MODEL CHECKING UNDER JAM21

by

Michał Rączkiewicz

Members of the Supervisory Committee

Chair:	prof. dr. **A. van Deursen**	Delft University of Technology
Supervisor:	dr. **S. Chakraborty**	Delft University of Technology
Core Member:	dr. **A. Costea**	Delft University of Technology

Submitted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Abstract

This thesis presents the first known implementation of a model checker for the Java memory model JAM21 within the GenMC framework - a tool for stateless model checking using custom memory models. In addition to the baseline GenMC implementation, we introduce a more efficient model checking algorithm based on vector clocks, which significantly outperforms the initial version. We provide a formal proof of equivalence between the new vector clock algorithm and the original implementation to ensure correctness. Our approach is evaluated using the standard suite of litmus tests included with GenMC. The results confirm that both algorithms correctly enforce the guarantees of JAM21. Furthermore, we successfully replicate prior findings by disallowing erroneous behaviors that went undetected under JAM19.

Acknowledgments

This work would not be possible if not for Soham's ability to effortlessly guide me through the complex labyrinth of the world of model checking; and his contribution to my development as a researcher. I would like to thank Arie for his enthusiasm and interest in the work, and Andreea for generously dedicating her time to sit on the defense committee.

I would like to thank my partner, Dominika, who empowered and motivated me to complete this work. I would like to thank my parents, who supported me thought the entire journey into my higher education. Finally, I would like to thank my friends: Tibet, Filip and Duru, who made me feel at home in the Netherlands.

Contents

1 Introduction 1

2 Background 2
- 2.1 Architecture Memory Models . . . 2
- 2.2 Language Memory Models . . . 4
- 2.3 Execution Graphs . . . 6
- 2.4 Model Checking . . . 8

3 GenMC 11
- 3.1 Stateless Model Checking . . . 11
- 3.2 Custom Memory Models . . . 14

4 Java Access Modes 21 16
- 4.1 Compilation Problem . . . 16
- 4.2 Definition . . . 19
- 4.3 JAM21 Example . . . 21

5 Implementation 23
- 5.1 Relation . . . 23
- 5.2 Vector Clock . . . 24
- 5.3 Equivalence . . . 30

6 Evaluation 36
- 6.1 Correctness . . . 36
- 6.2 Equivalence . . . 40
- 6.3 Performance . . . 43

7 Related Work 48

8 Conclusions 50
- 8.1 Future Work . . . 51

9 References 52

10 Appendices 55
- A Reproduction Package . . . 56
- B Pseudocode Struct $\mathbb{Q}$ . . . 57
- C Consistent Executions . . . 58
- D Fences Program . . . 63
- E Only-sc/Only-rlx Programs . . . 65

1. INTRODUCTION

Concurrent programming is difficult. There are numerous challenges associated with it, one of them being the inherit unintuitive nature of execution of concurrent programs. Many modern programming languages operate on a high level of abstraction removed from the hardware, however it is the hardware that ultimately dictates this seemingly random behavior. We want to maintain the high-level abstraction that benefits the programmer while also considering the low-level behavior of the processor. How can we resolve this dilemma?

This problem can be solved with *memory models*. They can be thought of as a contract between the programmer and the processor about how the written program may behave during execution in the processor. The pursuit of efficiency has resulted in more complex processor design, which in turn complicated the memory models. Manual checking of a program under such complex memory models is impractical, and thus the field of *model checking* emerged.

Model checking aims at verifying if every possible execution of a program is free of implementation errors. This is achieved with automated tools such as *CDSChecker* [1], *Nidhugg* [2], *Tracer* [3] or *GenMC* [4]. In order to carry out such verification, a memory model for a certain programming language must be formalized. Because of rather lax documentation for both compilers and processors, such standards emerged shockingly late, even for well-established languages like C or Java. C memory model was formalized in 2011 as *C/C++11* [5] and Java memory model in 2019 as *JAM19* [6]. Both models were prooven unsound with respect to IBM Power processor architecture, and thus their repaired versions, *RC11* [7] and *JAM21* [8], were introduced.

We implement model checking with JAM21 in GenMC [4], a tool for checking programs with custom memory models. To our knowledge, this is the only existing implementation of a JAM21 model checker. Our implementation leverages GenMC's modular architecture and stateless model checking algorithm. We verify our implementation over a set of litmus tests included with GenMC. We present that our model checker disallows incorrect execution in Power architecture as shown by [8].

This paper is written to be a standalone piece of literature that can be understood with a minimal knowledge in the field of model checking. We exhaustively present all prerequisite concepts needed to understand this paper in section 2. We motivate the existence of memory models and build intuition for model checking. Readers familiar with model checking are encouraged to skip this section. Section 3 presents GenMC and its stateless model checking algorithm. We also explore how it can be adopted to a custom memory model. In section 4, we introduce the JAM21 model and the compilation problem to IBM Power architecture that it solves. Section 5 dives into implementation details and presents an alternative, more efficient approach to model checking. Finally, in section 6 we evaluate our solution on some litmus tests.

2. BACKGROUND

Before presenting the JAM21 model and its implementation in GenMC, we would like to explain and give intuition of all the prerequisites needed to understand this paper. Readers familiar with model checking are encouraged to skip this section.

Concurrent programs are an indispensable part of modern software design. This stems both from software engineering requirements and the nature of hardware architecture. Processors have long been designed to have multiple cores, creating multiprocessors. Each core of a multiprocessor has its own register, and multiple caches. Moreover, oftentimes multiprocessors have multiple levels of caches, with some of them shared between cores. Such a design improves computational efficiency and reduces the risk of overheating; however, it significantly complicates the execution of concurrent programs.

On the other hand, compilers perform transformations during the compilation process, which may also affect the execution of concurrent programs. These transformations depend on various factors, with the target architecture, compiler version, and settings being the most significant. More importantly, the order in which threads are scheduled is impossible to predict in the software design phase. All of this variability prohibits predicting how the program will exactly execute. Moreover, two executions of the same program might yield different results.

To solve this problem, *memory models* were introduced. A memory model describes all possible executions that a program can have. An execution is *consistent* under some memory model, if the memory model's axioms are not violated. Such axioms are usually defined formally, however they might also simply state which reordering of instructions are allowed. In this section we build an intuition of what do memory models actually describe. We discuss the architecture (hardware) side and then the language (software) side. We firstly describe memory models with respect to which reorderings they allow and then move on to their axiomatic definitions.

2.1. ARCHITECTURE MEMORY MODELS

In this section we present how memory models are shaped with respect to hardware architecture. We give examples of some executions under different memory models and compare differences. A memory model defines rules for how a program can execute and thus which transformations on the program are allowed. In the context of hardware, program transformations are reorderings of machine code instructions.

The simplest and most intuitive memory model is *sequential consistency* (SC) defined by Lamport [9]. Essentially, this memory model does not allow for any reordering of instructions, the only differences between executions can arise from thread interleaving. SC model ensures a total ordering of all memory operations and this ordering is observed by all threads. This can be viewed as all threads submitting their instructions to a global queue, from which the instructions are retrieved and executed in sequence. We present

SC on program 2.1.

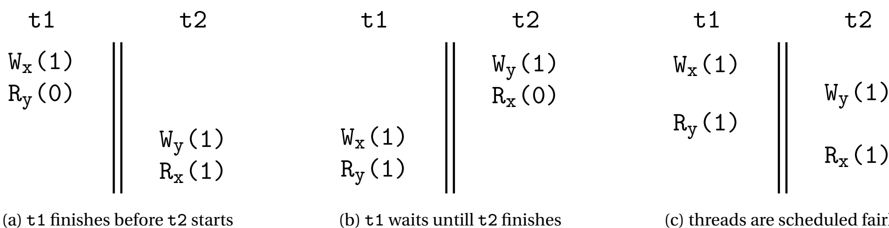
The program consists of two threads `t1` and `t2` and their parallel execution is denoted by the double vertical bars. Both write (`W`) and read (`R`) instructions are assumed to be atomic and access shared variables `x` and `y`, which are initialized to `0`. In our notation $W_x(1)$ means write to variable `x` of value `1`. Values returned by read instructions are unknown before the execution and thus marked with ?. We will supplement this notation further in the remainder of the paper.

t1	‖	t2
$W_x(1)$	‖	$W_y(1)$
$R_y(?)$	‖	$R_x(?)$

Program 2.1: Load buffering test program. Read instructions can read either 0 or 1.

Some possible executions of the program under SC are shown in Figure 2.2. Although three additional executions are possible, their observable results are identical to those presented. The differences between the three results stem from thread scheduling. In the first result (see figure 2.2a) thread `t1` was scheduled first and thus have reached the read instruction from memory address `y` before `t2` reached its write instruction to that memory address. This resulted in `t1` reading `0` and `t2` reading `1` from variable `x`. Conversely, in result 2.2b `t2` was scheduled first and thus it has read `0` from `x`. In execution 2.2c both threads firstly reached their respective write instructions and then continued to read resulting in both threads reading `1`.

Even such a simple program produces executions with different results; a behavior attributed to nondeterministic thread scheduling. SC does not allow for any reordering of instructions; therefore, the three observable execution results of program 2.1 are the only ones consistent under SC. Whereas SC memory model is intuitive to reason about, unfortunately most mainstream processor architectures exhibit behavior inconsistent with SC [10].



(a) `t1` finishes before `t2` starts (b) `t1` waits untill `t2` finishes (c) threads are scheduled fairly

Figure 2.2: Some possible executions of load buffering test program under *sequential consistency* memory model.

Each core in an x86 multiprocessor uses a write buffer acting as an exclusive cache to that core. Write accesses to the shared memory can be delayed by that buffer. This is sufficient to create non-SC behavior in the processor. For example, suppose we run program 2.1 on an x86 multiprocessor. If the processor would schedule threads as in

result 2.2c, both threads would write to their respective write buffers and then immediately read from shared memory. In such scenario, writes could be delayed beyond reads, leading both reads to return 0 (see figure 2.3).

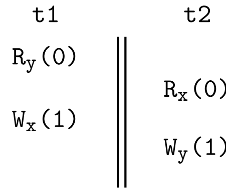


Figure 2.3: An execution of the load buffering test program (program 2.1) on an x86 multiprocessor. Both write accesses have been delayed in the write buffer. This execution is consistent in TSO but not SC.

To describe behavior of x86 processors a new memory model called *total store order* (TSO) was introduced by [11]. It mimics *sequential consistency* memory model but allows for execution of `W-R` pairs in reverse order. As this relaxation reflects the effects of the write buffer, reordering of reverse pairs `R-W` is forbidden. Intuitively, read instructions immediately access shared memory and thus can never be delayed past the preceding write instruction. Multiprocessor architectures such as IBM Power and ARM exhibit even more relaxed behavior. Writes to different memory locations in those architectures might be reordered arbitrary. Moreover, such writes can propagate to threads in different order [10] disallowing threads to trivially establish total ordering over all memory operations.

Sequential consistency can be recovered in all memory models with the use of memory fences. Effects of those instructions differ between hardware architectures [12]. In most architectures, usually two kinds of barriers are available, the light barrier and the heavy barrier. A relevant example is Power multiprocessor architecture which includes `lwsync` and `hwsync` instructions. Whereas both instructions propagate completed write memory accesses to other threads, `lwsync` does not await an acknowledgment from other threads [8].

Restoring SC in relaxed memory models is a delicate balance between efficiency and correctness. Inserting too few memory fences might result in undesirable program behavior, inserting too many will impact its performance. It might moreover not be desired to restore SC in the first place, as some concurrent computations are more efficient in a relaxed setting or outright require it. For example the Linux kernel heavily utilizes non-SC semantics [10] and so do programs designed for operation in graphic processing units (GPU). For this reason, most programming languages enable programmer to choose which set of transformation semantics to use for every memory access, a concept covered in the following section 2.2.

2.2. LANGUAGE MEMORY MODELS

So far we have discussed memory models that describe allowed program executions in hardware. Program transformations might also be applied during compilation. Describing those transformed programs with hardware-level semantics would not capture those transformations. A program, although written adhering to the hardware memory model,

can be transformed by the compiler and produce an inconsistent execution in the same memory model [13]. For this reason, language-level memory models are needed. These models consider both the transformations performed by the compiler and the translation from the language syntax to machine code primitives.

A frequent compiler optimization is *common subexpression elimination*. Two identical instructions might be eliminated [10] as their effect is redundant. For example, two write instructions of the same value to the same memory location can be replaced by a single write. In a multithreaded context such optimizations can influence program execution. Language-level memory models also take into account transformations from source to target code. In some cases, those translations might introduce data races [14].

Above problems prohibit the use of *sequential consistency* memory model on language level. A memory model that accepts such behavior is *data race free - sequential consistency* (DRF-SC) as presented by [10]. This model states that any data-race-free program will exhibit SC behavior. Otherwise its behavior will be undefined. Whereas C/C++ family of languages can already exhibit undefined behavior (for example, null pointer dereference), in Java-like languages such behavior is not acceptable [10].

Accepting DRF-SC as standard, general-purpose memory model would also significantly decrease performance. Maintaining SC between threads would require frequent insertion of memory fences, which are very costly, taking up to 100 processor cycles to execute. To avoid this, most languages implement *memory access modes* which are applied to atomic memory accesses (see table 2.1). Those accesses allow for relaxation from SC exactly to the point required by the program, and are mostly used for communication between threads. We use notation $\mathtt{W}^{\mathtt{m}}$ to signify a write with access mode `m`.

Memory Access Mode	**Guarantees**
Sequentially Consistent (`sc`)	Maintains SC between all SC accesses in the program.
Release (`rel`)	Any instruction cannot be reordered after a release write.
Acquire (`acq`)	Any instruction cannot be reordered before an acquire read.
Relaxed (`rlx`)	Maintains only atomicity guarantees, can be reordered arbitrarily.
Non-atomic (`na`)	No guarantees of any kind. Can produce undefined behavior.

Table 2.1: Memory access modes, ordered from strongest to weakest.

To give further intuition for *release-acquire* (RA) semantics, we would like to illustrate them on an example program 2.4. The program consists of a non-atomic write (in `t1`) and a non-atomic read (in `t2`), both from shared variable `x`. Similarly, there is a release-acquire pair of accesses on variable `y`. We can actually predict the only consistent execution of the program, by consulting the release-acquire semantics (see table 2.1). Namely, since a release instruction waits for all previous instructions to finish executing, the write to `x` in `t1` must execute before the write to `y`. Conversely in `t2`, acquire access guaran-

tees that no instruction will be reordered before itself, thus it will execute before the read from x. In this program, one might think of variable y as a memory barrier.

Program 2.4: Message passing test program.

Figure 2.5: The only execution of message passing program not violating release-acquire semantics.

The DRF-SC memory model does not capture *release-acquire* semantics and usage of various access modes on the same variable. In response, a new memory model was defined as a joint effort and formalized by [5]. The model is specific to C-like languages and thus is called *C/C++11* (C11). This model weds the DRF-SC and memory access modes into one memory model, allowing for usage of both semantics within the scope of a single programming language [7]. The appeal of C11 stems from the guarantee that when only SC accesses are used, the model will behave equivalently to DRF-SC, whereas the usage of relaxed atomics is still possible. C11 thus enables intuitive reasoning about multithreaded code while also allowing the implementation of high-performance multithreaded programs using relaxed atomics.

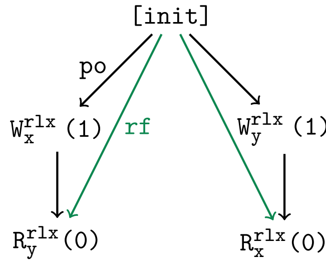
Despite C11 being widely adopted, it unfortunately is broken. Certain inconsistent executions on language level can actually execute after compilation to Power processor architecture. The problem appears when SC and relaxed accesses to the same variable are mixed. During compilation, compiler might not insert all required fences between SC accesses, thus potentially violating total order between SC memory accesses, which is guaranteed by C11 [7]. Whereas exact problems of C11 are beyond the scope of this paper, the Java memory model, JAM19 also encountered problems with memory fences while compiling to power (see section 4.1). Guarantees lost in compilation were restored with the *repaired C11* (RC11) model by [7]. The model has been prooven sound with respect to compilation to x86, Power and ARM architectures thus establishing the state of the art memory model for C-like languages.

Modern memory models are most often defined by axioms. Namely, some execution is consistent with a memory model if it does not violate any of the axioms. Those axioms are usually defined by some properties of binary relations between events in the execution. We show how to calculate such relations on execution graphs in section 2.3.

2.3. EXECUTION GRAPHS

When discussing memory models we introduced a distinction between a program and its execution. Every program could produce multiple distinct executions, which are dictated by thread scheduling, processor behavior and compiler transformations. Simply presenting the execution order of events, as we have done in previous sections, is rather cumbersome. For example, understanding the modification order of a memory address

requires navigating back and forth between writes. Moreover, whenever instructions are reordered we loose information about their syntactic ordering (for example, reordering of W-R pairs under TSO, see figure 2.3). In this section we will introduce a more intuitive and expressive approach to presenting program executions, in the form of execution graphs. For example, graph 2.6 presents the execution of load buffering program under TSO (figure 2.3).



Graph 2.6: Execution graph of the execution 2.3 with some selected relations.

Execution presented in a graph form does not compromise between showing the reordering of W-R pairs and showing their syntactic order. The syntactic order is indicated with *program-order* (po) relation. It trivially links instructions as they appear in the order dictated by the program. All po edges start at the [init] event. It signifies program entry point. We additionally treat it as initialization of all variables and therefore consider it as writing 0 to all memory locations. For clarity we do not always show it, however it is always the first node in every execution graph.

Reordering of W-R pairs is implicitly captured with the *reads-from* (rf) relation. It connects a read operation to the write operation that most recently modified the memory location being read. In this execution, rf edges connect read accesses from memory locations x and y to [init]. This means that the most recent write to both of those memory location was [init] and not W_x^{rlx}(1) and W_y^{rlx}(1) respectively. Together with po relation, this implies that both W-R pairs have been reordered in this execution.

