Algorithmic Verification of Concurrent and Distributed Data Structures

Jad Hamza

Reviewer	Parosh Aziz Abdulla	Uppsala University
Reviewer	Ganesan Ramalingam	Microsoft Research India
Examiner	Paul Gastin	École Normale Supérieure de Cachan
Examiner	Christoph Kirsch	University of Salzburg
Examiner	Viktor Vafeiadis	Max Planck Institute for Software Systems
Advisor	Ahmed Bouajjani	Université Paris Diderot
Advisor	Constantin Enea	Université Paris Diderot

November 27, 2015

Acknowledgments. Many thanks to my advisors Ahmed Bouajjani and Constantin Enea, who were always very available for me, and always knew how to motivate me with interesting problems. I have learned a lot about research, making presentations, writing papers, etc., thanks to you, it was a pleasure.

Thanks a lot to the reviewers, Parosh Aziz Abdulla and Ganesan Ramalingam, who accepted to read and report on the manuscript in minimal time. And thanks to the examiners Paul Gastin, Christoph Kirsch, and Viktor Vafeiadis for accepting to be part of the jury.

Thanks to the whole LIAFA team. It's a really nice working environment thanks to all of you. The administrative (Nathalie/Noëlle) and technical (Houy/Laifa) teams are particularly effective. Special thanks to Mike who helped me a lot during my first years. Thanks also to all the PhD and post-doc students, it was very nice meeting and discussing with all of you.

Thanks a lot to my parents, brothers, family and friends for their support, and especially to Coralie. I love you very much.

Contents

Chapter 1

Introduction

1.1 Context – Concurrency at Every Level

Computers are an integral part of our everyday lives. They are used in critical applications such as health or transportation, but also for communication in general, sharing of information, and exchange of data. Data and information are usually shared by people all around the world, and as a result, can usually be accessed *concurrently* by several people at once.

Concurrency is naturally present at every level of a computer system. At the hardware level, even the cheapest smartphones now have two cores. As it becomes harder and harder to improve the efficiency of a single processor, constructors use multiple cores as a cost-efficient way to improve the performance of a machine. At the software level, the operating system is made of several components running concurrently. These components include device drivers interacting directly with the hardware for input or output (keyboard, sound, display, ...) and system services, for instance upgrade mechanisms or power management. They must all share the resources provided by the hardware, such as the shared memory, or the computational resources.

Then, the multitude of user applications are running alongside the components of the operating system, sharing the same resources. Some user applications are themselves concurrent, either because it makes sense conceptually to develop them as such, and/or because parallelizing some computations makes them more efficient.

Furthermore, most machines are connected to the Internet, and connect with one another to share information, exchange data, connect to online services, and so on. Here again, we see concurrent behaviors, when thousands of machines can connect to the same server. As a technique to reduce the latency to their users, online services distribute their servers in strategically placed countries. Such servers also run concurrently, and synchronize using message-passing protocols.

1.2 Problems in Presence of Concurrency

The design of concurrent and distributed programs is very challenging, as they are composed of several interacting components. The interactions can be through a shared memory or communication channels, and create a combinatorial explosion in the number of possible behaviors of the programs. This makes it hard for the programmers to fully predict and comprehend the behaviors of their own programs. It also makes the problem of analyzing concurrent programs, and proving properties on them, very difficult, as all possible interactions need to be taken into account.

Moreover, developers of concurrent or distributed programs must deal with issues specific to the setting, such as low-level synchronization protocols, network failures and partitions, unbounded communication channels, reordering of messages by the network, reordering of memory instructions by optimized hardware, and must also take care of problems created by the sharing of resources between concurrent users, such as deadlocks.

In order to hide these difficulties, programmers use different levels of abstraction. More precisely, they use interfaces at a particular level of the software stack, which abstracts away the low-level details of the implementation. Such interfaces provide for instance abstract data structures which can be manipulated through high-level operations to add or remove elements.

Due to the intricacy of writing implementations using the low-level features mentioned above, it is essential to develop automated tools to help programmers write concurrent or distributed software – for instance, verification tools used to catch violations, or certify programs with respect to a given abstract specification.

1.3 Observational Refinement

In order to provide abstractions which hide lower-level implementation details, developers write *libraries*, providing a clean interface to the *clients* using them. They can be used to represent well-defined data structures (such as stacks or queues), synchronization objects (such as locks or semaphores), or more generally any code that can be reused in more than one place.

Consider a library representing a stack. If there is only one user of the library, the semantics is simple. The user expects a Pop operation to return the last element *Pushed* on the stack. In presence of concurrency however, different users may access the stack at the same time. They want to imagine they are updating a unique stack, one after the other, in a high-level interleaving of their operations (see Fig 1.1a). This must be the case regardless of the low-level implementation of the stack.

The developer of the library can ensure the *atomicity* expected by the users by making sure no two users access the data structure at the same time. This can be done by using *coarse-grained* locking, as seen in the implementation Fig 1.2a of the Atomic Stack. In this implementation, every time a user wants

to do an operation on the stack, they must acquire a global lock, preventing every other user to access the stack during the operation. At the end of the operation, they release the global lock.

For performance reasons however, this is not an acceptable solution, as users can only access the structure one at a time. As a result, the developers of concurrent libraries don't use coarse-grained locking, or sometimes don't use locking at all. Instead, they use low-level synchronization mechanism (such as compare-and-swap (`cas`) instructions). One particular schema is to let several threads try to update the data structures, and using the low-level synchronization mechanisms provided by the hardware, check for interference from other threads. In case of interference, they abort the update and try at a later time.

Fig 1.2b) gives the example of the Treiber Stack, and a concurrent execution is depicted in Fig 1.1b. The Treiber Stack enables a better *throughput* than the Atomic Stack, meaning that when several users are using the library, it can perform more operations per second. The goal of the developer is to create efficient libraries, which can be accessed concurrently with the highest possible *throughput*, while giving the illusion to the clients of the library that they are accessing the data structure atomically, one after the other.

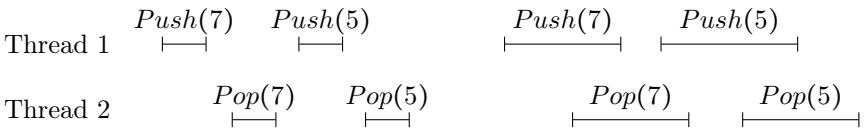
For our example, the clients will use the Treiber Stack for efficiency, but will have the illusion of using the Atomic Stack. This notion is called *observational refinement*. More specifically, we say that a library $\mathcal{L}_1$ refines a library $\mathcal{L}_2$, if any behavior of any program (client) making calls in its code to the library $\mathcal{L}_1$ can be obtained were $\mathcal{L}_2$ used instead. Here, the Treiber Stack refines the Atomic Stack. The Atomic Stack can be considered as the abstract high-level interface (or specification) for the Treiber Stack.

When observational refinement is established, clients using the Treiber Stack in their own programs can reason as though they are using the Atomic Stack. More precisely, they can prove safety properties on their programs using the Atomic Stack, which, by observational refinement, will automatically transfer to their programs using the Treiber Stack.

Going a level higher, clients can implement more complex libraries, which use the Treiber Stack as a subroutine. They can then prove properties on these libraries, such as observational refinement itself, as though they were in fact using the Atomic Stack [58].

As the low-level implementations of concurrent data structures are complex and difficult to get right, it is important to study the problem of automatically verifying whether a library refines another one. As presented, the definition is very hard to work with. Specifically, there is a universal quantification over all possible client programs. Client programs can be unboundedly large, and can call the library in an arbitrary manner. They can use an unbounded number of concurrent threads to call the library.

As a consequence, the first step towards automation is to obtain a deep and formal understanding of observational refinement, in order to reduce it to properties which are easier to reason about. The problem of observational refinement can be posed at different levels. At the level of a single computer, the different operations of the library which are running in parallel communicate



(a) Sequential access.

(b) Concurrent access.

Figure 1.1: Time goes from left to right. Horizontal lines delimit the time interval of an *operation*, i.e. the time between when it was called, and when it returned. Each execution has four operations.

```
int Pop () {
   LOCK
    if (Top == NULL) return EMP
    int result = Top->data
    Top = Top->next
   UNLOCK
  return result
}
```

(a) Atomic Stack
Reference implementation, where each thread wanting to pop an element locks the entire structure.

```
int Pop () {
  node *t
  // Retry mechanism
  while (true) {
    *t = Top
    if (t == NULL) return EMP
    node n = t->next
    // Low-level sync
    // Interference checking
    if cas(&Top,t,n) {
      int result = t->data
      return result
    }
  }
}
```

(b) Treiber Stack
Optimistic Implementation, where several threads can try to pop at the same time; they abort and retry in case of interference.

Figure 1.2: The *Pop* method of two implementations.

through a global *shared memory*. At the level of a network, the operations of a library can run on different *nodes* or *sites*, which are machines communicating through *message-passing* systems.

1.4 Shared Memory

In the context of shared memory, *linearizability* was introduced in 1990 by Herlihy and Wing [31] as a correctness criterion for concurrent libraries. For two decades, it was informally assumed that linearizability implies observational refinement. Only recently, Filipovic et al. [21] formally proved that a library $\mathcal{L}_1$ refines $\mathcal{L}_2$ if and only if every possible execution of $\mathcal{L}_1$ is *linearizable* with respect to (an execution of) $\mathcal{L}_2$. The proof has some limitations which we will discuss later.

A concurrent execution e of a library is linearizable with respect to a sequential[1] execution e' if it is possible to find an order on the *operations* of e, respecting the precedence relation between them, in order to obtain the execution e'. Concurrent operations, which are overlapping in time, can thus be ordered arbitrarily. Equivalently, e is linearizable with respect to e' if for each operation of e, we can find a point in time during its interval of execution, called a *linearization point*, such that the sequence of linearization points corresponds to e'.

In linearizability, there is no more a universal quantification over all possible client programs (as there is in observational refinement). The definition is still hard to reason about, as we need to find, for every possible execution of $\mathcal{L}_1$, a valid linearization order, or a valid placement of linearization points. Each execution can have an exponential number of orders in the number of operations. As a consequence, it remains very challenging to write efficient libraries, and to prove that they are linearizable.

1.4.1 Theoretical Complexity

The first complexity results about the automatic verification of linearizability were published in 1996 and 1997. In particular, Alur et al. [3] studied the complexity of the verification problem in a setting where the library only has a finite shared memory, and where it can be called by a bounded number of threads. They showed that verifying whether all the executions of a library are linearizable with respect to a regular specification [2] is in EXPSPACE.

For context, this represents an exponential blowup with respect to more common model-checking problem, such as *state reachability*. The question of state reachability asks whether a particular control-state in the code of the library can be reached in some execution of the library. This problem is known to be PSPACE-complete when the library can only be used by a bounded number of threads.

Even though no lower bound (besides PSPACE-hardness) was given by Alur et al. [3], the EXPSPACE complexity obtained suggested that there is no trivial way of reducing linearizability to state reachability in polynomial time. Overall, Alur et al. [3] gave the first complexity results for the problem of verifying whether a library is linearizable, but left open a complexity gap, and did not address the problem in presence of an unbounded number of threads.

Gibbons and Korach [24] studied the complexity of checking whether a single trace is linearizable, and showed that this problem is already NP-complete, which suggests that there is no way around testing the exponential number of all possible linearizations, and that it might not be possible to avoid the exponential blowup obtained in Alur et al. [3].

[1] We will also define linearizability with respect to concurrent executions later on.

[2] Regular language described by a non-deterministic finite automaton.

1.4.2 Fixed Linearization Points

Due to the difficulty of reasoning about linearizability in general, a stricter criterion has been considered. It consists in manually specifying the linearization points. For example, in the Treiber Stack, the linearization point of the *Pop* operation is the `cas` instruction. This is the point at which the effect of the operation becomes visible to other threads which are executing concurrently.

From a developer's perspective, it is much easier to write libraries using this technique. The fixed linearization points enable the developer to consider the whole effect of the operation is taking place on a single step. It also becomes easier to check the correctness of the library, as it is only needed to verify whether the linearization points which are specified form a valid linearization with respect to the given specification.

This approach has been applied successfully to prove the correctness of several libraries [55, 4, 54, 7, 39, 20, 44, 51, 57, 15, 47, 1]. Checking that the specified linearization points are valid can by reduced to standard model-checking approaches such as invariant checking or control-state reachability.

The issue with this approach is that it cannot never conclude that a library is not linearizable, as it can be that the developer simply made a mistake by choosing those points. Moreover, this approach is difficult to use for some implementations, as it is not always clear where to place the linearization points. They can depend from the execution of future operations or placed in concurrently executing operations.

1.4.3 Non-fixed Linearization Points

For instance, the Time-Stamped Stack [19] was recently introduced as a very efficient stack implementation. Internally, the implementation does not maintain a totally ordered stack, but instead uses partial orders where concurrent *Push* operations are not ordered. The correctness of this implementation could not be obtained with the approach of manually specifying the linearization points. Instead, the proof was done by carefully analyzing the specification against which linearizability is checked (in that case, a stack). They extracted properties which are equivalent to checking linearizability with respect to the stack, and proved that their implementation satisfies them.

This follows a line of work started by Henzinger et al. [29], which reduce linearizability with respect to the queue to the verification of simpler properties. This approach can be applied regardless of the implementation, and without the need of manually specifying the linearization points. Dodds et al. [19] studied the same question for the stack, but their reduction is not fully independent from the implementation.

The work of Abdulla et al. [1] also reduces linearizability to checking simple properties, and applies to more data structures than queues or stacks, but still requires manually specified linearization points.

Another way to reduce the complexity of checking linearizability is to restrict the kind of implementations which are allowed. Černý et al. [53] studied

the problem of verifying implementations which make use of lists. They proposed an automata-based model which can express such implementations, and showed that it was decidable to check whether a finite number of operations are linearizable. This is far from the problem of checking whether all executions of a library (unbounded number of operations) are linearizable, but it is a step in the right direction.

1.5 Message-Passing Systems

In theory, linearizability can be applied to any system, be it a concurrent system using a shared memory (a single computer) or a distributed one, with the different nodes communicating through message-passing protocols. However, distributed systems are usually designed to be *partition tolerant*, meaning that they can still function in case the distributed network becomes partitioned. Moreover, they are designed to be *available*, meaning that when a client calls a method, the library should respond to clients only based on local information, without waiting to synchronize with other sites. It was shown that these two requirements combined with linearizability are not possible to implement in general [25]. Some recent work [17] identify conditions (like the commutativity of operations) under which linearizability is possible in message-passing systems, but still at the cost of availability.

Thus, in practice, developers of libraries used on a distributed network tend to drop the requirement of linearizability (and thus, of observational refinement). They instead fall back to weaker criteria that they believe are sufficient for the needs of their clients, and which can be implemented efficiently without the need for too much synchronization between the nodes (see Riak, Cassandra, MongoDB, Redis, HBase, Bayou [50], C-Praxis [42], IceCube [35], Telex [6]).

We are still in the very early stages of understanding these weaker criteria. As they are becoming widely used, even by large tech companies, the need for formalization and automated tools grows. Such companies often have dedicated research teams studying these weaker criteria, such as *eventual consistency* or *causal consistency*. They sometimes propose implementations which can be used by their users, or used in their internal applications (e.g., Amazon Simple Storage Service, Google App Engine Datastore), which are supposed to be eventually or causally consistent.

Informally, eventual consistency lets the different nodes of the distributed system temporarily diverge, but ensure that they eventually converge to the same state. The way we define the freedom that the nodes have, and their convergence, leads to several possible formal definitions of eventual consistency [13, 27, 46]. Burckhardt et al. [13] gave the first formal framework to reason in a uniform way about the different weak consistency criteria used in practice, including eventual consistency. As mentioned in their paper, their definition is too strict to be applied for systems with speculative executions and rollbacks, which are techniques used by the systems mentioned above.

For some applications, eventual consistency is too weak (for all definitions).

For instance, a user of a library could want the guarantee that all operations submitted to the same site are being executed in that order. Several of the distributed systems mentioned above also implement such stronger consistency criteria. Causal consistency, for instance, ensures that operations which are causally dependent[3] are always executed in the same order by all sites.

These criteria, being weaker than linearizability, do not ensure the notion of observational refinement mentioned previously. To this date, there is very little work on understanding how we could formally connect them to weaker forms of observational refinement. Such work would help understand precisely the guarantees that one obtain by programming on top of libraries satisfying these criteria.

Despite this, since they are widely used in practice, there is value in developing formal methods to verify them. As we are still in the early stages of even obtaining formal definitions for these criteria, there are no work about their automatic verification (as far as we know).

1.6 Contributions

1.6.1 Observational Refinement and Linearizability

We revisit the connection between observational refinement and linearizability and prove that they are equivalent in the general case. This is an extension of the result given by Filipovic et al. [21], which can be applied in the general case, in presence of *pending* operations (where an operation of the library didn't return).

Moreover, our study led to important new characterizations for observational refinement. We show in particular that observational refinement is equivalent to the inclusion of the executions of the libraries. We abstract the executions of a library using *histories*, which are partial orders describing the *happens-before* order (the precedence relation) between operations of an execution, and show that linearizability is also equivalent to the inclusion of the histories of the libraries. More precisely, we show that a library $\mathcal{L}_1$ refines a library $\mathcal{L}_2$ if the executions (resp., the histories) generated by $\mathcal{L}_1$ are a subset of the ones generated by $\mathcal{L}_2$.

These characterizations give us a fresh perspective on observational refinement, and a better understanding for it. They form the basis of the bug-detection techniques of Chapter 5, which will be described later on.

Originally, linearizability was defined for libraries with respect to sequential specifications. However, when considered as an equivalent to observational refinement, linearizability is defined for a library with respect to another library. There are different ways of defining this extension, and we argue that our definition is the correct one, as we show the equivalence with observational refinement, which was the original (informal) goal of linearizability. We give in Section 3.4

[3] An operation o_2 *depends* on another operation o_1 if a chain of messages originating from o_1 arrives on the site where o_2 is executed, before o_2 is executed.

another possible way to define linearizability, sometimes used in the literature, but show that it is not suitable to characterize observational refinement. The difference is subtle, and only holds in presence of pending operations.

1.6.2 High Complexity of Verifying Linearizability

Then, as a mean to study the complexity of observational refinement, we studied the open problems about the theoretical complexity of linearizability. As explained above, a concurrent execution e of a library is linearizable with respect to a sequential execution e' if it is possible to reorder the concurrent *operations* of e, in a way respecting the happens-before relation, in order to obtain the execution e'.

We show first that checking linearizability is strictly harder than checking control-state reachability in libraries, even in the case where the *specification* $\mathcal{L}_2$ is sequential, and given as a regular language. When there are unboundedly many concurrent threads, we prove that verifying linearizability is undecidable – while control-state reachability is known to be decidable (EXPSPACE-complete). Our proof relies on a reduction from a reachability problem on counter machines with zero tests. The intuition is that the specification $\mathcal{L}_2$ can be used to check that the number of increments and decrements between consecutive zero tests is equal, and this can be done by a regular language because of all the different orderings that linearizations permits. In particular, we can check that the number of increments is equal to the number of decrements by ordering each decrement right after its corresponding increment.

When the number of threads is bounded, we prove that linearizability is EXPSPACE-complete, while control-state reachability for libraries is known to be PSPACE-complete. This closes the complexity gap left open by Alur et al. [3]. We prove the EXPSPACE-hardness even in the case where the specification is sequential, and given as a deterministic finite automaton. We introduce for this a new problem on regular languages, called letter insertion. This problem is easier than linearizability complexity-wise (it can be reduced to it in polynomial time). Still, it captures the main difficulties of linearizability, as we show it is itself EXPSPACE-hard. Moreover, we prove the membership in EXPSPACE even in the case where the specification is a (concurrent) library, while Alur et al. [3] only proved it in the case where the specification is sequential.

In accordance with the intuition that fixing the linearization order makes the verification of linearizability more practical, we give the first decidability and complexity results for the technique of manually specifying the linearization points. In particular, we show that in this context, the complexity of linearizability becomes EXPSPACE-complete for an unbounded number of threads and PSPACE-complete when it is bounded, just like control-state reachability. When the linearization points are manually specified, there is no need to check for all the possible linearizations of an execution, and the linearizability problem reduces to a simple inclusion problem, much easier to solve that linearizability in general. A drawback of this approach is that, for some implementations, the locations of linearization points may be hard to obtain, as they may depend

on the execution of future operations. For the same reason, this technique can never conclude that a library is not linearizable.

This chapter is part of two publications [9, 28].

1.6.3 Efficient Violation Detection

Verification techniques aimed at finding linearizability violations are usually based on the explicit enumeration of all possible linearizations [14, 16, 15]. As a consequence, they are not scalable as the number of operations increase.

Our goal here is to present an efficient bug-finding technique for linearizability, which doesn't suffer from the exponential blowup inherent to other methods. We defined an exploration strategy which has a good coverage, in the sense that violations can be found in the early stages of the exploration, and which is scalable as the number of operations increases. This means that it shouldn't rely on the explicit enumeration of the exponential number of linearizations. Moreover, as it is a bug-detection technique, it doesn't rely on manually specified linearization points; instead, it looks for an execution for which there is no valid linearization.

Our method is based on: 1) our theoretical study linking observational refinement and linearizability to the inclusion the history sets of the libraries, 2) on fundamental properties of histories. We notice that, given an execution of a library, its history is not an arbitrary partial order, but it is an *interval order*. Intuitively, interval orders are partial orders which can be described as a precedence relation on (integer or real) intervals. A history represents the precedence relation between operations of an execution, which is why it can be seen as an interval order, with each interval representing the beginning and end of each operation.

Interval orders have a well-defined measure called their *length*, or *interval length*. Our exploration strategy bounds the interval length of the histories for which we check the inclusion. On the theoretical side, we show that bounding the interval length simplifies linearizability, and makes the problem decidable, even when the number of threads (and operations) is unbounded.

In practice, we show that bounding the interval length is indeed relevant for finding violations, as all violations we experimented could be found with an interval length of 0 or 1. Moreover, we show that the overhead brought by our construction is very small compared to existing techniques based on enumerating all possible linearizations. In particular, when the interval length is fixed, the overhead is not exponential. Using experimental results, we show that our technique can be used in static analysis as well as for testing.

Both the theoretical and practical results rely on representing interval orders of a bounded length using a bounded number of counters, each counter representing the number of operations of a certain kind which have *equivalent* time intervals. Bounding the interval length ensures that there are finitely many equivalent classes of time intervals. Moreover, it enables the use of arithmetic formulas to describe sets of (bounded interval length) histories. Using these

symbolic representations enabled us to avoid the explicit enumeration of all linearizations.

This chapter is part of a publication [11].

1.6.4 Particular Specifications

Since the previous method is based on an underapproximation, it cannot be used to prove that a library is linearizable. We give in this chapter a reduction from linearizability to state reachability, which can be used both to prove or disprove linearizability, and which does not rely on manually specifying the linearization points either. Instead it relies on limiting the type of specifications against which we check linearizability. The method in the previous section can be applied to a larger class of specifications.

We believe that fixing the specification parameter of the verification problem is justified, since in practice, there are few abstract data types (ADTs) for which specialized concurrent implementations have been developed. We provide a uniform methodology for carrying out a reduction from linearizability to state reachability in libraries, and instantiate it on four ADTs: the atomic queue, stack, register, and mutex.

Each ADT is given using an inductive language, which we use to extract a finite number of *bad patterns*. We show that a history is not linearizability with respect to the ADT if and only if it exhibits one of the bad patterns. This means that the detection of linearizability violations using these bad patterns is *complete*. This work can be seen as a generalization of the previously mentioned work of Henzinger et al. [29], enabling to reduce linearizability to simple properties for queues, stacks, registers, and mutexes.

As the bad patterns we introduce are relatively easy to detect, we use them to reduce linearizability to state reachability. This means that in the case of an unbounded number of threads (and a finite-state library), we obtain decidability (EXPSPACE); and in the case of a bounded number of threads, we obtain a PSPACE complexity. This represents an exponential improvement from the complexity of linearizability in general (EXPSPACE-complete).

Our methodology is general and uniform, and we believe it can be applied to other specifications. This represents the first decidability result for linearizability which does not rely on bounding the number of threads or fixing the linearization points.

This chapter is part of a publication [12].

1.6.5 Message-Passing Systems

For message-passing systems, the formalization of the consistency criteria used in practice, and their connection to observational refinement, is still in its very early stages. We give a general framework which can be used to define consistency criteria used in message-passing systems and use it to define eventual consistency, RYW consistency, FIFO consistency, and causal consistency. Un-

like the definitions of Burckhardt et al. [13], our definitions can be applied in the context of speculative executions and rollbacks.

We study the limits of decidability of these criteria when the number of sites[4] is bounded, and show that only the weakest one is decidable. All the other criteria mentioned above are undecidable, even when only two sites are running the library. To prove this, we use the same methodology as for proving the EXPSPACE-hardness of linearizability. We introduce problems over regular or weighted automata which we show equivalent to the correctness criteria, and we prove that these intermediary problems are undecidable. In fact our result is even stronger, and shows that verification for any consistency criterion stronger than RYW consistency and weaker than causal consistency is not decidable.

Giving problems over automata which are equivalent to the consistency criteria is interesting on its own. It enables to connect these notions to problems from a different field, and gives us a deeper understanding over them.

The formalization and decidability results for eventual consistency are part of a publication [10], while the undecidability results are not published.

1.7 Summary

Overall, this work studies the formalization of consistency criteria for concurrent and distributed libraries, as well as the problem of their verification. We give a general proof of the equivalence between observational refinement and linearizability (Chapter 3). We show that linearizability in general is strictly harder than state reachability, as it is EXPSPACE-complete when the number of threads is bounded, and undecidable otherwise. We prove however that fixing the linearization points makes the problem decidable (Chapter 4).

We then provide two approaches which do not rely on manually specified linearization points. We give in Chapter 5 a bug-detection technique which is based on fundamental properties of the histories of a library and on bounding their interval length. We show that the interval length is a relevant bounding parameter, as most violations can be found with a bound of 0 or 1. Additionally, based on representations using counters, we show how to, on one hand, obtain the decidability of linearizability when the interval length is bounded, and on the other hand, find violations in real implementations, using static analysis as well as testing.

In Chapter 6, we study another approach which doesn't rely on manually specifying the linearization points. It can be applied both to prove or disprove linearizability. We study the effect of limiting the specifications against which we check for linearizability. More precisely, we propose a uniform reduction from linearizability to state reachability for four specifications which are widely used: the queue, stack, register, and mutex. This reduction is done on polynomial time, and doesn't suffer from the exponential blowup that linearizability has in general. We believe this reduction can be carried out for more data structures.

[4] In the context of message-passing systems, we use the notion of sites or nodes instead of the one of threads.

The approach of bounding the interval length can be applied to a wider class of specifications, but uses a heavier instrumentation, using counters and arithmetic formulas.

In the context of message-passing system, we define a formal framework applicable to a wide class of libraries, and used it to define several consistency criteria such as eventual consistency, RYW consistency, FIFO consistency, and causal consistency. We give the first decidability result for eventual consistency (when the number of sites is bounded), as well as undecidability results for all the other criteria mentioned above, even when there are only two sites and the specification is sequential and given as a regular language.

Chapter 2

Preliminaries

2.1 Introduction

We start by fixing some general notations in Section 2.2. We then give separately in Sections 2.3 and 2.4 our formalisms for libraries, and clients calling libraries. Informally, *libraries* are composed of *methods* (usually implementing a well-defined abstract object), and *clients* of the library can call the methods in an arbitrary manner. We then formally define the *composition* between a client and a library. Intuitively, clients and libraries only communicate through the call's to and return's from methods.

Initially, we explore the shared memory setting, where the different instances of the methods of the library communicate by reading and writing to a global memory. This covers the case where libraries are used on a single machine, when there is no need of decentralization. We will present a different setting, where communication is done by message-passing, in Chapter 7.

In this context, the *specification* of a library is another library (generally more abstract, such as the Atomic Stack) and *satisfying a specification* means *observationally refining* it (see Section 2.5). More precisely, a library $\mathcal{L}_1$ *refines* $\mathcal{L}_2$ if any behavior of any client program $\mathcal{C}$ using $\mathcal{L}_1$ can also be obtained were the client $\mathcal{C}$ using $\mathcal{L}_2$ instead. As a consequence, safety properties proven for clients of $\mathcal{L}_2$ are transferred to clients of $\mathcal{L}_1$.

Finally, we give the definition of *linearizability*, which, as we will show in Chapter 3, is a characterization of observational refinement. Intuitively, a library $\mathcal{L}_1$ is *linearizable* with respect to $\mathcal{L}_2$ if for every execution of $\mathcal{L}_1$, there is a corresponding execution in $\mathcal{L}_2$, 1) containing the same operations 2) preserves the timing constraints (i.e. an operation ending before another one) but may have less operations which overlap in time.

2.2 Notations

2.2.1 Sets, Sequences and Functions

For a set A, $\mathcal{P}(A)$ denotes the set of all subsets of A. Let Σ be a set representing an *alphabet*. The set of all finite sequences over Σ is denoted Σ^*, while the set of all infinite sequences is denoted Σ^ω. Also, $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. The empty sequence is denoted ϵ, and the set of non-empty sequences over Σ is denoted Σ^+. Given $u \in \Sigma^*$ a finite sequence, and $v \in \Sigma^\infty$ a possibly infinite sequence, we define $u \cdot v$ as their concatenation. A set of sequences S is called *prefix-closed* if the prefix of any sequence in S is also in S, i.e. for all $u \cdot u' \in S$, we have $u \in S$.

Given two sets A and B, a function $f : A \to B$, and two elements $a \in A$, $b \in B$, we denote by $f[a \leftarrow b]$ the function from A to B which maps a to b, and every other element $a' \in A$ to $f(a')$. Given a subset $A' \subseteq A$, we denote by $f(A')$ the *image* of A', that is $f(A') = \{b \in B \mid \exists a' \in A'.\ f(a') = b\}$.

2.2.2 Posets

A *partially ordered set (poset)* is a pair $(O, <)$, where $<$ is a transitive and irreflexive relation. For $o_1, o_2 \in O$, we denote by $o_1 \leq o_2$ the fact that $o_1 = o_2$ or $o_1 < o_2$. A *path* of length n is a sequence of elements x_0,x_1,..., $x_n \in O$ such that for all $0 < i \leq n-1$, $x_i < x_{i+1}$.

The downward closure of $O' \subseteq O$ with respect to $<$, denoted by $\downarrow^< O'$, is the set of all elements in O smaller or equal than an element in O', i.e. $\downarrow^< O' = \{o \in O \mid \exists o' \in O'.\ o \leq o'\}$. Similarly, the upward closure of $O' \subseteq O$ with respect to $<$, denoted by $\uparrow^< O'$, is the set of all elements in $o \in O$ for which there exists $o' \in O'$ with $o' \leq o$. The superscript may be omitted when the partial order $<$ is clear from the context. We say that O' is downward closed, resp., upward closed, with respect to $<$ if and only if $O' = \downarrow O'$, resp., $O' = \uparrow O'$.

The poset $(O_1, <_1)$ is called a *prefix* of $(O_2, <_2)$, denoted by $(O_1, <_1) \leq (O_2, <_2)$, if and only if $O_1 \subseteq O_2$, $<_1$ is the intersection of $<_2$ and $O_1 \times O_1$, and O_1 is downward closed with respect to $<_2$. We say that a poset $(O, <)$ is *prefix-founded* if and only if for all finite sets $O' \subseteq O$, $\downarrow O'$ is finite.

2.2.3 Labeled Posets

A Σ *labeled poset* is a triple $(O, <, \ell)$, where $(O, <)$ is a poset and $\ell : O \to \Sigma$ is a function that labels each element of O with a symbol in Σ. The set of all Σ labeled posets is denoted by PoSet_Σ.

A labeled poset $\rho_1 = (O_1, <_1, \ell_1)$ is called a prefix of $\rho_2 = (O_2, <_2, \ell_2)$ if and only if the poset $(O_1, <_2)$ is a prefix of $(O_2, <_2)$ and $\ell_1(o) = \ell_2(o)$, for all $y \in O_1$. We also say that ρ_2 is a *completion* of ρ_1. This is denoted by $\rho_1 \preceq \rho_2$. Moreover, if $\ell_2(O_2 \setminus O_1) \subseteq \Sigma'$ where $\Sigma' \subseteq \Sigma$, we say that ρ_2 is a Σ'-completion of ρ_1.

Two labeled posets $(O_1, <_1, \ell_1)$ and $(O_2, <_2, \ell_2)$ are isomorphic if and only if there exists a bijection $h : O_1 \to O_2$ such that for all $x, y \in O_1$, $x <_1 y$ if and only if $h(x) <_2 h(y)$ and $\ell_1(x) = \ell_2(h(x))$. A set of labeled posets $\mathcal{A}$ is called

isomorphism-closed if and only if any labeled poset isomorphic to some labeled poset in $\mathcal{A}$ belongs also to $\mathcal{A}$.

2.2.4 Labeled Transition Systems

A *labeled transition system* (LTS) is a tuple (Q, I, F, Δ) where Q is the set of states, $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states, and $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation, for some set of *labels* Σ. A *run* in the LTS is a sequence of transitions $q_0 \xrightarrow{a_1} q_1 \cdots \xrightarrow{a_l} q_l$, where $q_0 \in I$, $q_l \in F$, and for all $0 \leq i < l$, $(q_i, a_{i+1}, q_{i+1}) \in \Delta$. When there is a run to some state $q \in Q$, we say that q is *reachable*.

When the final states are not specified, and the LTS is given as a tuple (Q, I, Δ), we assume that all states are final. LTSs are sometimes also given with a unique initial state and/or a unique final state, and written (Q, q_0, q_f, Δ).

Non-deterministic finite automata (NFA) are a particular form of LTSs where the transitions are labeled by a finite alphabet Σ and the set of states is finite. *Deterministic finite automata* (DFA) are NFA with determinism constraints, meaning there is a unique initial state, and for all $q \in Q$, $a \in \Sigma$, there exists a unique q' such that $(q, a, q') \in \Delta$.

2.3 Libraries

2.3.1 Syntax

Each *library* is composed of *methods*, which, as we will see later, can be called in an arbitrary way. The different instances of the methods communicate through a set of shared variables $\mathcal{X}$, with $\mathbb{G}$ being the domain of each variable. We represent the code of each method using an LTS, a model general enough to represent the semantics of usual programming languages.

Methods take a parameter and give a return value. In our formalism, each initial state of the LTS corresponds to a possible parameter for the method, while each final state corresponds to a possible value that the method can return.

The LTS representing the code of a method has three kinds of transitions. The first one writes a particular value $v \in \mathbb{G}$ to a particular shared variable $x \in \mathcal{X}$. The second one is a compare-and-swap (`cas`), which updates (atomically) a variable $\mathcal{X}$ to a particular value v' only if it is equal to a given value v. The third one is a `read`, and guesses the current value $v \in \mathbb{G}$ of a particular variable $x \in \mathcal{X}$. This guessing might be considered unusual, but it is not a restriction. In particular, it can be understood as, first reading a variable, and then branching into an infinite loop if the value is not the one expected, or equivalently having an `assume` statement to constrain the value of the read variable. Formally, a *library* $\mathcal{L}$ is a tuple $(\mathcal{X}, \mathbb{G}, K)$, where for each $m \in \mathbb{M}$, $K(m)$ is an LTS whose set of labels is $(\{\texttt{read}, \texttt{write}\} \times \mathcal{X} \times \mathbb{G}) \cup (\{\texttt{cas}\} \times \mathcal{X} \times \mathbb{G} \times \mathbb{G})$.

For complexity purposes, we define the size of the library as being the sum of the number of states in each $K(m)$ for $m \in \mathbb{M}$, plus $|\mathbb{G}|$ (it can be infinite).

When the size is finite, we say that the library is *finite-state*. We assume that the number of variables in $\mathcal{X}$ is fixed, to avoid an exponential blowup in the size of the domain of the overall shared memory, and focus on the complexity which comes from the concurrency of the multiple threads calling the library.

Example 1. *As an illustration of the definition, we define a library* $\mathcal{L}_{AS} = (\mathcal{X}, \mathbb{G}, K)$ *for an Atomic Stack. Let* $\mathbb{D}$ *be the domain of symbols which can be pushed on the stack. The domain of the unique shared variable* $x \in \mathcal{X}$ *is defined as* $\mathbb{G} = \mathbb{D}^*$*, and represents the content of the stack.*

The Atomic Stack has two methods, a Push, used to add an element $d \in \mathbb{D}$ *(given as an argument) on top of the stack, and a method Pop, which takes no argument, but removes and returns the top of the stack.*

Formally, $K(Push)$ *is an LTS* (Q, I, F, Δ) *where* $Q = I \cup F$*,* $I = \mathbb{D}$ *is the set of possible initial states, representing the possible arguments of Push, and* $F = \{\top\}$ *a final state representing the only possible return value. For each possible sequence* $v \in \mathbb{G}$*, and for each initial state* $d \in I$*, there is a transition* $(d, (\texttt{cas}, x, v, d \cdot v), \top)$ *to the final state* $\top$*.*

The LTS (Q, I, F, Δ) *for* $K(Pop)$ *is defined similarly. The set of initial states is* $I = \{\top\}$*, corresponding to the fact that Pop doesn't take an argument. The set of final states is* $F = \mathbb{D} \uplus \{Empty\}$*, where the final state Empty denotes the fact that the stack was found to be empty. The set of states is* $Q = I \cup F$*. Moreover, for each possible* $d \in \mathbb{D}$*, and each possible sequence* $v \in \mathbb{D}^*$*, there is a transition* $(\top, (\texttt{cas}, x, d \cdot v, v), d)$ *to the final state* d*, which effectively removes* d *from the top of the stack, and returns it. Finally, there is a transition* $(\top, (\texttt{read}, x, \epsilon), Empty)$ *which returns Empty when the stack is empty.*

2.3.2 Semantics

We now define the set $[\![\mathcal{L}]\!]$ of *concurrent executions* (executions for short) generated by a library $\mathcal{L} = (\mathcal{X}, \mathbb{G}, K)$. Let $\mathbb{T}$ be an infinite set of *thread identifiers*. Intuitively, each concurrent *thread* $t \in \mathbb{T}$ can call any method with any argument (i.e. in any initial state), and the executions of each thread interleave, and influence each other by reading or writing to the shared variables $\mathcal{X}$.

We define $Q_{\mathbb{M}}$ to be the (disjoint) union of all sets of states of $K(m)$ for $m \in \mathbb{M}$. Formally, we define the semantics of $\mathcal{L}$ using an LTS L. The states of this LTS are called *configurations*, and transitions between configurations are called *steps*. Formally, a *configuration* is a pair $\gamma = (\nu, \mu)$ where $\nu : \mathcal{X} \to \mathbb{G}$ is the current valuation of the shared variables and μ is a map from $\mathbb{T}$ to $Q_{\mathbb{M}} \uplus \{\top\}$ specifying the method state each threads is in. The symbol $\top$ is used for *idle* threads, which are not calling any method at the moment.

In the initial configuration, all threads are idle, and each variable is set to $v_0 \in \mathbb{G}$ for some initial value v_0. All configurations are considered final, meaning that the execution of a library can end at any time. A *step* from a configuration $\gamma = (\nu, \mu)$ to $\gamma' = (\nu', \mu')$ can be:

- An idle thread calling a method m with $K(m) = (Q, \Delta, I, F)$, denoted by

$\gamma \xrightarrow{\texttt{call}_t\ m(q_0)} \gamma'$, with $\mu(t) = \top$, $\mu' = \mu[t \leftarrow q_0]$, $\nu' = \nu$, and $q_0 \in I$. The label $\texttt{call}_t\ m(q_0)$ is a *call action*.

- A thread t returning from a method m with $K(m) = (Q, \Delta, I, F)$, and returning to being idle, denoted by $\gamma \xrightarrow{\texttt{ret}_t\ q_f} \gamma'$, with $\mu(t) = q_f \in F$, $\mu' = \mu[i \leftarrow \top]$, and $\nu' = \nu$. The label $\texttt{ret}_t\ q_f$ is a *return action*.
- A thread t doing a read in a method m with $K(m) = (Q, \Delta, I, F)$, denoted by $\gamma \rightarrow \gamma'$ (no label) with $\mu(t) = q$, $\mu' = \mu[t \leftarrow q']$, $(q, (\texttt{read}, x, \nu(x)), q') \in \Delta$, and $\nu' = \nu$. Note that the value read must correspond to the value $\nu(x)$ in the shared memory. This is an *internal action*.
- A thread t doing a write in a method m with $K(m) = (Q, \Delta, I, F)$, denoted by $\gamma \rightarrow \gamma'$ (no label) with $\mu(t) = q$, $\mu' = \mu[t \leftarrow q']$, $(q, (\texttt{write}, x, v'), q') \in \Delta$, and $\nu' = \nu[x \leftarrow v']$. This is an *internal action*.
- A thread t doing a write in a method m with $K(m) = (Q, \Delta, I, F)$, denoted by $\gamma \rightarrow \gamma'$ (no label) with $\mu(t) = q$, $\mu' = \mu[t \leftarrow q']$, $(q, (\texttt{cas}, x, \nu(x), v'), q') \in \Delta$, and $\nu' = \nu[x \leftarrow v']$. This is an *internal action*.

A (concurrent) *execution* $e \in (\mathsf{Call} \cup \mathsf{Return})^*$ of $\mathcal{L}$ is the sequence of call and return actions labeling the steps of a run of the LTS L, where Call and Return are respectively the sets of all call and return actions. The set of all executions of $\mathcal{L}$ is denoted by $[\![\mathcal{L}]\!]$. The set of executions using at most $k \in \mathbb{N}$ distinct thread identifiers is denoted by $[\![\mathcal{L}]\!]^k$. When clear from context, we will abuse notations and denote these two sets of executions respectively by $\mathcal{L}$ and $\mathcal{L}^k$.

Given an action in $a \in \mathsf{Call} \cup \mathsf{Return}$, we denote by $\mathsf{thread}(a) \in \mathbb{T}$ its thread identifier. Moreover, in an execution e, given a call action c and a return action r, we say that c and r are matching if $\mathsf{thread}(c) = \mathsf{thread}(r)$ and all actions between c and r belong to other threads. We also say that the return action r *corresponds* to c.

Example 2. *An execution of $\mathcal{L}_{AS}$ is*

$$\texttt{call}_{t_1}\ Push(d_1) \cdot \texttt{call}_{t_2}\ Push(d_2) \cdot \texttt{ret}_{t_2}\ \top \cdot \texttt{ret}_{t_1}\ \top \cdot \texttt{call}_{t_1}\ Pop(\top) \cdot \texttt{ret}_{t_1}\ d_2.$$

The return action $\texttt{ret}_{t_1}\ \top$ corresponds to the call action $\texttt{call}_{t_1}\ Push(d_1)$.

2.3.3 Closure Properties

Libraries dictate the execution of methods between their call and return points. Accordingly, a library cannot prevent a method from being called, though it can decide not to return. Furthermore, any library action performed in the interval between call and return points can also be performed should the call have been made earlier, and/or the return made later.

As a result, libraries satisfy a number of closure properties, which we describe below. We will make use of them in several proofs presented later in this thesis, most notably in the proof of equivalence between linearizability and observational refinement.

Lemma 1. *The set of executions of a library $\mathcal{L}$ satisfies the following closure properties: (c denotes a call action, r denotes a return action, a denotes any action, and e, e' denote executions)*

1. *It is closed under prefix: $e \cdot e' \in \mathcal{L}$ implies $e \in \mathcal{L}$.*
2. *The threads identifiers do not matter for the execution: for every execution $e \in \mathcal{L}$, $t \in \mathbb{T}$ and for every pair of matching call and return actions $c = \mathtt{call}_t\ m(q_0)$ and $r = \mathtt{ret}_t\ q_f$, if replacing c and r respectively by $\mathtt{call}_{t'}\ m(q_0)$ and $\mathtt{ret}_{t'}\ q_f$ for some t' gives an execution e', then $e' \in \mathcal{L}$. (The same can be said for call actions with no matching return.) This property is called* thread-independence.
3. *Clients can call library methods at any point in time: $e \cdot e' \in \mathcal{L}$ implies $e \cdot c \cdot e' \in \mathcal{L}$ if $e \cdot c \cdot e'$ is an execution.*
4. *Calls can be made earlier, without changing the execution: $e \cdot a \cdot c \cdot e' \in \mathcal{L}$ implies $e \cdot c \cdot a \cdot e' \in \mathcal{L}$ if $\mathsf{thread}(a) \neq \mathsf{thread}(c)$.*
5. *Returns can be made later, without changing the execution: $e \cdot r \cdot a \cdot e' \in \mathcal{L}$ implies $e \cdot a \cdot r \cdot e' \in \mathcal{L}$ if $\mathsf{thread}(a) \neq \mathsf{thread}(r)$.*

Proof. (1) When defining the semantics $[\![\mathcal{L}]\!]$ of a library, we consider that all configurations are final and as a result an execution can be stopped at any time.

(2) Let $e \in \mathcal{L}$, $t \in \mathbb{T}$, and c, r a matching pair of call and return actions performed by thread t. Assume that changing the thread identifier of c and r to t' gives an execution e'.

Since e' is indeed an execution, this means that, in e, thread t' must be idle from before c to after r. Thus, in the run corresponding to e, we can change the thread identifier of the call action c, the return action r, as well as all the other internal actions in-between done by t, to t'. This proves $e' \in \mathcal{L}$.

(3) By the semantics of a library, we can always add a call action by thread $t \in \mathbb{T}$ in an execution, as long as t is not used afterwards. This is because a call action doesn't affect the valuation of the shared memory.

(4) For the same reason, we can move a call action c to the left of an action a by another thread, as c has no visible effect to the other threads.

(5) Similarly, return actions have no effect to the shared memory, and we can apply the same reasoning. □

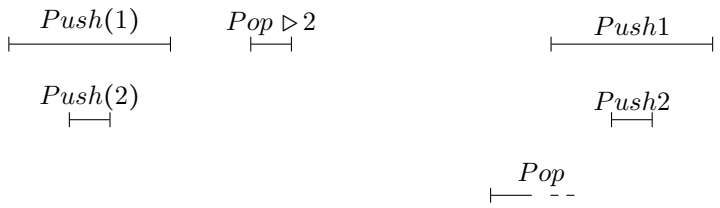
Example 3. *If a library contains the execution (see Fig 2.1a)*

$$\mathtt{call}_{t_1}\ Push(1) \cdot \mathtt{call}_{t_2}\ Push(2) \cdot \mathtt{ret}_{t_2}\ \top \cdot \mathtt{ret}_{t_1}\ \top \cdot \mathtt{call}_{t_1}\ Pop(\top) \cdot \mathtt{ret}_{t_1}\ 2,$$

then it must also contain the execution (see Fig 2.1b)

$$\mathtt{call}_{t_3}\ Pop(\top) \cdot \mathtt{call}_{t_1}\ Push(1) \cdot \mathtt{call}_{t_2}\ Push(2) \cdot \mathtt{ret}_{t_2}\ \top \cdot \mathtt{ret}_{t_1}\ \top.$$

Indeed, by thread independence, we can assume that the Pop is done by another thread t_3. Then, we can use closure property 4 to move its call to the left, and finally, we can take the prefix with closure property 1 to get rid of the final return action.



(a) Execution with 2 threads.

(b) Execution with 3 threads.

Figure 2.1: Time goes from left to right. A horizontal line represents the time interval between matching call and return actions. Operations on the same level are done by the same thread. If a library contains the execution represented on the left, then it must also contain the execution represented on the right. The notation $Pop \triangleright 2$ means that the method Pop returned value 2.

2.4 Clients

2.4.1 Syntax

So far, in order to define the semantics of a library $[\![\mathcal{L}]\!]$, we considered that it was called by a *most general client*, which calls arbitrary methods of the library, with arbitrary arguments, and with arbitrarily many threads. However, *clients* of a library may call the library in a restricted way, and we give in this section our formalism to define specific clients. A *client* is composed of threads calling methods of the library. The different threads of the client communicate through a finite set of shared variables $\mathcal{Y}$. The shared variables of the client are disjoint from the shared variables $\mathcal{X}$ of the library. Let Obs be a (possibly infinite) set of *observable actions*. Intuitively, these correspond to `print` statements.

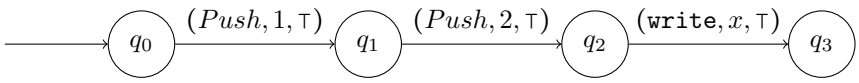
Let $\mathbb{D}$ be a set representing the domain of argument and return values of all methods, i.e. the union of their initial and final states. A *client* $\mathcal{C}$ is a tuple $(\mathcal{Y}, \mathbb{G}, K)$, where $\mathcal{Y}$ is the set of shared variables, $\mathbb{G}$ is their domain, and K is a mapping from $\mathbb{T}$ to an LTS (Q, q_0, Δ) whose transitions are labeled by $(\{\texttt{read}, \texttt{write}\} \times \mathcal{Y} \times \mathbb{G}) \cup (\mathbb{M} \times \mathbb{D} \times \mathbb{D}) \cup \mathsf{Obs}$. We denote by Client the set of all clients.

