Communications on Applied Nonlinear Analysis
ISSN: 1074-133X
Vol 32 No. 3 (2025)

A Comparative Analysis of Memory Models: SC, TSO, and ARM in Concurrent Programming

Mr. Mohammed H. Abdulwahhab[1], Dr. Parosh Aziz Abdulla[2], Dr. Karwan Jacksi[3]

[1]Assistant Lecturer, Department of Computer Science, University of Zakho, Duhok, Iraq, mohammad.abdulwahab@uoz.edu.krd.

[2]Chaired Professor, Department of Information Technology, Uppsala University, Uppsala, Sweden, parosh@it.uu.se.

[3]Assistant Professor, Semantic Web Lab, Research Center, University of Zakho, Duhok, Iraq, karwan.jacksi@uoz.edu.krd.

Article History:

Received: *24-07-2024*

Revised: *11-09-2024*

Accepted: *30-09-2024*

Abstract:

Concurrent programming has long been a challenge due to the complex interactions between processors and shared memory. The intuition of programmers often diverges from the actual behavior of concurrent programs due to factors such as modern processor technologies, compiler optimizations, and performance-driven data store implementations. To address this discrepancy, memory models were introduced to formally define the permissible and prohibited behaviors in concurrent program execution. Memory models can be broadly categorized into strong and weak models. Sequential Consistency (SC) is a strong model that preserves the order of events in each thread or process, aligning with programmers' expectations. In contrast, weak models, such as Total Store Order (TSO) and ARM, allow for relaxed ordering, potentially leading to unexpected behaviors. This paper presents a practical comparison of SC, TSO, and ARM memory models. A suite of six litmus tests were executed on architectures implementing each model to highlight their differences. The results were analyzed using the axiomatic models of each architecture to understand why certain behaviors were accepted or rejected. The herd tool was employed to simulate memory model behavior. The findings revealed a significant disparity in the acceptance of behaviors across the three models. SC emerged as the most restrictive, followed by TSO, while ARM demonstrated the least stringent acceptance criteria. These results underscore the importance of understanding memory model nuances when developing concurrent programs to ensure correct and predictable behavior.

Keywords: Concurrent Programming, Memory Models, Sequential Consistency (SC), Total Store Order (TSO), ARM, Litmus Test.

1. Introduction

Concurrency, the interleaving of logical control flows in multithreaded or multi-process programs, has become an integral part of modern software development[1]. Concurrent programs involve multiple processes or threads working concurrently with shared objects, such as shared memory[2], [3]. The foundational concepts of concurrency were introduced by Edsger Dijkstra in the mid-1960s to construct reliable operating system kernels[4], [5]. Since then, concurrency has found applications in various domains, including I/O devices, human interaction, computer networks, servers, parallel or multi-core computing, robotics, database management systems, and industrial systems[1], [6], [7], [8], [9].

The benefits of concurrency are multifaceted. It improves system responsiveness by executing tasks asynchronously rather than synchronously. It is essential for leveraging modern multi-core CPUs, which offer superior performance over single-core processors while consuming less power[6]. From a program modularity perspective, concurrency logically separates independent control flows[10], [11]. However, the execution of concurrent programs on modern CPUs is not guaranteed to follow the order of events as assumed by programmers. This is due to techniques like cache hierarchies and store buffers, which are employed to optimize performance. Programmers often assume a Sequential

Consistency (SC) memory model[12], which preserves the order of event execution within each process. However, this assumption does not hold in non-SC models prevalent in modern CPUs[13]. Consequently, running concurrent programs on multi-core CPUs may deviate from programmer expectations, necessitating careful consideration of these architectures over SC[12].

Unlike sequential programs, concurrent programs exhibit a range of behaviors due to the interleaving and reordering of read and write operations across threads. These behaviors can be categorized into permitted behaviors, which align with a specified memory model, and prohibited behaviors, which are unattainable[14]. Understanding these behaviors on a specific architecture is crucial for programmers[15]. To address this challenge, Memory Consistency Models (MCMs) were introduced to depict execution on a given architecture and enable programmers to write concurrent programs that avoid undesired behaviors. MCMs define the order in which shared variables are accessed, governing permissible concurrent program behavior[16].

MCMs are categorized into strong and weak models. Sequential Consistency (SC) is a strong model that ensures the write operation is executed as an atomic operation and the order of operations within each thread or process is preserved[13]. In contrast, relaxed memory models, provided by modern multiprocessors and high-level programming languages, reorder memory access operations. This reordering can improve performance but also introduces challenges in understanding and predicting program behavior[17], [18], [19].

1.1 Simulation of MCMs

Numerous simulation and verification tools have been proposed for MCMs[20]. One such tool is herd tools[21], which enables users to define their own memory models using the cat language[22], a domain-specific language commonly employed to describe parallel program consistency properties[23]. Litmus tests, concise programs featuring multiple threads with shared variables, can then be executed on these custom models. Each litmus test includes an assertion that validates the final values of local variables, illustrating the test's behavior. Upon completion of a litmus test's execution, the assertion is evaluated to determine whether a specified behavior, as defined by a predefined MCM, is permissible or not[20], [21], [22].

In the subsequent sections, six litmus tests are presented to demonstrate the functioning of memory models and analyze their behavioral variations. These litmus tests are described in detail, accompanied by pseudocode of each one of them.

1.2 Litmus Tests Used in The Comparison

In the subsequent sections, six litmus tests are presented to demonstrate the functioning of memory models and analyze their behavioral variations. These litmus tests are described in detail, accompanied by pseudocode of each one of them.

1.2.1 Message Passing (MP) Litmus Test:

In the MP litmus test, two threads (T_0 and T_1) run simultaneously, with two shared memory locations (x and y) initialized to a value of 0 for inter-thread communication. Thread T_0 executes two store operations: first, it stores the value 1 in location x, followed by storing the value 1 in location y according to the program order. In contrast, T_1 loads the content of location y into register r_1 and then loads the contents of location x into register r_2. It is crucial to note that memory locations x and y are shared between both threads, whereas registers r_1 and r_2 are private to thread T_1. The message-passing pattern between threads is illustrated in Figure 1a, where T_0 updates the value of x, subsequently activating the flag y. T_1, in turn, waits for flag y to be enabled before reading the value of x. For

simplicity, the waiting mechanism of T_1 was omitted. The objective of the MP litmus test is to ascertain whether T_1 can read the value 1 from the shared location y while x remains at a value of 0 [24].

