pi)$, we use the definitions of the paper to define oppo, ippo, rf_{b}, $\mathsf{rf}_{\overline{\mathsf{b}}}$, rb, rb_{b}, ar, ob, and ib.

Lemma 1. $w \in \mathtt{nlW} \implies \exists r. \left(\begin{array}{c} r \in \mathtt{nrR} \wedge (r, w) \in \mathsf{po}|_{\mathrm{imm}} \\ \vee\ r \in \mathtt{narR} \wedge (r, w) \in \mathsf{po}|_{\mathrm{imm}} \cup (\mathsf{po}|_{\mathrm{imm}})^2 \end{array} \right)$

Proof. By definition of po, we can only have such $w \in \mathtt{nlW}$ if there is some $\lambda = \mathtt{Push}\langle a\rangle$ which generates w in π. Then we can consider the cases of a such that $\mathtt{Push}\langle a\rangle$ generates some $w \in \mathtt{nlW}$. Either:

- $a \in \mathtt{Put}$, then there is some $r \in \mathtt{nrR}$ with $(r, w) \in \mathsf{po}|_{\mathrm{imm}}$
- $a \in \mathtt{nCAS} \cup \mathtt{nFAA}$, then there is some $r \in \mathtt{narR}$ with either $(r, w) \in \mathsf{po}|_{\mathrm{imm}}$ (in the case of a failed $\mathtt{nCAS}$) or $(r, w) \in (\mathsf{po}|_{\mathrm{imm}})^2$ (in the case of a successful $\mathtt{nCAS}$ or $\mathtt{nFAA}$)

□

Theorem 3. $\mathsf{getEG}(\pi)$ is well-formed.

Proof. We need to check the conditions of a pre-execution (Def. 2) and of well-formedness (Def. 3). For the pre-execution conditions:

- Checking $\mathsf{Event}^0 \times (\mathsf{Event} \setminus \mathsf{Event}^0) \subseteq \mathsf{po}$:

  by definition.

- Checking po is a union of strict partial orders each on one thread:

  If $t(\mathsf{e}_1) \neq t(\mathsf{e}_2)$, then $(\mathsf{e}_1, \mathsf{e}_2) \notin \mathsf{po}$ and $(\mathsf{e}_2, \mathsf{e}_1) \notin \mathsf{po}$ by definition. If $t(\mathsf{e}_1) = t(\mathsf{e}_2)$, then either $(\mathsf{e}_1, \mathsf{e}_2) \in \mathsf{po}$ or $(\mathsf{e}_2, \mathsf{e}_1) \in \mathsf{po}$. This comes from the second case of the definition of po: if there is λ_1 and λ_2 such that λ_i generates e_i in π, then either $\lambda_1 \prec_\pi \lambda_2$ or $\lambda_2 \prec_\pi \lambda_1$.

- Checking that rf is functional on its range:

  If $r \in \mathcal{R} \subseteq \{\mathsf{getA}(\lambda) \mid \lambda \in \pi\}$, then we have either $\mathtt{lR}\langle r, \_\rangle$, $\mathtt{nlR}\langle r, \_, \_, \_\rangle$, $\mathtt{nrR}\langle r, \_, \_, \_\rangle$, $\mathtt{naF}\langle r, \_, \_, \_\rangle$, or $\mathtt{narR}\langle r, \_, \_, \_, \_\rangle$ in π, and r have at least one antecedent.

  If $(w, r) \in \mathsf{rf}$, let us assume $r \in \mathtt{nlR}$, then by definition $\mathtt{nlR}\langle r, w, \_, \_\rangle \in \pi$. Since $\mathsf{nodup}(\pi)$, for all $w' \neq w$, we have $\mathtt{nlR}\langle r, w', \_, \_\rangle \notin \pi$, and syntactically we cannot write $\mathtt{lR}\langle r, \_\rangle$ or $\mathtt{nrR}\langle r, \_, \_, \_\rangle$, so $(w', r) \notin \mathsf{rf}$. Similarly, $r \in \mathtt{lR}$, $r \in \mathtt{nrR}$, $r \in \mathtt{naF}$ or $r \in \mathtt{narR}$ only have one antecedent.

- Checking that rf relates events on the same location with matching values:

  By syntactic definition of the annotated labels $\mathtt{lR}$, $\mathtt{nlR}$, $\mathtt{nrR}$, $\mathtt{naF}$ and $\mathtt{narR}$, e.g., $\mathtt{lR}\langle r, w\rangle \implies \mathsf{eq}_{\mathtt{loc\&v}}(r, w)$.

- Checking that mo is a union of strict total orders for writes on each variables:

  By definition of mo, given that we have $\mathsf{complete}(\pi)$, e.g., if $\mathtt{lW}\langle w\rangle \in \pi$ then $\mathsf{B}\langle w\rangle \in \pi$.

- Checking that $\mathsf{pf} \subseteq \mathsf{po} \cap \mathtt{sqp}$:

  If $(w, p) \in \mathsf{pf}$ with $w \in \mathtt{nlW}$ (resp. $\mathtt{nrW}$), then we have $\mathtt{nlW}\langle w, e\rangle \prec_\pi \mathtt{P}\langle p, e\rangle$. There is λ such that λ generates w in π, and we have $\lambda \prec_\pi \mathtt{nlW}\langle w, e\rangle \prec_\pi \mathtt{P}\langle p, e\rangle$. Also, $t(p) = t(w)$ and $\overline{n}(p) = \overline{n}(e) = \overline{n}(w)$, so we have $(w, p) \in \mathsf{po}$ and $(w, p) \in \mathtt{sqp}$.

- Checking that pf is functional on its domain:

  If $(w, p) \in \mathsf{pf}$ with $w \in \mathtt{nlW}$ (resp. $\mathtt{nrW}$), then we have $\mathtt{nlW}\langle w, e\rangle \prec_\pi \mathtt{P}\langle p, e\rangle$. From $\mathsf{nodup}(\pi)$, for all $p' \neq p$ we have $\mathtt{P}\langle p, e\rangle \notin \pi$, so w has at most one image.

- Checking that pf is total and functional on its range:

  If $p \in \mathsf{Event}$, then there is $e \in \mathtt{nlEX}$ (resp. $\mathtt{nrEX}$) such that $\mathtt{P}\langle p, e\rangle \in \pi$. From $\mathsf{backComp}(\pi)$ there is $w \in \mathtt{nlW}$ (resp. $\mathtt{nrW}$) such that $\mathtt{nlW}\langle w, e\rangle \prec_\pi \mathtt{P}\langle p, e\rangle$, and so $(w, p) \in \mathsf{pf}$. From $\mathsf{nodup}(\pi)$, e cannot be used in another $\mathtt{nlW}$ (resp. $\mathtt{nrW}$) annotated label, and p has exactly one antecedent.

- Checking that for all $(a, b) \in \mathtt{sqp}$, $a \in \mathtt{nrR} \cup \mathtt{naF} \cup \mathtt{narR}$, $b \in \mathtt{nrW} \cup \mathtt{narW}$, (resp. $\mathtt{nlR}/\mathtt{nlW}$) then $(a, b) \in \mathsf{nfo} \cup \mathsf{nfo}^{-1}$:

  By definition of nfo, given that $\mathsf{bufFlushOrd}(\pi)$ forbids such interleavings as $\mathtt{nrW}\langle w, \_\rangle \prec_\pi \mathtt{nrR}\langle r, \_, \_, \_\rangle \prec_\pi \mathsf{B}\langle w\rangle$ (resp. $\mathtt{nlW}$ and $\mathtt{nlR}$) when $\mathsf{sameqp}(r, w)$.

- Checking that rao is a union of strict total orders for remote atomic reads:

  By definition of rao.

For the well-formedness conditions:

(1) Let us assume $(w_1, w_2) \in \mathsf{po} \cap \mathtt{sqp}$ and $(w_2, p_2) \in \mathsf{pf}$. The three events are on the same thread and queue pair.

  If $w_1 \in \mathtt{nlW}$, then by $\mathsf{complete}(\pi)$ there is a chain $\mathtt{Push}\langle a_1\rangle \prec_\pi \mathtt{NIC}\langle a_1\rangle \prec_\pi \mathtt{nR} \prec_\pi \mathtt{nlW}\langle w_1, e_1\rangle$ for some $\mathtt{nR} \in \{\mathtt{nrR}\langle \_, \_, a_1, w_1\rangle, \mathtt{naF}\langle \_, \_, a_1, w_1\rangle, \mathtt{narR}\langle \_, \_, a_1, w_1, \_\rangle\}$; if $w_1 \in \mathtt{nrW}$, there is instead a chain $\mathtt{Push}\langle a_1\rangle \prec_\pi \mathtt{NIC}\langle a_1\rangle \prec_\pi \mathtt{nlR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nrW}\langle w_1, e_1\rangle \prec_\pi \mathsf{CN}\langle e_1\rangle$. Similarly there is a chain for w_2. By $(w_1, w_2) \in \mathsf{po}$ we have $\mathtt{Push}\langle a_1\rangle \prec_\pi \mathtt{Push}\langle a_2\rangle$, and by $\mathsf{bufFlushOrd}(\pi)$ we have $\mathtt{NIC}\langle a_1\rangle \prec_\pi \mathtt{NIC}\langle a_2\rangle$.

  Let us call λ_1 the last annotated label on the chain for w_1, i.e., either $\mathtt{nlW}\langle w_1, e_1\rangle$ or $\mathsf{CN}\langle e_1\rangle$. Similarly, λ_2 is the last annotated label on the chain for w_2. There are four cases to consider, but in all four $\mathsf{nicActOrder}(\pi)$ implies $\lambda_1 \prec_\pi \lambda_2$.

  Then, from $\mathsf{pollOrder}(\pi)$, there is p_1 such that $\mathtt{P}\langle p_1, e_1\rangle \prec_\pi \mathtt{P}\langle p_2, e_2\rangle$. By definitions, we have both $(w_1, p_1) \in \mathsf{pf}$ and $(p_1, p_2) \in \mathsf{po}$.

(2) If $r \in \mathtt{nlR}$, then there is $w \in \mathtt{nrW}$ (taken from $\mathtt{nlR}\langle r, \_, \_, w\rangle$) such that $(r, w) \in \mathsf{po}|_{\mathrm{imm}}$. This is by the last case of definition of po, since there is λ_a such that we have both λ_a generates r in π and λ_a generates w in π.
  Similarly for $\mathtt{nrR}/\mathtt{nlW}$ and $\mathtt{nrW}/\mathtt{nlR}$.

(3) If $(r, w) \in \mathsf{po}|_{\mathrm{imm}}$, $\mathsf{type}(r) \in \{\mathtt{nlR}, \mathtt{nrR}\}$, and $\mathsf{type}(w) \in \{\mathtt{nlW}, \mathtt{nrW}\}$, then $(r, w) \in \mathsf{po}$ comes from the third case of the definition of po, and we have either $\mathtt{nlR}\langle r, \_, \_, w\rangle$ or $\mathtt{nrR}\langle r, \_, \_, w\rangle$ in π. In both cases, we have $v_{\mathrm{r}}(r) = v_{\mathrm{w}}(w)$ by syntactic definition of the annotated labels.

(4) (a) If $r \in \texttt{narR}$, then either: There is $\texttt{naF}\langle r, \_, \_, w\rangle \in \pi$ such that $w \in \texttt{nlW}$ and $(r, w) \in \mathsf{po}|_{\text{imm}}$. This follows from the second case definition of po. There is $\texttt{narR}\langle r, \_, \_, w_2, w_1\rangle \in \pi$ such that $w_1 \in \texttt{narW}$, $w_2 \in \texttt{nlW}$, and $(r, w_1), (w_1, w_2) \in \mathsf{po}|_{\text{imm}}$. This follows from the second and third cases of the definition of po since there is λ_a which generates r, w_1 and w_2 in π. (b) If $w \in \texttt{narW}$ then $(r, w), (w, w') \in \mathsf{po}|_{\text{imm}}$ with $r \in \texttt{narR}$ and $w' \in \texttt{nlW}$ comes from the second case definition of po.