Intuitively, the `read` and `write` are client actions which operate on the shared variables of the client (distinct from the shared variables of the library), and which can be observed by other threads of the client.

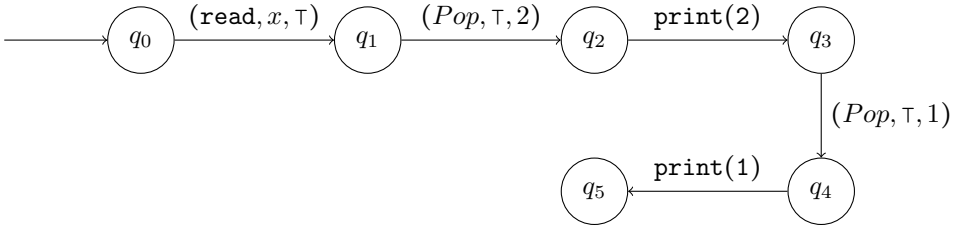
The second kind of transitions, labeled by $\mathbb{M} \times \mathbb{D} \times \mathbb{D}$, correspond to calling a method of the library with a particular argument, and expecting a particular value in return (similarly to how `read` operates). The third kind of transitions, labeled by Obs, correspond to the previously mentioned observable actions.

Example 4. *We give a client $\mathcal{C} = (\mathcal{Y}, \mathbb{G}, K)$ for the library $\mathcal{L}_{AS}$. This client is composed of two threads, i.e. K maps every thread identifier, except $t_1, t_2 \in \mathbb{T}$, to a trivial LTS with no transition.*

Thread t_1 will push two elements on the stack, set the shared variable $x \in \mathcal{Y}$



(a) LTS of t_1.



(b) LTS of t_2.

to $\top$ (its initial value is $\bot$). Thread t_2 expects to read value $\top$ from x, and then pop two elements, and prints them using the observable actions $\mathsf{Obs} = \{\texttt{print(1)}, \texttt{print(2)}\}$. *The LTSs for t_1 and t_2 are given respectively in Figures 2.2a and 2.2b.*

2.4.2 Semantics

We now define the set $[\![\mathcal{C}]\!]$ of *(concurrent) client executions* generated by a client $\mathcal{C}$. Let $Q_{\mathbb{M}}$ be the (disjoint) union of all sets of states of $K(t)$ for $t \in \mathbb{T}$.

Every time a thread makes a call to a method, its goes into an intermediary state, waiting for the method to return. The semantics assumes that the library called by the client can return any value for any method call. In a sense we assume a *most general library*, dually to how we assumed a *most general client* when defining the semantics of libraries.

We give an LTS C to define the semantics of $\mathcal{C}$. Formally, a *configuration* of C is a pair $\gamma = (\nu, \mu)$ where $\nu : \mathcal{Y} \to \mathbb{G}$ is the current valuation of the shared variables and μ is a map from $\mathbb{T}$ to $Q_{\mathbb{M}} \uplus (\{q_{to} \mid q \in Q_{\mathbb{M}}\} \times \mathbb{D})$. A state in $(\{q_{to} \mid q \in Q_{\mathbb{M}}\} \times \mathbb{D})$ is said to be *intermediary*. When a thread is in an intermediary state $(q_{to}, d) \in \{q_{to} \mid q \in Q_{\mathbb{M}}\} \times \mathbb{D}$, it means that it is currently waiting for a method to return value d, and that it will be going to state q when the method returns.

Advancing from a configuration $\gamma = (\nu, \mu)$ to $\gamma' = (\nu', \mu')$ can be done with:

- A thread t doing a read in the LTS (Q, q_0, Δ), denoted by $\gamma \to \gamma'$ (no label) with $\mu(t) = q$, $\mu' = \mu[t \leftarrow q']$, $(q, (\texttt{read}, y, \nu(y)), q') \in \Delta$, and $\nu' = \nu$.
- A thread t doing a write in the LTS (Q, q_0, Δ), denoted by $\gamma \to \gamma'$ (no label) with $\mu(t) = q$, $\mu' = \mu[t \leftarrow q']$, $(q, (\texttt{write}, y, v'), q') \in \Delta$, and $\nu' = \nu[y \leftarrow v']$.

- A thread t doing an observable action in the LTS (Q, q_0, Δ), denoted by $\gamma \xrightarrow{a} \gamma'$ with $\mu(t) = q$, $\mu' = \mu[t \leftarrow q']$, $(q, a, q') \in \Delta$, and $\nu' = \nu$.
- A thread t calling a method in the LTS (Q, q_0, Δ), denoted by $\gamma \xrightarrow{\texttt{call}_t\ m(d_1)} \gamma'$ with $\mu(t) = q$, $\mu' = \mu[t \leftarrow (q'_{to}, d_2)]$, $(q, (m, d_1, d_2), q') \in \Delta$, and $\nu' = \nu$. This step makes the client go into the intermediary state (q'_{to}, d_2), waiting for method m to return value $d_2 \in \mathbb{D}$.
- A thread t getting a return from method in the LTS (Q, q_0, Δ), denoted by $\gamma \xrightarrow{\texttt{ret}_t\ d_2} \gamma'$ with $\mu(t) = (q'_{to}, d_2)$, $\mu' = \mu[t \leftarrow q']$, and $\nu' = \nu$. This step corresponds to a method returning.

A *client execution* e of $\mathcal{C}$ is the sequence of call, return, and observable actions of a run of the semantics LTS C. The set of all client executions of $\mathcal{C}$ is denoted by $[\![\mathcal{C}]\!]$. When clear from context, we will abuse notations and denote this set $\mathcal{C}$.

2.4.3 Composing Libraries and Clients

Given a library $\mathcal{L}$ and a client $\mathcal{C}$, the composition $\mathcal{C} \times \mathcal{L}$ is defined as the set of client executions $e \in \mathcal{C}$ whose projection over $\mathsf{Call} \cup \mathsf{Return}$ belongs to $\mathcal{L}$. It can also be understood as the synchronized product of $[\![\mathcal{L}]\!]$ and $[\![\mathcal{C}]\!]$ over the common alphabet $\mathsf{Call} \cup \mathsf{Return}$. As we define composition of clients with libraries only using call and return actions $\mathsf{Call} \cup \mathsf{Return}$, our formalization supposes that the different threads of the client communicate with libraries only through method calls and returns, and not, e.g., through shared memory. Fig 2.3 gives a visual representation of a client calling a library.

2.5 Correctness Criteria

2.5.1 Observational Refinement

A library $\mathcal{L}_1$ refining another library $\mathcal{L}_2$ means that for any client $\mathcal{C}$, any sequence of observable actions which can be obtained with $\mathcal{L}_1$ can also be obtained with $\mathcal{L}_2$.

Definition 1. *The library* $\mathcal{L}_1$ observationally refines $\mathcal{L}_2$*, written* $\mathcal{L}_1 \trianglelefteq \mathcal{L}_2$*, iff*

$$\forall \mathcal{C} \in \mathsf{Client}.\ (\mathcal{C} \times \mathcal{L}_1)_{|\mathsf{Obs}} \subseteq (\mathcal{C} \times \mathcal{L}_2)_{|\mathsf{Obs}}$$

2.5.2 Histories and Linearizability

Let $\mathbb{M}$ be a set of methods and let $\mathbb{D}$ be the domain of their possible argument and return values. A *history* h is a $\mathbb{M} \times \mathbb{D} \times (\mathbb{D} \uplus \perp)$ labeled poset $(O, <, \ell)$. The symbol $\perp$ is used for operations with no return actions. We call O the set of *operations* of h, and $<$ is called the *happens-before* relation. We say that an operation o is pending if its label $\ell(o)$ contains the symbol $\perp$, and completed

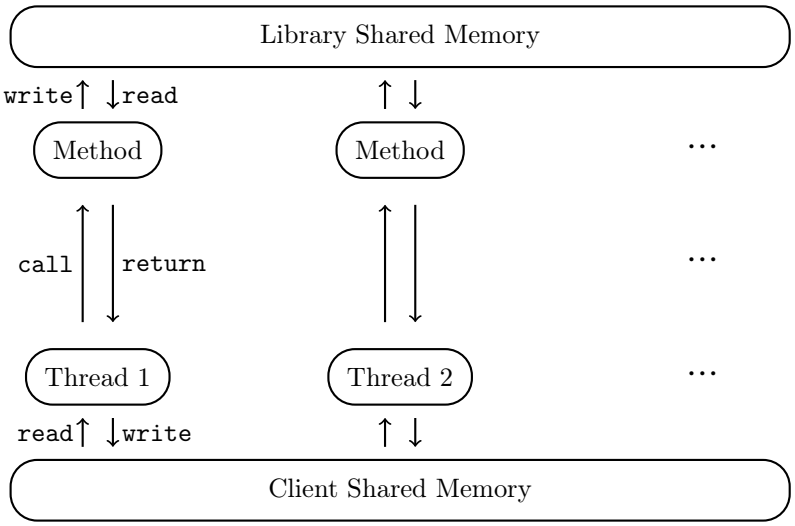


Figure 2.3: Representation of threads of a client calling methods a library. The shared memory of the library is disjoint from the shared memory of the client.

otherwise. For $m \in \mathbb{M}$, $d_1, d_2 \in \mathbb{D}$, the label (m, d_1, d_2) is said to be a *completion* of $(m, d_1, \perp)$.

Given two histories $h_1 = (O_1, <_1, \ell_1)$ and $h_2 = (O_2, <_2, \ell_2)$, we denote by $h_1 \sqsubseteq h_2$ the fact that there exists a partial map f from O_1 to O_2 satisfying the following conditions:

- f is a bijection from its domain to O_2,
- every completed $o_1 \in O_1$ is mapped to $o_2 = f(o_1) \in O_2$ with $\ell_1(o_1) = \ell_2(o_2)$,
- if $o_2 = f(o_1)$ is defined for a pending operation $o_1 \in O_1$, then $\ell_2(o_2) = \ell_1(o_1)$, or $\ell_2(o_2)$ is a completion of $\ell_1(o_1)$,
- for o_1, o_1' in the domain of f, if $o_1 <_1 o_1'$, then $f(o_1) <_2 f(o_1')$.

We say that h_1 is *linearizable with respect to* h_2, or that h_1 is *weaker* than h_2. We extend this notation to sets of histories H_1 and H_2 as follows. H_1 is *linearizable* with respect to H_2, also denoted $H_1 \sqsubseteq H_2$, if $\forall h_1 \in H_1.\ \exists h_2 \in H_2.\ h_1 \sqsubseteq h_2$.

We simplify reasoning on executions by abstracting them into histories. Given an execution e, for each matching pair of call and return actions, we associate an *operation* o, as well as for each call action which no corresponding return action.

The *history* $H(e)$ *of an execution* e is defined as $(O, <, l)$ where:

- O is the set of operations of e,
- $o_1 < o_2$ iff the return action of o_1 is before the call action of o_2 in e,

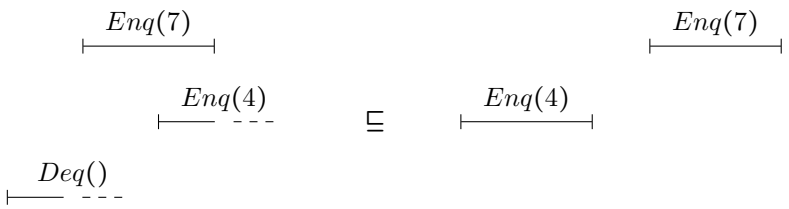


Figure 2.4: The execution on the left is linearizable with respect to the execution on the right.

- an operation o whose call and return actions are respectively $\texttt{call}_\texttt{t}\ m(d_1)$ and $\texttt{ret}_\texttt{t}\ d_2$ is labeled by (m, d_1, d_2),
- an operation o whose call action is $\texttt{call}_\texttt{t}\ m(d_1)$ and which has no return action is labeled by $(m, d_1, \perp)$,

Linearizability extends to executions by taking their corresponding histories.

Example 5. *The history of the execution*

$$\texttt{call}_{\texttt{t}_1}\ Enq(7) \cdot \texttt{call}_{\texttt{t}_2}\ Enq(4) \cdot \texttt{ret}_{\texttt{t}_1} \cdot \texttt{ret}_{\texttt{t}_2}$$

is $h = (\{o_1, o_2\}, <, l)$ *with* $l(o_1) = Enq(7)$, $l(o_2) = Enq(4)$, *and with* $<$ *being the empty order relation, since* o_1 *and* o_2 overlap.

Example 6. *The execution*

$$\texttt{call}_{\texttt{t}_3}\ Deq() \cdot \texttt{call}_{\texttt{t}_1}\ Enq(7) \cdot \texttt{call}_{\texttt{t}_2}\ Enq(4) \cdot \texttt{ret}_{\texttt{t}_1}$$

is linearizable with respect to

$$\texttt{call}_{\texttt{t}_2}\ Enq(4) \cdot \texttt{ret}_{\texttt{t}_2} \cdot \texttt{call}_{\texttt{t}_1}\ Enq(7) \cdot \texttt{ret}_{\texttt{t}_1}.$$

The pending call action $\texttt{call}_{\texttt{t}_2}\ Enq(4)$ *is completed with a return action* $\texttt{ret}_{\texttt{t}_2}$. *The pending call action* $\texttt{call}_{\texttt{t}_3}\ Deq()$ *is removed. And the operation from* t_2 *is ordered before the operation from* t_1, *which is allowed, as they overlap in the original history. See Fig 2.4 for a visual representation.*

Since histories arise from executions, their happens-before relations are *interval orders*. Those are posets which satisfy the following property: for o_1, o_2, o_3, o_4, if $o_1 < o_2$ and $o_3 < o_4$ then either $o_1 < o_4$, or $o_3 < o_2$. Intuitively, this comes from the fact that the happens-before relation is defined as the precedence relation of the time intervals of the operations of an execution, and the precedence relation of any set of intervals is always an interval order.

Lemma 2. *The history* $H(e) = (O, <, f)$ *of an execution* e *forms an interval order* $(O, <)$.

Proof. Suppose $o_1 < o_3$ and $o_2 < o_4$ in $H(e)$. Let r_1 and r_2 be the return actions of o_1 and o_2, and let c_3 and c_4 be the call actions of o_3 and o_4. Since r_1 is before c_3, r_2 is before c_4 in e, and because the actions of an execution are totally ordered, then either r_1 is before c_4 in e, in which case $o_1 < o_4$, or r_2 is before c_3, in which case $o_2 < o_3$. □

In the rest of this work, we consider that all histories $(O, <, \ell)$ have the additional property of $(O, <)$ being an interval order.

Chapter 3

Linearizability and Observational Refinement

3.1 Introduction

In Section 3.2, we give a theorem describing the guarantees that clients get when programming on top of linearizable data structures. Formally, we show that linearizability is a characterization of observational refinement (this extends a theorem of Filipovic et al. [21] for executions containing pending operations).

We also prove two additional characterizations. We prove that a library $\mathcal{L}_1$ is linearizable with respect to $\mathcal{L}_2$ if and only if the executions of $\mathcal{L}_1$ are a subset of the executions of $\mathcal{L}_2$, and also if and ony if the histories of $\mathcal{L}_1$ are a subset of the histories of $\mathcal{L}_2$. The proofs of these characterizations rely on the closure properties of Lemma 1. They give us a new perspective on observational refinement, which we will use for our subsequent results.

Our definition of linearizability applies for a library with respect to another library, while the original definition of linearizability [31] used sequential specifications. We show in Section 3.3 how the two definitions relate, and how we can express the original definition in our context. Intuitively, a library is linearizable with respect to a sequential specification $\mathcal{S}$ if and only if it is linearizable with respect to a library extending $\mathcal{S}$ to the concurrent setting, for instance by using global locks, or atomic compare-and-swap instructions.

Finally, we discuss in Section 3.4 another definition of linearizability which can be found in the literature [21]. We call this definition *strong linearizability*, as it is more strict than ours. The difference comes from a subtlety in the definition for pending operations. Because of this, strong linearizability is not suitable for characterizing observational refinement. However, when the specification is sequential, we show that the two definitions coincide. As most specifications used in practice are sequential, the difference is mostly of theoretical interest.

3.2 Characterizations of Observational Refinement

Our goal in this section is to prove the following characterizations:

Theorem 1 (Characterizations of Observational Refinement)**.** *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two libraries. The following statements are equivalent:*

1. $\mathcal{L}_1 \trianglelefteq \mathcal{L}_2$ *(observational refinement)*
2. $H(\mathcal{L}_1) \sqsubseteq H(\mathcal{L}_2)$ *(linearizability)*
3. $H(\mathcal{L}_1) \subseteq H(\mathcal{L}_2)$ *(history-set inclusion)*
4. $\mathcal{L}_1 \subseteq \mathcal{L}_2$ *(execution-set inclusion)*

We first give two important lemmas regarding the set of histories of a library. The first property says that the history-set of a library $\mathcal{L}$ is *closed under weakening*, meaning that if $\mathcal{L}$ can produce some history h, it can also produce all histories $h' \sqsubseteq h$ which are linearizable (weaker) than h. This lemma is key to prove that linearizability implies history-set inclusion.

The second property states that the history abstraction of an execution is suitable to describe the set of executions of a library. As long as a library can produce an execution with history h, it can produce *all* executions whose history is h. This lemma is used to prove that history-set inclusion implies execution-set inclusion.

Lemma 3 (Closure under Weakening)**.** *If $h_2 \in H(\mathcal{L})$ and $h_1 \sqsubseteq h_2$ then $h_1 \in H(\mathcal{L})$.*

Proof. The histories h_1 and h_2 are given respectively by $(O_1, <_1, \ell_1)$ and $(O_2, <_2, \ell_2)$. By definition of linearizability, we know there exists a partial map f from O_1 to O_2, satisfying the following conditions:

- f is a bijection from its domain to O_2,
- every completed $o_1 \in O_1$ is mapped to $o_2 = f(o_1) \in O_2$ with $\ell_1(o_1) = \ell_2(o_2)$,
- a pending operation $o_1 \in O_1$ doesn't have to be mapped, but if it is, it must be mapped to $o_2 = f(o_1) \in O_2$ with $\ell_2(o_2) = \ell_1(o_1)$, or with $\ell_2(o_2)$ being a completion of $\ell_1(o_1)$,
- for o_1, o_1' in the domain of f, if $o_1 <_1 o_1'$, then $f(o_1) <_2 f(o_1')$.

In the following, we'll identify operations from O_1 and operations from O_2 which are connected by the partial bijection f. Let $e_2 \in \mathcal{L}$ be an execution whose history is h_2. We're going to apply the closure properties of Lemma 1 in order to obtain an execution $e_1 \in \mathcal{L}$ whose history is h_1. By thread independence, assume without loss of generality that all the operations from e_2 use a distinct thread.

First, we need to add to e_2 call actions corresponding to the pending operations which are in h_1 and not in h_2 (i.e. the pending operations which are

not in the domain of f). The closure properties allow us to add pending call actions anywhere in the execution, so we obtain an execution e_2' which now has the same operations as e_1.

Second, some operations which are pending in h_1 might be completed in h_2. We remove the return actions corresponding to these operations in e_2', and obtain a new execution e_2'' which has the same operations as e_1, with the same labels. This is also allowed by the closure properties of Lemma 1.

Finally, the only difference between e_1 and e_2'' is the happens-before relation between the operations. By definition of f, we know that the happens-before relation of e_1 is weaker than the one of e_2''. Since these two relations are interval orders, we can, in e_2'', move call actions to the left, and return actions to the right, in order to obtain the same happens-before relation.

More precisely, in e_2'', the call action of an operation o will be moved to the left, and set right after the last return action of an operation $o' <_1 o$ (or at the beginning if no such operation exists). Similarly, the return action of an operation o will be moved to right, and set right before the first call action of an operation $o <_1 o'$ (or at the end if no such operation exists). Moving the call and return actions respectively to the left and to the right is also allowed by the closure properties of Lemma 1. Thus, we obtain an execution $e_1 \in \mathcal{L}$ whose history is exactly h_1. □

Lemma 4 (History Abstraction)**.** *For two executions e_1, e_2 and a library $\mathcal{L}$, if $H(e_1) = H(e_2)$ and $e_1 \in \mathcal{L}$, then $e_2 \in \mathcal{L}$.*

Proof. Since e_1 and e_2 have the same histories, they must have the same operations, and the same happens-before relation between them. Thus, if e_1 and e_2 differ, they can only differ for operations not related by the happens-before relation, i.e. which are overlapping. This entails that they must agree on the order of call actions with respect to return actions, but that they can disagree on the order of several call actions, or on the order of several return actions.

We show that, even in the case where e_1 and e_2 differ, we can apply the closure properties of Lemma 1 to $\mathcal{L}$ to prove that $\mathcal{L}$ also contains e_2. Assume that e_1 and e_2 do differ, and consider the first position where they disagree on the order of call actions. For instance, after some common prefix, we might have a call action c in e_1, and a different call action c' in e_2. Using the closure properties, we can move c' to the left in e_1, and put it before c' to obtain an execution $e_1' \in \mathcal{L}$ which agrees with e_2 on the order of more call actions than e_1 did. For return actions, we make a similar reasoning, but using the last position instead of the first, and moving return actions to the right instead of the left. □

Having proved the two lemmas, we can now prove the theorem showing the equivalence between observational refinement, linearizability, history-set inclusion, and execution-set inclusion.

Theorem 1 (Characterizations of Observational Refinement)**.** *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two libraries. The following statements are equivalent:*

1. $\mathcal{L}_1 \preccurlyeq \mathcal{L}_2$ *(observational refinement)*
2. $H(\mathcal{L}_1) \sqsubseteq H(\mathcal{L}_2)$ *(linearizability)*
3. $H(\mathcal{L}_1) \subseteq H(\mathcal{L}_2)$ *(history-set inclusion)*
4. $\mathcal{L}_1 \subseteq \mathcal{L}_2$ *(execution-set inclusion)*

Proof. $(1 \Rightarrow 2)$ We introduce a client $\mathcal{C}$ whose set of observable actions is a copy of the call and return actions $\mathsf{Call} \cup \mathsf{Return}$.

$$\mathsf{Obs} = \{\texttt{print}(c) \mid c \in \mathsf{Call}\} \cup \{\texttt{print}(r) \mid r \in \mathsf{Return}\}$$

The client $\mathcal{C}$ we build is inspired from the proof of this same implication in Filipovic et al. [21], but which applied only to completed executions. It is similar to a most general client, in the sense that it calls all the methods of the library arbitrarily, but it also records, using the observable actions, every method that it called, as well as every method which returned from the library.

Formally, $\mathcal{C}$ is the tuple $(\varnothing, \varnothing, K)$ (no shared variables), where for each $t \in \mathbb{T}$, $K(t)$ is a non-deterministic LTS, where each choice is of the form:

$$\texttt{print}(\texttt{call}_\texttt{t}\ m(d_1)) \cdot (m, d_1, d_2) \cdot \texttt{print}(\texttt{ret}_\texttt{t}\ d_2).$$

Note that this LTS can be infinite, if $\mathbb{M}$ or $\mathbb{D}$ is infinite. The client $\mathcal{C}$ can use an unbounded number of threads, but each thread calls only one method. Each client execution of one such LTS can be interpreted as follows:

- $\texttt{print}(c)$ means that the client is planning on doing the call action c,
- $\texttt{print}(c) \cdot c$ means that the thread did the call,
- $\texttt{print}(c) \cdot c \cdot r$ means that the library returned from the call,
- $\texttt{print}(c) \cdot c \cdot r \cdot \texttt{print}(r)$ means the client knows the library returned.

We will use observational refinement for $\mathcal{C}$, i.e. that $(\mathcal{C} \times \mathcal{L}_1)_{|\mathsf{Obs}} \subseteq (\mathcal{C} \times \mathcal{L}_2)_{|\mathsf{Obs}}$, in order to prove linearizability, i.e. $H(\mathcal{L}_1) \sqsubseteq H(\mathcal{L}_2)$. Let $h_1 \in H(\mathcal{L}_1)$, and let $e_1 \in \mathcal{L}_1$ such that $H(e_1) = h_1$. We build a client execution $e_1' \in \mathcal{C}$ from e_1 as follows. Before every call action c in e, we insert the corresponding $\texttt{print}(c)$ observable action, and after every return action r in e, we insert the corresponding $\texttt{print}(r)$.

The obtained client execution e_1' thus belongs to $\mathcal{C} \times \mathcal{L}_1$. By using observational refinement, we know there exists e_2' in $\mathcal{C} \times \mathcal{L}_2$ whose sequence of observable actions is the same as e_1', i.e. $e_{1|\mathsf{Obs}}' = e_{2|\mathsf{Obs}}'$.

Let $e_2 = e_{2|\mathsf{Call} \cup \mathsf{Return}}'$ and $h_2 = H(e_2)$ be its history. Our goal is to show that $h_1 \sqsubseteq h_2$, which would end the proof, as $h_2 \in H(\mathcal{L}_2)$. Even though e_1' and e_2' have the same sequence of observable actions, it could happen that a pending call c made in e_1' using $\texttt{print}(c) \cdot c$ does not appear in e_2', even though $\texttt{print}(c)$ does. Indeed, if the client $\mathcal{C}$ didn't observe the return action, it cannot know whether

the method was actually called. However this is allowed by the definition of linearizability, as we're allowed to remove pending operations from h_1.

Similarly, it is possible that a pending call c made in e'_1 using $\texttt{print}(c)\cdot c$ was completed into $\texttt{print}(c)\cdot c\cdot r$ in e'_2, even though $\mathcal{C}$ didn't observe the return action yet using $\texttt{print}(r)$. Again, this is allowed by the definition of linearizability, as we're allowed to complete pending operations of h_1.

Finally, if an operation o_a happens-before an operation o_b in h_1, then the observable action $\texttt{print}(r_a)$ corresponding to the return action of o_a will appear before the observable action $\texttt{print}(c_b)$ corresponding to the call action of o_b in e'_1. Since they must appear in that same order in e'_2, this implies that o_a happens-before o_b in h_2 as well.

$(2 \Rightarrow 3)$ Let $h_1 \in H(\mathcal{L}_1)$. By assumption, there exists $h_2 \in H(\mathcal{L}_2)$ such that $h_1 \sqsubseteq h_2$. Since the histories of a library are closed under weakening (Lemma 3), $h_1 \in H(\mathcal{L}_2)$.

$(3 \Rightarrow 4)$ Let $e_1 \in \mathcal{L}_1$ and let $h_1 = H(e_1)$. By assumption, since $h_1 \in H(\mathcal{L}_1)$, we have $h_1 \in H(\mathcal{L}_2)$. Thus, there exists an execution $e_2 \in \mathcal{L}_2$ whose history h_2 equals h_1. By applying Lemma 4 on library $\mathcal{L}_2$, we conclude that $e_1 \in \mathcal{L}_2$.

$(4 \Rightarrow 1)$ Let $\mathcal{C} \in \mathsf{Client}$. Since $\mathcal{L}_1 \subseteq \mathcal{L}_2$, we can deduce that $(\mathcal{C} \times \mathcal{L}_1)_{|\mathsf{Obs}}$ is a subset of $(\mathcal{C} \times \mathcal{L}_2)_{|\mathsf{Obs}}$. □

We can prove a similar theorem when the number of threads is bounded.

Theorem 2 (Characterizations of Observational Refinement)**.** *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two libraries and k be an integer representing the number of threads. The following statements are equivalent:*

1. ${\mathcal{L}_1}^k \trianglelefteq {\mathcal{L}_2}^k$ *(observational refinement)*
2. $H({\mathcal{L}_1}^k) \sqsubseteq H({\mathcal{L}_2}^k)$ *(linearizability)*
3. $H({\mathcal{L}_1}^k) \subseteq H({\mathcal{L}_2}^k)$ *(history-set inclusion)*
4. ${\mathcal{L}_1}^k \subseteq {\mathcal{L}_2}^k$ *(execution-set inclusion)*

3.3 Atomic Specifications

Originally, linearizability was defined for concurrent libraries with respect to sequential specifications. We show in this section how the original definition reduces to ours, which only uses concurrent libraries.

We define *sequential executions* as a particular case of (concurrent) executions, where each call action $\texttt{call}_\texttt{t}\ m(d_1)$ is immediately followed by a matching return action $\texttt{ret}_\texttt{t}\ d_2$. Since linearizability is not defined using thread identifiers, we can abstract away the thread identifiers and represent a sequential execution as a sequence from $(\mathbb{M} \times \mathbb{D} \times \mathbb{D})^*$. A *sequential specification* $\mathcal{S} \subseteq (\mathbb{M} \times \mathbb{D} \times \mathbb{D})^*$ is a set of sequential executions.

Example 7. *The LTS (Q, q_0, Δ) where:*

- $Q = \mathbb{D}$,
- $q_0 = d_0 \in \mathbb{D}$ *for some initial value* d_0,
- $\Delta = \{(d, (\texttt{write}, d', \top), d') \mid d, d' \in \mathbb{D}\} \cup \{(d, (\texttt{read}, \top, d), d) \mid d \in \mathbb{D}\}$.

represents the sequential executions of a register over the domain $\mathbb{D}$*. The method* `write` *takes as argument the value to write into the register, and returns* $\top$*, while the method* `read` *takes no argument (default argument* $\top$*) and returns the current value of the register.*

The following lemma states that we can go from a sequential specification represented as an LTS to one represented as a library in polynomial time.

Lemma 5. *Let* $\mathcal{S}$ *be a prefix-closed sequential specification represented by a (possibly infinite) LTS. There exists a library* $\mathcal{L}_\mathcal{S}$*, whose size is polynomial in the number of states of the LTS (when finite), representing the executions which are linearizable with respect to* $\mathcal{S}$*.*

Proof. Let (Q, q_0, Δ) be the LTS representing $\mathcal{S}$. All states can be considered final because of the prefix-closure.

We define the library $\mathcal{L}_\mathcal{S} = (\mathcal{X}, \mathbb{G}, K)$ as follows. We will use one shared variable, called x, i.e. $\mathcal{X} = \{x\}$. Its domain is the set of states of the LTS, i.e. $\mathbb{G} = Q$. Then, for each $m \in \mathbb{M}$, we define an LTS $K(m) = (Q_m, I_m, F_m, \Delta_m)$ where

- $I_m = \{\texttt{arg}[d] \mid d \in \mathbb{D}\}$ is a copy of $\mathbb{D}$,
- $F_m = \{\texttt{rv}[d] \mid d \in \mathbb{D}\}$ is another copy of $\mathbb{D}$,
- $Q_m = I_m \cup F_m$,
- $\Delta_m = \{(\texttt{arg}[d_1], \texttt{cas}(x, q, q'), \texttt{rv}[d_2]) \mid (q, (m, d_1, d_2), q') \in \Delta\}$.

Formally, the executions produced by $\mathcal{L}_\mathcal{S}$ will have call actions of the form $\texttt{call}_\texttt{t}\ m(\texttt{arg}[d])$. On the other hand, executions which are linearizable with respect to $\mathcal{S}$ will have call actions of the form $\texttt{call}_\texttt{t}\ m(d)$. We identify these two kinds of call actions and consider them equal. We make a similar identification $\texttt{ret}_\texttt{t}\ \texttt{rv}[d] \equiv \texttt{ret}_\texttt{t}\ d$ We can then prove:

$$\forall e.\ e \in \mathcal{L}_\mathcal{S} \iff e \sqsubseteq \mathcal{S}$$

($\Rightarrow$) Let $e \in \mathcal{L}_\mathcal{S}$. We remove from e pending operations which stayed in the initial state. We complete the pending operations which ended in some final state $\texttt{rv}[d]$ with the return value $\texttt{rv}[d]$. Finally, we order the operations in the order in which they executed their `cas` actions. The sequence must belong to $\mathcal{S}$, as the `cas` actions exactly mimic the transition relation Δ of $\mathcal{S}$.

($\Leftarrow$) Let e such that $e \sqsubseteq \mathcal{S}$. We know there exists a sequential execution $w \in \mathcal{S}$ such that $e \sqsubseteq w$. When considering w as an execution with call and return actions, we can see that $w \in \mathcal{L}_\mathcal{S}$, as the `cas` actions reproduce the transitions Δ of $\mathcal{S}$. Moreover, by Lemma 3, $\mathcal{L}_\mathcal{S}$ is closed under weakening, and we have $e \in \mathcal{L}_\mathcal{S}$. □

Using this lemma, we can show that linearizability with respect to a sequential specification $\mathcal{S}$ is equivalent to linearizability with respect to the concurrent library $\mathcal{L}_{\mathcal{S}}$.

Corollary 1. *For all libraries $\mathcal{L}$:*

$$\mathcal{L} \sqsubseteq \mathcal{S} \iff \mathcal{L} \sqsubseteq \mathcal{L}_{\mathcal{S}} \text{ and}$$
$$\forall k \in \mathbb{N}.\ \mathcal{L}^k \sqsubseteq \mathcal{S} \iff \mathcal{L}^k \sqsubseteq {\mathcal{L}_{\mathcal{S}}}^k$$

Proof. The first equivalence can be proven as follows:

$$\begin{aligned} & \mathcal{L} \sqsubseteq \mathcal{S} \\ \iff & \mathcal{L} \subseteq \mathcal{L}_{\mathcal{S}} \text{ (execution-set inclusion, Lemma 5)} \\ \iff & \mathcal{L} \sqsubseteq \mathcal{L}_{\mathcal{S}} \text{ (linearizability, Theorem 1)} \end{aligned}$$

The second equivalence can be proven similarly using Theorem 2. □

3.4 Linearizability Definition Variant

Some publications give a different definition for linearizability. To avoid confusion, and since this definition is in general stronger than ours, we will call it here *strong linearizability*. The difference with our definition of linearizability is subtle, and appears in the third item of the definition below, about pending operations. Strong linearizability states that a pending operation must be either discarded, or completed, while our definition allows a pending operation of h_1 to stay pending in h_2.

Given two histories $h_1 = (O_1, <_1, \ell_1)$ and $h_2 = (O_2, <_2, \ell_2)$, we say that h_1 is *strongly linearizable* with respect to h_2 if there exists a partial map f from O_1 to O_2 satisfying the following conditions:

- f is a bijection from its domain to O_2,
- every completed $o_1 \in O_1$ is mapped to $o_2 = f(o_1) \in O_2$ with $\ell_1(o_1) = \ell_2(o_2)$,
- if $o_2 = f(o_1)$ is defined for a pending operation $o_1 \in O_1$, then $\ell_2(o_2)$ is a completion of $\ell_1(o_1)$,
- for o_1, o_1' in the domain of f, if $o_1 <_1 o_1'$, then $f(o_1) <_2 f(o_1')$.

When checking linearizability against a sequential specification $\mathcal{S}$, the two definitions coincide, as there are no pending operations in the executions of $\mathcal{S}$. Using one definition or the other doesn't affect our complexity results, as almost all of them are given for sequential specifications.

The example below shows that strong linearizability is indeed stronger than our definition of linearizability, and it is not suitable as a precise characterization of observational refinement when $\mathcal{L}_2$ is a concurrent library.

Example 8. *Let $\mathcal{L}$ be the library containing $e = \texttt{call}_{t_1}\ m() \cdot \texttt{call}_{t_2}\ m'() \cdot \texttt{ret}_{t_1}$ and closed by the properties given in Lemma 1. Although $\mathcal{L}$ observationally refines itself (and is linearizable with respect to itself), $\mathcal{L}$ is not strongly linearizable with respect to itself, since e could only be strongly linearizable with respect to $\mathcal{L}$ if $\mathcal{L}$ were to contain an execution of one of these forms:*

- $\texttt{call}_{t_1}\ m() \cdot \texttt{ret}_{t_1}$*, or*
- $\texttt{call}_{t_1}\ m() \cdot \texttt{ret}_{t_1} \cdot \texttt{call}_{t_2}\ m'() \cdot \texttt{ret}_{t_2}$ *(or any interleaving with these two operations),*

but $\mathcal{L}$ contains none of them.

3.5 Summary

This chapter gives a fundamental understanding of observational refinement, by giving several properties which characterize it. In particular, we proved that linearizability is equivalent to observational refinement in general, extending the result of Filipovic et al. [21]. Moreover, we also proved that observational refinement is equivalent to the execution-set inclusion of the libraries, as well as the history-set inclusion of the libraries.

We also showed how to express any atomic specification as a concurrent library. This proves formally the unsurprising result that the definition of linearizability over concurrent libraries is more general that the definition of linearizability over atomic specifications. More importantly, this result also shows the link between proving linearizability with respect to an atomic specification and observational refinement. It shows how to effectively construct a library $\mathcal{L}_S$ such that a library $\mathcal{L}$ refines $\mathcal{L}_S$ if and only if $\mathcal{L}$ is linearizable with respect to the atomic specification S.

These results give us a new perspective on observational refinement, and open the door to new kind of analyses for its verification. In particular, we will show in Chapter 5 how to leverage the equivalence with the history-set inclusion in bug-detection algorithms.

Chapter 4

Complexity Results for Linearizability

4.1 Introduction

In this chapter, we study the complexity of verifying, given two libraries $\mathcal{L}_1$ and $\mathcal{L}_2$, whether $\mathcal{L}_1$ observationally refines $\mathcal{L}_2$, or equivalently, whether $\mathcal{L}_1$ is linearizable with respect to $\mathcal{L}_2$. In order to focus on the complexity of verifying linearizability, and to avoid the difficulties which come from verifying Turing-complete programs in general, we will study the verification problem for finite-state libraries. We show that linearizability is strictly harder than the more common problem of *state reachability*. State reachability asks, given a library and a state of the LTS of a method, whether there is an execution which can reach that state. This proves that, in general, there is no polynomial time reduction from linearizability to state reachability.

In particular, we show in Section 4.4 that when the number of threads is unbounded, checking linearizability of a library with respect to a regular sequential specification is not decidable – while state reachability is known to be reducible to coverability in vector addition systems with states (VASS). Our proof relies on a reduction from a reachability problem in counter machines with zero tests, and goes as follows. First, we show how to simulate the executions of a counter machine without zero tests with the executions of a library. Then, we use linearizability with respect to a regular specification to add zero tests, and make sure that the number of increments and decrements between consecutive zero tests is equal. Intuitively, we prove that there is a run in the counter machine with zero tests if and only if there is an execution of the library which is not linearizable. This means that non-linearizability will encode the fact that there are the same number of increments and decrements. To this end, increments, decrements, and zero tests of the counter machines are encoded as inc, dec and zero operations in the library, and we use a regular specification $\mathcal{S}$ which is roughly composed of all the words which contain $\mathsf{zero} \cdot (\mathsf{inc} \cdot \mathsf{dec})^* (\mathsf{inc} + \mathsf{dec})^+ \cdot \mathsf{zero}$ as a

subword[1].

As we make sure that all increment and decrement operations between two consecutive zero tests overlap, an execution which is not linearizable with respect to $\mathcal{S}$ must necessarily have an equal number of increment and decrement operation between *every* two consecutive zero tests, thus simulating a valid run of the counter machine. Indeed, if it had more increments than decrements (or vice versa) between two zero tests, we could order each decrement after a corresponding increment, and all additional increments at the end, and build a linearization containing a subword of the form: $\mathsf{zero} \cdot (\mathsf{inc} \cdot \mathsf{dec})^* (\mathsf{inc} + \mathsf{dec})^+ \cdot \mathsf{zero}$.

When the number of threads is bounded, we show that there is an exponential blowup, as we prove that linearizability is EXPSPACE-complete, while state reachability is known to be PSPACE-complete. We show the membership in EXPSPACE even in the case where the specification is given as a library (see Section 4.6), and show the EXPSPACE-hardness even in the case where the specification is given as a DFA (see Section 4.5). The membership in EXPSPACE follows from the fact that linearizability is equivalent to the execution-set inclusion of the libraries, and the execution-set of a finite-state library can be represented by an automaton whose size is exponential in the size of the library (because of the state-explosion due to the presence of several threads). Inclusion between regular automata is known to be PSPACE, so we obtain overall a complexity of EXPSPACE.

To prove the EXPSPACE-hardness, we introduce a new problem on regular languages, called letter insertion, which can be reduced in polynomial time to linearizability. We then show that letter insertion is EXPSPACE-hard. Our proof is similar to the proofs of EXPSPACE-hardness for the problems of inclusion of extended regular expressions with intersection operator, or interleaving operator, given in Hunt [33], Fürer [23] and Mayer and Stockmeyer [40]. They all use a similar encoding of runs of Turing machines as words, and then use the problem at hand – letter insertion in this case – to recognize erroneous runs.

Showing that linearizability is equivalent to letter insertion is interesting on its own, for two reasons. First, it permits to connect linearizability to other fields, and as a result gain intuition on the problem. Moreover, letter insertion helps pinpoint the source of difficulty of linearizability, as letter insertion can be reduced to a restricted form of linearizability, where the executions contain a sequence of operations (executed by one thread), overlapping with several other operations (the overlapping operations all overlap with one another as well).

These results close the complexity gap left open by Alur et al. [3], who proved that for a bounded number of threads, linearizability with respect to a regular sequential specification is in EXPSPACE, and that it is PSPACE-hard (hardness follows from the fact that state reachability can be reduced to linearizability).

We conclude from that study that in practice, there is no efficient way of reducing linearizability to more standard verification problems such as state reachability. Due to the intricacy of checking linearizability, Herlihy and Wing [31] have introduced a stricter criterion, where the so-called "linearization points"

[1] For two sets of sequences S_1 and S_2, $S_1 + S_2$ denotes their union.

– the points at which operations' effects become instantaneously visible – are specified manually. We will call it here *static linearizability*. This criterion is vastly used, but no complexity results were known. We show in Section 4.7 that static linearizability has the same complexity as state reachability, meaning that it is decidable and EXPSPACE-complete when the number of threads is unbounded, and PSPACE-complete when the number of threads is bounded.

Remark 1. *In this chapter (except Sections 4.3 and 4.7), we consider that methods do not take arguments and do not return particular return values. We do this because the hardness proofs do not rely on the existence of arguments or return values, and because it simplifies the presentation. A sequential specification is now a subset of* $\mathbb{M}^*$*, instead of being a subset of* $(\mathbb{M} \times \mathbb{D} \times \mathbb{D})^*$.

4.2 Counter Machines and VASS

We start by giving some notations concerning counter machines and vector addition system with states (VASS). We will use them in several construction and proofs, as well as in several reductions.

4.2.1 Syntax

A *n-counter machine* $\mathcal{V} = (Q, q_0, \Delta)$ is an LTS whose set of states Q is finite and whose transitions are labeled by $\{\mathsf{inc}_i \mid 1 \leq i \leq n\} \cup \{\mathsf{dec}_i \mid 1 \leq i \leq n\} \cup \{\mathsf{zero}_i \mid 1 \leq i \leq n\}$. These transitions are respectively called *increment*, *decrement*, and *zero-test* transitions. A *VASS* is a counter machine without zero-test transitions.

4.2.2 Semantics

The semantics of a counter machine are given using another LTS whose configurations are pairs $c = (q, \vec{v})$ where $q \in Q$, and $\vec{v}$ is a vector in $\mathbb{N}^n$, storing the (non-negative) value of each counter. We denote component i of $\vec{v}$ by $\vec{v}(i)$, for $1 \leq i \leq n$. The transition relation $\to$ between configurations is defined as $(q_1, \vec{v}_1) \to (q_2, \vec{v}_2)$ if and only if

- $(q_1, \mathsf{inc}_i, q_2) \in \Delta$ and $\vec{v}_2 = \vec{v}_1[i \leftarrow \vec{v}_1(i) + 1]$, or
- $(q_1, \mathsf{dec}_i, q_2) \in \Delta$ and $\vec{v}_2 = \vec{v}_1[i \leftarrow \vec{v}_1(i) - 1]$, or
- $(q_1, \mathsf{zero}_i, q_2) \in \Delta$, $\vec{v}_1(i) = 0$ and $\vec{v}_2 = \vec{v}_1$.

The initial configuration is $(q_0, \vec{0})$ where $\vec{0}(i) = 0$ for all $1 \leq i \leq n$. The *configuration reachability* problem for counter machines is to determine if a given configuration $(q, \vec{v})$ is reachable from q_0, while the *control-state reachability* problem consists in determining whether, for a given state q, there exists $\vec{v}$ such that $(q, \vec{v})$ is reachable.

4.3 State Reachability

As a reference point to measure the complexity of linearizability, we will use the state reachability problem in libraries. In our setting, it can be stated as follows:

Problem 1 (State Reachability)**.**

- **Input:** *$\mathcal{L}$ a library and $q \in F$ a state in the LTS (Q, I, F, Δ) of a method m in $\mathcal{L}$. Optionally, we add a bound $k \in \mathbb{N}$ on the number of threads*
- **Output:** *Yes if and only if $\mathcal{L}$ has an execution using at most k threads (if there is a bound) where some thread reaches state q.*

The following lemma shows that state reachability can be reduced to linearizability. We will show later that the converse is not true in general.

Lemma 6. *State reachability in a library $\mathcal{L}$ can be reduced to (non-)linearizability of $\mathcal{L}$ with respect to a regular sequential specification in polynomial time.*

Proof. State reachability asks whether a state $q_f \in F$ is reachable in the LTS (Q, I, F, Δ) of a method m. Define $\mathcal{S}$ to be the set of sequences of $(\mathbb{M} \times \mathbb{D} \times \mathbb{D})^*$ which do not contain a symbol (m, q_0, q_f) with $q_0 \in I$. Then, by definition of linearizability, an execution is linearizable with respect to $\mathcal{S}$ if and only if it doesn't contain a (m, q_0, q_f) operation. Worded differently, there exists an execution in $\mathcal{L}$ which is not linearizable with respect to $\mathcal{S}$ if and only if there exists an execution containing a (m, q_0, q_f) operation, meaning that state q_f was reached in the LTS of method m. □

There is a well-known complexity result for state reachability in (concurrent) libraries, stating that it is equivalent to VASS control-state reachability. We sketch the proof, as the construction of a VASS from a library is standard, and we will use it throughout the paper.

Lemma 7 (Murata [43])**.** *Let $\mathcal{L}$ be a finite-state library. State reachability is decidable and has the same complexity as VASS control-state reachability.*

Proof. We sketch the construction of a VASS from a library.

Remember that when defining the runs of $\mathcal{L}$, a configuration is a pair composed of the valuation of the shared variables $\mathcal{X}$, and a mapping gives the state each thread is currently in. The set of states of the VASS $\mathcal{V}$ we construct is the set of all possible valuations of $\mathcal{X}$. (Since we consider the number of variables to be fixed, this set is polynomial in the size of $\mathcal{L}$.)

Moreover, due to thread-independence, instead of having a mapping which gives the state each thread is currently in, we can equivalently count the number of threads which are in a particular state. When the library is finite-state, there are finitely many states in the LTS of each method, and we only need a finite number of counters in $\mathcal{V}$ to represent a configuration.

The VASS $\mathcal{V}$ can then simulate the runs of $\mathcal{L}$ simply by changing state when the shared memory is modified, and decrementing/incrementing counters when a thread moves from a location to another.

If we want to know whether there exists an execution which reaches some state $q \in Q$ in the LTS of a method, we can add a transition is $\mathcal{V}$ which decrement the counter of state q, and goes into a fresh final state. This ensures that the fresh final state is reachable if and only if there is an execution where a thread can go into state q.

Finally, in this construction, bounding the number of threads using $\mathcal{L}$ is equivalent to bounding the sum of the counters. □

As a consequence, we obtain the following complexity results:

Corollary 2. *State reachability is* EXPSPACE*-complete when the number of threads is not bounded, and* PSPACE*-complete when it is.*

Proof. Follows from the complexity of VASS control-state reachability for unbounded counters and bounded counters respectively. □

4.4 Undecidability of Linearizability (unbounded threads)

We show in this section that verifying linearizability of a finite-state library with respect to a regular specification is not decidable when the number of threads is unbounded. This contrasts with the fact that state reachability in concurrent programs is EXPSPACE-complete for an unbounded number of threads.

Theorem 3 (Undecidability of Linearizability)**.** *Let $\mathcal{L}$ be a finite-state library, and $\mathcal{S}$ be a sequential specification represented by a finite automaton. Verifying whether $\mathcal{L} \sqsubseteq \mathcal{S}$ is undecidable.*

Proof. Our goal is to reduce control-state reachability in a counter machine to (non-)linearizability of a finite-state library with respect to a regular specification. Formally, given a counter machine $\mathcal{V} = (Q, q_0, \Delta)$ and a state q_f, we construct a finite-state library $L_\mathcal{V} = (\mathcal{X}, \mathbb{G}, K)$ and a regular specification $S_\mathcal{V}$ such that $L_\mathcal{V}$ is *not* linearizable with respect to $S_\mathcal{V}$ exactly when $\mathcal{V}$ has a run reaching q_f.