1.2.2 S Litmus Test:

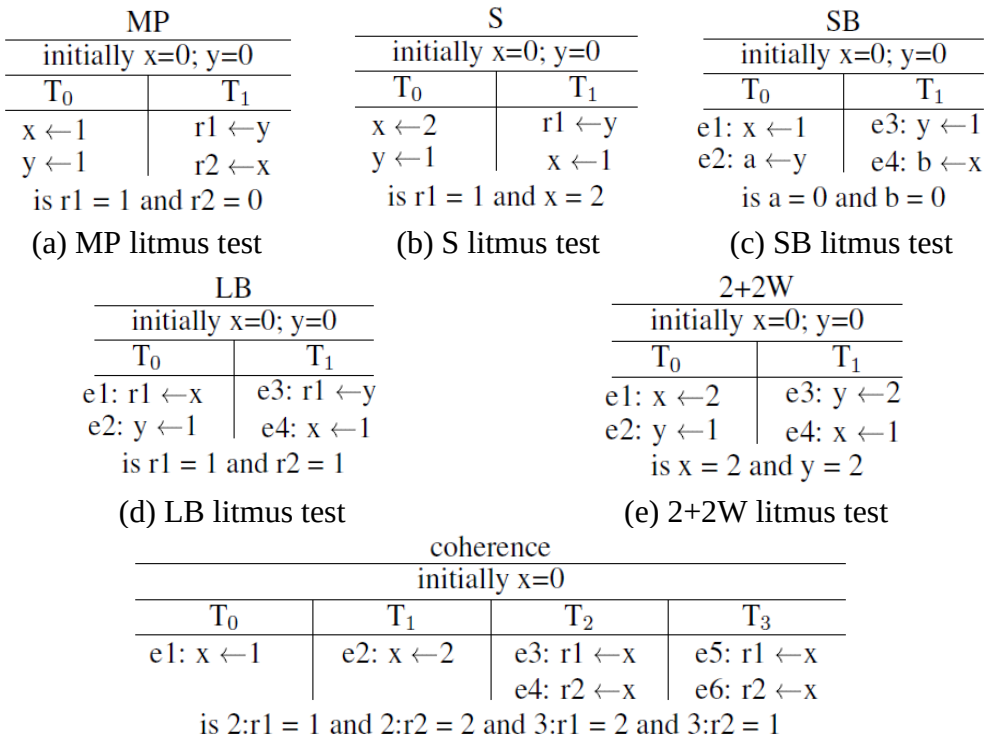
It is an adaptation of the MP litmus test; it differs from the MP litmus test by writing value 2 to the shared location x by the first event in T_0 in addition to writing the value 1 to the shared location x instead of the second read event in T_1. Its' constraint involves checking whether the second event in T_1 is committed after the first write of T_0 or not. It aimed to check the validity of the control dependency between read-to-write events. The S litmus test's psuedo-code is shown in Figure 1b [24].

1.2.3 Store Buffer (SB) Litmus Test:

This litmus test involves two threads (T_0 and T_1) running concurrently. It has two shared variables (x and y) between the two threads. Variable a is a private variable for T_0, whereas variable b is a private variable for T_1. Initially, all shared and private variables are set to 0. T_0 writes the value 1 to the shared variable x; after that, the value of y is read into private variable a. Meanwhile, T_1 writes the value 1 to the shared variable y, followed by loading the value of x into the private variable b. The constraint of this litmus test assesses whether the two read events retrieve values from the initial state or from the write events within T_0 and T_1. Figure 1c shows the SB litmus test's pseudo-code [24].

1.2.4 Load Buffer (LB) Litmus Test:

In contrast to the SB litmus test, the two threads (T_0 and T_1) in the LB litmus test first perform read operations from the shared locations x and y, followed by writing the value 1 to the same shared locations y and x, respectively. It seeks to determine whether read events read from subsequent write events or from the initial values of x and y. The LB litmus test's pseudo-code is shown in Figure 1d [24].



(a) MP litmus test (b) S litmus test (c) SB litmus test

(d) LB litmus test (e) 2+2W litmus test

(f) Coherence litmus test

Figure 1: Pseudo-code of 6 litmus tests which used in the comparison.

1.2.5 2+2W Litmus Test:

It is a variation of the LB litmus test, where the read event of T_0 is substituted by writing value 2 to the shared location x, and similarly, replaces the read event in T_1 with writing the value 2 to the shared location y. Its purpose is to scrutinize the executing order of events in threads; in other words, determining if the final values of both x and y are 1 or 2. Figure 1e illustrates the pseudo-code of the 2+2W litmus test [24].

1.2.6 Coherence Litmus Test:

One shared variable, x, is the focal point of this litmus test. There are four threads; T_0 changes the value of x with 1, and T_1 updates it by 2. The value of x will be loaded into the private registers of T_2 and T_3 twice. This litmus test is intended to determine whether or not T_3 reads the value of x in the reverse order of T_2 while reading the data. The constraint of this litmus test is to determine whether it is feasible for T_2 to read value 1 of the first read before reading value 2 of the second read and T_3 to read value 2 of x before reading value 1 of the second read. In other words, this litmus test focuses on the consistency of data. The coherence litmus test's pseudo-code is shown in Figure 1f [24].

2. Axiomatic model in depth

Axiomatic memory model refers to precise definition of a memory model. It is language independent as it mathematically describes multi-threaded programs in terms of events and their relationships. Distinguishing between allowed and disallowed behaviors is achieved by applying a set of rules or constraints to these relationships which considered as relations. The ability to implement its concepts in tools for verifying concurrent programs sets this memory model apart. However, it's worth noting that understanding memory models specified in this way can sometimes be challenging. [25]. Using an axiomatic model to classify behaviors of concurrent programs into allowed and disallowed behaviors involves three essential steps [21]. To demonstrate, these steps have been applied to the MP litmus test, illustrated in Figure 1a. The required steps are:

a. Control-flow semantics: in this step, instructions in a concurrent programs will be translated into events, for example the instruction $x \leftarrow 1$ contained in T_0 will be translated into a write event as: Wx = 1, while the instruction $r_1 \leftarrow y$ in T_1 will be Ry = ?. A write event is abbreviated by the letter (W) and a read event by the letter (R), followed by the variable name that was written to or read from. The question mark in the event Ry = ? refers to an undetermined read value that is because it depends on the interleaving between events from multiple threads. Branching decisions events and fences will be determined in this step. The output of creating the control-flow semantic of the MP litmus test is depicted in Figure 2.