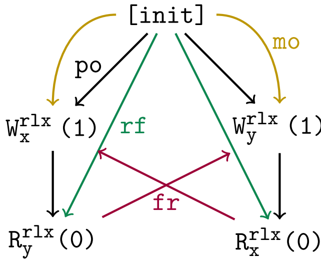
Merely by defining two basic relations on the execution graph we have shown the reordering of instructions but also their syntactic order. By defining other relations we can show even more complex relationships in the execution. A useful relation is *modification-order* (mo), which orders write operations to the same memory location in the order in which they execute. We know by po that in both threads write accesses executed after [init], thus the modification order will add two edges: one from [init] to W_x^{rlx}(1) and the other from [init] to W_y^{rlx}(1) (see graph 2.7). An intuitive way to think about mo is to understand it can only create edges between write accesses to the same memory location, thereby forming a history of modification of this memory location.

Relations can also be defined by other relations. For example, *from-reads* (fr) is defined as the inverse of rf composed with mo. Formally:

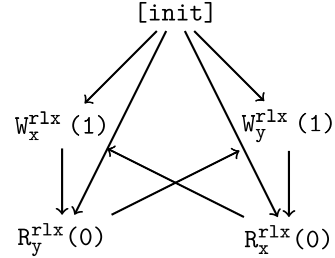
$$\mathtt{fr} \quad := \quad \mathtt{rf}^{-1} \, ; \mathtt{mo}$$

Inverse of a relation r is defined as r^{-1} and composition of two relations ra and rb is

defined as `ra ; rb`. Notation is comprehensively explained in table 2.2. The `fr` relation links a read operation with the first write operation that happened after this read was executed. Intuitively, we firstly refer to the write instruction that the read operation reads from by $\texttt{rf}^{-1}$, then we obtain the next write to that memory location by `mo`. All `fr` relations have been shown in graph 2.7. Execution graphs are used for axiomatic model checking, we show how in section 2.4.



Graph 2.7: Execution graph of the execution in figure 2.3 with relations required to verify consistency axiom of SC.



Graph 2.8: The union of po, `rf`, `mo` and `fr` in the execution in figure 2.3.

2.4. MODEL CHECKING

So far we have presented various problems that concurrent programs cause. In this section we will describe how the field of model checking aims to resolve them. Before this though, we must ask a somewhat philosophical question. What is actually the goal of software engineering? There are obviously many answers, however in the context of software verification, the goal is to produce an error free program. Especially in concurrent programs it is difficult to verify weather a program will cause undesired behavior. Certain errors might appear under very specific circumstances owing to the nature of compilers and hardware, as discussed in section 2. For this reason, to verify a concurrent program, every possible execution must be checked for erroneous behavior, and only then a program can be declared correct.

But how can we obtain all of the possible executions of a program? Before the program executes, it is unknown in which order instructions will execute. This implies the values accessed by read instructions are not yet known. In other words, the execution graph does not yet contain any `rf` edges. All possible executions can be produced by inserting `rf` edges in all permutations in the graph. Certain insertions of `rf` edges might produce executions that would be impossible to create given hardware and software transformations. In other words, naive insertion of `rf` edges can create executions that are inconsistent under some memory model. This approach causes combinatorial explosion, we only present it for illustrative purposes. Practical model checkers use fully efficient algorithms to produce all possible executions; we discuss such an algorithm in section 3.

Suppose we are verifying the load buffering program 2.1. Suppose the program will run in an environment exhibiting the sequential consistency behavior. We have pro-

Operation	Notation	Definition	Example
Event	$\mathtt{E}^{\mathtt{m}}_{\mathtt{x}}\mathtt{(0)}$	Event in an execution with memory access mode `m` to memory location `x` of value `0`. Can be a read (`R`), a write (`W`) or a memory fence (`F`).	$\mathtt{W}^{\mathtt{rel}}_{\mathtt{x}}\mathtt{(42)}$
Identity	`[E]`	If `E` is some event in an execution graph, then `[E]` is an identity relation on that event.	E
Inverse	$\mathtt{r}^{-1}$	If `r` is some relation on an execution graph from `E` to `E'`, then $\mathtt{r}^{-1}$ is a relation from `E'` to `E`.	r E E' $\mathtt{r}^{-1}$
Composition	`ra ; rb`	If `ra` is a relation from `E` to `E'` and `rb` is a relation from `E'` to `E"` then their composition is a relation from `E` to `E"`.	E ra E' rb ra;rb E"
Transitive Closure	$\mathtt{r}^{+}$	If `r` is some relation, then $\mathtt{r}^{+}$ is a composition of an arbitrary nonzero number of `r`.	E r E' r $\mathtt{r}^{+}$ $\mathtt{r}^{+}$ E" r E"'
Union	$\mathtt{ra} \cup \mathtt{rb}$	Union of relations `ra` and `rb` contains all edges from both `ra` and `rb`.	E ra E' $\mathtt{ra} \cup \mathtt{rb}$ rb $\mathtt{ra} \cup \mathtt{rb}$ E"

Table 2.2: Execution graph notation used through the paper.

duced all the possible executions by inserting the `rf` edges and checked every possible execution for various bugs. Suppose a bug was found in execution graph 2.7. Is this actually a problem? It is possible that by naively inserting `rf` edges, an inconsistent execution was created. By definition, inconsistent execution is impossible to occur and thus it does not matter if it contains a bug or not.

To check consistency of the execution, we must verify if none of the axioms defined by a memory model are violated. To check if execution graph 2.7 is consistent under sequential consistency memory model, we must consult SC axioms. In this case, there is just a single condition to be checked [15], namely:

$$\mathtt{po} \cup \mathtt{rf} \cup \mathtt{mo} \cup \mathtt{fr} \quad \text{is acyclic}$$

The above axiom states that the union of `po`, `rf`, `mo` and `fr` relations does not contain a

cycle. In other words, those four relations taken together must not create a cycle. In this case, a cycle is created:

$$\mathtt{R}_{\mathtt{y}}^{\mathtt{rlx}} \xrightarrow{\mathtt{fr}} \mathtt{W}_{\mathtt{y}}^{\mathtt{rlx}} \xrightarrow{\mathtt{po}} \mathtt{R}_{\mathtt{x}}^{\mathtt{rlx}} \xrightarrow{\mathtt{fr}} \mathtt{W}_{\mathtt{x}}^{\mathtt{rlx}} \xrightarrow{\mathtt{po}} \mathtt{R}_{\mathtt{y}}^{\mathtt{rlx}}$$

This execution is inconsistent under SC, which confirms our findings from section 2.1. In other words, this execution will never occur under sequential consistency model and thus its erroneous behavior will not be exhibited. Assuming all remaining executions of the program are bug-free, we can declare the entire program bug-free.

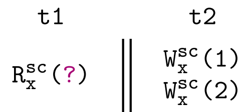
Whereas *sequential consistency* is a relatively simple memory model and thus it can rely on just one axiom, more sophisticated memory models usually have multiple. For example, RC11-consistent execution must satisfy four axioms, and JAM21 checks for two axioms. Additionally, relations referenced by those axioms are more intricate than the ones presented. Nevertheless we have equipped the reader with sufficient knowledge to understand an arbitrary axiom-based memory model and to verify if an execution is consistent with respect to that model.

3. GenMC

Model checking is a tedious task when carried out manually. We walked the reader through an example instance of model checking for a single execution in section 2.4. In this chapter we present GenMC [4] - a tool that carries out model checking automatically. Whereas there exist a wide array of model checking tools, they do not allow for model checking with a custom memory model. For this reason we choose GenMC as the implementation platform. In this chapter we start by explaining GenMC's fully efficient algorithm that explores all possible executions of a program. Our description is based on explanation given by [16], however more intuitive and tied to model checking. We then show how the tool can be extended with an arbitrary memory model, as long as it meets some well defined axioms.

3.1. Stateless Model Checking

As explained in section 2.4, to fully verify a concurrent program we must enumerate all of its possible executions. To avoid combinatorial explosion, GenMC avoids all redundant executions of a program thanks to algorithm presented by [16]. We will explain the algorithm with some examples on program 3.1. The general idea of the exploration algorithm is to build all execution graphs one event at a time. The algorithm starts with an empty graph and keeps adding events in some predefined, consistent order.



Program 3.1: Example concurrent program.

Suppose the algorithm building the graph starts with the first thread and adds all of its events according to po. The first event added to the graph is [init]. The algorithm immediately proceeds to the first thread and adds the read event. Since every read event must have exactly one rf edge, such edge is also added to the graph. The rf edge must start at a write access, in this case [init], which implicitly acts as initialization write to all memory locations.

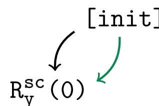


Figure 3.2: Execution graph after addition of events from thread t1. The rf edge was added between the only two eligible events.

Proceeding further with the algorithm and adding the write event from t2, we observe a problem. The rf edge is fixed between the read and [init] but it might also originate at the t2 write. Already one possible execution graph is missing.

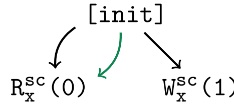


Figure 3.3: Execution graph after addition of t2 write. The rf edge is fixed at [init] as dictated by the event addition order.

This problem is solved by marking the read *revisitable*. In practice, this means the rf edge of the read can be changed to originate at a different write. Every read is marked as revisitable upon addition to the execution graph, which is indicated as a box around it. When a write access is added, all reorderings of rf edges with all revisitable reads are considered. For each reordering of rf, a new execution graph is created. For example, when t2 write is added, two execution graphs are created, reflecting the two possible rf edge configurations.

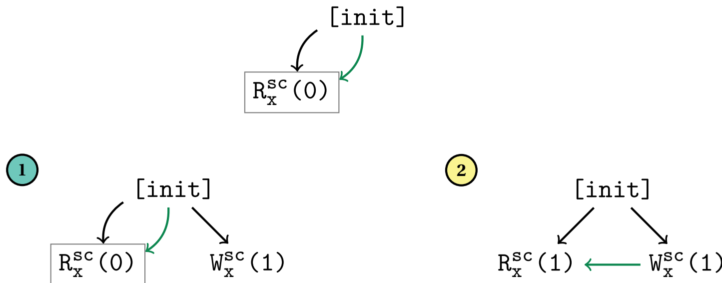


Figure 3.4: Execution graph after addition of t2 write. Two exploration steps were created, to reflect all possible rf edge configurations. The read instruction is marked revisitable in exactly one exploration step to avoid redundant explorations.

The algorithm recursively acts on the newly created execution graphs. To avoid redundancy, the read remains revisitable in exactly one newly created execution graph. Since at least one read remains revisitable, the rf edge can still be altered if another write is added. Indeed, this is what happens when the final write is added. Figure 3.5 illustrates addition of a write instruction to graph 1; the rf edge is changed again and one read remains marked as revisitable. For every rf edge, a new execution graph is created, in this case 3 and 4. Since the revisit set is empty in 2, no rf edges can be reordered and thus only one execution graph is created when the final write instruction is added (see figure 3.6). No more instructions are left to be added, the algorithm finishes; execution graphs 3, 4 and 5 are the output of the algorithm.

Seemingly, this exploration algorithm hinges on some details that might decide which execution graphs are created. Two major concerns might be the order in which events are added to the graph, and the choice of revisitable reads. Whereas we do not provide a

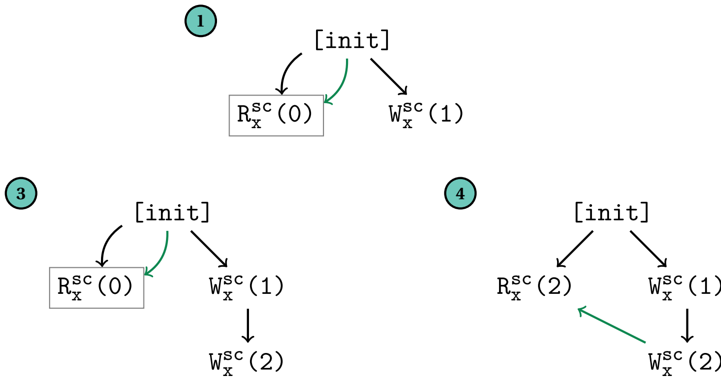


Figure 3.5: Execution graph in exploration 1 after addition of the final write. Two more explorations are created.

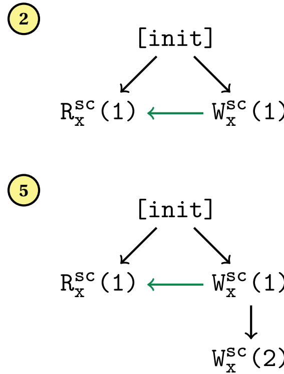


Figure 3.6: Execution graph 2 after addition of the final write. Since the read is not revisitable, the `rf` edges cannot be reordered and only one execution graph is produced.

formal proof, we try to give some intuition behind the algorithm. Firstly, we mentioned that the order of addition of events to the graph can be arbitrary, as long as it is consistent between all intermediate execution graphs. As stated by [4], the addition order should only stay consistent to guarantee optimality and not correctness. In other words, if the addition ordered is altered, all possible execution graphs will still be produced, albeit with some redundancy.

Secondly, revisit sets can be decided arbitrarily, as long as exactly one read is in the revisit set. Suppose, in the execution graphs in figure 3.4, we set read in execution 2 as revisitable, instead of the read in execution 1. Then, execution 1 would only produce execution 3, but the execution 2 would yield 4 and 5, because it would be possible to alter the `rf` edge. The output of the algorithm would remain the same, merely the intermediate steps would differ.

Although the exploration algorithm is more complex, we omit a further detailed discussion of it, as our goal is to present GenMC, which abstracts it away. Discussion of

addition of multiple read accesses in sequence or handling of read modify write instructions can be found in [16].

The algorithm presented here can be used to carry out *stateless model checking*. Stateless model checking is characterized by not maintaining any states in the memory while performing the exploration. As mentioned previously (see section 2.4), one of the challenges of model checking is obtaining all possible execution graphs efficiently. Stateless approach helps to resolve this problem. Since all execution graphs are obtained optimally, performing stateless model checking is maximally efficient when it comes to computation. Additionally, since no intermediate states are stored, algorithm is also memory efficient.

3.2. CUSTOM MEMORY MODELS

GenMC was created with support of custom memory models in mind. Modular architecture of the tool makes it easy to add new memory models. The main component of GenMC is the *driver*. It manages the process of exploration and requests model checking for selected execution graphs. Driver has an *interpreter* and a *work queue* at its disposal. The interpreter executes the program and its directly based on the LLVM interpreter. LLVM is a widely used compiler infrastructure suitable for compilation of an arbitrary programming language [17]. GenMC can thus support a wide array of programming languages; direct interpretation from C is also available. The driver is notified by the interpreter every time the interpreter executes an instruction. The driver manages the work queue, which stores alternate options for later expansion of the execution graph. The work queue is more or less equivalent to the readers set as presented in the exploration algorithm (see section 3.1) [4].

Driver also updates the *execution graph* component according to the interpreter output and instructions from the work queue. Execution graph is built once, and then `rf` edges of instructions in the work queue are altered following the optimal exploration algorithm. Driver also appends *calculators* to the execution graph, which calculate consistency of an execution upon request. Each calculator calculates a single axiom. Modular structure of GenMC allows for overwriting the driver and appending arbitrary calculators to the execution graph.

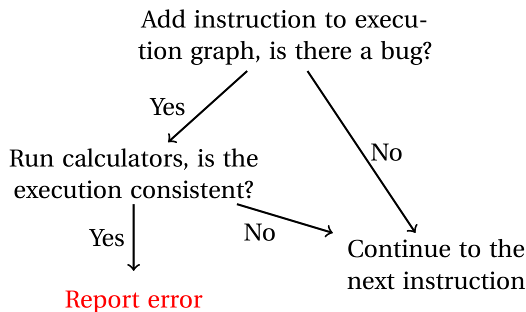


Figure 3.7: High level GenMC verification logic.

To achieve high efficiency, GenMC optimizes model checking by only running calculators on selected executions. Since the goal of GenMC is to verify correctness of a program, it firstly checks if an execution contains an implementation error, such as access to freed memory or a data race. Only then, execution is checked for consistency. If the execution is inconsistent, then the bug will never occur since the memory model disallows it (see figure 3.7). This logic can be overridden by `-check-consistency-point` flag, which can be set to `exec`, to check at every execution or `point` to check after every instruction. The latter is possible due to some prerequisites that GenMC enforces on the memory model. Checking consistency for every instruction might counterintuitively yield better performance. If an execution is marked inconsistent there is no reason to explore it further. Some model checking algorithms take this approach by incrementally calculating relations in their axioms.

This approach is not valid for all memory models. Some memory models might have inconsistent intermediate execution graphs, that can be expanded to consistent graphs with addition of events [18]. Model checking with such models cannot be done with GenMC. In fact, memory models must meet the following requirements in order to be implementable in GenMC [4]:

- **Prefix-Closedness:** Given an execution graph G, consistent under memory model M, any subgraph of G representing a complete execution of a program, is also consistent under M.
- **Extensibility:** Given a consistent execution graph G under a memory model M, adding an event consistent with M will yield a consistent execution graph.
- **No-Thin-Air:** If an execution graph is consistent, the $\texttt{po} \cup \texttt{rf}$ relation in that graph must be acyclic. This prevents circular dependencies and the *out-of-thin-air* problem as described in [10].

Above requirements stem from the design of GenMC and its graph construction algorithm. Prefix-closedness and extensibility can be though of as inverses of themselves. Namely, extensibility states the consistency of execution after a consistent event is added, conversely prefix-closedness states consistency of execution after an event is removed. The graph construction algorithm cannot create execution graphs with $\texttt{po} \cup \texttt{rf}$ cycles; this is enforced by the events being added according to `po` order. Any memory model that allows for such cycles would not be exhaustively checked by GenMC.

4. JAVA ACCESS MODES 21

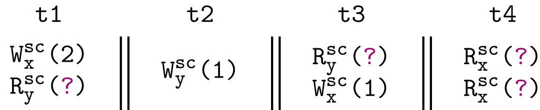
To accommodate efficient concurrent code, the OpenJDK 9 Java platform has introduced atomic accesses characterized by increasing synchronization guarantees. Semantics of those accesses follow guarantees of the memory access modes presented in table 2.1, with exception of the volatile access. To help reason about java memory access modes, the *Java Access Modes 19* (JAM19) memory model was introduced.

The model was later amended to JAM21, in response to an incorrect compilation scheme. We describe this compilation problem in section 4.1 and we present the formal JAM21 definition in section 4.2. Finally, in section 4.3, to build reader's intuition, we show JAM21 model checking on an example program.