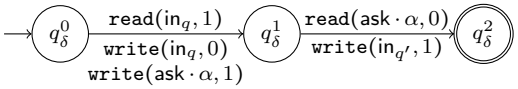
The variables $\mathcal{X}$ of $L_\mathcal{V}$ are boolean variables whose domain is $\mathbb{G} = \{0, 1\}$. In particular, they include a boolean variable in_q for each state $q \in Q$ of the counter machine. Initially, only the variable in_{q_0} corresponding to the initial state q_0 is set to 1. During the executions of $L_\mathcal{V}$, we keep the invariant that exactly one of the in_q variables is set to 1. Intuitively, this represents the current state of the run of $\mathcal{V}$ that we're simulating.

Let n be the number of counters in $\mathcal{V}$. The set $\mathcal{X}$ also includes variables askinc_i, askdec_i, $\mathsf{allowdec}_i$, and $\mathsf{askzero}_i$, for each counter $i \in \{1, \ldots, n\}$. For each transition δ of the counter machine, $L_\mathcal{V}$ contains a method $\mathsf{M}[\delta]$ which

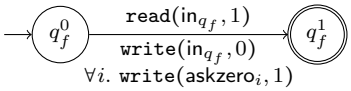
simulates the update of the control state by δ, and for each counter x_i of $\mathcal{V}$, $L_{\mathcal{V}}$ contains the methods MInc_i, MDec_i, and MZero_i, that simulate respectively an increment, a decrement, and a zero-test on counter x_i. Finally, $L_{\mathcal{V}}$ contains a method $\mathsf{M_f}$ which can return only when the value of the boolean variable corresponding to the final state q_f is set to 1. The method automata we will construct all have a single initial state and a single final state. Thus, we do not mention argument or return value in the remainder of the proof.

Formally, the LTSs for the methods are given as follows. For ease of presentation, we label some transitions by multiple read and write instructions. This is syntactic sugar to denote the fact that they must be executed atomically, without interference from the other threads. In practice, this can be implemented using locks implemented using the `cas` instruction, or any mutual exclusion algorithm.

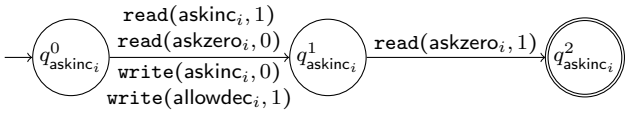
- for each transition $\delta = (q, \alpha, q')$ of $\mathcal{V}$ with $i \in \mathbb{N}^*$ and $\alpha \in \{\mathsf{inc}_i, \mathsf{dec}_i, \mathsf{zero}_i\}$, $\mathsf{M}[\delta]$ is



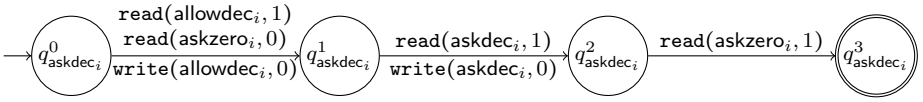
- a method $\mathsf{M_f}$ that can be executed when the simulated run of $\mathcal{V}$ reaches the final state:



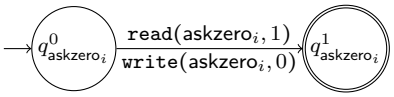
- for each i, a method MInc_i that simulates an increment of the counter i:



- for each i, a method MDec_i that simulates a decrement of the counter i



- for each i, a method MZero_i that simulates a zero-test for the counter i



Initially, all variables are set to 0 except $\mathsf{in}_{q_0} = 1$. The specification $S_{\mathcal{V}}$ is defined such that all the executions of $L_{\mathcal{V}}$ which don't correctly simulate the counter machine or which simulate runs of the counter machines not reaching the final state, are linearizable with respect to $S_{\mathcal{V}}$.

The specification $S_{\mathcal{V}}$ is a regular language that constrains only the order between the methods MInc_i, MDec_i, MZero_i, for any $i \in \{1, \ldots, n\}$, and $\mathsf{M_f}$. Let $\mathbb{M}_i = \{\mathsf{MInc}_i, \mathsf{MDec}_i, \mathsf{MZero}_i, \mathsf{M_f}\}$, for $i \in \{1, \ldots, n\}$. We define $S_{\mathcal{V}}$ as follows. A word w belongs to $S_{\mathcal{V}}$ if there exists $i \in \{1, \ldots, n\}$ such that the projection of w on the alphabet $\mathbb{M}_i$ satisfies one of the following constraints:

- it doesn't contain $\mathsf{M_f}$,
- it contains a prefix of the form

$$(\mathsf{MInc}_i \cdot \mathsf{MDec}_i)^* \cdot (\mathsf{MInc}_i^+ + \mathsf{MDec}_i^+) \cdot \mathsf{MZero}_i$$

- it contains sub-words of the form

$$\mathsf{MZero}_i \cdot (\mathsf{MInc}_i \cdot \mathsf{MDec}_i)^* \cdot (\mathsf{MInc}_i^+ + \mathsf{MDec}_i^+) \cdot \mathsf{MZero}_i.$$

Intuitively $S_{\mathcal{V}}$ recognizes the fact that a zero test was taken while the number of increments was different that the number of decrements. Lemmas 8 and 9 prove the correspondence between $\mathcal{V}$ and $L_{\mathcal{V}}$.

Lemma 8. *If there exists a run ξ of $\mathcal{V}$ that goes through q_f, then there exists an execution e of $L_{\mathcal{V}}$ that is not linearizable with respect to $S_{\mathcal{V}}$.*

Proof. Let $\xi = (q_0, \nu_0), \ldots, (q_s, \nu_s)$, where $q_s = q_f$, be such a run in $\mathcal{V}$. Let $\delta_k = (q_k, \alpha_k, q_{k+1}) \in \Delta$ denote the transition taken between q_k and q_{k+1}, for every $0 \leq k < s$.

We will construct an execution e_ξ of $L_{\mathcal{V}}$ such that for every i, the MZero_i operations do not overlap MInc_i nor MDec_i operations and such that, between every two successive MZero_i operations (and before the first MZero_i operation), all the MInc_i and MDec_i operations overlap.

Formally, for each transition $\delta_k = (q_k, \alpha_k, q_{k+1})$, a thread t_1 calls method $\mathsf{M}[\delta_k]$, and executes the first step, which sets in_{q_k} to 0, and sets $\mathsf{ask} \cdot \alpha_k$ to 1. Then t_1 interrupts, and wait for $\mathsf{ask} \cdot \alpha$ to be set to 0 by another thread.

During the interruption, depending on α_k, the following happens:

- if $\alpha_k = \mathsf{askinc}_i$ then:
  - a new thread calls the method MInc_i and also performs the first step of this method, setting askinc_i back to 0, and setting $\mathsf{allowdec}_i$ to 1, effectively enabling a method MDec_i to execute its first step;
  - a new thread calls the method MDec_i and also performs the first step of this method, going to $q^2_{\mathsf{askdec}_i}$, setting $\mathsf{allowdec}_i$ back to 0. This method MDec_i can then be used for a subsequent decrement transition.

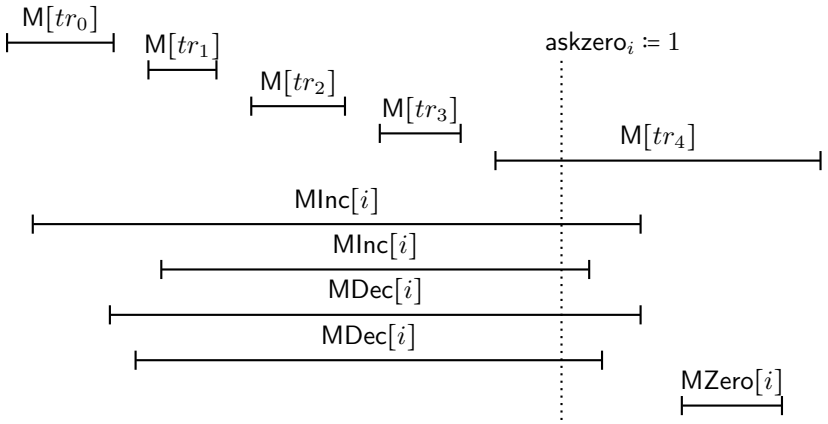


Figure 4.1: An execution of $L_{\mathcal{V}}$, representing a run of $\mathcal{V}$.

- if $\alpha_k = \mathsf{askdec}_i$ then we know there exists a thread executing method MDec_i in $q^2_{\mathsf{askdec}_i}$ because in any prefix of ξ, the number of increments on i are greater than or equal to the number of decrements on i; we take such a thread, and makes a transition from state $q^2_{\mathsf{askdec}_i}$ of MDec_i to the state $q^3_{\mathsf{askdec}_i}$, setting askdec_i back to 0.
- if $\alpha_k = \mathsf{askzero}_i$ then
  - since $\mathsf{askzero}_i$ has been set to 1 by thread t_1, all the MInc_i and MDec_i operations can execute their last step and return. This part ensures that all the MInc_i and MDec_i operations between two successive MZero_i operations overlap
  - a new thread calls MZero_i and performs the first step, setting $\mathsf{askzero}_i$ back to 0

Thread t_1 can then proceed, check that $\mathsf{ask} \cdot \alpha$ has been set to 0, set $\mathsf{in}_{q_{k+1}}$ to 1, and return. After we finished simulating the run, variable in_{q_f} equals 1, and we can finish the execution e_ξ by calling $\mathsf{M_f}$ and returning after the first step.

The execution e_ξ is not linearizable with respect to $S_{\mathcal{V}}$ because for any $i \in \{1, \ldots, n\}$, the projection of e_ξ on call and return actions corresponding to MInc_i, MDec_i, MZero_i, and $\mathsf{M_f}$, denoted e_i, satisfies the following:

- e_i contains a completed $\mathsf{M_f}$ operation,
- before the first MZero_i operation, there is the same number of (completed) MInc_i and MDec_i operations. Consequently, we cannot order them to form a sequence $(\mathsf{MInc}_i \cdot \mathsf{MDec}_i)^* \cdot (\mathsf{MInc}_i^+ + \mathsf{MDec}_i^+) \cdot \mathsf{MZero}_i$,
- similarly, between every two successive MZero_i operations, there is the same number of MInc_i and MDec_i operations. Thus, we cannot order them to form a sequence $\mathsf{MZero}_i \cdot (\mathsf{MInc}_i \cdot \mathsf{MDec}_i)^* \cdot (\mathsf{MInc}_i^+ + \mathsf{MDec}_i^+) \cdot \mathsf{MZero}_i$.

□

Lemma 9. *If there exists an execution e of $L_{\mathcal{V}}$, not linearizable with respect to $S_{\mathcal{V}}$, then there exists a run ξ of $\mathcal{V}$, that goes through q_f.*

Proof. Let e be an execution of $L_{\mathcal{V}}$. Notice that in every configuration of e there exist at most one variable in_q with q a state of $\mathcal{V}$ which is set to 1. This means that during the execution e there is at most one $\mathsf{M}[\delta]$ operation which is in state q_δ^1.

Let $\mathsf{M}[\delta_0]$,...,$\mathsf{M}[\delta_{s-1}]$ be the labels of all the $\mathsf{M}[\delta]$ operations of with are completed, or which are in state q_δ^2 at the end of e. We order them by the order in which they reached state q_δ^2

We will show that the transitions δ_0,...,δ_{s-1} form a run ξ in $\mathcal{V}$. The first thing to note is that if we ignore the value of the counters, δ_0,...,δ_{s-1} does form a path in the underlying graph of $\mathcal{V}$, as the methods $\mathsf{M}[\delta]$ exactly encode the transitions of the underlying graph, using the variables in_q, for $q \in Q$.

Second, we can see the values of the counter can never go negative in ξ, as each $\mathsf{M}[\delta]$ corresponding to a decrement transition requires a MDec_i operation to set askdec_i back to 0, and we know that an operation MDec_i operation can execute only if a previous MInc_i operation enabled it by setting $\mathsf{allowdec}_i$ to 1. Thus, we know that in any prefix of ξ, the number of increment transition is greater or equal than the number of decrement transitions.

Third, we know that, since the methods MInc_i and MDec_i can only start when $\mathsf{askzero}_i$ is 0, and can only end when $\mathsf{askzero}_i$ is 1, all the MInc_i and MDec_i operations between two successive (resp., before the first) MZero_i operations must overlap.

Moreover, since e is not linearizable with respect to $S_{\mathcal{V}}$, the MInc_i and MDec_i operations between two successive (resp., before the first) MZero_i operations cannot be ordered into a sequence of the form $(\mathsf{MInc}_i\mathsf{MDec}_i)^*(\mathsf{MInc}_i^+ + \mathsf{MDec}_i^+)$, meaning that

- there is no MInc_i or MDec_i operation overlapping with a MZero_i operation; if there was such an operation, by the definition of linearizability, it could be moved arbitrarily before or after the MZero_i operation, thus leaving an unequal number of overlapping MInc_i and MDec_i operations between two successive (resp., before the first) MZero_i operations, and allowing us to order them in a sequence of the form $(\mathsf{MInc}_i\mathsf{MDec}_i)^*(\mathsf{MInc}_i^+ + \mathsf{MDec}_i^+)$,
- the number of (overlapping) MInc_i and MDec_i operations between two successive (resp., before the first) MZero_i operations must be the same, otherwise we could also order them in a sequence of the form $(\mathsf{MInc}_i\mathsf{MDec}_i)^*(\mathsf{MInc}_i^+ + \mathsf{MDec}_i^+)$ (contradicting the fact that e is not linearizable with respect to $S_{\mathcal{V}}$.

This shows that every time a zero-test transition is taken in ξ, the value of the corresponding counter was indeed 0. Finally, since e is not linearizable with respect to $S_{\mathcal{V}}$, we know that it contains a completed $\mathsf{M_f}$ operation, and that ξ must go through state q_f. □

This ends the undecidability proof of linearizability. □

4.5 Linearizability is EXPSPACE-hard (bounded threads)

We prove in this section that linearizability is EXPSPACE-hard when the number of threads is bounded. We show in Section 4.5.1 that letter insertion can be reduced in polynomial time to linearizability, and in Section 4.5.2, that letter insertion is EXPSPACE-hard, which is the most technical part of the overall proof. When combined, these results prove that linearizability is EXPSPACE-hard.

An important detail is that, for the reduction from letter insertion to linearizability, we need to use an NFA to represent the sequential specification. We show in fact in Section 4.5.3 that, even when the sequential specification is represented by a DFA, linearizability is still EXPSPACE-hard.

Formally, our goal is to prove the following:

Theorem 4 (Linearizability is EXPSPACE-hard (DFA))**.** *Let $\mathcal{L}$ be a finite-state library, $\mathcal{S}$ be a sequential specification represented by an DFA, and $k \in \mathbb{N}$ be a bound on the number of threads.*

Verifying that $\mathcal{L}^k \sqsubseteq \mathcal{S}$ is EXPSPACE*-hard*[2]*.*

4.5.1 Reduction from Letter Insertion to Linearizability

The letter insertion problem is defined as follows.

Problem 2 (Letter Insertion)**.**

- **Input:** *A set of letters $T = \{a_1, \ldots, a_l\}$. An NFA $\mathcal{A}$ over an alphabet $\Gamma \uplus T$.*[3]
- **Output:** *Yes, iff for all words $w \in \Gamma^*$, there exists a decomposition $w = w_0 \cdots w_l$, and a permutation p of $\{1, \ldots, l\}$, such that $w_0 \cdot a_{p(1)} \cdot w_1 \cdots a_{p(l)} \cdot w_l$ is accepted by $\mathcal{A}$.*

Said differently, for any word of Γ^*, can we insert the letters $\{a_1, \ldots, a_l\}$ (each of them exactly once, in any order, anywhere in the word) to obtain a word accepted by $\mathcal{A}$?

Example 9. *Let $\Gamma = \{b, c\}$, $T = \{a\}$, and $\mathcal{A}$ an automaton representing the regular language $a \cdot (b + c)^*$. Then for any word in Γ^*, we can insert the letter a in order to obtain a word accepted by $\mathcal{A}$ (inserted at the beginning). This is a 'Yes' instance of the letter insertion problem.*

We show that letter insertion can be reduced in polynomial time to linearizability. Intuitively, the letters $T = \{a_1, \ldots, a_l\}$ of letter insertion represent methods which are all overlapping with every other method, and the word w

[2] The size of the input is the size of $\mathcal{L}$, plus the size of $\mathcal{S}$, plus k.

[3] The size of the input is the size of $\mathcal{A}$, to which we add l.

represents methods which are in sequence. letter insertion asks whether we can insert the letters in w in order to obtain a sequence of $\mathcal{A}$ while linearizability asks whether there is a way to order all the letters, while preserving the order of w, to obtain a sequence of $\mathcal{A}$, which is equivalent.

Lemma 10 (Letter Insertion to Linearizability)**.** *Given a letter insertion instance $(T, \mathcal{A})$, we can build, in polynomial time, a finite-state library $\mathcal{L}$, a regular sequential specification $\mathcal{S}_\mathcal{A}$, and an integer $k \in \mathbb{N}$, such that $(T, \mathcal{A})$ is a* yes *instance for letter insertion if and only if $\mathcal{L}^k \sqsubseteq \mathcal{S}_\mathcal{A}$.*

Proof. Let $T = \{a_1, \ldots, a_l\}$ and $\mathcal{A}$ an NFA over some alphabet $T \uplus \Gamma$.

Define k, the number of threads, to be $l + 2$.

We will define a library $\mathcal{L}$ composed of

- methods $m_1, \ldots, m_l$, one for each letter of T,
- methods m_γ, one for each letter of Γ,
- a method m_{Tick}.

and a specification $\mathcal{S}_\mathcal{A}$, such that $(T, \mathcal{A})$ is a valid instance of letter insertion if and only if $\mathcal{L}^k$ is linearizable with respect to $\mathcal{S}_\mathcal{A}$. Similarly to the undecidability proof of Theorem 3, the method automata we will construct all have a single initial state and a single final state and thus we won't mention argument or return value in the remainder of the proof.

The library $\mathcal{L}$ will only use one shared variable, whose domain is $\{\mathsf{Begin}, \mathsf{Run}, \mathsf{End}\}$ with Begin being the initial value. Since there is only one variable, we will omit the variable name in the `read` and `write` transitions.

The methods m_γ are all identical. They just read the value Run from the shared variable (see Fig 4.2).

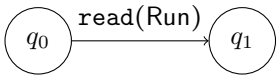


Figure 4.2: Description of m_γ, $\gamma \in \Gamma$

The methods $m_1, \ldots, m_l$ all read Begin, and then read End (see Fig 4.3).

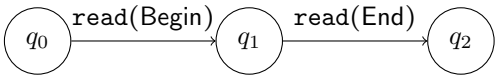


Figure 4.3: Description of $m_1, \ldots, m_l$

The method m_{Tick} writes Run, and then End (see Fig 4.4).

The specification $\mathcal{S}_\mathcal{A}$ is defined as the set of words w over the alphabet $\{m_1, \ldots, m_l\} \cup \{m_{\mathsf{Tick}}\} \cup \{m_\gamma \mid \gamma \in \Gamma\}$ such that one the following condition holds:

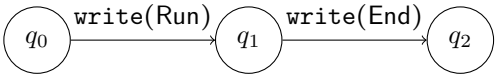


Figure 4.4: Description of m_{Tick}

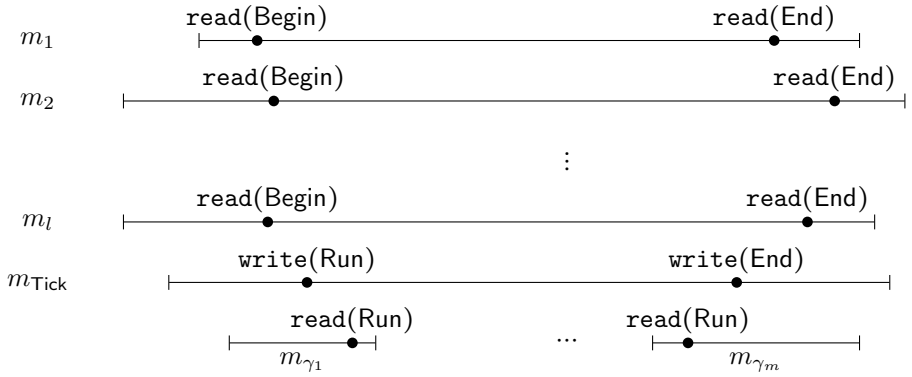


Figure 4.5: Non-linearizable execution corresponding to a word $\gamma_1 \dots \gamma_m$ in which we cannot insert the letters from $T = \{a_1, \dots, a_l\}$ to make it accepted by $\mathcal{A}$. The points represent steps in the automata.

- w contains 0 letter m_{Tick}, or more than 1, or
- for a letter m_i, $i \in \{1, \dots, l\}$, w contains 0 such letter, or more than 1, or
- when projecting over the letters m_γ, $\gamma \in \Gamma$ and m_i, $i \in \{1, \dots, l\}$, w is in $\mathcal{A}_m$, where $\mathcal{A}_m$ is $\mathcal{A}$ where each letter γ is replaced by the letter m_γ, and where each letter a_i is replaced by the letter m_i.

Since $\mathcal{A}$ is an NFA, $\mathcal{S}_\mathcal{A}$ is also an NFA. Moreover, its size is polynomial is the size of $\mathcal{A}$. We can now show the following equivalence:

1. there exists a word w in Γ^*, such that there is no way to insert the letters from T in order to obtain a word accepted by $\mathcal{A}$
2. there exists an execution of $\mathcal{L}$ with k threads which is not linearizable with respect to $\mathcal{S}_\mathcal{A}$

(1) $\implies$ (2). Let $w \in \Gamma^*$ such that there is no way to insert the letters T in order to obtain a word accepted by $\mathcal{A}$. We construct an execution of $\mathcal{L}$ following Fig 4.5, which is indeed a valid execution.

This execution is not linearizable since

- it has exactly one m_{Tick} method, and
- for each $i \in \{1, \dots, l\}$, it has exactly one m_i method, and

- no linearization of this execution can be in $\mathcal{A}_m$, since there is no way to insert the letters T into w to be accepted by $\mathcal{A}$.

Note: The value of the shared variable is initialized to Begin, allowing the methods m_i ($i \in \{1, \ldots, l\}$) to make their first transition. m_{Tick} then sets the value to Run, thus allowing the methods $m_\gamma, \gamma \in \Gamma$ to execute. Finally, m_{Tick} sets the value to End, allowing the methods $m_\gamma, \gamma \in \Gamma$ to make their second transition and return. This tight interaction will enable us to show in the second part of the proof that all non-linearizable executions of this library have this very particular form.

(2) $\implies$ (1). Let e be an execution which is not linearizable with respect to $\mathcal{S}_\mathcal{A}$. We first show that this execution should roughly be of the form shown in Fig 4.5. First, since it is not linearizable with respect to $\mathcal{S}_\mathcal{A}$, it must have at least one completed m_{Tick} operation. It it only had pending m_{Tick} events (or no m_{Tick} events at all), it could be linearized by dropping all the pending calls to m_{Tick}. Moreover, it cannot have more than m_{Tick} operation (completed or pending), as it could also linearized, since $\mathcal{S}_\mathcal{A}$ accepts all words with more than one m_{Tick} letter.

We can show similarly that for each $i \in \{1, \ldots, l\}$, it has exactly one m_i method which is completed (and none pending).

Moreover, the methods m_i ($i \in \{1, \ldots, l\}$ can only start when the value of the shared variable is Begin, and they can only return after reading the value End. Since this value can only be changed (once) by the single m_{Tick} method of our executions, this ensures that the methods m_i ($i \in \{1, \ldots, l\}$) (and m_{Tick} itself) all overlap with one another, and with every other completed method.

This implies that the completed methods $m_\gamma, \gamma \in \Gamma$ can only appear in a single thread t (since $m_1, \ldots, m_l, m_{\mathsf{Tick}}$ already occupy $l+1$ threads among the $l+2$ available). Thus, we define $w \in \Gamma^*$ to be the word corresponding to the completed methods m_γ, $\gamma \in \Gamma$ of the execution in the order in which they appear in thread t.

Since e is not linearizable, we cannot insert m_i ($i \in \{1, \ldots, l\}$) into the completed methods of thread t order to be accepted by $\mathcal{S}_\mathcal{A}$. In particular, this implies that there is no way to insert the letters T in w in order to be accepted by $\mathcal{A}$. □

4.5.2 Letter Insertion is EXPSPACE-hard

We now reduce, in polynomial time, arbitrary exponentially space-bounded Turing machines, to the letter insertion problem, which shows it is EXPSPACE-hard. We first give a few notations.

A *deterministic Turing machine* $\mathcal{M}$ is a tuple (Q, q_0, q_f, Δ) where:

- Q is the set of states,
- $\Delta : (Q \times \{0,1\}) \to (Q \times \{0,1\} \times \{\leftarrow, \rightarrow\})$ is the transition function,
- q_0, q_f are the initial and final states, respectively.

A computation of $\mathcal{M}$ is said to be *accepting* if it ends in q_f.

Problem 3 (Reachability)**.** *Input: A finite word t.*

Output: Yes iff: the computation of $\mathcal{M}$ starting in state q_0, with the tape initialized with t, is accepting.

Lemma 11. *Letter insertion is* EXPSPACE*-hard.*

Note: the sublemmas 12,13,14,15,16 are all part of the proof of Lemma 11.

Proof. We reduce in polynomial time the reachability problem for EXPSPACE Turing machines to the *negation* of letter insertion. This still shows that letter insertion is EXPSPACE-hard, as the EXPSPACE complexity class is closed under complement.

For the rest of the proof, we fix a deterministic Turing machine $\mathcal{M}$ and a polynomial P such that all runs of $\mathcal{M}$ starting with an input of size n use at most $2^{P(n)}$ cells.

Let t be a word of size n. Our goal is to define a set of letters T and an NFA $\mathcal{A}$ over an alphabet $\Gamma \uplus T$, such that the following two statements are equivalent:

- the run of $\mathcal{M}$ starting in state q_0 with the tape initialized with t is accepting (which, by definition of $\mathcal{M}$, uses at most $2^{P(n)}$ cells),
- there exists a word w in Γ^*, such that there is no way to insert (see Problem 2) the letters T in order to obtain a word accepted by $\mathcal{A}$.

More specifically, we will encode runs of our Turing machine as words, and the automaton $\mathcal{A}$, with the additional set of letters T, will be used in order to detect words which:

- don't represent *well-formed sequence of configurations* (defined below),
- or represent a sequence of configurations where the initial configuration is not initialized with t and state q_0, or where the final configuration isn't in state q_f,
- or contain an error in the computation, according to the transition rules of $\mathcal{M}$.

A *configuration* of $\mathcal{M}$ is an ordered sequence $(c_0, \ldots, (q, c_i), \ldots, c_{2^{P(n)}-1})$ representing that the content of the tape is $c_0, \ldots, c_{2^{P(n)}-1} \in \{0, 1\}$, the current control state is $q \in Q$, and the head is on cell i.

We denote by $\mathbf{i}$ the binary representation of $0 \leq i < 2^{P(n)}$ using $P(n)$ digits. Given a configuration, we represent cell i by: "$\mathbf{i} : c_i$;" if the head of $\mathcal{M}$ is not on cell i, and by "$\mathbf{i} : qc_i$;" if the head is on cell i and the current state of $\mathcal{M}$ is q. The configuration given above is represented by the word:

$$\$\mathbf{0} : c_0; \mathbf{1} : c_1; \ldots \mathbf{i} : qc_i; \ldots \mathbf{2^{P(n)} - 1} : c_{2^{P(n)}-1}; \hookleftarrow$$

.

Words which are of this form for some $c_0, \ldots, c_{2^{P(n)}-1} \in \{0,1\}$, $q \in Q$, are called *well-formed configurations*. A sequence of configurations is then encoded as $\triangleright \mathsf{cfg}_1 \ldots \mathsf{cfg}_k \square$ where each cfg_i is a well-formed configuration. A word of this form is called a *well-formed sequence of configurations*. We now fix Γ to be $\{0, 1, \triangleright, \square, \$, \hookleftarrow, ;, :\}$.

Lemma 12. *There exists an NFA $\mathcal{A}_{\mathsf{notWF}}$ of size polynomial in n, which recognizes words which are not well-formed configurations.*

Proof. A word is not a well-formed configuration if and only if one of the following holds:

- it is not of the form $\$((0+1)^{P(n)} : (Q+\epsilon)(0+1);)^* \hookleftarrow$, or
- it has no symbol from Q, or more than one, or
- it doesn't start with $\$\mathbf{0}:$, or
- it doesn't end with $\mathbf{2^{P(n)} - 1} : (Q+\epsilon)(0+1); \hookleftarrow$, or
- it contains a pattern $\mathbf{i} : (Q+\epsilon)(0+1); \mathbf{j} :$ where $j \neq i+1$.

For all violations, we can make an NFA of size polynomial in n recognizing them, and then take their union. The most difficult one is the last, for which there are detailed constructions in Fürer [23] and Mayer and Stockmeyer [40].

We here give a sketch of the construction. Remember that $\mathbf{i}$ and $\mathbf{j}$ are binary representation using $P(n)$ bits. We want an automaton recognizing the fact that $j \neq i+1$. The automaton guesses the least significant bit b ($P(n)$ possible choices) which makes the equality $i+1 = j$ fails, as well as the presence or not of a carry (for the addition $i+1$) at that position. We denote by $\mathbf{i}[b]$ the bit b of $\mathbf{i}$ and likewise for $\mathbf{j}$. Then, the automaton checks that: 1) there is indeed a violation at that position (for instance: no carry, $\mathbf{i}[b] = 0$ and $\mathbf{j}[b] = 1$) and 2) there is carry if and only if all bits less significant that b are set to 1 is $\mathbf{i}$. □

Lemma 13. *There exists an NFA $\mathcal{A}_{\mathsf{NotSeqCfg}}$ of size polynomial in n, which recognizes words which:*

- *are not a well-formed sequence of configurations, or where*
- *the first configuration is not in state q_0, or*
- *the first configuration is not initialized with t, or*
- *the last configuration is not in state q_f.*

Proof. Non-deterministic union between $\mathcal{A}_{\mathsf{notWF}}$ and simple automata recognizing the last three conditions. □

The problem is now in making an NFA which detects violations in the computation with respect to the transition rules of $\mathcal{M}$. Indeed, in our encoding, the length of one configuration is about $2^{P(n)}$, and thus, violations of the transition rules from one configuration to the next are going to be separated by about $2^{P(n)}$ characters in the word. We conclude that we cannot make directly an automaton of polynomial size which recognize such violations.

This is where we use the set of letters T. We are going to define and use it here, in order to detect words which encode a sequence of configurations where there is a computation error, according to the transition rules of $\mathcal{M}$.

The set T, containing $2P(n)$ new letters, is defined as $T = \{p_1, \ldots, p_{P(n)}, m_1, , \ldots, m_{P(n)}\}$.

We want to construct an NFA $\mathcal{A}_{\mathsf{NotDelta}}$, such that, for a word w which is a well-formed sequence of configurations, these statements are equivalent:

- w has a computation error according to the transition rules Δ of $\mathcal{M}$
- we can insert the letters T in w to obtain a word accepted by $\mathcal{A}_{\mathsf{NotDelta}}$.

The idea is to use the letters T in order to identify two places in the word corresponding to the same cell of $\mathcal{M}$, but at two successive configurations of the run.

As an example, say we want to detect a violation of the transition $\Delta(q, 0) = (q', 1, \rightarrow)$, that is, which reads a 0, writes a 1, moves the head to the right, and changes the state from q to q'.

Assume that w contains a sub-word of the following form:

$$\mathbf{i} : q0; \ldots \$ \ldots \mathbf{i} : 1; \mathbf{i} + \mathbf{1} : q''c_{i+1};$$

where q'' is different than q'

The single \$ symbol on the middle of the sub-word ensures that we are checking violations in successive configurations. Here, with the current state being q, the head read 0 on cell i, wrote 1 successfully, and moved to the right. But the state changed to q'' instead of q'. Since we assumed that $\mathcal{M}$ is deterministic, this is indeed a violation of the transition rules.

We now have all the ingredients in order to construct $\mathcal{A}_{\mathsf{NotDelta}}$. It will be built as a non-deterministic choice (or union) of $\mathcal{A}_t$ for all possible transitions $t \in \Delta$ (with Δ seen as a relation).

As an example, we show how to construct the automaton $\mathcal{A}^{(1)}_{((q,0),(q',1,\rightarrow))}$, part of $\mathcal{A}_{\mathsf{NotDelta}}$, and recognizing violations of $\Delta(q, 0) = (q', 1, \rightarrow)$, where the head was indeed moved to right, but the state was changed to some state q'' instead of q', like above. Other violations may be recognized similarly.

$\mathcal{A}^{(1)}_{((q,0),(q',1,\rightarrow))}$ starts by finding a sub-word of the form:

$$(m_1 0 + p_1 1) \ldots (m_{P(n)} 0 + p_{P(n)} 1) : q0; \tag{4.1}$$

meaning the state is q and the head points to a cell containing 0. After that, it reads arbitrarily many symbols, but exactly one \$ symbol, which ensures that

the next letters it reads are from the next configuration. Finally, it looks for a sub-word of the form

$$(p_1 0 + m_1 1) \dots (p_{P(n)} 0 + m_{P(n)} 1) : (0+1); (0+1)^* : q;'' \tag{4.2}$$

for some $q'' \neq q'$.

We can now show the following.

Lemma 14. *For a well-formed sequence of configurations w, these two statements are equivalent:*

1. *there is a way to insert the letters T into w to be accepted by $\mathcal{A}^{(1)}_{((q,0),(q',1,\rightarrow))}$*

2. *in the sequence of configurations encoded by w, there is a configuration where the state was q and the head was pointing to a cell containing 0, and in the next configuration, the head was moved to the right, but the state was not changed to q' (computation error).*

Proof. ($\Leftarrow$). We insert the letters T in front of the binary representation of the cell number where the violation occurs. The violation involves two configurations: in the first, we insert m's in front of 0's, and p's in front of 1's, and in the second, it's the other way around.

This way, we inserted all the letters of T (exactly) once into w, and $\mathcal{A}^{(1)}_{((q,0),(q',1,\rightarrow))}$ is now able to recognize the patterns (4.1) and (4.2) described above.

($\Rightarrow$). For the other direction, let w be a well-formed sequence of configurations such that there exists a way to insert the letters T into w, in order to obtain a word w_T accepted by $\mathcal{A}^{(1)}_{((q,0),(q',1,\rightarrow))}$.

Since each letter of T can be inserted only once, the sub-word matched by $(m_1 0 + p_1 1) \dots (m_{P(n)} 0 + p_{P(n)} 1)$ in pattern (4.1) in $\mathcal{A}^{(1)}_{((q,0),(q',1,\rightarrow))}$ has to be the same as the one matched by $(p_1 0 + m_1 1) \dots (p_{P(n)} 0 + m_{P(n)} 1)$ in pattern (4.2), up to exchanging m's and p''s.

Moreover, having exactly one \$ symbol in between the two patterns ensures that they correspond to the same cell, but in two successive configurations.

Finally, the facts that q'' is different that q' and that $\mathcal{M}$ is deterministic ensures that the sequence of configurations represented by w indeed contains a computation error according to the rule $\Delta(q, 0) = (q', 1, \rightarrow)$. □

We thus get the following lemma for the automaton $\mathcal{A}_{\mathsf{NotDelta}}$.

Lemma 15. *For a word w which is well-formed sequence of configurations, these statements are equivalent:*

- *we can insert the letters T in w to obtain a word accepted by $\mathcal{A}_{\mathsf{NotDelta}}$,*
- *w has a computation error according to the transition rules Δ of $\mathcal{M}$.*

Proof. Construct all the $\mathcal{A}_t$ for $t \in \Delta$ (with Δ considered as a relation). Construct similarly an automaton recognizing the violation where a cell changes while the head was not here. Take the union of all these automata, the proof then follows from Lemma 14. □

By taking the union $\mathcal{A} = \mathcal{A}_{\mathsf{NotSeqCfg}} \cup \mathcal{A}_{\mathsf{NotDelta}}$, we finally get the intended result, which ends the reduction.

Lemma 16. *The following two statements are equivalent.*

- *the run of $\mathcal{M}$ starting in state q_0 with the tape initialized with t is accepting,*
- *there exists a word w in Γ^*, such that there is no way to insert the letters T in order to obtain a word accepted by $\mathcal{A}$.*

Proof. ($\Rightarrow$) Let w be the well-formed sequence of configurations representing the sequence of configurations of the accepting run in $\mathcal{M}$, with the tape initialized with t. Then by Lemma 13 and Lemma 15, there is no way to insert the letters T in order to obtain a word accepted by $\mathcal{A}_{\mathsf{NotSeqCfg}}$ or $\mathcal{A}_{\mathsf{NotDelta}}$.

($\Leftarrow$) Let $w \in \Gamma^*$ be a word such that there is no way to insert the letters T in order to obtain a word accepted by $\mathcal{A}$. First, since w is not accepted by $\mathcal{A}_{\mathsf{NotSeqCfg}}$, it represents a well-formed sequence of configurations, starting in state q_0 with the tape initialized with t and ending in state q_f (Lemma 13). Moreover, since there is no way to insert the letters to obtain a word from $\mathcal{A}_{\mathsf{NotDelta}}$, w has no computation error according to the transition rules Δ of $\mathcal{M}$ (Lemma 15). □

This ends the proof of Lemma 11. □

As a consequence, we get the EXPSPACE-hardness for linearizability when the specification is given as an NFA, and when the number of threads is bounded.

Theorem 5 (Linearizability is EXPSPACE-hard (NFA))**.** *Let $\mathcal{L}$ be a finite-state library, $\mathcal{S}$ be a sequential specification represented by an NFA, and $k \in \mathbb{N}$ be a bound on the number of threads.*

Verifying that $\mathcal{L}^k \sqsubseteq \mathcal{S}$ is EXPSPACE*-hard.*

4.5.3 Hardness for Deterministic Specifications

The hardness result of Theorem 5 only covers specifications represented by NFA's. We show here that the hardness result still holds even when the specification is given as a DFA.

Lemma 17 (Linearizability NFA to DFA)**.** *Let $\mathcal{L} = (\mathcal{X}, \mathbb{G}, K)$ be a library, $\mathcal{S}$ be a specification represented by an NFA, and k a number of threads. We can build, in polynomial time, a library $\mathcal{L}'$ and a specification $\mathcal{S}'$ represented by a DFA such that $\mathcal{L}^k \sqsubseteq \mathcal{S}$ if and only if ${\mathcal{L}'}^{3*k} \sqsubseteq \mathcal{S}'$.*

Proof. Let $\mathcal{S} = (Q, q_0, q_f, \Delta)$ be an NFA over the alphabet $\mathbb{M}$. To simplify the presentation, we assume that there are at most two outgoing transitions from each state, a *left* one and a *right* one. The idea of the proof generalizes easily when there are more (we will discuss the generalization along the proof).

We introduce two new letters m_a and m_b and build $\mathcal{S}' = (Q', q_0, q_f, \Delta')$ from $\mathcal{S}$ as follows. The set of states Q' contains Q, but will have additional intermediary states for each transition. The transition relation Δ' is similar to Δ, except that it will be deterministic, and use these intermediary states (Δ' will not contain any transition from Δ).

For a left transition $\delta = (q, m, q_1) \in \Delta$, we introduce two fresh states q^a_δ and q^b_δ in Q', and we replace the transition δ by three transitions (q, m_a, q^a_δ), $(q^a_\delta, m_b, q^b_\delta)$, and (q^b_δ, m, q_1). Similarly, for a right transition $\delta = (q, m, q_2) \in \Delta$, we introduce two fresh states q^a_δ and q^b_δ in Q', and replace δ by (q, m_b, q^b_δ), $(q^a_\delta, m_a, q^a_\delta)$, (q^b_δ, m, q_2).

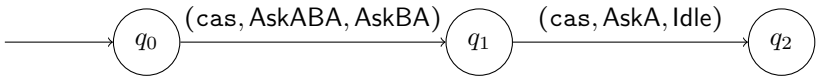
In total, we introduce two fresh states per transitions, so $|Q'| = |Q| + 2 * |\Delta|$ and each transition is replaced by three transitions, so $|\Delta'| = 3 * |\Delta|$. This concludes the construction of $\mathcal{S}'$. The intuition here is that the two new methods m_a and m_b correspond to operations which are overlapping in the executions of $\mathcal{L}'$. Thus, when linearizing, we can linearize them in any order, and either take the left transition to q_1, or the right transition to q_2. If the maximum number of outgoing transitions per state is p, then we insert p letters instead of 2, and do a similar construction.

We now build the library $\mathcal{L}'$ so that there is a correspondence between the executions of $\mathcal{L}^k$ and the ones of $\mathcal{L}'^{3*k}$. The library $\mathcal{L}'$ will have the same methods as $\mathcal{L}$, plus two additional methods for m_a and m_b. For each $\mathbb{M}$ operation, we will have one m_a operation, and one m_b operation overlapping with it (first *requirement*). This is the reason we use $3 * k$ threads in $\mathcal{L}'$. In order to ensure that a m_a operation and a m_b operation are overlapping with each $\mathbb{M}$ operation, we introduce a variable x, which can take values from the set of fresh values $\{\mathsf{Idle}, \mathsf{AskABA}, \mathsf{AskBA}, \mathsf{AskA}\}$, and which will be used to synchronize these three operations.

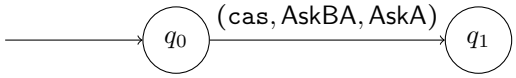
Note that adding threads may introduce behaviors which were not present in $\mathcal{L}^k$. To avoid this, we modify the methods $m \in \mathbb{M}$ so that at most k $\mathbb{M}$ operations may run in parallel (second *requirement*). We introduce a variable y, which can take values from $\{0, \ldots, k\}$ and which will play the role of a semaphore. Every time a method $m \in \mathbb{M}$ wants to execute, it will increment y (if $y < k$), and it will decrement it after the execution. The operations get blocked while $y = k$.

Formally, let $m_i \in \mathbb{M}$ be a method from $\mathcal{L}$ and let $\mathcal{A}_i = (Q^i, q^i_0, q^i_f, \Delta^i)$ be its method implementation (i.e. $\mathcal{A}_i = K(m_i)$). We change it into $\mathcal{A}'_i = (Q^{i'}, q^{i'}_0, q^{i'}_f, \Delta^{i'})$ as follows. We introduce four states and define $Q^{i'} = Q^i \uplus \{q_{\mathsf{start}}, q_{\mathsf{Idle}}, q_{\mathsf{AskABA}}, q_{\mathsf{end}}\}$. These will be used to ensure both requirements. The new initial state $q^{i'}_0$ is q_{start}, while the new final state is q_{end}.

To ensure the second requirement, i.e. that no more than k $\mathbb{M}$ operations may execute in parallel, we add k transitions $(q_{\mathsf{start}}, (\mathtt{cas}, y, i, i+1), q_{\mathsf{Idle}})$ for $0 \leq i < k$, as well as k transitions $(q^i_f, (\mathtt{cas}, y, i+1, i), q_{\mathsf{end}})$.



(a) LTS $\mathcal{A}_a$ of m_a.



(b) LTS $\mathcal{A}_b$ of m_b.

For the first requirement, i.e. that a m_a operation and a m_b operation are overlapping with each $\mathbb{M}$ operation, we introduce transitions $(q_{\mathsf{Idle}}, (\mathtt{cas}, x, \mathsf{Idle}, \mathsf{AskABA}), q_{\mathsf{AskABA}})$ and $(q_{\mathsf{AskABA}}, (\mathtt{read}, x, \mathsf{Idle}), q_0^i)$ and define the LTSs $\mathcal{A}_a$ and $\mathcal{A}_b$ of m_a and m_b (see Figures 4.6a and 4.6b). .

This completes the construction of the library $\mathcal{L}' = (\mathcal{X} \uplus \{x, y\}, \mathbb{G} \uplus \{0, \ldots, k\} \uplus \{\mathsf{Idle}, \mathsf{AskABA}, \mathsf{AskBA}, \mathsf{AskA}\}, K')$, where K' maps each m_i to $\mathcal{A}'_i$ as defined above, and maps m_a and m_b respectively to $\mathcal{A}_a$ and $\mathcal{A}_b$. We now have all the ingredients to show that:

$$\mathcal{L}^k \sqsubseteq \mathcal{S} \iff {\mathcal{L}'}^{3*k} \sqsubseteq \mathcal{S}'$$

($\Rightarrow$) Let e be an execution of ${\mathcal{L}'}^{3*k}$. Without loss of generality, we ignore the $\mathbb{M}$ operation in e which go through the added transitions, i.e. which didn't reach the state $q_0^{i'}$. These operations do not affect the overall executions, and by the definition of linearizability, if we can linearize an execution without them, we can also linearize an execution where they are present, as linearizability allows to remove operations which are pending.

Using the fact that no more than k $\mathbb{M}$ operations can execute at the same time in e, we can build an execution e' of $\mathcal{L}^k$ simply by changing the thread in which some operations are executed, and by removing all m_a and m_b operations.

By assumption, e' is linearizable with respect to some $w \in \mathcal{S}$. Moreover, since every $\mathbb{M}$ operation in e is overlapping with a m_a and a m_b operation, we can linearize e with respect to $\mathcal{S}'$, by inserting before every letter $m \in \mathbb{M}$ of w the m_a and m_b letters in the correct order, depending on whether a left or right transition was taken for the transition of m.

($\Leftarrow$) Let e be an execution of $\mathcal{L}^k$. We modify e into an execution e' of ${\mathcal{L}'}^{3*k}$ by adding overlapping m_a and m_b operations in parallel with each $\mathbb{M}$ operations. This is possible because we have $3 * k$ threads at our disposal. Moreover, e' satisfy the requirement that no more than k $\mathbb{M}$ operation execute at the same time, as it is derived from e.

By assumption, e' is linearizable with respect to some $w \in \mathcal{S}'$, and we can conclude that e is linearizable with respect to the projection $w_{|\mathbb{M}} \in \mathcal{S}$. □

As a corollary, we get the hardness theorem.

Theorem 4 (Linearizability is EXPSPACE-hard (DFA))**.** *Let $\mathcal{L}$ be a finite-state library, $\mathcal{S}$ be a sequential specification represented by an DFA, and $k \in \mathbb{N}$ be a bound on the number of threads.*

Verifying that $\mathcal{L}^k \sqsubseteq \mathcal{S}$ is EXPSPACE*-hard*[4]*.*

4.6 Linearizability is in EXPSPACE (bounded threads)

In this section, we prove that even when the specification is another finite-state library, checking linearizability is still in EXPSPACE for a bounded number of threads. This extends the result of Alur et al. [3], which was given for sequential specifications.

Theorem 6. *Let $\mathcal{L}_1 = (\mathcal{X}_1, \mathbb{G}_1, K_1)$ and $\mathcal{L}_2 = (\mathcal{X}_2, \mathbb{G}_2, K_2)$ be two finite-state librarys and $k \in \mathbb{N}$ a bound on the number of threads. Checking whether ${\mathcal{L}_1}^k \sqsubseteq {\mathcal{L}_2}^k$ is in* EXPSPACE[5]*.*

Proof. By Theorem 2, we know that linearizability is equivalent to executions inclusion. Our goal is thus to check ${\mathcal{L}_1}^k \sqsubseteq {\mathcal{L}_2}^k$.

Going back to the definition of a run of ${\mathcal{L}_1}^k$, we notice that a configuration is made of a pair giving: 1) a value in $\mathbb{G}_1$ for each variable in $\mathcal{X}_1$, 2) the state in which each thread $t_1, \ldots, t_k$ is. Thus, the state-space of configurations has size (roughly): $|\mathbb{G}_1|^{|\mathcal{X}_1|} * {N_1}^k$ where N_1 is the sum of the number of states in each LTS $K_1(m)$ for $m \in \mathbb{M}$.

We can thus build an automaton, of exponential size, whose states are the configurations, and which recognizes exactly the executions ${\mathcal{L}_1}^k$. We then apply the same reasoning for ${\mathcal{L}_2}^k$. Since checking inclusion between two non-deterministic finite automata is PSPACE-complete, we can solve our execution-set inclusion problem in EXPSPACE. □

This leads to the following result, which states that in general, the complexity of linearizability is EXPSPACE-complete when the number of threads is bounded, regardless of whether the specification is a concurrent library $\mathcal{L}_2$, or a sequential specification $\mathcal{S}$ given by a DFA

Corollary 3 (Linearizability EXPSPACE-complete)**.** *Let $\mathcal{L}_1$ be a finite-state library, $\mathcal{L}_2$ be either a finite-state library, or a sequential specification given by a finite LTS, and $k \in \mathbb{N}$ a bound on the number of threads. Checking whether ${\mathcal{L}_1}^k \sqsubseteq {\mathcal{L}_2}^k$ is* EXPSPACE*-complete.*

Proof. The hardness follows from Theorem 4, even in the case where the specification is sequential and given by a DFA, while the membership in EXPSPACE follows from Theorem 6, and holds even when the specification $\mathcal{L}_2$ is a finite-state library. □

[4]The size of the input is the size of $\mathcal{L}$, plus the size of $\mathcal{S}$, plus k.

[5]The size of the input is k plus the sizes of $\mathcal{L}_1$ and $\mathcal{L}_2$.