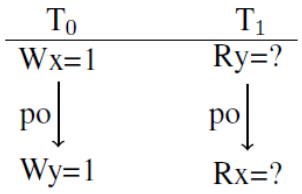


Figure 2: Control flow semantics of MP litmus test.

In this litmus test, there are two implicit write events representing the initiation of locations x and y with value 0. In this stage, the program order (*po*) relation will be determined as well. It is represented as arrow with *po* title in Figure 2. The program order shows the order of events within a thread with respect to decision branching and address, data and control dependencies. In other words, *po* represent the order of events which is corresponds to instructions in which a program was written. For example

in Figure 2, T_0 has two write events related by a *po* arrow, that is means that at the code level the Wx = 1 written before Wy = 1.

b. Data flow semantics: a set of data flow semantics will be derived for each control flow in this stage. All possible interleaving between events from different threads will be taken into account. Readfrom (*rf*) and coherence order (*co*) relations will be extracted in this level. The *rf* relation describes the sources of values to be read by a read event. Referring to Figure 3b, the Ry = 1 event in T_1 of the MP litmus test will read the value 1 from the Wy = 1 event of T_0. In other words, the read event Ry = 1 of T1 will be executed after the Wy = 1 event of T0. The rf relation could be illustrated as arrows with *rf* captions, beginning with the write events, which are the source of reading values, and ending with the read events. The *rf* arrow with no source like the Ry = 0 event of T_1 in the Figure 3a means that the read event will take its' value from the implicit initial write of location *y*. The *co* relation describes the order of writes events that hits the same memory location. In the current example, the initial write of *x* (not showed in the figure) took the effect before the Wx=1 of T_0, so the two write events are ordered in coherence. The *co* relation could be explained as arrows titled by *co* caption. Shortly, Figure 3 illustrates all possible executions (candidate executions) of the program showed in Figure 1.

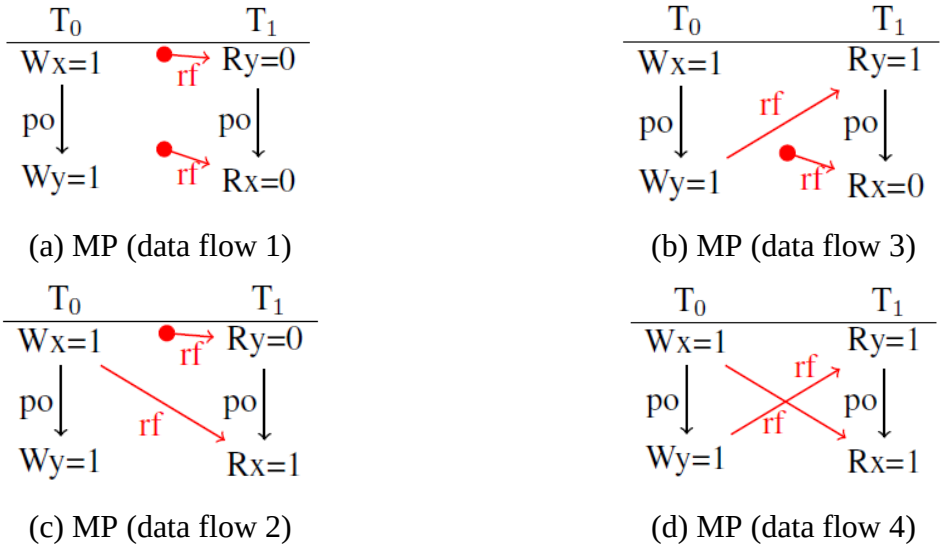


Figure 3: Data-flow semantics for the control-flow semantics given in Figure 2.

c. Constraints specification: for a given memory model, a set of constraints should be determined in a form of empty or acyclicity or irreflexivity of various combinations of the (*po*, *rf* and *co*) relations over events given by the candidate execution. Checking these constraints will tell if a given candidate execution is valid or not.

Up to this point, three relations (*po*, *rf*, and *co*) have been derived from the previous steps. Concurrent programs verification tools, such as herdtools [21] extract these relations alongside some other relations outlined in Table 1. Table 1 describes all relations provided by herd tools, extracted from a given concurrent program.

Table 1: Provided relations from herdtools [21].

Relation Name	Description
loc	The association between events accessing the same memory location.
int	Contains all pairs of events that belong to the same thread.

lxsx	Load exclusive store exclusive relation, contains pairs of read of the load exclusive and write of the successful store exclusive.
amo	A relation that contains pairs of atomic read and atomic write operations.
rmw	Read modify write operations relation

Further relations can be derived from the (*po*, *rf*, and *co*) relations, these relations which are listed in Table 2 could be derived by applying some mathematical operations on previously described relations. The necessary mathematical operations include: (union, intersection, difference, complement, inverse, reflexive closure, transitive closure, and reflexive transitive closure)[23]. Table 2 explains some of required relations for the (SC, TSO and ARM) MCMs and how to derive these relations [21].

Table 2: Derived relations that are required for the specification of SC, TSO and ARM model specification [21].

Relation Name	Derived by	Description
po-loc	*po & loc*	po relation restricted to the same location.
fr	*rf−1;co*	from-read relation makes one step of reads-from backward, then one step of coherence.
ext	*∼int*	Contains all pairs of events that belong to different threads.
rfe	*rf & ext*	External read from.
coe	*co & ext*	External coherence.
fre	*fr & ext*	External from-read.

3. Compared Memory Models

In the following subparagraphs is the description of the compared memory models (SC, TSO, and ARM).

3.1 Sequential Consistency (SC) Model

Sequential consistency, as defined by Lamport [13] which it gives what a programmer expects from running concurrent programs. It places significant emphasis on the order of memory requests, mirroring their appearance in the code. In other words, SC avoids reordering memory access operations. As a result, it stands out as an intuitive and robust memory model compared to others. Two specific requirements must be met for a system to be considered sequentially consistent:

Requirement 1: ”Each processor issues memory requests in the order specified by its program.”

Requirement 2: ”Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue.”

While the requirement 1 emphasizes the order of operations in individual processor, the second requirement ensures the preservation of the order of operations from different processors [13].