4.1. COMPILATION PROBLEM

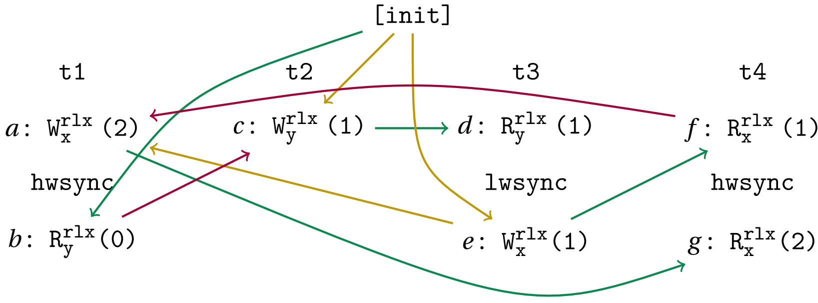
Volatile memory accesses in Java offer stronger guarantees than an equivalent memory access mode in C, which is an SC access. For example, in contrast with C, Java does not allow register promotion on memory locations with volatile accesses. Such semantics equip programmers with more intuitive memory access mode, which can not be optimized by compiler in unexpected ways [8].

A basic property of volatile accesses in Java, just as SC accesses in C, is that all such accesses in a program can be totally ordered. Unfortunately, in JAM19, this property is violated, even if programs only make use of volatile accesses. This inconsistency arises from the compilation scheme of C1 and C2 compilers in OpenJDK 9 HotSpot Java Virtual Machine. The problem occurs when compiling to IBM Power architecture, when a volatile read is followed by a volatile write. HotSpot compiler uses a leading lightweight synchronization barrier (`lwsync`) when compiling a volatile write, which can result in some instructions being delayed during execution in effect violating SC-consistency [8].



Program 4.1: Example program [8] which should exhibit exclusively SC behavior.

A four thread program 4.1 is given as an example of wrong compilation [8]. We mark volatile accesses as SC to keep consistent with notation. This program should only exhibit SC behaviour, as all accesses are SC [10]. Whereas this appears to be the case before compilation, memory access modes are a language-level abstraction, and thus compilers must enforce synchronization guarantees using memory fences. On Power, a volatile read is compiled to `hwsync ; lwz ; lwsync`, and volatile write is compiled to `lwsync ; stw ; hwsync` [8]. To keep consistent with notation, we represent `lwz` and `stw` instructions as R^{rlx} and W^{rlx} respectively. This representation reflects the relaxed nature



Graph 4.2: Non SC execution graph of program 4.1.

of Power architecture.

In certain executions, memory fences inserted by the compiler are not sufficient to guarantee SC on Power architecture. Execution graph 4.2 presents such a case. The po relation has been omitted for readability. The mo edges from [init] event are given by the definition of [init] as the initial event. The mo edge from e to a is implied by po between f and g. Those two events cannot be reordered due to hwsync barrier, which implies the memory location x must first have value of 1 and then 2. The SC axiom (acyclicity of po ∪ rf ∪ mo ∪ fr) is violated in the above execution. A cycle is created between $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow a$.

To intuitively understand why SC is violated, observe the ordering of memory accesses in t1 and t2. The fr edge between t1 and t2 establishes synchronization between threads. Since read b reads 0, it happens before the write c; and the hwsync barrier forbids reordering of a and b. Reads in t4 are ordered by hwsync, thus read f of value 1 happens before read g of value 2. This implies the *modification order* between accesses to x, which further implies that e happened before a. Finally by rf and lwsync c happens before e. Summarizing, a happens before c, c happens before e and e happens before a, an impossible arrangement.

The execution in graph 4.2 might happen under the Power memory model due to the behavior of lwsync and hwsync barriers. In particular, the lwsync does not await a confirmation from other threads before proceeding with execution. This can result in a situation where threads have different view of the memory [8]. Suppose the execution has reached the lwsync barrier in t3. The barrier informs other threads about the state of the thread's memory, which for t3 at this point is y=1. The thread continues onto instruction e and terminates. Thread t4 executes f and then the hwsync barrier, which informs other threads about current memory state (x=1 by rf). The barrier of t4 does not propagate y=1, because the message from lwsync has been delayed and has not yet reached t4. Then, thread t1 executes a, hwsync and b. Only after b is executed, the message from lwsync arrives at t1. Timeline of events creating this inconsistent execution is presented in figure 4.3.

This compilation problem can be easily fixed by changing the compilation scheme of volatile write accesses to compile to hwsync ; stw [8]. Leading lightweight barrier in previous compilation scheme was upgraded to heavyweight barrier which ensures full

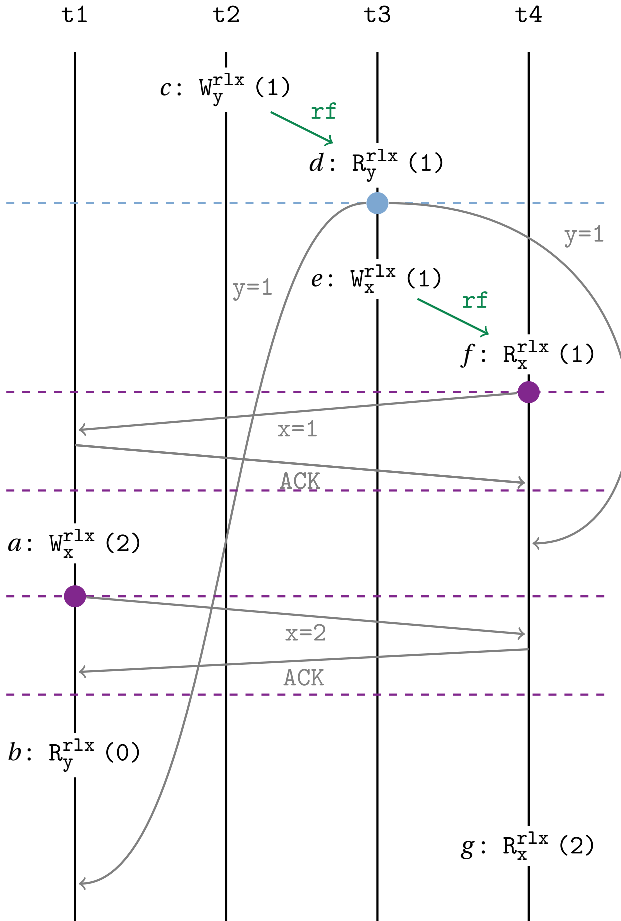


Figure 4.3: Propagation timeline of values between threads in execution graph 4.2. Selected barrier synchronization messages are shown in gray. The `lwsync` barrier in `t3` does not await acknowledgments thus allowing for non-SC behavior. Unlike lightweight barrier, `hwsync` only allows for continued execution after receiving acknowledgments from all threads.

synchronization before every volatile write. Since volatile reads are compiled to have leading heavyweight barrier, trailing barrier in volatile writes is not needed. In such arrangement, there always is at least one heavyweight barrier between any two volatile accesses. Current memory model, JAM19, does not however recognize this new compilation scheme. In fact, it mimics Power PC memory model, meaning if only language-level volatile (SC) accesses are used, non-SC behavior is still allowed. In response to this issue, [8] have developed a memory model that correctly recognizes non-SC behavior and discards such executions.

4.2. DEFINITION

In this section we present a fixed Java memory model developed by [8]. JAM21 is based on visibility (vo); a relation equivalent to *happens-before* in some other memory models. The purpose of vo is to partially order events in the graph; which can be used to derive co-jom. In some memory models, co-jom is approximately equivalent to mo, as it orders writes to the same location in the order they have executed. An execution is said to be consistent under JAM21 if $\texttt{po} \cup \texttt{rf}$ and co-jom are acyclic. We herby present the formal definition of JAM21 [8]; which will be further explained below.

$$\begin{aligned}
\texttt{ra} &:= \texttt{po} \;;\; ([E^{sc}] \cup [E^{rel}] \cup [E^{acq}]) \;;\; \texttt{po} \\
\texttt{svo} &:= \texttt{po} \;;\; [F^{rel}] \;;\; \texttt{po} \;;\; [W \cup R] \;;\; \texttt{po} \;;\; [F^{acq}] \;;\; \texttt{po} \\
\texttt{spush} &:= \texttt{po} \;;\; [F^{sc}] \;;\; \texttt{po} \\
\texttt{volint} &:= [E^{sc}] \;;\; \texttt{po} \;;\; [E^{sc}] \\
\texttt{vvo} &:= (\texttt{rf} \cup \texttt{ra} \cup \texttt{svo} \cup \texttt{spush} \cup \texttt{volint}) \cup (\texttt{pushto} \;;\; (\texttt{spush} \cup \texttt{volint})) \\
\texttt{vo} &:= \texttt{vvo}^{+} \cup \texttt{po-loc}
\end{aligned}$$

Where po-loc is a po relation between accesses to the same memory location; and pushto are all possible linearisations of the domain of $\texttt{spush} \cup \texttt{volint}$ not violating $\texttt{po} \cup \texttt{rf}$. Coherence order co-jom is defined as a relation between distinct writes to the same memory location. Formally, those writes are asserted as $\forall p, q \in W_x$ s.t. $p \neq q$, where x signifies an arbitrary memory location. An execution is JAM21-consistent if co-jom and $\texttt{po} \cup \texttt{rf}$ are acyclic.

$$\begin{aligned}
\texttt{coww} &:= [p \in W_x] \;;\; \texttt{vo} \;;\; [q \in W_x] \\
\texttt{cowr} &:= [p \in W_x] \;;\; \texttt{vo} \;;\; \texttt{rf}^{-1} \;;\; [q \in W_x] \\
\texttt{corw} &:= [p \in W_x] \;;\; \texttt{vo} \;;\; \texttt{po} \;;\; [q \in W_x] \\
\texttt{corr} &:= [p \in W_x] \;;\; \texttt{rf} \;;\; \texttt{po} \;;\; \texttt{rf}^{-1} \;;\; [q \in W_x] \\
\texttt{co-jom} &:= \texttt{coww} \cup \texttt{cowr} \cup \texttt{corw} \cup \texttt{corr}
\end{aligned}$$

The definition is formal, we will therefore discuss the memory model in order to give some intuition to the reader. Firstly, the first four relations ra, svo spush and volint establish an ordering of events within a single thread. For example, volint creates an edge between two sc events related by po. In practice, by creating an edge between two such events, volint forbids their reordering in execution, as the edge is added to vo. Two events related by vo must be executed in the order specified by the relation (as vo is equivalent to *happens-before* in some memory models). Similarly, spush forbids reordering of any events over an SC fence. Ordering of release-acquire accesses is captured with ra and svo. Ordering of acquire, release and SC memory accesses is enforced by ra; no events can be reordered over an SC, release or acquire access. Finally, svo captures the semantics of release and acquire fences. No event can be reordered over a memory access sandwiched between a release and an acquire fence.

Establishing inter-thread visibility is mostly done by including rf in vo. Intuitively, for an rf edge to exist, the write must have happened before the read. Additionally, pushto relation is calculated, and represents the order in which SC instructions execute.

4

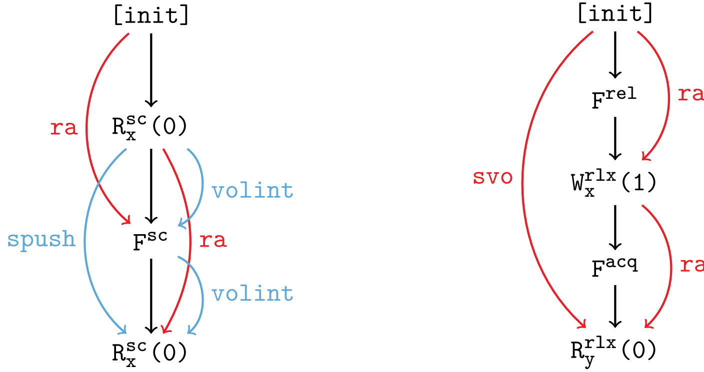


Figure 4.4: Examples of intra-thread visibility relations, rf edges not shown for readability.

Adding a pushto edge that would create a cycle in po ∪ rf would represent reordering of SC instructions within a single thread, which is prohibited by SC semantics [9]. Since threads can interleave in multiple ways, all valid linearisations must be considered for every execution graph. If at least one linearisation fulfills the consistency axiom, the execution is consistent. In other words, if a valid linearisation is found, there exists an order in which threads interleave to create given execution.

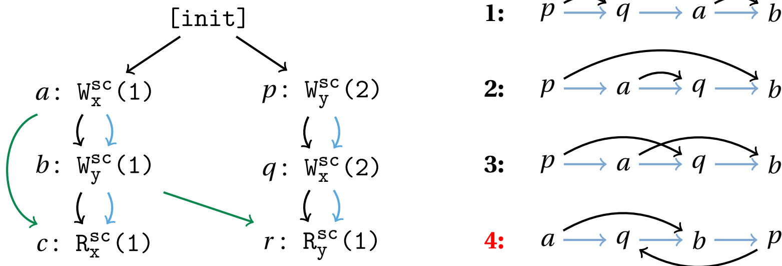


Figure 4.5: Execution graph with spush edges (left), with selected linearisations (right). Events a, b, p and q are in the domain of spush ∪ volint. Linearisation **1** produces consistent execution. Linearisations **2** and **3** are valid but produce inconsistent execution graphs. Linearisation **4** is invalid, as it violates po ∪ rf.

Figure 4.5 shows an execution graph with some possible linearisations of the domain of spush ∪ volint, which contains a, b, p and q. Calculation of the domain can be done trivially by including all events with an outgoing spush or volint edge (or both). The linearisation represented by pushto shows the order in which events have executed. Some example linearisations are presented in figure 4.5 by blue arrows; po relation is marked with black arrows. Linearisation **4** is invalid, as it creates a cycle with po ($q \rightarrow b \rightarrow p \rightarrow q$). Linearisations **2** and **3** are valid but produce an inconsistent execution. For this execution to be consistent, a and b must not be overwritten by q and p respectively, thus $q \rightarrow a$ and $p \rightarrow b$ must hold. In linearisation **2** condition $q \rightarrow a$ is violated, conversely in **3**, condition $p \rightarrow b$ does not hold. Only in linearisation **1** both conditions hold.

Upon successful calculation of pushto which represents linearisation, it is composed with spush ∪ volint to create a new relation io := pushto; spush ∪ volint (we call it *interleaving order*, hence io). Whereas pushto only orders the domain, io takes into account all SC accesses, even ones not in the domain of spush ∪ volint. All relations establishing ordering are combined in vvo, which is constructed as the union of ra, svo, spush, volint (intra-thread ordering), rf (ordering by *reads from*) and io (thread execution order). Finally, vo is the transitive closure of vvo, with addition of po-loc (*program order per location*). Addition of po-loc forbids reordering of accesses in the same thread to the same memory location. This obvious condition is a basic piece of semantics of any architecture, however it must be added explicitly, as JAM21 does not enforce it with existing intra-thread relations.

Consistency in JAM21 is stated by the acyclicity of co-jom, which is equivalent to mo in some memory models. The goal of co-jom is to totally order writes to the same memory location. If such an ordering is acyclic, the execution is consistent. In other words, there must be some order in which all writes execute in a consistent execution. A logical ordering cannot be cyclical as it poses several logical problems, for example which event should execute last?

The co-jom relation is calculated as the union of four coherence implications. These rules are equivalent to *modification order* implications by [19] (see table 5.1). The simplest rule coww (write-write coherence) states that two writes to the same location ordered by vo are also ordered by co-jom. Intuitively, if two writes are related by vo, that means there already is an established order of their execution. Similarly for two writes related by vo and po (corw). If a read and a write are related by vo, but the read is related by rf with a different write, then a co-jom edge must be added between those two writes (cowr). Analogously, for two vo related reads, if they have rf edges outgoing from different writes, those writes are related by co-jom (corr).

4.3. JAM21 EXAMPLE

In this section we apply JAM21 on program 4.1 and check its consistency. The goal is to check if JAM21 correctly recognizes inconsistent non-SC execution. We apply the memory model to machine code compiled with the repaired compilation scheme for Java volatile (SC) accesses as presented by [8]. To not introduce extra notation, we use $\mathtt{F}^{\mathtt{sc}}$ to represent heavyweight hwsync barrier and $\mathtt{F}^{\mathtt{rlx}}$ to represent lightweight lwsync barrier. Execution graph 4.2, deemed consistent by JAM19, is presented in figure 4.6.

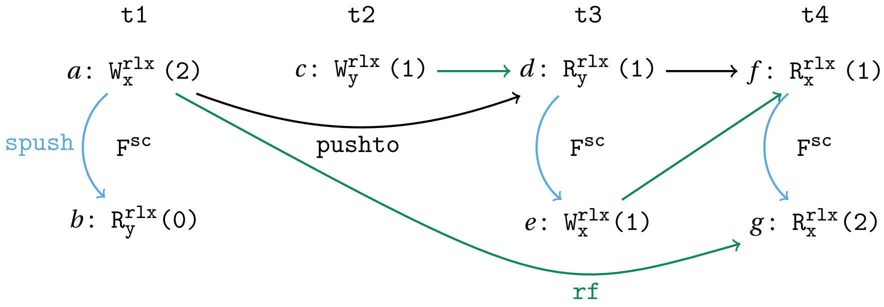
To check consistency, firstly vo must be calculated. Figure 4.6a shows all intra thread relations, in this case just spush. Note, we do not show po relation for readibility. Linearisation pushto is calculated among a, d and f since only those events are in the domain of spush ∪ volint. Events d and f are related by po ∪ rf and thus must obey $d \to f$. There are thus three valid linearisations:

$$a \to d \to f \tag{4.1}$$

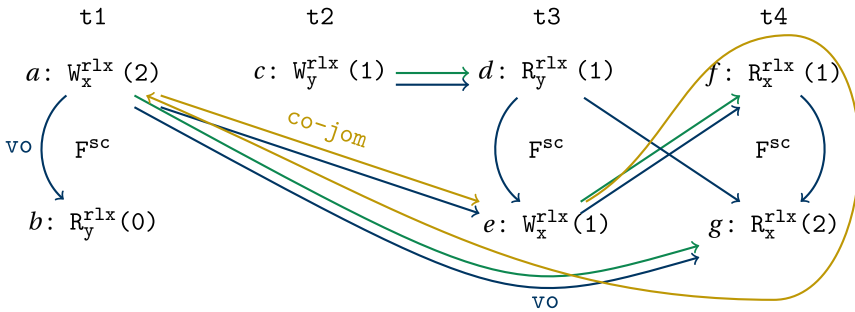
$$d \to a \to f \tag{4.2}$$

$$d \to f \to a \tag{4.3}$$

Due to space constraints, we only show calculations for linearisation 4.1, leaving the



(a) Execution graph with intra-thread relations and pushto as linearisation 4.1.