4.7 Static Linearizability

Due to the high complexity of checking linearizability, a stronger criterion, called *static linearizability* is used in practice. Intuitively, an expert with knowledge of how the library works can manually specify where each operation should be linearized. Treiber Stack, as given in the introduction, is an example of an implementation where we can easily find the linearization points. They correspond to the `cas` instructions in the *Push* and *Pop* methods, in case the instruction succeeds in updating the stack. Fixing the linearization points permit to avoid checking all possible linearizations for every execution, and instead fixes the order.

To formalize static linearizability, we enrich executions by adding manually specified linearization points of the form: $\mathsf{lin}_t\, m(d_1, d_2)$ for $m \in \mathbb{M}$, $d_1, d_2 \in D$ and $t \in \mathbb{T}$. Such an action is called a *linearization action* and means that we should linearize the operation being executed by thread t at this point, using the return value d_2. We denote by LP the set of all linearization actions.

Definition 2. *A* marked *execution e is a sequence of $(\mathsf{Call} \cup \mathsf{Return} \cup \mathsf{LP})^*$, such that:*

- $e_{|\mathsf{Call} \cup \mathsf{Return}}$ *is an execution,*
- *for every matching call and return actions $c = \mathtt{call}_t\ m(d_1)$ and $r = \mathtt{ret}_t\ d_2$, there exists a unique action $\mathsf{lin}_t\, m(d_1, d_2)$ between c and r in e,*
- *a pending call $c = \mathtt{call}_t\ m(d_1)$ may have zero or one action $\mathsf{lin}_t\, m(d_1, d_2)$ after it, with $d_2 \in \mathbb{D}$.*

Let $\mathcal{S}$ be a sequential specification. A marked execution e is *linearizable* with respect to $\mathcal{S}$ if the sequence of tuples (m, d_1, d_2) corresponding to its linearization actions belongs to $\mathcal{S}$.

Remark 2. *If a marked execution e is linearizable with respect to $\mathcal{S}$, then the underlying execution $e_{|\mathsf{Call} \cup \mathsf{Return}}$ is too.*

A library *with manually specified linearization points* is a library where the LTS have additional transitions, labeled by $\mathbb{D}$. A transition labeled by $d_2 \in \mathbb{D}$ produce a linearization action $\mathsf{lin}_t\, m(d_1, d_2)$ when executed by thread t which is currently executing method m with argument d_1.

We put the restriction that such a library must always produce marked executions. In particular, every operation should have a linearization point which is specified. This can be verified syntactically, by making sure that there always exists a single linearization point on every control-flow path of a method.

A library with manually specified linearization points is said to be *linearizable* with respect to a sequential specification if all of its marked executions are. When linearization points are manually specified, we do not need to check all possible linearizations. This enables us to reduce linearizability checking to an inclusion problem, leading to decidability, as demonstrated in the following theorem.

Theorem 7. *Let $\mathcal{L}$ be a finite-state library with manually specified linearization points (producing marked executions), and let $\mathcal{S}$ be an NFA representing a sequential specification. Let k be an integer representing the number of threads*

The problems $\mathcal{L} \sqsubseteq \mathcal{S}$ and $\mathcal{L}^k \sqsubseteq \mathcal{S}$ are respectively EXPSPACE*-complete and* PSPACE*-complete.*

Proof. Showing $\mathcal{L}^k \sqsubseteq \mathcal{S}$ amounts to proving that every sequence of linearization actions produced by $\mathcal{L}^k$ is included in $\mathcal{S}$ (up to renaming $\mathsf{lin}_t\, m(d_1, d_2)$ into (m, d_1, d_2)).

Using the same construction as Theorem 6, we can build a finite automaton of exponential size which recognizes all the sequences of linearization actions produced by $\mathcal{L}^k$. Checking the inclusion between such an exponentially-size automaton and $\mathcal{S}$ can be done in PSPACE, since we can build it on-the-fly.

Simulating the marked executions produced by $\mathcal{L}$ is more complicated, and cannot be done by a finite automaton, because of the unbounded number of threads calling the library. We use the for this the VASS defined in Lemma 7. Then, checking the inclusion between sequences produced by a VASS and a finite automaton can be done in EXPSPACE.

The hardness of both results follows from the fact that state reachability can be reduced to linearizability, as shown in Lemma 6. □

4.8 Summary

In this chapter, we studied the theoretical complexity of linearizability (and as a result, of observational refinement). Our results show that, in general, linearizability cannot be trivially reduced to more common problems, such as state reachability.

In particular, we showed that linearizability of a finite-state library with respect to a regular sequential specification is not decidable, which is the first result studying the algorithmic complexity of linearizability when the number of threads is not bounded. We also showed that, when the number of threads is bounded, linearizability is EXPSPACE-complete, regardless of whether the specification is another finite-state library, or a regular sequential specification given by a DFA.

We then showed that the most used approach for proving linearizability, i.e. static linearizability, doesn't actually suffer from the hardness results mentioned above. Instead, we show that it has the same complexity as state reachability, namely EXPSPACE-complete, and PSPACE-complete when the number of threads is bounded.

However, not all libraries can be proved linearizable using this method, as the linearization points of some operations can sometimes depend from operations happening arbitrarily later in the execution. We study in the next two chapters techniques which can be used for arbitrary libraries, and show how to mitigate the hardness results.

Chapter 5

Checking Linearizability using Approximations

5.1 Introduction

We showed in the previous chapter that linearizability has a high theoretical complexity, but that static linearizability, the approach which is the most widely used to prove linearizability, can be reduced to more standard model-checking problems, in particular to state reachability. Since static linearizability requires finding the linearization points, it may not be applied to detect linearizability violations, as the reason for a negative answer can be a wrong choice for the linearization points.

In this chapter, our goal is to give an approach to finding linearizability violations. Techniques for finding linearizability violations are essentially based on an explicit enumeration of all possible linearization orders [14]. This solution had some success for finding bugs in small executions, but doesn't scale well as the number of operations increases, because of the exponential number of possible linearizations.

Our aim in this chapter is to improve these methods, and are based on finding a *bounding parameter* for which linearizability can be efficiently checked, and which has a good coverage of linearizability violations (i.e. most violations can be found with a small parameter bound). Our technique is built upon the theoretical work done in Theorem 1, most precisely the part which states that observational refinement and linearizability are equivalent to the history-set inclusion $H(\mathcal{L}_1) \subseteq H(\mathcal{L}_2)$. Moreover, our technique crucially depends on the fact that the partial order underlying a history is not arbitrary, but is an *interval order*, as noticed in Chapter 2.

Interval orders have a well defined measure, called the *interval length* (define in the following section). Our idea is to check the history-set inclusion only for histories whose interval length is smaller than some fixed bound b (the bounding parameter is the interval length), i.e. checking the inclusion $H(\mathcal{L}_1)^{\leq b} \subseteq H(\mathcal{L}_2)^{\leq b}$

where $H(\mathcal{L}_1)^{\leq b}$ is the set of histories of $\mathcal{L}_1$ whose interval length is smaller than b. As we will see, bounding the interval length doesn't bound the number of operations or threads in an execution.

Interval orders have a *canonical representation*, which is a representation of an interval order using integer vectors. We propose a way to instrument a library, by adding counters, in order to maintain the canonical representation of the history being executed. As a result, we obtain a library whose counter valuations represent $H(\mathcal{L}_1)^{\leq b}$. The second step is to find a symbolic representation F for the specification $H(\mathcal{L}_2)^{\leq b}$, and ensure that the counter valuations of the instrumentation of $\mathcal{L}_1$ always stay in F.

We give several ways of representing $H(\mathcal{L}_2)^{\leq b}$. We prove that when the specification $\mathcal{L}_2$ is sequential and given as a context-free language, this set can be described by a Presburger formula. We also prove that when the specification is a sequential data structure such as a stack or a queue, we can describe it using an *Operation Counting Logic* we introduce. Even though such specifications typically have an unbounded data domain, we can describe them using our logic, by using the notion of *data independence* [1], which intuitively states that the behaviors of such data structure do not depend on the actual data value being stored in the structure.

Overall, the contributions of this chapter are twofold. On the theoretical side, we show that bounding the interval length leads to the decidability for the problem of checking linearizability of a finite-state library with respect to a regular (or context-free) sequential specification, thus avoiding the undecidability result of Theorem 3.

In practice, we show that our technique can be used both for static and dynamic analysis. We show in particular that it is scalable in the size of the executions, meaning that our instrumentation using counters has a much smaller overhead than techniques which enumerate all possible linearizations. Moreover, we prove the interval length is a good bounding parameter, and leads to a good coverage of bugs, as all the violations we encountered could be found with an interval bound of 0 or 1.

5.2 Interval Length

We start by defining the interval length, which we will use later as our bounding parameter. The *past* of an element $o \in O$ of a poset $(O, <)$ is the set

$$\mathsf{past}(o) = \{o' \in O : o' < o\}$$

of elements ordered before o. This notion of operations' pasts induces a linear notion of time into histories due to the following fact.

Lemma 18 (Rabinovitch [45])**.** *The set* $\{\mathsf{past}(o) \mid o \in O\}$ *of pasts of an interval order* $(O, <)$ *is linearly ordered by set inclusion.*

Furthermore, this linear notion of time has an associated notion of *length*, which corresponds to the length of the linear order on operation's pasts.

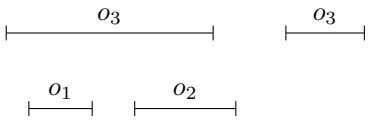


Figure 5.1: An execution with four operations. The interval order of its history has length 2.

.

Definition 3 (Greenough [26])**.** *The* interval length *(*length *for short) of an interval order* $(O, <)$ *is one less than the number of its distinct pasts.*

We denote as $\mathsf{len}(h)$ *the length of the interval order underlying a history* h*.*

Example 10. *Consider the executions composed of operations* o_1, o_2, o_3, o_4 *in Fig 5.1. Ignoring the labels of these operations, the happens-before relation contains the following constraints:* $o_1 < o_2 < o_4$ *and* $o_3 < o_4$*. It forms an interval order.*

The past of o_1 *and* o_3 *are the empty set . The past of* o_2 *is* $\{o_1\}$*. The past of* o_4 *is* $\{o_1, o_2, o_3\}$*. We notice the pasts are linearly ordered (by set inclusion). The length of this interval order is* 2*, as there are* 3 *distinct pasts.*

5.3 Bounding Interval Length

Given a set of histories H, we denote by $H^{\leq b}$ the set of histories in H whose interval length is at most b.

Assume that, for some bound b, we have found a violation to the inclusion $H(\mathcal{L}_1)^{\leq b} \subseteq H(\mathcal{L}_2)^{\leq b}$, that is, a history whose interval length is less than b, and which is in $H(\mathcal{L}_1)$ but not in $H(\mathcal{L}_2)$. Then we obtain a violation to the original inclusion $H(\mathcal{L}_1) \subseteq H(\mathcal{L}_2)$.

Interval orders have canonical representations which associate to elements integer-bounded intervals on the real number line; their canonical representations minimize the interval bounds to the interval-order length.

Lemma 19 (Greenough [26])**.** *An interval order* $(O, <)$ *of length* $b \in \mathbb{N}$ *has a canonical representation* $I : O \to \{0, \ldots, b\}^2$ *such that* $o_1 < o_2$ *iff* $\sup I(o_1) < \inf I(o_2)$*.*

Consider a history $h = (O, <, \ell)$ whose underlying interval length is at most b. Using the canonical representation $I : O \to \{0, \ldots, b\}^2$ of Lemma 19, h can be represented by a vector $v(h)$ in $(\mathbb{M} \times \mathbb{D} \times (\mathbb{D} \uplus \bot) \times \{0, \ldots, b\}^2 \to \mathbb{N}$, which counts, for each label $(m, d_1, d_2) \in \mathbb{M} \times \mathbb{D} \times (\mathbb{D} \uplus \bot)$, and for each interval $[i, j]$ in $\{0, \ldots, b\}$, the number of operations $o \in O$ of that label which are mapped to the interval $[i, j]$ (i.e. $I(o) = [i, j]$ and $\ell(o) = (m, d_1, d_2)$). We call $v(h)$ the *Parikh image* of h, as it can be seen as an extension of the Parikh image defined for words.

Thus, the (point-wise) Parikh image of $H(\mathcal{L}_1)^{\leq b}$ can be represented by a set of vectors $v(h)$ in $(\mathbb{M} \times \mathbb{D} \times (\mathbb{D} \uplus \bot) \times \{0, \ldots, b\}^2 \to \mathbb{N}$.

For a library $\mathcal{L}$, we show in Lemma 20 that the Parikh image of $H(\mathcal{L}_1)^{\leq b}$ can be represented as the set of reachable counter valuations of integer variables added to $\mathcal{L}$.

Lemma 20. *Let $\mathcal{L}$ be a library and $b \in \mathbb{N}$ an integer. We can construct a library $\mathcal{L}_b$ with additional shared integer variables such that the set of reachable counter valuations of $\mathcal{L}_b$ represents the Parikh image of $H(\mathcal{L})^{\leq b}$.*

Proof. We add one integer variable per element in $\mathbb{M} \times \mathbb{D} \times (\mathbb{D} \uplus \bot) \times \{0, \ldots, b\}^2$, counting the number of operations with this label and with this interval.

The new library also maintains, in a shared variable, a bounded integer $k \in \{0, \ldots, b\}$, specifying the current round. When the library sees a return action immediately followed by a call action, it increments k (it enters a new round).

Every time a method m is called with argument d_1 in round k, the counter for $(m, d_1, \bot, k, b)$ is incremented. When it returns with value d_2 in round k', the counter for $(m, d_1, \bot, k, b)$ is decremented, while the counter for (m, d_1, d_2, k, k') is incremented.

Then, a history h belongs to $H(\mathcal{L}_{\mathcal{S}})^{\leq b}$ if and only if there exists a run in $\mathcal{V}^b$ which reaches a valuation of the newly added counters equal to $v(h)$. □

5.4 Context-Free Specification – Decidability

We show in Lemma 21 that, when $\mathcal{S}$ is a sequential specification given by a context-free grammar, the Parikh image of $H(\mathcal{L}_{\mathcal{S}})^{\leq b}$ can be represented by a Presbuger formula $\varphi_{\mathcal{S}}$ (where $\mathcal{L}_{\mathcal{S}}$ is a library giving the set of executions linearizable with respect to $\mathcal{S}$).

Lemma 21. *Let $\mathcal{S}$ be a sequential specification given by a context-free grammar ($\mathbb{M}$ and $\mathbb{D}$ are here finite sets), $\mathcal{L}_{\mathcal{S}}$ the library representing the set of executions which are linearizable with respect to $\mathcal{S}$, and $b \in \mathbb{N}$ an integer. Then, the Parikh image of $H(\mathcal{L}_{\mathcal{S}})^{\leq b}$ (the histories linearizable with respect to $\mathcal{S}$, and whose interval length is length than b) is Presburger definable.*

Proof. We perform a series of transformations on $\mathcal{S}$, in order to obtain a new context-free language $\mathcal{S}^b$ over alphabet $\mathbb{M} \times \mathbb{D} \times (\mathbb{D} \uplus \bot) \times \{0, \ldots, b\}^2$ whose Parikh image is equal to the one of $H(\mathcal{L}_{\mathcal{S}})^{\leq b}$.

First, we define $\mathcal{S}'$ to be the set of sequences over $\mathbb{M} \times \mathbb{D} \times \mathbb{D} \times \{0, \ldots, b\}^2$ whose projection on $\mathbb{M} \times \mathbb{D} \times \mathbb{D}$ belongs to $\mathcal{S}$, and whose projection on $\{0, \ldots, b\}^2$ belongs to the regular language:

$$\left(\mathop{+}_{\substack{i \leq j \wedge \\ i \leq 0 \wedge j \geq 0}} [i, j] \right)^* \cdot \left(\mathop{+}_{\substack{i \leq j \wedge \\ i \leq 1 \wedge j \geq 1}} [i, j] \right)^* \cdots \left(\mathop{+}_{\substack{i \leq j \wedge \\ i \leq b \wedge j \geq b}} [i, j] \right)^*$$

Then, we define $\mathcal{S}''$ as the set of sequences over alphabet $\mathbb{M} \times \mathbb{D} \times (\mathbb{D} \uplus \bot) \times \{0, \ldots, b\}^2$ which can be obtained from $\mathcal{S}'$ by changing some letters (m, d_1, d_2, i, b) to $(m, d_1, \bot, i, b)$. Intuitively, this corresponds to the fact in the definition of linearizability, pending operations can be completed with a return value d_2.

Finally, $\mathcal{S}^b$ is defined as $(\mathbb{M} \times \mathbb{D} \times \{\bot\} \times \{0, \ldots, b\} \times \{b\})^* \cdot \mathcal{S}''$. The label of pending operations that we add in front to $\mathcal{S}''$, correspond to the pending operations that we're allowed to remove from the definition of linearizability.

We can then show:

$$\forall h.\ h \in H(\mathcal{L}_{\mathcal{S}})^{\leq b} \iff v(h) \in \Pi(\mathcal{S}^b)$$

($\Rightarrow$) Let $h \in H(\mathcal{L}_{\mathcal{S}})^{\leq b}$, and let $w \in \mathcal{S}$ such that $h \sqsubseteq \mathcal{S}$ By definition of linearizability, we know that after removing some pending operations, and completing the others, there is bijection between the operations and the operations of the sequential execution w. For each operation in w, we add the interval specified by the canonical representation I of h. We obtain a sequence in $\mathcal{S}'$.

Then, for the operations in w which correspond to a pending operation in h (i.e. they were completed for the linearization), we rewrite (m, d_1, d_2, i, b) into $(m, d_1, \bot, i, b)$. We get a sequence in $\mathcal{S}''$.

Finally, we add at the beginning letters from $\mathbb{M} \times \mathbb{D} \times \{\bot\} \times \{0, \ldots, b\} \times \{b\}$ for all the pending operations in h which were discarded for the linearization to w, and obtain a sequence in $\mathcal{S}^b$, whose Parikh image is exactly $v(h)$.

($\Leftarrow$) Let $w^b \in \mathcal{S}^b$ such that $v(h) = \Pi(w^b)$. Let $w'' \in \mathcal{S}'', w' \in \mathcal{S}'$ and $w \in \mathcal{S}$, be the sequences used to construct $w^b \in \mathcal{S}^b$. The fact that $v(h) = \Pi(w^b)$ gives up a bijection between the operations of h and the letters of w^b.

First, remove from h the pending operations which are at the beginning of w^b, but which do not appear in w''. Then, complete the other pending operations of h, by using the return values which were deleted when going from w' to w''. Finally, order the operations as dictated by w', The fact that the projection of w' on $\{0, \ldots, b\}^2$ belongs to the regular language:

$$\left(\mathop{+}_{\substack{i \leq j \wedge \\ i \leq 0 \wedge j \geq 0}} [i,j]\right)^* \cdot \left(\mathop{+}_{\substack{i \leq j \wedge \\ i \leq 1 \wedge j \geq 1}} [i,j]\right)^* \cdots \left(\mathop{+}_{\substack{i \leq j \wedge \\ i \leq b \wedge j \geq b}} [i,j]\right)^*$$

ensures that the order given by w' respects the happens-before relation of h (for every two operation o_1, o_2, if $\sup I(o_1) < \inf I(o_2)$, then o_1 is before o_2 in w'. We conclude that h is linearizable with respect to $w \in \mathcal{S}$, and belongs to $H(\mathcal{L}_{\mathcal{S}})^{\leq b}$. □

By negating $\varphi_{\mathcal{S}}$, we can then reduce the inclusion $H(\mathcal{L})^{\leq b} \subseteq H(\mathcal{L}_{\mathcal{S}})^{\leq b}$ to the intersection between the reachable markings of a VASS, and a Presburger formula, which can be reduced to VASS configuration reachability [8].

Theorem 8 (Bounded Interval Length)**.** *Let $\mathcal{L}$ be a finite-state library, $\mathcal{S}$ be a context-free language representing a sequential specification, and $b \in \mathbb{N}$ a bound on the interval length. Verifying whether there exists a history of $\mathcal{L}$ whose length is smaller than b, and which is not linearizable with respect to $\mathcal{S}$ is decidable*

(i.e. $H(\mathcal{L}_1)^{\leq b} \notin \mathcal{S}$) is decidable, and can be reduced to VASS configuration reachability.

Proof. Let $\mathcal{L}_{\mathcal{S}}$ be the library representing the set of executions with are linearizable with respect to $\mathcal{S}$ (defined in Lemma 5). Thus, $H(\mathcal{L})^{\leq b} \sqsubseteq \mathcal{S}$ is equivalent to $H(\mathcal{L})^{\leq b} \subseteq H(\mathcal{L}_{\mathcal{S}})$, and in turn equivalent to $H(\mathcal{L})^{\leq b} \subseteq H(\mathcal{L}_{\mathcal{S}})^{\leq b}$.

Since Parikh images describes interval orders without loss of precision (up to operation identifiers renaming), we can check the Parikh images inclusion between the two sets. Lemma 20 states that the Parikh image of $H(\mathcal{L})^{\leq b}$ can be represented by the set of reachable counter valuations of a library $\mathcal{L}_b$, while Lemma 21 states that the one of $H(\mathcal{L}_{\mathcal{S}})^{\leq b}$ can be described by a Presburger formula φ. The library $\mathcal{L}_b$ is not finite-state, but can still be simulated by a VASS. This is because the shared counters we added (one for each element of $\mathbb{M} \times \mathbb{D} \times (\mathbb{D} \uplus \bot) \times \{0, \ldots, b\}^2$) are never tested to 0, and can just be represented as another counter of the VASS simulating $\mathcal{L}$.

Overall, this means that checking $H(\mathcal{L})^{\leq b} \notin \mathcal{S}$ is equivalent to finding a valuation in $\mathcal{V}$ which satisfies $\neg\varphi$. This question of intersection between the reachable valuations of a VASS and a Presburger formula was studied by Bouajjani and Habermehl [8], and shown to be reducible to VASS configuration reachability.

The intuition is that, $\neg\varphi$ being a Presburger formula, can also be represented by the set of reachable valuations of a VASS as well. Then, finding a valuation which is reachable in two VASS can be done by using transitions which decrements counters from the two VASS simultaneously. If the counters can all reach 0, this means that the set of reachable valuations of the two VASS have a non-empty intersection. □

5.5 Experiments

5.5.1 Operation Counting Logic

Applying the decidability result of the previous section to real libraries is not so trivial. The main issue is that the specifications used in practice, such as stacks and queues, have an infinite data domain $\mathbb{D}$. Thus, they do not represent context-free languages and we cannot apply *Lemma* 21 to obtain a Presburger formula for the set $H(\mathcal{L}_{\mathcal{S}})^{\leq b}$. To circumvent this issue, we introduce Operation Counting Logic (see Fig 5.2), which is similar to Presburger arithmetic, but can also deal with the infinite domain $\mathbb{D}$.

We allow predicates P of arbitrary arity over operation labels, so long as they are evaluated in polynomial time. Furthermore, operation-label variables are quantified only over the operation labels which occur in the history over which a formula is evaluated. The satisfaction relation $\vDash$ for quantified formulas is defined by:

$$h \vDash \exists x.\ F \quad \text{iff} \quad \exists o \in O.\ h \vDash F[x \leftarrow f(o)].$$

$$
\begin{aligned}
&i, j \in \mathbb{N} \qquad \text{integer constants} \\
&\quad d \in \mathbb{M} \times \mathbb{D} \times \mathbb{D} \qquad \text{operation-label constants} \\
&\quad x : \mathbb{M} \times \mathbb{D} \times \mathbb{D} \qquad \text{operation-label variables} \\
&X ::= d \mid x \\
&T ::= i \mid \#(X, i, j) \mid T + T \\
&F ::= T \leq T \mid P(X, ., X) \mid \neg F \mid F \wedge F \mid \exists x.\ F
\end{aligned}
$$

Figure 5.2: The syntax of Operation Counting Logic.

where $h = (O, <, f)$. The predicate $\#(X, i, j)$ is interpreted over a history h as being the number of operations o whose interval in the canonical representation of h is $[i, j]$ and whose label is X. The *quantifier count* of an operation counting formula F is the number of quantified variables in F.

Lemma 22. *Checking if a history h satisfies an operation counting formula F of fixed quantifier count is decidable in polynomial time.*

Proof. This follows from the facts that: (1) functions and predicates are polynomial-time computable, and (2) quantifiers are only instantiated over labels occurring in h. The latter implies that quantifiers can be replaced by a disjunction over the labels occurring in $h = (O, <, f)$:

$$h \vDash \exists x.\ F \quad \text{iff} \quad h \vDash \bigvee_{o \in O} F[x \leftarrow f(o)]. \quad \square$$

Figure 5.3 defines four operation-counting formulas:

- φ_{rv} describes *remove violations* in which a *Pop* operation (resp., *Deq*) returns a value for which there is no corresponding *Enq*.
- φ_{ev} describes *empty violations* in which some *Pop* (resp., *Deq*) operation returns *Empty*, yet whose span is covered by the presence of some pushed (resp., enqueued)element which has not (yet) been popped (resp., dequeued).
- φ_{fv} describes *FIFO violations* in which some pair of *Deq*'s occur in the *opposite* order of their corresponding *Enq*'s.
- φ_{lv} describes *LIFO violations* in which some pair of *Pop*s occur in the *same* order as their corresponding *Push*'s, and the second *Push* occurs before the first *Pop*.

Let $F_{\mathsf{Stack}} = \varphi_{\text{rv}} \vee \varphi_{\text{ev}} \vee \varphi_{\text{fv}}$ and $F_{\mathsf{Queue}} = \varphi_{\text{rv}} \vee \varphi_{\text{ev}} \vee \varphi_{\text{lv}}$. These formulas are complete when considered against histories of interval length at most 2, meaning that a history of length at most 2 is not linearizable with respect to the Stack (resp., Queue) if and only if it satisfies formula F_{Stack} (resp., F_{Stack}). We will see

$$\mathsf{total}(x,i,j) = \sum_{i\le i'\le j'\le j} \#(x,i',j')$$

$$\mathsf{before}_b(x,y) = \bigvee_{0\le i<b} \left(\begin{array}{l} \mathsf{total}(x,0,i) > 0 \wedge \\ \mathsf{total}(y,0,i) = 0 \wedge \mathsf{total}(x,i{+}1,b) = 0 \end{array} \right)$$

$$\mathsf{match}(x,y) = \mathsf{Push}(x) \wedge \mathsf{Pop}(y) \wedge \mathsf{SameVal}(x,y)$$

$$\begin{aligned}
\varphi_{\mathrm{rv}} &= \exists x, y.\ \mathsf{match}(x,y) \wedge \mathsf{before}_b(y,x) \\
\varphi_{\mathrm{ev}} &= \exists x, y, z.\ \mathsf{match}(x,z) \wedge \mathsf{EmptyVal}(y) \\
&\quad \wedge \mathsf{before}_b(x,y) \wedge \mathsf{before}_b(y,z) \\
\varphi_{\mathrm{fv}} &= \exists x_1, x_2, y_1, y_2.\ \mathsf{match}(x_1,y_1) \wedge \mathsf{match}(x_2,y_2) \\
&\quad \wedge \mathsf{before}_b(x_1,x_2) \wedge \mathsf{before}_b(y_2,y_1) \\
\varphi_{\mathrm{lv}} &= \exists x_1, x_2, y_1, y_2.\ \mathsf{match}(x_1,y_1) \wedge \mathsf{match}(x_2,y_2) \\
&\quad \wedge \mathsf{before}_b(x_1,x_2) \wedge \mathsf{before}_b(y_1,y_2) \wedge \mathsf{before}_b(x_2,y_1)
\end{aligned}$$

Figure 5.3: Operation-counting formulas characterizing atomic data structure violations, parameterized by the interval length $b \in \mathbb{N}$. The predicates $\mathsf{Push}(x)$, $\mathsf{Pop}(x)$, $\mathsf{EmptyVal}(x)$ hold when x is the label of a push, pop, or empty-pop operation, respectively, and $\mathsf{SameVal}(x,y)$ holds when x and y are labels with the same value, either in the argument or return position. All formulas are of size polynomial in b, and have fixed quantifier counts.

in next chapter a systematic way of obtaining the violating patterns for several data structures including the Stack and the Queue. We can extract from this study complete formulas for any interval length.

As in previous work [1, 29], our arguments for using these properties rely on *data independence* [56], i.e., that library executions are closed under consistent renaming $\mathbb{D} \to \mathbb{D}$. Data independence allows to assume that the values pushed on the stack (or enqueues on the queue) are unique. This is not a restriction as most implementations of stacks and queues are data independent in practice.

5.5.2 Static Analysis

Given an operation counting formula F describing possible violations of a data structure $\mathcal{S}$, we add assertions into the instrumented library $\mathcal{L}_b$ to ensure that the inclusion $H(\mathcal{L})^{\le b} \subseteq H(\mathcal{L}_{\mathcal{S}})^{\le b}$ holds. Whenever an assert fails, i.e. whenever a counter valuation is found to violate formula F, it means we have a history h which violates the inclusion, and which is not linearizable.

Despite the difficultly of precise static reasoning about thread interleavings, we have successfully applied our approach using CSeq [22] and CBMC [37] as a back-end. CSeq reduces concurrent programs to sequential ones using a method called sequentialization [38] and treat sequential programs using SAT/SMT-based bounded model checking (results in Table 5.1). Among 5 data structure implementations, we manually injected 9 realistic concurrency bugs. All bugs were uncovered as refinement violations with approximation $b = 0$ or $b = 1$, in

Library	Bug	P	b	m	n	Time
Michael-Scott Queue	B_1 (`head`)	2×2	1	2	2	24.76s
Michael-Scott Queue	B_1 (`tail`)	3×1	1	2	3	45.44s
Treiber Stack	B_2	3×4	1	1	2	52.59s
Treiber Stack	B_3 (`push`)	2×2	1	1	2	24.46s
Treiber Stack	B_3 (`pop`)	2×2	1	1	2	15.16s
Elimination Stack	B_4	4×1	0	1	4	317.79s
Elimination Stack	B_5	3×1	1	1	4	222.04s
Elimination Stack	B_2	3×4	0	1	2	434.84s
Lock-coupling Set	B_6	1×2	0	2	2	11.27s
LFDS Queue	B_7	2×2	1	1	2	77.00s

Table 5.1: Runtimes for the static detection of injected refinement violations with CSeq & CBMC. For a given program $P_{i \times j}$ with i and j invocations to the push and pop methods, we explore the n-round round-robin schedules of $P_{i \times j}$ with m loop iterations unrolled, with an interval length bound of b. Bugs are (B_1) non-atomic lock operation, (B_2) ABA bug Michael [41], (B_3) non-atomic CAS operation, (B_4) misplaced brace, (B_5) forgotten assignment, (B_6) misplaced lock, (B_7) predetermined capacity exceeded.

round-robin executions of up to 4 rounds, of a program with at most 4 *Push* and 4 *Pop* operations, and with loops unrolled up to 2 times. Although the time complexity of concurrent exploration is high independently of our operation-counting instrumentation, particularly as the number of rounds increases, one observes that our approximation is effective in detecting violations statically.

5.5.3 Monitoring

We show in this section that it also possible to apply the idea of bounding the interval length for the dynamic analysis of a library. In this context, executions become arbitrarily large, and as a consequence we cannot bound the interval length of histories produced by a library.

We define instead an approximation, which, for a given $b \in \mathbb{N}$, maps a history h to a history $A_b(h)$ which is linearizable with respect to h, and which has interval length (at most) b. We know that if a library can produce a history h, then it can also produce $A_b(h)$. As a result, if we find a history for which $v(A_b(h))$ does not satisfy the specification formula F, we have found a history of the library which is not linearizable.

Any definition of $A_b(h)$ satisfying the properties mentioned above would work, but for our purpose, we define $A_b(h)$ which maintains the last b intervals of h precisely, and merging all previous ones. Fig 5.4 gives an illustration.

Lemma 23. *Let $\mathcal{L}$ be a library and b an integer. We can add shared counters to the library, which maintain the Parikh image of $A_b(h)$, where h is the history which is currently being executed.*

Proof. We start by instrumenting $\mathcal{L}$ like in Lemma 20, by using an integer variable per element of $\mathbb{M} \times \mathbb{D} \times (\mathbb{D} \uplus \perp) \times \{0, \ldots, b\}^2$, as well as an integer variable $k \in \{0, \ldots, b\}$ representing the current round.

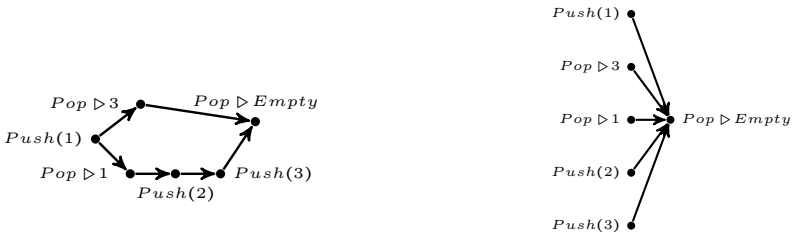


Figure 5.4: On the right, the approximation $A_1(h)$ of the history h on the left.

Instead of blocking when entering round $k = b + 1$, we apply an operation on the counters, which effectively merges the first two intervals, and shifts all intervals to the left.

Formally, a counter for the operations (m, d_1, d_2, i, j) with $i = 0$ or $i = 2$ is transferred to $(m, d_1, d_2, 0, j{-}1)$, while a counter for the operations (m, d_1, d_2, i, j) with $i > 1$ is transferred to $(m, d_1, d_2, i - 1, j - 1)$.

The integer variable k then stays at b, and the instrumentation continues as in Lemma 20, until the next round where we'll reapply the shifting of intervals, and so on. □

Figure 5.5 demonstrates that A_b covers most or all violations with small values of b. While A_4 suffices to cover all violations at nearly all data points — besides the first point, where the sample size of 1023 executions is small relative to the 8 operations — all values $b > 0$ capture a nontrivial and increasing number of violations. As the execution-sample size increases (the x-axis is ordered by number of executions over operations) the value of b required to capture a violation appears to decrease, all violations being captured by A_3 after a certain point.

Figure 5.6 compares the runtime overhead of our A_2 approximation versus a traditional linearizability checker sampling executions with up to 20 operations on Scal's nonblocking Michael-Scott queue. Despite our best-effort implementation, one observes the cost incurred by enumerating all possible linearizations: as the number of operations increases, the number of linearizations increases exponentially, and performance plummets. With only 20 operations, instrumentation overhead is nearly 1000x. Our counting-based implementation of A_2 avoids this dramatic overhead: though not seriously optimized, we observe a 2.01x geometric-mean runtime overhead. This scalability suggests that our approximation can be used not only for systematic concurrency testing with few method invocations, but also for runtime monitoring, where the number of operations grows without limit.

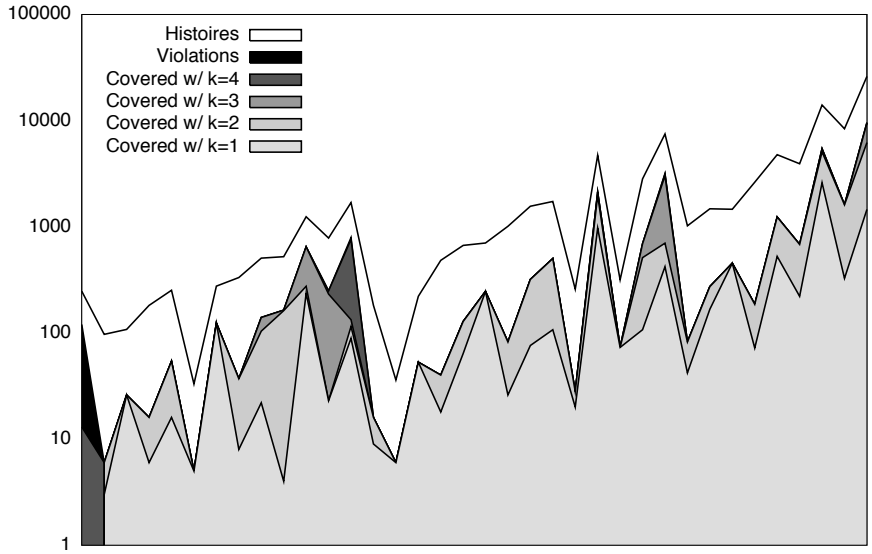


Figure 5.5: Comparison of violations covered with $b \leq 4$. Each data point counts histories on a logarithmic scale over all executions up to 5 preemptions on Scal's nonblocking bounded-reordering queue with $i \leq 4$ enqueue operations and $j \leq 4$ dequeue operations. The x-axis is ordered by increasing number of executions (1023–2359292) over $i+j$; we show only points with over 1000 executions. The largest data points measure the total number of unique histories encountered over a given set of executions. Second are the number of those histories violating refinement. Following are the numbers of those violations covered by A_b, for varying values of b. In this experiment A_0 exposed no violations.

5.6 Summary

We proposed in this chapter bug-detection techniques exploiting various non-trivial properties of the set of histories of a given library. More precisely, they are based on the fact that histories are interval orders, and on bounding the interval length of those orders.

In theory, we prove that checking linearizability for a given interval bound is decidable, enabling us to understand even further the limits of decidability outlined in Chapter 4.

Moreover, we showed in practice that bounding the interval length is a good exploration strategy, as most violations are found with a bound of 0 or 1. The technique is also scalable, as it is based on a representation of histories using counters, and specifications using arithmetic formulas, and does not rely on an explicit enumeration of all linearization orders. We showed applications in both static and dynamic analysis contexts.

A limitation of this work is that it cannot be applied to prove that a li-

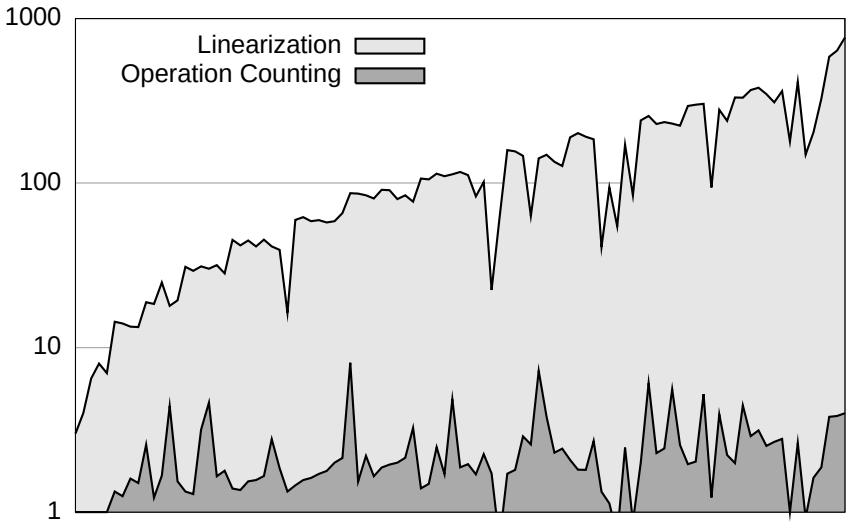


Figure 5.6: Comparison of runtime overhead between linearization-based monitoring and operation counting (for A_2) for up to 20 operations. Each data point measures runtime on a **logarithmic scale**, normalized to unmonitored execution time, over all executions up to 3 preemptions on Scal's nonblocking Michael-Scott queue with $i \leq 10$ enqueue operations and $j \leq 10$ dequeue operations. The x-axis is ordered by increasing $i+j$, and each data point is sampled from up to 126600 executions. Times do not include pre-calculation of sequential histories for linearization-based monitoring. While our A_b monitor scales well, usually maintaining under 3x overhead, the linearization monitor scales **exponentially**, running with nearly **1000x overhead** with 20 operations.

brary is linearizable. In the next chapter, we show that by carefully analyzing the specifications used in practice, such as stacks and queues, we can create algorithms which can be used to prove linearizability, even in the case where linearization points are not fixed.

Chapter 6

Linearizable Stacks, Queues, and more

6.1 Introduction

In the previous chapters, we showed that linearizability has a high theoretical complexity, but that despite the hardness results, several approaches can be used to attack the problem. One of these approaches, static linearizability, is the most common approach to prove linearizability, and is based on manually finding the linearization points of each method. This approach can never be used to find linearizability violations. On the other hand, bounding the interval length leads to an efficient bug-finding technique, which both is scalable and has a good coverage for linearizability violations (i.e. most violations can be found with bound 0 or 1). However, this approach cannot be used to prove linearizability.

The goal of this chapter is to provide a technique which can prove that a library is linearizable, without the need for manually specifying the linearization points. For this, we explore the problem of verifying linearizability for certain *fixed* specifications, and show that we can reduce it to state reachability, despite being harder for *arbitrary* specifications (as shown in Chapter 4). We believe that fixing the specification parameter of the verification problem is justified, since in practice, there are few specifications for which specialized concurrent implementations have been developed. We provide a general methodology for carrying out this reduction, and instantiate it on four specifications: the atomic queue, stack, register, and mutex.

Our reduction to state reachability holds on any library model which is closed under intersection with regular languages and which is *data independent* – informally meaning that the behaviors of the library do not depend on the actual data value being passed around. For the specifications in question, our approach relies on expressing its violations as a finite union of regular languages. This approach is similar in spirit to the one in the previous chapter, where we

expressed violations of such data-structures in Operation Counting Logic, except that it gives a complete description of all possible violations, and does not rely on bounding the interval length.

In our methodology, we express the atomic object specifications using inductive rules to facilitate the incremental construction of valid executions. For instance in our atomic queue specification, one rule specifies that a dequeue operation returning empty can be inserted in any execution, so long as each preceding enqueue has a corresponding dequeue, also preceding the inserted empty-dequeue. This form of inductive rule enables a locality aspect for reasoning about linearizability violations.

First, we prove that a sequential execution is invalid if and only if some subsequence could not have been produced by one of the rules. Under certain conditions this result extends to concurrent executions: an execution is not linearizable if and only if some projection of its operations cannot be linearized to a sequence produced by one of the rules. We thus correlate the finite set of inductive rules with a finite set of classes of non-linearizable concurrent executions. We then demonstrate that each of these classes of non-linearizable executions is regular, which characterizes the violations of a given specification as a finite union of regular languages. The fact that these classes of non-linearizable executions can be encoded as regular languages is somewhat surprising since the number of data values, and thus alphabet symbols, is, a priori, unbounded. Our encoding thus relies on the aforementioned *data independence* property, and we show that we only need 2 or 3 distinct data values to find all possible violations.

To complete the reduction to state reachability, we show that linearizability is equivalent to the emptiness of the language intersection between the set of executions and a finite union of regular languages. When the library is finite-state, this reduces to the coverability problem for VASS, which is decidable, and EXPSPACE-complete.

To summarize, our contributions are:

- a generic reduction from linearizability to state reachability,
- its application to the atomic queue, stack, register, and mutex specifications,
- the methodology enabling this reduction, which can be reused on other specifications, and
- the first decidability results for linearizability without bounding the number of concurrent threads.

Henzinger et al. [29] had the idea of studying linearizability with respect to a specific data structure, namely the queue. Their result can be seen as a particular case of ours. In our study, we gave a systematic way to obtain the violation properties, which can be applied to different data structures in a uniform way. Dodds et al. [19] tried to extend this idea to the stack, but were only partially successful, as their method wasn't completely independent from the implementation. Abdulla et al. [1] gave the violating properties for more

data structures, including the queue and stack, but their method requires to give the linearization points in a constructive manner.

We give in Section 6.2 some notations and definitions specific to this chapter. Then, we give in Section 6.3 our formalism used to define data-structures using inductive rules. We show how to use our formalism to define the aforementioned specifications.

Section 6.4 gives the general methodology we use to reduce linearizability to state reachability. The conditions needed for this reduction are discussed in Section 6.5 and Section 6.6, and proven true for the specifications we study.

Finally, we explain in Section 6.7 how to use our reduction to obtain the decidability of linearizability for finite-state libraries with respect to the queue, stack, register and mutex. This constitutes the first decidability result for verifying linearizability when the number of threads is not bounded.

6.2 Notations and Definitions

For this chapter, we consider that methods have exactly one argument, or one return value. Return values are transformed into argument values for uniformity. In order to differentiate methods taking an argument (e.g., the *Enq* method which inserts a value into a queue) from the other methods, we identify a subset $\mathbb{M}_i \subseteq \mathbb{M}$ of *input* methods which do take an argument. In that context, labels of operations are denoted $m(d)$, with $m \in \mathbb{M}$, $d \in \mathbb{D}$, instead of $(m, d, \perp)$.

We also restrict ourselves to *completed* executions, where each call action has a corresponding return action. This simplification is sound when methods can always make progress in isolation [29]: formally, for any execution e with pending operations, there exists an execution e' obtained by extending e only with the return actions of the pending operations of e. Intuitively, this means that methods can always return without any help from outside threads.

The *projection* $u_{|D}$ of a sequential execution u to a subset $D \subseteq \mathbb{D}$ of data values is obtained from u by erasing all operations with a data value not in D. The set of projections of u is denoted $\mathsf{proj}(u)$. We write $u \setminus d$ for the projection $u_{|\mathbb{D} \setminus \{d\}}$. The notion of projection extends to executions and histories.

A sequential execution u is said to be *differentiated* if, for all input methods $m \in \mathbb{M}_i$, and every $d \in \mathbb{D}$, there is at most one $m(d)$ operation in u. The subset of differentiated sequential executions of a set S is denoted by $S_{\neq}$. The definition extends to executions and histories. For instance, an execution is differentiated if for all input methods $m \in \mathbb{M}_i$ and every $d \in \mathbb{D}$, there is at most one call action $\mathtt{call}_o\ m(d)$.

Example 11. $\mathtt{call}_{t_1}$ *Enq*(7)$\cdot\mathtt{call}_{t_2}$ *Enq*(7)$\cdot\mathtt{ret}_{t_1}\cdot\mathtt{ret}_{t_2}$ *is not differentiated, as there are two call actions with the same input method (Enq) and the same data value.*

A *renaming* r is a function from $\mathbb{D}$ to $\mathbb{D}$. Given a sequential execution (resp., execution or history) u, we denote by $r(u)$ the sequential execution

(resp., execution or history) obtained from u by replacing every data value d by $r(d)$.

Definition 4. *The set of sequential executions (resp., executions or histories) S is* data independent *if:*

- *for all $u \in S$, there exists $u' \in S_{\neq}$, and a renaming r such that $u = r(u')$,*
- *for all $u \in S$ and for all renaming r, $r(u) \in S$.*

When checking that a data-independent library $\mathcal{L}$ is linearizable with respect to a data-independent specification S, it is enough to do so for differentiated executions [1]. Thus, in the remainder of the paper, we focus on characterizing linearizability for differentiated executions, rather than arbitrary ones.

Lemma 24 (Abdulla et al. [1])**.** *A data-independent implementation $\mathcal{L}$ is linearizable with respect to a data-independent specification S, if and only if $\mathcal{L}_{\neq}$ is linearizable with respect to $S_{\neq}$.*

Proof. ($\Rightarrow$) Let e be a (differentiated) execution in $\mathcal{L}_{\neq}$. By assumption, it is linearizable with respect to a sequential execution u in S, and the bijection between the operations of e and the ones of u ensure that u is differentiated and belongs to $S_{\neq}$.

($\Leftarrow$) Let e be an execution in $\mathcal{L}$. By data independence of $\mathcal{L}$, we know there exists $e_{\neq} \in \mathcal{L}_{\neq}$ and a renaming r such that $r(e_{\neq}) = e$. By assumption, $e_{\neq}$ is linearizable with respect to a sequential execution $u_{\neq} \in S_{\neq}$. We define $u = r(u_{\neq})$, and know by data independence of S that $u \in S$. Moreover, we can use the same bijection used for $e_{\neq} \sqsubseteq u_{\neq}$ to prove that $e \sqsubseteq u$. □

6.3 Inductively-Defined Data Structures

A *data-structure* S is given syntactically as an ordered sequence of rules $R_1, \ldots, R_n$, each of the form $u_1 \cdot u_2 \cdots u_k \in S \wedge Guard(u_1, \ldots, u_k) \Rightarrow Expr(u_1, \ldots, u_k) \in S$, where the variables u_i are interpreted over sequential executions, and

- $Guard(u_1, \ldots, u_k)$ is a conjunction of conditions on $u_1, \ldots, u_k$ with atoms
  - $u_i \in \mathbb{M}^*$ ($\mathbb{M} \subseteq \mathbb{M}$)
  - $\mathsf{matched}(m, u_i)$
- $Expr(u_1, \ldots, u_k)$ is an *expression* $e = a_1 \cdot a_2 \cdots a_l$ where
  - $u_1, \ldots, u_k$ appear in that order, exactly once, in e,
  - each a_i is either some u_j, a method m, or a Kleene closure m^* ($m \in \mathbb{M}$),
  - a method $m \in \mathbb{M}$ appears at most once in e.

We allow k to be 0 for base rules, such as $\epsilon \in S$.

A condition $u_i \in \mathbb{M}^*$ ($\mathbb{M} \subseteq \mathbb{M}$) is satisfied when the methods used in u_i are all in $\mathbb{M}$. The predicate $\mathsf{matched}(m, u_i)$ is satisfied when, for every $m(d)$ operation in u_i, there exists another operation in u_i with the same data value d.