SC model could be defined as an axiomatic model as the following [24]:

1 empty rmw & (fre;coe) as atom

2 acyclic po | fr | rf | co as sc

The first statement emphasizes that the atomicity of memory access operations by checking the intersection between *rmw* relation and both *fre* and *coe* relations which should be an empty set, meaning that memory access operations should take effect on memory immediately, which is a basic principle in SC model. The second statement focuses on the forbidden reordering of events in SC

model, which defined as a combination between all the three relations (*fr*, *rf* and *co*), the resulted relation from the combination should be acyclic.

3.2 Total Store Order (TSO)

In order to achieve improved efficiency and performance, modern microprocessors have departed from the SC approach. To prevent a thread from stalling while waiting for a write operation on shared memory to take effect, current designs employ sophisticated technologies such as cache hierarchies and store buffers. One such MCM is Total Store Ordering (TSO). TSO serves as a formalization of SPARC and Intel x86 architectures [12].

TSO allows event reordering in one case: when there is a store then load operations for different variables or memory addresses, it could be executed as load then store due to the store operation latency in the FIFO store buffer. As a result, TSO allows some non-SC executions [15]. The axiomatic formalization of TSO in cat language is depicted bellow [24], [26].

1 (Uniproc check specialized for TSO *)*

*2 irreflexive po-loc & (R*W); rfi as uniprocRW*

*3 irreflexive po-loc & (W*R); fri as uniprocWR*

4 (Communication relations that order events*)*

5 let com-tso = rfe | co | fr

6 (Program order that orders events *)*

*7 let mfence = po & (* MFENCE) ; po*

*8 let po-tso = po & (W*W | R*M) | mfence*

9 (TSP global-happens-before *)*

10 let ghb = po-tso | com-tso

11 acyclic ghb as tso

From the axiomatic model perspective, the difference between the semantics of SC and TSO lies in the acyclicity constraint verification. In TSO, this verification does not encompass the checks for write-to-read pairs, whereas in SC, it does. The explanation of the TSO axiomatic model is listed below:

– lines (2-3): in this part the program order for the same memory location is maintained for two cartesian products (Read * Write, and Write * Read). This program order should not be reflexive.
– line 5: in this declaration the communication relation *com-tso* will be created by the union of (*rfe*, *co*, and *fr*) relations. If the *rfi* included in the communication relation then the model will be stronger which means it will forbid some allowed behaviors in the x86 architecture, for that reason only the *rfe* relation which is a sub-relation of *rf* realtion is included in the communication relation.
– line 7: for taking the memory barriers commands into account.
– line 8: for creating the (*po-tso*) relation which is resulted from the union of (write to write pairs and read to read or write pairs with the exclusion of write to read pairs) plus the mfences which is created in line 7. The creation of the cartesian pairs depending on predefined sets which are (W: write events, R: read events, and M: memory events (read and write)).
– line 10: creating the global-happens-before (*ghb*) relation by combining the *po-tso* and *com-tso* relations.
– line 11: for checking the acyclicity constraint of the *ghb* relation.

3.3 ARM Model

ARM MCM was generally formalized by the use of cat language [27]. In the original formalization of ARM, checking three criterion is required to categorize the candidate executions into allowed and disallowed executions. These criterions are (the internal visibility, the atomicity, and the external visibility). The ARM axiomatic model requires a special function called intervening-write and a set of basic relations to be used in checking the previously mentioned criteria. Following is the description and the code in the cat language of the intervening-write function.

intervening-write(r) function: this function will specify write event w which interleaving between two read events. it could be declared as:

let intervening-write(r) = r; [W]; r

In addition, specification of the axiomatic model of the ARM MCM requires some derived relations, following are the descriptions of them:

– Local read successor (*lrs*) relation: which contains the next read event *r* of the direct previous write event *w*. Events *r* and *w* should be two events accessing the same memory location. This relation could be derived as follow:

let lrs = [W]; (po-loc \ intervening-write(po-loc)); [R]

– Local write successor (*lws*) relation: which contains all writes events that are follow the write event *w*. All these writes events should write to the same memory location. It could be generated as:

let lws = po-loc; [W]

– Coherence after (*ca*) relation: which is the union of *rf* and *co* relations, a write event *w* is coherence after an event *e* (read or write) if event *w* changes the value that event *e* deals with. It could be expressed in the cat language as:

let ca = fr | co

– Observed by (*obs*) relation: which gathers the external read-from (*rfe*), external from-read (*fre*) relations, and the external coherence order (*coe*). It is clear that this relation maintains events from different threads over shared memory. Its' code is:

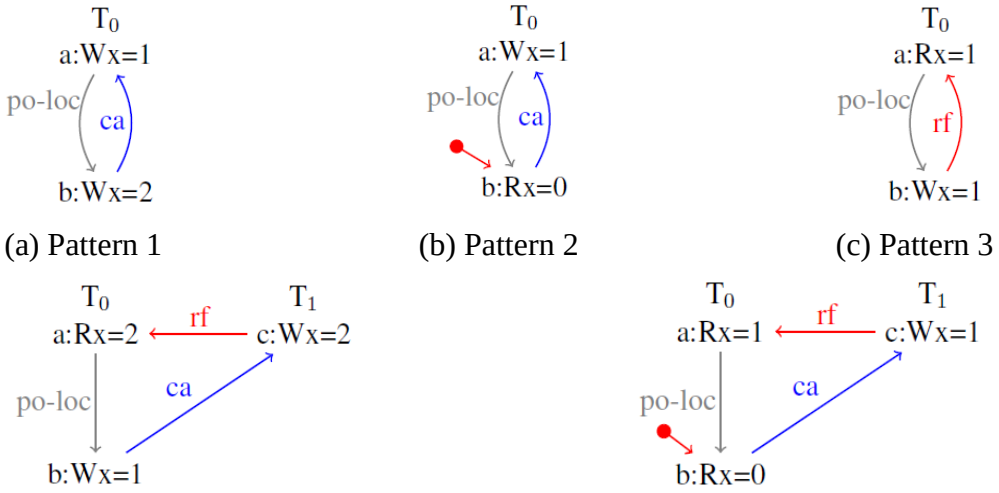
let obs = rfe | fre | coe

Following paragraphs describing the three criterions required to check if a candidate execution is allowed or forbidden in the ARM model:

a. The internal visibility: it could be simplified as the checking of the correctness of the interactions between threads and inside each thread to prevent undesirable behaviors like (lack of coherence, reading from future write event, and ignoring some write events). This constraint could be achieved by ignoring any candidate execution that contains a cycle in the union of (*po-loc* and *ca* and *rf* relations). Figure 4 depicts the five forbidden patterns of behaviors according to the internal visibility constraint. All patterns showed in Figure 4 contain forbidden cycles for that they are forbidden executions in the ARM model. This constraint could be expressed in the cat language by the following expression:

(Internal visibility requirement *)*

acyclic po-loc | ca | rf as internal



(a) Pattern 1 (b) Pattern 2 (c) Pattern 3

(d) Pattern 4 (e) Pattern 5

Figure 4: The five forbidden behaviors in the ARM model based on the internal visible constraint.