(5) If $(r, w) \in G.\mathsf{po}|_{\text{imm}}$, $\mathsf{type}(r) = \texttt{narR}$ and $\mathsf{type}(w) = \texttt{nlW}$, then (r, w) comes from the second case definition of po and we have $\texttt{naF}\langle r, \_, \_, w\rangle \in \pi$. Then $v_{\text{r}}(r) = v_{\text{w}}(w)$ by the syntax of annotated labels. If $(r, w_1), (w_1, w_2) \in G.po|_{\text{imm}}$, $\mathsf{type}(r) = \texttt{narR}$, $\mathsf{type}(w_1) = \texttt{narW}$ and $\mathsf{type}(w_2) = \texttt{nlW}$, then (r, w_1) comes from the second case definition of po and (w_1, w_2) from the third case, so we have $\texttt{narR}\langle r, \_, \_, w_2, w_1\rangle \in \pi$. Then $v_{\text{r}}(r) = v_{\text{w}}(w_2)$ by the syntax of annotated labels.

(6) Comes from lemma 1.

□

Lemma 2. $\mathsf{OB}; [\texttt{Inst}] \subseteq \mathsf{IB}$ and $[\texttt{Inst}]; \mathsf{IB} \subseteq \mathsf{OB}$.

Proof. If $(e_1, e_2) \in \mathsf{OB}; [\texttt{Inst}]$, then $\mathsf{getO}\lambda(e_1, \pi) \prec_\pi \mathsf{getO}\lambda(e_2, \pi) = \mathsf{getI}\lambda(e_2, \pi)$.

- If $e_1 \in \texttt{Inst}$, then $\mathsf{getO}\lambda(e_1, \pi) = \mathsf{getI}\lambda(e_1, \pi)$, so we have $\mathsf{getI}\lambda(e_1, \pi) \prec_\pi \mathsf{getI}\lambda(e_2, \pi)$ and $(e_1, e_2) \in \mathsf{IB}$.
- If $e_1 \in \{\texttt{lW}, \texttt{nlW}, \texttt{nrW}, \texttt{narW}\}$, there is λ such that $\mathsf{type}(\lambda) \in \{\texttt{lW}, \texttt{nlW}, \texttt{nrW}, \texttt{narW}\}$, $\mathsf{getA}(\lambda) = e_1$, and $\mathsf{getI}\lambda(e_1, \pi) = \lambda \prec_\pi \mathsf{B}\langle e_1\rangle = \mathsf{getO}\lambda(e_1, \pi)$. By transitivity we again have $\mathsf{getI}\lambda(e_1, \pi) \prec_\pi \mathsf{getI}\lambda(e_2, \pi)$ and $(e_1, e_2) \in \mathsf{IB}$.

With a similar reasoning, we can see that $[\texttt{Inst}]; \mathsf{IB} \subseteq \mathsf{OB}$.

□

Theorem 4. $\mathsf{getEG}(\pi)$ is consistent.

Proof. From Definition 4, we need to check that both ib and ob are irreflexive. Since IB and OB are irreflexive, it is enough to show that $\mathsf{ib} \subseteq \mathsf{IB}$ and $\mathsf{ob} \subseteq \mathsf{OB}$.

The explicit definition using limits is the following (where $\mathsf{rf}_{\overline{\mathsf{b}}} \triangleq (\mathsf{rf} \setminus \mathsf{rf}_{\mathsf{b}})$ includes $(\mathsf{rf} \cap \texttt{sqp})$ since we assume the PCIe guarantees hold):

$$\begin{aligned}
\mathsf{ib}^0 &\triangleq (\mathsf{ippo} \cup \mathsf{rf} \cup \mathsf{pf} \cup \mathsf{rb}_{\mathsf{b}} \cup \mathsf{nfo})^+ \\
\mathsf{ob}^0 &\triangleq (\mathsf{oppo} \cup \mathsf{rf}_{\overline{\mathsf{b}}} \cup [\texttt{nlW}]; \mathsf{pf} \cup \mathsf{rb} \cup \mathsf{nfo} \cup \mathsf{mo} \cup \mathsf{ar}; \mathsf{rao})^+ \\
\mathsf{ib}^{n+1} &\triangleq (\mathsf{ib}^n \cup \mathsf{ob}^n; [\texttt{Inst}])^+ \\
\mathsf{ob}^{n+1} &\triangleq (\mathsf{ob}^n \cup [\texttt{Inst}]; \mathsf{ib}^n)^+ \\
\mathsf{ib} &\triangleq \lim_{n\to\infty} \mathsf{ib}^n \\
\mathsf{ob} &\triangleq \lim_{n\to\infty} \mathsf{ob}^n
\end{aligned}$$

It is then enough to show that $\mathsf{ib}^0 \subseteq \mathsf{IB}$ and $\mathsf{ob}^0 \subseteq \mathsf{OB}$. Using Lemma 2 above, we can check the induction case:

$$\begin{aligned}
\mathsf{ib}^{n+1} &= (\mathsf{ib}^n \cup \mathsf{ob}^n; [\texttt{Inst}])^+ \subseteq (\mathsf{ib}^n \cup \mathsf{OB}; [\texttt{Inst}])^+ \subseteq (\mathsf{IB} \cup \mathsf{IB})^+ = \mathsf{IB} \\
\mathsf{ob}^{n+1} &= (\mathsf{ob}^n \cup [\texttt{Inst}]; \mathsf{ib}^n)^+ \subseteq (\mathsf{ob}^n \cup [\texttt{Inst}]; \mathsf{IB})^+ \subseteq (\mathsf{OB} \cup \mathsf{OB})^+ = \mathsf{OB}
\end{aligned}$$

Since IB and OB are transitive, we need to check the components of ib^0 and ob^0. There are twelve cases to verify.

- Checking ippo $\subseteq$ IB.
  Let $E^{\text{cpu}} = \{\texttt{lR}, \texttt{lW}, \texttt{CAS}, \texttt{F}, \texttt{P}\}$ and $E^{\text{nic}} = \{\texttt{nlR}, \texttt{nrR}, \texttt{narR}, \texttt{naF}, \texttt{nlW}, \texttt{nrW}, \texttt{narW}, \texttt{nF}\}$. $[E^{\text{cpu}}]; \mathsf{po} \subseteq \mathsf{IB}$ by definition of po and IB: E^{cpu} are the events for which the same annotated label is used to define po and IB, i.e., $\forall e \in E^{\text{cpu}}$, $\mathsf{getl}\lambda(e, \pi)$ generates e in π. To check that $[E^{\text{nic}}]; \mathsf{ippo}; [E^{\text{nic}}] \subseteq \mathsf{IB}$, there are 36 cases to consider. They are all trivially satisfied by $\mathsf{nicActOrder}(\pi)$ and $\mathsf{backComp}(\pi)$.

- Checking oppo $\subseteq$ OB.
  From above we have $[\texttt{Inst}]; \mathsf{oppo} \subseteq [\texttt{Inst}]; \mathsf{ippo} \subseteq [\texttt{Inst}]; \mathsf{IB} \subseteq \mathsf{OB}$.
  $[\texttt{lW}]; \mathsf{po}; [\mathsf{Event} \setminus (\texttt{lR} \cup \texttt{P})] \subseteq \mathsf{OB}$ by using $\mathsf{bufFlushOrd}(\pi)$.
  For the remaining cases:

  (G7) $[\texttt{nrW}]; (\mathsf{po} \cap \texttt{sqp}); [\texttt{nrW}] \subseteq \mathsf{OB}$ comes from $\mathsf{nicActOrder}(\pi)$ (i.e., $\texttt{nrW}\langle\ldots\rangle \prec_\pi \texttt{nrW}\langle\ldots\rangle$) and $\mathsf{bufFlushOrd}(\pi)$ (i.e., $\mathsf{B}\langle\ldots\rangle \prec_\pi \mathsf{B}\langle\ldots\rangle$).

  (G8) $[\texttt{nrW}]; (\mathsf{po} \cap \texttt{sqp}); [\texttt{narR}] \subseteq \mathsf{OB}$ comes from $\mathsf{nicActOrder}(\pi)$ (i.e., $\texttt{nrW}\langle\ldots\rangle \prec_\pi \texttt{narR}\langle\ldots\rangle$) and $\mathsf{bufFlushOrd}(\pi)$ (i.e., $\mathsf{B}\langle\ldots\rangle \prec_\pi \texttt{narR}\langle\ldots\rangle$).

  (G9) If $e_1 \in \texttt{nrW}$, $e_3 \in \texttt{narW}$, and $(e_1, e_3) \in (\mathsf{po} \cap \texttt{sqp})$, then from Def. 3 there is $e_2 \in \texttt{narR}$ such that $(e_2, e_3) \in \mathsf{po}|_{\text{imm}}$ and thus $(e_1, e_2) \in (\mathsf{po} \cap \texttt{sqp})$. From case G8 above, we have $(e_1, e_2) \in \mathsf{OB}$. From $\mathsf{backComp}(\pi)$, we have $(e_2, e_3) \in [\texttt{Inst}]; \mathsf{IB} \subseteq \mathsf{OB}$. Thus $[\texttt{nrW}]; (\mathsf{po} \cap \texttt{sqp}); [\texttt{narW}] \subseteq \mathsf{OB}$.

  (G10) $[\texttt{nrW}]; (\mathsf{po} \cap \texttt{sqp}); [\texttt{nrR}] \subseteq \mathsf{OB}$ comes from $\mathsf{nicActOrder}(\pi)$ (i.e., $\texttt{nrW}\langle\ldots\rangle \prec_\pi \texttt{nrR}\langle\ldots\rangle$) and $\mathsf{bufFlushOrd}(\pi)$ (i.e., $\texttt{nrW}\langle\ldots\rangle \prec_\pi \mathsf{B}\langle\ldots\rangle \prec_\pi \texttt{nrR}\langle\ldots\rangle$).

  (G11) If $e_1 \in \texttt{nrW}$, $e_3 \in \texttt{nlW}$, and $(e_1, e_3) \in (\mathsf{po} \cap \texttt{sqp})$, then from Def. 3 there is $e_2 \in (\texttt{narR} \cup \texttt{nrR})$ such that $(e_2, e_3) \in \mathsf{po}|_{\text{imm}}^{\{1,2\}}$ and thus $(e_1, e_2) \in (\mathsf{po} \cap \texttt{sqp})$. Then $(e_1, e_2) \subseteq \mathsf{OB}$ comes from cases G9 and G10 respectively. From $\mathsf{backComp}(\pi)$, we have $(e_2, e_3) \in [\texttt{Inst}]; \mathsf{IB} \subseteq \mathsf{OB}$. Thus $[\texttt{nrW}]; (\mathsf{po} \cap \texttt{sqp}); [\texttt{nlW}] \subseteq \mathsf{OB}$.

  (I7) $[\texttt{narW}]; (\mathsf{po} \cap \texttt{sqp}); [\texttt{nrW}] \subseteq \mathsf{OB}$ comes from $\mathsf{nicActOrder}(\pi)$ (i.e., $\texttt{narW}\langle\ldots\rangle \prec_\pi \texttt{nrW}\langle\ldots\rangle$) and $\mathsf{bufFlushOrd}(\pi)$ (i.e., $\mathsf{B}\langle\ldots\rangle \prec_\pi \mathsf{B}\langle\ldots\rangle$).

  (I8) $[\texttt{narW}]; (\mathsf{po} \cap \texttt{sqp}); [\texttt{narR}] \subseteq \mathsf{OB}$ follows from Def. 3, $\mathsf{nicActOrder}(\pi)$ and $\mathsf{bufFlushOrd}(\pi)$ by similar reasoning to I7.

  (I9) $[\texttt{narW}]; (\mathsf{po} \cap \texttt{sqp}); [\texttt{narW}] \subseteq \mathsf{OB}$ follows similarly to I7.

  (I10) $[\texttt{narW}]; (\mathsf{po} \cap \texttt{sqp}); [\texttt{nrR}] \subseteq \mathsf{OB}$ follows similarly to I7.

  (K11) $[\texttt{nlW}]; (\mathsf{po} \cap \texttt{sqp}); [\texttt{nlW}] \subseteq \mathsf{OB}$ comes from $\mathsf{nicActOrder}(\pi)$ (i.e., $\texttt{nlW}\langle\ldots\rangle \prec_\pi \texttt{nlW}\langle\ldots\rangle$) and $\mathsf{bufFlushOrd}(\pi)$ (i.e., $\mathsf{B}\langle\ldots\rangle \prec_\pi \mathsf{B}\langle\ldots\rangle$).