Given a sequential execution $u = u_1 \cdot \ldots \cdot u_k$ and an expression $e = Expr(u_1, \ldots, u_k)$, we define $[\![e]\!]$ as the set of sequential executions which can be obtained from e by replacing the methods m by $m(d)$ and the Kleene closures m^* by a sequence of 0 or more $m(d)$, using the same $d \in \mathbb{D}$.

A rule $R \equiv \ u_1 \cdot u_2 \cdots u_k \in S \wedge Guard(u_1, \ldots, u_k) \Rightarrow Expr(u_1, \ldots, u_k) \in S$ is applied to a sequential execution w to obtain a new sequential execution w' from the set:

$$\bigcup_{\substack{w = w_1 \cdot w_2 \cdots w_k \wedge \\ Guard(w_1, \ldots, w_k)}} [\![Expr(w_1, \ldots, w_k)]\!]$$

We denote this $w \xrightarrow{R} w'$. The set of sequential executions $[\![S]\!] = [\![R_1, \ldots, R_n]\!]$ is then defined as the set of sequential executions w which can be derived from the empty word:

$$\epsilon = w_0 \xrightarrow{R_{i_1}} w_1 \xrightarrow{R_{i_2}} w_2 \ldots \xrightarrow{R_{i_p}} w_p = w,$$

where $i_1, \ldots, i_p$ is a non-decreasing sequence of integers from $\{1 \ldots, n\}$. This means that the rules must be applied in order, and each rule can be applied 0 or several times.

Below we give inductive definitions for the atomic queue and stack data-structures. Other data structures such as atomic registers and mutexes also have inductive definitions.

Example 12. *The queue has a method Enq to add an element to the data-structure, and a method Deq to remove the elements in a FIFO order. The method DeqEmpty can only return when the queue is empty (its parameter is not used). The only input method is Enq. Formally,* Queue *is defined by the rules* R_0, R_{Enq}, R_{EnqDeq} *and* $R_{DeqEmpty}$.

$$\begin{aligned}
R_0 &\equiv \epsilon \in \mathsf{Queue} \\
R_{Enq} &\equiv u \in \mathsf{Queue} \wedge u \in Enq^* \Rightarrow u \cdot Enq \in \mathsf{Queue} \\
R_{EnqDeq} &\equiv u \cdot v \in \mathsf{Queue} \wedge u \in Enq^* \wedge v \in \{Enq, Deq\}^* \Rightarrow Enq \cdot u \cdot Deq \cdot v \in \mathsf{Queue} \\
R_{DeqEmpty} &\equiv u \cdot v \in \mathsf{Queue} \wedge \mathit{matched}(Enq, u) \Rightarrow u \cdot DeqEmpty \cdot v \in \mathsf{Queue}
\end{aligned}$$

One derivation for Queue *is:*

$$\begin{aligned}
\epsilon \in \mathsf{Queue} &\xrightarrow{R_{EnqDeq}} Enq(1) \cdot Deq(1) \in \mathsf{Queue} \\
&\xrightarrow{R_{EnqDeq}} Enq(2) \cdot Enq(1) \cdot Deq(2) \cdot Deq(1) \in \mathsf{Queue} \\
&\xrightarrow{R_{EnqDeq}} Enq(3) \cdot Deq(3) \cdot Enq(2) \cdot Enq(1) \cdot Deq(2) \cdot Deq(1) \in \mathsf{Queue} \\
&\xrightarrow{R_{DeqEmpty}} Enq(3) \cdot Deq(3) \cdot DeqEmpty \cdot Enq(2) \cdot Enq(1) \cdot Deq(2) \cdot Deq(1) \in \mathsf{Queue}
\end{aligned}$$

Similarly, Stack *is composed of the rules* $R_0, R_{PushPop}, R_{Push}, R_{PopEmpty}$.

$$
\begin{aligned}
R_0 &\equiv \epsilon \in \mathsf{Stack} \\
R_{PushPop} &\equiv u \cdot v \in \mathsf{Stack} \wedge \mathit{matched}(Push, u) \wedge \mathit{matched}(Push, v) \wedge u, v \in \{Push, Pop\}^* \\
&\quad \Rightarrow Push \cdot u \cdot Pop \cdot v \in \mathsf{Stack} \\
R_{Push} &\equiv u \cdot v \in \mathsf{Stack} \wedge \mathit{matched}(Push, u) \wedge u, v \in \{Push, Pop\}^* \Rightarrow u \cdot Push \cdot v \in \mathsf{Stack} \\
R_{PopEmpty} &\equiv u \cdot v \in \mathsf{Stack} \wedge \mathit{matched}(Push, u) \Rightarrow u \cdot PopEmpty \cdot v \in \mathsf{Stack}
\end{aligned}
$$

The register has a method $Write$ used to write a data-value, and a method Read which returns the last written value. The only input method is $Write$. Its rules are R_0 and R_{WR}:

$$
\begin{aligned}
R_0 &\equiv \epsilon \in \mathsf{Register} \\
R_{WR} &\equiv u \in \mathsf{Register} \Rightarrow Write \cdot Read^* \cdot u \in \mathsf{Register}
\end{aligned}
$$

The mutex has a method Lock, used to take ownership of the Mutex*, and a method $Unlock$, to release it. The only input method is $Lock$. It is composed of the rules R_0, R_{Lock} and R_{LU}:*

$$
\begin{aligned}
R_0 &\equiv \epsilon \in \mathsf{Mutex} \\
R_{Lock} &\equiv Lock \in \mathsf{Mutex} \\
R_{LU} &\equiv u \in \mathsf{Mutex} \Rightarrow Lock \cdot Unlock \cdot u \in \mathsf{Mutex}
\end{aligned}
$$

In practice, $Lock$ and $Unlock$ methods do not have a parameter. Here, the parameter represents a ghost variable *which helps us relate $Unlock$ to their corresponding $Lock$. Any implementation will be data independent with respect to these ghost variables.*

We assume that the rules defining a data-structure S satisfy a non-ambiguity property stating that the last step in deriving a sequential execution in $[\![S]\!]$ is unique and it can be effectively determined. Since we are interested in characterizing the linearizations of a history and its projections, this property is extended to permutations of projections of sequential executions which are admitted by S. Thus, we assume that the rules defining a data-structure are *well-formed*, that is:

- for all $u \in [\![S]\!]$, there exists a unique rule, denoted by $\mathtt{last}(u)$, that can be used as the last step to derive u, i.e., for every sequence of rules $R_{i_1}, \ldots, R_{i_n}$ leading to u, $R_{i_n} = \mathtt{last}(u)$. For $u \notin [\![S]\!]$, $\mathtt{last}(u)$ is also defined but can be arbitrary, as there is no derivation for u.
- if $\mathtt{last}(u) = R_i$, then for every permutation $u' \in [\![S]\!]$ of a projection of u, $\mathtt{last}(u') = R_j$ with $j \leq i$. If u' is a permutation of u, then $\mathtt{last}(u') = R_i$.

Given a (completed) history h, all the u such that $h \sqsubseteq u$ are permutations of one another. The last condition of non-ambiguity thus enables us to extend the function $\mathtt{last}$ to histories: $\mathtt{last}(h)$ is defined as $\mathtt{last}(u)$ where u is any sequential execution such that $h \sqsubseteq u$. We say that $\mathtt{last}(h)$ is the rule *corresponding* to h.

Example 13. *For* Queue*, we define* $\mathtt{last}$ *for a sequential execution u as follows:*

- *if u contains a DeqEmpty operation,* $\texttt{last}(u) = R_{DeqEmpty}$,
- *else if u contains a Deq operation,* $\texttt{last}(u) = R_{EnqDeq}$,
- *else if u contains only Enq's,* $\texttt{last}(u) = R_{Enq}$,
- *else (if u is empty),* $\texttt{last}(u) = R_0$.

Since the conditions we use to define `last` *are closed under permutations, we get that for any permutation u_2 of u,* $\texttt{last}(u) = \texttt{last}(u_2)$, *and* `last` *can be extended to histories. Therefore, the rules $R_0, R_{EnqDeq}, R_{DeqEmpty}$ are well-formed.*

Definition of `last` *for a sequential execution $u \in$* Stack*:*

- *if u contains a PopEmpty operation,* $\texttt{last}(u) = R_{PopEmpty}$,
- *else if u contains an unmatched Push operation,* $\texttt{last}(u) = R_{Push}$,
- *else if u contains a Pop operation,* $\texttt{last}(u) = R_{PushPop}$,
- *else (if u is empty),* $\texttt{last}(u) = R_0$.

Definition of `last` *for a sequential execution $u \in$* Register*:*

- *if u is not empty,* $\texttt{last}(u) = R_{WR}$,
- *else,* $\texttt{last}(u) = R_0$.

Definition of `last` *for a sequential execution $u \in$* Mutex*:*

- *if u contains an Unlock operation,* $\texttt{last}(u) = R_{LU}$,
- *else if u is not empty,* $\texttt{last}(u) = R_{Lock}$,
- *else,* $\texttt{last}(u) = R_0$.

6.4 Reducing Linearizability to Reachability

Our end goal for this section is to show that for any data-independent library $\mathcal{L}$, and any specification S satisfying several conditions defined in the following, there exists a computable finite-state automaton $\mathcal{A}$ (over call and return actions) such that:

$$\mathcal{L} \sqsubseteq S \iff \mathcal{L} \cap \mathcal{A} = \varnothing$$

Then, given a model of $\mathcal{L}$, the linearizability of $\mathcal{L}$ is reduced to checking emptiness of the synchronized product between the model of $\mathcal{L}$ and $\mathcal{A}$. The automaton $\mathcal{A}$ represents (a subset of the) executions which are not linearizable with respect to S.

The first step in proving our result is to show that, under some conditions, we can partition the concurrent executions which are not linearizable with respect

to S into a finite number of classes. Intuitively, each non-linearizable execution must correspond to a violation for one of the rules in the definition of S.

We identify a property, which we call *step-by-step linearizability*, which is sufficient to obtain this characterization. Intuitively, step-by-step linearizability enables us to build a linearization for an execution e incrementally, using linearizations of projections of e.

The second step is to show that, for each class of violations (i.e. with respect to a specific rule R_i), we can build a regular automaton $\mathcal{A}_i$ such that: a) when restricted to well-formed executions, $\mathcal{A}_i$ recognizes a subset of this class; b) each non-linearizable execution has a corresponding execution, obtained by data independence, accepted by $\mathcal{A}_i$. If such an automaton exists, we say that R_i is *co-regular* (formally defined later in this section).

We prove that, provided these two properties hold, we have the equivalence mentioned above, by defining $\mathcal{A}$ as the union of the $\mathcal{A}_i$'s built for each rule R_i.

6.4.1 Reduction to a Finite Number of Violations

Our goal here is to give a characterization of the sequential executions which belong to a data-structure, as well as to give a characterization of the concurrent executions which are linearizable with respect to the data-structure. This characterization enables us to classify the linearization violations into a finite number of classes.

Our characterization relies heavily on the fact that the data-structures we consider are *closed under projection*, i.e. for all $u \in S, D \subseteq \mathbb{D}$, we have $u_{|D} \in S$. The reason for this is that the guards used in the inductive rules are closed under projection.

Lemma 25. *Any data-structure S defined in our framework is closed under projection.*

Proof. Let $u \in S$ and let $D \subseteq \mathbb{D}$. Since $u \in S$, there is a sequence of applications of rules starting from the empty word ϵ which can derive u. We remove from this derivation all the rules corresponding to a data-value $d \notin D$, and we project all the sequential executions appearing in the derivation on the D. Since the predicates which appear in the conditions are all closed under projection, the derivation remains valid, and proves that $u_{|D} \in S$. □

A sequential execution u is said to *match* a rule R with conditions $Guard$ if there exist a data value d and sequential executions $u_1, \ldots, u_k$ such that u can be written as $[\![Expr(u_1, \ldots, u_k)]\!]$, where d is the newly introduced data value, and such that $Guard(u_1, \ldots, u_k)$ holds. We call d the *witness* of the decomposition. We denote by MR the set of sequential executions which match R, and we call it the *matching set* of R.

Example 14. *MR_{EnqDeq} is the set of sequential executions of the form $Enq(d) \cdot u \cdot Deq(d) \cdot v$ for some $d \in \mathbb{D}$, and with $u \in Enq^*$.*

Lemma 26. *Let $S = R_1, \ldots, R_n$ be a data-structure and u a differentiated sequential execution. Then,*

$$u \in S \iff \mathsf{proj}(u) \subseteq \bigcup_{i \in \{1,\ldots,n\}} MR_i$$

Proof. ($\Rightarrow$) Using Lemma 25, we know that S is closed under projection. Thus, any projection of a sequential execution u of S is itself in S and has to match one of the rules $R_1, \ldots, R_n$.

($\Leftarrow$) By induction on the size of u. We know $u \in \mathsf{proj}(u)$, so it can be decomposed to satisfy the conditions *Guard* of some rule R of S. The recursive condition is then verified by induction. □

This characterization enables us to get rid of the recursion, so that we only have to check non-recursive properties. We want a similar lemma to characterize $e \sqsubseteq S$ for an execution e. This is where we introduce the notion of *step-by-step linearizability*, as the lemma will hold under this condition.

Definition 5. *A data-structure $S = R_1, \ldots, R_n$ is said be to* step-by-step linearizable *if for any differentiated execution e, if e is linearizable with respect to MR_i with witness d, we have:*

$$e \setminus d \sqsubseteq [\![R_1, \ldots, R_i]\!] \implies e \sqsubseteq [\![R_1, \ldots, R_i]\!]$$

This notion applies to the usual data-structures, as we will prove in Section 6.5 Intuitively, step-by-step linearizability will help us prove the right-to-left direction of Lemma 27 by allowing us to build a linearization for e incrementally, from the linearizations of projections of e.

Lemma 27. *Let S be a data-structure with rules $R_1, \ldots, R_n$. Let e be a differentiated execution. If S is step-by-step linearizable, we have (for any j):*

$$e \sqsubseteq [\![R_1, \ldots, R_j]\!] \iff \mathsf{proj}(e) \sqsubseteq \bigcup_{i \leq j} MR_i$$

Proof. ($\Rightarrow$) We know there exists $u \in S$ such that $e \sqsubseteq u$. Each projection e' of e can be linearized with respect to some projection u' of u, which belongs to $\bigcup_i MR_i$ according to Lemma 26.

($\Leftarrow$) By induction on the size of e. We know $e \in \mathsf{proj}(e)$ so it can be linearized with respect to a sequential execution u matching some rule R_k ($k < j$) with some witness d. Let $e' = e \setminus d$.

Since S is well-formed, we know that no projection of e can be linearized to a matching set MR_i with $i > k$, and in particular no projection of e'. Thus, we deduce that $\mathsf{proj}(e') \sqsubseteq \bigcup_{i \leq k} MR_i$, and conclude by induction that $e' \sqsubseteq [\![R_1, \ldots, R_k]\!]$.

We finally use the fact that S is step-by-step linearizable to deduce that $e \sqsubseteq [\![R_1, \ldots, R_k]\!]$ and $e \sqsubseteq [\![R_1, \ldots, R_j]\!]$ because $k < j$. □

Thanks to Lemma 27, if we're looking for an execution e which is not linearizable with respect to some data-structure S, we must prove that $\mathsf{proj}(e) \notin$

$\bigcup_i MR_i$, i.e. we must find a projection $e' \in \mathsf{proj}(e)$ which is not linearizable with respect to any MR_i ($e' \notin \bigcup_i MR_i$).

This is challenging as it is difficult to check that an execution is not linearizable with respect to a union of sets simultaneously. Using non-ambiguity, we simplify this check by making it more modular, so that we only have to check one set MR_i at a time.

Lemma 28. *Let S be a data-structure with rules $R_1, \ldots, R_n$. Let e be a differentiated execution. If S is step-by-step linearizable, we have:*

$$e \sqsubseteq S \iff \forall e' \in \mathsf{proj}(e).\ e' \sqsubseteq MR \text{ where } R = \mathtt{last}(e')$$

Proof. ($\Rightarrow$) Let $e' \in \mathsf{proj}(e)$. By Lemma 27, we know that e' is linearizable with respect to MR_i for some i. Since S is well-formed, $\mathtt{last}(e')$ is the only rule such that $e' \sqsubseteq MR$ can hold, which ends this part of the proof.

($\Leftarrow$) Particular case of Lemma 27. □

Lemma 28 gives us the finite kind of violations that we mentioned in the beginning of the section. More precisely, if we negate both sides of the equivalence, we have: $e \notin S \iff \exists e' \in \mathsf{proj}(e).\ e' \notin MR$. This means that whenever an execution is not linearizable with respect to S, there can be only finitely reasons, namely there must exist a projection which is not linearizable with respect to the matching set of its corresponding rule.

6.4.2 Regularity of Each Class of Violations

Our goal is now to construct, for each R, an automaton $\mathcal{A}$ which recognizes (a subset of) the executions e, which have a projection e' such that $e' \notin MR$. More precisely, we want the following property.

Definition 6. *A rule R is said to be* co-regular *if we can build an automaton $\mathcal{A}$ such that, for any data-independent library $\mathcal{L}$, we have:*

$$\mathcal{A} \cap \mathcal{L} \neq \varnothing \iff \exists e \in \mathcal{L}_{\neq}, e' \in \mathsf{proj}(e).\ \mathtt{last}(e') = R \wedge e' \notin MR$$

A data-structure S is co-regular *if all of its rules are co-regular.*

Formally, the alphabet of $\mathcal{A}$ is $\{\mathtt{call}\ m(d) \mid m \in \mathbb{M}, d \in D\} \cup \{\mathtt{ret}\ m(d) \mid m \in \mathbb{M}, d \in D\}$ for a finite subset $D \subseteq \mathbb{D}$. The automaton doesn't read thread identifiers, thus, when taking the intersection with $\mathcal{L}$, we ignore them. Note that we have added the method name and arguments to return events. Indeed, since the finite automaton doesn't track thread identifiers, we need to add this additional information to help it know which returns match which calls. This information is less precise than the thread identifiers, but is still enough for our purpose.

As we will show in Section 6.6, all the data-structures we defined are co-regular. When we have a data-structure which is both step-by-step linearizable and co-regular, we can make a linear time reduction from the verification of linearizability with respect to S to a reachability problem, as illustrated in Lemma 29. This is the main lemma which we will use to prove decidability of checking linearizability later on.

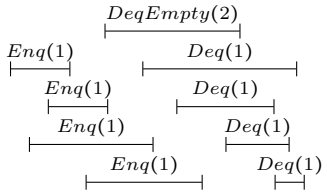


Figure 6.1: A four-pair $R_{DeqEmpty}$ violation. Lemma 42 demonstrates that this pattern with arbitrarily-many pairs is regular.

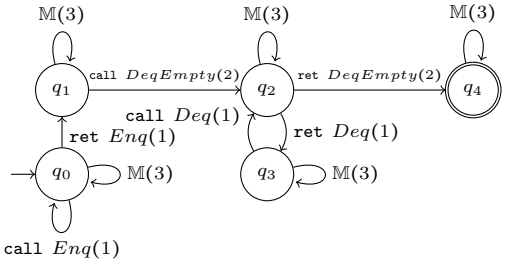


Figure 6.2: An automaton recognizing $R_{DeqEmpty}$ violations, for which the queue is non-empty, with data value 1, for the span of $DeqEmpty$. We assume all `call` $Enq(1)$ actions occur initially without loss of generality due to implementations' closure properties.

Lemma 29. *Let S be a step-by-step linearizable and co-regular data-structure and let $\mathcal{L}$ be a data-independent library. There exists a regular automaton $\mathcal{A}$ such that:*

$$\mathcal{L} \sqsubseteq S \iff \mathcal{L} \cap \mathcal{A} = \varnothing$$

Proof. Let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be the regular automata used to show that $R_1, \ldots, R_n$ are co-regular, and let $\mathcal{A}$ be the (non-deterministic) union of the $\mathcal{A}_i$'s.

($\Rightarrow$) Assume there exists an execution $e \in \mathcal{L} \cap \mathcal{A}$. For some i, $e \in \mathcal{A}_i$. From the definition of "co-regular", we deduce that there exists $e' \in \mathsf{proj}(e)$ such that $e' \notin MR_i$, where R_i is the rule corresponding to e'. By Lemma 28, e is not linearizable with respect to S.

($\Leftarrow$) Assume there exists an execution $e \in \mathcal{L}$ which is not linearizable with respect to S. By Lemma 28, it has a projection $e' \in \mathsf{proj}(e)$ such that $e' \notin MR_i$, where R_i is the rule corresponding to e'. By definition of "co-regular", this means that $\mathcal{L} \cap \mathcal{A}_i \neq \varnothing$, and that $\mathcal{L} \cap \mathcal{A} \neq \varnothing$. □

6.5 Step-by-step Linearizability

The goal of this section is to prove that all data-structures considered so far are step-by-step linearizable. More specifically, we want to prove, given a data-structure S with rules $R_1, \ldots, R_n$, that for any differentiated history h, if h is linearizable with respect to MR_i with witness x, we have:

$$h \setminus x \sqsubseteq [\![R_1, \ldots, R_i]\!] \implies h \sqsubseteq [\![R_1, \ldots, R_i]\!].$$

The proofs follow a generic schema which consists in the following: we let $u' \in [\![R_1, \ldots, R_i]\!]$ be a sequential execution such that $h \setminus d \sqsubseteq u'$ and build a graph G from u', whose acyclicity implies that $h \sqsubseteq [\![R_1, \ldots, R_i]\!]$. Then we show that we can always choose u' so that this G is acyclic.

Lemma 30. Queue, Stack, Register, *and* Mutex *are step-by-step linearizable.*

Proof. For better readability we make a sublemma per data-structure. We begin by proving that Queue is step-by-step linearizable. Concerning rule R_{EnqDeq}, our goal is to prove that, if a history h has an $Enq(d)$ which is minimal (among all operations), and a corresponding $Deq(d)$ which is minimal among all Deq operations such that $h \setminus d$ is linearizable, then h is linearizable as well.

Similarly, with rule $R_{DeqEmpty}$, we will prove that if a history h is linearizable with respect to the matching set of $R_{DeqEmpty}$, i.e. it can be linearized to $u \cdot DeqEmpty \cdot v$ – with $\mathsf{matched}(Enq, u)$ – and h without $DeqEmpty$ is linearizable with respect to Queue, then h itself is linearizable with respect to Queue.

Lemma 31. Queue *is step-by-step linearizable.*

Proof. Let h be a differentiated history, and u a sequential execution such that $h \sqsubseteq u$. We have three cases to consider:

1) u matches R_{Enq} with witness d: let $h' = h \setminus d$ and assume $h' \sqsubseteq [\![R_0, R_{Enq}]\!]$. Since u matches R_{Enq}, we know h only contain Enq operations. The set $[\![R_0, R_{Enq}]\!]$ is the set of sequential executions formed by Enq operations, which means that $h \sqsubseteq [\![R_0, R_{Enq}]\!]$.

2) u matches R_{EnqDeq} with witness d: let $h' = h \setminus d$ and assume $h' \sqsubseteq [\![R_0, R_{Enq}, R_{EnqDeq}]\!]$. Let $u' \in [\![R_0, R_{Enq}, R_{EnqDeq}]\!]$ such that $h' \sqsubseteq u'$. We define a graph G whose nodes are the operations of h and there is an edge from operation o_1 to o_2 if

1. o_1 happens-before o_2 in h,
2. o_1 is before o_2 in u',
3. $o_1 = Enq(d)$ and o_2 is any other operation,
4. $o_1 = Deq(d)$ and o_2 is any other Deq operation.

If G is acyclic, any total order compatible with G forms a sequence u_2 such that $h \sqsubseteq u_2$ and such that u_2 can be built from u' by adding $Enq(d)$ at the beginning and $Deq(d)$ before all Deq operations. Thus, $u_2 \in [\![R_0, R_{Enq}, R_{EnqDeq}]\!]$ and $h \sqsubseteq [\![R_0, R_{Enq}, R_{EnqDeq}]\!]$.

Assume that G has a cycle, and consider a cycle C of minimal size. We show that there is only one kind of cycle possible, and that this cycle can be avoided by choosing u' appropriately. Such a cycle can only contain one happens-before edge (edges of type 1), because if there were two, we could apply the interval order property to reduce the cycle. Similarly, since the order imposed by u' is a total order, it also satisfies the interval order property, meaning that C can only contain one edge of type 2.

Moreover, C can also contain only one edge of type 3, otherwise it would have to go through $Enq(d)$ more than once. Similarly, it can contain only one edge of type 4. It cannot contain a type 3 edge $Enq(d) \to o_1$ at the same time as a type 4 edge $Deq(d) \to o_2$, because we could shortcut the cycle by a type 3 edge $Enq(d) \to o_2$.

Finally, it cannot be a cycle of size 2. For instance, a type 2 edge cannot form a cycle with a type 1 edge because $h' \sqsubseteq u'$. The only form of cycles left are the two cycles of size 3 where:

- $Enq(d)$ is before o_1 (type 3), o_1 is before o_2 in u' (type 2), and o_2 happens-before $Enq(d)$: this is not possible, because h is linearizable with respect to u which matches R_{EnqDeq} with d as a witness. This means that u starts with the a $Enq(d)$ operation, and that no operation can happen-before $Enq(d)$ in h.

- $Deq(d)$ is before o_1 (type 4), o_1 is before o_2 in u' (type 2), and o_2 happens-before $Deq(d)$: by definition, we know that o_1 is a Deq operation; moreover, since h is linearizable with respect to u which matches R_{EnqDeq} with d as a witness, no Deq operation can happen-before $Deq(d)$ in h, and o_2 is an Enq operation (or Enq). Let $d_1, d_2 \in \mathbb{D}$ such that $Deq(d_1) = o_1$ and $Enq(d_2) = o_2$.

  Since o_1 is before o_2 in u', we know that d_1 and d_2 must be different. Moreover, there is no happens-before edge from o_1 to o_2, or otherwise, by transitivity of the happens-before relation, we'd have a cycle of size 2 between o_1 and $Deq(d)$.

  Assume without loss of generality that o_1 is the rightmost Deq operation which is before o_2 in u', and let $o_2^1, \ldots, o_2^s$ be the Enq (or Enq) operations between o_1 and o_2. There is no happens-before edge $o_1 < o_2^i$, because by applying the interval order property with the other happens-before edge $o_2 < Deq(d)$, we'd either have $o_1 < Deq(d)$ (forming a cycle of size 2) or $o_2 < o_2^i$ (not possible because $h' \sqsubseteq u'$ and o_2^i is before o_2 in u').

  Let u'_2 be the sequence u' where $Deq(d)$ has been moved after o_2. Since we know there is no happens-before edge from $Deq(d)$ to o_2^i or to o_2, we can deduce that: $h' \sqsubseteq u'_2$. Moreover, if we consider the sequence of deductions which proves that $u' \in [\![R_0, R_{Enq}, R_{EnqDeq}]\!]$, we can alter it when we insert the pair $Enq(d_1)$ and $o_1 = Deq(d_1)$ by inserting o_1 after the o_2^i's and after o_2, instead of before (the conditions of the rule R_{EnqDeq} allow it).

This concludes case 2), as we're able to choose u' so that G is acyclic, and prove that $h \sqsubseteq [\![R_0, R_{Enq}, R_{EnqDeq}]\!]$.

3) u matches $R_{DeqEmpty}$ with witness d: let o be the $DeqEmpty$ operation corresponding to the witness. Let $h' = h \setminus d$ and assume $h' \sqsubseteq \mathsf{Queue}$. Let L be the set of operations which are before o in u, and R the ones which are after. Let D_L be the data-values appearing in L and D_R be the data-values appearing in R. Since u matches $R_{DeqEmpty}$, we know that L contains no unmatched Enq operations.

Let $u' \in \mathsf{Queue}$ such that $h' \sqsubseteq u'$. Let $u'_L = u'_{|D_L}$ and $u'_R = u'_{|D_R}$. Since Queue is closed under projection, $u'_L, u'_R \in \mathsf{Queue}$. Let $u_2 = u'_L \cdot o \cdot u'_R$. We can show that $u_2 \in \mathsf{Queue}$ by using the derivations of u'_L and u'_R. Intuitively, this is because Queue is closed under concatenation when the left-hand sequential execution has no unmatched Enq operation, like u'_L.

Moreover, we have $h \sqsubseteq u_2$, as shown in the following. We define a graph G whose nodes are the operations of h and there is an edge from operation o_1 to o_2 if

1. o_1 happens-before o_2 in h,
2. o_1 is before o_2 in u_2.

Assume there is a cycle in G, meaning there exists o_1, o_2 such that o_1 happens-before o_2 in h, but the corresponding operations are in the opposite order in u_2.

- If $o_1, o_2 \in L$, or $o_1, o_2 \in R$, this contradicts $h' \sqsubseteq u'$.
- If $o_1 \in R$ and $o_2 \in L$, this contradicts $h \sqsubseteq u$.
- If $o_1 \in R$ and $o_2 = o$, or if $o_1 = o$ and $o_2 \in L$, this contradicts $h \sqsubseteq u$.

This shows that $h \sqsubseteq u_2$. Thus, we have $h \sqsubseteq \mathsf{Queue}$ and concludes the proof that the Queue is step-by-step linearizable. □

Proving that Stack is step-by-step linearizable can be done like for the rule $R_{DeqEmpty}$ of Queue. The idea is again to combine two linearizations of subhistories into a linearization for the full history h.

Lemma 32. Stack *is step-by-step linearizable.*

Proof. Let h be a differentiated history, and u a sequential execution such that $h \sqsubseteq u$. We have three cases to consider:

1) (very similar to case 3 of the Queue) u matches $R_{PushPop}$ with witness d: let a and b be respectively the Push and Pop operations corresponding to the witness. Let $h' = h \setminus d$ and assume $h' \sqsubseteq [\![R_{PushPop}]\!]$. Let L be the set of operations which are before b in u, and R the ones which are after. Let D_L be the data-values appearing in L and D_R be the data-values appearing in R. Since u matches $R_{PushPop}$, we know that L contains no unmatched Push operations.

Let $u' \in [\![R_{PushPop}]\!]$ such that $h' \sqsubseteq u'$. Let $u'_L = u'_{|D_L}$ and $u'_R = u'_{|D_R}$. Since $[\![R_{PushPop}]\!]$ is closed under projection, $u'_L, u'_R \in [\![R_{PushPop}]\!]$. Let $u_2 = a \cdot u'_L \cdot b \cdot u'_R$. We can show that $u_2 \in [\![R_{PushPop}]\!]$ by using the derivations of u'_L and u'_R.

Moreover, we have $h \sqsubseteq u_2$, because if the total order of u_2 didn't respect the happens-before relation of u_2, it could only be because of four reasons, all leading to a contradiction:

- the violation is between two L operations or two R operations, contradicting $h' \sqsubseteq u'$
- the violation is between a L and an R operation, contradicting $h \sqsubseteq u$
- the violation is between b and another operation, contradicting $h \sqsubseteq u$
- the violation is between a and another operation contradicting $h \sqsubseteq u$

This shows that $h \sqsubseteq [\![R_{PushPop}]\!]$ and concludes case 1.

2) u matches R_{Push} with witness d: similar to case 1

3) u matches $R_{PopEmpty}$ with witness d: identical to case 3 of the Queue $\square$

Proving step-by-step linearizability for Register and Mutex is easier than for Stack or Queue, as the rules involved in the definitions are simpler.

Lemma 33. Register *is step-by-step linearizable.*

Proof. Let h be a differentiated history, and u a sequential execution such that $h \sqsubseteq u$ and such that u matches the rule R_{WR} with witness d. Let a and $b_1, \ldots, b_s$ be respectively the $Write$ and $Read$'s operations of h corresponding to the witness.

Let $h' = h \setminus d$ and assume $h' \sqsubseteq [\![R_{WR}]\!]$. Let $u' \in [\![R_{WR}]\!]$ such that $h' \sqsubseteq u'$. Let $u_2 = a \cdot b_1 \cdot b_2 \cdots b_s \cdot u'$. By using rule R_{WR} on u', we have $u_2 \in [\![R_{WR}]\!]$. Moreover, we prove that $h \sqsubseteq u_2$ by contradiction. Assume that the total order imposed by u_2 doesn't respect the happens-before relation of h. All three cases are not possible:

- the violation is between two u' operations, contradicting $h' \sqsubseteq u'$,
- the violation is between a and another operation, i.e. there is an operation o which happens-before a in h, contradicting $h \sqsubseteq u$,
- the violation is between some b_i and a u' operation, i.e. there is an operation o which happens before b_i in h, contradicting $h \sqsubseteq u$.

Thus, we have $h \sqsubseteq u_2$ and $h \sqsubseteq [\![R_{WR}]\!]$, which ends the proof. $\square$

Lemma 34. Mutex *is step-by-step linearizable.*

Proof. Identical to the Register proof, expect there is only one Unlock operation (b), instead of several Read operations ($b_1, \ldots, b_s$). $\square$

$\square$

6.6 Co-Regularity

Our goal in that section is to prove that each rule R we considered is co-regular, meaning, we can build an automaton $\mathcal{A}$ such that, for any data-independent library $\mathcal{L}$, we have:

$$\mathcal{A} \cap \mathcal{L} \neq \varnothing \iff \exists e \in \mathcal{L}_{\neq}, e' \in \mathsf{proj}(e).\ \mathtt{last}(e') = R \wedge e' \notin MR.$$

We have a generic schema to build the automaton, which is first to characterize a violation by the existence of a cycle of some kind, and then build an automaton recognizing such cycles. For some of the rules, we prove that these cycles can always be bounded, thanks to a *small model property*. For the others, even though the cycles can be unbounded, we can still build an automaton.

We prove the co-regularity of R_{EnqDeq} and $R_{DeqEmpty}$ respectively in Section 6.6.1 and Section 6.6.2. The two rules require different approaches, but all other rules we consider will look like one of these two. We will then explain the similarities with the rules of Stack in Section 6.6.3.

6.6.1 Co-Regularity of R_{EnqDeq}

Our approach in this section is to prove a small model property for the rule R_{EnqDeq}. More precisely, we want to prove that when a history is not linearizable with respect to the matching set of R_{EnqDeq}, then it has a *small* projection which not linearizable either. We can then build an automaton which only recognizes the small violations.

Lemma 35. *Given a history h, if $\forall d_1, d_2 \in \mathbb{D}_h$, $h_{|\{d_1,d_2\}} \sqsubseteq MR_{EnqDeq}$, then $h \sqsubseteq MR_{EnqDeq}$.*

Note: Lemmas 36, 37, 38 are part of the proof of Lemma 35.

Proof. We first identify constraints which are sufficient to prove $h \sqsubseteq MR_{EnqDeq}$.

Lemma 36. *Let h be a history and d_1 a data value of $\mathbb{D}_h$. If $Enq(d_1) \not\curvearrowright Deq(d_1)$, and for all operations o, we have $Enq(d_1) \not\curvearrowright o$, and for all Deq operations o, we have $Deq(d_1) \not\curvearrowright o$, then h is linearizable with respect to MR_{EnqDeq}*

Proof. We define a graph G whose nodes are the element of h, and whose edges include both the happens-before relation as well as the constraints depicted given by the Lemma. G is acyclic by assumption and any total order compatible with G corresponds to a linearization of h which is in MR_{EnqDeq}. □

Given $d_1, d_2 \in \mathbb{D}_h$, we denote by d_1 $\mathbf{W_{h,MR}}$ d_2 the fact that $h_{|\{d_1,d_2\}}$ is linearizable with respect to MR, by using d_1 as a witness. We reduce the notation to d_1 $\mathbf{W}$ d_2 when the context is not ambiguous.

First, we show that if the same data value d_1 can be used as a witness for all projections of size 2, then we can linearize the whole history (using this same data value as a witness).

Lemma 37. *For $d_1 \in \mathbb{D}_h$, if $\forall d \neq d_1$, d_1 $\mathbf{W}$ d, then $h \sqsubseteq MR_{EnqDeq}$.*

Proof. Since $\forall d \neq d_1$, d_1 $\mathbf{W}$ d, the happens-before relation of h respects the constraints given by *Lemma* 36, and we can conclude that $h \sqsubseteq MR_{EnqDeq}$. □

Next, we show the key characterization, which enables us to reduce non-linearizability with respect to MR_{EnqDeq} to the existence of a cycle in the $\not\mathbf{W}$ relation.

Lemma 38. *If $h \notin MR_{EnqDeq}$, then h has a cycle $d_1 \not\mathbf{W} d_2 \not\mathbf{W} \ldots \not\mathbf{W} d_p \not\mathbf{W} d_1$*

Proof. Let $d_1 \in \mathbb{D}_h$. By Lemma 37, we know there exists $d_2 \in \mathbb{D}_h$ such that $d_1 \mathrel{\mathcal{W}} d_2$. Likewise, we know there exists $d_3 \in \mathbb{D}_h$ such that $d_2 \mathrel{\mathcal{W}} d_3$. We continue this construction until we form a cycle. □

We can now prove the small model property. Assume $h \notin R$. By Lemma 38, it has a cycle $d_1 \mathrel{\mathcal{W}} d_2 \mathrel{\mathcal{W}} \dots \mathrel{\mathcal{W}} d_p \mathrel{\mathcal{W}} d_1$. If there exists a data-value d such that $Deq(x)$ happens-before $Enq(x)$, then $h_{|\{d\}} \notin R_{EnqDeq}$, which contradicts our assumptions.

For each i, there are two possible reasons for which $d_i \mathrel{\mathcal{W}} d_{(i \; mod \; p)+1}$. The first one is that $Enq(d_i)$ is not minimal in the subhistory of size 2 (reason (a)). The second one is that Deq_{d_i} is not minimal with respect to the Deq operations (reason (b)).

We label each edge of our cycle by either (a) or (b), depending on which one is true (if both are true, pick arbitrarily). Then, using the interval order property, we have that, if $d_i \mathrel{\mathcal{W}} d_{(i \; mod \; p)+1}$ for reason (a), and $d_j \mathrel{\mathcal{W}} d_{(j \; mod \; p)+1}$ for reason (a) as well, then either $d_i \mathrel{\mathcal{W}} d_{(j \; mod \; p)+1}$, or $d_j \mathrel{\mathcal{W}} d_{(i \; mod \; p)+1}$ (for reason (a)). This enables us to reduce the cycle and leave only one edge for reason (a).

The same applies for reason (b). This allows us to reduce the cycle to a cycle of size 2 (one edge for reason (a), one edge for reason (b)). If d_1 and d_2 are the two data-values appearing in the cycle, we have: $h_{|\{d_1,d_2\}} \notin R_{EnqDeq}$, which is what we wanted to prove. □

Having the small model property, we can build a regular automaton which recognizes each of the small violation, to prove that indeed rule R_{EnqDeq} is co-regular.

Lemma 39. *The rule R_{EnqDeq} is co-regular.*

Proof. We proved in Lemma 35 that a differentiated history h has a projection h' such that $\mathtt{last}(h') = R_{EnqDeq}$ and $h' \notin MR_{EnqDeq}$ if and only if it has such a projection on 1 or 2 data-values. Violations of histories with two values are: *i*) there is a value d such that $Deq(d)$ happens-before $Enq(d)$ (or $Enq(d)$ doesn't exist in the history) or *ii*) there are two operations $Deq(d)$ in h or, *iii*) there are two values d and y such that $Enq(d)$ happens-before $Enq(y)$, and $Deq(y)$ happens-before $Deq(d)$ ($Deq(d)$ doesn't exist in the history).

The automaton $\mathcal{A}_{R_{EnqDeq}}$ in Fig 6.3 recognizes all such small violations (top-left branch for *i*, top-right branch for *ii*, bottom branch for *iii*).

Let $\mathcal{L}$ be any data-independent implementation. We show that

$$\mathcal{A}_{R_{EnqDeq}} \cap \mathcal{L} \neq \varnothing \iff \exists e \in \mathcal{L}_{\neq}, e' \in \mathsf{proj}(e). \quad \mathtt{last}(e') = R_{EnqDeq} \wedge e' \notin MR_{EnqDeq}$$

($\Rightarrow$) Let $e \in \mathcal{L}$ be an execution which is accepted by $\mathcal{A}_{R_{EnqDeq}}$. By data independence, let $e_{\neq} \in \mathcal{L}$ and r a renaming such that $e = r(e_{\neq})$, and assume without loss of generality that r doesn't rename the data-values 1 and 2. If e is accepted by one of the top two branches of $\mathcal{A}_{R_{EnqDeq}}$, we can project $e_{\neq}$ on value

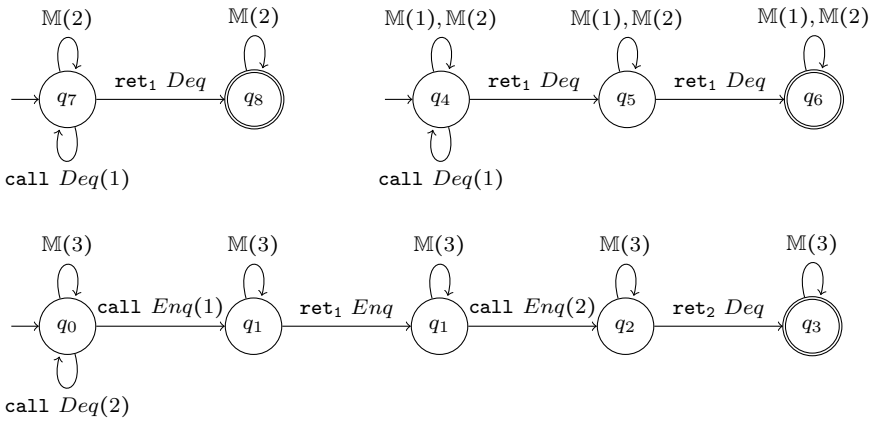


Figure 6.3: A non-deterministic automaton recognizing R_{EnqDeq} violations. The top-left branch recognizes executions which have a Deq with no corresponding Enq. The top-right branch recognizes two Deq's returning the same value, which is not supposed to happen in a differentiated execution. The bottom branch recognizes FIFO violations. By the closure properties of libraries, we can assume the $\texttt{call}\ Deq(2)$ are at the beginning.

1 to obtain a projection e' such that $\texttt{last}(e') = R_{EnqDeq}$ and $e' \notin MR_{EnqDeq}$. Likewise, if e is accepted by the bottom branch, we can project $e_{\neq}$ on $\{1, 2\}$, and obtain again a projection e' such that $\texttt{last}(e') = R_{EnqDeq}$ and $e' \notin MR_{EnqDeq}$.

($\Leftarrow$) Let $e_{\neq} \in \mathcal{L}_{\neq}$ such that there is a projection e' such that $\texttt{last}(e') = R_{EnqDeq}$ and $e' \notin MR_{EnqDeq}$. As recalled at the beginning of the proof, we know $e_{\neq}$ has to contain a violation of type i, ii, or ii. If it is of type i or ii, we define the renaming r, which maps d to 1, and all other data-values to 2. The execution $r(e_{\neq})$ can then be recognized by one of the top two branches of $\mathcal{A}_{R_{EnqDeq}}$ and belongs to $\mathcal{L}$ by data independence.

Likewise, if it is of type iii, r will map d to 1, and y to 2, and all other data-values to 3, so that $r(e_{\neq})$ can be recognized by the bottom branch of $\mathcal{A}_{R_{EnqDeq}}$. □

6.6.2 Co-Regularity of $R_{DeqEmpty}$

As opposed to R_{EnqDeq}, the rule $R_{DeqEmpty}$ doesn't have a small model property. Yet, we show that we can still define a regular automaton to recognize violations. We first define the notion of *gap*, which intuitively corresponds to a point in an execution where the Queue could be empty.

Definition 7. *Let h be a differentiated history and o an operation of h. We say that h has a* gap on operation o *if there is a partition of the operations of h into $L \uplus R$ satisfying:*

- *L has no unmatched Enq operation, and*
- *no operation of R happens-before an operation of L or o, and*
- *no operation of L happens-after o.*

Then, we show that the absence of a gap on a *DeqEmpty* operation exactly characterizes violations to $MR_{DeqEmpty}$.

Lemma 40. *A differentiated history h has a projection h' such that* $\texttt{last}(h') = R_{DeqEmpty}$ *and $h' \notin MR_{DeqEmpty}$ if and only there exists a DeqEmpty operation o in h such that there is no gap on o.*

Proof. ($\Rightarrow$) Assume there exists a projection h' such that $\texttt{last}(h') = R_{DeqEmpty}$ and $h' \notin MR_{DeqEmpty}$. Let o be a *DeqEmpty* operation in h' (exists by definition of `last`).

Assume by contradiction that there is a gap on o. By the properties of the gap, we can linearize h' into a sequential execution $u \cdot o \cdot v$ where u and v respectively contain the L and R operations of the partition.

($\Leftarrow$) Assume there exists a *DeqEmpty* operation o in h such that there is no gap on o. Let h' be the projection which contains all the operations of h as well as o, except the other *DeqEmpty* operations.

Assume by contradiction that there exists a sequential execution $w \in MR_{DeqEmpty}$ such that $h' \sqsubseteq w$. By definition of $MR_{DeqEmpty}$, w can be decomposed into $u \cdot o \cdot v$ such that u has no unmatched operation. Let L be the operations of u, and R the operation of v. Since $h' \sqsubseteq w$, the partition $L \uplus R$ forms a gap on operation o. □

We exploit the characterization of Lemma 40 by showing how we can recognize the existence of gaps in the next two lemmas. First, we define the notion of *left-right constraints* of an operation, and show that these constraints have a solution if and only if there is a gap on the operation.

Definition 8. *Let h be a distinguished history, and o an operation of h. The left-right constraints of o is the graph G where:*

- *the nodes are $\mathbb{D}_h$, the data-values of h, to which we add a node for o,*
- *there is an edge from data-value d_1 to o if $Enq(d_1)$ happens-before o,*
- *there is an edge from o to data-value d_1 if o happens-before $Deq(d_1)$,*
- *there is an edge from data-value d_1 to d_2 if $Enq(d_1)$ happens before $Deq(d_2)$.*

The next lemma shows that we can characterize the existence of a gap with the absence of a cycle in the leftleft-right constraints. We will then use this characterization to prove co-regularity of $R_{DeqEmpty}$ by making an automaton recognizing such cycles.

Lemma 41. *Let h be a differentiated history and o an operation of h. Let G be the graph representing the left-right constraints of o. There is a gap on o if and only if G has no cycle going through o.*

Proof. ($\Rightarrow$) Assume that there is a gap on o, and let $L \uplus R$ be a partition corresponding to the gap. Assume by contradiction there is a cycle $d_p \to \cdots \to d_1 \to o \to d_p$ in G (which goes through o). By definition of G, and since $o \to d_p$, and by definition of a gap, we know that all operations with data-value d_p must be in R. Since $d_p \to d_{p-1}$, the operations with data-value d_{p-1} must be in R as well. We iterate this reasoning until we deduce that d_1 must be in R, contradicting the fact that $d_1 \to o$.

($\Leftarrow$) Assume there is no cycle in G going through o. Let L be the set of operations having a data-value d which has a path to o in G, and let R be the set of other operations. By definition of the left-right constraints G, the partition $L \uplus R$ forms a gap for operation o. □

Corollary 4. *A differentiated history h has a projection h' such that $\texttt{last}(h') = R_{DeqEmpty}$ and $h' \notin MR_{DeqEmpty}$ if and only if it has a DeqEmpty operation o and data-values $d_1, \ldots, d_p \in \mathbb{D}_h$ such that:*

- *$Enq(d_1)$ happens-before o in h, and*
- *$Enq(d_i)$ happens before $Deq(d_{i-1})$ in h for $i > 1$, and*
- *o happens-before $Deq(d_p)$, or $Deq(d_p)$ doesn't exist in h.*

We say that o is covered *by $d_1, \ldots, d_p$.*

Proof. By definition of the left-right constraints, and following from Lemmas 40 and 41. □

Using Corollary 4, we can now make an automaton to recognize the absence of a gap on a *DeqEmpty* operation.

Lemma 42. *The rule $R_{DeqEmpty}$ is co-regular.*

Proof. We proved in Corollary 4 that a history has a projection such that $\texttt{last}(h') = R_{DeqEmpty}$ and $h' \notin MR_{DeqEmpty}$ if and only if it has a *DeqEmpty* operation which is *covered* by other operations, as depicted in Fig 6.1. The automaton $\mathcal{A}_{R_{DeqEmpty}}$ in Fig 6.2 recognizes such violations.

Let $\mathcal{L}$ be any data-independent implementation. We show that

$$\mathcal{A}_{R_{DeqEmpty}} \cap \mathcal{L} \neq \varnothing \iff \exists e \in \mathcal{L}_{\neq}, e' \in \mathsf{proj}(e).\ \texttt{last}(e') = R_{DeqEmpty} \wedge e' \notin MR_{DeqEmpty}$$

($\Rightarrow$) Let $e \in \mathcal{L}$ be an execution which is accepted by $\mathcal{A}_{R_{DeqEmpty}}$. By data independence, let $e_{\neq} \in \mathcal{L}$ and r a renaming such that $e = r(e_{\neq})$. Let $d_1, \ldots, d_p$ be the data values which are mapped to value 1 by r.