Referring to Figure 4, the pattern 1 imposes that when there are two write events to the same location in the program order they should have the same order in the ca relation as in *po-loc*. In pattern 2, reading event *b* is denied from reading from an overridden event (like writing the initial value) when there is a write event like event a followed immediately by a read event like event *b*. Pattern 3 forbids a read event like event *a* to read from a future write in the program order such as event *b*. Pattern 4 forbids a read event such as event *a* to read from a write event such as event *c* which is coherence after a write event *b* where event *b* is after event *a* in the program order, in other words this pattern is another form of reading from a future write. Finally pattern 5 illustrates when a reading event like event *a* read from a write event *c*, then all successor reads after the event *a* should read from event *c* or other subsequent write events after event *c*. It is unrealistic that read event *b* reads from the initial value which is overwritten by event *c* and it have been read by event *a* which is before event *b* in the program order.

b. The atomicity: in this constraint the ARM model focuses on the load exclusive (LDXR) and store exclusive (STXR) instructions which are special instructions provided by ARM and POWER architectures. These two instructions are paired together to provide concurrent programs with atomic access to the shared locations. Successful (LDXR) and (STXR) should not be interrupted or interleaved by any other write access to the same location from another thread. Figure 5 illustrates a failed (STXR) caused by interleaving write access to the location *x* from thread T_1 between the (LDXR) and (STXR) of thread T_0. The * mark in the events *a* and *b* means that this read-write pair is an atomic operation. This constraint does not cover just the (LDXR) and (STXR) instructions but the swap (SWP) and compare-and-swap (CAS) atomic operations as well, which are already contained in the (*amo*) relation while (LDXR) and (STXR) pairs are already contained in the *lxsx* relation. Both (*amo*) and (*lxsx*) relations are described in Table 1. In the definition of the ARM model both (*amo*) and (*lxsx*) relations merged in one relation called read-modify-write (*rmw*). The *rmw* relation could be defined in cat language as:

let rmw = lxsx | amo

Failed atomic operations contained in the (*rmw*) relations happens if there is write event from another thread between the atomic read-write pair. For that, the atomicity constraint could be verified by checking if there is an external write event which is coherence-after the read of the atomic operation and coherence-before the write of the same atomic operation. By using the cat syntax, it could be

checked by checking if the intersection between the (*rmw*) relation and both the *fre* and *coe* relations is empty. Defining this constraint in the cat language is as follows:

(Atomic: Basic LDXR/STXR constraint to forbid intervening writes. *)*

empty rmw & (fre; coe) as atomic

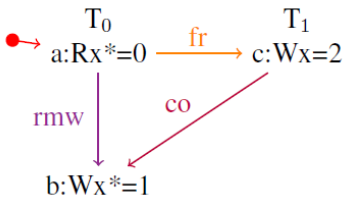


Figure 5: Failed (STXR) instruction according to the atomicity constraint in the ARM model.

c. External visibility: which filters candidate executions in a way that excludes undesirable executions that their synchronization of events in a thread eventually gives forbidden results and affects other threads. In this constraint, ARM model contains two building blocks: locally-ordered-before (*lob*) relation and observed-by (*ob*) relation. Following are the details of each relation:

1. *lob* relation: it is a recursive relation that contains all possible ways to synchronize two events within one thread. In cat language it could be generated as follows:

(Locally-ordered-before *)*

let rec lob = lws | dob | aob | bob | lob; lob

As explained previously, *lob* relation should be derived from the union of other relations, these relations will be described in the following:

– Dependency order before (*dob*) relation which gathers all possible chains of dependencies provided by the ARM architecture. It could be generated in cat language as follows:

let dob = addr | data | ctrl; [W] | (ctrl | (addr; po)); [ISB]; po; [R] | addr; po; [W] | (addr | data); lrs

– Atomic order before (*aob*) relation which states how to make use of exclusive pairs (LDXR/STXR) and atomic operations to provide order. Following is the syntax of *aob* relation in cat language:

let aob = Rmw | [W & range(rmw)]; lrs; [A | Q]

– Barrier order before (*bob*) relation which gathers all possible ways to use fences to provide order. In cat language it could be derived as following:

let bob = po; [dmb.full]; po | po; ([A];amo;[L]); po | [L]; po; [A] | [R]; po; [dmb.ld]; po | [A | Q]; po | [W]; po; [dmb.st]; po; [W] | po; [L]

2. *ob* relation: is an irreflexive relation that contains all ways of the interaction between two threads. It could be defined in cat language by the following:

(Ordered-before *)*

let rec ob = obs | lob | ob; ob

following snippet of code is to check the external visibility, it is obvious, the *ob* relation should be irreflexive.

(External visibility requirement *)*

irreflexive ob as external

Finally, each candidate execution that passes the three constraints will be considered as allowed execution whereas failing in one of the previous criterion will be considered as forbidden execution [27].

