Universidad Nacional Autónoma de México

Posgrado en Ciencias e Ingeniería de la Computación

Verificación de la linealizabilidad en tiempo de ejecución

Tesis

QUE PARA OPTAR POR EL GRADO DE:

Maestra en Ciencia e Ingeniería en Computación

PRESENTA:

Ing. Gilde Valeria Rodríguez Jiménez

DIRECTOR DE TESIS:

Dr. Armando Castañeda Rojano / Facultad de Ciencias

Ciudad universitaria, CDMX, Octubre, 2022

Agradecimientos

Al Dr. Armando Castañeda Rojano que fue director de este trabajo de tesis por su apoyo, asesoría, accesibilidad y atención.

A mis sinodales por su tiempo y comentarios que ayudaron a mejorar este trabajo de tesis.

Al Conacyt por la beca otorgada durante mis estudios de maestría.

A Victoria Itzayana que fue mi asesora creativa en este trabajo de tesis por ayudarme a estilizar mis ideas y hacerme el trayecto ameno y sencillo.

A Guadalupe, a Atala y a Victoria por brindarme siempre su cariño, apoyo y amor.

A Victoria por mostrarme siempre lo importante de la vida.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Las arquitecturas de computadoras multinúcleo tuvieron un auge muy fuerte en la industria debido a que fueron diseñadas para evadir la imposibilidad de disipar el calor que genera un solo *core* o *CPU* que trabaja a altas velocidades. Estas arquitecturas se pueden analizar en un modelo de cómputo concurrente, en este modelo también se pueden analizar otros problemas como el desarrollo de arquitecturas de *microservicios* [28] o el desarrollo de sistemas de blockchain [24].

Para poder resolver estos problemas se requiere diseñar e implementar algoritmos concurrentes que cumplan con cierto rendimiento, correctez, escalabilidad, disponibilidad y tolerancia a fallas.

Los algoritmos concurrentes son difíciles de diseñar e implementar [22,32, 36] debido a la asincronía, porque ocasiona que exista un número exponencial de posibles ejecuciones a considerar. Además entre más se intenta mejorar el desempeño de los algoritmos concurrentes, más difícil es razonar acerca de su corrección y al requerir una condición de corrección fuerte, se puede afectar su desempeño y como consecuencia, su disponibilidad.

Para evadir estas dificultades los algoritmos concurrentes que se desarrollan suelen ser complejos y sofisticados, y por lo tanto, son propensos a fallar [32]. Debido a esto es importante verificar que estos algoritmos concurrentes sean correctos.

1.1. Contexto

En un modelo de cómputo concurrente un conjunto de procesos pueden modificar al mismo tiempo un objeto compartido. Al conjunto de algoritmos locales, también llamados programas secuenciales, de cada proceso se le conoce como algoritmo concurrente.

La corrección de un algoritmo concurrente es diferente a la corrección de un programa secuencial.

Un programa secuencial puede ser descrito por una especificación secuencial debido a que como un único proceso realiza operaciones a un objeto, se puede describir el estado del objeto antes y después de cada operación. Sin embargo, si un conjunto de procesos aplica operaciones de forma concurrente a un solo objeto, entonces puede ser que algunas operaciones se traslapen y no sea posible obtener el estado del objeto entre cada par de operaciones.

Para analizar la corrección de un algoritmo concurrente, Lamport plantea en [30] obtener un orden parcial de las operaciones de un algoritmo concurrente. Este orden es parcial porque a veces es imposible saber el orden en el cual ciertas operaciones ocurren. Además plantea que un orden parcial se puede extender a un orden total, siendo un orden parcial una historia concurrente y un orden total una historia secuencial, por lo tanto, es posible comparar un orden total con una especificación secuencial.

Herlihy y Wing establecen la condición de corrección de linealizabilidad [27], esta consiste en extender un orden parcial a un orden total. Para realizarlo la regla, a grandes rasgos, consiste en que si una llamada a un método precede a otra, entonces la llamada que sucedió antes debe tomar efecto en el orden total antes que la llamada que sucedió después. En cambio si dos llamadas suceden al mismo tiempo en el orden parcial, entonces pueden ordenarse de la manera que más convenga en el orden total. Si la historia secuencial obtenida (el orden total) es válida con respecto a la especificación secuencial de un objeto O, se dice que el algoritmo concurrente es linealizable con respecto al objeto O.

La linealizabilidad es una condición de corrección muy relevante debido a que muchos algoritmos basan su corrección en si son linealizables o no con respecto a un objeto O [16]. Para garantizar que estos algoritmos cumplan con esta condición se han desarrollado métodos de verificación de la lineali-

zabilidad [3, 11, 12, 16].