- Checking $\mathsf{rf}_{\overline{\mathsf{b}}} \subseteq \mathsf{OB}$.
  If $(w, r) \in \mathsf{rf}_{\overline{\mathsf{b}}}$, there is π_1 and π_2 such that $\pi = \pi_2 \cdot \mathsf{getO}\lambda(r, \pi) \cdot \pi_1$, and we use $\mathsf{wfrd}(\pi)$.

  – If $r \in \texttt{lR}$, we have $\mathsf{wfrdCPU}(r, w, \pi_1)$. The definition allow for three different cases. In the first case, $\lambda \in \{\mathsf{B}\langle w\rangle, \texttt{CAS}\langle w, \_\rangle\}$ is in π_1; we have $\lambda = \mathsf{getO}\lambda(w, \pi) \prec_\pi \mathsf{getO}\lambda(r, \pi)$ and so $(w, r) \in \mathsf{OB}$. In the second case, we have $\lambda = \texttt{lW}\langle w\rangle$ and $t(w) = t(r)$; so $(w, r) \in [\texttt{lW}]; (\mathsf{rf} \cap \mathsf{sthd}); [\texttt{lR}] = \mathsf{rf}_{\mathsf{b}}$, which contradicts $(w, r) \in \mathsf{rf}_{\overline{\mathsf{b}}} = \mathsf{rf} \setminus \mathsf{rf}_{\mathsf{b}}$. In the third case, $w = init_x$ for some location x, so $(w, r) \in \mathsf{Event}_0 \times (\mathsf{Event} \setminus \mathsf{Event}_0) \subseteq \mathsf{OB}$.

  - If $r \in \texttt{CAS}$, similarly to above, except the second case of $\mathsf{wfrdCPU}(r, w, \pi_1)$ is not possible because of $\mathsf{bufFlushOrd}(\pi)$: $\mathsf{B}\langle w\rangle \notin \pi_1$ while $\texttt{CAS}$ acts as a memory fence.
  - If $r \in \texttt{nlR}$, we have $\mathsf{wfrdNIC}(r, w, \pi_1)$, with two possibilities. In the first case, $\lambda \in \{\mathsf{B}\langle w\rangle, \texttt{CAS}\langle w, \_\rangle\}$ is in π_1; we have $\lambda = \mathsf{getO}\lambda(w, \pi) \prec_\pi \mathsf{getO}\lambda(r, \pi)$ and so $(w, r) \in \mathsf{OB}$. In the second case, $w = init_x$ for some location x, so $(w, r) \in \mathsf{Event}_0 \times (\mathsf{Event} \setminus \mathsf{Event}_0) \subseteq \mathsf{OB}$.
  - If $r \in \texttt{nrR}$ or $\texttt{narR}$, similarly to above.

- Checking $\mathsf{rf} \subseteq \mathsf{IB}$.
  From above we have $\mathsf{rf}_{\overline{\mathsf{b}}} = \mathsf{rf}_{\overline{\mathsf{b}}}; [\texttt{Inst}] \subseteq \mathsf{OB}; [\texttt{Inst}] \subseteq \mathsf{IB}$.
  If $(w, r) \in \mathsf{rf}_{\mathsf{b}} \subseteq [\texttt{lW}]; \mathsf{rf}; [\texttt{lR}]$, then there is $\texttt{lR}\langle r, w\rangle \in \pi$. There is π_1 and π_2 such that $\pi = \pi_2 \cdot \texttt{lR}\langle r, w\rangle \cdot \pi_1$. So by $\mathsf{wfrd}(\pi)$ we have $\mathsf{wfrdCPU}(r, w, \pi_1)$ which implies $\texttt{lW}\langle w\rangle \prec_\pi \texttt{lR}\langle r, w\rangle$ and $(w, r) \in \mathsf{IB}$.

- Checking $[\texttt{nlW}]; \mathsf{pf} \subseteq \mathsf{OB}$.
  If $(w, p) \in \mathsf{pf}$ with $w \in \texttt{nlW}$, then there exists e such that $\texttt{nlW}\langle w, e\rangle \prec_\pi \texttt{P}\langle p, e\rangle$. From $\mathsf{backComp}(\pi)$, we have $\texttt{nlW}\langle w, e\rangle \prec_\pi \mathsf{B}\langle w\rangle \prec_\pi \texttt{P}\langle p, e\rangle$ and so $(w, p) \in \mathsf{OB}$.

- Checking $\mathsf{pf} \subseteq \mathsf{IB}$.
  If $(w, p) \in \mathsf{pf}$, then there exists e such that either $\texttt{nlW}\langle w, e\rangle \prec_\pi \texttt{P}\langle p, e\rangle$ or $\texttt{nrW}\langle w, e\rangle \prec_\pi \texttt{P}\langle p, e\rangle$. In both cases we immediately have $(w, p) \in \mathsf{IB}$.

- Checking $\mathsf{rb}_{\mathsf{b}} \subseteq \mathsf{IB}$.
  If $(r, w') \in \mathsf{rb}_{\mathsf{b}}$ then $r \in \texttt{lR}$, $w' \in \texttt{lW}$, $t(r) = t(w')$, and there exists w such that $(w, r) \in \mathsf{rf}$ and $(w, w') \in \mathsf{mo}$. There is π_4 and π_3 such that $\pi = \pi_4 \cdot \texttt{lR}\langle r, w\rangle \cdot \pi_3$. So by $\mathsf{wfrd}(\pi)$ we have $\mathsf{wfrdCPU}(r, w, \pi_3)$, and there is three cases to consider.

  - In the first case, $\pi_3 = \pi_2 \cdot \lambda_w \cdot \pi_1$, with $\lambda_w \in \{\mathsf{B}\langle w\rangle, \texttt{CAS}\langle w, \_\rangle\}$, and $\mathsf{B}\langle w'\rangle \notin \pi_2$. Since $(w, w') \in \mathsf{mo}$ we have $\mathsf{B}\langle w'\rangle \notin \pi_1$, an so $\mathsf{B}\langle w'\rangle \notin \pi_3$. The last condition of the first case then gives us $\texttt{lW}\langle w'\rangle \notin \pi_3$, which implies $(r, w') \in \mathsf{IB}$.
  - In the second case, $\pi_3 = \pi_2 \cdot \lambda_w \cdot \pi_1$, with $\lambda_w = \texttt{lW}\langle w\rangle$, $\mathsf{thread}(w) = \mathsf{thread}(r)$, and $\mathsf{B}\langle w\rangle \notin \pi_3$. Then w and w' are on the same thread, and by $\mathsf{bufFlushOrd}(\pi)$ and $(w, w') \in \mathsf{mo}$ we have $\texttt{lW}\langle w\rangle \prec_\pi \texttt{lW}\langle w'\rangle$ and $\texttt{lW}\langle w'\rangle \notin \pi_1$. The last condition of the second case gives us $\texttt{lW}\langle w'\rangle \notin \pi_2$, so $\texttt{lW}\langle w'\rangle \notin \pi_3$ and $(r, w') \in \mathsf{IB}$.
  - In the last case, $w = init_x$ for some location x, and we immediately get $\texttt{lW}\langle w'\rangle \notin \pi_3$, which implies $(r, w') \in \mathsf{IB}$.

- Checking $\mathsf{rb} \subseteq \mathsf{OB}$.
  If $(r, w') \in \mathsf{rb}$, then there exists w such that $(w, r) \in \mathsf{rf}$ and $(w, w') \in \mathsf{mo}$. By definition of rf, there is π_4 and π_3 such that $\pi = \pi_4 \cdot \lambda_r \cdot \pi_3$, with $\lambda_r \in \{\texttt{lR}\langle r, w\rangle, \texttt{CAS}\langle r, w\rangle, \texttt{nlR}\langle r, w, \_, \_\rangle, \texttt{nrR}\langle r, w, \_, \_\rangle, \texttt{naF}\langle r, w, \_, \_\rangle, \texttt{narR}\langle r, w, \_, \_, \_\rangle\}$. So by $\mathsf{wfrd}(\pi)$ we have either $\mathsf{wfrdNIC}(r, w, \pi_3)$ or $\mathsf{wfrdCPU}(r, w, \pi_3)$, and there are five cases to consider.

  - In the first case of $\mathsf{wfrdNIC}(r, w, \pi_3)$, $\pi_3 = \pi_2 \cdot \mathsf{getO}\lambda(w, \pi) \cdot \pi_1$, and $\mathsf{getO}\lambda(w', \pi) \notin \pi_2$. Since $(w, w') \in \mathsf{mo}$ we have $\mathsf{getO}\lambda(w', \pi) \notin \pi_1$, and thus $\mathsf{getO}\lambda(w', \pi) \notin \pi_3$. So $\mathsf{getO}\lambda(w', \pi) \in \pi_4$ and $(r, w') \in \mathsf{OB}$.

  - In the last case $\mathsf{wfrdNIC}(r, w, \pi_3)$, $w = init_x$ for some location x, and we immediately have $\mathsf{getO}\lambda(w', \pi) \notin \pi_3$, which implies $(r, w') \in \mathsf{OB}$.
  - For the first case of $\mathsf{wfrdCPU}(r, w, \pi_3)$, same reasoning as for the first case of $\mathsf{wfrdNIC}$.
  - For the second case of $\mathsf{wfrdCPU}(r, w, \pi_3)$, $\pi_3 = \pi_2 \cdot \mathsf{getI}\lambda(w, \pi) \cdot \pi_1$, with $\mathsf{thread}(w) = \mathsf{thread}(r)$, and $\mathsf{getO}\lambda(w, \pi) \notin \pi_3$. So $\mathsf{getO}\lambda(w, \pi) \in \pi_4$, and since $(w, w') \in \mathsf{mo}$ we have $\mathsf{getO}\lambda(w', \pi) \in \pi_4$ as well, and $(r, w') \in \mathsf{OB}$.
  - For the last case of $\mathsf{wfrdCPU}(r, w, \pi_3)$, same reasoning as for the last case of $\mathsf{wfrdNIC}$.

- Checking $\mathsf{nfo} \subseteq \mathsf{IB}$.
  By definition of nfo.

- Checking $\mathsf{nfo} \subseteq \mathsf{OB}$.
  By definition of nfo.

- Checking $\mathsf{mo} \subseteq \mathsf{OB}$.
  By definition of mo, as what matters are the $init_x$, $\mathsf{B}\langle w \rangle$, and $\mathtt{CAS}\langle w, \_ \rangle$ events.

- Checking $\mathsf{rao} \subseteq \mathsf{IB}$.
  By definition of rao.

- Checking $\mathsf{ar}; \mathsf{rao} \subseteq \mathsf{OB}$.
  If $(w, r_1) \in \mathsf{ar}$ then $\mathtt{narR}\langle r_1, a_1, \_, \_, w \rangle \in \pi$ for some $a_1 \in \mathtt{nRMW}$, and if $(r_1, r_2) \in \mathsf{rao}$ then $\mathtt{narR}\langle r_1, \_, a_1, \_, w \rangle \prec_\pi \lambda_r$ for some $\lambda_r \in \{\mathtt{naF}\langle r_2, \_, a_2, \_ \rangle, \mathtt{narR}\langle r_2, \_, a_2, \_, \_ \rangle\}$, with $\overline{n}(a_1) = \overline{n}(a_2)$. Then using $\mathsf{nicAtomicity}(\pi)$ we have that $\mathsf{B}\langle w \rangle \prec_\pi \lambda_r$.

□

B.4 From Declarative Semantics to Annotated Semantics

From a program P and a well-formed consistent execution graph $G = (\mathsf{Event}, \mathsf{po}, \mathsf{rf}, \mathsf{pf}, \mathsf{mo}, \mathsf{nfo}, \mathsf{rao})$, where $(\mathsf{Event}, \mathsf{po})$ is generated by P, we want to reconstruct an annotated semantics execution.

Theorem 5. ib and ob can be extended into total relations IB and OB on Event such that:

- IB and OB are irreflexive and transitive;
- $\mathsf{OB}; [\mathtt{Inst}] \subseteq \mathsf{IB}$ and $[\mathtt{Inst}]; \mathsf{IB} \subseteq \mathsf{OB}$.

Proof. We show that if ib is not already total we can extend it (and maybe ob) into a strictly bigger relation satisfying the constraints of the theorem. Let us assume that there is $(a, b) \in \mathsf{Event}^2$ such that $(a, b) \notin \mathsf{ib}$ and $(b, a) \notin \mathsf{ib}$. We arbitrarily decide to include (a, b) in our relation and we define $\mathsf{ib}' = (\mathsf{ib} \cup \{(a, b)\})^+$ and $\mathsf{ob}' = (\mathsf{ob} \cup [\mathtt{Inst}]; \mathsf{ib}')^+$.