4. Comparison between the SC, TSO, and ARM Memory Models

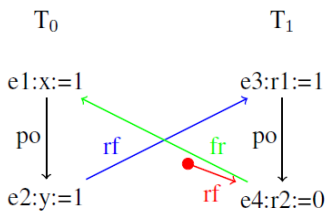
In this section, we will compare the (SC, TSO, and ARM) memory models by distinguishing the behaviors obtained from executing the previously mentioned litmus tests and discussing why we can get such behaviors while it is impossible to obtain other behaviors. A note should be made that the experiments presented in this section were conducted using the Herd Tools 7 tool. The following paragraphs are the explanation of the experiments conducted in the three architectures described previously:

4.1 MP litmus test:

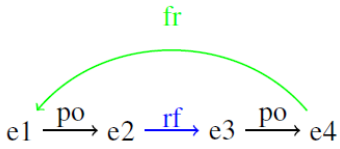
Returning to the MP litmus test depicted in the Figure 1, the result column of the Table 3 shows all potential values of r_1 and r_2 arising from the overlapping execution of the events. The permissible behaviors when running this litmus test under the SC or TSO architecture are a, b, and c, whereas the outcome d is prohibited. While doing this litmus test with the ARM architecture, all results can be obtained. As was already explained, this litmus test determines whether or not the register r_1 contains the value 1 while the value 0 may be retrieved to register r_2. When examining the cause of not getting the result d in the SC architecture and making reference to the SC axiomatic model, we will notice that getting such behavior as shown in the Figure 6a leads to a forbidden cycle which is depicted in Figure 6b. This goes against the SC's semantics that there be no cycle in the union of relations *po* | (*fr* | *rf* | *co*). As for the talk about not accepting the result d in the TSO architecture, the reason is due to the presence of a cycle in the relation *ghb*, shown in Figure 7b which denies accepting such behavior according to the TSO axiomatic model. The flow semantics of this litmus test running under the TSO architecture is shown in Figure 7a. However, since all the requirements for permissible behavior in the ARM axiomatic model are met, behavior d is acceptable in the ARM architecture.

Table 3: MP behaviors in SC, TSO, and ARM architectures.

Behavior	Result	SC	TSO	ARM
a	r1=0,r2=0	Allowed	Allowed	Allowed
b	r1=0,r2=1	Allowed	Allowed	Allowed
c	r1=1,r2=1	Allowed	Allowed	Allowed
d	r1=1,r2=0	Disallowed	Disallowed	Allowed



(a) MP flow semantics in SC.



(b) MP forbidden cycle in SC.

Figure 6: MP litmus test under SC.

(a) MP flow semantics in TSO. (b) MP forbidden cycle in TSO.

Figure 7: MP litmus test under TSO.

4.2 S litmus test:

Table 4 displays the results of running the S litmus test under the SC, TSO, and ARM architectures. It should be observed that the result d indicates that the first event of T_0 is committed after all other events of both T_0 and T_1. Both the SC and TSO designs restrict this sort of behavior. According to the axiomatic models of both SC and TSO, the absence of this behavior in the two architectures indicated is caused by the occurrence of forbidden cycles. Figure 8 depicts the forbidden cycle found in behavior d in the SC, while Figure 9 depicts the forbidden cycle found in the TSO. In contrast, all behaviors are permitted by the ARM architecture since all flow semantics for all behaviors do not go against the axiomatic model of ARM's behavior acceptance requirements.

Table 4: S behaviors in SC, TSO, and ARM architectures.

Behavior	Result	SC	TSO	ARM
a	x=1,r1=0	Allowed	Allowed	Allowed
b	x=2,r1=0	Allowed	Allowed	Allowed
c	x=1,r1=1	Allowed	Allowed	Allowed
d	x=2,r1=1	Disallowed	Disallowed	Allowed

(a) S flow semantics in SC. (b) S forbidden cycle in SC.

Figure 8: S litmus test under SC.

(a) S flow semantics in TSO. (b) S forbidden cycle in TSO.

Figure 9: S litmus test under TSO.

4.3 SB litmus test:

Going back to the SB litmus test, it aims to determine whether it is possible to read each of T_0 and T_1 of the shared variables x and y from initial values through events e_2 and e_4 while ignoring writing operations on the variables x and y that occur before the reading process through events e_1 and e_3. The outcomes that can be attained by running this litmus test on SC architecture are represented in Table 5 as a, b, and c. While both TSO and ARM architectures can be used to achieve all results. Due to the existence of a banned loop, as indicated in Figure 10b, the behavior of result d is prohibited in the SC design. The absence of a forbidden loop in the semantic flow for each of the TSO and ARM architectures from the perspective of the axiomatic model is a justification for accepting the outcome d. When this test is run, the semantic flow of the TSO architecture is depicted in Figure 11. It is obvious that there is no forbidden cycle in the *ghb* relation.

Table 5: SB behaviors in SC, TSO, and ARM architectures.

Behavior	Result	SC	TSO	ARM
a	a=0,b=1	Allowed	Allowed	Allowed
b	a=1,b=0	Allowed	Allowed	Allowed
c	a=1,b=1	Allowed	Allowed	Allowed
d	a=0,b=0	Disallowed	Allowed	Allowed

(a) SB flow semantics in SC. (b) SB forbidden cycle in SC.

Figure 10: SB litmus test under SC.

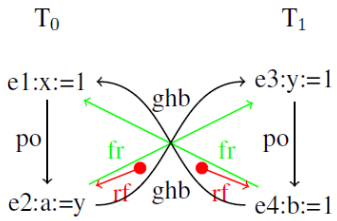


Figure 11: SB flow semantics in TSO.

4.4 LB litmus test:

The outcomes of conducting the LB litmus test in the SC architecture are identical to those achieved when running it in the TSO architecture. Due to the existence of a prohibited cycles in the flow semantics, which are depicted in Figure 12 and 13 for both SC and TSO structures, respectively. Both architectures disallow result d, while behaviors a, b, and c which are shown in Table 6 are allowed in the both SC and TSO models. The ARM architecture, on the other hand, may implement this litmus test and receive all results since the flow semantics does not contain any violations of the requirements for acceptable behaviors in the ARM architecture.

Table 6: LB behaviors in SC, TSO, and ARM architectures.

Behavior	Result	SC	TSO	ARM

a	0:r1=0,1:r1=0	Allowed	Allowed	Allowed
b	0:r1=0,1:r1=1	Allowed	Allowed	Allowed
c	0:r1=1,1:r1=0	Allowed	Allowed	Allowed
d	0:r1=1,1:r1=1	Disallowed	Disallowed	Allowed

(a) LB flow semantics in SC. (b) LB forbidden cycle in SC.

Figure 12: LB litmus test under SC.

(a) LB flow semantics in TSO. (b) LB forbidden cycle in TSO.

Figure 13: LB litmus test under TSO.

4.5 2+2W litmus test:

While this litmus test seeks to check whether the order of writing events execution in the threads is maintained or not, we observe that the results that can be obtained from running it in both SC and TSO architectures are the same results, which are results a, b, and c which are shown in the Table 7, while result d is not allowed. The requirement of accepting the behavior of both structures is not met in either of the two designs given. When this test is applied to the SC design, Figure 14 depicts a violation of the acceptance requirement. However, the prohibited cycle depicted in Figure 14b violates the SC model's behavior acceptance requirement. On the other hand, Figure 15 depicts a behavior that is prohibited by the TSO architecture because of the cycle that is depicted in Figure 15b. Regarding the ARM architecture, all outcomes are possible and acceptable.