(b) Relations needed to check consistency derived from intra-thread ordering, pusho and rf.

Figure 4.6: JAM21 relations in execution graph 4.2. For clarity, intra thread relations and pushto are shown in 4.6a (top), vo and cojom shown in 4.6b (bottom); po relation not shown. A co-jom cycle is found, making this execution inconsistent.

remaining calculations as an exercise for the reader. Linearisation 4.1 adds two edges to vo: $a \to e$ and $d \to g$, both by composition with spush. Remaining vo edges are trivially added by rf and spush (see figure 4.6b). Finally, co-jom can only be calculated between a and e as they are they only two write accesses to the same memory location. Edge starting at a is added by coww and another edge starting at e is added by corr. In effect a cycle is created and the execution is inconsistent.

5. IMPLEMENTATION

To present our solution, we have implemented two model checking algorithms in GenMC. We firstly present simple, relation-based approach in section 5.1. Relation-based implementation aims at showcasing the feasibility of JAM21 model checking over performance. We develop a more efficient solution by modeling the relations with vector clocks in section 5.2.

5.1. RELATION

Our implementation of JAM21 model checking in GenMC revolves exclusively around a custom calculator and not a custom driver. We found that using RC11 driver already implemented in GenMC satisfies JAM21 model checking needs, as nothing of note is tracked when new events are added to an execution graph. Instead, our implementation focuses on a calculator, which calculates the `co-jom` relation and checks it acyclicity. We only use a custom driver to query the calculator, but we do not carry out any calculations in the driver itself.

The calculator completes several passes over the execution graph to calculate intra-thread relations. Each intra-thread relation is calculated in linear time with respect to graph size, as it suffices to query each instruction once. Relations are stored in an adjacency list, a data structure defined by GenMC. Next, `pushto` is computed as linearisation of the domain of `spush` ∪ `volint`. Fortunately, GenMC provides a function that returns all topological sorts of an adjacency list, which we use. All linearisations are checked for potential conflict with `po`∪`rf`; conveniently, GenMC tracks `po`∪`rf` vector clocks for each event, thus to verify linearisation, it suffices to check if vector clocks create ascending order in `pushto`.

Topological sort traverses all vertices and edges, which is has complexity of $v+e$, which is linear. The bigger complexity concern however is the amount of linearisations produced, since calculation of `co-jom` is carried out separately for each linearisation. In the worst case, $v!$ linearisations can be produced, though unlikely, as it occurs when graph has no edges.

Calculation of `vo` is done by merging all intra-thread relations and calculating a transitive closure on the resulting union. A custom transitive closure algorithm is used, that performs a depth first traversal of the graph. GenMC already provides a transitive closure calculation algorithm, however the edges it creates are not avaliable for iteration, which is required to calculate `co-jom`. Traversal is implemented recursively and stops if a cycle was created or an event with no outgoing `vo` edges was reached. In addition, previously calculated `po-loc` is added.

Finally, `co-jom` is calculated by iterating over all `vo` edges. Both events related by an edge are then checked for the four coherence axioms. Acyclicity of `co-jom` is checked with application of transitive closure, this time provided by GenMC, and irreflexivity, also already implemented by GenMC. If a graph is irreflexive, it means no nodes have an edge

from itself to itself. In other words, there are no identity edges. If a graph has a cycle, its transitive closure will have at least one identity edge.

In section 3 we described that checking consistency after each event is added to a graph might be more efficient for certain memory models. This is not the case with our implementation of JAM21 model checker. Each time the `co-jom` calculator is run, relations are calculated from scratch for the entire graph. In this case, calculations should be carried out once for the entire execution, after GenMC has finished building the graph. It is possible to adopt this algorithm to cache relations between events, however our goal was to present a minimum viable model checker. A more efficient implementation is described in the following section 5.2.

5.2. VECTOR CLOCK

Ordering of events can be represented with vector clocks, an approach pioneered by [9]. Rather than storing relations as a graph or adjacency list, we will assign each event a vector clock. By comparing two vector clocks we can see what is their ordering enforced by the memory model. Vector clocks can also decide if two events are concurrent to one another.

In our approach we will use the work of [15] who have designed an algorithm that can decide consistency for an arbitrary memory model given the *happens-before*, *reads-from* and *modification-order* relations. JAM21 does not define `hb` and `mo`, however both can be computed using existing JAM21 relations. Firstly, we have mentioned before the `vo` relation is somewhat equivalent to `hb`. If two events a and b are related by `vo`, it means b has seen the effects of a; therefore a must have happened before b. Whereas the reverse is not true, `vo` must be contained in `hb`. We can thus define `hb` as a relation that exclusively consists of `vo` edges. Further, in the vector clock model checking algorithm, the `hb` relation is represented by vector clocks; we indicate the complete set of such vector clocks for all events in the graph as $\mathbb{HB}$.

Additionally, `rf` is contained in `vo` thus `rf` edges are also contained in `hb`. Calculation of *modification-order* can not be directly performed with JAM21 relations, however given `hb` and `rf` we can derive it. This approach is used by [19] to derive `mo` in their model checking tool. In figure 5.1 we present the implications of `hb` and `rf` on `mo`.

The simplest instance are two write accesses related in `hb`. Intuitively, if one access happens before the other, the memory will be modified in accordance with `hb`. This property is called *write-write coherence*. If a read happens before a write, the memory was first modified by a write preceding the read. This is called the *read-write coherence*. In our case, `rf` edges are in `hb`, thus this case is equivalent to *write-write coherence*. If a read happens after a write but its `rf` edge points to a different write, `mo` can be established between the writes (*write-read coherence*). Lastly, if two reads are ordered by `hb`, their `rf`-related writes establish a `mo` ordering (*read-read coherence*) This case is equivalent to *write-read coherence*, since `rf` edges are in `hb`.

With `mo` calculated, consistency can be checked. As stated by [15], $\texttt{hb} \cup \texttt{rf} \cup \texttt{mo}$ must be acyclic for the execution to be consistent.

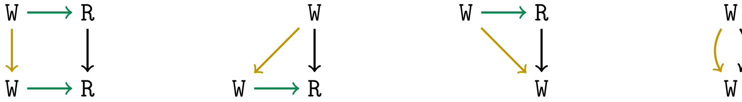


Figure 5.1: Implications of *happens-before* and *reads-from* on *modification-order* [19]. Left to right: read-read coherence, write-read coherence, read-write coherence and write-write coherence.

5.2.1. HAPPENS BEFORE

Calculation of *happens-before* vector clocks is performed by algorithm 1. The algorithm calculates vector clocks for all events in a single thread. Additionally, for any incoming *reads-from* edges to this thread, the origin thread's vector clocks will also be calculated. This is required to establish synchronization of the `rf` relation.

Through the execution, all vector clocks are stored in a globally accessible $\mathbb{HB}$ map, indexed on the event itself, pointing to the *happens-before* vector clock corresponding to that event. Access to the vector clock of event e is denoted as $\mathbb{HB}[e]$. Each vector clock has dimension equal to the total number of threads in the program. Individual indices can be accessed with $V[i]$; shown here access to index i of vector clock V. We denote merging of two vector clocks as $V \cup V'$, which is equivalent to pairwise application of max function on all elements of the vector. Set of all events in the execution is $\mathtt{E}$ with all events in some thread t indicated as $\mathtt{E}_t$.

Upon the start of the algorithm, the initial *happens-before* vector clock for the given thread is determined (see lines 2-6). Intuitively, all events in this thread must have happened after the thread has been created. In other words, the event that has created the thread ($e_{\text{thread parent}}$), must happen before all events of the current thread. Formally, all vector clocks must be strictly greater than $\mathbb{HB}[e_{\text{thread parent}}]$ vector clock. This is achieved by setting the first event's vector clock to $\mathbb{HB}[e_{\text{thread parent}}]$, and incrementing it on the thread's index (see lines 4-6). Resulting vector clock V_{base} is the minimal vector clock for any event in the thread.

After the minimal *happens-before* vector clock has been calculated, the algorithm iterates over all events in the thread $\mathtt{E}_t$ in accordance with *program-order*. To avoid redundant calculations, the algorithm checks if the vector clock for current event was already calculated. If so, the algorithm proceeds to the next event in the thread, otherwise it assigns event the minimal vector clock (see lines 9-13). This check is necessary to ensure efficiency, as the algorithm can invoke recursive calls to different threads on line 19.

If currently considered event is a read access, its vector clock must reflect the synchronization effect of the incoming `rf` edge. This is done by merging the vector clock of the write event where `rf` edge comes from (see line 21). Since the edge might originate in a separate thread, for which *happens-before* vector clocks are not yet calculated, a recursive call is made to calculate vector clocks up to and including the write event (see line 19).

Next, the intra-thread relations `ra`, `svo`, `spush` and `volint` are calculated by calling the `StepVectorClock` function defined in algorithm 2. In this presentation of the algorithm, we encapsulate access to previously seen events in struct $\mathbb{Q}$. It is a globally accessible structure, that holds pointers to last events with various characteristics. For example, pointer to the last `sc` event is kept in order to calculate `volint` edges. It suf-

Algorithm 1: Calculation of happens-before vector clocks in a thread

```
1  Function ComputeHappensBefore(t, e_halt):
2      e_thread start ← E_t[0]
3      e_thread parent ← e_thread start.getParentCreate
4      V_parent ← HB[e_thread parent]
5      V_base ← V_parent[t] + 1
6      HB[e_thread start] ← V_base
7
8      for e ∈ E_t do
9          if HB[e] = ∅ then
10             HB[e] ← V_base
11         else
12             continue
13         end
14
15         if e ∈ R then
16             e_rf ← e.getRf
17             t_rf ← e_rf.thread
18             if HB[e_rf] = ∅ then
19                 ComputeHappensBefore(t_rf, e_rf)
20             end
21             HB[e] ← HB[t_rf] ∪ HB[e]
22         end
23
24         StepVectorClock(e)    /* See algorithm 2                */
25
26         if e ∈ W ∪ R then
27             if A[e.address] ≠ ∅ then
28                 HB[e] ← HB[e] ∪ A[e.address]
29                 HB[e][t] ← HB[e][t] + 1
30             end
31             A[e.address] ← HB[e]
32         end
33
34         if e = e_halt then
35             return
36         end
37     end
38 end
```

Algorithm 2: Calculation of happens-before vector clocks for a single event

```
1  Function StepVectorClock(e) is
       /* Calculation of ra relation                              */
2      V ← HB[e] ∪ HB[Q.synch]
3      V[e.thread] += 1
4      HB[e] ← V
5
       /* Calculation of svo relation                             */
6      V ← HB[e] ∪ HB[Q.acqFence]
7      V[e.thread] += 1
8      HB[e] ← V
9
       /* Calculation of spush relation                           */
10     V ← HB[e] ∪ HB[Q.scFence]
11     V[e.thread] += 1
12     HB[e] ← V
13     D ← scFence.previousEvent
14
       /* Calculation of volint relation                          */
15     V ← HB[e] ∪ HB[Q.sc]
16     V[e.thread] += 1
17     HB[e] ← V
18     D ← sc
19
20     Q.AddEvent(e)   /* See structure 4 in appendix B           */
21 end
```

fices to reference the last event in each relation to establish all intra-thread edges, thus $\mathbb{Q}$ only stores the last event for each relation. A pseudocode for the struct is presented in appendix B.

Calculation of po-loc is performed by ensuring all memory accesses to the same location as ordered by po have increasing vector clocks. This is done by storing the *happens-before* vector clock of the last memory access in map $\mathbb{A}$. The vector clock of the current memory access is then advanced to be strictly grater than the previous access's vector clock (see lines 26-32).

To summarize, we reflect the ordering of intra-thread relations with StepVectorClock function which works in conjunction with struct $\mathbb{Q}$. The ordering induced by rf edges is reflected by merging the vector clock of the writer into the vector clock of the reader. Finally po-loc is also accounted for by allways incrementing the vector clock if an access to the same location is encountered.

Only the pushto relation is missing. Because we must consider all of its possible linearisations, we account for it after hb has been calculated. A copy of $\mathbb{HB}$ without pushto edges is preserved, and the linearisations are added right before mo is calculated. During hb vector clock calculation, we only gather the linearisation domain (lines 13 and 18).

This approach avoids redundant `hb` vector clock calculations.

To calculate lineraisations of `pushto` we once again use the function provided by GenMC. We create a temporary relation containing the events in the domain of `spush` ∪ `volint` and we directly apply the function on this relation. Whereas it is possible to calculate linearisation with just vector clocks, we have decided to use the relation based approach as it guarantees correctness over our custom implementation.

Finally, to reflect the synchronization effect of `pushto`, `hb` vector clocks must be amended. Namely, `hb` clocks must be increasing in the order dictated by the `pushto` relation. This is achieved by traversing `pushto` edges and merging vector clocks of events with incoming edges with the vector clocks of events with outgoing edges. This ensures two events with an edge in `pushto` will always have `hb` vector clocks that do not violate the linearisation. In other words, the ordering of `pushto` is reflected in `hb` vector clocks.

5.2.2. MODIFICATION ORDER

5

Algorithm 3: Calculation of modification-order from happens-before

```
1  Function ComputeModificationOrder():
2      HB_sorted ← HB.sort
3
4      for (e, V) ∈ HB_sorted do
5          for (e_prevoius, V_previous) ∈ HB_sorted do
6
7              if V_previous < V and e_previous.address = e.address then
8
9                  if e ∈ W then
10                     MO ← (e_previous, e)
11                 else if e ∈ R then
12                     MO ← (e_previous, e.getRf)
13                 end
14
15             end
16         end
17     end
18 end
```

Modification order is induced by `hb` and `rf` implications (see figure 5.1). Only *write-write* and *write-read* coherence cases must be considered in our algorithm. To calculate `mo` steaming from *write-write coherence*, it suffices to traverse all `hb` edges and add `mo` edges between write accesses to the same memory location. For *write-read coherence*, the process is similar, however the final write access must be reached by a $\texttt{rf}^{-1}$ edge.

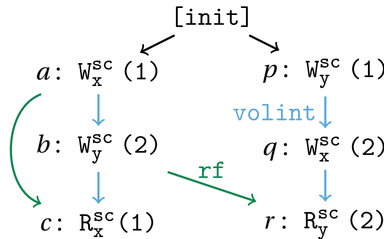
Algorithm 3 implements the above logic. The `hb` vector clocks are sorted in ascending order (line 2), which allows for their traversal in increasing order. This is equivalent to traversing the *happens-before* edges in the relation based approach. For each event, its successor in `hb` is found; which is done by finding the first event with a strictly greater

vector clock (lines 4-7). If those two events are accesses to the same memory location, an `mo` edge can be established between them. If a write access was encountered, the `mo` edge is added by *write-write coherence* (line 10), whereas for a read access, the *write-read coherence* rule is used (line 12).

5.2.3. LINEARISATIONS

Whereas we use the built in GenMC linearisation function, for completeness we present our custom implementation. Our goal is to show that the vector clock model checking algorithm can also be implemented independently of GenMC. Additionally, we show that it is possible to completely eliminate the need for the GenMC's relation data structure possibly resulting in efficiency gains.

The high level description of our linearisation algorithm is as follows. Throughout the vector clock calculation algorithm, a list of all linearisations so far calculated is maintained. Upon encountering of an event in the domain of $\texttt{spush} \cup \texttt{volint}$, the event is submitted to be added to all linearisations. This happens when a rule for advancing vector clock for either `spush` or `volint` is triggered, which implies the event is in the linearisation domain. When the linearisation algorithm receives a new event, it inserts it at every non-$\texttt{po} \cup \texttt{rf}$ violating position. If there are multiple possibilities to insert the new event, a copy of linearisation is created.



Graph 5.2: Example execution graph that produces multiple linearisations. Selected JAM21 relations shown; a, b, p and q are subject to linearisation.

We present the algorithm on an example execution graph 5.2. The intermediate and final linearisations produced by the algorithm are shown in figure 5.3. The vector clock calculation algorithm starts at the first thread and continues according to po. The first event to be added to linearisation is thus a. Since at the beginning of the execution there are no linearisations, the addition of a constitutes the initialization step. The algorithm moves to b, which can only be added behind a because both events are in po. At this point, there is thus just one valid linearisation: $a \rightarrow b$.

Event c is not in the domain of $\texttt{spush} \cup \texttt{volint}$ thus it is omitted. The next event to be considered is p, which is concurrent to both a and b. For this reason, p can be inserted at all positions between a and b, thus creating three linearisations. Further, event q is added, which is in po after p, therefore in all linearisations q must also be after p. Inserting q in all valid positions results in six linearisations total, of which only one creates a consistent JAM21 execution graph.

5

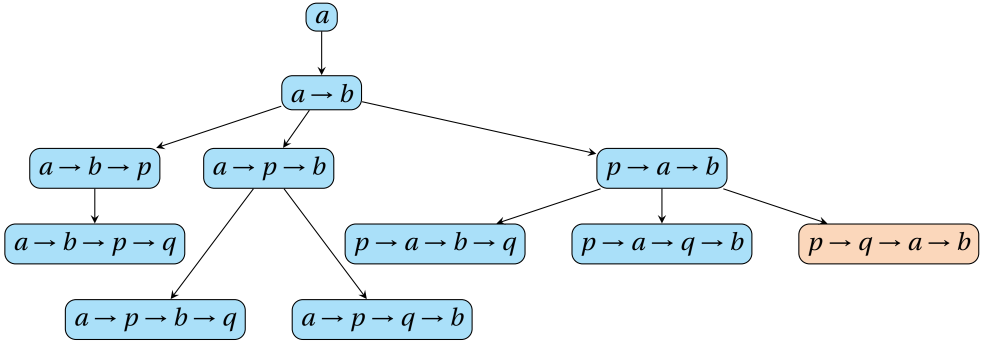


Figure 5.3: Step-by-step calculation of linearisations in execution graph 5.2. Linearisation marked in orange produces a consistent execution graph under JAM21.