Let d be the data value which is mapped to value 2 by r. Let o the *DeqEmpty* operation with data value d. By construction of the automaton we can prove

that o is covered by $d_1, \ldots, d_p$, and using Corollary 4, conclude that h has a projection such that $\texttt{last}(h') = R_{DeqEmpty}$ and $h' \notin MR_{DeqEmpty}$.

($\Leftarrow$) Let $e_{\neq} \in \mathcal{L}_{\neq}$ such that there is a projection e' such that $\texttt{last}(e') = R_{DeqEmpty}$ and $e' \notin MR_{DeqEmpty}$. Let $d_1, \ldots, d_p$ be the data values given by Corollary 4, and let d be the data value corresponding to the $DeqEmpty$ operation.

Without loss of generality, we can always choose the cycle so that $Enq(d_i)$ doesn't happen before $Deq(d_{i-2})$ (if it does, drop d_{i-1}).

Let r be the renaming which maps $d_1, \ldots, d_p$ to 1, d to 2, and all other values to 3. Let $e = r(e_{\neq})$. The execution e can be recognized by automaton $\mathcal{A}_{R_{DeqEmpty}}$, and belongs to $\mathcal{L}$ by data independence. □

6.6.3 Co-Regularity of the Stack rules

The Stack rule $R_{PushPop}$ is very similar to the $R_{DeqEmpty}$ rule of the Stack. Using the notion of *gap*, we can also give a characterization for a violation with respect to $MR_{PushPop}$.

Lemma 43. *A differentiated history h has a projection h' such that $\texttt{last}(h') = R_{PushPop}$ and $h' \notin MR_{PushPop}$ if and only if there exists a projection such that $\texttt{last}(h') = R_{PushPop}$ and either*

- *there exists an unmatched $Pop(d)$ operation in h', or*
- *there is a $Pop(d)$ which happens-before $Push(d)$ in h', or*
- *for all $Push(d)$ operations minimal in h', there is no gap on $Pop(d)$ in $h' \setminus d$.*

Proof. Can be proved as in Lemma 40. □

Lemma 44. *A differentiated history h has a projection h' such that $\texttt{last}(h') = R_{PushPop}$ and $h' \notin MR_{PushPop}$ if and only if either:*

- *there exists an unmatched $Pop(d)$ operation, or*
- *there is a $Pop(d)$ which happens-before $Push(d)$, or*
- *there exist a data-value $d \in \mathbb{D}_h$ and data-values $d_1, \ldots, d_p \in \mathbb{D}_h$ such that*
  - *$Push(d)$ happens-before $Push(d_i)$ for every i,*
  - *$Pop(d)$ is covered by $d_1, \ldots, d_p$.*

Proof. ($\Leftarrow$) We have three cases to consider

- there exists an unmatched $Pop(d)$ operation: define $h' = h_{|\{d\}}$,
- there is a $Pop(d)$ which happens-before $Push(d)$: define $h' = h_{|\{d\}}$,
- there exist a data-value $d \in \mathbb{D}_h$ and data-values $d_1, \ldots, d_p \in \mathbb{D}_h$ such that

– $Push(d)$ happens-before $Push(d_i)$ for every i

– $Pop(d)$ is covered by $d_1, \ldots, d_p$.

Define $h' = h_{|\{d,d_1,\ldots,d_p\}}$. We have $\texttt{last}(h') = R_{PushPop}$ because h' doesn't contain $PopEmpty$ operations nor unmatched Push operations. Assume by contradiction that $h' \sqsubseteq MR_{PushPop}$, and let $w \in MR_{PushPop}$ such that $h' \sqsubseteq u$. Since $Push(d)$ happens-before $Push(d_i)$ (for every i) the witness d of $w \in MR_{PushPop}$ has to be the data-value d. This means that $w = Push(d) \cdot u \cdot Pop(d) \cdot v$ for some u and v with no unmatched $Push$.

Thus, there is a gap on operation $Pop(d)$ in $h' \setminus d$, and that $Pop(d)$ cannot be covered by $d_1, \ldots, d_p$.

($\Rightarrow$) Let h' be a projection of h such that $\texttt{last}(h') = R_{PushPop}$ and $h' \notin MR_{PushPop}$. Assume there are no unmatched $Pop(d)$ operation, and that for every d, $Pop(d)$ doesn't happens-before $Push(d)$. This means that h' is made of pairs of $Push(d)$ and $Pop(d)$ operations.

Let $Push(d)$ be a Push operation which is minimal in h'. We know there is one, because we assumed that $\texttt{last}(h') = R_{PushPop}$, and we know that there is a Push which is minimal because for every d, $Pop(d)$ doesn't happens-before $Push(d)$.

By Lemma 43, we know that there is no gap on $Pop(d)$. Similarly to Lemma 41 and Corollary 4, we deduce that there are data-values $d_1, \ldots, d_p \in \mathbb{D}_{h'}$ such that $Pop(d)$ is covered by $d_1, \ldots, d_p$. Our goal is now to prove that we can choose d and $d_1, \ldots, d_p$ such that, besides these properties, we also have that $Push(d)$ happens-before $Push(d_i)$ for every i. Assume there exists i such that $Push(d)$ doesn't happen-before $Push(d_i)$. We have two cases, either $Pop(d)$ is covered by $d_1, \ldots, d_{i-1}, d_{i+1}, \ldots, d_p$, in which case we can just get rid of d_i; or this is not the case, and we can choose our new d to be d_i and remove d_i from the list of data-values. We iterate this until we have a data-value $d \in \mathbb{D}_h$ such that

- $Push(d)$ happens-before $Push(d_i)$ for every i,
- $Pop(d)$ is covered by $d_1, \ldots, d_p$.

□

Lemma 45. *The rule $R_{PushPop}$ is co-regular.*

Proof. The automaton Fig 6.4 recognizes the violations given by Lemma 44. The proof is then similar to Lemma 42. □

The rules R_{Push} and $R_{PopEmpty}$ can also be proven co-regular using the same techniques.

Lemma 46. *The rule R_{Push} is co-regular.*

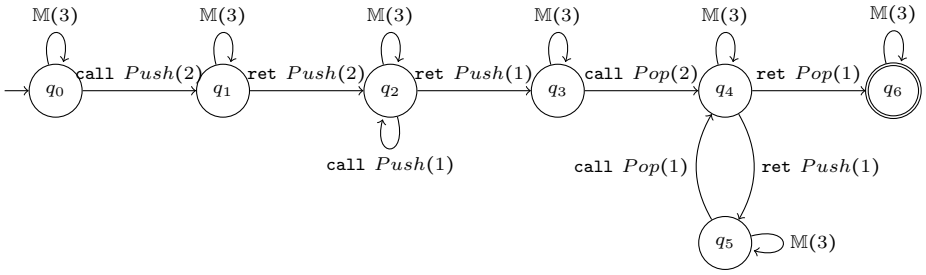


Figure 6.4: An automaton recognizing $R_{PushPop}$ violations. Here we have a $Push(2)$ operation, whose corresponding $Pop(2)$ operation is covered by $Push(1)/Pop(1)$ pairs. The $Push(2)$ happens-before all the pairs. Intuitively, the element 2 cannot be popped from the Stack there is always at least an element 1 above it in the Stack (regardless of how linearize the execution).

Proof. We can make a characterization of the violations similar to Lemma 44. This rule is in a way simpler, because the $Push$ in this rule plays the role of the Pop in $R_{PushPop}$. □

Lemma 47. *The rule $R_{PopEmpty}$ is co-regular.*

Proof. Identical to Lemma 42 (replace Enq by $Push$, Deq by Pop, and $DeqEmpty$ by $PopEmpty$). □

6.7 Decidability and Complexity

Lemma 29 implies that the linearizability problem with respect to any step-by-step linearizable and co-regular specification is decidable for any data-independent library for which checking the emptiness of the intersection with finite-state automata is decidable. Here, we give a class $\mathcal{C}$ of data-independent libraries for which the latter problem, and thus linearizability, is decidable.

The libraries in $\mathcal{C}$ are finite-state libraries, enriched with a finite number of variables storing data values from $\mathbb{D}$, as well as one variable to store the argument of the method, which is itself in $\mathbb{D}$. Data values may be copied from one variable to the other, but their value cannot be used to branch in the program, using a `read` command. This class captures typical implementations, or finite-state abstractions thereof, e.g., obtained via predicate abstraction.

Let $\mathcal{L}$ be a library from class $\mathcal{C}$. All the automata $\mathcal{A}$ constructed to prove co-regularity use only data values 1, 2, and 3. Checking emptiness of $\mathcal{L} \cap \mathcal{A}$ is thus equivalent to checking emptiness of $\mathcal{L}_3 \cap \mathcal{A}$ with the three-valued library $\mathcal{L}_3 = \{e \in \mathcal{L} \mid e = e_{|\{1,2,3\}}\}$. The set $\mathcal{L}_3$ can be represented by a VASS as shown in Lemma 7.

Emptiness of the intersection with regular automata reduces to the EXPSPACE-complete state-reachability problem for VASS. Limiting verification to a bounded

number of threads lowers the complexity of coverability to PSPACE. The hardness part of Theorem 9 comes from the hardness of state reachability in libraries.

Theorem 9. *Verifying linearizability of a library in $\mathcal{C}$ with respect to a step-by-step linearizable and co-regular specification is* PSPACE*-complete for a bounded number of threads, and* EXPSPACE*-complete otherwise.*

Proof. Decidability and complexity follow from Lemmas 7 and 29, while hardness can be derived as in Lemma 6. □

6.8 Summary

We have demonstrated a linear-time reduction from linearizability with respect to specifications to state reachability, and the application of this reduction to atomic queues, stacks, registers, and mutexes. Besides yielding novel decidability results, our reduction enables the use of existing safety-verification tools for linearizability. Moreover, this technique can be used both for proving linearizability and for finding violations, unlike static linearizability and bounding the interval length.

While this work only applies the reduction to these four objects, we believe our methodology can be applied to other data structures. Although this methodology currently does not capture priority queues, which are not data independent, we think our approach can be extended to include them. We leave this for future work.

Chapter 7

Weaker Consistency Criteria

7.1 Introduction

Online services often use a technique called *replication* to improve the latency for their users, sometimes connecting from places all around the world. Instead, the content of the data structure is replicated on several *sites*, or *nodes*. Thus, when a client makes a call to a method, the site which is the closest geographically will respond, resulting in a low latency operation. Replication can however bring inconsistencies, as it is not possible to synchronize all nodes instantaneously, and it is not possible to implement linearizability on a partition tolerant and available network [25].

As a result, designers of such systems sacrifice linearizability (and observational refinement, as defined in Section 2.5), and settle for weaker consistency criteria such as *eventual consistency* and *causal consistency*, which can be implemented using weaker synchronization mechanisms. Works investigating the formal definition of eventual consistency (and correctness criteria for distributed data structures in general) are still very rare. To our knowledge, Burckhardt et al. [13] is the first attempt to provide a formal framework for reasoning about eventual consistency.

However, very little is known about the formal guarantees given to clients using libraries satisfying these criteria, similarly to how the notion observational refinement defined in Section 2.5 is guaranteed by linearizability. We are also not aware of any work aimed at the automatic verification of these criteria. This chapter will focus first on giving formal definitions for these criteria, which can be applied to a wide class of distributed libraries, namely systems that use speculative executions and roll-backs. Then, we study the complexity of the verification problems for these criteria when the number of sites is bounded, and show that only the weakest one, eventual consistency, is decidable.

In its simplest formulation[1], eventual consistency requires that if the clients stop submitting operations, then all the sites (in a message-passing setting, we

[1] Also called, quiescent consistency [30]

use the notion of *sites* or *nodes*, instead of threads as in the shared memory setting) will eventually reach a consistent state (i.e. they agree on the way operations should be executed). However, as mentioned in Burckhardt et al. [13], this formulation is too loose and the reason is twofold. First, this definition does not impose some notion of correctness for the operations executed by the system, i.e., the fact that they should satisfy some well-defined specification. Second, this property offers no guarantees when the system never stops, i.e. when the clients continuously submit new operations. For that, eventual consistency should take into account infinite executions of the system involving infinitely many operations.

Eventual consistency is defined as the composition of a safety property that specifies the correct effects of the operations, and a liveness property guaranteeing that sites will eventually agree on the order in which the operations should be executed. Let us look closer to both of these components.

Safety: The return value of a method called on some site N depends on (1) the operations received and scheduled by N before o, and the order in which these operations are executed, (2) the conflict detection and conflict resolution policies applied by N, and (3) the behavior of the executed operations.

An execution is called *safe* if and only if the return values of all its operations are correct in a sense described hereafter. Like linearizability, the correctness is defined with respect to a *specification* that, roughly, models the expected outcome of executing a poset of operations on a single site. Concretely, a poset of operations models a schedule, where incomparable elements are considered to be in conflict (i.e. submitted concurrently to different sites) and executing a poset of operations involves the actual implementations of the operations together with the conflict resolution policy (that defines the effect of concurrently submitted operations). A specification S associates return values of operations with posets of operations. Intuitively, the return value r of an operation o is associated with some poset ρ if o returns r whenever it is executed after the poset of operations ρ.

More precisely, an execution e is safe with respect to a specification S if and only if for each operation o, there exists a poset $li[o]$ of operations in e, which is associated by S with the return value of o. The poset $li[o]$ is called the *local interpretation* of o. Additionally, because of physical constraints, we define an *executed-before* relation eb over the operations in the execution such that $(o', o) \in eb$ if and only if o' belongs to the local interpretation of o, and we require that the union of eb with the program order relation is an acyclic relation. Concretely, $(o', o) \in eb$ if and only if o is an operation submitted to some site N and o' is scheduled by N before o while the local interpretation $li[o]$ models the result of applying the scheduling and conflict detection policies.

Note that, for distributed libraries that use speculative executions and rollbacks, the local interpretations associated with two operations can be arbitrarily different, even if the two operations are submitted to the same site. (The issue of convergence is left to the liveness part.)

Liveness: The liveness part of eventual consistency requires that there exists

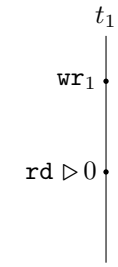


Figure 7.1: Times goes from top to bottom. An execution which is safe, but not RYW consistent. The read operation is ignoring the write which was done on the same site t_1.

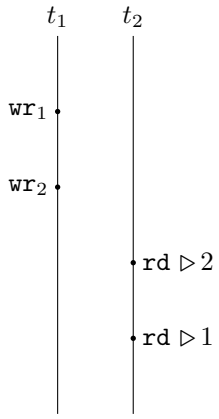


Figure 7.2: An execution which is RYW consistent, but not FIFO consistent. The write operations done on t_1 are received in the wrong order in t_2.

some partial order relation gi (for global interpretation) over all the operations in an infinite execution, such that it is possible to choose some local interpretations satisfying the safety property, which *converge* towards gi. The convergence is formally stated as follows: for any prefix P of gi (a prefix of a poset is a restriction of the poset to a downward closed subset) there exists only finitely many local interpretations for which P is not a prefix. This corresponds to the informal definition given in Saito and Shapiro [46].

The safety part of eventual consistency is sometimes too weak for some applications. For instance, a user might want to have the guarantee that all previous methods executed on the same node are taken into account. The consistency criterion which ensures this property is called *read-your-writes (RYW) consistency*. For instance, the execution depicted in Fig 7.1 is not RYW consistent.

FIFO consistency is yet stronger, as it imposes that whenever an operation takes into account an operation on another site, it must also take into account all previous operations. Fig 7.2 gives an example of an execution which is RYW consistent but not FIFO consistent.

Finally, we describe causal consistency. Intuitively, it imposes that operations which are causally related must be executed by every site in that order. Operations which are done on the same site are always causally related, and thus causal consistency is stronger than FIFO consistency. Fig 7.3 gives an execution which is FIFO consistent, but not causally consistent. We give two variants of

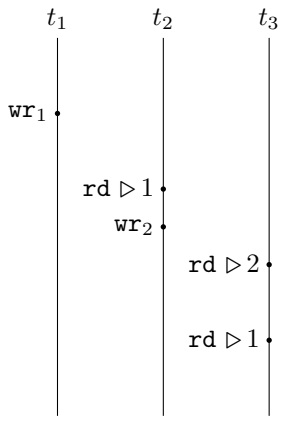


Figure 7.3: An execution which is FIFO consistent, but not causally consistent. As t_2 reads value 1 before writing value 2, the operations $\mathtt{wr}_1$ and $\mathtt{wr}_2$ operations are causally related. Yet, site t_3 sees them in the wrong order.

causal consistency. One called *explicit causally consistency*, where the causal constraints are imposed by the exchange of messages between the nodes of the library, and one called *implicit causally consistency*, which is weaker, and ignores the actual exchange of messages in the execution, but ensures that there exists a causal order which explains the return values observed in the execution.

All of these safety criteria (RYW, FIFO, and causal consistency), can also be combined with the liveness condition described for eventual consistency, so as to ensure that all sites converge towards the same order of operations.

Concerning verification, we show that eventual consistency is decidable when the number of sites is bounded. However, we prove that all the safety conditions, stronger than the safety part of eventual consistency, are undecidable. To prove the undecidability results, we use the same methodology as for the EXPSPACE-hardness of linearizability. We introduce problems over formal languages which we show undecidable, and prove that they can be reduced to the verification of those consistency criteria.

We show that any criterion which is stronger than RYW consistency but weaker than implicit causally consistency is undecidable. This is based on the undecidability of the containment problem of weighted automata over the semiring $(\mathbb{N}, \mathsf{min}, +)$. The containment problem asks, given two weighted automata W_A and W_B, whether for every word w, the weight of w in W_A is larger or equal than the weight of w in W_B.

In addition, we prove that any criterion stronger than FIFO consistency but weaker than explicit causally consistency is also undecidable. This range covers criteria where not all operations and/or not all messages impose causal constraints. Explicit causal consistency is one extreme, where every exchange of message imposes causal constraints, while the only causality constraints respected by FIFO consistency are the one due to the program order of each node.

To prove undecidability, we introduce a problem using the *shuffling* operator $\|$ over words (the shuffle of two words is the language of all words which can be obtained by interleaving the two words in an arbitrary manner), and show that

it is undecidable. Given regular languages A, B, and L, it asks whether:

$$\exists u \in A, v \in B.\ u \| v \cap L = \varnothing.$$

.

These results leave open the possibility that there exists a criterion stronger than RYW consistency, weaker than explicit causally consistency, and incomparable to both implicit causally consistency and FIFO consistency, for which verification is decidable. Yet, they give a good picture on the decidability limits for consistency criteria weaker than linearizability.

To summarize, the contributions of these chapter are:

- a new framework for formalizing consistency criteria, and its application to eventual consistency, RYW consistency, FIFO consistency, and causal consistency (Section 7.3),
- reduction of the safety and liveness part of eventual consistency to standard model-checking problems (Sections 7.4 and 7.5),
- a model which can be used to describe data structure specifications composed of posets (Section 7.6),
- a proof of decidability for verifying that a finite-state library satisfies eventual consistency with respect to this class of specifications, based on the reductions to model-checking problems mentioned above (Section 7.7),
- a proof of undecidability for all criteria stronger than RYW consistency and weaker than implicit causally consistency (Section 7.8),
- a proof of undecidability for all criteria stronger than FIFO consistency and weaker than explicit causally consistency (Section 7.9),
- problems over regular or weighted automata which are equivalent to these criteria, providing a better understanding of their complexity.

7.2 Modeling Distributed Libraries

7.2.1 Syntax

In the concurrent setting, libraries used a memory shared by all the threads. In the distributed setting, the notion of thread is replaced by the one of *node* or *site*. Each node of a *distributed library* has its own local memory, and there is no global shared memory. Nodes instead communicate by sending messages through channels.

In order to simplify the presentation, we assume that when a method is called on some node, it immediately returns a value based on its local state, and then broadcasts a message to the other nodes. This way we won't have call and return actions like in the shared memory setting, and will only focus

on the message-passing aspect of the distributed libraries. This simplification is motivated by the fact that we assume that nodes are *available*. Informally, this means that a client calling a method on a node should never have to wait for a response. Moreover, we will assume for ease of presentation that methods do not take arguments, or equivalently, that arguments are encoded in the method name.

We consider that messages are made of code, which executes on the node upon reception. Formally, each node is in a state $q \in Q$, and a *message* is a encoded as a function from Q to Q, which updates the local state of a node upon reception. Thus, we define the set of messages Msg as the set of functions $Q \to Q$.

A *distributed library* $\mathcal{L}$ associates to each method $m \in \mathbb{M}$ a function from $q \in Q$ to a subset of $Q \times \mathbb{D} \times \mathsf{Msg}$, which describes, in every possible state q a node can be, what can happen when a method m is called. More precisely, if $(q', d, \mathsf{msg}) \in \mathcal{L}(m)(q)$, then when $m \in \mathbb{M}$ is called on a node $t \in \mathbb{T}$ which is in state $q \in Q$, it can return the value $d \in \mathbb{D}$, update the local state of t to q', and broadcast the message $\mathsf{msg} \in Q \to Q$ to all other nodes. When the message is received by another node, the function is applied to its local state, in order to update it. We impose that $\mathcal{L}(m)(q)$ is never empty, as a method can always be called, and this set can contain more than one element, to represent non-determinism.

A distributed library is *finite-state* if the sets Q, $\mathbb{M}$, and $\mathbb{D}$ are finite.

7.2.2 Semantics

We now describe the semantics of a distributed library $\mathcal{L}$ using an LTS L. For this, we assume that the methods of the library can be called on any site $t \in \mathbb{T}$, arbitrarily. This corresponds to the notion of *most general client*. Messages are broadcast through unbounded unordered channels. This is the usual assumption in large-scale networks such as the Internet.

Let Mid be an infinite set of *message identifiers*. These are used to match *send actions* with corresponding *receive actions*. Each time a message is sent, a fresh message identifier $\mathsf{msgid} \in \mathsf{Mid}$ is attached to the message. A configuration of the LTS L is a pair (μ, chan) where $\mu : \mathbb{T} \to Q$ is a map associating to each node t the state it is in, and $\mathsf{chan} : \mathbb{T} \times \mathbb{T} \to \mathcal{P}(\mathsf{Mid} \times \mathsf{Msg})$ is a map describing the messages with identifiers contained in each unbounded unordered channel. In the initial configuration cfg_i, all the nodes are in state q_i and all the channels are empty.

We describe the possible steps between two configurations $\mathsf{cfg} = (\mu, \mathsf{chan})$ and $\mathsf{cfg}' = (\mu', \mathsf{chan}')$, noted $\mathsf{cfg} \to \mathsf{cfg}'$, in Fig. 7.4. The *execution* e of $\mathcal{L}$ is the sequence of labels of a run of the corresponding LTS L. The set of all executions of $\mathcal{L}$ is denoted by $[\![\mathcal{L}]\!]$. The set of executions using at most $k \in \mathbb{N}$ distinct site identifiers is denoted by $[\![\mathcal{L}]\!]^k$. When clear from context, we will abuse notations and denote these two sets of executions respectively by $\mathcal{L}$ and $\mathcal{L}^k$.

The labels of the form $(t, !(\mathsf{msgid}, \mathsf{msg}))$ and of the form $(t_2, ?(\mathsf{msgid}, \mathsf{msg}))$ are respectively called *send* and *receive actions*, while the ones of the form

$$\textsc{Method call and Broadcast}\quad \frac{\begin{array}{c}(q', d, \mathsf{msg}) \in \mathcal{L}(m)(q) \\ \mu(t_1) = q \qquad \mu' = \mu[t_1 \mapsto q'] \\ \forall t_2 \neq t_1.\ \mathsf{chan}'(t_1, t_2) = \mathsf{chan}(t_1, t_2) \cup \{(\mathsf{msgid}, \mathsf{msg})\} \\ \forall t \in t.\ \mathsf{chan}'(t_2, t) = \mathsf{chan}(t_2, t)\end{array}}{(\mu, \mathsf{chan}) \xrightarrow{(t, m \triangleright d) \cdot (t, !(\mathsf{msgid}, \mathsf{msg}))} (\mu', \mathsf{chan}')}$$

$$\textsc{Receive action}\quad \frac{\begin{array}{c}\mathsf{msg}(q) = q' \qquad \mu(t_2) = q \qquad \mu' = \mu[t_2 \mapsto q'] \\ \mathsf{chan}(t_1, t_2) = \{(\mathsf{msgid}, \mathsf{msg})\} \cup C \qquad \mathsf{chan}' = \mathsf{chan}[(t_1, t_2) \mapsto C]\end{array}}{(\mu, \mathsf{chan}) \xrightarrow{(t_2, ?(\mathsf{msgid}, \mathsf{msg}))} (\mu', \mathsf{chan}')}$$

Figure 7.4: The transition relation between configurations.

$(t, m \triangleright d)$ are called *client actions*. An operation o is a particular instance of a client action in the execution. An operation corresponding to a client action $(t, m \triangleright d)$ is called an $m \triangleright d$ operation. We denote by O_e the set of operations of e.

We define the *po-abstraction* of an execution as being the $\mathbb{M} \times \mathbb{D}$ labeled poset (O, po, ℓ) where O is the set of operations of e, $po \subseteq O \times O$ is the order such that any two operations submitted to different sites are incomparable and the order between any two operations submitted to the same site is consistent with the order in which these two operations appear in e and ℓ labels each $m \triangleright d$ operation by $m \triangleright d$.

The *causal order co* between the actions of an execution is defined as being the smallest partial order (closed under transitivity) containing the following:

- two actions done by the same site are ordered in the same order as they appear in the execution,
- a send action $!(\mathsf{msgid}, \mathsf{msg})$ is ordered before all of its corresponding receive actions $?(\mathsf{msgid}, \mathsf{msg})$ when they exist (one per site).

Since operations correspond to client actions of an execution, we extend *co* to order operations. We define the *causal-abstraction* of an execution as being the tuple (O, po, co, ℓ) where (O, po, ℓ) is the po-abstraction of the execution, and $co \subseteq O \times O$ is the causal order on the operations.

7.2.3 Examples

Next, we give examples of the $\mathbb{M}$ and $\mathbb{D}$ sets for several distributed libraries used throughout the paper.

Example 15 (One-Value Register)**.** *The One-Value Register (Register) maintains an integer register and supports the set of methods* $\mathbb{M}_{\mathsf{R}} = \{\mathtt{wr}_i \mid i \in \mathbb{N}\} \cup \{\mathtt{rd}\}$*, where* $\mathtt{wr}_i$ *assigns value* i *to the register and* `rd` *reads the current value of the register. A method* `rd` *can return any value from* $\mathbb{N}$ *while the methods* $\mathtt{wr}_i$ *return some special value* $\top$*, i.e., the domain of return values is* $\mathbb{D}_{\mathsf{R}} = \mathbb{N} \cup \{\top\}$*.*

Example 16 (Multi-Value Register)**.** *The Multi-Value Register [18, 32] (MV-Register) maintains an integer register and supports the same set of methods as the Register. A method* `rd` *can return any* set *of values from* $\mathbb{N}$*. Thus, the domain of return values of the MV-Register is* $\mathbb{D}_{\mathsf{MV\text{-}R}} = \mathcal{P}(\mathbb{N}) \cup \{\top\}$*.*

Example 17 (Observed-Remove Set)**.** *The Observed-Remove Set [48] (OR-Set) maintains a set of integers over which one can apply the set of methods* $\mathbb{M}_{\mathsf{OR\text{-}S}} = \{\mathtt{add}_i, \mathtt{rem}_i, \mathtt{rem}_i \mid i \in \mathbb{N}\}$*, where* $\mathtt{add}_i$ *adds the integer* i *to the set,* $\mathtt{rem}_i$ *removes* i *from the set, and* $\mathtt{rem}_i$ *tests if the integer* i *is in the set. We assume that the methods* `add` *and* `rem` *return some fixed value* $\top$*, while* $\mathtt{rem}_i$ *can return 1 or 0. Thus, the set of return values is* $\mathbb{D}_{\mathsf{OR\text{-}S}} = \{1, 0, \top\}$*.*

7.3 Consistency Criteria Definitions

In this section, we introduce formal definitions for eventual consistency and causal consistency, whose main artifacts are presented hereafter.

Specification: In the context of shared-memory, the correctness of the operations associated to some object involves some mechanism of conflict resolution in order to define the effect of a set of operations executed by different threads. As we saw, this is specified by the notion of linearizability. In the context of distributed libraries, where each site maintains its own copy of the object, some more general mechanisms of conflict resolution are required.

To specify both the sequential semantics of the operations and the conflict resolution mechanisms, we use posets labeled by methods instead of sequences of methods, as in the case of linearizable objects. We don't use a total order because, in general, it is unfeasible that all sites agree at all time on a total order of operations, and sometimes even unnecessary, in case of commutative operations for instance.

We assume that the specification can't distinguish between two posets that are identical (isomorphic) when ignoring the identities and the return values of the operations. The insensitivity to return values is motivated by the fact that, in real implementations, the sites exchange operations without their return values (in such systems, it is not expected that an operation returns the same value when executed at different sites).

Thus, a specification associates to each pair $m \triangleright d$ a set of $\mathbb{M}$ labeled posets closed under isomorphism. The fact that a poset ρ is associated to $m \triangleright d$ means that any site that sees the methods in ρ in that order should reach a state where the call to m produces the value d.

Definition 9 (Specification)**.** *A* specification $\mathcal{S} : \mathbb{M} \times \mathbb{D} \to \mathcal{P}(\mathsf{PoSet}_{\mathbb{M}})$ *is a function where for each* $m \in \mathbb{M}$, $d \in \mathbb{D}$, $\mathcal{S}(m \rhd d)$ *is a set of* $\mathbb{M}$ *labeled posets closed under isomorphism.*

We give several examples of specifications for the replication systems mentioned in the previous section.

Example 18 (Register specification)**.** *The specification* $\mathcal{S}_{\mathsf{R}}$ *of the Register is given by: (1) for every* i, $\mathcal{S}_{\mathsf{R}}(\mathtt{wr}_i \rhd \top)$ *is the set of all* $\mathbb{M}_{\mathsf{R}}$ *labeled totally-ordered sets, and (2) for every* i, $\mathcal{S}_{\mathsf{R}}(\mathtt{rd} \rhd i)$ *is the set of all* $\mathbb{M}_{\mathsf{R}}$ *labeled totally-ordered sets where the maximal element labeled by a write is labeled by* $\mathtt{wr}_i$*. Figure 7.5a contains two examples of totally-ordered sets in* $\mathcal{S}_{\mathsf{R}}(\mathtt{rd} \rhd 0)$*.*

Example 19. *[MV-Register specification] The specification* $\mathcal{S}_{\mathsf{MV\text{-}R}}$ *of the MV-Register is defined by: (1) for every* i, $\mathcal{S}_{\mathsf{MV\text{-}R}}(\mathtt{wr}_i \rhd \top)$ *is the set of all* $\mathbb{M}_{\mathsf{R}}$ *labeled posets and (2) for every* I, $\mathcal{S}_{\mathsf{MV\text{-}R}}(\mathtt{rd} \rhd I)$ *is the set of all* $\mathbb{M}_{\mathsf{R}}$ *labeled posets* ρ *such that* $i \in I$ *iff there exists a maximal element labeled by* $\mathtt{wr}_i$ *in the projection of* ρ *over the elements labeled by write methods* $\{\mathtt{wr}_i \mid i \in \mathbb{N}\}$*. Figure 7.5b contains an* $\mathbb{M}_{\mathsf{R}}$ *labeled poset in* $\mathcal{S}_{\mathsf{R}}(\mathtt{rd} \rhd \{0, 1, 2\})$*.*

Example 20. *[OR-Set specification] The specification* $\mathcal{S}_{\mathsf{OR\text{-}S}}$ *is given by: (1) for every* i, $\mathcal{S}_{\mathsf{OR\text{-}S}}(\mathtt{add}_i \rhd \top)$ *and* $\mathcal{S}_{\mathsf{OR\text{-}S}}(\mathtt{rem}_i \rhd \top)$ *are the set of all* $\mathbb{M}_{\mathsf{OR\text{-}S}}$ *labeled posets, and (2) for every* i, $\mathcal{S}_{\mathsf{OR\text{-}S}}(\mathtt{rem}_i \rhd 1)$*, resp.,* $\mathcal{S}_{\mathsf{OR\text{-}S}}(\mathtt{rem}_i \rhd 0)$*, is the set of all* $\mathbb{M}_{\mathsf{OR\text{-}S}}$ *labeled posets* ρ *such that the projection of* ρ *over the elements labeled by* $\mathtt{add}_i$ *or* $\mathtt{rem}_i$ *contains a maximal element labeled by* $\mathtt{add}_i$*, resp., contains no maximal element labeled by* $\mathtt{add}_i$*. Figure 7.5c contains an example of a labeled poset that belongs to both* $\mathcal{S}_{\mathsf{OR\text{-}S}}(\mathtt{rem}_1 \rhd 1)$ *and* $\mathcal{S}_{\mathsf{OR\text{-}S}}(\mathtt{rem}_0 \rhd 0)$*.*

Local interpretation: The return value of some operation o submitted to some site N depends on the set of operations applied at N before o and on the effect of applying the scheduling and conflict detection policies over this set of operations. Taken together they can be represented by a poset of operations, called the *local interpretation* of o and denoted by $li[o]$. Because of speculative executions, one has to consider a local interpretation for each operation o in the execution (the order in which known operations are executed can change at any time).

The local interpretations define another relation over the operations in the execution, called *executed-before* and denoted by eb. We say that some operation o' is executed before another operation o, i.e. $(o', o) \in eb$, iff o' belongs to the local interpretation of o. For example, Figure 7.6a pictures an execution of the Register, where the arrows define a possible eb relation. Note that the $\mathtt{wr}_1$ operation is executed before the first occurrence of $\mathtt{rd} \rhd 0$ but not before the second occurrence of $\mathtt{rd} \rhd 0$.

We say that the return value of some $m \rhd d$ operation o is *correct* iff the labeled poset defined by $li[o]$, where every operation o' is labeled by $\mathsf{meth}(o')$, belongs to $S(m \rhd d)$. Then, an execution e is *safe* if and only if the return values of all operations in e are correct.

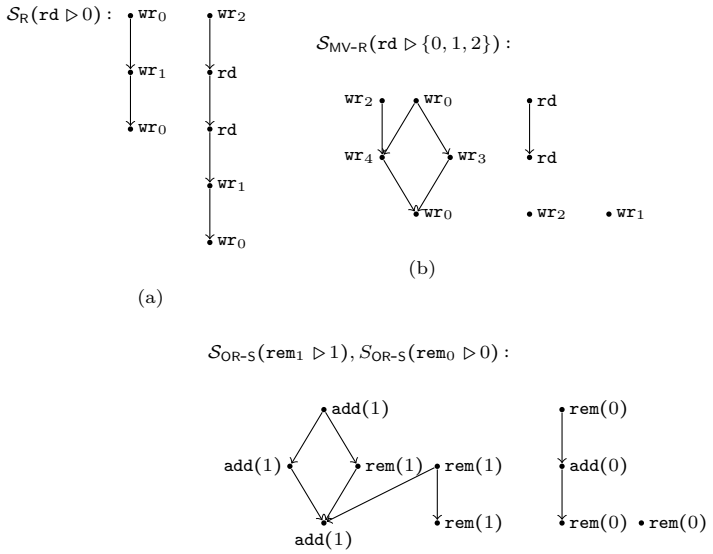


Figure 7.5: Examples of labeled posets belonging to (a) the specification of the Register (b) the specification of the MV-Register, and (c) the specification of the OR-Set. The order relations are defined by arrows: an arrow from an element x to some element y means that x is ordered before y. We omit arrows implied by transitivity.

For example, in the case of the first $\mathtt{rd} \triangleright 0$ operation in Figure 7.6a, one can choose to order the $\mathtt{wr}_1$ operation before the $\mathtt{wr}_0$ operation. This local interpretation defines an $\mathbb{M}_{\mathsf{R}}$ labeled poset, which belongs to the specification $S_{\mathsf{R}}(\mathtt{rd} \triangleright 0)$ of the Register. Similarly, one can show that all return values in Figure 7.6a are correct.

Because of physical constraints, *eb* must not create cycles together with the program order. For example, the execution in Figure 7.6b could be one of the Register if the relation *eb* is defined by the arrows in the figure (we assume that the initial value of the register is 0). However, this means that the site executing the operations in the left received a message from the other site containing the $\mathtt{wr}_1$ operation and this message was created after $\mathtt{rd} \triangleright 2$ has finished. Thus, in real time, $\mathtt{rd} \triangleright 2$ has happened before the $\mathtt{wr}_2$ operation, which contradicts the *eb* relation.

Global interpretation: The fact that the sites eventually agree on the way operations should be executed is defined as a liveness property over infinite executions of the system. Given an infinite execution e, we consider a partial-order over all the operations in e, called the *global interpretation* and denoted by gi. The liveness property requires that the local interpretations defined for

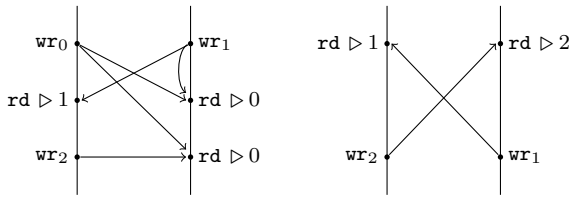


Figure 7.6: Po-abstractions of the Register. For simplicity, the return values of the `wr` operations and the ids of each operation are omitted. The program order is defined by the vertical lines, from top to bottom.

the operations in e *converge* towards gi. More precisely, for any finite prefix P of gi, the number of local interpretations for which P *is not* a prefix is finite. Note that this implies that any operation o in the execution is executed before all operations in τ, except some finite set. A system that satisfies this property is called *weak eventually consistent*.

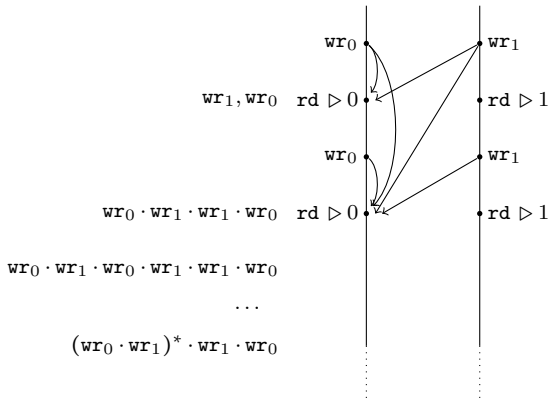


Figure 7.7: The po-abstraction of an infinite execution of Register, where one site executes $(\mathtt{wr}_0 \cdot \mathtt{rd} \triangleright 0)^\omega$ and another site executes $(\mathtt{wr}_1 \cdot \mathtt{rd} \triangleright 1)^\omega$.

We require that gi satisfies some sanity condition, i.e. that it is prefix-founded which means that there cannot be infinitely many operations executing before an operation. For example, let us consider the infinite execution in Fig 7.7. We consider a global interpretation gi, which in this case is a total order, that arranges the write operations in a sequence of the form $(\mathtt{wr}_0 \cdot \mathtt{wr}_1)^\omega$, and keeps the same order as in the execution for the $\mathtt{wr}_0$ operations (resp., the $\mathtt{wr}_1$ operations). In the following, we ignore the order between the `rd` operations because they are not important for justifying the correctness of the return values. We show that it is possible to choose local interpretations for the $\mathtt{rd} \triangleright 0$ operations such that the return values are correct, and such that the orders converge towards gi.

In a similar way, this can be shown for all the other operations in the execution.

For each $\texttt{rd} \triangleright 0$ operation o, the local interpretation $li[o]$ is the poset that consists of all the write operations that occur before o (i.e. if o is the ith occurrence of $\texttt{rd} \triangleright 0$ then the poset $li[o]$ contains the jth occurrence of $\texttt{wr}_1$ and $\texttt{wr}_0$, for all $j < i$), totally ordered in a sequence of the form $(\texttt{wr}_0 \cdot \texttt{wr}_1)^* \cdot \texttt{wr}_1 \cdot \texttt{wr}_0$ consistent with the program order. The relation eb is pictured by arrows in Figure 7.7. Defined as such, the local interpretations converge towards the global interpretation gi.

We now give the definition of eventual consistency. For any poset $li[o] = (O, <)$ as above, $li_{\mathbb{M}}[o]$ denotes the labeled poset $li[o]$ where every operation is labeled by its the corresponding method name in $\mathbb{M}$.

Definition 10. *[Safety, Eventual Consistency, RYW Consistency, FIFO Consistency, and Causal Consistency] An execution with po-abstraction (O, po, ℓ) is called* eventually consistent *with respect to a specification S iff:*

$$\begin{array}{l} \exists\ gi \text{ an partial order over } O \\ \quad \forall\ o \in O\ \exists\ li[o] \text{ a poset.} \\ \quad\quad \textsc{GIpf} \wedge \textsc{ThinAir} \wedge \textsc{Rval} \wedge \textsc{Eventual} \end{array}$$

It is said to be weak eventually consistent *with respect to S when axiom* Eventual *is replaced by* WeakEventual *in the condition above (thus, gi and* GIpf *can be removed). It is said to be* safe *with respect to S if and only if the axioms* ThinAir *and* Rval *are satisfied, i.e.,*

$$\begin{array}{l} \forall\ o \in O\ \exists\ li[o] \text{ a poset.} \\ \quad \textsc{ThinAir} \wedge \textsc{Rval} \end{array}$$

It is RYW consistent *with respect to S if*

$$\begin{array}{l} \forall\ o \in O\ \exists\ li[o] \text{ a poset.} \\ \quad \textsc{ThinAir} \wedge \textsc{Rval} \wedge \textsc{RywWF} \wedge \textsc{Ryw} \end{array}$$

It is FIFO consistent *with respect to S if*

$$\begin{array}{l} \forall\ o \in O\ \exists\ li[o] \text{ a poset.} \\ \quad \textsc{ThinAir} \wedge \textsc{Rval} \wedge \textsc{FifoWF} \wedge \textsc{FifoWF} \end{array}$$

Finally, an execution with causal-abstraction (O, po, co, ℓ) is called implicitly causally consistent *with respect to S iff*

$$\begin{array}{l} \forall\ o \in O\ \exists\ li[o] \text{ a poset.} \\ \quad \textsc{ThinAir} \wedge \textsc{Rval} \wedge \textsc{Causal} \wedge \textsc{CausalWF} \end{array}$$

and it is said to be explicitly causally consistent *with respect to S when*

$$\begin{array}{l} \forall\ o \in O\ \exists\ li[o] \text{ a poset.} \\ \quad \textsc{ThinAir} \wedge \textsc{Rval} \wedge \textsc{Causal} \wedge \textsc{CausalWF} \wedge co = eb \end{array}$$

meaning that we must whose the $li[o]$ posets so that the relation eb coincides with the causality order co.

Remark 3. *Using these axioms, we can also define* causal consistency with convergence *[5] as being causal consistency augmented with the axioms* Eventual *and* GIpf. *However, we won't discuss this criterion from a verification perspective as it is at least as difficult as causal consistency (for finite executions), and we will show that causal consistency is undecidable. The axioms* Eventual *and* GIpf *can also be added to the other safety criteria (RYW and FIFO consistency) to ensure convergence towards a global order for all sites.*

The relation eb is defined by: $(o', o) \in eb$ iff $o' \in li[o]$

ThinAir	$eb \cup po$ is acyclic
Rval	for all (m, d)-operation $o \in O$, $li_{\mathbb{M}}[o] \in S(m, d)$
RywWF	$po \subseteq eb$
Ryw	for all operations o in site t, $po^t_{\lvert eb^{-1}(o)} \subseteq loc_o$
FifoWF	$po \subseteq eb$ and $po \circ eb \subseteq eb$
Fifo	for all operations o, $po_{\lvert eb^{-1}(o)} \subseteq loc_o$
CausalWF	$(eb \cup po)^+ = eb$
Causal	for all operations o, $eb_{\lvert eb^{-1}(o)} \subseteq loc_o$
GIpf	gi is prefix-founded
WeakEventual	for all $o \in O, \{o' \mid (o, o') \notin eb\}$ is finite
Eventual	for any finite prefix P of the poset (O, gi), $\{o \mid P \not\leq li[o]\}$ is finite

Table 7.1: The list of axioms used in Definition 10.

Axioms ThinAir and Rval are thus safety conditions, and ensure that the operations respect the specification S. When augmented with the axioms RywWF and Ryw, they ensure that an operation must always take into account the previous operations from its site, and in the order they were executed. The axioms FifoWF and Fifo ensure that when an operation takes into account an operation from another site, it must also take into account all previous operations from the other site, and it must always consider the operation in an order compatible with the program order. Finally, the axioms CausalWF and Causal ensure that the operation are always executed respecting the causality order co.

Concerning the liveness, axiom WeakEventual ensures that eventually, every operation will be executed before all the other operations in the system, meaning there are only finitely many operations which can ignore it. Axiom Eventual is a stronger liveness condition which ensures that all nodes eventually agree on a (possibly partial) order in which to execute all the operations.

We lift the definition of consistency criteria to distributed libraries $\mathcal{L}$ by requiring that they hold for all executions. In the next sections, we consider the problem of verifying these criteria, and we assume that $\mathbb{M}$ and $\mathbb{D}$ are finite.

7.4 Safety

7.4.1 Notations

In this chapter, the *Parikh image* of a possibly infinite Σ-labeled poset $\rho = (A, \leq, \ell)$ is the vector $\Pi_\Sigma(\rho) : \Sigma \to \mathbb{N} \cup \{\omega\}$ mapping each symbol $a \in \Sigma$ to the number of elements x of A such that $\ell(x) = a$. If some symbol a occurs infinitely often then $\Pi_\Sigma(\sigma)(a) = \omega$. The notion is extended to sets of labeled posets as usual. By an abuse of notation, we use $\Pi_{\mathbb{M}}(e)$ even when e is an execution; in that case, the map counts for each method m the number of $m \triangleright d$ operations (for some $d \in \mathbb{D}$) in e.

Remark 4. *The Parikh image defined here is different than the one defined for interval orders in Chapter 5.*

For vectors $f : A \to \mathbb{N} \cup \{\omega\}$, and $g : A' \to \mathbb{N} \cup \{\omega\}$ with $A' \subseteq A$, $f + g : A \to \mathbb{N} \cup \{\omega\}$ maps each $a' \in A'$ to $(f + g)(a') = f(a') + g(a')$ and each $a \in A \setminus A'$ to $(f + g)(a) = f(a)$. The addition over integers is extended to ω so that ω plus any integer (or itself) equals ω.

7.4.2 Characterization

We consider the problem of checking that a distributed library is safe and we prove that it can be reduced to a reachability problem. First, we show that in any unsafe execution e, one can find a $m \triangleright d$ operation o such that there aren't sufficiently many operations before o in e to construct a labeled poset belonging to its specification $\mathcal{S}(m \triangleright d)$. For example, for an unsafe execution of the Register, there will always exist a read that returns a value not written by a previous write.

This allows us to define a monitor for checking the safety of a distributed library $\mathcal{L}$, that records all the operations executed by $\mathcal{L}$, and stops with a negative answer at any time that it detects a $m \triangleright d$ operation o for which, with the recorded set of operations, it cannot build a labeled poset belonging to the specification of $m \triangleright d$. Actually, we prove that it is sufficient that the monitor only counts the number of times the system executes each of the methods in $\mathbb{M}$ (until some bound) and then, compare the counter values with the minimal vectors in the Parikh image of the specification.

We show that if e is an unsafe execution, it has a prefix which ends in a $m \triangleright d$ operation whose return value is not correct, i.e. it is not possible to define a labeled poset that contains only operations in this prefix that belongs to the specification of $m \triangleright d$. This is stated formally in the following lemma.

Lemma 48. *Given a distributed library $\mathcal{L}$, the following are equivalent:*

1. *there exists an execution $e \in \mathcal{L}$ which is not safe,*
2. *there exists an execution $e = e' \cdot (N, m \triangleright d) \in \mathcal{L}$ such that there exists no poset $(V_o, <_o)$ whose elements are a subset of $O_{e'}$, such that $(V_o, <_o, \mathsf{meth}) \in \mathcal{S}(m \triangleright d)$.*

Proof. (2) ⇒ (1) Assume that (2) holds for $e = e' \cdot (N, m \rhd d)$. Let o be the last operation of e, i.e. the one corresponding to the last $(N, m \rhd d)$. If e was safe, we would have a local interpretation $li[o]$ such that $li_{\mathbb{M}}[o] \in \mathcal{S}(m \rhd d)$. Note that the operations in $li[o]$ belong to $O_{e'}$, which contradicts (2). Thus, e is not safe and (2) implies (1).

(1) ⇒ (2) Let $e = e' \cdot (N, m \rhd d) \in \mathcal{L}$ be a minimal execution which is not safe. Let o be the last operation of e, corresponding to $(N, m \rhd d)$. Assume there exists a poset $(V_o, <_o)$ whose elements are a subset of $O_{e'}$, such that $(V_o, <_o, \mathsf{meth}) \in \mathcal{S}(m \rhd d)$. By a minimality argument, we know e' is safe, and we can use $(V_o, <_o)$ to prove that e itself is safe, without creating cycles in $eb \cup po$. □

The following corollary is a reformulation of Lemma 48 in terms of Parikh images.