Table 7: 2+2W behaviors in SC, TSO, and ARM architectures.

Behavior	Result	SC	TSO	ARM
a	x=1,y=1	Allowed	Allowed	Allowed
b	x=1,y=2	Allowed	Allowed	Allowed
c	x=2,y=1	Allowed	Allowed	Allowed
d	x=2,y=2	Disallowed	Disallowed	Allowed

(a) 2+2W flow semantics in SC. (b) 2+2W forbidden cycle in SC.

Figure 14: 2+2W litmus test under SC.

(a) 2+2W flow semantics in TSO. (b) 2+2W forbidden cycle in TSO.

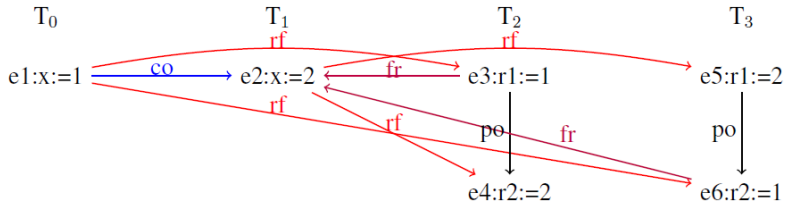
Figure 15: 2+2W litmus test under TSO.

4.6 CO litmus test:

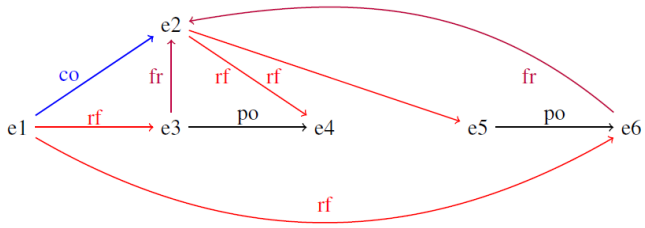
All comparable MCMs in this litmus test ensured data consistency by forbidding the production of any result that would violate data consistency. Table 8 demonstrates that in the aforementioned designs, it is not possible to obtain different readings. The SC model's flow semantics for this litmus test is shown in Figure 16a, while Figure 16b explains why the SC model would reject such behavior because it contains a banned cycle. Figure 17 demonstrates that the TSO design makes it impossible to achieve this result. According to the axiomatic model of the ARM architecture the second and third conditions are acceptable when verifying this litmus test. Acceptance of such behavior is hampered by the first condition where the relation *po-loc* | *ca* | *rf* which is denoted as *IV* in Figure 18 contains a forbidden cycle. Figure 18a illustrates the flow semantics of this litmus test when run on the ARM architecture, whereas Figure 18b illustrates the banned cycle on the same architecture.

Table 8: CO behaviors in SC, TSO, and ARM architectures.

Behavior	Result	SC	TSO	ARM
a	2:r1=1,2:r2=2,3:r1=2,3:r2 =1	Disallowed	Disallowed	Disallowed

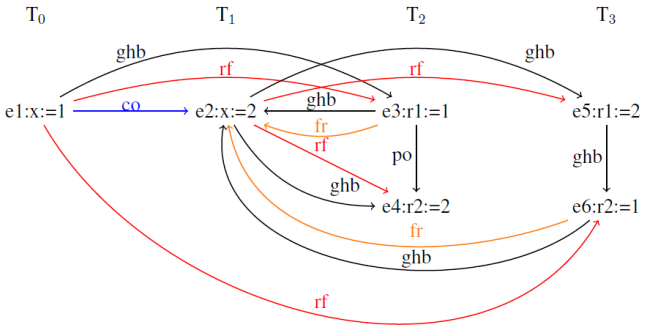


(a) CO flow semantics in SC.

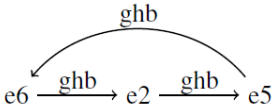


(b) CO forbidden cycle in SC.

Figure 16: CO litmus test under SC.

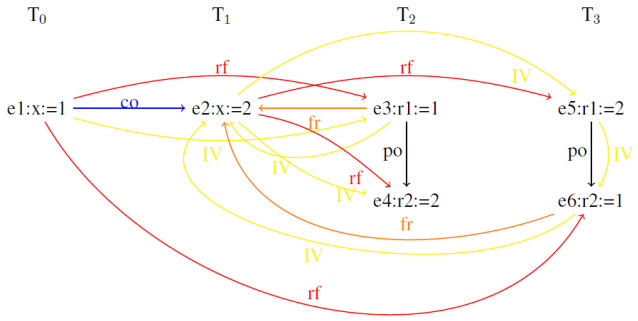


(a) CO flow semantics in TSO.

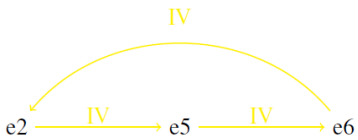


(b) CO forbidden cycle in TSO.

Figure 17: CO litmus test under TSO.



(a) CO flow semantics in ARM.



(b) CO forbidden cycle in ARM.

Figure 18: CO litmus test under ARM.

5. Conclusion

The comparative analysis conducted in this study revealed significant disparities among the SC, TSO, and ARM memory models regarding their acceptance of program behaviors. It was determined that SC, as the strongest model, imposes the most stringent constraints on program execution, accepting only a limited set of behaviors. TSO, while less restrictive than SC, was found to maintain a degree of order, particularly with respect to data consistency. ARM, the weakest model, was observed to offer the greatest flexibility, allowing for a wider range of program behaviors. Despite these variations, data consistency was ensured by all three models, a fundamental requirement for concurrent programming. However, their adherence to program order was found to vary significantly. SC was observed to strictly preserve program order, while TSO was noted to relax this requirement to some extent. ARM, on the other hand, was determined to deviate the most from program order, potentially leading to unexpected behaviors if not carefully managed. The results of these experiments underscore the importance of understanding memory model nuances when designing and implementing concurrent programs. By carefully selecting the appropriate memory model and considering its implications, developers can mitigate the risks associated with non-deterministic behavior and ensure the correctness and reliability of their applications. Furthermore, the herd tools were found to be a valuable asset in this research. Their flexibility and ease of use make them an ideal choice for modeling and analyzing memory models, facilitating a deeper understanding of their characteristics and limitations. As the field of concurrent programming continues to evolve, tools like herd will likely play a crucial role in supporting the development of more efficient and reliable concurrent applications.