5

5.2.4. CONSISTENCY CHECKING

Finally, since `mo` is equivalent to `co-jom` defined by JAM21, consistency can be checked by checking the acyclicity of `mo`. In our implementation we use relation cycle checking algorithm provided by GenMC (first calculate transitive closure, then irreflexivity). We must therefore create a temporary relation which includes all `mo` edges. It is possible to implement a cycle checking algorithm by only using vector clocks, however we chose to use GenMC for this purpose as the performance is satisfactory.

For completeness, we provide a high-level description of a vector clock based `mo` cycle checking algorithm. Namely, the algorithm keeps track of `mo` vector clock for every event in globally accessible data structure $\mathbb{MO}$. These vectors are indexed on memory locations, meaning the dimension of every vector clock is equal to the number of distinct memory locations used through the program. Upon addition of a `mo` edge, the value of the vector clock of event with incoming `mo` edge is advanced by one. This results in `mo` vector clocks which can be used to partially order all memory accesses in the execution. If an `mo` edge is added, such that it is incoming into an event with smaller `mo` vector clock, this new edge creates a cycle.

5.3. EQUIVALENCE

The vector clock model checking algorithm can not be trivially mapped to JAM21 memory model. In order to convince the reader that the vector clock model checking algorithm is equivalent to the relation based algorithm, we provide an equivalence proof. We prove that `mo` and `co-jom` are equivalent which trivially implies that both algorithms will have identical output for any execution. We firstly complete the soundness proof and then the completeness proof.

Theorem 1. *Let a and b be two events related by* `mo`*. Then, a and b are related by JAM21* `co-jom` *relation.*

Proof. Since `mo` is implied by either *write-write coherence* or *write-read coherence* (see

table 5.1, algorithm 3), then:

$$\frac{a \xrightarrow{\texttt{mo}} b}{(\exists V_a\, V_r \;\; \text{s.t.} \;\; V_a < V_r \;\wedge\; b \xrightarrow{\texttt{rf}} r) \;\vee\; (\exists V_a\, V_b \;\; \text{s.t.} \;\; V_a < V_b)}$$

namely, there either exist two events a and b which vector clocks can be ordered according to the direction of mo edge (line 10 of algorithm 3); or there exists a read event which vector clock is greater than event a but it has a rf edge originating at event b (line 12 of algorithm 3).

If two vector clocks can be ordered, they are either in the same thread and thus in po or they have been ordered by rf (see line 21 of algorithm 1). Alternatively, their ordering is induced by linearisation of pushto.

$$\frac{V_a < V_b}{a \xrightarrow{\texttt{po}} b \quad \vee \quad a \xrightarrow{\texttt{rf}} b \quad \vee \quad a \xrightarrow{\texttt{pushto};(\texttt{spush} \cup \texttt{volint})} b}$$

Cases with $a \xrightarrow{\texttt{rf}} b$ and $a \xrightarrow{\texttt{pushto};(\texttt{spush} \cup \texttt{volint})} b$ are trivially satisfied, as both relations are included in vo; this will be prooven later. For $a \xrightarrow{\texttt{po}} b$, the ordering of vector clocks was derived from either five of the intra-thread relations.

According to algorithm 2, an event's vector clock is advanced only if a previous event exists that is classified as sc, rel, or acq (see 5.1); or if there is a chain of events with strictly increasing vector clocks where the events belong to $\texttt{F}^{\texttt{rel}}$, $\texttt{W} \cup \texttt{R}$, and $\texttt{F}^{\texttt{acq}}$ respectively (see 5.2); or if an sc fence exists between the events (see 5.3); or if both events are sc (see 5.4). Additionally, by algorithm 1 (lines 26-32), if two events in po access the same memory location, their vector clocks must be orderable in accordance with po (see 5.5). Above implications are put formally:

$$\frac{V_a < V_b \quad \wedge \quad a \xrightarrow{\texttt{po}} b}{\;}$$

$$(\exists V_e \quad \text{s.t.} \quad V_a < V_e < V_b \quad \wedge \quad e \in \texttt{E}^{\texttt{sc}} \cup \texttt{E}^{\texttt{rel}} \cup \texttt{E}^{\texttt{acq}}) \tag{5.1}$$

$$\vee \quad (\exists V_k, V_l, V_m \quad \text{s.t.} \quad V_a < V_k < V_l < V_m < V_b \tag{5.2}$$
$$\wedge \quad k \in \texttt{F}^{\texttt{rel}} \quad \wedge \quad l \in \texttt{W} \cup \texttt{R} \quad \wedge \quad m \in \texttt{F}^{\texttt{acq}})$$

$$\vee \quad (\exists V_e \quad \text{s.t.} \quad V_a < V_e < V_b \quad \wedge \quad e \in \texttt{F}^{\texttt{sc}}) \tag{5.3}$$

$$\vee \quad (a \in \texttt{E}^{\texttt{sc}} \quad \wedge \quad b \in \texttt{E}^{\texttt{sc}}) \tag{5.4}$$

$$\vee \quad a.\texttt{address} = b.\texttt{address} \tag{5.5}$$

The rules for vector clock incrementation map directly onto the intra-thread relations defined by JAM21. For clarity, we omit the assumption $a \xrightarrow{\texttt{po}} b$ in all implications below; it is to be understood as holding implicitly in each case.

$$\frac{V_a < V_e < V_b \quad \wedge \quad e \in \texttt{E}^{\texttt{sc}} \cup \texttt{E}^{\texttt{rel}} \cup \texttt{E}^{\texttt{acq}}}{a \xrightarrow{\texttt{ra}} b}$$

$$\frac{V_a < V_k < V_l < V_m < V_b \quad \wedge \quad k \in \mathtt{F}^{\mathtt{rel}} \quad \wedge \quad l \in \mathtt{W} \cup \mathtt{R} \quad \wedge \quad m \in \mathtt{F}^{\mathtt{acq}}}{a \xrightarrow{\mathtt{svo}} b}$$

$$\frac{V_a < V_e < V_b \quad \wedge \quad e \in \mathtt{F}^{\mathtt{sc}}}{a \xrightarrow{\mathtt{spush}} b}$$

$$\frac{V_a < V_b \quad \wedge \quad a \in \mathtt{E}^{\mathtt{sc}} \quad \wedge \quad b \in \mathtt{E}^{\mathtt{sc}}}{a \xrightarrow{\mathtt{volint}} b}$$

$$\frac{V_a < V_b \quad \wedge \quad a.\mathtt{address} = b.\mathtt{address}}{a \xrightarrow{\mathtt{po\text{-}loc}} b}$$

Grouping the implications together, we obtain:

$$\frac{V_a < V_b \quad \wedge \quad a \xrightarrow{\mathtt{po}} b}{a \xrightarrow{\mathtt{ra}} b \quad \vee \quad a \xrightarrow{\mathtt{svo}} b \quad \vee \quad a \xrightarrow{\mathtt{spush}} b \quad \vee \quad a \xrightarrow{\mathtt{volint}} b \quad \vee \quad a \xrightarrow{\mathtt{po\text{-}loc}} b}$$

and by adding the inter-thread vector clock incrementation rules we have:

$$\frac{V_a < V_b}{\begin{array}{c} a \xrightarrow{\mathtt{ra}} b \quad \vee \quad a \xrightarrow{\mathtt{svo}} b \quad \vee \quad a \xrightarrow{\mathtt{spush}} b \quad \vee a \xrightarrow{\mathtt{volint}} b \quad \vee \quad a \xrightarrow{\mathtt{po\text{-}loc}} b \\ \vee \quad a \xrightarrow{\mathtt{rf}} b \quad \vee \quad a \xrightarrow{\mathtt{pushto}\,;\,(\mathtt{spush} \cup \mathtt{volint})} b \end{array}}$$

If two vector clocks V_a and V_b are ordered, that means events a and b are in one of the seven relations constituting `vo` (either one of the five intra-thread relations or one of the two inter-thread relations). This means the ordering of `hb` vector clocks is equivalent to `vo`. The transitive closure of `vvo` is implicitly included by the properties of vector clocks. The implication can thus be simplified to:

$$\frac{V_a < V_b}{a \xrightarrow{\mathtt{vo}} b}$$

Using the above implication, we can simplify the initial `mo` rule:

$$\frac{a \xrightarrow{\mathtt{mo}} b}{\left(\exists r \text{ s.t. } a \xrightarrow{\mathtt{vo}} r \xleftarrow{\mathtt{rf}} b\right) \quad \vee \quad a \xrightarrow{\mathtt{vo}} b}$$

If two events are in `mo` they are both write accesses to the same memory location. Therefore, in the above implication, there either is a `vo` edge between two write accesses to the same memory location, or a $\mathtt{vo}\,;\,\mathtt{rf}^{-1}$ edge between such accesses. This is equivalent to JAM21 `coww` and `cowr` relations respectively. Thus in both cases a `co-jom` edge is established:

$$\frac{a \xrightarrow{\mathtt{mo}} b}{a \xrightarrow{\mathtt{co\text{-}jom}} b}$$

□

With the soundness proof complete, we present the completeness proof.

Theorem 2. *Let a and b be two events related by JAM21* `co-jom` *relation. Then, a and b are related by* `mo` *relation.*

Proof. The JAM21 `co-jom` relation is a union of four relations `coww`, `cowr`, `corw` and `corr`. In the implication below we represent these relations by their definitions:

$$\frac{a \xrightarrow{\texttt{co-jom}} b}{a \xrightarrow{\texttt{vo}} b \quad \vee \quad a \xrightarrow{\texttt{vo ; po}} b \quad \vee \quad a \xrightarrow{\texttt{vo ; rf}^{-1}} b \quad \vee \quad a \xrightarrow{\texttt{rf ; po ; rf}^{-1}} b}$$

Since `co-jom` is a relation between two write accesses to the same memory location, we can simplify the implication further. Namely, `vo ; po` is equivalent to `vo` in this case, as `po` between accesses to the same memory locations is `po-loc` which is already included in `vo`. Similarly, `rf ; po` is also in `vo`. We have explained this in depth in section 5.2. The implication now becomes:

$$\frac{a \xrightarrow{\texttt{co-jom}} b}{a \xrightarrow{\texttt{vo}} b \qquad \vee \qquad \exists r \text{ s.t. } a \xrightarrow{\texttt{vo}} r \xleftarrow{\texttt{rf}} b} \tag{5.6}$$

If two events are related by `vo`, then they are either in `po-loc` or the transitive closure of `vvo`:

$$\frac{a \xrightarrow{\texttt{vo}} b}{a \xrightarrow{\texttt{vvo}^+} b \quad \vee \quad a \xrightarrow{\texttt{po-loc}} b}$$

The transitive closure of any relation can be split into two cases. Either the events in the transitive closure are directly related by that relation or there exists an intermediate event through which the transitive closure was established:

$$\frac{a \xrightarrow{\texttt{vvo}^+} b}{a \xrightarrow{\texttt{vvo}} b \quad \vee \quad \exists e \text{ s.t. } a \xrightarrow{\texttt{vvo}} e \xrightarrow{\texttt{vvo}^+} b} \tag{5.7}$$

If two events are related by `vvo` relation, then they are related by either four of the intra-thread relations, `rf` or the linearisation:

$$\frac{a \xrightarrow{\texttt{vvo}} b}{\begin{gathered} a \xrightarrow{\texttt{ra}} b \quad \vee \quad a \xrightarrow{\texttt{svo}} b \quad \vee \quad a \xrightarrow{\texttt{spush}} b \quad \vee \, a \xrightarrow{\texttt{volint}} b \\ \vee \quad a \xrightarrow{\texttt{rf}} b \quad \vee \quad a \xrightarrow{\texttt{pushto ; (spush} \cup \texttt{volint)}} b \end{gathered}}$$

By definition of JAM21, every intra-thread relation is created if there exists an intermediate event (or series of events) that meets certain characteristics:

$$\frac{a \xrightarrow{\texttt{ra}} b}{\exists e \text{ s.t. } e \in \mathtt{E}^{\mathtt{sc}} \cup \mathtt{E}^{\mathtt{rel}} \cup \mathtt{E}^{\mathtt{acq}} \quad \wedge \quad a \xrightarrow{\texttt{po}} e \xrightarrow{\texttt{po}} b}$$

$$\frac{a \xrightarrow{\texttt{svo}} b}{\exists k\ l\ m \text{ s.t.} \quad k \in \texttt{F}^{\texttt{rel}} \land l \in \texttt{W} \cup \texttt{R} \land m \in \texttt{F}^{\texttt{acq}} \quad \land \quad a \xrightarrow{\texttt{po}} k \xrightarrow{\texttt{po}} l \xrightarrow{\texttt{po}} m \xrightarrow{\texttt{po}} b}$$

$$\frac{a \xrightarrow{\texttt{spush}} b}{\exists e \text{ s.t. } e \in \texttt{F}^{\texttt{sc}} \quad \land \quad a \xrightarrow{\texttt{po}} e \xrightarrow{\texttt{po}} b}$$

$$\frac{a \xrightarrow{\texttt{volint}} b}{a \in \texttt{E}^{\texttt{sc}} \quad \land \quad b \in \texttt{E}^{\texttt{sc}} \quad \land \quad a \xrightarrow{\texttt{po}} b}$$

By algorithm 2, vector clocks of two events in `ra`, `svo`, `spush` or `volint` must be increasing in accordance with those relations. Vector clock of an event is increased every time a `sc`, `rel` or `acq` event was encountered (case of `ra`); a chain of `rel` fence, a write or read access and `acq` fence was seen (case of `svo`); an `sc` fence was seen (case of `spush`); or if the current event is `sc` and a previous `sc` event exists (case of `volint`). These semantics are a one to one mapping to the definition of JAM21 intra-thread relations.

Additionally, vector clocks reflect the ordering of `rf` edges (see line 21 of algorithm 1). Therefore, if two events are related by `rf`, their vector clocks are ordered according to the `rf` edge. The vector clocks of events in linearisation `pushto` are ordered as well. Summarizing, all four intra-thread relations, `rf` and `pushto ; spush ∪ volint` enforce increasing order of vector clocks of events they relate. In other words, for any two events related by those relations, their vector clocks must be increasing; or put formally:

$$\frac{a \xrightarrow{\texttt{vvo}} b}{V_a < V_b} \tag{5.8}$$

Whereas the same property holds for the transitive closure of `vvo`, because of the transitive property of $<$, an induction proof can be completed with two cases established in equation 5.7. The base case trivially holds by equation 5.8. The inductive case holds by recursive application of the proof. Since `vvo` is acyclic, recursion must at some point terminate. The following thus holds:

$$\frac{a \xrightarrow{\texttt{vvo}^+} b}{V_a < V_b}$$

The final relation that advances the vector clock is `po-loc`. If two events are related by `po-loc`, their vector clocks can be sorted in increasing order (algorithm 1 lines 26-32). This implication can be integrated as follows:

$$\frac{a \xrightarrow{\texttt{vvo}^+} b \quad \lor \quad a \xrightarrow{\texttt{po-loc}} b}{V_a < V_b}$$

By definition of `vo` and the above implication, the following holds:

$$\frac{a \xrightarrow{\texttt{vo}} b}{V_a < V_b}$$

By substituting the `vo` relation by vector clocks in equation 5.6, we obtain:

$$\frac{a \xrightarrow{\texttt{co-jom}} b}{V_a < V_b \quad \vee \quad (\exists V_r \text{ s.t. } V_a < V_r \ \wedge \ r \xleftarrow{\texttt{rf}} b)}$$

Which is precisely the definition of `mo`, thus:

$$\frac{a \xrightarrow{\texttt{co-jom}} b}{a \xrightarrow{\texttt{mo}} b}$$

□

With both directions of the proof complete, the equivalence holds:

$$\forall a, b \quad a \xrightarrow{\texttt{co-jom}} b \Leftrightarrow a \xrightarrow{\texttt{mo}} b$$

5

6. Evaluation

We have implemented JAM21 in both relation-based and vector-clock-based variants. In this section we check correctness and performance of both solutions. To evaluate our work, we pose the following research questions:

RQ1 How can consistency be checked under the JAM21 model using GenMC?

RQ2 How can consistency be checked under the JAM21 model using vector clocks?

RQ3 What is the performance implication of this new approach?

Whereas our implementation in section 5 rather exhaustively answers **RQ1** on how to implement a JAM21 model checker in GenMC, in this section we will evaluate our solution for correctness. The most tangible difference between memory models is the number of consistent executions of the same program different models allow. In our evaluation we take this metric as a benchmark and compare JAM21 to RC11 - a memory model already implemented in GenMC. We further investigate selected programs that differ in the number of consistent executions and manually check their results. We do this in section 6.1.

We have already provided pseudocode of a vector clock model checking algorithm in section 5.2, which answers **RQ2**. We supplemented this algorithm with proof of equivalence with the relation based implementation. In this section, we confirm that both implementations are indeed equivalent. This is achieved by comparing the amount of consistent executions reported by the two algorithms. Results are given in section 6.2. For each program model checked, we gather the two most important performance data points, namely the execution time and the memory used. We use these statistics to answer **RQ3** in section 6.3.

To evaluate our implementation we will use test programs included with GenMC. We have chosen three sets of programs to experiment on: *litmus*, *data-structures*, and *synthetic*. Those three test groups pose an increasing computational challenge to model checking algorithms; starting with litmus programs with just a few memory accesses, finishing with synthetic benchmarks with several interrelated accesses. This approach allows to test correctness and explain differences on case by case basis with litmus and data-structure tests, and benchmark our solution on programs from the synthetic set.

6.1. Correctness

Our strategy to answer **RQ1** is to find programs that have different number of consistent executions under JAM21 and RC11. Next, we manually verify the model checking algorithm for executions that are consistent under only one memory model but not the other. Such approach allows us to check our implementation against multiple litmus programs but focuses our attention on outliers (sections 6.1.1 – 6.1.4). Finally, as an additional

Test Group	Test Name	RC11	JAM21
litmus	2+2W+2sc+scf	3	4
	2+2W+4sc	3	4
	2+2W+scfs	3	4
	SB+rfis	4	3
	peterZ	7	8
	psc-ar	1	0
	WWR+2WR	0	2
	W+RWC	7	8
	Z6+acq	7	8
data-structures	dq *(variant 0)*	1432	1321
	dq *(variant 1)*	1432	1370
	dq-opt	1432	1321

Table 6.1: Number of consistent executions of selected test programs under RC11 and JAM21.

confirmation, we replicate the behavior of correct and wrong compilation schemes presented by [8] (section 6.1.5).