Corollary 5. *A distributed library $\mathcal{L}$ is not safe iff there exists $e' \cdot (N, m \rhd d) \in \mathcal{L}$ such that $\Pi_{\mathbb{M}}(e') \notin \uparrow \Pi_{\mathbb{M}}(\mathcal{S}(m \rhd d))$.*

Here, we use the notation $\uparrow$ to denote the upward-closure of a set of vectors, where the comparison of vectors is done component-wise, i.e. a vector over $\mathbb{N} \cup \{\omega\}$ is smaller than another one if all of its components are (with ω being greater than every integer).

Given an execution e and an integer i, $\Pi^i_{\mathbb{M}}(e)$ is the Parikh image of e, where all the components larger than i are set to i, that is $\Pi^i_{\mathbb{M}}(e) = (m \in \mathbb{M} \mapsto min(i, \Pi_{\mathbb{M}}(e)(m)))$.

For each $m \in \mathbb{M}$, $d \in \mathbb{D}$, let $V_{m \rhd d}$ be the set of minimal elements of $\Pi_{\mathbb{M}}(\mathcal{S}(m \rhd d))$ (with respect to the ordering relation over vectors of natural numbers), so that $\uparrow \Pi_{\mathbb{M}}(\mathcal{S}(m \rhd d)) = \uparrow V_{m \rhd d}$, and let $i_{m \rhd d}$ be the maximum value appearing in the vectors of $V_{m \rhd d}$. Let $i = \max\{i_{m \rhd d} \mid m \in \mathbb{M}, d \in \mathbb{D}\}$.

For $a = m \rhd d$ with $m \in \mathbb{M}$, $d \in \mathbb{D}$, we remark that $\Pi_{\mathbb{M}}(e') \notin \uparrow \Pi_{\mathbb{M}}(\mathcal{S}(a))$ is equivalent to the fact that $\Pi_{\mathbb{M}}(e')$ is *not* greater than one of the minimal elements $v_a \in V_a$. Moreover, since all the components of the vectors of V_a are smaller than i, $\Pi_{\mathbb{M}}(e') \notin \uparrow \Pi_{\mathbb{M}}(\mathcal{S}(a))$ is equivalent to $\bigwedge_{v_a \in V_a} \Pi^i_{\mathbb{M}}(e') \not\geq v_a$.

In general, the sets $V_{m \rhd d}$ cannot be computed, but in Section 7.6, we give a class of specifications for which they can. We define a monitor $\mathcal{M}_{\mathtt{safe}}$, which counts all the methods it sees up to the bound i, and every time it reads a symbol $(N, m \rhd d)$, it goes to an error state $q_{\mathtt{err}}$ iff the vector of methods seen is not larger than some $v_{m \rhd d} \in V_{m \rhd d}$.

Formally, $\mathcal{M}_{\mathtt{safe}}$ is a DFA (Q, q_0, δ) where

- $Q = (\mathbb{M} \to \{0, \ldots, i\}) \cup \{q_{\mathtt{err}}\}$ is the finite set of states, storing the Parikh image of the execution currently read, upto a bound of i,
- $q_0 = (m \in \mathbb{M} \mapsto 0)$ is the initial state,
- $\delta \subseteq Q \times (\mathcal{N} \times \mathbb{M} \times \mathbb{D}) \times Q$ with (the site id $N \in \mathcal{N}$ is ignored)
  - $(q, (N, m \rhd d), q_{\mathtt{err}}) \in \delta$ iff $\bigwedge_{v_a \in V_{m \rhd d}} q \not\geq v_a$

– $(q_1, (N, m \triangleright d), q_2) \in \delta$ iff $\bigvee_{v_a \in V_a} q \geq v_a$ and
$q_2 = q_1[\mathsf{meth}(a) \mapsto \min(q_1(\mathsf{meth}(a)) + 1, i)]$

Theorem 10 (Safety Monitoring)**.** *A distributed library $\mathcal{L}$ is not safe if and only if the parallel composition $\mathcal{L} \times \mathcal{M}_{\mathtt{safe}}$ can reach the error state $q_{\mathtt{err}}$.*

Proof. $\mathcal{L} \times \mathcal{M}_{\mathtt{safe}}$ denotes the system whose set of states is the cartesian product of the states of $[\![\mathcal{L}]\!]$ and the ones of $\mathcal{M}_{\mathtt{safe}}$, and where the transitions labeled by $(N, m \triangleright d)$ are taken synchronously on both systems.

By definition, $\mathcal{M}_{\mathtt{safe}}$ goes into an error state if only if it reads an execution whose Parikh image is not greater than one of the minimal elements v_a, i.e. an execution which is not safe. □

7.5 Liveness

In this section, we give properties which characterize (weak) eventually consistent executions that will be used to define reductions of deciding (weak) eventual consistency to LTL model checking.

7.5.1 Weak Eventual Consistency

We first consider the case of weak eventual consistency because it is simpler while already showing some of the difficulties we have to solve. Moreover, for some systems, weak eventual consistency implies eventual consistency, for instance, when all the operations are commutative.

For an infinite execution e to be weak eventually consistent, there must exist some local interpretations which show that e is safe but also, which ensure that each operation $o \in O_e$ is *seen* by all other operations, except for some finite set. The latter implies that any finite set of operations is executed before every operation after some finite prefix. Thus, for $a = m \triangleright d$ with $m \in \mathbb{M}$ and $d \in \mathbb{D}$, if there are infinitely many a operations in e, then the specification $\mathcal{S}(a)$ must contain arbitrarily large posets. This property of $\mathcal{S}(a)$ can be stated as a property of the Parikh image of $\mathcal{S}(a)$ and this allows us to define a reduction of checking if some distributed library $\mathcal{L}$ is weak eventually consistent to checking if $\mathcal{L}$ is safe and if the parallel composition of $\mathcal{L}$ with a monitor that counts the methods executed by $\mathcal{L}$ satisfies some LTL formula. Mainly, the temporal operators in this formula are used to identify the infinitely occurring methods in some execution.

In the following lemma, we characterize weak eventually consistent executions. To identify the infinitely occurring methods in some execution e we use the following notation. Given $B \subseteq \mathbb{M} \times \mathbb{D}$ and a finite execution e_p, let $e_p \cdot B^\omega$ be the set of all executions e which extend e_p (i.e. e_p is a prefix of e) by an infinite set of B operations such that there are infinitely many a operations in e, for each $a \in B$.

Lemma 49. *[Characterization of Weak Eventual Consistency] Given $B \subseteq \mathbb{M} \times \mathbb{D}$ such that $\mathsf{meth}(B) = \{m_1, \ldots, m_k\}$, an execution $e \in e_p \cdot B^\omega$ is weak eventually consistent with respect to $\mathcal{S}$ if and only if*

- *e is safe and*
- *$\forall a \in B.\ \forall n \in \mathbb{N}.\ \exists n_1, \ldots, n_k \geq n.$*
  $\Pi_{\mathbb{M}}(e_p) + (m_1 \to n_1, \ldots, m_k \to n_k) \in \Pi_{\mathbb{M}}(\mathcal{S}(a))$

Proof. ($\Rightarrow$) If e is weak eventually consistent, then for each operation $o \in O_e$, there exists a local interpretation $li[o]$ such that the axioms ThinAir, Rval, and WeakEventual hold.

Let $a \in B$, and $n \in \mathbb{N}$. Since there are infinitely many a operations in e, we deduce from axiom WeakEventual that there exists one, noted o, such that $li[o]$ (or equivalently, $eb^{-1}(o)$) contains the operations of e_p and at least n additional m operations for each $m \in \mathsf{meth}(B)$. From axiom Rval, we know that $li_{\mathbb{M}}[o] \in \mathcal{S}(a)$, which shows that there exists $n_1, \ldots, n_k \geq n$ such that $\Pi_{\mathbb{M}}(e_p) + (m_1 \to n_1, \ldots, m_k \to n_k) \in \Pi_{\mathbb{M}}(\mathcal{S}(a))$.

($\Leftarrow$) For each operation $o \in O_e$, there exists a local interpretation $li[o]$ such that the axioms ThinAir, Rval hold. Let t be a total order consistent with both eb and po. In the following proof, when using the expressions such as *first*, *last*, and *between*, we refer to the order t.

For each $n \in \mathbb{N}^*$, let o_n be the last operation which occurs after at most n m_i operations for each $m_i \in \mathsf{meth}(B)$. For each $a \in B$ and $n \in \mathbb{N}^*$, let $n_1^a, \ldots, n_k^a \geq n$ such that

$$\Pi_{\mathbb{M}}(e_p) + (m_1 \to n_1^a, \ldots, m_k \to n_k^a) \in \Pi_{\mathbb{M}}(\mathcal{S}(a))$$

and let o_n^a be the first operation in e such that the prefix e' of e that ends in o_n^a, satisfies

$$\Pi_{\mathbb{M}}(e') \geq \Pi_{\mathbb{M}}(e_p) + (m_1 \to n_1^a, \ldots, m_k \to n_k^a). \tag{7.1}$$

The existence of o_n^a is ensured by the fact that e contains infinitely many a operations, for each $a \in B$.

For all $n \in \mathbb{N}^*$ and a operation o in e between o_n^a and o_{n+1}^a, the local interpretation $li[o] = (V_o, \leq_o)$ is defined as follows. The set V_o consists of all the operations of e_p, all the m_i operations before o_n, and some m_i operations before o s.t.

$$\Pi_{\mathbb{M}}(V_o) = \Pi_{\mathbb{M}}(e_p) + (m_1 \to n_1^a, \ldots, m_k \to n_k^a).$$

This is possible because, for each $m_i \in \mathsf{meth}(B)$, there are at most n m_i operations before o_n, and o occurs after o_n^a, that satisfies (7.1).

Now, since $\Pi_{\mathbb{M}}(V_o) \in \Pi_{\mathbb{M}}(\mathcal{S}(a))$, there exists a partial order $\leq_o$ over the set V_o such that $(V_o, \leq_o, \mathsf{meth}) \in \mathcal{S}(a)$. For the finite number of a operations o that occur in e before o_1^a, we use the local interpretations whose existence is ensured by the safety of e.

Since both eb and po are consistent with the total order t, axiom ThinAir holds. For each operation, we have chosen $li[o]$ so that axiom Rval holds.

Moreover, for all $n \in \mathbb{N}^*$, each operation o that occurs before o_n is executed-before all operations, except for a finite set – those that precede some o_n^a with $a \in B$. Thus, eb satisfies axiom WeakEventual, which concludes the proof. □

By Lemma 49, a distributed library $\mathcal{L}$ violates weak eventual consistency if and only if it violates safety or if it produces a execution in $e_p \cdot B^\omega$, for some e_p and $B \subseteq \mathbb{M} \times \mathbb{D}$ with $\{m_1, \ldots, m_k\} = \mathsf{meth}(B)$, such that there exists $a \in B$ satisfying:

$$\exists n \in \mathbb{N}. \forall n_1, \ldots, n_k \geq n. \\ \Pi_{\mathbb{M}}(e_p) + (m_1 \to n_1, \ldots, m_k \to n_k) \notin \Pi_{\mathbb{M}}(\mathcal{S}(a)) \tag{7.2}$$

Given $B \subseteq \mathbb{M} \times \mathbb{D}$ and $a \in B$, let

$$\Pi_{\mathsf{notWEC}}(\mathcal{S}, B, a) = \{v \mid \exists n \in \mathbb{N}. \forall n_1, \ldots, n_k \geq n. \\ v + (m_1 \to n_1, \ldots, m_k \to n_k) \notin \Pi_{\mathbb{M}}(\mathcal{S}(a))\}.$$

Then, (7.2) can be rewritten as $\Pi_{\mathbb{M}}(e_p) \in \Pi_{\mathsf{notWEC}}(\mathcal{S}, B, a)$. Like for safety, we construct a monitor $\mathcal{M}_{\mathtt{live}}$, which counts the methods executed by the distributed library, but this time without any bound. Finding a violation of weak eventual consistency reduces to finding a $B \subseteq \mathbb{M} \times \mathbb{D}$ and a finite execution e_p in the monitored system $\mathcal{L} \times \mathcal{M}_{\mathtt{live}}$, such that $\bigvee_{a \in B} \Pi_{\mathbb{M}}(e_p) \in \Pi_{\mathsf{notWEC}}(\mathcal{S}, B, a)$ and e_p can be extended by only using B operations as well as infinitely many a operations, for each $a \in B$, so that it belongs to $e_p \cdot B^\omega$. The latter can be checked using LTL model checking by adding to each state of the monitor, a register recording the method and return value of the last operation.

Formally, $\mathcal{M}_{\mathtt{live}}$ is a transition system (Q, I, δ), where

- $Q = (\mathbb{M} \to \mathbb{N}) \times \mathbb{M} \times \mathbb{D}$,
- I is the set of initial states: $(q_0, a) \in I$ if and only if $q_0 = (m \in \mathbb{M} \mapsto 0)$ and $a \in \mathbb{M} \times \mathbb{D}$,
- $\delta \subseteq Q \times (\mathcal{N} \times \mathbb{M} \times \mathbb{D}) \times Q$ where $((q_1, a), (N, m \rhd d), (q_2, m \rhd d)) \in \delta$ if and only if $q_2 = q_1[m \mapsto q_1(m) + 1]$.

Now, given a distributed library $\mathcal{L}$, we consider the system $\mathcal{L} \times \mathcal{M}_{\mathtt{live}}$ defined as the parallel composition of $\mathcal{L}$ and $\mathcal{M}_{\mathtt{live}}$. Define the following LTL formula:

$$\varphi_{\mathsf{notWEC}} = \bigvee_{B \subseteq \mathbb{M} \times \mathbb{D}} \bigvee_{a \in B} \Diamond(\Pi_{\mathsf{notWEC}}(\mathcal{S}, B, a) \wedge \bigcirc \Box B \wedge \bigwedge_{b \in B} \Box \Diamond b).$$

By an abuse of notation, $\Pi_{\mathsf{notWEC}}(\mathcal{S}, B, a)$ denotes also an atomic proposition, which holds in a state of the monitored system if and only if the vector formed by the counters of $\mathcal{M}_{\mathtt{live}}$ is in $\Pi_{\mathsf{notWEC}}(\mathcal{S}, B, a)$. As for the minimal elements used in monitoring safety, the sets $\Pi_{\mathsf{notWEC}}(\mathcal{S}, B, a)$ cannot be computed in general. In Section 7.6, we give a class of specifications for which they can. For each $B \subseteq \mathbb{M} \times \mathbb{D}$ (resp., $b \in \mathbb{M} \times \mathbb{D}$), B (resp., b) is an atomic proposition which holds in a state if and only if the second part of the state of $\mathcal{M}_{\mathtt{live}}$ is in B (resp., is b). Also, $\Diamond$, $\bigcirc$, and $\Box$ denote the temporal operators of LTL eventually, next, and always, respectively.

Theorem 11 (Weak Eventual Consistency Monitoring)**.** *A distributed library $\mathcal{L}$ is weak eventually consistent if and only if $\mathcal{L} \times \mathcal{M}_{\texttt{safe}}$ cannot reach $q_{\texttt{err}}$ and $\mathcal{L} \times \mathcal{M}_{\texttt{live}} \vDash \neg\varphi_{\mathsf{notWEC}}$.*

Proof. ($\Leftarrow$) Assume there exists an execution $e \in \mathcal{L}$ which is not weakly eventually consistent. Either e is not safe, and thus $\mathcal{L} \times \mathcal{M}_{\texttt{safe}}$ can reach $q_{\texttt{err}}$, or it is safe but not weakly eventually consistent. This means that e is an infinite execution, and thus, for some finite prefix e_p, and some non-empty set $B \subseteq \mathbb{M} \times \mathbb{D}$, we have $e = e_p \cdot B^\omega$. From Lemma 49, we deduce that

$$\exists a \in B. \exists n \in \mathbb{N}. \forall n_1, \ldots, n_k \geq n. \\ \Pi_{\mathbb{M}}(e_p) + (m_1 \to n_1, \ldots, m_k \to n_k) \notin \Pi_{\mathbb{M}}(\mathcal{S}(a))$$

Let us fix such $a \in B$. We thus know that $\Pi_{\mathbb{M}}(e_p) \in \Pi_{\mathsf{notWEC}}(\mathcal{S}, B, a)$. This implies that *execution* $\vDash \varphi_{\mathsf{notWEC}}$, and that $\mathcal{L} \times \mathcal{M}_{\texttt{live}} \nvDash \neg\varphi_{\mathsf{notWEC}}$.

($\Rightarrow$) Conversely, if $\mathcal{L} \times \mathcal{M}_{\texttt{safe}}$ can reach $q_{\texttt{err}}$, $\mathcal{L}$ is not safe and thus not weakly eventually consistent. If $\mathcal{L} \times \mathcal{M}_{\texttt{live}} \nvDash \neg\varphi_{\mathsf{notWEC}}$, then there exists an infinite execution e satisfying $\varphi_{\mathsf{notWEC}}$. The execution e can thus be split into a finite part e_p and an infinite suffix e_B where, for some $B \subseteq \mathbb{M} \times \mathbb{D}$, and some $a \in B$, $\Pi_{\mathbb{M}}(e_p) \in \Pi_{\mathsf{notWEC}}(\mathcal{S}, B, a)$, and e_B visits only states where proposition B holds and visits infinitely often states where b holds, for each $b \in B$. This means that there exists an infinite execution in $e_p \cdot B^\omega$, for which

$$\exists a \in B. \exists n \in \mathbb{N}. \forall n_1, \ldots, n_k \geq n. \\ \Pi_{\mathbb{M}}(e_p) + (m_1 \to n_1, \ldots, m_k \to n_k) \notin \Pi_{\mathbb{M}}(\mathcal{S}(a))$$

By using Lemma 49, we deduce that e is not weakly eventually consistent, and thus that $\mathcal{L}$ is not weakly eventually consistent either. □

7.5.2 Eventual Consistency

By following the same line of reasoning as the one used for weak eventual consistency, we first derive a necessary and sufficient condition for an execution to be eventually consistent.

In the case of an eventually consistent infinite execution e, axiom Eventual ensures that, for every finite prefix P of the global interpretation order gi, after some finite prefix of e, all the operations have P as a prefix of their local interpretations.

If $B \subseteq \mathbb{M} \times \mathbb{D}$ is the set of all a such that e contains infinitely many a operations, then the specification of each $a \in B$ must contain an infinite sequence of $\mathbb{M}$ labeled posets such that any prefix P of gi is a prefix of all these posets, except for some finite set. Thus, any prefix P of gi can be extended in order to belong to each of the $\mathcal{S}(a)$ with $a \in B$. Moreover, if P contains all the finitely occurring operations in e, then the extension can only add elements labeled by methods in $\mathsf{meth}(B)$. The set of labeled posets which can be extended in such a way is denoted by $Quot(S, B)$. This implies that an infinite sequence of increasing prefixes of gi must belong to $Quot(S, B)$, which, by extending a

classical definition of limit from words to labeled posets, can be stated as the poset defined by gi is in the *limit* of $Quot(S, B)$. Now, since gi is a reordering of e, this is equivalent to the fact that the multiset of methods that occur in e is the same as the multiset of methods that occur in some infinite labeled poset belonging to the limit of $Quot(S, B)$. Thus, the eventual consistency of an execution e can also be characterized in terms of Parikh images. We show in the following that the same monitor $\mathcal{M}_{\texttt{live}}$ defined previously can be used to reduce the problem of checking eventual consistency to LTL model checking.

Next, we formally define $Quot(S, B)$ and the notion of limit.

Definition 11 (Quotient)**.** *Given a specification $\mathcal{S}$, $B \subseteq \mathbb{M} \times \mathbb{D}$, and $a \in B$, let $\mathcal{S}(a)\mathsf{meth}(B)^{-1}$ – the* quotient *of $\mathcal{S}$ by $\mathsf{meth}(B)$ – be the set of labeled posets for which there exists an $\mathsf{meth}(B)$-completion in $S(a)$. Then, let*

$$Quot(S, B) = \bigcap_{a \in B} \mathcal{S}(a)\mathsf{meth}(B)^{-1}.$$

Definition 12 (Limit)**.** *Given a set $\mathcal{A}$ of finite labeled posets, we denote by $lim(\mathcal{A})$ the set of infinite labeled posets $(A, \leq, \ell)$ which have an infinite sequence of increasing prefixes in $\mathcal{A}$ such that every element in A is in a prefix (and all greater ones).*

Remark 5. *In the context of totally-ordered sets, the condition that every element in A is in a prefix is already implied by the rest of the definition. However, this is not true in the general case. For instance, let $(A, \leq, \ell)$ be an infinite labeled poset, where $A = \{a_i \mid i \in \mathbb{N}\} \cup \{b_i \mid i \in \mathbb{N}\}$, $\ell(a_i) = a$, $\ell(b_i) = b$ for all $i \geq 0$, $a_0 \leq a_1 \leq \ldots$, and $b_0 \leq b_1 \leq \ldots$. This poset is not in the limit of $\mathcal{A}$, the set of all finite totally-ordered sets where all elements are labeled by a, even though it has an infinite increasing sequence of prefixes which are in $\mathcal{A}$. This is due to the fact that the prefixes don't contain all the elements of $(A, \leq, \ell)$.*

This leads us to the following necessary and sufficient condition for eventual consistency.

Lemma 50 (Characterization of Eventual Consistency)**.** *Let $e = (O_e, po)$ be an infinite in $e_p \cdot B^\omega$. The execution e is eventually consistent if and only if it is safe and $\Pi_{\mathbb{M}}(e) \in \Pi_{\mathbb{M}}(lim(Quot(S, B)))$.*

Proof. ($\Rightarrow$) If e is eventually consistent, then there exist $gi \subseteq O_e \times O_e$, and for each operation $o \in O_e$, $li[o] \subseteq O_e \times O_e$ satisfying the axioms GIPF, THINAIR, RVAL, EVENTUAL.

It is enough to show that $(\mathbb{O}_\tau, gi, \mathsf{meth}) \in lim(Quot(S, B))$. Let P be a finite prefix of $(\mathbb{O}_\tau, gi, \mathsf{meth})$ containing at least the operations of e_p. For any $a \in B$, by axiom EVENTUAL, since there are infinitely many a operations, there exists at least one, o, such that P is a prefix of $li_{\mathbb{M}}[o]$. By RVAL, $li_{\mathbb{M}}[o]$ belongs to $\mathcal{S}(a)$, and by the fact that P contains all operations in e_p, $li_{\mathbb{M}}[o]$ is a $\mathsf{meth}(B)$-completion of P. Thus, $P \in Quot(S, B)$. Since we can find an infinite increasing sequence of such prefixes P containing every operation of O_e, we have $(\mathbb{O}_\tau, gi, \mathsf{meth}) \in lim(Quot(S, B))$.

($\Leftarrow$) This part of the proof is illustrated on Figure 7.8. Let $(A, \leq, \ell)$ be an $\mathbb{M}$ labeled poset in $\Pi_{\mathbb{M}}(lim(Quot(S, B)))$ with $\Pi_{\mathbb{M}}(\tau) = \Pi_{\mathbb{M}}(A)$. Define gi such that $(\mathbb{O}_\tau, gi, \mathsf{meth})$ is isomorphic to $(A, \leq, \ell)$ and let f be an isomorphism from A to O_e.

There exists an infinite sequence of increasing prefixes $A_1, A_2, \ldots$ of A such that every $f(A_i)$ contains the operations of e_p and such that, for all $a \in B$, and for all $n \in \mathbb{N}^*$, $A_n \in \mathcal{S}(a)\mathsf{meth}(B)^{-1}$. Moreover, each operation of O_e appears in at least one $f(A_i)$, for some i (and in every $f(A_j)$ with $j \geq i$).

The properties on $A_1, A_2, \ldots$ imply that, for every $a \in B$ and every $n \in \mathbb{N}^*$, there exists an $\mathsf{meth}(B)$-completion A_n^a of $f(A_n)$ such that $A_n^a \in \mathcal{S}(a)$. We assume that the elements added to $f(A_n)$ to obtain A_n^a come from O_e and that they occur after the operations in $f(A_n)$. This is possible because e contains infinitely many a operations, for each $a \in B$. For every $a \in B$ and $n \in \mathbb{N}^*$, let o_n^a be the first operation that occurs in e after all the operations in A_n^a.

In the following, we define $li[o]$, for every o, such that all the axioms of eventual consistency hold:

- for every $a \in B$, for the finite number of a operations o, that occur in e before o_1^a, we use the local interpretations whose existence is ensured by the fact that e is safe. Similarly, for every $a' \in \mathbb{M} \times \mathbb{D}$ such that e contains finitely many a' operations.
- for every $n \in \mathbb{N}^*$, $a \in B$, and for every a operation o between o_n^a and o_{n+1}^a, we define $li[o] = A_n^a$. Note that this implies that $f(A_n) \preceq li_{\mathbb{M}}[o]$ and that $li_{\mathbb{M}}[o] \in \mathcal{S}(a)$.

Axioms ThinAir and Rval hold for the same reasons given in the proof of Lemma 49. Axiom GIpf holds because of the way we have defined the limit of a set of labeled posets. It remains to show that axiom Eventual holds. Let P be a prefix of (O_e, gi). Let A_n be one of the previously defined prefixes such that $P \preceq f(A_n)$. For every $a \in B$ and for every a operation o that occurs after o_n^a, P is a prefix of $li[o]$. Thus, there are only finitely many operations which do not have P as a prefix of their local interpretations, which concludes the proof. □

Monitoring for eventual consistency is similar to the monitoring used for weak eventual consistency. Let $\Pi_{\mathsf{notEC}}(\mathcal{S}, B)$ be the set

$$\Pi_{\mathsf{notEC}}(\mathcal{S}, B) = \{v \in \mathbb{M} \to \mathbb{N} \mid v + (m \in \mathsf{meth}(B) \mapsto \omega) \notin \Pi_{\mathbb{M}}(lim(Quot(S, B)))\}$$

According to Lemma 50, in order to find an execution which is not eventually consistent, it is enough to look for an execution in $e_p \cdot B^\omega$ for some e_p and some $B \subseteq \mathbb{M} \times \mathbb{D}$ such that $\Pi_{\mathbb{M}}(e_p) \in \Pi_{\mathsf{notEC}}(\mathcal{S}, B)$. This problem can be again reduced to LTL model checking over the parallel composition of $\mathcal{L}$ and the same monitor $\mathcal{M}_{\texttt{live}}$. In this case, the LTL formula to be checked is:

$$\varphi_{\mathsf{notEC}} = \bigvee_{B \subseteq \mathbb{M} \times \mathbb{D}} \Diamond(\Pi_{\mathsf{notEC}}(\mathcal{S}, B) \wedge \bigcirc\Box B \wedge \bigwedge_{b \in B} \Box\Diamond b)$$

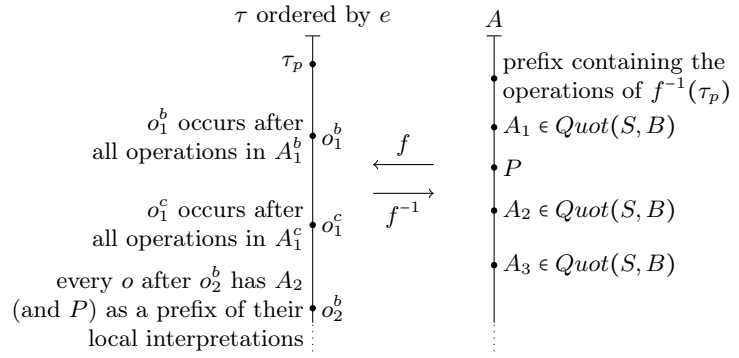


Figure 7.8: Illustration of the proof of the second part of Lemma 50 for $B = \{b, c\}$

As previously, for any $B \subseteq \mathbb{M} \times \mathbb{D}$, $\Pi_{\mathsf{notEC}}(\mathcal{S}, B)$ holds in a state of the monitored system if and only if the vector formed by the counters of $\mathcal{M}_{\mathtt{live}}$ is in $\Pi_{\mathsf{notEC}}(\mathcal{S}, B)$.

Theorem 12 (Eventual Consistency Monitoring)**.** *A distributed library $\mathcal{L}$ is eventually consistent if and only if $\mathcal{L} \times \mathcal{M}_{\mathtt{safe}}$ cannot reach $q_{\mathtt{err}}$ and $\mathcal{L} \times \mathcal{M}_{\mathtt{live}} \vDash \neg\varphi_{\mathsf{notEC}}$.*

Proof. Similar to the proof of Theorem 11, but using Lemma 50. □

7.6 Specifications of labeled posets

The reductions of eventual consistency to reachability and LTL model checking are effective if one can compute the set of minimal elements in the Parikh image of the specification and effective representations for the sets of vectors $\Pi_{\mathsf{notWEC}}(\mathcal{S}, B, a)$ and $\Pi_{\mathsf{notEC}}(\mathcal{S}, B)$ defined in Section 7.5.1 and 7.5.2. In the following, we introduce automata-based representations for specifications of finite-state optimistic replication systems, for which this is possible.

Essentially, each Σ labeled poset is abstracted as a sequence of multisets of symbols in Σ which is then recognized by a finite-state automaton where the transitions, instead of being labeled by symbols as in the case of automata over words, are labeled by Presburger constraints. By viewing multisets of symbols as vectors of integers, a sequence of multisets is recognized by an automaton if there exists a run such that the sequence satisfies the constraints imposed by the transitions of this run at each step. The abstraction of a poset as a sequence of multisets is defined based on its decomposition in *levels*, used in algorithms for parallel tasks scheduling [49].

These automata offer a good compromise between simplicity and expressiveness. Since they can recognize words over an alphabet Σ, they can represent specifications, that contain only totally-ordered sets, required by distributed library based on the Last Writer Wins conflict resolution policy [34]. They

are also able to represent specifications of distributed library with commutative conflict resolution policies (i.e. the effect of a set of conflicting operations does not depend on the order in which they are read) such as the CRDTs [18, 48] (see Example 22). For the latter case, the specifications represented by multiset automata are not exactly the ones introduced by the designers of these systems. However, we can prove that eventual consistency with respect to the original specification is equivalent to eventual consistency with respect to the specification recognized by the multiset automaton. In the following, we give a precise statement of this result.

Let $\mathcal{S} : \mathbb{M} \times \mathbb{D} \to \mathcal{P}(\mathsf{PoSet}_{\mathbb{M}})$ be a specification and $\sim$ a symmetric binary relation over $\mathbb{M}$, called *commutativity relation*. We say that an $\mathbb{M}$ labeled poset $(A, <, \ell)$ is *canonical with respect to* $\sim$ if and only if any two elements labeled by symbols, that are in the relation $\sim$, are incomparable, i.e. for any $x, y \in A$, $\ell(x) \sim \ell(y)$ implies that $x \not< y$ and $y \not< x$. Then, the specification $\mathcal{S}$ is called *$\sim$-closed* if and only if, for every $a \in \mathbb{M} \times \mathbb{D}$, if $\mathcal{S}(a)$ contains a labeled poset $\rho = (A, <, \ell)$, then $\mathcal{S}(a)$ also contains a labeled poset $\rho' = (A, <', \ell)$, which is canonical with respect to $\sim$ and such that $<' \subseteq <$. Furthermore, let $\mathcal{S}_\sim$ be a specification s.t. for every $a \in \mathbb{M} \times \mathbb{D}$, $\mathcal{S}_\sim(a)$ is the set of posets in $\mathcal{S}(a)$, that are canonical with respect to $\sim$.

For example, the labeled poset in Figure 7.5c, belonging to the OR-Set specification, is canonical with respect to the relation $\sim_O$ that consists of any pair of methods having different arguments (e.g., $\mathtt{add}_i$ and $\mathtt{rem}(j)$ with $i \neq j$), and any pair formed of an add or a remove and respectively, a lookup (e.g., $\mathtt{add}_i$ and $\mathtt{rem}_i$). The OR-Set specification in Example 20 is $\sim_O$-closed. Also, the MV-Register specification in Example 19 is $\sim_M$-closed, where $\sim_M$ contains any pair of a $\mathtt{wr}_i$ method and a $\mathtt{rd}$ method.

The following result follows from the fact that eventual consistency imposes rather weak constraints on the local interpretations associated to the operations in a trace.

Lemma 51. *Let $\mathcal{L}$ be a distributed library, $\sim$ a symmetric binary relation over $\mathbb{M}$, and $\mathcal{S}$ a $\sim$-closed specification. Then, $\mathcal{L}$ is eventually consistent with respect to $\mathcal{S}$ if and only if $\mathcal{L}$ is eventually consistent with respect to $\mathcal{S}_\sim$.*

A specification $\mathcal{S}$ is called *canonical with respect to* $\sim$ if and only if for every $a \in \mathbb{M} \times \mathbb{D}$, $\mathcal{S}(a)$ contains only posets, that are canonical with respect to $\sim$. Lemma 51 shows that, for distributed library like the MV-Register and the OR-Set, it is enough to consider canonical specifications. In the following, we define multiset automata and show how they can be used to represent canonical specifications.

Let $\rho = (A, <, \ell)$ be a Σ labeled poset. The ith level of ρ is the set of elements x in A such that the length of the longest path starting in x is i. A *decomposition* of ρ is a sequence $A_{n-1}, \ldots, A_0$, where $n-1$ is the length of the longest path in ρ and, for all $0 \leq i \leq n-1$, A_i is the ith level of ρ. Note that the decomposition of a labeled poset is unique. The *Σ-decomposition* of ρ is the sequence $\Gamma_{n-1}, \ldots, \Gamma_0$, where for all $0 \leq i \leq n-1$, Γ_i is the Parikh image of A_i. By definition, the length of $\Gamma_{n-1}, \ldots, \Gamma_0$ is n.

Example 21. *[Σ-decomposition] Consider the $\mathbb{M}_{\mathsf{OR\text{-}S}}$ labeled poset in Figure 7.5c. The $\mathbb{M}_{\mathsf{OR\text{-}S}}$-decomposition of this labeled poset is:*

$$\Gamma_2 = \{\mathtt{add}(1), \mathtt{rem}(1)\},$$
$$\Gamma_1 = \{\mathtt{add}(1), \mathtt{rem}(1), \mathtt{rem}(1), \mathtt{add}(0)\},$$
$$\Gamma_0 = \{\mathtt{add}(1), \mathtt{rem}(1), \mathtt{rem}(0), \mathtt{rem}(0)\}.$$

We define an effective representation for isomorphism-closed sets of $\mathbb{M}$ labeled posets by finite automata that recognize their $\mathbb{M}$-decompositions, i.e. sequences of *non-empty* multisets of symbols from $\mathbb{M}$.

In order to represent multisets of symbols from $\mathbb{M}$, we consider Presburger formulas φ over a set of free variables $\{\#m \mid m \in \mathbb{M}\}$, where $\#m$ denotes the number of occurrences of the symbol m. Let $\mathcal{F}_{\mathbb{M}}$ denote the set of all such formulas. Also, for any Presburger formula φ, let $[\varphi]$ denote the set of models of φ.

Definition 13 (Multiset Automata)**.** *A* multiset automaton *over $\mathbb{M}$ and $\mathbb{M} \times \mathbb{D}$ is a tuple*

$$\mathcal{A} = (Q, \delta, Q_0, (Q_a \mid a \in \mathbb{M} \times \mathbb{D})),$$

where Q is a finite set of states, $\delta \subseteq Q \times \mathcal{F}_{\mathbb{M}} \times Q$ is the transition relation, $Q_0 \subseteq Q$ is the set of initial states, and, for any $a \in \mathbb{M} \times \mathbb{D}$, $Q_a \subseteq Q$ (we say that the states in Q_a are labeled by a).

Intuitively, an $\mathbb{M}$ labeled poset is recognized by $\mathcal{A}$ iff there exists a run in $\mathcal{A}$ starting in the initial state such that the transitions are labeled by formulas that are satisfied successively by the Parikh images of the $n-1$th level, $n-2$th level, etc. (where $n-1$ is the length of the longest path).

For example, the multiset automaton in Figure 7.9 recognizes the labeled poset in Figure 7.5c because there exists a run starting in the initial state that goes through the states labeled by $\varnothing$, $\{1\}$, $\{0,1\}$, $\{1\}$ such that the associated sequence of formulas describe the $\mathbb{M}_{\mathsf{OR\text{-}S}}$-decomposition in Example 21.

A run of a multiset automaton $\mathcal{A}$ is a sequence $q_0 \overset{\varphi_0}{\to} q_1 \overset{\varphi_1}{\to} \dots \overset{\varphi_{n-1}}{\to} q_n$, s.t. for every $0 \le i \le n-1$, $(q_i, \varphi_i, q_{i+1}) \in \delta$. Such a run *recognizes* an $\mathbb{M}$ labeled poset ρ iff the $\mathbb{M}$-decomposition of ρ is $\Gamma_{n-1}\Gamma_{n-2}\dots\Gamma_0$ and for every $0 \le i \le n-1$, $\Gamma_{n-1-i} \in [\varphi_i]$. We say that the length of this run is n.

Given $q \in Q$, the set of all $\mathbb{M}$ labeled posets recognized by a run of $\mathcal{A}$, that ends in q, is denoted by $\mathcal{L}(\mathcal{A}, q)$. The labeled posets in $\mathcal{L}(\mathcal{A}, q)$ are said to be *interpreted to q*. Given a set of states $F \subseteq Q$, $\mathcal{L}(\mathcal{A}, F)$ denotes the union of $\mathcal{L}(\mathcal{A}, q)$ with $q \in F$.

Definition 14 (Canonical specifications and multiset automata)**.** *A specification $\mathcal{S}$ canonical with respect to $\sim$ is* recognized *by a multiset automaton $\mathcal{A}$ iff for every $a \in \mathbb{M} \times \mathbb{D}$, $S(a)$ is the set of posets in $\mathcal{L}(\mathcal{A}, Q_a)$, that are canonical with respect to $\sim$.*

Example 22. *[A multiset automaton for the OR-Set] Let $\mathcal{S}^2_{\mathsf{OR\text{-}S}}$ be a finite-state restriction of the OR-Set specification in Example 20 s.t. (1) the set object*

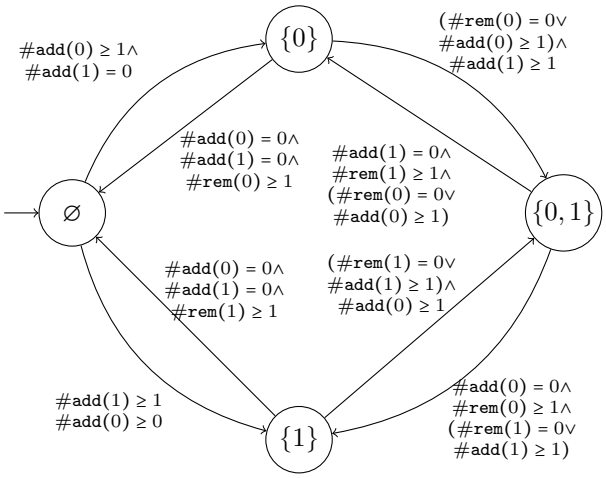


Figure 7.9: Some transitions of a multiset automaton that recognizes a specification of an OR-Set with at most two elements. $\texttt{lookup}(0) \rhd 0$ labels the states $\varnothing$ and $\{1\}$, $\texttt{lookup}(0) \rhd 1$ labels the states $\{0\}$ and $\{0,1\}$, and so on.

contains at most two elements 0 *and* 1 *and (2)* $\mathcal{S}^2_{\mathsf{OR\text{-}S}}$ *contains only posets canonical with respect to* $\sim_O$*. This specification is defined over the set of methods* `add`*,* `rem`*, and* `lookup` *with arguments* 0 *and* 1*, denoted by* $\mathbb{M}^2_{\mathsf{OR\text{-}S}}$*. The automaton in Figure 7.9 recognizes* $\mathcal{S}^2_{\mathsf{OR\text{-}S}}$*.*

The following theorem is a direct consequence of the fact that, for any multiset automaton $\mathcal{A}$, one can construct a finite-automaton over sequences whose language has exactly the same Parikh image as the set of labeled posets recognized by $\mathcal{A}$.

Lemma 52. *Let* $\mathcal{S}$ *be a specification recognized by a multiset automaton* $\mathcal{A}$*. Then, for every* $a \in \mathbb{M} \times \mathbb{D}$*, there exists an effectively computable Presburger formula* ψ_a *s.t.* $\Pi(\mathcal{S}(a)) = [\psi_a]$*.*

Proof. Given $a \in \mathbb{M} \times \mathbb{D}$, every vector in V_a is the Parikh image of a minimal labeled poset interpreted to some $q \in Q_a$, i.e. a labeled poset ρ recognized by a run of $\mathcal{A}$ without cycles $q_0 \xrightarrow{\varphi_0} q_1 \xrightarrow{\varphi_1} \ldots \xrightarrow{\varphi_{n-1}} q_n$ with $q_0 \in Q_0$, $q_n = q$, and $q_i \neq q_j$, for every $i \neq j$, such that the $\mathbb{M}$-decomposition of ρ is of the form $\Gamma_{n-1},\ldots,\Gamma_0$, where for every $0 \leq i \leq n-1$, Γ_{n-1-i} is a minimal model, with respect to the ordering over vectors of natural numbers, of φ_i. The fact that the number of runs in $\mathcal{A}$ without cycles is bounded and the number of minimal solutions for some Presburger formula is also bounded concludes our proof. □

Lemma 52 implies that, for every $a \in \mathbb{M} \times \mathbb{D}$, the set of minimal elements in $\Pi(\mathcal{S}(a))$ is effectively computable and that there exists a computable Presburger formula describing $\Pi_{\mathsf{notWEC}}(\mathcal{S}, B, a)$.

Given $B \subseteq \mathbb{M} \times \mathbb{D}$, we show that $\Pi_{\mathsf{notEC}}(\mathcal{S}, B)$ defined in Section 7.5.2 is definable as an effectively computable Presburger formula provided that the

specification $\mathcal{S}$ is recognized by a multiset automaton $\mathcal{A}$ satisfying some conditions.

First, $\mathcal{A}$ must recognize a prefix-closed set of labeled posets, i.e. $\bigcup_q \mathcal{L}(\mathcal{A}, q)$ is prefix-closed. This is quite natural since the set of all posets in some specification, i.e. $\bigcup_a S(a)$, is usually prefix-closed. Another condition that $\mathcal{A}$ must satisfy can be roughly stated as follows: given a set of methods M and a labeled poset ρ interpreted to some state q, the fact that there exists an M-completion of ρ interpreted to q' depends only on the states q, q', and the set of methods M. Formally, for all $q, q' \in Q$, $M \subseteq \mathbb{M}$, and $\rho_1, \rho_2 \in \mathcal{L}(\mathcal{A}, q)$, $\rho_1 \in \mathcal{L}(\mathcal{A}, q')M^{-1}$ iff $\rho_2 \in \mathcal{L}(\mathcal{A}, q')M^{-1}$. For example, if we consider the automaton in Figure 7.9, for any labeled poset interpreted to the state labeled by $\{1\}$ there exists an $\{\texttt{add}(0)\}$-completion interpreted to the state labeled by $\{0, 1\}$. In the case of the poset in Figure 7.5c this completion contains one more element labeled by $\texttt{add}(0)$, which is greater than all the elements labeled by $\texttt{rem}(0)$. Finally, we require that either $\mathcal{A}$ is a *word automaton*, i.e. the transitions are labeled by Presburger formulas that describe singleton multisets, or that for all q, if the limit of $\mathcal{L}(\mathcal{A}, q)$ contains an infinite poset ρ, then it also contains an infinite poset ρ' with the same Parikh image as ρ and such that the decomposition of ρ' has at most $|Q|$ levels. This last condition is satisfied by the automaton in Figure 7.9 (and an automaton representing an MV-Register specification). Actually, the bound $|Q|$ can be replaced by the constant 2. It is satisfied also by an automaton that describes a specification for the MV-Register. An automaton $\mathcal{A}$ satisfying all these conditions is called *completion-bounded*.

Lemma 53. *Let $\mathcal{S}$ be a specification recognized by a completion-bounded multiset automaton $\mathcal{A}$. Then, for every $B \subseteq \Sigma$, there exists an effectively computable Presburger formula φ_B such that $[\varphi_B] = \Pi_{\mathsf{notEC}}(\mathcal{S}, B)$.*

Proof. Given $a \in \mathbb{M} \times \mathbb{D}$ and $M \subseteq \mathbb{M}$, we can prove that there exists a maximal set of states $F_{M,a}$ such that $\mathcal{S}(a)M^{-1} = \mathcal{L}(\mathcal{A}, F_{M,a})$. To decide if a state q belongs to $F_{M,a}$, one has to consider a minimal labeled poset interpreted to q and search for an M-completion interpreted to a state in Q_a. It can be proved that if there exists such a completion there also exists one of bounded size.

Let $B \subseteq \mathbb{M} \times \mathbb{D}$, $M_B = \mathsf{meth}(B)$ and $F_B = \bigcap_{a \in B} F_{M_B,a}$. If $F_B = \varnothing$ then $\varphi_B = true$.

Otherwise, we compute a Presburger formula φ that describes the complement of $\Pi_{\mathsf{notEC}}(\mathcal{S}, B)$, i.e. the set of vectors $v \in \mathbb{M} \to \mathbb{N}$ such that $v + (m \in \mathsf{meth}(B) \mapsto \omega) \in \Pi_{\mathbb{M}}(lim(Quot(S, B)))$. By definition, $Quot(S, B) = \mathcal{L}(\mathcal{A}, F_B)$.

First, we prove that $lim(\mathcal{L}(\mathcal{A}, F_B))$ is the union of $lim(\mathcal{L}(\mathcal{A}, q))$ with $q \in F_B$. Let ρ be an infinite labeled poset, which has an infinite set of increasing prefixes ρ_0, ρ_1, ... in $\mathcal{L}(\mathcal{A}, F_B)$.

Also, let θ_0, θ_1, ... be an infinite sequence of runs in $\mathcal{A}$ such that, for all i, θ_i is a run that recognizes ρ_i. By Ramsey's theorem, there exist infinitely many runs θ_{j_1}, θ_{j_2},..., that end in the same state $q \in F_B$. Thus, the infinitely many posets ρ_{j_1}, ρ_{j_2},... belong to $\mathcal{L}(\mathcal{A}, q)$, which shows that $\rho \in lim(\mathcal{L}(\mathcal{A}, q))$. Next, we compute for each $q \in F_B$, a formula φ_q that describes the set of vectors $v \in \mathbb{M} \to \mathbb{N}$ such that $v + (m \in \mathsf{meth}(B) \mapsto \omega) \in \Pi_{\mathbb{M}}(lim(\mathcal{L}(\mathcal{A}, q)))$. If $\mathcal{A}$ is a word

automaton, then φ_q describes the Parikh image of all the sequences accepted by a run of $\mathcal{A}$ that ends in a cycle on q with at least one transition for each symbol in M_B and only transitions labeled by symbols in M_B. Thus, φ_q is effectively computable. Otherwise, we enumerate all the runs $q_0 \xrightarrow{\varphi_0} q_1 \xrightarrow{\varphi_1} \dots \xrightarrow{\varphi_{n-1}} q_n = q$ of $\mathcal{A}$ of length at most $|Q|$. For every such run θ, every index $0 \leq i \leq n-1$, and every surjective mapping $f : M_B \to \{i, \dots, n-1\}$, we define a run θ_f as follows. For each $i \leq j \leq n-1$, let φ'_j be a formula describing the set of all multisets v of symbols from $\mathbb{M} \setminus M_B$ s.t. for every integer L, there exists a multiset modeled by φ_j, that consists of v, at least L symbols m, for all $m \in f^{-1}(j)$, and possibly other symbols from M_B. The run θ_f is obtained from θ by replacing φ_j with $\varphi_j \wedge \varphi'_j$, for all $i \leq j \leq n-1$.

The models of φ_q are the Parikh images of all the posets, which are recognized by a run θ_f as above. To prove that the formula φ_q is effectively computable, one can use the same reasoning as in the proof of Theorem 52.

Finally, φ is the disjunction of φ_q with $q \in F_B$ and $\varphi_B = \neg\varphi$. □

7.7 Decidability of Eventual Consistency

We give decidability results for the case where the specifications are given by multiset automata, and where the distributed library is finite-state, and the number of sites is bounded.

When the number of sites is bounded, a finite-state distributed library $\mathcal{L}$ can be modeled operationally using a VASS $\mathcal{V}$. In the semantics of $\mathcal{L}$, a configuration of $\mathcal{L}$ is composed of two parts: the first part is a tuple where each component describes the state each node is in; the second part is a function, describing the content of each channel. In $\mathcal{V}$, the first part can be encoded in the control-state. Moreover, the content of an unbounded unordered channel ch of a distributed library can be modeled by $|\mathsf{Msg}|$ counters $ch_1, \dots, ch_{|\mathsf{Msg}|}$ used to count how many of each kind of messages there are in ch.