Clearly ib' and ob' are transitive, ib' is irreflexive, and $[\mathtt{Inst}]; \mathsf{ib}' \subseteq \mathsf{ob}'$. We need to prove the following two facts: ob' is still irreflexive; and $\mathsf{ob}'; [\mathtt{Inst}] \subseteq \mathsf{ib}'$.

First, let us check that $(\mathsf{ob} \cup [\mathtt{Inst}]; \mathsf{ib}')^+$ is irreflexive. Since ob and $([\mathtt{Inst}]; \mathsf{ib}')$ are both transitive and irreflexive, a cycle would only be possible by alternating between the two components, so it is enough to show that $(\mathsf{ob}; ([\mathtt{Inst}]; \mathsf{ib}'))^+$ is irreflexive. But $(\mathsf{ob}; ([\mathtt{Inst}]; \mathsf{ib}'))^+ = ((\mathsf{ob}; [\mathtt{Inst}]); \mathsf{ib}')^+ \subseteq (\mathsf{ib}; \mathsf{ib}')^+ \subseteq \mathsf{ib}'$ is irreflexive. Thus ob' is irreflexive.

Then, we need to check that $\mathsf{ob}'; [\mathtt{Inst}] \subseteq \mathsf{ib}'$. Using some rewriting, $\mathsf{ob}' = (\mathsf{ob} \cup [\mathtt{Inst}]; \mathsf{ib}')^+ = \mathsf{ob} \cup (\mathsf{ob}^*; ([\mathtt{Inst}]; \mathsf{ib}'))^+; \mathsf{ob}^*$. We know $\mathsf{ob}; [\mathtt{Inst}] \subseteq \mathsf{ib}'$, which also implies $\mathsf{ob}^*; [\mathtt{Inst}] \subseteq \mathsf{ib}'^*$. So $\mathsf{ob}'; [\mathtt{Inst}] = \mathsf{ob}; [\mathtt{Inst}] \cup ((\mathsf{ob}^*; [\mathtt{Inst}]); \mathsf{ib}')^+; (\mathsf{ob}^*; [\mathtt{Inst}]) \subseteq \mathsf{ib}' \cup (\mathsf{ib}'^*; \mathsf{ib}')^+; \mathsf{ib}'^* \subseteq \mathsf{ib}'$.

Once ib is a total relation on Event, we can similarly freely extend ob into a total relation. □

We use Theorem 5 above to extend ib and ob into total relations IB and OB.

Since $(\mathsf{Event}, \mathsf{po})$ is derived from P, by §5.2 we have that for all $t \in \mathsf{Tid}$ there are s_t and G_t such that $G_t \in G^t(s_t)$, $\mathsf{P}(t) \rightarrowtail s_t$ and $(\mathsf{Event}, \mathsf{po}) = G_{init}; (\|_{t \in \mathsf{Tid}}\, G_t)$. We consider each premise of the form $C \rightarrowtail s$, where C is a primitive command, to generate new events and annotated labels.

- If $s = r \in \mathtt{lR}$, from well-formedness conditions, there is w such that $(w, r) \in \mathsf{rf}$ and $\mathsf{eq}_{\mathtt{loc\&v}}(r, w)$. We create an annotated label $\mathtt{lR}\langle r, w\rangle$.
- If $s = u, s'$ where $u \in \mathtt{CAS}$, from well-formedness conditions, there is w such that $(w, u) \in \mathsf{rf}$ and $\mathsf{eq}_{\mathtt{loc\&v}}(u, w)$. We create an annotated label $\mathtt{CAS}\langle u, w\rangle$, then process s'.
- If $s = f, r, s'$ where $f \in \mathtt{F}$, $r \in \mathtt{lR}$, and $w \in \mathtt{lW}$, from well-formedness conditions, there is w' such that $(w', r) \in \mathsf{rf}$ and $\mathsf{eq}_{\mathtt{loc\&v}}(r, w')$. We create annotated labels $\mathtt{F}\langle f\rangle$, $\mathtt{lR}\langle r, w'\rangle$, $\mathtt{lW}\langle w\rangle$ and $\mathsf{B}\langle w\rangle$, then process s'.
- If $s = w \in \mathtt{lW}$, we create annotated labels $\mathtt{lW}\langle w\rangle$ and $\mathsf{B}\langle w\rangle$.

- If $s = f \in \mathtt{F}$, we create annotated labels $\mathtt{F}\langle f \rangle$.
- If $s = r, w$ where $r \in \mathtt{nlR}$ and $w \in \mathtt{nrW}$, we create two events $a \in \mathtt{Put}$ and $e \in \mathtt{nrEX}$, and the annotated labels $\mathtt{Push}\langle a \rangle$, $\mathtt{NIC}\langle a \rangle$, $\mathtt{nlR}\langle r, w', a, w \rangle$ (where $(w', r) \in \mathsf{rf}$), $\mathtt{nrW}\langle w, e \rangle$, $\mathsf{B}\langle w \rangle$, and $\mathsf{CN}\langle e \rangle$. If there is p such that $(w, p) \in \mathsf{pf}$, we also create an annotated label $\mathtt{P}\langle p, e \rangle$. To simplify later definition, we also extend po such that the event a is placed just before r, and e just after w. I.e., let $\mathsf{po}' = \mathsf{po} \cup \{(e', a) \mid (e', r) \in \mathsf{po}\} \cup \{(a, e') \mid (r, e') \in \mathsf{po}^*\}$ and redefine $\mathsf{po} = \mathsf{po}' \cup \{(e', e) \mid (e', w) \in \mathsf{po}'^*\} \cup \{(e, e') \mid (w, e') \in \mathsf{po}'\}$.
  Note: from well-formedness conditions, every $\mathtt{nlR}$ and every $\mathtt{nrW}$ are part of such a pair.
- If $s = r, w$ where $r \in \mathtt{nrR}$ and $w \in \mathtt{nlW}$, we similarly create $a \in \mathtt{Get}$, $e \in \mathtt{nlEX}$, $\mathtt{Push}\langle a \rangle$, $\mathtt{NIC}\langle a \rangle$, $\mathtt{nrR}\langle \ldots \rangle$, $\mathtt{nlW}\langle \ldots \rangle$, $\mathsf{B}\langle \ldots \rangle$, and potentially $\mathtt{P}\langle \ldots \rangle$.
- If $s = r, w$ where $r \in \mathtt{narR}$ and $w \in \mathtt{nlW}$, we have C of the form $z := \mathtt{nCAS}(\overline{x}, e, e')$, so we use the values $[\![e]\!]$ and $[\![e']\!]$ to create $a \in \mathtt{nCAS}$, $\mathtt{Push}\langle a \rangle$, $\mathtt{NIC}\langle a \rangle$, $\mathtt{naF}\langle \ldots \rangle$, $\mathtt{nlW}\langle \ldots \rangle$, $\mathsf{B}\langle \ldots \rangle$, and potentially $\mathtt{P}\langle \ldots \rangle$.
- If $s = r, w_1, w_2$ where $r \in \mathtt{narR}$, $w_1 \in \mathtt{narW}$, $w_2 \in \mathtt{nlW}$, we have C either of the form $z := \mathtt{nFAA}(\overline{x}, e)$ or $z := \mathtt{nCAS}(\overline{x}, e_1, e_2)$, so we create $a \in \mathtt{nFAA}$ or $a \in \mathtt{nCAS}$ accordingly, and $\mathtt{Push}\langle a \rangle$, $\mathtt{NIC}\langle a \rangle$, $\mathtt{narR}\langle \ldots \rangle$, $\mathtt{narW}\langle w_1 \rangle$, $\mathtt{nlW}\langle w_2, \ldots \rangle$, $\mathsf{B}\langle w_1 \rangle$, $\mathsf{B}\langle w_2 \rangle$ and potentially $\mathtt{P}\langle \ldots \rangle$.
- If $s = f \in \mathtt{nF}$, we create the annotated labels $\mathtt{Push}\langle f \rangle$, $\mathtt{NIC}\langle f \rangle$, and $\mathtt{nF}\langle f \rangle$.
- We ignore $s = p \in \mathtt{P}$, as this is already handled by our earlier cases.

Then, we use IB and OB to reconstruct a partial path from these annotated labels. We define a path π_0 such that:

- $\pi_0 \in (\mathsf{ALabel} \setminus (\mathtt{Push} \cup \mathtt{NIC} \cup \mathsf{CN}))^*$
- $\mathsf{getI}\lambda(e_1, \pi_0) \prec_{\pi_0} \mathsf{getI}\lambda(e_2, \pi_0) \iff (e_1, e_2) \in \mathsf{IB}$
- $\mathsf{getO}\lambda(e_1, \pi_0) \prec_{\pi_0} \mathsf{getO}\lambda(e_2, \pi_0) \iff (e_1, e_2) \in \mathsf{OB}$
- $\forall w \in \{\mathtt{lW}, \mathtt{nlW}, \mathtt{nrW}, \mathtt{narW}\}$, $\mathsf{getI}\lambda(w, \pi_0) \prec_{\pi_0} \mathsf{getO}\lambda(w, \pi_0)$

This is possible from the properties of IB and OB. For pairs of annotated labels not ordered by IB or OB, we decide to order $\mathtt{lW}\langle w \rangle / \mathtt{nlW}\langle w, \_ \rangle / \mathtt{nrW}\langle w, \_ \rangle / \mathtt{narW}\langle w \rangle$ first and $\mathsf{B}\langle w \rangle$ last. Note that the annotated labels $\mathtt{Push}\langle \ldots \rangle$, $\mathtt{NIC}\langle \ldots \rangle$, and $\mathsf{CN}\langle \ldots \rangle$ not covered by IB/OB are not yet integrated in π_0.

Then we extend π_0 to add annotated labels not considered by the declarative semantics. We use the following extension function that introduces a new annotated label as early as possible after a set of dependencies.

$$\mathsf{extend}(\pi, \lambda, S) \triangleq \begin{cases} \pi_2 \cdot \lambda \cdot \lambda' \cdot \pi_1 & \text{if } \pi = \pi_2 \cdot \lambda' \cdot \pi_1 \wedge \lambda' \in S \wedge \pi_2 \cap S = \emptyset \\ \pi \cdot \lambda & \text{if } \pi \cap S = \emptyset \end{cases}$$

We define a new function to recover the first annotated label corresponding to an event:

$$E^{\mathrm{ext}} \triangleq \mathsf{Event} \cup (\mathtt{Get} \cup \mathtt{Put} \cup \mathtt{nCAS} \cup \mathtt{nFAA} \cup \mathtt{nlEX} \cup \mathtt{nrEX})$$

$$\mathsf{getCPU} : E^{\text{ext}} \rightharpoonup \mathsf{ALabel}$$

$$\mathsf{getCPU}(e) \triangleq \begin{cases} \mathsf{getI}\lambda(e, \pi_0) & \text{if } e \in E^{\text{cpu}} = \{\texttt{lR}, \texttt{lW}, \texttt{CAS}, \texttt{F}, \texttt{P}\} \\ \texttt{Push}\langle e\rangle & \text{if } e \in \{\texttt{Put}, \texttt{Get}, \texttt{nCAS}, \texttt{nFAA}, \texttt{nF}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

And a similar function for events emptying a CPU buffer:

$$\mathsf{getTSO} : E^{\text{ext}} \rightharpoonup \mathsf{ALabel}$$

$$\mathsf{getTSO}(e) \triangleq \begin{cases} \mathsf{B}\langle e\rangle & \text{if } e \in \texttt{lW} \\ \texttt{NIC}\langle e\rangle & \text{if } e \in \{\texttt{Put}, \texttt{Get}, \texttt{nCAS}, \texttt{nFAA}, \texttt{nF}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Let us consider $(a_1, \ldots, a_n) = \mathsf{Event} \cap \{\texttt{Put}, \texttt{Get}, \texttt{nCAS}, \texttt{nFAA}, \texttt{nF}\}$ in po order, i.e., if $i < j$ then $(a_j, a_i) \notin \mathsf{po}$. We extend π_0 successively until we get π_n:

- We introduce $\texttt{Push}$ as early as possible:
  Let $\pi' = \mathsf{extend}(\pi_{i-1}, \texttt{Push}\langle a_i\rangle, \{\mathsf{getCPU}(e) \mid (e, a_i) \in \mathsf{po}\})$
- We introduce $\texttt{NIC}$ as early as possible:
  Let $\pi'' = \mathsf{extend}(\pi', \texttt{NIC}\langle a_i\rangle, \{\texttt{Push}\langle a_i\rangle\} \cup \{\mathsf{getTSO}(e) \mid (e, a_i) \in \mathsf{po}\})$
- If $a_i \in \texttt{Put}$, there is $e_i \in \texttt{nrEX}$ such that $\texttt{nlR}\langle \_, \_, a_i, w\rangle \prec_{\pi_0} \texttt{nrW}\langle w, e_i\rangle$. We also introduce CN: Let $S = \{\texttt{nrW}\langle w, e_i\rangle\} \cup \{\texttt{nlW}\langle \_, e\rangle \mid (e, e_i) \in \mathsf{po} \cap \texttt{sqp}\} \cup \{\mathsf{CN}\langle e\rangle \mid (e, e_i) \in \mathsf{po} \cap \texttt{sqp}\}$, we pose $\pi_i = \mathsf{extend}(\pi'', \mathsf{CN}\langle e_i\rangle, S)$.
  Otherwise, i.e. $a_i \notin \texttt{Put}$, we simply have $\pi_i = \pi''$

Finally, $\pi = \pi_n$ is our path for an annotated semantics reduction. We clearly have $\mathsf{complete}(\pi)$ by definition. Our goal is then to prove that $\mathsf{wfp}(\pi)$ holds. It is composed of seven properties. Note that we already have the existence of the relevant annotated labels, and we need to show that the ordering constraints are respected.

nodup

$\mathsf{nodup}(\pi)$ directly comes from the definition of annotated labels. There is no conflict in event usage.

backComp

Here are a couple lemmas showing that the new annotated labels are not placed too late and do not disturb the expected ordering.

Lemma 3. For all $a \in \{\texttt{Put}, \texttt{Get}, \texttt{nCAS}, \texttt{nFAA}, \texttt{nF}\}$ and $b \in \mathsf{Event}$, if $(a, b) \in \mathsf{po}^*$, then $\texttt{Push}\langle a\rangle \prec_\pi \mathsf{getI}\lambda(b, \pi_0)$.

Proof. We take an arbitrary b, and proceed for a in po order, i.e., we can assume it holds for $e \in \{\texttt{Put}, \texttt{Get}, \texttt{nCAS}, \texttt{nFAA}, \texttt{nF}\}$ such that $(e, a) \in \mathsf{po}$. By definition, $\texttt{Push}\langle a\rangle$ comes from an extension $\pi'' = \mathsf{extend}(\pi', \texttt{Push}\langle a\rangle, \{\mathsf{getCPU}(e) \mid (e, a) \in \mathsf{po}\})$ and has been placed either first—and the result is trivial—or just after some $\mathsf{getCPU}(e)$ with $(e, a) \in \mathsf{po}$. If $e \in \{\texttt{Put}, \texttt{Get}, \texttt{nCAS}, \texttt{nFAA}, \texttt{nF}\}$, we have $\texttt{Push}\langle e\rangle \prec_{\pi''} \texttt{Push}\langle a\rangle \prec_{\pi''} \mathsf{getI}\lambda(b, \pi_0)$ by induction hypothesis. If $e \in E^{\text{cpu}} = \{\texttt{lR}, \texttt{lW}, \texttt{CAS}, \texttt{F}, \texttt{P}\}$, we have $\mathsf{getI}\lambda(e, \pi_0) \prec_{\pi''} \texttt{Push}\langle a\rangle \prec_{\pi''} \mathsf{getI}\lambda(b, \pi_0)$ since $(e, b) \in \mathsf{ippo} \subseteq \mathsf{IB}$. □

Lemma 4. $\forall a \in \{\texttt{Put}, \texttt{Get}, \texttt{nCAS}, \texttt{nFAA}, \texttt{nF}\}$, $\forall b \in \{\texttt{nF}, \texttt{nrR}, \texttt{nlR}, \texttt{narR}, \texttt{lW}\}$, if $(a, b) \in \mathsf{po}^*$, then $\texttt{NIC}\langle a\rangle \prec_\pi \mathsf{getO}\lambda(b, \pi_0)$.

Proof. We take an arbitrary $b \in \{\texttt{nF}, \texttt{nrR}, \texttt{nlR}, \texttt{narR}\}$, and proceed for a in po order, i.e., we can assume it holds for $e \in \{\texttt{Put}, \texttt{Get}, \texttt{nCAS}, \texttt{nFAA}, \texttt{nF}\}$ such that $(e, a) \in \mathsf{po}$. By definition, $\texttt{NIC}\langle a\rangle$ comes from an extension $\pi'' = \mathsf{extend}(\pi', \texttt{NIC}\langle a\rangle, S)$, with $S = \{\texttt{Push}\langle a\rangle\} \cup \{\mathsf{getTSO}(e) \mid (e, a) \in \mathsf{po}\}$, and has been placed just after some $\lambda \in S$.

- If $\lambda = \texttt{Push}\langle a\rangle$, then we have $\lambda \prec_{\pi''} \texttt{NIC}\langle a\rangle \prec_{\pi''} \mathsf{getO}\lambda(b, \pi_0)$ using Lemma 3 above, since $\mathsf{getI}\lambda(b, \pi_0) = \mathsf{getO}\lambda(b, \pi_0)$ or $\mathsf{getI}\lambda(b, \pi_0) \prec_{\pi''} \mathsf{getO}\lambda(b, \pi_0)$.
- If $\lambda = \mathsf{getTSO}\langle e\rangle = \texttt{NIC}\langle e\rangle$ for some $e \in \{\texttt{Put}, \texttt{Get}, \texttt{nCAS}, \texttt{nFAA}, \texttt{nF}\}$, then we have $\lambda \prec_{\pi''} \texttt{NIC}\langle a\rangle \prec_{\pi''} \mathsf{getO}\lambda(b, \pi_0)$ by induction hypothesis.
- If $\lambda = \mathsf{getTSO}\langle e\rangle = \mathsf{B}\langle e\rangle$ for some $e \in \texttt{lW}$, then we have $\mathsf{B}\langle e\rangle \prec_{\pi''} \texttt{NIC}\langle a\rangle \prec_{\pi''} \mathsf{getO}\lambda(b, \pi_0)$ since $(e, b) \in \mathsf{oppo} \subseteq \mathsf{OB}$.

□

Lemma 5. Forall w,e,p, if $\texttt{nrW}\langle w, e\rangle \in \pi$ and $\texttt{P}\langle p, e\rangle \in \pi$, then $\mathsf{CN}\langle e\rangle \prec_\pi \texttt{P}\langle p, e\rangle$.

Proof. Once again, we proceed for e in po order, i.e., we can assume the result holds for $e' \in \texttt{nrEX}$ such that $(e', e) \in \mathsf{po}$. $\mathsf{CN}\langle e\rangle$ is inserted in some operation $\pi'' = \mathsf{extend}(\pi', \mathsf{CN}\langle e\rangle, S)$, with $S = \{\texttt{nrW}\langle w, e\rangle\} \cup \{\texttt{nlW}\langle \_, e'\rangle \mid (e', e) \in \mathsf{po} \cap \texttt{sqp}\} \cup \{\mathsf{CN}\langle e'\rangle \mid (e', e) \in \mathsf{po} \cap \texttt{sqp}\}$. It is then placed just after some label $\lambda \in S$.

- If $\lambda = \texttt{nrW}\langle w, e\rangle$, we have $\lambda \prec_{\pi''} \mathsf{CN}\langle e\rangle \prec_{\pi''} \texttt{P}\langle p, e\rangle$ because $(w, p) \in \mathsf{pf} \subseteq \mathsf{IB}$.
- If $\lambda = \mathsf{CN}\langle e'\rangle$ with $(e', e) \in \mathsf{po} \cap \texttt{sqp}$, then there is some w' such that $(w', w) \in \mathsf{po} \cap \texttt{sqp}$ and $\texttt{nrW}\langle w', e'\rangle \in \pi'$. From well-formedness condition number 1 (see Definition 3), there is some p' such that $(w', p') \in \mathsf{pf}$ and $(p', p) \in \mathsf{po}$. By induction hypothesis, we have $\mathsf{CN}\langle e'\rangle \prec_{\pi'} \texttt{P}\langle p', e'\rangle$, and from $(p', p) \in \mathsf{IB}$ we have $\texttt{P}\langle p', e'\rangle \prec_{\pi'} \texttt{P}\langle p, e\rangle$. In the end, we have the result $\mathsf{CN}\langle e'\rangle \prec_{\pi''} \mathsf{CN}\langle e\rangle \prec_{\pi''} \texttt{P}\langle p, e\rangle$.
- If $\lambda = \texttt{nlW}\langle w', e'\rangle$ with $(e', e) \in \mathsf{po} \cap \texttt{sqp}$, then we also have $(w', w) \in \mathsf{po} \cap \texttt{sqp}$, so from well-formedness condition number 1 (see Definition 3), there is some p' such that $(w', p') \in \mathsf{pf}$ and $(p', p) \in \mathsf{po}$. We have $\texttt{nlW}\langle w', e'\rangle \prec_{\pi''} \mathsf{CN}\langle e\rangle \prec_{\pi''} \texttt{P}\langle p', e'\rangle \prec_{\pi''} \texttt{P}\langle p, e\rangle$.

□

We can then check that we have $\mathsf{backComp}(\pi)$:

- $\texttt{lW}\langle w\rangle \prec_\pi \mathsf{B}\langle w\rangle$ comes from the third property when defining π_0; similarly for $\texttt{nlW}$, $\texttt{nrW}$ and $\texttt{narW}$.
- $\texttt{Push}\langle a\rangle \prec_\pi \texttt{NIC}\langle a\rangle$ comes from the extension process.
- $\texttt{NIC}\langle f\rangle \prec_\pi \texttt{nF}\langle f\rangle$ comes from Lemma 4; similarly for $\texttt{NIC}\langle a\rangle \prec_\pi \texttt{nlR/nrR/naF/narR}\langle \ldots\rangle$.
- $\texttt{nlR}\langle r, w, a, w'\rangle \prec_\pi \texttt{nrW}\langle w', e\rangle$ comes from $(r, w') \in \mathsf{ippo} \subseteq \mathsf{IB}$; similarly for $\texttt{nrR/nlW}$, $\texttt{naF/nlW}$, $\texttt{narR/nlW}$ and $\texttt{narR/narW}$.
- $\texttt{nrW}\langle w, e\rangle \prec_\pi \mathsf{CN}\langle e\rangle$ comes from the extension process

- $\texttt{nlW}\langle w, e\rangle \prec_\pi \mathsf{B}\langle w\rangle \prec_\pi \texttt{P}\langle p, e\rangle$ comes from $(w, p) \in [\texttt{nlW}]; \mathsf{pf} \subseteq \mathsf{OB}$.
- $\mathsf{CN}\langle e\rangle \prec_\pi \texttt{P}\langle p, e\rangle$ comes from Lemma 5.