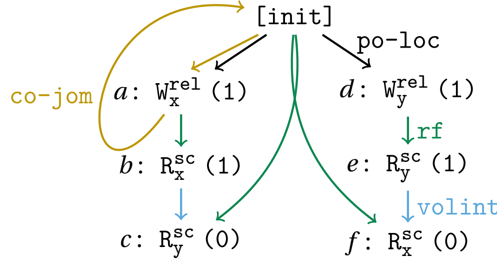
To check our implementation, we firstly enumerate all consistent executions under both memory models. GenMC already reports this number as *number of executions seen*. An execution is checked for consistency by GenMC only if an implementation error is found. An inconsistent but bug-free execution therefore counts as a *seen execution*. GenMC can be forced to check consistency at the end of every execution, by setting the `-check-consistency-point=exec` flag. Additionally, to prohibit GenMC from using its approximation mechanism we set the `-check-consistency-type=full` flag. Those two settings ensure that GenMC model checks every produced execution graph, which results in the tool reporting the count of all possible executions of a program. Finally, we obtain all consistent execution graphs with `-print-exec-graphs`.

To test correctness of JAM21, we have model checked approximately 180 programs from litmus and data-structures sets. GenMC reported the same amount of executions for most tested programs. Programs with different amount of consistent executions, and thus a different behavior under the two models, are presented in table 6.1. The complete list of programs tested is available in appendix C. We choose programs *SB+rfis*, *psc-ar*, *W+RWC* and *2+2W+4sc*, to investigate differences between the two models in detail. These four programs serve as representative examples of properties of JAM21.

6.1.1. PROGRAM SB+RFIS

Graph 6.1 shows an execution of program *SB+rfis*, which execution is consistent in RC11 but not in JAM21. In JAM21 a `co-jom` cycle is created between events a and `[init]`. A `co-jom` edge between `[init]` and a is created by `coww`, with `po-loc` constituting the `vo` edge between `[init]` and a. Similarly, there exists a `vo` edge from a to c by `rf` and `volint`. By `cowr` relation, there exists a `co-jom` edge from a via c to `[init]`.

The reason why JAM21 marks this execution inconsistent, as opposed to RC11, is the interpretation of `[init]` as a write to all memory locations. Whereas in RC11, read ac-

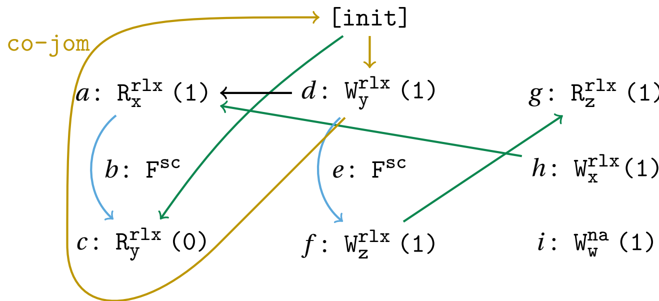


Graph 6.1: Execution graph of program *SB+rfis* consistent in RC11 but inconsistent in JAM21. The graph includes selected JAM21 relations.

cesses with `rf` edges originating at the `[init]` event implicitly return some initial value, in JAM21 that value is explicitly the default value for a variable (we only consider integer variables, thus it is always 0). This behavior is seemingly the same between the two models, however it is technically possible that in RC11 initialization reads return stale values.

Intuitively, execution graph 6.1 is inconsistent in JAM21, because of the order of `sc` accesses. In both threads, `sc` read accesses b and e return `1`. This implies that both writes a and d happened before those reads. Further, reads b and e happened before c and f, which is indicated by `volint`. Therefore the writes of a and d must have overwritten both memory addresses x and y and it is thus impossible for c and f to return the initial value of 0.

6.1.2. PROGRAM PSC-AR



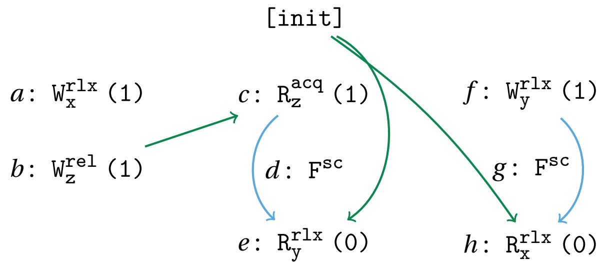
Graph 6.2: Execution graph of program *psc-ar* consistent in RC11 but inconsistent in JAM21. The graph includes `rf`, `spush` and the only valid linearisation `pushto` in JAM21. A consistency-violating `co-jom` cycle is also shown.

Execution graph 6.2 is RC11-consistent but JAM21-inconsistent execution of program *psc-ar*. A `co-jom` edge is created between `[init]` and d by `coww` (by `po-loc`). The only valid linearisation `pushto` is $d \to a$. The reverse linearisation $a \to d$ creates a `po` ∪ `rf` cycle ($a \to d \to f \to g \to h \to a$) and thus is invalid. A `co-jom` edge is thus cre-

ated by `cowr`, from d via c to `[init]`. This completes the cycle and marks the execution inconsistent.

This program highlights the unique feature of JAM21, namely the explicit ordering of `sc` accesses captured by `pushto`. JAM21 attempts to find all valid linearisations and adds each of them to a distinct copy of the execution graph. If at least one execution graph is consistent, the execution is also consistent. The synchronization effect of `pushto` reflects the effect of `hwsync` barrier inserted at machine level, which enforces full synchronization between all threads.

6.1.3. PROGRAM W+RWC



Graph 6.3: Execution graph of program *W+RWC* consistent in JAM21 but inconsistent in RC11. The graph includes `rf` and `spush` JAM21 relations.

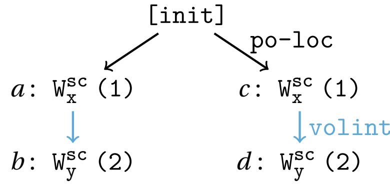
Graph 6.3 presents an execution of program *W+RWC* consistent in JAM21 but inconsistent in RC11. This execution shows another characteristics of JAM21, namely the fact that intermediate events in relations, are not a part of those relations. For example, an `ra` edge is established between a and the thread end event (not shown in the graph). Events a and b cannot be ordered however, as no other relations are established between them.

Suppose a and b would be in `vo`, and `pushto` is established between c and f. Then a `co-jom` edge exists between a and `[init]`, (by `cowr` via h), and a cycle creating `co-jom` edge between `[init]` and a (by `coww`). This execution would be inconsistent. Because a and b are not in `vo`, this of course is not the case. This poses an interesting situation, where a `rel` access does not provide synchronization guarantees, further suggesting *release-acquire* semantics differ in JAM21.

6.1.4. PROGRAM 2+2W+4SC

Program *2+2W+4sc* produces more consistent executions in JAM21 rather than in RC11. One such execution is presented as graph 6.4. The execution is inconsistent under RC11, because of RC11 *modification-order* relation. Namely, RC11 establishes a `mo` relation for all memory accesses. RC11 rejects the execution with cycle inducing `mo`; in this case the execution with $b \rightarrow c$ and $d \rightarrow a$ both in `mo`.

As stated by [8], JAM21 revolves around visibility. The only way to observe a result of a program is by a read operation. Moreover, only read operations can establish visibility between threads. With no `rf` edges, there is no `vo` established between the threads, so no `co-jom` cycle can exist. This is because with no `rf` edges, all components of `vo` are



Graph 6.4: Execution graph of program *2+2W+4sc* consistent in JAM21 but inconsistent in RC11. The graph includes selected JAM21 relations.

constrained in a single thread and must be consistent with `po`.

6.1.5. PROGRAM FENCES

To complete our verification, we return to program 4.1 which was used as an example of wrong compilation scheme by [8]. We simulate the compilation scheme by implementing the program in C. We follow the compilation scheme presented in section 4. We use `sc` fences to simulate `hwsync` barriers and `rel-acq` fences to simulate `lwsync`. We only use `rlx` accesses to reflect relaxed nature of IBM Power architecture. Full implementation of the program is included in appendix D.

We have verified both correct and wrong compilation schemes. JAM21 marks execution with the `sc` fence (correct compilation scheme) as inconsistent. Conversely, the execution with weaker fence is marked as consistent. This confirms that our implementation is correct. We have replicated the behavior of program 4.1 compiled incorrectly and confirmed the correctness of the repaired scheme.

6.2. EQUIVALENCE

For an exhaustive description of the vector clock based model checking algorithm we refer the reader to section 5.2 with the accompanying equivalence proof found in section 5.3. Whereas we have prooven and implemented a vector clock based model checking algorithm for JAM21, we have not evaluated it experimentally. Our goal is to show that the two implementations are equivalent, meaning they mark the same executions consistent.

We check equivalence of the two solutions by comparing the number of consistent executions both algorithms report. The number must necessarily be the same; intuitively, the two solutions must have the same output in order to be equivalent. Additionally, for outliers specified in table 6.1, we conduct manual comparison of executions marked as consistent. Although unlikely, it is possible that the number of consistent executions returned by both algorithms is the same, however the two algorithms might have marked the same execution differently.

We have run both algorithms on programs from *litmus* and *data-structures* groups. We have set `-check-consistency-type=full` and `-check-consistency-point=exec` flags to capture all possible executions. The set of consistent executions produced by both programs was obtained with `-print-exec-graphs` flag. For all test programs the number of consistent executions is the same, suggesting the vector clock and relation

based algorithms are equivalent. Moreover, after a manual inspection of executions produced by outlier programs (see table 6.1), it was found that for each test program the set of consistent executions produced by both implementations is the same. Above findings, in conjunction with the equivalence proof, suggest that our vector clock algorithm is correct and thus reinforce our answer to **RQ2**.

In order to give the reader a practical example on how the vector clock model checking algorithm works, we present the vector clock values produced by our algorithm on the examples discuses while answering **RQ1**. The graphs presented in this section additionally contain thread create events (`[tn create]`), which are abstracted away in relation based model checking, however pose an important point for advancing and ordering vector clocks. We further discuss how `mo` edges emerge from the `hb` vector clocks and ultimately present their equivalence with `co-jom`.

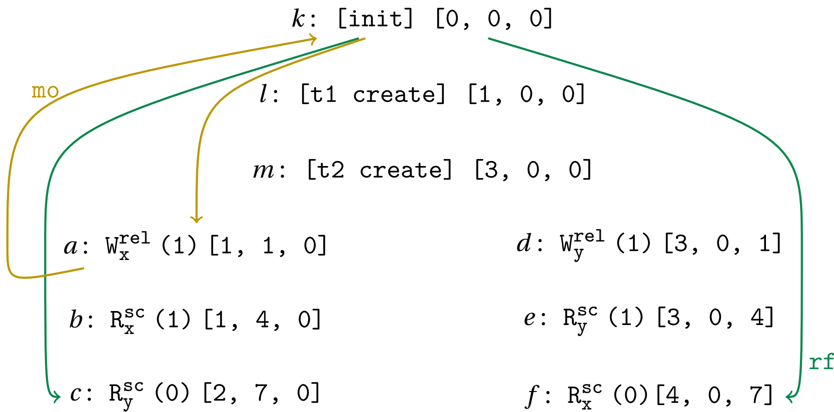


Figure 6.5: Vector clocks capturing the *happens-before* relation in the JAM21-inconsistent execution of the program *SB+rfis*. Additionally, the cycle inducing `mo` edges are marked, as well as selected `rf` edges.

In program *SB+rfis* vector clocks in all threads are increasing according to `po` (see figure 6.5). Firstly, the `[init]` and the thread create events cannot be reordered, as `[init]` by definition is the first event in the execution and thread create events have *release-acquire* semantics. In both `t1`, and `t2`, all accesses have `sc` access mode, which increases the value under thread's index by 3. This is to account for possible intermediate relations. Finally, for reads c and f, the `rf` edge reaches to the initial thread, therefore the index corresponding to `t0` increases by 1.

The `mo` edge from k to a is established because the vector clocks of these events are not concurrent; namely, `[0, 0, 0]` < `[1, 1, 0]`, therefore k has happened before a. Similarly, the read c has a strictly greater vector clock than a (`[1, 1, 0]` < `[2, 7, 0]`), which means the `mo` edge can be established by `hb;rf`$^{-1}$. In this case we consider a read with a different memory access than the write, because the read reaches to `[init]`, which is a write to all memory locations. In other cases, we can only consider reads to the same memory location as the write.

Figure 6.6 presents `hb` vector clocks for program the only execution of program *pscar*. Vector clocks take into account the `pushto` edge from d to a. The `sc` fences in threads

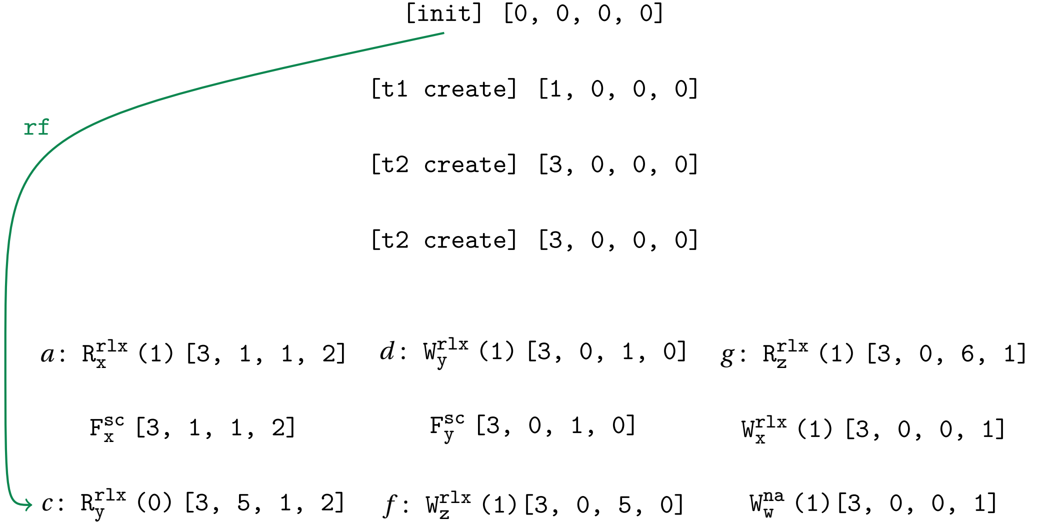


Figure 6.6: Vector clocks capturing the *happens-before* relation in the JAM21-inconsistent execution of the program *psc-ar*. Linearisation is established from d to a. Relevant `rf` edge shown.

6

`t1` an `t2` advance vector clocks of relaxed accessed. In `t3`, read g advances its vector clock on the third index to account for `rf` edge from f. A `mo` edge is established from `[init]` to d (`[0, 0, 0, 0] < [3, 0, 1, 0]`), and another edge from d to `[init]` via c (`[3, 0, 1, 0] < [3, 5, 1, 2]` and then by $\texttt{rf}^{-1}$ from c to `[init]`).

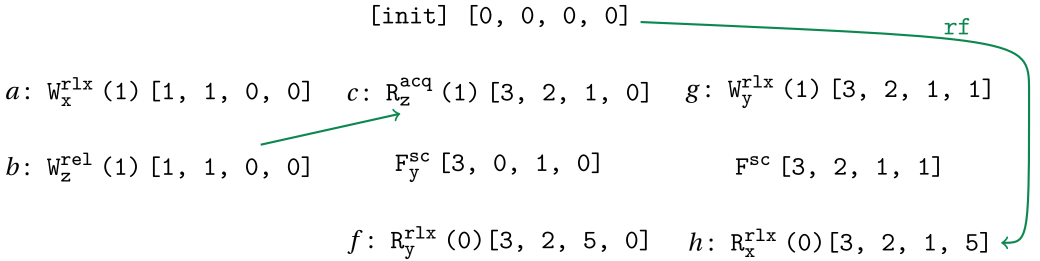


Figure 6.7: Vector clocks capturing the *happens-before* relation in one of the executions of the program *W+RWC*. Linearisation is established from c to g. Selected `rf` edges are shown. Thread create events are omitted for readability.

In figure 6.7 we show the `hb` vector clocks for the execution of program *W+RWC* that is consistent under JAM21 but inconsistent under RC11. For readability, we omit thread create events in `t0`. In `t1` there just two relaxed accesses to different memory locations; meaning they can be reordered. This is reflected by their vector clocks being concurrent. In `t2` and `t3` the accesses are separated by an `sc` fence, which is also reflected by the vector clocks, namely both f and h are strictly grater than c and g respectively.

An `mo` edge is established between `[init]` and a (`[0, 0, 0, 0] < [1, 1, 0, 0]`).

Looking at just the vector clocks, there exists a possibility to establish a cycle inducing `mo` edge from a to `[init]` via h; namely: `[1, 1, 0, 0]` < `[3, 2, 1, 5]`. This is obviously incorrect, as a and b can be reordered, therefore the synchronization effect of the `rf` edge between b and c might not capture the effects of write a. For this reason, our `mo` calculation algorithm contains an additional check for such situations; an `mo` edge can only be established via `rf` edges outgoing from reads with a strictly greater `hb` vector clock.

k: `[init] [0, 0, 0]`

a: $\mathtt{W}^{\mathtt{sc}}_{\mathtt{x}}$ `(1) [1, 1, 0]` c: $\mathtt{W}^{\mathtt{sc}}_{\mathtt{x}}$ `(1) [3, 1, 1]`

b: $\mathtt{W}^{\mathtt{sc}}_{\mathtt{y}}$ `(2) [3, 4, 0]` d: $\mathtt{W}^{\mathtt{sc}}_{\mathtt{y}}$ `(2) [3, 1, 4]`

Figure 6.8: Vector clocks capturing the *happens-before* relation in the JAM21-consistent execution of the program *2+2W+4sc*. Linearisation applied to this execution is $a \rightarrow c$. Thread create events are omitted for readability.

For completeness, we present the vector clocks produced while model checking program *2+2W+4sc* in figure 6.8. We apply an example linearisation $a \rightarrow c$. The execution is consistent in JAM21, because the execution graph consists no `rf` edges. The `hb` vector clocks must then be consistent with `po`; creating a `mo` cycle is thus impossible as `po` cannot have cycles.