We are given a distributed library $\mathcal{L}$ and a specification $\mathcal{S}$ described by a multiset automaton $\mathcal{A}$. As a direct consequence of Lemma 52, given $a \in \mathbb{M} \times \mathbb{D}$, the set of minimal elements in $\Pi(\mathcal{S}(a))$ is effectively computable. This implies that the safety monitor $\mathcal{M}_{\texttt{safe}}$ can be also be effectively constructed. Moreover, we can construct the parallel composition $\mathcal{V} \times \mathcal{M}_{\texttt{safe}}$ of $\mathcal{V}$ with $\mathcal{M}_{\texttt{safe}}$. We get a VASS $\mathcal{V}_{\texttt{safe}}$, on which we can solve the problem of control state reachability, in order to know if it is possible to reach the error state $q_{\texttt{err}}$ of the monitor.

The bound i for the minimal vectors V_a used to construct $\mathcal{M}_{\texttt{safe}}$ is exponential in $|A|$ and thus, the number of states in $\mathcal{M}_{\texttt{safe}}$ (and in $\mathcal{V} \times \mathcal{M}_{\texttt{safe}}$) is also exponential in $|A|$. Moreover, solving control-state reachability in VASS is known to be EXPSPACE-complete, which leads to the following theorem.

Theorem 13 (Decidability of Safety)**.** *For a finite-state distributed library $\mathcal{L}$, and a specification $\mathcal{S}$ described by a multiset automaton, the problem of checking the safety of $\mathcal{L}$ with respect to $\mathcal{S}$ is decidable, and in* 2 – EXPSPACE.

Proof. Follows from Theorem 10 □

The following lemma is used for proving the decidability of both weak eventual consistency and eventual consistency.

Lemma 54. *Let $\mathcal{V} = (Q, d, \delta)$ be a* VASS *where the states are labeled with atomic propositions coming from a finite set AP. Let φ be a Presburger formula with d free variables, and $P \subseteq AP$. The problem of checking the* LTL *formula*

$$\mathcal{V} \vDash \neg\Diamond(\varphi \wedge \bigcirc\Box P \wedge \bigwedge_{p \in P} \Box\Diamond p)$$

is decidable, and can be reduced to reachability in VASS.

Proof. For the formula to be satisfied, there must exist an infinite execution in $\mathcal{V}$ of the form $(q_0, \boldsymbol{v}_0)\dots(q_i, \boldsymbol{v}_i)\dots$ where (1) the valuation of the counters in $\boldsymbol{v}_i$ satisfies φ, (2) for all $j > i$, q_j is labeled by a proposition in P, and (3) for each $p \in P$, there are infinitely many q_j labeled by p.

First, Th. 2.14 in Valk and Jantzen [52] shows that it is possible to compute the minimal elements of the upward-closed set of configurations U_P from which there exists an infinite execution visiting only states labeled by a proposition in P and infinitely many states labeled by p, for each $p \in P$. Then, we construct a Presburger formula φ' representing the intersection of $[\varphi]$ and U_P. The problem of checking if there exists a reachable configuration in $\mathcal{V}$ satisfying φ' can be reduced to (configuration) reachability in VASS [8]. □

Theorem 14 (Decidability of (Weak) Eventual Consistency)**.** *Given a finite-state distributed library $\mathcal{L}$, and a specification $\mathcal{S}$ described by a multiset automaton, the problem of checking the (weak) eventual consistency of $\mathcal{L}$ with respect to $\mathcal{S}$ is decidable when the number of sites is bounded by k.*

Proof. Let $\mathcal{V}$ be a VASS used to model $\mathcal{L}^k$, i.e. the executions of $\mathcal{L}$ when used on k sites. For weak eventual consistency, we know from Theorem 11 that, in addition to checking the safety of $\mathcal{L}$, it is enough to check whether the parallel composition $\mathcal{V} \times \mathcal{M}_{\texttt{live}}$ of $\mathcal{V}$ with $\mathcal{M}_{\texttt{live}}$ has an infinite run satisfying $\varphi_{\mathsf{notWEC}}$. Moreover, as a direct consequence of Theorem 52, for any $B \subseteq \mathbb{M} \times \mathbb{D}$ and $a \in B$, the set of vectors $\Pi_{\mathsf{notWEC}}(\mathcal{S}, B, a)$ appearing in $\varphi_{\mathsf{notWEC}}$ can be described by a Presburger formula.

Thanks to Lemma 54, we can check for each disjunct of $\varphi_{\mathsf{notWEC}}$ whether there is a run in $\mathcal{V} \times \mathcal{M}_{\texttt{live}}$ satisfying it, which ends the proof for weak eventual consistency.

For eventual consistency, the decidability follows in a similar way from Lemma 54 and Theorem 53, which shows that for all $B \subseteq \mathbb{M} \times \mathbb{D}$, $\Pi_{\mathsf{notEC}}(\mathcal{S}, B)$ can be represented by a Presburger formula. □

7.8 Undecidability From RYW to Implicit Causal Consistency

We first introduce a problem over formal languages which is undecidable, and can be reduced to any consistency criteria stronger than RYW consistency and

weaker than implicit causally consistency (inclusive), showing that a whole range of criteria is undecidable.

The undecidable problem we use is the containment problem of weighted automata over the semiring $(\mathbb{N}, \mathsf{min}, +)$. A *weighted automaton* W over alphabet Σ is an LTS (Q, q_0, q_f, Δ) whose transitions are labeled by $\Sigma \times \mathbb{N}$, with the integer representing the *weight* of a transition.

Since we're using the semiring $(\mathbb{N}, \mathsf{min}, +)$, the *weight* of a particular run of the automaton is the sum of weights along the transitions. Moreover, the *weight* of a word w is defined as being the minimal weight of all runs recognizing w. We denote it $W(w)$. For simplicity, we assume that weighted automata recognize all words, so that the weight is always defined.

Given two weighted automata W_A and W_B, the *containment* problem asks

$$\forall w \in \Sigma^*.\ W_A(w) \geq W_B(w)$$

and is known to be undecidable [2, 36].

We know have all the ingredients to prove the undecidability theorem.

Theorem 15. *Given a consistency criterion X stronger than RYW consistency and weaker than implicit causally consistency (inclusive), a finite-state distributed library $\mathcal{L}$ and a specification $\mathcal{S}$ given by a regular automaton, the problem of checking whether all executions of $\mathcal{L}$ when used on two sites ($\mathcal{L}^2$) respect criterion X is undecidable.*

Proof. Let W_A, B be two weighted automata languages over the alphabet Σ. We show how to reduce the containment problem to X consistency by constructing a distributed library $\mathcal{L}$.

For the methods, we use $\mathbb{M} = \{m_b \mid b \in \Sigma\} \uplus \{m_a, \mathsf{Check}, \mathsf{Choice}\}$, where Check and Choice are fresh symbols. The possible return values are $\top$ and $\bot$. The only message is $\mathsf{Msg} = \{\mathsf{Did_a}\}$. We will describe its effect hereafter.

Let n be the maximum weight which appears in W_A. The local states of the nodes belong to the set $Q_{\mathcal{L}} = (Q_A \times 1, \ldots, n) \uplus \{q_{\mathsf{init}}, q_a, q_{\mathsf{lost}}\}$ where Q_A is the set of states of W_A, and $\{q_{\mathsf{init}}, q_a, q_{\mathsf{lost}}\}$ are fresh states.

We now describe the effect of each method in each possible state. When undefined, we assume that $\mathcal{L}(m)(q) = \{(q_{\mathsf{lost}}, \bot, \_)\}$. We use the symbol $\_$ to denote the fact that no message is broadcast after the call to the method (or equivalently, a message whose effect is the identity on the set of states is broadcast). When normally simulating the containment problem, we should never encounter such a situation, i.e. a method should never be called in a state where the effect is undefined. If it does, the simulation effectively ends with an execution which is X consistent by default, as we will see later.

We also use the symbol $\_$ for return values which do not matter. The only method for which the return value matters is Check, as it is the only method which as a non-trivial specification.

The effect of Choice is defined as $\mathcal{L}(\mathsf{Choice})(q_{\mathsf{init}}) = \{(q_0, 0), \_, \_), (q_a, \_, \_)\}$ with q_0 being the initial state of W_A. The effect of m_a is defined as $\mathcal{L}(m_a)(q_a) = \{(q_a, \_, \mathsf{Did_a})\}$. The effect $\mathcal{L}(m_b)(q, i)$ of m_b for $b \in \Sigma$ is defined as the set of

all $(q', 0), \_, \_)$ such that there is a transition $(q, (b, j), q')$ with $i \geq j$. If no such transition exists, $\mathcal{L}(m_b)(q, i) = \{(q_{\mathsf{lost}}, \_, \_)\}$. Finally, $\mathcal{L}(\mathsf{Check})(q_f, i) = (q_{\mathsf{lost}}, \top, \_)$ where q_f is a final state of W_A. Fig 7.10 gives an illustration of an execution of $\mathcal{L}$, and how it used to simulate W_A.

The idea is that the sites first non-deterministically go into state $(q_0, 0)$ or q_a, because of calls to the method Choice. Then the site who went to q_a (called t_2) broadcasts a message $\mathsf{Did_a}$ every time method m_a is executed. Upon reception of $\mathsf{Did_a}$, the other sites (called t_1) increments the second component of its state (called *accumulator*). Formally, $\mathsf{Did_a}$ is the function which maps (q, i) to $(q, \mathsf{min}(i + 1, n))$, with q being a state of W_A, and maps every other state to q_{lost}. When a method m_b is called on t_1, it can update its state, following a transition whose weight is smaller than the value of the accumulator, and reset the accumulator to 0. If everything went according to plan (i.e. all the methods were called as described, and t_1 did not go into state q_{lost}), and t_1 reached a final state then the method Check returns $\top$.

Finally, the specification $\mathcal{S}$ is defined as follows. $\mathcal{S}(\mathsf{Check} \triangleright \top)$ is the regular language (in $\mathbb{M}^*$ obtained by replacing every weighted transition $q \xrightarrow{(b,p)} q'$ (with $b \in \Sigma$ and $p \in \mathbb{N}$) from W_A by a sequence of transitions: $q \xrightarrow{m_a} \cdots \xrightarrow{m_a} \cdot \xrightarrow{m_b} q'$ with fresh intermediary states, and with p m_a-transitions. The goal is that the number of a's in a word corresponds to the weight of the (projection over Σ of the) word. We also add self-loops labeled by Choice on every state (this method doesn't matter for the specification). The specification $\mathcal{S}$ maps every other $m \triangleright d$ for $m \in \mathbb{M}$ and $d \in \mathbb{D}$ to the full set $\mathbb{M}^*$.

We can now prove the following equivalence:

1. $\mathcal{L}^2$ is implicitly causally consistent
2. $\mathcal{L}^2$ is X consistent
3. $\mathcal{L}^2$ is RYW consistent
4. $\forall w \in \Sigma^*.\ W_A(w) \geq W_B(w)$

(1) $\Rightarrow$ (2) $\Rightarrow$ (3) By assumption on X.

(3) $\Rightarrow$ (4) Let $w \in \Sigma^*$. We build an execution $e \in \mathcal{L}^2$ for w following the schema of Fig 7.10. First, method Choice is executed in t_1, and brings t_1 into state $(q_0, 0)$, with q_0 being the initial state of W_A. Then method Choice is executed in t_2 and brings it into state q_a.

Consider a run of weight $W_A(ww)$ which accepts w in W_A. For each transition $(q_1, (b, i), q_2)$ taken by the run, the following happens. Method m_a is executed on t_2, resulting in a message $\mathsf{Did_a}$ being broadcast. Site t_1 receives the message, and increments its accumulator as a result. This is repeated i times, until the value of the accumulator is i. Then method m_b is called on t_1, updating the state from (q_1, i) to $(q_2, 0)$, effectively resetting the accumulator, and updating the state of W_A.

At the end, as the first component of t_1 is now a final state, method Check can be executed and returns value $\top$. Since $e \in \mathcal{L}^2$ is RYW consistent, there

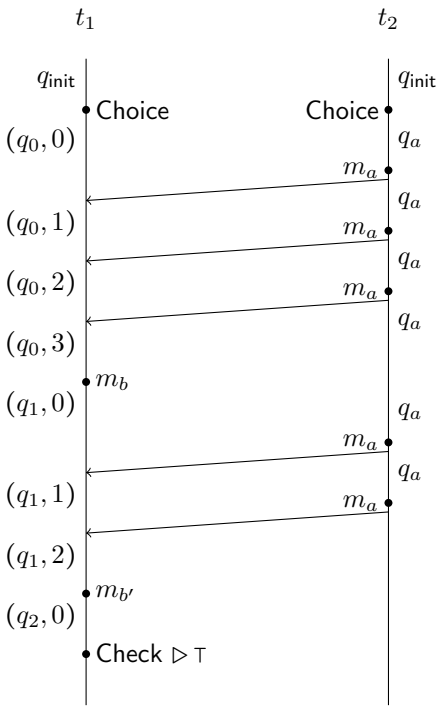


Figure 7.10: Execution simulating the transitions $(q_0, (b, 3), q_1)$ and $(q_1, (b', 2), q_2)$ of the weighted automaton W_A.

must exist a local interpretation order for this last $\mathsf{Check} \triangleright \top$ operation. By the axioms of RYW consistency, this local interpretation order must respect the order of operations on t_1, and include a subset of the operations from t_2. Moreover, the word formed by the local interpretation order must belong to $\mathcal{S}(\mathsf{Check} \triangleright \top)$. Moreover, since the number of m_a which were executed in t_2 is exactly equal to $W_A(w)$ (the sum of weights of each transitions), we obtain a word in $\mathcal{S}(\mathsf{Check} \triangleright \top)$ which has necessarily a number of m_a letters at most equal to $W_A(w)$.

By construction of $\mathcal{S}(\mathsf{Check} \triangleright \top)$, this corresponds to a run in W_B for the word w, whose weight is smaller than $W_A(w)$. As a result, $W_A(w) \geq W_B(w)$.

(4) $\Rightarrow$ (1) Let e be an execution of $\mathcal{L}^2$. If it doesn't contain a $\mathsf{Check} \triangleright \top$ operation, it is automatically implicitly causally consistent, as the specification for all other operations is $\mathbb{M}^*$. Otherwise, we let t_1 be the site identifier of the site executing a $\mathsf{Check} \triangleright \top$ operation o. Let $w \in \Sigma^*$ be the sequence formed by the m_b operations (with $b \in \Sigma$) executed on t_1 before the o. By construction of $\mathcal{L}$, we know that t_1 must have reached a final state of W_A before executing this operation. Moreover, we know that, in order to update its state, t_1 must first increment its accumulator to at least the weight of the transition it wants to use, meaning that t_2 must have sent a number of $\mathsf{Did_a}$ messages at least equal to $W_A(w)$.

Consider an accepting run for w in W_B, whose weight is less or equal than $W_A(w)$. Using this run, and knowing that there are at least $W_A(w)$ m_a operations executed by t_2, we can construct a local interpretation order for o which respects the axioms of implicit causally consistency. □

Remark 6. *The proof of Theorem 15 does not rely on the unboundedness of the communication channels between the nodes, and is solely based on the intricacy of the consistency criteria. In fact, we can assume that the size of the channels is bounded by* 1*, and still obtain the same undecidability result. This is because, in the proof of* (3) $\Rightarrow$ (4)*, we can always assume that* t_1 *reads the message* $\mathsf{Did_a}$ *right after it was broadcast by* t_2*.*

7.9 Undecidability from FIFO to Causal Consistency

We now study the problem of deciding whether a distributed library is causally consistent. We show that, even when the distributed library is finite-state, and the specification is sequential and regular, this problem is undecidable. We will in fact show undecidability for any criteria stronger than FIFO consistency and weaker than explicit causally consistency (inclusive). This range includes criteria where only a subset of messages impose causality constraints. FIFO consistency corresponds to the situation where the only causality constraints are the ones imposed by the program order, while explicit causally consistency considers that every message imposes a causality constraint.

FIFO and causal consistency are closely related to the shuffling operator over words. Given two words $u, v \in \Sigma^*$, the *shuffling* operator returns the set of words which can be obtained from u and v by interleaving their letters. Formally, we define $u \| v \subseteq \Sigma^*$ inductively: $\epsilon \| v = \{v\}$, $u \| \epsilon = \{wu\}$ and $(a \cdot u) \| (b \cdot v) = a \cdot (u \| (b \cdot v)) \cup b \cdot ((a \cdot u) \| v)$.

We first prove the undecidability of a problem about shuffling of words in Lemma 55, and then reduce it to any criterion between FIFO and causal consistency, showing that this whole range is undecidable.

Lemma 55. *Let $A, B, L \subseteq \Sigma^*$ be regular languages represented by finite automata. The problem*

$$\exists u \in A, v \in B.\ u \| v \cap L = \varnothing$$

is undecidable.

Proof. Let $\Sigma_{\mathsf{PCP}} = \{a, b\}$ and $(u_1, v_1), \ldots, (u_n, v_n) \in (\Sigma^*_{\mathsf{PCP}} \times \Sigma^*_{\mathsf{PCP}})$ be n pairs forming the input of a PCP problem P. Let $\Sigma_u = \{a_u, b_u\}$ and $\Sigma_v = \{a_v, b_v\}$ be two disjoint copies of Σ_{PCP}, and let $h : (\Sigma_u + \Sigma_v)^* \to \Sigma_{\mathsf{PCP}}$, $h_u : \Sigma_{\mathsf{PCP}} \to \Sigma_u$, $h_v : \Sigma_{\mathsf{PCP}} \to \Sigma_v$, be the homomorphisms which map corresponding letters. Moreover, let s_u and s_v be two new letters, and $\Sigma = \Sigma_u \cup \Sigma_v \cup \{s_u, s_v\}$.

Our goal is to define regular languages $A, B, L \subseteq \Sigma^*$ such that

$$\begin{aligned} &\text{The PCP problem } P \text{ has a positive answer } u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k},\ (k > 0) \\ &\Leftrightarrow \exists u \in A, v \in B.\ u \| v \cap L = \varnothing \end{aligned} \tag{7.3}$$

The idea is to encode in u the sequence $u_{i_1}, \ldots, u_{i_k}$ as $h_u(u_{i_1}) \cdot s_u \cdots h_u(u_{i_k}) \cdot s_u$ by using s_u as a separator (and end marker), and likewise for v with the separator s_v. Then, we define the language L by a disjunction of regular properties that no shuffling of an encoding of a valid PCP answer to P could satisfy.

Formally, we have:

- $A = (\Sigma_u + s_u)^*$
- $B = (\Sigma_v + s_v)^*$
- $w \in L$ iff one of the following conditions holds:
  - (i) when ignoring the letters s_u and s_v, w starts with an alternation of Σ_u and Σ_v such that two letters don't match
    $w_{|\Sigma_u \cup \Sigma_v} \in (\Sigma_u \Sigma_v)^* (a_u b_v + b_u a_v) \Sigma^*$
  - (ii) when ignoring the letters s_u and s_v, w starts with an alternation of Σ_u and Σ_v and ends with only Σ_u letters or only Σ_v letters
    $w_{|\Sigma_u \cup \Sigma_v} \in (\Sigma_u \Sigma_v)^* (\Sigma_u^+ + \Sigma_v^+)$
  - (iii) when only keeping s_u and s_v letters, either w starts with an alternation of s_u and s_v and ends with only s_u or only s_v, or w is the empty word ε
    $w_{|\{s_u, s_v\}} \in (s_u s_v)^* (s_u^+ + s_v^+) + \varepsilon$

(iv) w contains a letter from Σ_u not followed by s_u, or a letter from Σ_v not followed by s_v
$w \in \Sigma^*\Sigma_u(\Sigma \setminus s_u)^* + \Sigma^*\Sigma_v(\Sigma \setminus s_v)^*$

(v) w starts with an alternation of $\Sigma_u^* s_u$ and $\Sigma_v^* s_v$ such that one pair of Σ_u^*, Σ_v^* is not a pair of our PCP instance
$w \in (\Sigma_u^* s_u \Sigma_v^* s_v)^* (\Sigma_u^* s_u \Sigma_v^* s_v \setminus +_i h_u(u_i) \cdot s_u \cdot h_v(v_i) \cdot s_v)\Sigma^*$

We can now show that equivalence 7.3 holds.

($\Rightarrow$) Let $k > 0$, $i_1, \ldots, i_k \in \{1, \ldots, n\}$ such that $u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$. Let $u = h_u(u_{i_1}) \cdot s_u \cdots h_u(u_{i_k}) \cdot s_u$ and $v = h_u(v_{i_1}) \cdot s_v \cdots h_u(v_{i_k}) \cdot s_v$. We want to show that no word w in the shuffling of u and v satisfies one of the conditions of S. If w starts with an alternation of Σ_u and Σ_v, since $u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$, any pair of letters match, and thus, condition (i) cannot hold. Condition (ii) cannot hold since w contains as many letters from Σ_u as from Σ_v. Likewise for condition (iii) since w contains as many s_u as s_v (and at least 1).

Since u (resp., v) doesn't contain a letter from Σ_u not followed by s_u (resp., a letter from Σ_v not followed by s_v), neither does w, which shows that condition (iv) doesn't hold. Finally, if w starts with an alternation of $\Sigma_u^* s_u$ and $\Sigma_v^* s_v$, then all the corresponding pairs of Σ_u^*, Σ_v^* are pairs from the PCP input, and condition (v) cannot hold either.

($\Leftarrow$) Let $u \in A, v \in B$ such that $u \| v \cap L = \varnothing$. Since no word in $u \| v$ satisfies condition (iii), nor condition (iv), u ends with s_u, v ends with s_v, and u has as many s_u as v has s_v (and at least 1). This shows that, $u = x_1 s_u \cdots x_k s_u$ and $v = y_1 s_v \cdots y_k s_v$ for some $k > 0$, $x_1, \ldots, x_k \in \Sigma_u^*$, $y_1, \ldots, y_k \in \Sigma_v^*$. Moreover, since no word in $u \| v$ satisfies condition (v), for any j, (x_j, y_j) corresponds to a pair (u_{i_j}, v_{i_j}) of our input P – more precisely, $h(x_j) = u_{i_j}$ and $h(y_j) = v_{i_j}$ for some $i_j \in \{1, \ldots, n\}$. Finally, the fact that no word in $u \| v$ satisfies condition (i), nor condition (ii) ensures that $h(x_1 \cdots x_k) = h(y_1 \cdots y_k)$ and that the PCP problem P has a positive answer $u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$. □

The undecidability theorem holds for any consistency criterion between FIFO consistency and explicit causally consistency.

Theorem 16. *Given a consistency criterion X stronger than FIFO consistency but weaker than explicit causally consistency (inclusive), a finite-state distributed library $\mathcal{L}$ and a specification $\mathcal{S}$ given by an NFA, the problem of checking whether all executions of $\mathcal{L}^2$ respect criterion X is undecidable.*

Proof. Let A, B, L be three regular languages over Σ. We construct a distributed library $\mathcal{L}$, showing how to reduce the problem from Lemma 55 to X consistency.

For the methods, we use $\mathbb{M} = \{m_a \mid a \in \Sigma\} \cup \{\mathsf{Choice}, \mathsf{End_B}, \mathsf{Check}\}$, and $\mathbb{D} = \{\bot, \top\}$ for the domain of the return values. We will use a unique message $\mathsf{Finished_B} \in \mathsf{Msg}$.

The idea is to let two nodes, t_A and t_B, execute m_a operations for $a \in \Sigma$, to form words from A and B respectively. When t_B reaches a final state of B,

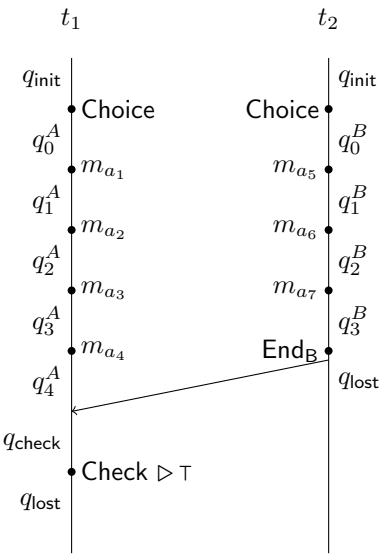


Figure 7.11: Node t_A executes the word $a_1a_2a_3a_4 \in A$ while node t_B executes $a_5a_6a_7 \in B$. When $\mathsf{End_B}$ is executed on t_B, a message $\mathsf{Finished_B}$ is broadcast. Upon reception, t_A reaches a state where Check returns $\top$. FIFO or causal consistency will then ensure that there is a shuffling or $a_1a_2a_3a_4$ and $a_5a_6a_7$ that belongs to L.

and method $\mathsf{End_B}$ is executed on t_B, a message $\mathsf{Finished_B}$ is broadcast. Upon reception, t_A reaches a state where Check returns $\top$.

Then, we will fix the specification $\mathcal{S}(\mathsf{Check} \triangleright \top)$ to be the language L (up to technical details we will explain later). As a result, FIFO or causal consistency will ensure that there exists a local interpretation order for the $\mathsf{Check} \triangleright \top$ operation, effectively giving a possible shuffling of the words executed by t_A and t_B which belongs t L.

Let $Q = Q_A \uplus Q_B \uplus \{q_{\mathsf{init}}, q_{\mathsf{lost}}, q_{\mathsf{check}}\}$ be the set of local states of the nodes. We show in Fig 7.11 how the states are updated during the execution of the methods.

We now describe the effect of each method in each possible state. Similarly to the proof of Theorem 15, when undefined, we assume that $\mathcal{L}(m)(q) = \{(q_{\mathsf{lost}}, \bot, \_)\}$. We use the symbol $\_$ to denote the fact that no message is broadcast after the call to the method (or equivalently, a message whose effect is the identity on the set of states is broadcast). When normally simulating the containment problem, we should never encounter such a situation, i.e. a method should never be called in a state where the effect is undefined. If it does, the simulation effectively ends with an execution which is X consistent by default, as we will see later.

We also use the symbol $\_$ for return values which do not matter. The only method for which the return value matters is Check, as it is the only method which as a non-trivial specification.

The effect of Choice is defined as $\mathcal{L}(\mathsf{Choice})(q_{\mathsf{init}}) = \{(q_0^A, \_, \_), (q_0^B, \_, \_)\}$ where q_0^A and q_0^B are the initial states of A and B (we assume without loss of generality that they have a unique initial state). The effect $\mathcal{L}(m_a)(q)$ is defined as the set of all possible $(q', \_, \_)$ for which there exists a transition (q, a, q')

in A or B. When no such transition exists, $\mathcal{L}(m_b)(q,i) = \{(q_{\mathsf{lost}}, \_, \_)\}$. The effect of $\mathsf{End_B}$ is defined as $\mathcal{L}(\mathsf{End_B})(q_f^B) = (q_{\mathsf{lost}}, \_, \mathsf{Finished_B})$ when q_f^B is a final state of B. Finally, $\mathcal{L}(\mathsf{Check})(q_{\mathsf{check}}) = (q_{\mathsf{lost}}, \top, \_)$.

As defined in the effect of $\mathsf{End_B}$, the message $\mathsf{Finished_B}$ is sent when method $\mathsf{End_B}$ is executed in a final state of B. If the receiving site is in a final state of A, the site updates its state to q_{check}. Formally, $\mathsf{Finished_B}$ is the function which maps every final state of A to q_{check}, and every other state to q_{lost}. Then, when Check is executed in state Check it returns value $\top$.

Formally, the specification $\mathcal{S}(\mathsf{Check} \rhd \top)$ is defined as $L' \cdot \mathsf{End_B}$ where L' is L where we replace every letter a by m_a. We also add self-loops for Choice on every state of L, so that the specification ignores that method. The specification of every other method is defined as $\mathbb{M}^*$.

We will now prove the following equivalence:

1. $\mathcal{L}^2$ is explicitly causally consistent
2. $\mathcal{L}^2$ is X consistent
3. $\mathcal{L}^2$ is FIFO consistent
4. $\forall u \in A, v \in B.\ u \| v \cap L \neq \varnothing$

(1) $\Rightarrow$ (2) $\Rightarrow$ (3) By definition of X.

(3) $\Rightarrow$ (4) Let $u \in A$ and $v \in B$. We construct an execution e in $\mathcal{L}^2$ as follows. We call the two sites t_A and t_B, with $t_A, t_B \in \mathbb{T}$. Site t_A first executes a Choice operation and updates its state to the initial state of A. Similarly, Choice is also executed on t_B, thus updating the state to the initial state of B.

Consider an accepting run of u in A. For each transition (q, a, q') of the run, we execute a m_a operation is executed on t_A, thus updating the state to q'. At the end of the simulation of u, site t_A is in a state of A which is final. We do the same simulation on t_B, for the word v, which brings t_B in a state of B which is final.

Then, $\mathsf{End_B}$ is executed on site t_B, and as a result, a message $\mathsf{Finished_B}$ is broadcast to t_A. Upon reception, and since t_A is in a final state of A, its state is update to q_{check}. Finally, method Check is executed on A, and returns $\top$, because t_A is in state q_{check}.

Since this execution is FIFO consistent, there must exist a local interpretation order for this last $\mathsf{Check} \rhd \top$ operation. The specification of $\mathsf{Check} \rhd \top$ is given as $L' \cdot \mathsf{End_B}$, thus the only $\mathsf{End_B}$ operation of the execution must belong to the local interpretation order. By the axioms of FIFO consistency, all operations executed by t_B prior to $\mathsf{End_B}$ must also be in the local interpretation order, and in the order they were executed. Finally, the local interpretation order must also contain all the operations made on t_A before Check. Moreover, since the resulting sequence must belong to $L' \cdot \mathsf{End_B}$, we obtain a shuffling of u and v which is in L.

(4) $\Rightarrow$ (1) Let e be an execution of $\mathcal{L}^2$. If it doesn't contain a $\mathsf{Check} \rhd \top$ operation, it is automatically explicitly causally consistent, as the specification for all other operations is $\mathbb{M}^*$.

Otherwise, let t_A be the site identifier of the site executing a $\mathsf{Check} \triangleright \top$ operation o. We know that method Check can only return value $\top$ when it is called in state q_{check}. Moreover, the only way for t_A to be in q_{check} is if it was previously in a final state of A, and received message $\mathsf{Finished_B}$ from the other site (call it t_B).

This means that, prior to o, t_A must have executed a sequence of operations corresponding to a word in $u \in A$. Similarly, the only way that site t_B could have sent message $\mathsf{Finished_B}$ is if method $\mathsf{End_B}$ was executed in a final state of B; meaning that prior to $\mathsf{End_B}$, t_B must have executed a sequence of operations corresponding to a word in $v \in B$.

By assumption, we know there exists a word in the shuffling of u and v which belongs to L. As a result, we can construct a local interpretation order for o, containing the $\mathsf{End_B}$ operation as well as all operations corresponding to u and v, to form a sequence which belongs to the specification $\mathcal{S}(\mathsf{Check} \triangleright \top) = L' \cdot \mathsf{End_B}$.

Moreover, there can only be one $\mathsf{Check} \triangleright \top$ operation. Indeed, after executing it, site t_A goes into state q_{lost}, while t_B also goes into state q_{lost} after sending message $\mathsf{Finished_B}$. Every method or message received never updates the state q_{lost}, so it not possible for one of the sites to go into state q_{check} again so that method Check returns $\top$. □

Remark 7. *Similarly to Section 7.8, our proof does not rely on the unboundedness of the communication channels, and we can assume that their size is bounded by* 1*.*

7.10 Summary

We presented in this chapter a general framework which can be used to define consistency criteria for distributed data structures, containing both safety and liveness properties. We showed how to define several criteria, such as eventual consistency, RYW consistency, FIFO consistency, and causal consistency.

Moreover, we studied the decidability of the verification for such criteria, when the number of sites is bounded. In particular, we showed that only eventual consistency is decidable, while all other criteria are not. More precisely, we show that any criterion stronger than RYW consistency and weaker than implicit causally consistency is not decidable, as well as any criterion stronger than FIFO consistency and weaker than explicit causally consistency. This leaves open the possibility that there exists a criterion stronger than RYW consistency and weaker than explicit causally consistency, which is incomparable to both FIFO consistency and implicit causally consistency for which the verification is decidable.

Our undecidability results only require 2 sites communicating through bounded channels, and using a sequential specification for the consistency criteria.

However, as we have shown for linearizability, obtaining hardness or undecidability results does not close the door to efficient verification techniques. In

particular, it could be the case that consistency criteria such as causal consistency, become decidable when looking at particular specifications which are used in practice, such as the Multi-Value Register, or even Stacks and Queues. We leave this for future work.

Chapter 8

Conclusion

8.1 Contributions

Overall, this work is about understanding observational refinement in the context of concurrent or distributed libraries, and about its automatic verification.

Our first contribution was to revisit the connection between observational refinement and linearizability, in a shared memory setting. We proved that linearizability and observational refinement are equivalent in the general case, thus extending a result of Filipovic et al. [21] to the case where executions can have pending operations. Moreover our study led to two additional characterizations, namely the fact that linearizability and observational refinement are equivalent to the inclusion of the execution sets of the libraries, as well as to the inclusion of the history sets. These results provide a new perspective on the study of observational refinement and linearizability, and open the doors to new methods, as we showed in the bug-detection techniques of Chapter 5.

Concerning the theoretical complexity of observational refinement, we have demonstrated that linearizability is undecidable when the number of threads calling the library is unbounded, and they are EXPSPACE-complete when the number of threads is bounded. We prove the membership in EXPSPACE even when the specification is given as another library, and the hardness is proven even in the case where the specification is sequential, and given as a deterministic finite automaton. These results closed problems left open by Alur et al. [3].

We then propose two approaches to mitigate the high complexity of linearizability in general, which do not rely on manually specifying the linearization points. We provide a bug-detection technique which is scalable and has a good coverage of linearizability violations. Our approach is based on our characterization of linearizability using the inclusion of the history sets of the libraries, as well as on fundamental properties of those histories. We use the fact that histories are interval orders, which have a well-defined notion of *interval length*. More specifically, we bound the interval length for which we check the inclusion of histories.

Concerning the theoretical aspects, we show that this enables to obtain decidability for linearizability, even for an unbounded number of threads. In practice, we show that our approach can be used for real implementations. First, we demonstrate that the interval length is a relevant bounding parameter, as all the violations we know of could be detected with an interval bound of 0 or 1. Moreover, when the interval bound is fixed, this approach scales well in the size of the executions, and does not suffer for the exponential blowup related to enumerating all histories. Both the theoretical and practical results are based on symbolic representation of interval orders using counters.

Since this technique is an underapproximation, it cannot be applied to prove that libraries are linearizable. We thus give another approach, where we study in more details particular specifications which are widely used. Such specifications typically include queues, stacks, registers, or mutexes. We proved that checking linearizability with respect to those four data structures is reducible in polynomial time to state reachability. This reduction avoids the exponential blowup present for linearizability in general, and we obtain a PSPACE complexity when the number of threads is bounded (instead of the EXPSPACE complexity for linearizability in general, when the specification can be an arbitrary regular language).

Using this reduction, we also obtain decidability for linearizability with respect to those four data structures, when the number of threads is unbounded. (When the specification can be an arbitrary regular language, we have proved that linearizability is undecidable.) Moreover, this reduction enables the use of existing safety-verification tools for linearizability. While this work only demonstrates the reduction to these four objects, we believe our methodology is general enough to be applied to other atomic objects such as semaphores.

In the context of message-passing systems, the state of the art is not as advanced as for linearizability. Ensuring linearizability in an available and partition-tolerant distributed system is not possible. Designers of such systems thus fall back to weaker consistency criteria, such as eventual consistency. We gave a framework to formalize weak consistency criteria, and used it to define eventual consistency, RYW consistency, FIFO consistency, and causal consistency, which can be applied to a wide class of distributed libraries, including ones using speculative executions and rollbacks.

We prove that eventual consistency is decidable for a fixed number of sites, as long as the specification is represented as a multiset automaton, a model we introduced for that purpose, and capable of representing usual specifications. For safety conditions which are stronger than the safety part of eventual consistency (namely RYW consistency, FIFO consistency, and causal consistency), we prove that the verification problem is undecidable, even when the specification is a regular specification, and even when there are only two sites. This suggests that, in order to get decidability for a finite number of sites or threads, we need either really weak criteria, such as eventual consistency, or really strong criteria, such as linearizability, with everything in-between being undecidable (see Fig 8.1).

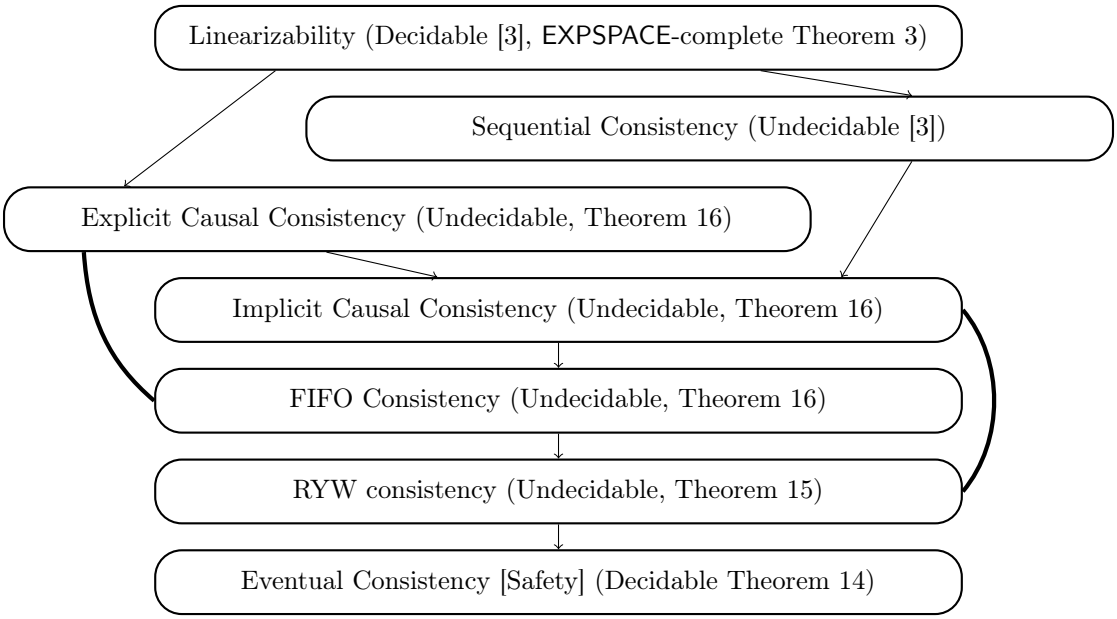


Figure 8.1: Decidability for the different consistency criteria, when considered for a bounded number of threads or sites. The arrows depict implication between the different criteria. A bold connection between two criteria means that all criteria in-between (present in the picture or not) are undecidable. Only verification for very strong or very weak criteria is decidable, while verification for the criteria in-between seems to always be undecidable.

8.2 Perspectives

Concerning the shared memory setting, we would like to improve the results of Chapter 6, which reduces linearizability to state reachability. Our goal is to define a syntax in which one could define any specification, and for which we could automatically derive the *bad patterns* that we found for the queues, stacks, registers, and mutexes. Even though our current approach is uniform, it still requires manual proofs specific to each data structure.

Furthermore, we didn't implement this technique on real implementations. We believe it would perform extremely well, as the bad patterns we provide to characterize non-linearizability seem relatively easy to detect (using regular automata). We would like to apply this technique in the context of proving that implementations are linearizable, and also combine it with the approach of bounding the interval length, in order to obtain an efficient bug-detection technique.

Concerning message-passing systems, we would like to explore the connection with notions of observational refinement, similarly to linearizability. Moreover, our results showed that the most widely used safety properties are all undecidable. However, as in the case of the undecidability of linearizability for an unbounded number of threads, we still believe it is possible to create efficient verification techniques, as shown in Chapters 5 and 6. For instance, we plan to study the problem of verifying these criteria with respect to specifications used in practice (such as the multi-value register), as the problem may be simpler than for arbitrary specifications, and may be decidable. In which case, we could obtain efficient algorithms for proving or disproving causal consistency or other criteria with respect to these particular specifications.

Bibliography

[1] P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. An integrated specification and verification technique for highly concurrent data structures. In *TACAS '13*. Springer, 2013.

[2] S. Almagor, U. Boker, and O. Kupferman. What's decidable about weighted automata? In T. Bultan and P.-A. Hsiung, editors, *Automated Technology for Verification and Analysis*, volume 6996 of *Lecture Notes in Computer Science*, pages 482–491. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-24371-4. doi: 10.1007/978-3-642-24372-1_37. URL `http://dx.doi.org/10.1007/978-3-642-24372-1_37`.

[3] R. Alur, K. L. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. *Inf. Comput.*, 160(1-2), 2000.

[4] D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV '07: Proc. 19th Intl. Conf. on Computer Aided Verification*, volume 4590 of *LNCS*, pages 477–490. Springer, 2007.

[5] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-5. doi: 10.1145/2463676.2465279. URL `http://doi.acm.org/10.1145/2463676.2465279`.

[6] L. Benmouffok, J.-M. Busca, J. M. Marquès, M. Shapiro, P. Sutra, and G. Tsoukalas. Telex: A semantic platform for cooperative application development. In *CFSE*, Toulouse, France, 2009.

[7] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and S. Sagiv. Thread quantification for concurrent shape analysis. In *CAV '08: Proc. 20th Intl. Conf. on Computer Aided Verification*, volume 5123 of *LNCS*, pages 399–413. Springer, 2008.

[8] A. Bouajjani and P. Habermehl. Constrained properties, semilinear systems, and petri nets. In *CONCUR '96: Proc. 7th Intl. Conf. on Concurrency Theory*, volume 1119 of *LNCS*, pages 481–497. Springer, 1996.

[9] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Verifying concurrent programs against sequential specifications. In *ESOP '13*. Springer, 2013.

[10] A. Bouajjani, C. Enea, and J. Hamza. Verifying eventual consistency of optimistic replication systems. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 285–296, 2014. doi: 10.1145/2535838.2535877. URL `http://doi.acm.org/10.1145/2535838.2535877`.

[11] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Tractable refinement checking for concurrent objects. In *POPL '15*. ACM, 2015.

[12] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. On reducing linearizability to state reachability. *CoRR*, abs/1502.06882, 2015. URL `http://arxiv.org/abs/1502.06882`.

[13] S. Burckhardt, A. Gotsman, and H. Yang. Understanding eventual consistency. Technical Report MSR-TR-2013-39, Microsoft Research.

[14] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-Up: a complete and automatic linearizability checker. In *PLDI '10*, pages 330–340. ACM, 2010.

[15] S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In *ESOP '12: Proc. 21st European Symp. on Programming*, volume 7211 of *LNCS*, pages 87–107. Springer, 2012.

[16] J. Burnim, G. C. Necula, and K. Sen. Specifying and checking semantic atomicity for multithreaded programs. In *ASPLOS '11: Proc. 16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 79–90. ACM, 2011.

[17] S. Chordia, S. Rajamani, K. Rajan, G. Ramalingam, and K. Vaswani. Asynchronous resilient linearizability. In Y. Afek, editor, *Distributed Computing*, volume 8205 of *Lecture Notes in Computer Science*, pages 164–178. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-41526-5. doi: 10.1007/978-3-642-41527-2_12. URL `http://dx.doi.org/10.1007/978-3-642-41527-2_12`.

[18] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, 2007.

[19] M. Dodds, A. Haas, and C. M. Kirsch. A scalable, correct time-stamped stack. In *POPL '15*. ACM, 2015.

[20] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Taşıran. Simplifying linearizability proofs with reduction and abstraction. In *TACAS '10: Proc. 16th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *LNCS*, pages 296–311. Springer, 2010.

[21] I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52), 2010.

[22] B. Fischer, O. Inverso, and G. Parlato. CSeq: a concurrency pre-processor for sequential C verification tools. In *ASE '13: Proc. 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 710–713. IEEE, 2013.

[23] M. Fürer. The complexity of the inequivalence problem for regular expressions with intersection. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 234–245, London, UK, UK, 1980. Springer-Verlag. ISBN 3-540-10003-2. URL `http://dl.acm.org/citation.cfm?id=646234.682559`.

[24] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM J. Comput.*, 26(4), 1997.

[25] S. Gilbert and N. A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33 (2):51–59, 2002.

[26] T. L. Greenough. *Representation and Enumeration of Interval Orders and Semiorders*. PhD thesis, Dartmouth College, 1976.

[27] R. Guerraoui and E. Ruppert. A paradox of eventual linearizability in shared memory. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 40–49, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2944-6. doi: 10.1145/2611462.2611484. URL `http://doi.acm.org/10.1145/2611462.2611484`.

[28] J. Hamza. On the complexity of linearizability. In *NETYS '15*. Springer, 2015.

[29] T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In *CONCUR '13*. Springer, 2013.

[30] M. Herlihy and N. Shavit. *The art of multiprocessor programming*.

[31] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.

[32] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[33] H. Hunt. *The Equivalence Problem for Regular Expressions with Intersection is Not Polynomial in Tape*. Department of Computer Science: Technical report. Cornell University, Department of Computer Science, 1973. URL `http://books.google.fr/books?id=52j6HAAACAAJ`.

[34] P. R. Johnson and R. H. Thomas. The maintenance of duplicate databases. Technical Report Internet Request for Comments RFC 677, Information Sciences Institute, January 1976.

[35] A.-M. Kermarrec, A. I. T. Rowstron, M. Shapiro, and P. Druschel. The icecube approach to the reconciliation of divergent replicas. In *PODC*, pages 210–218, 2001.

[36] D. Krob. The equality problem for rational series with multiplicities in the tropical semiring is undecidable. In W. Kuich, editor, *Automata, Languages and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 101–112. Springer Berlin Heidelberg, 1992. ISBN 978-3-540-55719-7. doi: 10.1007/3-540-55719-9_67. URL `http://dx.doi.org/10.1007/3-540-55719-9_67`.

[37] D. Kroening and M. Tautschnig. CBMC - C bounded model checker - (competition contribution). In *TACAS '14: Proc. 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *LNCS*, pages 389–391. Springer, 2014.

[38] A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1): 73–97, 2009.

[39] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model checking linearizability via refinement. In *FM '09*, volume 5850 of *LNCS*, pages 321–337.

[40] A. J. Mayer and L. J. Stockmeyer. The complexity of word problems - this time with interleaving. *Inf. Comput.*, 115(2):293–311, Dec. 1994. ISSN 0890-5401. doi: 10.1006/inco.1994.1098. URL `http://dx.doi.org/10.1006/inco.1994.1098`.

[41] M. M. Michael. ABA prevention using single-word instructions. Technical Report RC 23089, IBM Thomas J. Watson Research Center, January 2004.

[42] J. Michaux, X. Blanc, M. Shapiro, and P. Sutra. A semantically rich approach for collaborative model edition. In *SAC*, pages 1470–1475, 2011.

[43] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989. ISSN 0018-9219. doi: 10.1109/5.24143.

[44] P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In *PODC '10*, pages 85–94. ACM.

[45] I. Rabinovitch. The dimension of semiorders. *Journal of Combinatorial Theory, Series A*, 25(1):50 – 61, 1978. ISSN 0097-3165. doi: http://dx.doi.org/10.1016/0097-3165(78)90030-4. URL `http://www.sciencedirect.com/science/article/pii/0097316578900304`.

[46] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37 (1):42–81, 2005.

[47] O. Shacham, N. G. Bronson, A. Aiken, M. Sagiv, M. T. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. In *OOPSLA '11: Proc. 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 51–64. ACM, 2011.

[48] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, Jan. 2011.

[49] O. Sinnen. *Task Scheduling for Parallel Systems*. 2007.

[50] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, 29(5):172–182, Dec. 1995.

[51] V. Vafeiadis. Automatically proving linearizability. In *CAV '10*. Springer, 2010.

[52] R. Valk and M. Jantzen. The residue of vector sets with applications to decidability problems in petri nets. *Acta Inf.*, 21:643–674, 1985.

[53] P. Černý, A. Radhakrishna, D. Zufferey, S. Chaudhuri, and R. Alur. Model checking of linearizability of concurrent list implementations. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 465–479, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14294-X, 978-3-642-14294-9. doi: 10.1007/978-3-642-14295-6_41. URL `http://dx.doi.org/10.1007/978-3-642-14295-6_41`.

[54] M. T. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI '08: Proc. ACM SIGPLAN 2008 Conf. on Programming Language Design and Implementation*, pages 125–135. ACM, 2008.

[55] L. Wang and S. D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *PPOPP '05: Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 61–71. ACM, 2005.

[56] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *POPL '86: Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 184–193. ACM Press, 1986.

[57] S. J. Zhang. Scalable automatic linearizability checking. In *ICSE '11*, pages 1185–1187. ACM.

[58] O. Zomer, G. Golan-Gueta, G. Ramalingam, and M. Sagiv. Checking linearizability of encapsulated extended operations. In Z. Shao, editor, *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 311–330. Springer Berlin Heidelberg, 2014. ISBN 978-3-642-54832-1. doi: 10.1007/978-3-642-54833-8_17. URL `http://dx.doi.org/10.1007/978-3-642-54833-8_17`.