Thus we have $\mathsf{backComp}(\pi)$.

bufFlushOrd

- $\texttt{lW}\langle w_1\rangle \prec_\pi \texttt{lW}\langle w_2\rangle \iff \mathsf{B}\langle w_1\rangle \prec_\pi \mathsf{B}\langle w_2\rangle$ when $t(w_1) = t(w_2)$ comes the fact that $[\texttt{lW}]; \mathsf{po}; [\texttt{lW}] \subseteq (\mathsf{IB} \cup \mathsf{OB})$, so both sides are true if and only if $(w_1, w_2) \in \mathsf{po}$; similarly for $\texttt{nlW}$ and $\texttt{nrW}/\texttt{narW}$ on the same queue pair.
- When $t(a_1) = t(a_2)$, $\texttt{Push}\langle a_1\rangle \prec_\pi \texttt{Push}\langle a_2\rangle \iff \texttt{NIC}\langle a_1\rangle \prec_\pi \texttt{NIC}\langle a_2\rangle \iff (a_1, a_2) \in \mathsf{po}$ from the definition of the extension process (to define π_n).
- For $a \in \{\texttt{Put}, \texttt{Get}, \texttt{nF}, \texttt{nCAS}, \texttt{nFAA}\}$, $w \in \texttt{lW}$, such that $t(a) = t(w)$:
  - If $(w, a) \in \mathsf{po}$, then $\texttt{lW}\langle w\rangle \prec_\pi \texttt{Push}\langle a\rangle$ and $\mathsf{B}\langle w\rangle \prec_\pi \texttt{NIC}\langle a\rangle$ from the definition of the extension process.
  - If $(a, w) \in \mathsf{po}$, then $\texttt{Push}\langle a\rangle \prec_\pi \texttt{lW}\langle w\rangle$ and $\texttt{NIC}\langle a\rangle \prec_\pi \mathsf{B}\langle w\rangle$ from Lemmas 3 and 4.
- When $t(w) = t(f)$, $\texttt{lW}\langle w\rangle \prec_\pi \texttt{F}\langle f\rangle$ implies $(w, f) \in \mathsf{po}$ (since $[\texttt{F}]; \mathsf{po}; [\texttt{lW}] \subseteq \mathsf{ippo} \subseteq \mathsf{IB}$), which implies $\mathsf{B}\langle w\rangle \prec_\pi \texttt{F}\langle f\rangle$ (since $[\texttt{lW}]; \mathsf{po}; [\texttt{F}] \subseteq \mathsf{oppo} \subseteq \mathsf{OB}$); similarly for $\texttt{CAS}$.
- If $w \in \texttt{nlW}$, $r \in \texttt{nlR}$, and $\mathsf{sameqp}(w, r)$, then from the definition of pre-executions (see condition 6 of Definition 2), either $(w, r) \in \mathsf{nfo}$ or $(r, w) \in \mathsf{nfo}$. If $\texttt{nlW}\langle w, \_\rangle \prec_\pi \texttt{nlR}\langle r, \_, \_, \_\rangle$, then $(r, w) \notin \mathsf{nfo}$ (since $\mathsf{nfo} \subseteq \mathsf{IB}$) and $(w, r) \in \mathsf{nfo}$. Thus, $\mathsf{B}\langle w\rangle \prec_\pi \texttt{nlR}\langle r, \_, \_, \_\rangle$ (since $\mathsf{nfo} \subseteq \mathsf{OB}$); similarly for $w \in \{nrW, narW\}$ and $r \in \{\texttt{nrR}, \texttt{narR}\}$.

Thus we have $\mathsf{bufFlushOrd}(\pi)$.

pollOrder

Lemma 6. For all $e_1, e_2 \in \{\texttt{nlEX}, \texttt{nrEX}\}$, such that $\mathsf{sameqp}(e_1, e_2)$, let $\lambda_1 \in \{\texttt{nlW}\langle \_, e_1\rangle, \mathsf{CN}\langle e_1\rangle\}$, $\lambda_2 \in \{\texttt{nlW}\langle \_, e_2\rangle, \mathsf{CN}\langle e_2\rangle\}$, then $(e_1, e_2) \in \mathsf{po} \iff \lambda_1 \prec_\pi \lambda_2$.

Proof. By symmetry, we only need to show $(e_1, e_2) \in \mathsf{po} \implies \lambda_1 \prec_\pi \lambda_2$. Once again, we proceed for e_1 in po order, i.e., we can assume the result holds for $e' \in \texttt{nEX}$ such that $(e', e_1) \in \mathsf{po}$.

- If $\lambda_1 = \texttt{nlW}\langle w_1, e_1\rangle$ and $\lambda_2 = \texttt{nlW}\langle w_2, e_2\rangle$, then $(e_1, e_2) \in \mathsf{po}$ implies $(w_1, w_2) \in (\mathsf{po} \cap \mathsf{sqp})$, so $(w_1, w_2) \in \mathsf{ippo} \subseteq \mathsf{IB}$ and $\lambda_1 \prec_\pi \lambda_2$.
- If $\lambda_1 = \texttt{nlW}\langle w_1, e_1\rangle$ and $\lambda_2 = \mathsf{CN}\langle e_2\rangle$, then by definition of the extension process we have $\lambda_1 \prec_\pi \lambda_2$.
- If $\lambda_1 = \mathsf{CN}\langle e_1\rangle$ and $\lambda_2 = \texttt{nlW}\langle w_2, e_2\rangle$, then λ_1 is inserted in some operation $\pi'' = \mathsf{extend}(\pi', \mathsf{CN}\langle e_1\rangle, S)$, with $S = \{\texttt{nrW}\langle \_, e_1\rangle\} \cup \{\texttt{nlW}\langle \_, e'\rangle \mid (e', e_1) \in \mathsf{po} \cap \mathsf{sqp}\} \cup \{\mathsf{CN}\langle e'\rangle \mid (e', e_1) \in \mathsf{po} \cap \mathsf{sqp}\}$. It is then placed just after some label $\lambda \in S$.
  - If $\lambda = \texttt{nrW}\langle w_1, e_1\rangle$, we have $\lambda \prec_{\pi''} \lambda_1 \prec_{\pi''} \lambda_2$ because $(w_1, w_2) \in \mathsf{ippo} \subseteq \mathsf{IB}$.

  - If $\lambda = \mathsf{CN}\langle e' \rangle$ or $\lambda = \mathtt{nlW}\langle \_, e' \rangle$, with $(e', e_1) \in \mathsf{po} \cap \mathtt{sqp}$, then by induction hypothesis $\lambda \prec_{\pi''} \lambda_1 \prec_{\pi''} \lambda_2$.

- If $\lambda_1 = \mathsf{CN}\langle e_1 \rangle$ and $\lambda_2 = \mathsf{CN}\langle e_2 \rangle$, then by definition of the extension process we have $\lambda_1 \prec_\pi \lambda_2$.

□

Let us assume we have $e_1, e_2, p_2, \lambda_1, \lambda_2$ such that $\mathsf{sameqp}(e_1, e_2)$, $\lambda_1 \in \{\mathtt{nlW}\langle \_, e_1 \rangle, \mathsf{CN}\langle e_1 \rangle\}$, $\lambda_2 \in \{\mathtt{nlW}\langle \_, e_2 \rangle, \mathsf{CN}\langle e_2 \rangle\}$, $\lambda_1 \prec_\pi \lambda_2$, and $\mathtt{P}\langle p_2, e_2 \rangle \in \pi$.

From the creation of the events e_1 and e_2, there is some $w_1, w_2 \in \{\mathtt{nlW}, \mathtt{nrW}\}$ such that $(w_i, e_i) \in \mathsf{po}|_{\mathrm{imm}}$. From Lemma 6, we have $(e_1, e_2) \in \mathsf{po}$ and thus $(w_1, w_2) \in (\mathsf{po} \cap \mathtt{sqp})$. By definition, we also have $(w_2, p_2) \in \mathsf{pf}$. From well-formedness condition number 1 (see Definition 3), there is some p_1 such that $(w_1, p_1) \in \mathsf{pf}$ and $(p_1, p_2) \in \mathsf{po}$. Thus we have $\mathtt{P}\langle p_1, e_1 \rangle \prec_\pi \mathtt{P}\langle p_2, e_2 \rangle$ as required to prove $\mathsf{pollOrder}(\pi)$.

nicActOrder

Let a_1 and a_2 such that $\mathtt{NIC}\langle a_1 \rangle \prec_\pi \mathtt{NIC}\langle a_2 \rangle$ and $\mathsf{sameqp}(a_1, a_2)$. From the definition of the extension process, we have $(a_1, a_2) \in \mathsf{po}$.

- If $a_1 \in \mathtt{nF}$ or $a_2 \in \mathtt{nF}$, then most of the required results hold by definition of ippo. The only exception is $\mathsf{CN}\langle e \rangle \prec_\pi \mathtt{nF}\langle a_2 \rangle$ which holds (by induction on e in po order) because all the dependencies of $\mathsf{CN}\langle e \rangle$ are before $\mathtt{nF}\langle a_2 \rangle$ by ippo.
- If $(a_1 \in \mathtt{Get} \wedge a_2 \in \mathtt{Get})$, the result holds by ippo.
- If $(a_1 \in \mathtt{Get} \wedge a_2 \in \mathtt{Put})$, the result holds by Lemma 6.
- If $(a_1 \in \mathtt{Get} \wedge a_2 \in \mathtt{nCAS} \cup \mathtt{nFAA})$, the results hold by ippo.
- If $(a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{Get})$, the first result holds by ippo, the second by Lemma 6.
- If $(a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{Put})$, the first two results hold by ippo, the last one by Lemma 6.
- If $(a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{nCAS} \cup \mathtt{nFAA})$, the results hold by ippo.
- If $(a_1 \in \mathtt{nCAS} \cup \mathtt{nFAA} \wedge a_2 \in \mathtt{Get})$, the results hold by ippo.
- If $(a_1 \in \mathtt{nCAS} \cup \mathtt{nFAA} \wedge a_2 \in \mathtt{Put})$, the first result holds by ippo, the latter by Lemma 6.
- If $(a_1, a_2 \in \mathtt{nCAS} \cup \mathtt{nFAA})$, the first result holds by ippo, the latter by Lemma 6.

Thus we have $\mathsf{nicActOrder}(\pi)$.

nicAtomicity

For every $a_1, a_2 \in \mathtt{nRMW}$ where $\overline{n}(a_1) = \overline{n}(a_2)$, if $\mathtt{narR}\langle r_1, a_1, \_, \_, w \rangle \prec_\pi \lambda_r$ where $\lambda_r \in \{\mathtt{naF}\langle r_2, \_, a_2, \_ \rangle, \mathtt{narR}\langle r_2, \_, a_2 \_, \_ \rangle\}$, then from the extension process we have $(r_1, w) \in \mathsf{po}|_{\mathrm{imm}}$, and $w \in \mathtt{narW}$, so $(w, r_1) \in \mathsf{ar}$. Then we need to show that $(r_1, r_2) \in \mathsf{rao}$. Suppose, for contradiction, that $(r_1, r_2) \notin \mathsf{rao}$. By definition of rao, for each node n, rao_n is a total order on $\{e \in \mathtt{narR} \mid \overline{n}(e) = n\}$. Thus we have either $(r_1, r_2) \in \mathsf{rao}$ or $(r_2, r_1) \in \mathsf{rao}$, and by assumption the prior is not the case so $(r_2, r_1) \in \mathsf{rao} \subseteq \mathsf{IB}$. However, since $\mathtt{narR}\langle r_1, \ldots \rangle \prec_\pi \lambda_r$, we have $(r_1, r_2) \in \mathsf{IB}$, which is a contradiction, as IB is irreflexive. Therefore reject our original assumption. Thus $(r_1, r_2) \in \mathsf{rao}$, then we have $(w, r_2) \in \mathsf{ar}; \mathsf{rao} \subseteq \mathsf{OB}$, so $\mathsf{B}\langle w \rangle \prec_\pi \lambda_r$. Thus we have $\mathsf{nicAtomicity}(\pi)$.

wfrd

Let us assume we have $\pi = \pi_4 \cdot \lambda_r \cdot \pi_3$, with $\lambda_r \in \{\texttt{lR}\langle r, w\rangle, \texttt{CAS}\langle r, w\rangle, \texttt{nlR}\langle r, w, \_, \_\rangle, \texttt{nrR}\langle r, w, \_, \_\rangle, \texttt{naF}\langle r, w, \_, \_\rangle, \texttt{narR}\langle r, w, \_, \_, \_\rangle\}$. In all cases we have $(w, r) \in \mathsf{rf}$. Another important fact is that $\forall w', (w, w') \in \mathsf{mo} \implies (r, w') \in \mathsf{rb}$.