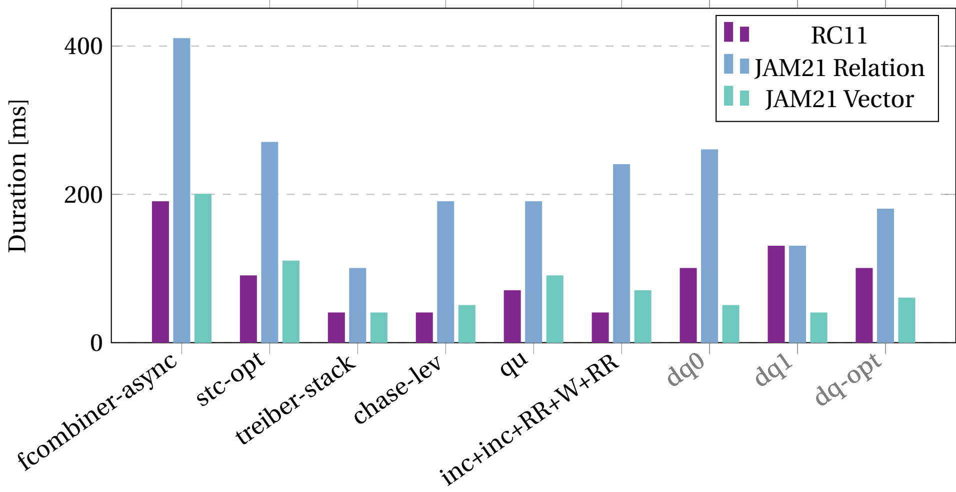
6.3. PERFORMANCE

To analyze performance of both algorithms, we have identified duration and memory usage as key metrics. We have run RC11, and both JAM21 implementations on test programs from *litmus*, *data-structures* and *synthetic* benchmarks. As previously, we have used `-check-consistency-type=full` and `-check-consistency-point=exec` flags.

In the preliminary measurement, we found that memory usage for all three algorithms is negligible. To measure memory usage, we used *htop* tool - a Linux process viewer that reports memory and processor utilization. Even for large benchmarks, for example *fib-bench0* or *peterson-sc*, approximately 200MB of random access memory (RAM) was allocated. The actual usage was near 160MB, which is approximately 1% of the total system capacity of 16GB. Moreover, the memory usage did not significantly differ between benchmarks. As memory usage is not a significant metric, we ignore it in further performance evaluation.

Contrary to RAM usage, all test programs saturated a single processor core during their model checking. The utilization of such core never drops below 99% during program execution. This reveals the single core computational capacity as the performance constraint of our implementations. Distributing the computational load to multiple processor cores would require parallelization of our algorithm, which is beyond the scope of this evaluation. We therefore focus on measuring just the execution time each benchmark takes, as the computational load is proportional to the execution time.

We only report execution duration for programs which execution time difference of



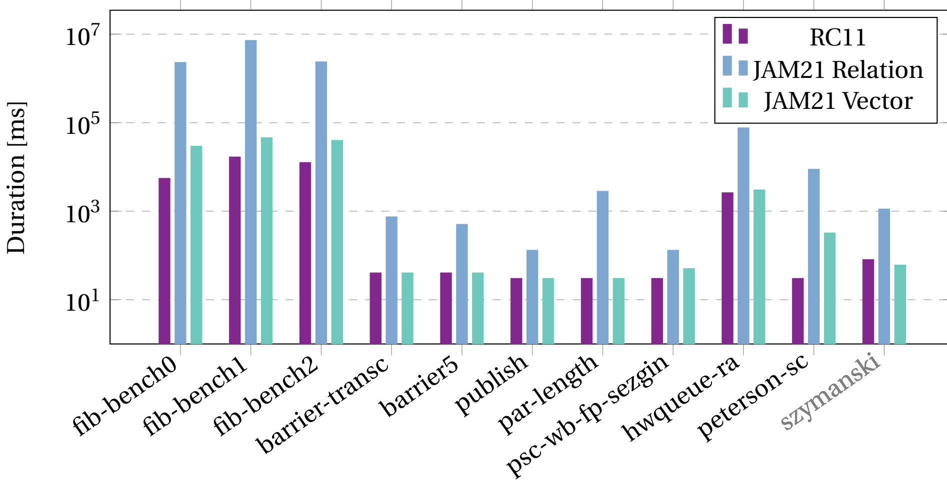
Plot 6.9: Model checking times of selected programs from *litmus* and *data-structures* benchmarks by RC11 and JAM21 model checkers. Programs marked in gray are inconsistent in JAM21 and thus terminated early.

any model checker is more than 10ms. This approach focuses on interesting outliers, rather than on majority of trivial test programs that take similar amount of time to model check. Results for programs in *litmus* and *data-strucutres* benchmarks are presented in bar graph 6.9. We center our analysis on the performance of the JAM21 Vector Clock algorithm, rather than the JAM21 Relation algorithm, since the latter is intended as a proof of concept for JAM21 model checking rather than a high-performance implementation.

For all test programs, either JAM21 model checker performs worse than RC11; exception being the *dq0*, *dq1* and *dq-opt* programs. This discrepancy occurs, because those programs all have a constraint violating consistent execution in JAM21, which forces the model checker to terminate without checking all possible execution graphs. The constraint is violated, because JAM21 does not mark constraint violating executions inconsistent. In our analysis we ignore these test programs as they would skew the results.

In all benchmarks except for *inc+inc+RR+W+RR*, the JAM21 vector implementation is never more than 30% slower than RC11. For the benchmarks where JAM21 takes up to 30% longer than RC11, at most one linearisation was identified per execution graph. The only exception is the *chase-lev* program, where four linearisations were found in the majority of executions; however, it still completed within the 30% performance margin. In the case of *inc+inc+RR+W+RR*, although each execution yields only a single linearisation, the total number of executions is significantly higher, which accounts for the larger performance gap.

Another round of benchmarking was performed on test programs from *synthetic* group. As previously, we only show benchmarks that differ by more than 10ms between any two algorithms. Model checking duration of selected programs is presented in plot 6.10 on a logarithmic scale. We have identified the szymanski test program as a constraint violating and thus we exclude it from the analysis. Similarly to previous results,



Plot 6.10: Model checking times of selected programs from *synthetic* benchmarks by RC11 and JAM21 model checkers. Mind the **logarithmic** scale. Szymanski test program marked in gray is inconsistent in JAM21 and thus terminated early.

JAM21 vector clock algorithm takes marginally longer to complete than RC11 in most benchmarks. Clear outliers are all *fib-bench* programs and *peterson-sc*.

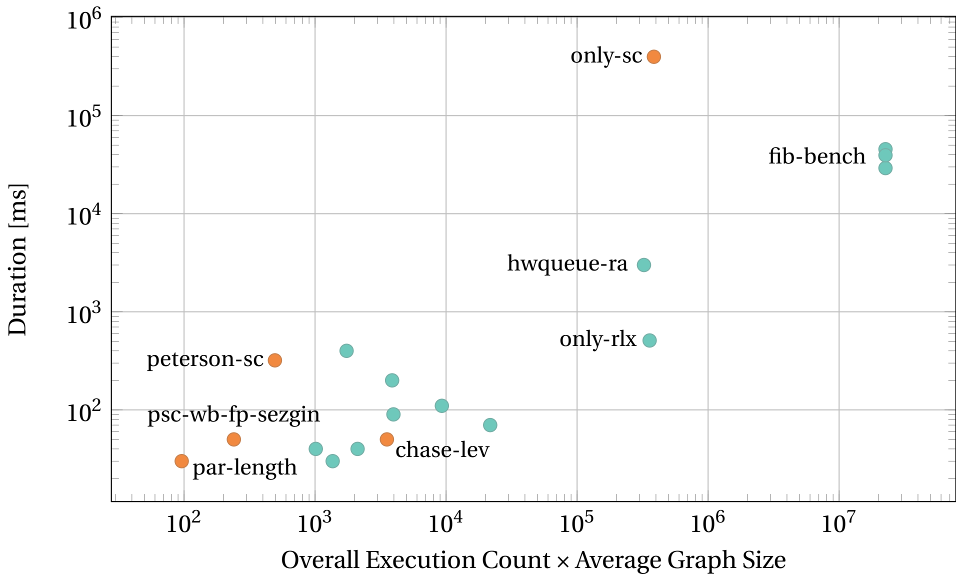
Especially *peterson-sc* program presents a big difference between RC11 and JAM21, which results from the large amount of linearisations. For this program, JAM21 produces, on average, 203 linearisations per execution. In other words, in the worst case, computational complexity of JAM21 is more than 200 times larger than RC11.

On the other hand, all three *fib-bench* programs only produce one linearisation per execution, however the JAM21 Vector algorithm still falls significantly behind RC11 in performance. The *fib-bench* program was model checked by RC11 in 547ms, while JAM21 Vector took 2920ms to complete - representing nearly half an order of magnitude difference in performance. Arguably, this is the worst result among the three: *fib-bench1* and *fib-bench2* were model checked by RC11 in 1664ms and 1248ms, respectively, compared to 4547ms and 3947ms by JAM21.

The number of linearisations per execution clearly has a significant impact on model checking duration. To better understand this effect, we isolate the influence of linearisation by normalizing execution time. This is necessary because directly comparing complex benchmarks like *fib-bench* with simpler ones like *treiber-stack* would obscure the role of linearisation in performance differences.

To normalize execution time, we consider the number of events observed during model checking. Intuitively, executions with larger execution graphs require more time to complete than those with smaller graphs. Additionally, the number of distinct execution graphs generated by GenMC for a given program also impacts model checking time, as each graph must be analyzed independently.

Therefore, we normalize execution time by two key factors: the number of execution graphs and their average size. Since model checking time scales proportionally with



Plot 6.11: Impact of linearisation on model checking duration. Scatter plot of execution time compared to the normalization factor. Benchmarks marked in orange produce more than one linearisation per execution.

both, we use their product as a normalization factor. Specifically, we multiply the total number of executions by the average graph size, which varies depending on factors such as control flow complexity. We refer to this product as the *normalization factor* from this point onward.

We present the above results in plot 6.11. Programs that have more than one linearisation in any execution graph are highlighted in orange. The three benchmarks with the smallest normalization factors (far left, *psc-wb-fp-sezgin*, *par-length*, and *peterson-sc*) take disproportionately longer than similarly sized model checking tasks with only a single linearisation per execution graph. The only exception is *chase-lev*, which appears to follow the performance trend of single-linearisation programs. This difference stems from the number of linearisations: *chase-lev* averages fewer than four linearisations per execution graph, whereas *psc-wb-fp-sezgin*, *par-length*, and *peterson-sc* all average above 200, with *psc-wb-fp-sezgin* averaging around 630.

In addition to the existing benchmarks, we introduce two sibling test programs: *only-sc* and *only-rlx* (implementation details can be found in appendix E). The goal of *only-sc* is to generate as many linearisations as possible within a single execution graph; it features three `sc` accesses in every four threads. Conversely, *only-rlx* is designed to produce only one linearisation while mimicking the structure of *only-sc*. This allows us to isolate other variables influencing model checking duration.

As shown in plot 6.11, both programs share the same normalization factor, which is expected since they are essentially duplicates differing only in memory access modes. However, their execution times differ by approximately three orders of magnitude. This difference aligns closely with the number of linearisations: the use of 12 `sc` accesses

causes *only-sc* to average around 1015 linearisations per execution graph. In other words, *only-sc* has roughly a 10^3 times greater computational load than *only-rlx*, which is clearly reflected in the plot.

7. Related Work

Our work primarily builds on GenMC and JAM21. We have described them extensively in sections 3 and 4 respectively. In this section, we describe additional relevant concepts that we considered or encountered during the course of our research.

In our work, we considered using Kater [20], a tool designed to work with GenMC. Kater takes a formally defined memory model, expressed in the *Kleene Algebra with Tests* (KAT) language, and generates a corresponding consistency checker for GenMC. It employs a novel approach based on *deterministic finite automation* (DFA) to determine the consistency of executions. Specifically, one DFA is constructed to represent the acyclicity axiom of the memory model, while another is derived from the execution graph. The product of these two DFAs is then computed and executed in parallel with the memory model DFA. If no starting state in the product DFA leads to a terminating state in the memory model DFA, then no cycle exists, and the execution is deemed consistent.

Whereas we have extensively described JAM21, we did not yet describe its predecessor - JAM19 [6]. JAM19 represents a major revision of the original Java Memory Model, introducing support for various memory access modes. The initial definitions in the Java memory model were difficult to interpret, and consistency checking was difficult to carry out due to lack of appropriate tools. To address these issues, [6] introduced a formalization of the memory model, including an implementation in the Herd tool for testing with litmus tests, and a mechanized proof in Coq that established key theorems. This work laid the groundwork for the current state of the art Java memory model represented by JAM21.

Although we selected GenMC as the platform for our model checking work, another promising candidate is the Dartagnan tool [21]. Dartagnan is a model checker for weak memory models that accepts memory models defined in the KAT language. This makes it a potential platform for implementing JAM21 model checking as well. However, we chose GenMC due to its more expressive driver model, which enabled the implementation of our vector clock based model checking algorithm. In contrast, Dartagnan relies on *satisfiability modulo theories* (SMT) solvers for verification, which offers a different trade-off in terms of flexibility and performance.

Another model checker for weak memory consistency is Nidhugg [2], an open source tool that supports the Power PC architecture; making it an interesting candidate for a JAM21 implementation. However, Nidhugg's architecture is not modular, unlike GenMC's, which means that integrating a new memory model would require significant intervention into its code. Additionally, Nidhugg currently supports only SC, TSO, and PSO memory models, and does not include support for more complex models such as C11. As a result, implementing a sophisticated memory model like JAM21 within Nidhugg would likely pose substantial challenges.

So far, we have identified only one Java model checker: Java Pathfinder [22]. Originally developed by NASA and released as open source in 2005, Java Pathfinder analyzes

compiled Java bytecode to detect issues such as deadlocks and data races. However, despite being the only dedicated Java model checking tool, it does not support versions of the Java Development Kit (JDK) beyond version 8. This presents a significant limitation, as JDK 9 introduced support for memory access modes - a critical feature in modern concurrency. As such, Java Pathfinder cannot be considered a state of the art model checker for programs that use mixed access modes.

8. CONCLUSIONS

We have implemented a JAM21 model checking algorithm in the generic model checker GenMC [4]. With just one consistency axiom, JAM21 is attractively simple, especially compared to other state of the art memory models, for example RC11. Simplicity of JAM21 steams from encapsulating all synchronization between memory events in the *visibility-order* relation; from which the consistency axiom can be directly computed.

We classify JAM21 as incomparable to RC11; for certain programs it restricts more executions than RC11, in others it is more relaxed. The more restrictive behavior of JAM21 often stems from treating the initial write as a write access to all memory locations. Such instruction creates dependencies between accesses to different memory locations. Possibly, if the initial write was treated as in RC11, for some programs, JAM21 could emerge as a more relaxed model than RC11.

JAM21 is also stronger than RC11 in its treatment of `sc` accesses (equivalent to Java's volatile). It explicitly enforces ordering of these accesses through the `pushto` relation. In fact, JAM21 attempts to establish a *sequentially consistent* ordering of all `sc` events in the program, as described by Lamport [9]. While this greatly aids intuitive reasoning about execution order during model checking, it comes at a significant performance cost due to the overhead of computing linearisations.

Apart from `pushto`, JAM21 establishes synchronization between threads through the *reads-from* relation. In the absence of these relations, no *visibility-order* is established between threads. As a result, threads are unable to observe each other's effects, meaning no inter-thread synchronization occurs, and therefore, such executions cannot be deemed inconsistent under JAM21 rules.

However, even in these cases, a program may still contain write operations that execute in some order. While such a program may be considered consistent, the final memory state it produces could result from a sequence of writes that would be deemed inconsistent with the addition of just a single arbitrary *reads-from* edge. This raises a broader question about how a program's output should be defined: is it the final memory state at the end of execution, or is it the data observed through read operations? JAM21 appears to favor the latter interpretation.

We found that our implementation of the JAM21 vector clock based model checking algorithm tends to take proportionally more time than RC11 when measured against the total number of events traversed during model checking; but only in cases where each execution graph contains a single linearisation.

When multiple linearisations are present, the model checking time increases proportionally to the average number of linearisations per execution graph. This effect is especially observed in programs with frequent `sc` accesses. In such cases, model checking time can grow exponentially, as the number of valid linearisations may reach the factorial of the number of `sc` events in the worst case.

8.1. FUTURE WORK

We hereby identify several promising directions for future work that could enhance the efficiency of JAM21 model checking. Most urgently, the performance bottleneck caused by linearisation computation should be addressed. One potential improvement is to calculate linearisations incrementally during model checking, rather than computing them separately upfront. In this approach, model checking would begin with the full set of possible linearisations, and as the algorithm traverses the execution graph, it would prune invalid linearisations along the way. If no valid linearisation remains by the end of the traversal, the execution would be declared inconsistent.

Additionally, in our work, we have hypothesized that it might be possible to replace the linearisation with *from-reads* edges. For example, in graph 4.6, the synchronization effect of `pushto` edge from a to d might be instead delivered by a `fr` edge from b to c. Whereas this approach would still require creating multiple possible execution graphs for each execution, it could decrease computational overhead required by calculating the linearisation.

Another important area of future work is mechanization of the equivalence proof given in this paper. Whereas our empirical evaluation suggests the two implementations are equivalent, perhaps an unaccounted edge case might be found while mechanizing the proof. Proof mechanization might also help in better understanding of the JAM21 model, which might further be helpfull in substituting the `pushto` relation with `fr` edges.

In the performance evaluation of our implementations, we have noticed the algorithms only utilize a single multiprocessor core with minimal memory usage. Moreover, in both JAM21 model checkers, consistency is computed for each valid linearisation, with the loop implemented at the top level of the control flow. This makes our program an ideal candidate for parallelization. Parallelizing the algorithm would bring the most benefit for programs with multiple `sc` accesses in different threads, as such programs yield the most possible linearisations.

Finally, we believe it might be interesting to investigate the ability to adopt JAM21 to be used with Kater [20]. Kater is a tool that can automatically generate a GenMC-compatible model checker given an arbitrary memory model. Kater uses a *deterministic finite automation* to decide wether an execution is consistent. Perhaps this approach could rival the performance of JAM21 vector clock algorithm.