References

[1] R. E. Bryant and D. O'Hallaron, *Computer systems: a programmer's perspective*, Third edition, Global edition. in Always learning. Boston Columbus Hoboken Indianapolis New York San Francisco Cape Town: Pearson, 2016.

[2] F. B. Schneider and G. R. Andrews, "Concepts for Concurrent Programming," in *Current Trends in Concurrency. Overviews and Tutorials*, Berlin, Heidelberg: Springer-Verlag, 1986, pp. 669–716.

[3] M. Di Pierro and D. Skinner, "Concurrency in Modern Programming Languages [Guest editors' introduction]," *Comput. Sci. Eng.*, vol. 14, no. 6, pp. 8–10, Nov. 2012, doi: 10.1109/MCSE.2012.111.

[4] P. Brinch Hansen, *The origin of concurrent programming: from semaphores to remote procedure calls*. New York: Springer, 2011.

[5] Microsoft Research and L. Lamport, "The computer science of concurrency: the early years," in *Concurrency: the Works of Leslie Lamport*, D. Malkhi, Ed., Association for Computing Machinery, 2019. doi: 10.1145/3335772.3335775.

[6] P. S. Pacheco, *An introduction to parallel programming*. Amsterdam : Boston: Morgan Kaufmann, 2011.

[7] I. J. Cox and N. H. Gehani, "Concurrent Programming and Robotics," *The International Journal of Robotics Research*, vol. 8, no. 2, pp. 3–16, Apr. 1989, doi: 10.1177/027836498900800201.

[8] R. Elmasri and S. Navathe, *Fundamentals of database systems*, Seventh edition. Hoboken, NJ: Pearson, 2016.

[9] J. van de Mortel-Fronczak and J. Rooda, "Application Of Concurrent Programming To Specification Of Industrial Systems," *Proceedings of the 1995 IFAC Symposium on Information Control Problems in Manufacturing (INCOM '95)*, pp. 421–426, Jan. 1995.

[10] H. Sutter and J. Larus, "Software and the Concurrency Revolution: Leveraging the full power of multicore processors demands new tools and new thinking from the software industry.," *Queue*, vol. 3, no. 7, pp. 54–62, Sep. 2005, doi: 10.1145/1095408.1095421.

[11] H. Sutter and others, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's journal*, vol. 30, no. 3, pp. 202–210, 2005.

[12] M. F. Atig, "What is decidable under the TSO memory model?," *ACM SIGLOG News*, vol. 7, no. 4, pp. 4–19, Oct. 2020, doi: 10.1145/3458593.3458595.

[13] Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Trans. Comput.*, vol. C–28, no. 9, pp. 690–691, Sep. 1979, doi: 10.1109/TC.1979.1675439.

[14] S. Mador-Haim *et al.*, "An Axiomatic Memory Model for POWER Multiprocessors," in *Computer Aided Verification*, vol. 7358, P. Madhusudan and S. A. Seshia, Eds., in Lecture Notes in Computer Science, vol. 7358. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 495–512. doi: 10.1007/978-3-642-31424-7_36.

[15] D. J. Sorin, M. D. Hill, and D. A. Wood, *A primer on memory consistency and cache coherence*. in Synthesis lectures on computer architecture, no. 16. San Rafael, Calif.: Morgan & Claypool Publishers, 2011.

[16] A. Naeem, A. Jantsch, and Z. Lu, "Architecture Support and Comparison of Three Memory Consistency Models in NoC Based Systems," in *2012 15th Euromicro Conference on Digital System Design*, Cesme, Izmir, Turkey: IEEE, Sep. 2012, pp. 304–311. doi: 10.1109/DSD.2012.27.

[17] A. Naeem, X. Chen, Z. Lu, and A. Jantsch, "Realization and performance comparison of sequential and weak memory consistency models in network-on-chip based multi-core systems," in *16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011)*, Yokohama, Japan: IEEE, Jan. 2011, pp. 154–159. doi: 10.1109/ASPDAC.2011.5722176.

[18] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi, "What's Decidable about Weak Memory Models?," in *Programming Languages and Systems*, vol. 7211, H. Seidl, Ed., in Lecture Notes in Computer Science, vol. 7211. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 26–46. doi: 10.1007/978-3-642-28869-2_2.

[19] F. Z. Nardelli *et al.*, "Relaxed Memory Models Must Be Rigorous," *Exploiting Concurrency Efficiently and Correctly, CAV 2009 Workshop*, Jun. 2009. [Online]. Available: https://kar.kent.ac.uk/31904/

[20] P. A. Abdulla, M. F. Atig, B. Jonsson, and T. P. Ngo, "Optimal stateless model checking under the release-acquire semantics," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 1–29, Oct. 2018, doi: 10.1145/3276505.

[21] J. Alglave, L. Maranget, and M. Tautschnig, "Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 2, pp. 1–74, Jul. 2014, doi: 10.1145/2627752.

[22] K. D. Jade Alglave Luc Maranget, Antoine Hacquard, Boqun Feng, Kate Deplaix, "Part III simulating memory models with herd7." [Online]. Available: http://diy.inria.fr/doc/herd.html

[23] J. Alglave, P. Cousot, and L. Maranget, "Syntax and semantics of the weak consistency model specification language cat," 2016, doi: 10.48550/ARXIV.1608.07531.

[24] "diy tools, Release Seven." Oct. 2014.

[25] G. PETRI, "Operational Semantics of Relaxed Memory Models," UNIVERSITE DE NICE - SOPHIA ANTIPOLIS, 2010.

[26] N. Gavrilenko, H. Ponce-de-León, F. Furbach, K. Heljanko, and R. Meyer, "BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings," in *Computer Aided Verification*, vol. 11561, I. Dillig and S. Tasiran, Eds., in Lecture Notes in Computer Science, vol. 11561. , Cham: Springer International Publishing, 2019, pp. 355–365. doi: 10.1007/978-3-030-25540-4_19.

[27] J. Alglave, W. Deacon, R. Grisenthwaite, A. Hacquard, and L. Maranget, "Armed Cats: Formal Concurrency Modelling at Arm," *ACM Trans. Program. Lang. Syst.*, vol. 43, no. 2, pp. 1–54, Jun. 2021, doi: 10.1145/3458926.