- If $\lambda_r = \texttt{lR}\langle r, w\rangle$, we need to show $\mathsf{wfrdCPU}(r, w, \pi_3)$.
  - If $w = init_{\texttt{loc}(w)}$, then we need to check that $\{\mathsf{B}\langle w'\rangle, \texttt{CAS}\langle w', \_\rangle \in \pi_3 \mid \texttt{loc}(w') = \texttt{loc}(r)\} = \emptyset$ and $\{\texttt{lW}\langle w''\rangle \in \pi_3 \mid \texttt{loc}(w'') = \texttt{loc}(r) \wedge t(w'') = t(r)\} = \emptyset$. For the first, such a w' would imply $(r, w') \in \mathsf{rb} \subseteq \mathsf{OB}$, which contradicts the ordering with λ_r. For the second, such an w'' would imply $(r, w'') \in \mathsf{rb_b} \subseteq \mathsf{IB}$, and $\lambda_r \prec_\pi \texttt{lW}\langle w''\rangle$ which similarly contradicts the ordering with λ_r.
  - If $w \in \texttt{lW}$, $t(w) = t(r)$, and $\mathsf{B}\langle w\rangle \notin \pi_3$. From $(w, r) \in \mathsf{rf_b} \subseteq \mathsf{IB}$, we have $\lambda_w = \texttt{lW}\langle w\rangle \prec_\pi \lambda_r$, i.e., $\pi_3 = \pi_2 \cdot \lambda_w \cdot \pi_1$. We need to show that $\{\texttt{lW}\langle w'\rangle \in \pi_2 \mid \texttt{loc}(w') = \texttt{loc}(r) \wedge t(w') = t(r)\} = \emptyset$. Such a w' would imply $(w, w') \in \mathsf{po}$ (from $[\texttt{lW}]; \mathsf{po}; [\texttt{lW}] \subseteq \mathsf{ippo} \subseteq \mathsf{IB}$, and the execution graph forcing either $(w, w') \in \mathsf{po}$ or $(w', w) \in \mathsf{po}$), $(w, w') \in \mathsf{mo}$ (from $[\texttt{lW}]; \mathsf{po}; [\texttt{lW}] \subseteq \mathsf{oppo} \subseteq \mathsf{OB}$, and well-formedness conditions forcing either $(w, w') \in \mathsf{mo}$ or $(w', w) \in \mathsf{mo}$), and $(r, w') \in \mathsf{rb_b} \subseteq \mathsf{IB}$ would contradicts the ordering with λ_r.
  - Else we have $\lambda_w \in \pi_3$, with $\lambda_w \in \{\mathsf{B}\langle w\rangle, \texttt{CAS}\langle w, \_\rangle\}$. If $w \in \texttt{lW}$ and $t(w) = t(r)$, this is the remaining subcase, else it comes from $(w, r) \in \mathsf{rf}_{\overline{\mathsf{b}}} \subseteq \mathsf{OB}$. Thus we have $\pi_3 = \pi_2 \cdot \lambda_w \cdot \pi_1$, and we need to check two properties. First, we check that $\{\mathsf{B}\langle w'\rangle, \texttt{CAS}\langle w', \_\rangle \in \pi_2 \mid \texttt{loc}(w') = \texttt{loc}(r)\} = \emptyset$. It holds because such a w' would again imply $(r, w') \in \mathsf{rb} \subseteq \mathsf{OB}$ and contradict the ordering with λ_r. Second, we check that $\left\{ w' \;\middle|\; \begin{array}{l} \texttt{lW}\langle w'\rangle \in \pi_3 \wedge \mathsf{B}\langle w'\rangle \notin \pi_3 \wedge \\ \texttt{loc}(w') = \texttt{loc}(r) \wedge t(w') = t(r) \end{array} \right\} = \emptyset$. It holds because such a w' would again imply $(w, w') \in \mathsf{mo}$, $(r, w') \in \mathsf{rb_b} \subseteq \mathsf{IB}$ and contradict the ordering with λ_r.
- If $\lambda_r = \texttt{CAS}\langle r, w\rangle$, we similarly check that $\mathsf{wfrdCPU}(r, w, \pi_3)$ holds. The difference is that cases that previously contradicted ($\mathsf{rb_b} \subseteq \mathsf{IB}$) now contradict $\mathsf{bufFlushOrd}(\pi)$ that forces the buffer of $t(r)$ to be empty when performing λ_r.
- If $\lambda_r = \texttt{nlR}\langle r, w, \_, \_\rangle$, we need to show $\mathsf{wfrdNIC}(r, w, \pi_3)$.
  - If $w = init_{\texttt{loc}(w)}$, then we need to check that $\{\mathsf{B}\langle w'\rangle, \texttt{CAS}\langle w', \_\rangle \in \pi_3 \mid \texttt{loc}(w') = \texttt{loc}(r)\} = \emptyset$. Such a w' would imply $(r, w') \in \mathsf{rb} \subseteq \mathsf{OB}$, which contradicts the ordering with λ_r.
  - Else we have $\lambda_w \in \pi_3$, with $\lambda_w \in \{\mathsf{B}\langle w\rangle, \texttt{CAS}\langle w, \_\rangle\}$. This comes from $(w, r) \in \mathsf{rf}_{\overline{\mathsf{b}}} \subseteq \mathsf{OB}$. Thus we have $\pi_3 = \pi_2 \cdot \lambda_w \cdot \pi_1$, and we need to check that $\{\mathsf{B}\langle w'\rangle, \texttt{CAS}\langle w', \_\rangle \in \pi_2 \mid \texttt{loc}(w') = \texttt{loc}(r)\} = \emptyset$. It holds because such a w' would again imply $(r, w') \in \mathsf{rb} \subseteq \mathsf{OB}$ and contradict the ordering with λ_r.
- If $\lambda_r = \texttt{nrR}\langle r, w, \_, \_\rangle$, $\texttt{naF}\langle r, w, \_, \_\rangle$ or $\texttt{narR}\langle r, w, \_, \_, \_\rangle$, we similarly check that $\mathsf{wfrdNIC}(r, w, \pi_3)$ for the same reasons.

Thus we have $\mathsf{wfrd}(\pi)$.

Theorem 6. Let G be a well-formed consistent execution graph generated from a program P. Let π be the path obtained from G by the process defined above. Then there is M', QP' (such that forall $t, \overline{n}$ we have $\mathsf{QP}'(t)(\overline{n}) = \langle \varepsilon, \varepsilon, \mathtt{nEX}^* \rangle$), and an equivalent path π' (producing the same outcome as π) such that $\mathsf{P}, \mathsf{M}_0, \mathsf{B}_0, \mathsf{A}_0, \mathsf{QP}_0, \varepsilon \Rightarrow^* (\lambda t.\mathtt{skip}), \mathsf{M}', \mathsf{B}_0, \mathsf{A}_0, \mathsf{QP}', \pi'$.

Proof. From above, we have $\mathsf{wf}(\pi)$. This shows that the program configuration can perform the events described by the annotated labels of π. The remaining part of the proof is simply to check that the command rewritings used when deriving the execution graph from P (see Fig. 5.3) can be used as $\mathcal{E}$ transitions in the annotated semantics for P, which follows from the definitions. □

B.5 Operational Semantics and Annotated Semantics

We define forgetful functions from annotated configurations to operational configurations. For memories, we replace the write event by the value written. For labels within annotated configurations, we drop some arguments to recover the data structure of the operational semantics.

$$[\![\cdot]\!]_{\mathrm{M}} : \mathsf{AMem} \to \mathsf{Mem}$$
$$[\![\mathsf{M}]\!]_{\mathrm{M}} \triangleq \lambda x.v_{\mathrm{w}}(\mathsf{M}(x))$$

$$[\![\cdot]\!]_{op} : E^{\mathrm{ext}} \rightharpoonup \left\{ \begin{array}{l} y^{\overline{n}} := x^n, y^{\overline{n}} := v, \mathtt{ack_p}, x^n := y^{\overline{n}}, x^n := v, \\ x := \mathtt{nCAS}(y^n, v, v'), x := \mathtt{nFAA}(y^n, v), \mathtt{cn}, \mathtt{rfence}\ \overline{n} \end{array} \right\}$$

$$\begin{aligned}
[\![\mathtt{lW}(x, v_{\mathrm{w}})]\!]_{op} &\triangleq x := v_{\mathrm{w}} \\
[\![\mathtt{nrW}(\overline{y}, v_{\mathrm{r}})]\!]_{op} &\triangleq \overline{y} := v_{\mathrm{r}} \\
[\![\mathtt{nlW}(x, v_{\mathrm{w}}, \overline{n})]\!]_{op} &\triangleq x := v_{\mathrm{w}} \\
[\![\mathtt{nF}(\overline{n})]\!]_{op} &\triangleq \mathtt{rfence}\ \overline{n} \\
[\![\mathtt{Put}(\overline{y}, x)]\!]_{op} &\triangleq \overline{y} := x \\
[\![\mathtt{Get}(x, \overline{y})]\!]_{op} &\triangleq x := \overline{y} \\
[\![\mathtt{nCAS}(z, \overline{x}, v, v')]\!]_{op} &\triangleq z := \mathtt{nCAS}(\overline{x}, v, v') \\
[\![\mathtt{nFAA}(z, \overline{x}, v)]\!]_{op} &\triangleq z := \mathtt{nFAA}(\overline{x}, v) \\
[\![\mathtt{nlEX}(\overline{n})]\!]_{op} &\triangleq \mathtt{cn} \\
[\![\mathtt{nrEX}(\overline{n})]\!]_{op} &\triangleq \mathtt{ack_p}
\end{aligned}$$

$$\begin{aligned}
[\![\mathtt{F}]\!]_{op} &\text{ is undefined} \\
[\![\mathtt{P}(\ldots)]\!]_{op} &\text{ is undefined} \\
[\![\mathtt{lR}(\ldots)]\!]_{op} &\text{ is undefined} \\
[\![\mathtt{CAS}(\ldots)]\!]_{op} &\text{ is undefined} \\
[\![\mathtt{nlR}(\ldots)]\!]_{op} &\text{ is undefined} \\
[\![\mathtt{nrR}(\ldots)]\!]_{op} &\text{ is undefined} \\
[\![\mathtt{naF}(\ldots)]\!]_{op} &\text{ is undefined} \\
[\![\mathtt{narR}(\ldots)]\!]_{op} &\text{ is undefined} \\
[\![\mathtt{narW}(\ldots)]\!]_{op} &\text{ is undefined}
\end{aligned}$$

$$[\![\cdot]\!]_{opl} : E^{\mathrm{ext}} \rightharpoonup \left\{ \begin{array}{l} y^{\overline{n}} := x^n, y^{\overline{n}} := v, x^n := y^{\overline{n}}, x^n := v, \\ x := \mathtt{nCAS}(y^n, v, v'), x := \mathtt{nFAA}(y^n, v), \mathtt{cn}, \mathtt{rfence}\ \overline{n} \end{array} \right\}$$

$$[\![l]\!]_{opl} = \begin{cases} \mathtt{cn} & \text{if } l = \mathtt{nrEX}(\overline{n}) \\ [\![l]\!]_{op} & \text{otherwise} \end{cases}$$

The labels that cannot appear in a well-formed annotated configuration are not mapped. For put operations, the operational semantics uses both ($\mathtt{ack_p}$) and ($\mathtt{cn}$) while the annotated semantics uses the label `nrEX`, so the mapping is different for labels in $\mathbf{wb_L}$.

$[\![\cdot]\!]_{op}$ and $[\![\cdot]\!]_{opl}$ are extended to lists in an obvious way.

We then extend this to configurations as expected. We overload notations to simplify the formulas.

For $\mathsf{qp} = \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \in \mathsf{AQPair}$, we define $[\![\mathsf{qp}]\!] \triangleq \langle [\![\mathbf{pipe}]\!]_{op}, [\![\mathbf{wb_R}]\!]_{op}, [\![\mathbf{wb_L}]\!]_{opl} \rangle$.

For $\mathsf{QP} \in \mathsf{AQPMap}$, we define $[\![\mathsf{QP}]\!] \triangleq \lambda t.\lambda \overline{n}.[\![\mathsf{QP}(t)(\overline{n})]\!]$.

For $\mathsf{B} \in \mathsf{ASBMap}$, we define $[\![\mathsf{B}]\!] \triangleq \lambda t.[\![\mathsf{B}(t)]\!]_{op}$.

Theorem 7. For all $\mathsf{P}, \mathsf{P}' \in \mathsf{Prog}$, $\mathsf{M}, \mathsf{M}' \in \mathsf{AMem}$, $\mathsf{B}, \mathsf{B}' \in \mathsf{ASBMap}$, $\mathsf{A}, \mathsf{A}' \in \mathsf{RAMap}$, $\mathsf{QP}, \mathsf{QP}' \in \mathsf{AQPMap}$, $\pi, \pi' \in \mathsf{Path}$, if $\mathsf{P}, \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi \Rightarrow \mathsf{P}', \mathsf{M}', \mathsf{B}', \mathsf{A}', \mathsf{QP}', \pi'$ and $\mathsf{wf}(\mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi)$, then $\mathsf{P}, [\![\mathsf{M}]\!]_{\mathrm{M}}, [\![\mathsf{B}]\!], \mathsf{A}, [\![\mathsf{QP}]\!] \Rightarrow \mathsf{P}', [\![\mathsf{M}']\!]_{\mathrm{M}}, [\![\mathsf{B}']\!], \mathsf{A}', [\![\mathsf{QP}']\!]$.