9. REFERENCES

[1] B. Norris and B. Demsky, "A practical approach for model checking c/c++11 code," *ACM Trans. Program. Lang. Syst.*, vol. 38, no. 3, May 2016, ISSN: 0164-0925. DOI: 10.1145/2806886. [Online]. Available: https://doi.org/10.1145/2806886.

[2] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas, "Stateless model checking for TSO and PSO," *CoRR*, vol. abs/1501.02069, 2015. arXiv: 1501.02069. [Online]. Available: http://arxiv.org/abs/1501.02069.

[3] 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, Oct. 2018. DOI: 10.1145/3276505. [Online]. Available: https://doi.org/10.1145/3276505.

[4] M. Kokologiannakis and V. Vafeiadis, "Genmc: A model checker for weak memory models," in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I*, Berlin, Heidelberg: Springer-Verlag, 2021, pp. 427–440, ISBN: 978-3-030-81684-1. DOI: 10.1007/978-3-030-81685-8_20. [Online]. Available: https://doi.org/10.1007/978-3-030-81685-8_20.

[5] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing c++ concurrency," *SIGPLAN Not.*, vol. 46, no. 1, pp. 55–66, Jan. 2011, ISSN: 0362-1340. DOI: 10.1145/1925844.1926394. [Online]. Available: https://doi.org/10.1145/1925844.1926394.

[6] J. Bender and J. Palsberg, "A formalization of java's concurrent access modes," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. DOI: 10.1145/3360568. [Online]. Available: https://doi.org/10.1145/3360568.

[7] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer, "Repairing sequential consistency in c/c++11," *SIGPLAN Not.*, vol. 52, no. 6, pp. 618–632, Jun. 2017, ISSN: 0362-1340. DOI: 10.1145/3140587.3062352. [Online]. Available: https://doi.org/10.1145/3140587.3062352.

[8] S. Liu, J. Bender, and J. Palsberg, "Compiling volatile correctly in java," in *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, 2022.

[9] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, 1979. DOI: 10.1109/TC.1979.1675439.

[10] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell, "The problem of programming language concurrency semantics," in *Programming Languages and Systems*, J. Vitek, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 283–307, ISBN: 978-3-662-46669-8.

[11] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, “X86-tso: A rigorous and usable programmer’s model for x86 multiprocessors,” *Commun. ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010, ISSN: 0001-0782. DOI: 10.1145/1785414.1785443. [Online]. Available: https://doi.org/10.1145/1785414.1785443.

[12] S. Chakraborty, “On architecture to architecture mapping for concurrency,” *CoRR*, vol. abs/2009.03846, 2020. arXiv: 2009.03846. [Online]. Available: https://arxiv.org/abs/2009.03846.

[13] J. Ševčík and D. Aspinall, “On validity of program transformations in the java memory model,” in *ECOOP 2008 – Object-Oriented Programming*, J. Vitek, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 27–51, ISBN: 978-3-540-70592-5.

[14] H.-J. Boehm, “Threads cannot be implemented as a library,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’05, Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 261–268, ISBN: 1595930566. DOI: 10.1145/1065010.1065042. [Online]. Available: https://doi.org/10.1145/1065010.1065042.

[15] H. C. Tunç, P. A. Abdulla, S. Chakraborty, S. Krishna, U. Mathur, and A. Pavlogiannis, “Optimal reads-from consistency checking for c11-style memory models,” *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Jun. 2023. DOI: 10.1145/3591251. [Online]. Available: https://doi.org/10.1145/3591251.

[16] M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis, “Effective stateless model checking for c/c++ concurrency,” *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017. DOI: 10.1145/3158105. [Online]. Available: https://doi.org/10.1145/3158105.

[17] LLVM Project, *The LLVM Compiler Infrastructure*, Accessed: 2025-03-28, 2025. [Online]. Available: https://llvm.org/.

[18] O. Lahav and V. Vafeiadis, “Explaining relaxed memory models with program transformations,” in *FM 2016: Formal Methods*, J. Fitzgerald, C. Heitmeyer, S. Gnesi, and A. Philippou, Eds., Cham: Springer International Publishing, 2016, pp. 479–495, ISBN: 978-3-319-48989-6.

[19] W. Luo and B. Demsky, “C11tester: A race detector for C/C++ atomics technical report,” *CoRR*, vol. abs/2102.07901, 2021. arXiv: 2102.07901. [Online]. Available: https://arxiv.org/abs/2102.07901.

[20] M. Kokologiannakis, O. Lahav, and V. Vafeiadis, “Kater: Automating weak memory model metatheory and consistency checking,” *Proc. ACM Program. Lang.*, vol. 7, no. POPL, Jan. 2023. DOI: 10.1145/3571212. [Online]. Available: https://doi.org/10.1145/3571212.

[21] H. Ponce-de-León, F. Furbach, K. Heljanko, and R. Meyer, “Dartagnan: Bounded model checking for weak memory models (competition contribution),” in *Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part II*, Dublin, Ireland: Springer-Verlag, 2020, pp. 378–382, ISBN: 978-3-030-

45236-0. DOI: 10.1007/978-3-030-45237-7_24. [Online]. Available: https://doi.org/10.1007/978-3-030-45237-7_24.

[22] Java PathFinder Contributors, *Java PathFinder (JPF) Core Wiki*, Accessed: 2025-05-23, 2025. [Online]. Available: https://github.com/javapathfinder/jpf-core/wiki.

10. Appendices

This page is left blank intentionally.

A. Reproduction Package

Our implementations can be found in a git repository published at

`https://github.com/MichRacz00/jam21`

at tag *v1.1*. The repository contains a copy of GenMC generic model checker together with both relation based and vector clock based implementations. To compile code follow instructions included in the *README.md* file. Additionally, the repository contains a *benchmark.sh* bash script for automated testing.

B. Pseudocode Struct $\mathbb{Q}$

The task of struct $\mathbb{Q}$ is to keep track of most recent events given certain characteristics. Those events are stored in public struct fields. The struct contains all necessary events to calculate JAM21 relations. The `AddEvent(e)` function recognizes given event and adds it to appropriate fields.

Struct 4: Struct $\mathbb{Q}$ keeps track of previously seen events

```
1  Struct Q:
       /* Structure public fields:                                  */
2      synch, WR, relFence, acqFence, scFence, sc
3
       /* Structure private fields                                  */
4      prevPendingRelFence, pendingRelFence, pendingWR,
5
6      Function AddEvent(e):
7          if e ∈ E^sc ∪ E^rel ∪ E^acq then
8              Q.synch ← e
9          end
10         if e ∈ W ∪ R then
11             Q.pendingRelFence ← Q.prevPendingRelFence
12             Q.pendingWR ← e
13         end
14         if e ∈ F^rel then
15             Q.prevPendingRelFence ← e
16         end
17         if e ∈ F^acq then
18             Q.relFence ← Q.pendingRelFence
19             Q.WR ← Q.pendingWR
20             Q.acqFence ← e
21         end
22         if e ∈ F^sc then
23             Q.scFence ← e
24         end
25         if e ∈ E^sc then
26             Q.sc ← e
27         end
28     end
29 end
```

C. CONSISTENT EXECUTIONS

This appendix contains the number of consistent executions of all tests used to check JAM21 implementation. Comparison is performed between JAM21 and RC11 memory model implemented in GenMC. All results were generated by running GenMC with `-check-consistency-point=exec` and `-check-consistency-type=full` flags. Tests producing different number of consistent executions under the two memory models are highlighted. Certain tests have multiple similar versions; we indicate their results with redundant table rows.

C.1. DATA-STRUCTURES

Test Name	RC11		JAM21	
	Produced	**Blocked**	**Produced**	**Blocked**
barrier	2	1	2	1
buf_ring	1	1	1	1
chase-lev	59	0	59	0
dq-opt	1432	63	111	1
dq	1432	48	111	1
dq	1432	64	62	1
fcombiner-async	24	22	24	22
linuxrwlocks	2	2	2	2
mpmc-queue	3	4	3	4
ms-queue	4	1	4	1
ms-queue-dynamic	2	8	2	8
mutex	12	1	12	1
qu	75	0	75	0
qu-opt	1	1	1	1
spinlock	4	2	4	2
stc	37	0	37	0
stc-opt	183	0	183	0
ticketlock	2	1	2	1
treiber-stack	18	4	18	4
treiber-stack-dynamic	8	8	8	8
ttaslock	4	3	4	3
ttaslock-opt	2	5	2	5
twalock	4	1	4	1

C.2. LITMUS

Test Name	RC11		JAM21	
	Produced	Blocked	Produced	Blocked
2+2W	4	0	4	0
2+2W	4	0	4	0
2+2W+2sc+scf	3	0	4	0
2+2W+3sc+rel1	4	0	4	0
2+2W+3sc+rel2	4	0	4	0
2+2W+4sc	3	0	4	0
2+2W+scfs	3	0	4	0
2CoWR	4	0	4	0
2CoWR	4	0	4	0
2CoWR	4	0	4	0
2CoWR	4	0	4	0
2CoWR	4	0	4	0
2CoWR	4	0	4	0
assume-ctrl	4	1	4	1
atomicpo	4	0	4	0
casdep	2	0	2	0
ccr	2	0	2	0
cii	6	0	6	0
CoRR	6	0	6	0
CoRR	6	0	6	0
CoRR0	4	0	4	0
CoRR0	4	0	4	0
CoRR0	4	0	4	0
CoRR0	4	0	4	0
CoRR1	36	0	36	0
CoRR1	36	0	36	0
CoRR1	36	0	36	0
CoRR1	36	0	36	0
CoRR1	36	0	36	0
CoRR2	72	0	72	0
CoRR2	72	0	72	0
CoRR2	72	0	72	0
CoRR2	72	0	72	0
CoRR2	72	0	72	0
CoRW	3	0	3	0
CoRW	3	0	3	0
CoWR	3	0	3	0
CoWR	3	0	3	0
cumul-release	8	0	8	0

default	12	0	12	0
default	12	0	12	0
default	12	0	12	0
default	12	0	12	0
detour	9	0	9	0
fr+w+w+w+reads	210	0	210	0
inc+inc+RR+W+RR	600	0	600	0
inc2w	6	0	6	0
IRIW-acq-sc	16	0	16	0
IRIWish	28	0	28	0
isa2	7	0	7	0
JLT	8	0	8	0
JLT	8	0	8	0
JLT	8	0	8	0
JLT	8	0	8	0
LB	3	0	3	0
LB	3	0	3	0
LB+acq	3	0	3	0
LB+addr	3	0	3	0
LB+addr-fun	3	0	3	0
LB+addr-ind-fun	3	0	3	0
LB+ctrl	3	0	3	0
LB+ctrl+rel	3	0	3	0
LB+dep	3	0	3	0
LB+dep	3	0	3	0
LB+fr-fr-data+data	18	0	18	0
LB+incMPs	15	0	15	0
LB+rel	3	0	3	0
LB2	3	0	3	0
LB3	7	0	7	0
malloc-dep	2	0	2	0
MCP-rrw	6	0	6	0
MCP-rrw	6	0	6	0
MCP-rrw	6	0	6	0
MP	3	0	3	0
MP	3	0	3	0
MP+incMP	7	0	7	0
MP+rel+acq	2	0	2	0
MP+rel+acq	2	0	2	0
MP+rel+acqf	2	0	2	0
MP+relf+acq	2	0	2	0
MP+relf+acqf	2	0	2	0
MP+rels+acq	3	0	3	0
MP+rels+acq	3	0	3	0

MP+rels+acqf	3	0	3	0
MP2	9	0	9	0
MPU+rel+acq	6	0	6	0
MPU+rel+acqf	6	0	6	0
MPU+relf+acq	6	0	6	0
MPU+relf+acqf	3	0	3	0
MPU+relf+acqf	3	0	3	0
MPU+relf+acqf	6	0	6	0
MPU+rels+acq	12	0	12	0
MPU+rels+acqf	12	0	12	0
MPU2+rel+acq	14	0	14	0
MPU2+rel+acqf	14	0	14	0
MPU2+relf+acq	14	0	14	0
MPU2+relf+acqf	14	0	14	0
MPU2+rels+acq	35	0	35	0
MPU2+rels+acqf	35	0	35	0
ori	5	0	5	0
peterZ	7	0	8	0
po-loc	3	0	3	0
psc-ar	1	2	0	2
psc-base-notin-ar	2	1	2	1
psc-sbhbsb	1	1	1	1
rel-B-cumul-acq	16	0	16	0
rel-pporf	3	0	3	0
relacq-B-cumul	8	0	8	0
relseq	16	0	16	0
rfi-preserved	3	0	3	0
rii	3	0	3	0
rinc+wr	3	0	3	0
riwi	3	0	3	0
RMWFix	4	0	4	0
RWC+syncs	7	0	7	0
S	4	0	4	0
S	4	0	4	0
S+rel+acq	2	0	2	0
S+rel+acq	2	0	2	0
S+rel+acqf	2	0	2	0
S+relf+acq	2	0	2	0
S+relf+acqf	2	0	2	0
S+rels+acq	3	0	3	0
S+rels+acq	3	0	3	0
S+rels+acqf	3	0	3	0
SB	4	0	4	0
SB	4	0	4	0

C

SB+2sc+scf	3	0	3	0
SB+3sc+acq	4	0	4	0
SB+3sc+rel	4	0	4	0
SB+4sc	3	0	3	0
SB+assert	3	1	3	1
SB+assert	3	1	3	1
SB+rfis	4	0	3	0
SB+scfs	3	0	3	0
small0	9	0	9	0
small0	9	0	9	0
small0	9	0	9	0
small0	9	0	9	0
small1	8	0	8	0
small1	8	0	8	0
small1	8	0	8	0
small1	8	0	8	0
TC1	3	0	3	0
TC2	4	0	4	0
viktor-relseq	180	0	180	0
W+JW	1	0	1	0
W+JW	1	0	1	0
W+RRR+W	20	0	20	0
W+RWC	7	0	8	0
wcii	24	0	24	0
WRC+dep	8	0	8	0
WWmerge1	18	0	18	0
WWmerge2	8	0	8	0
WWR+2WR	0	10	2	10
Z6.U	24	0	24	0
Z6+acq	7	0	8	0
Z6+acq	7	0	8	0

D. FENCES PROGRAM

In this appendix we present an implementation of program presenting incorrect compilation scheme presented by [8]. We have shown this program previously as program 4.1. Since GenMC natively takes C code, we have translated machine code instructions according to the compilation scheme. We implement `hwsync` barriers as `sc` fences and `lwsync` barrier as `ra` fence. Volatile accesses are compiled to relaxed accesses with appropriate barriers (see Java to IBM Power compilation scheme, section 4).

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdatomic.h>
#include <genmc.h>

atomic_int x;
atomic_int y;

void __VERIFIER_assume(int);

void *thread_1(void *unused)
{
  atomic_store_explicit(&x, 2, memory_order_relaxed);
  atomic_thread_fence(memory_order_seq_cst); //hwsync

  int y_local = atomic_load_explicit(&y, memory_order_relaxed);
  __VERIFIER_assume(y_local == 0);

  return NULL;
}

void *thread_2(void *unused)
{
  atomic_store_explicit(&y, 1, memory_order_relaxed);
  return NULL;
}

void *thread_3(void *unused)
{
  int y_local = atomic_load_explicit(&y, memory_order_relaxed);
  __VERIFIER_assume(y_local == 1);
  //atomic_thread_fence(memory_order_seq_cst);
  atomic_store_explicit(&x, 1, memory_order_relaxed);
  return NULL;
}
```

```
37
38 void *thread_4(void *unused)
39 {
40   int x_local = atomic_load_explicit(&x, memory_order_relaxed);
41   __VERIFIER_assume(x_local == 1);
42
43   atomic_thread_fence(memory_order_seq_cst); // hwsync
44
45   x_local = atomic_load_explicit(&x, memory_order_relaxed);
46   __VERIFIER_assume(x_local == 2);
47   return NULL;
48 }
49
50
51 int main()
52 {
53   pthread_t t1, t2, t3, t4;
54
55   // To break [init] as sc access
56   atomic_store_explicit(&x, 0, memory_order_relaxed);
57   atomic_store_explicit(&y, 0, memory_order_relaxed);
58
59   if (pthread_create(&t1, NULL, thread_1, NULL))
60     abort();
61   if (pthread_create(&t2, NULL, thread_2, NULL))
62     abort();
63   if (pthread_create(&t3, NULL, thread_3, NULL))
64     abort();
65   if (pthread_create(&t4, NULL, thread_4, NULL))
66     abort();
67
68   return 0;
69 }
```

E. ONLY-SC/ONLY-RLX PROGRAMS

This appendix contains a test program used to benchmark linearisation of `pushto` relation. The behaviour and model checking time drastically change if all accesses are replaced with `memory_order_relaxed`.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdatomic.h>
#include <genmc.h>

atomic_int x;
atomic_int y;

void *thread_1(void *unused)
{
  atomic_store_explicit(&x, 1, memory_order_seq_cst);
  atomic_load_explicit(&y, memory_order_seq_cst);
  atomic_store_explicit(&x, 2, memory_order_seq_cst);
  return NULL;
}

void *thread_2(void *unused)
{
  atomic_load_explicit(&y, memory_order_seq_cst);
  atomic_store_explicit(&x, 3, memory_order_seq_cst);
  atomic_load_explicit(&x, memory_order_seq_cst);
  return NULL;
}

void *thread_3(void *unused)
{
  atomic_store_explicit(&x, 4, memory_order_seq_cst);
  atomic_store_explicit(&y, 5, memory_order_seq_cst);
  atomic_load_explicit(&x, memory_order_seq_cst);
  return NULL;
}

void *thread_4(void *unused)
{
  atomic_store_explicit(&x, 6, memory_order_seq_cst);
  atomic_store_explicit(&y, 7, memory_order_seq_cst);
  atomic_load_explicit(&y, memory_order_seq_cst);
  return NULL;
```

```
40 }
41
42
43 int main()
44 {
45   pthread_t t1, t2, t3, t4;
46
47   if (pthread_create(&t1, NULL, thread_1, NULL))
48     abort();
49   if (pthread_create(&t2, NULL, thread_2, NULL))
50     abort();
51   if (pthread_create(&t3, NULL, thread_3, NULL))
52     abort();
53   if (pthread_create(&t4, NULL, thread_3, NULL))
54     abort();
55
56
57   return 0;
58 }
```

E