Proof. By straightforward induction on $\Rightarrow$. □

Theorem 8. For all $\mathsf{M} \in \mathsf{AMem}$, $\mathsf{M}'' \in \mathsf{Mem}$, $\mathsf{B} \in \mathsf{ASBMap}$, $\mathsf{B}'' \in \mathsf{SBMap}$, $\mathsf{A}, \mathsf{A}' \in \mathsf{RAMap}$, $\mathsf{QP} \in \mathsf{AQPMap}$, $\mathsf{QP}'' \in \mathsf{QPMap}$, and $\pi \in \mathsf{Path}$, if $\mathsf{P}, [\![\mathsf{M}]\!]_{\mathrm{M}}, [\![\mathsf{B}]\!], \mathsf{A}, [\![\mathsf{QP}]\!] \Rightarrow \mathsf{P}', \mathsf{M}'', \mathsf{B}'', \mathsf{A}', \mathsf{QP}''$ and $\mathsf{wf}(\mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi)$, then there exists $\mathsf{M}' \in \mathsf{AMem}$, $\mathsf{B}' \in \mathsf{ASBMap}$, $\mathsf{QP}' \in \mathsf{AQPMap}$, and $\pi' \in \mathsf{Path}$ such that $[\![\mathsf{M}']\!]_{\mathrm{M}} = \mathsf{M}''$, $[\![\mathsf{B}']\!] = \mathsf{B}''$, $[\![\mathsf{QP}']\!] = \mathsf{QP}''$, and $\mathsf{P}, \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi \Rightarrow \mathsf{P}', \mathsf{M}', \mathsf{B}', \mathsf{A}', \mathsf{QP}', \pi'$.

Proof. By straightforward induction on $\Rightarrow$. In some cases, the reduction enforces a specific annotated label λ and we have $\pi' = \lambda \cdot \pi$; we then need $\mathsf{wf}(\mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi)$ to check that λ is fresh enough for π. □

Theorem 9 (Operational and Annotated Semantics Equivalence)**.** For all program P.

- $[\![\mathsf{M}_0]\!]_{\mathrm{M}}$, $[\![\mathsf{B}_0]\!]$, A_0, and $[\![\mathsf{QP}_0]\!]$ are the initialisation for the operational semantics;
- If $\mathsf{P}, \mathsf{M}_0, \mathsf{B}_0, \mathsf{A}_0, \mathsf{QP}_0, \varepsilon \Rightarrow^* \mathsf{P}', \mathsf{M}', \mathsf{B}', \mathsf{A}', \mathsf{QP}', \pi'$ then $\mathsf{P}, [\![\mathsf{M}_0]\!]_{\mathrm{M}}, [\![\mathsf{B}_0]\!], \mathsf{A}_0, [\![\mathsf{QP}_0]\!] \Rightarrow^* \mathsf{P}', [\![\mathsf{M}']\!]_{\mathrm{M}}, [\![\mathsf{B}']\!], \mathsf{A}', [\![\mathsf{QP}']\!]$
- If $\mathsf{P}, [\![\mathsf{M}_0]\!]_{\mathrm{M}}, [\![\mathsf{B}_0]\!], \mathsf{A}_0, [\![\mathsf{QP}_0]\!] \Rightarrow^* \mathsf{P}', \mathsf{M}'', \mathsf{B}'', \mathsf{A}', \mathsf{QP}''$ then there exists $\mathsf{M}' \in \mathsf{AMem}$, $\mathsf{B}' \in \mathsf{ASBMap}$, $\mathsf{QP}' \in \mathsf{AQPMap}$, and $\pi' \in \mathsf{Path}$ such that $[\![\mathsf{M}']\!]_{\mathrm{M}} = \mathsf{M}''$, $[\![\mathsf{B}']\!] = \mathsf{B}''$, $[\![\mathsf{QP}']\!] = \mathsf{QP}''$, and $\mathsf{P}, \mathsf{M}_0, \mathsf{B}_0, \mathsf{A}_0, \mathsf{QP}_0, \varepsilon \Rightarrow^* \mathsf{P}', \mathsf{M}', \mathsf{B}', \mathsf{A}', \mathsf{QP}', \pi'$.

Proof. The first point comes from unfolding the definitions. The other two are proved by straightforward induction on $\Rightarrow^*$ and using Theorems 7 and 8. The condition $\mathsf{wf}(\mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi)$ is obtained by applying Theorem 2 when needed. □

C Encoding the Declarative Model in Alloy

These models are written for Alloy Analyzer 6.2.0, which is available for download at `https://alloytools.org`.

C.1 Prototype Encoding

```
module rdma_tso

fun memoryLoc[e: Event]: lone Loc {
  { l: Loc | e in MemEvent and loc[e & MemEvent] = l }
}

fun reads: set Event {
  lR + CAS + nlR + nrR + narR
}

fun writes: set Event {
  lW + CAS + nlW + nrW + narW
}

fun nicWrites: set Event {
  nlW + nrW
}

fun nicEvents: set Event {
  nlR + nrW + narR + narW + nrR + nlW + nF
}

fun nicMemEvents: set Event {
  nlR + nrW + narR + narW + nrR + nlW
}

fun instEvents: set Event {
  Event - Write
}

fun ippo: Event -> Event {
  {e1, e2: Event |
    e1 -> e2 in po and
    (
      e1 in lR + lW + CAS + F + P
      or (e1 in nlR + nF and e2 in nicEvents and e1 -> e2 in sqp)
      or (e1 in nrW + narR + narW and e2 in nicEvents - nlR
         and e1 -> e2 in sqp)
      or (e1 in nrR + nlW and e2 in nlW + nF and e1 -> e2 in sqp)
    )
  }
}
```

```
fun oppo: Event -> Event {
  {e1, e2: Event |
    e1 -> e2 in po and
    (
      e1 in lR + CAS + F + P
      or (e1 in lW and e2 in lW + CAS + F + nicEvents)
      or (e1 in nlR + nF and e2 in nicEvents and e1 -> e2 in sqp)
      or (e1 in nrW and e2 in nicEvents - (nlR + nF)
         and e1 -> e2 in sqp)
      or (e1 in narR and e2 in nicEvents - nlR and e1 -> e2 in sqp)
      or (e1 in narW and e2 in nicEvents - (nlR + nlW + nF)
         and e1 -> e2 in sqp)
      or (e1 in nrR + nlW and e2 in nlW + nF and e1 -> e2 in sqp)
    )
  }
}

fun sloc: Event -> Event {
  {e1, e2: Event | memoryLoc[e1] = memoryLoc[e2] }
}

fun sthd: Event -> Event {
  {e1, e2: Event | thread[e1] = thread[e2] }
}

fun sqp: Event -> Event {
  { e1, e2: Event | thread[e1] = thread[e2]
    and node[memoryLoc[e1]] = node[memoryLoc[e2]] }
}

pred totalOrder[s: set Event, r: Event -> Event] {
  all a: s | not a in r[a]
  all a, b, c: s |
    (a -> b in r and b -> c in r) implies a -> c in r
  all a, b: s | a != b implies (a -> b in r or b -> a in r)
}

abstract sig Execution {
  events: set Event,
  po: Event -> Event,
  rf: Event -> Event,
  mo: Event -> Event,
  pf: Event -> Event,
  nfo: Event -> Event,
  rao: Event -> Event
}

one sig E extends Execution {}

fun Events: set Event { E.events }
fun po: Event -> Event { E.po }
fun rf: Event -> Event { E.rf }
```

```
fun mo: Event -> Event { E.mo }
fun pf: Event -> Event { E.pf }
fun nfo: Event -> Event { E.nfo }
fun rao: Event -> Event { E.rao }

fact po {
  all e1: Init | all e2: Event - Init |
    e1 -> e2 in po

  all t: Thread |
    let es = { e: Event | e.thread = t } | totalOrder[es, po]
}

fact rf {
  all r: Event | some w: Event |
    r in reads implies (w in writes and w -> r in rf
                        and w.loc = r.loc
                        and valueWritten[w] = valueRead[r])
}

fact mo {
  all ex: Execution | all x: Loc {
    let wx = { w: writes | w.loc = x } | totalOrder[wx, ex.mo]
  }
}

fact pf {
  all ex: Execution | all p: P | some w: Event |
    w in nicWrites and w -> p in ex.pf
}

fact nfo {
  all ex: Execution | all e1, e2: Event |
    (e1 in nicMemEvents and e2 in nicMemEvents and e1 != e2)
      implies (e1 -> e2 in ex.nfo or e2 -> e1 in ex.nfo)
}

fact rao {
  all ex: Execution | all n: Node {
    let rn = { r: narR | r.loc.node = n } | totalOrder[rn, ex.rao]
  }
}

fun rb: Event -> Event {
  (~rf . mo) - (iden & (Event -> Event))
}

fun rfb: Event -> Event {
  (iden & (lW -> lW)) . (rf & sthd) . (iden & (lR -> lR))
}

fun rfbc: Event -> Event {
  rf - rfb
}
```

```
fun rbb: Event -> Event {
  (iden & (lR -> lR)) . (rb & sthd) . (iden & (lW -> lW))
}

fun imm(r: Event -> Event): Event -> Event {
  { a, b: Event |
      a -> b in r
      and no c: Event |
          a -> c in r and c -> b in r and a != c and b != c }
}

fun ar: Event -> Event {
  (iden & (narW -> narW)) . imm[~po]
}

fun ib: Event -> Event {
  ^(ippo + rf + pf + nfo + rbb + rao)
}

fun ob: Event -> Event {
  ^(oppo + rfb + ((iden & (nlW -> nlW)) . pf) + nfo + mo + (ar . rao))
}

pred preExecution[X:Execution] {
  Event in X.events
}

pred wellFormedExecution[X: Execution] {
  preExecution[X]
  all w1, w2: nlW + nrW + narW | all p2: P |
    ((w1 -> w2) in (po & sqp) and (w2 -> p2) in pf)
      implies (some p1: P | (w1 -> p1) in pf and (p1 -> p2) in po)
  all r: nlR | some w: nrW |
    (r -> w) in imm[po] and valueRead[r] = valueWritten[w]
  all r: nrR | some w: nlW |
    r -> w in imm[po] and valueRead[r] = valueWritten[w]
  all w: nrW | some r: nlR |
    r -> w in imm[po] and valueRead[r] = valueWritten[w]
  all w: nlW | some r: nrR + narR |
    r -> w in imm[po] and valueRead[r] = valueWritten[w]
  all r: narR | some wl: nlW |
    valueRead[r] = valueWritten[wl]
    and (r -> wl in imm[po]
        or some wr: narW |
           r -> wr in imm[po] and wr -> wl in imm[po])
  all w: narW | some r: narR, w2: nlW |
    r -> w in imm[po] and w -> w2 in imm[po]
    and valueWritten[w2] = valueRead[r]
}

pred consistentExecution[X: Execution] {
  wellFormedExecution[X]
```

```
  all e: Event | e -> e not in ib
  all e: Event | e -> e not in ob
  all e: Event |
    e -> e not in ^((iden & (instEvents -> instEvents)) . ib . ob)
}
```

C.2 Example Litmus Test

Alloy encoding of the litmus test Fig. 3.1a.

```
open rdma_tso

one sig x extends Loc {}
one sig y extends Loc {}

one sig n1 extends Node {}
one sig n2 extends Node {}

one sig t1 extends Thread {}
one sig t2 extends Thread {}

fact { t1.node = n1 }
fact { t2.node = n2 }

pred RDMA_seq_a[X:Execution] {
  consistentExecution[X]
  some disj e1, e2, e3, e4, e5: Event {
    e1 in Init and loc[e1] = x and thread[e1] = t1
    and e2 in Init and loc[e2] = y and thread[e2] = t2
    and e3 in lW and loc[e3] = x and valueWritten[e3] = 1 and thread[e3] = t1
    and e4 in nlR and loc[e4] = x and thread[e4] = t1
    and e5 in nrW and loc[e5] = y and thread[e5] = t1
    and e3 -> e4 in X.po
    and e4 -> e5 in X.po
  }
}

run RDMA_seq_a
for exactly 3 Loc, 2 Node, 2 Thread, 5 Event
```