Algunas técnicas fundamentales para la verificación de la linealizabilidad son prueba de teoremas, verificación de modelos, testeo y verificación en tiempo de ejecución [31]. En la verificación de sistemas comerciales se utiliza una combinación de estás técnicas de verificación porque los errores pueden ser muy costosos, un ejemplo de ello se muestra en [13]. Sin embargo, en este trabajo nos enfocamos en la técnica de la verificación en tiempo de ejecución.

La verificación de la linealizabilidad en tiempo de ejecución consiste en que dado un sistema a verificar, que suele ser un algoritmo que se especula es linealizable, y un sistema verificador, el sistema verificador determina en tiempo de ejecución si la ejecución actual del sistema a verificar es linealizable con respecto a un objeto O.

Los sistemas verificadores pueden ser centralizados o distribuidos, resolver o no el consenso y suelen estar desacoplados, es decir, que se diseñan y despliegan sin ser considerados por el sistema a verificar [10].

En este trabajo nos interesa cuando el sistema verificador es distribuido. Cuando el sistema verificador es capaz de resolver el consenso puede obtener un único estado global sobre la ejecución actual que se ejecuta y todos los procesos involucrados devuelven un mismo veredicto. O dado un conjunto de veredictos resuelven el consenso sobre el estado global de la ejecución actual. En cambio, si el sistema verificador no es capaz de resolver el consenso, entonces los procesos pueden obtener distintos veredictos. En [21] se estudia el número mínimo de veredictos de los procesos para decidir si la ejecución actual es correcta; este número depende de la condición de corrección.

Para poder caracterizar soluciones al problema de la verificación de la linealizabilidad en tiempo de ejecución, se propone una formalización a este problema. La cual revela que no existe un sistema verificador que resuelva este problema, incluso si utiliza el cómputo más poderoso; es decir, operaciones primitivas con un número de consenso infinito (teorema 8). Un ejemplo de esta imposibilidad se presenta en la figura 1.1 en la cual el sistema verificador detecta una historia (que se muestra en color azul) en donde las operaciones op_p y op_q que realizan los procesos p y q, respectivamente, son concurrentes. Sin embargo, en el orden real (en color rojo) la operación op_p precede a la operación op_q. El sistema verificador no tiene forma de saber que la historia que detectó no es la historia que sucedió en realidad y puede devolver un veredicto erróneo.

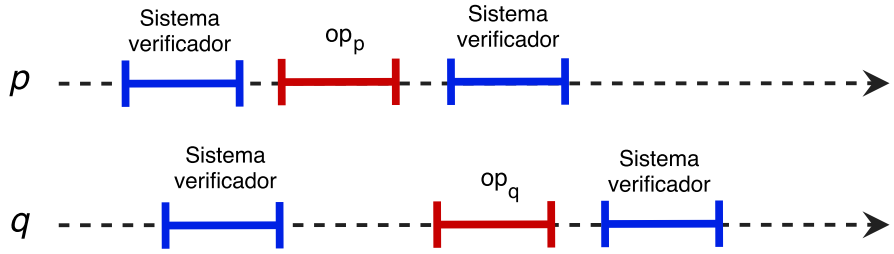


Figura 1.1: Ejemplo de una ejecución en la que un sistema verificador detecta una historia distinta a la que sucedió en realidad

Al analizar esta imposibilidad se puede entender el problema de la verificación en tiempo de ejecución como un problema particular a la imposibilidad de obtener el orden en el que ocurren ciertos eventos debido a la relación con la teoría de la relatividad que plantea Lamport en [30] y que es abordado en [8] al plantear que todo evento ocurre siempre en un intervalo de indistinguibilidad.

A pesar de que este problema no es soluble, es posible relajarlo y solucionarlo, así como muchos otros problemas en las ciencias de la computación abordados en [4].

1.2. Motivación

En [5] definen una clase de objetos que tienen implementaciones libre de esperas, para ello definen un grafo de precedencia, el cual es un grafo acíclico dirigido que representa el orden parcial de las operaciones de alguna historia. Este grafo utiliza una implementación de un objeto de tipo snapshot atómico.

Se pensó que como el mecanismo para obtener un grafo presentado en [5] permite construir ordenes parciales, entonces este mecanismo se puede utilizar para detectar la ejecución de un algoritmo que se quiera verificar en tiempo real. Este mecanismo no solo permitiría detectar una ejecución en tiempo real, sino que podría hacerlo de forma eficiente debido a que en la actualidad existen implementaciones de objetos de tipo snapshot que toman un tiempo lineal.

Así que como verificar la linealizabilidad de un algoritmo con la técnica de verificación de modelos es muy difícil computacionalmente debido a que se tienen que considerar todas las ejecuciones posibles de un algoritmo

concurrente, posiblemente comprobar unas cuantas ejecuciones en tiempo de ejecución una vez el algoritmo esté desplegado permita analizar unas cuantas ejecuciones en tiempo de ejecución para determinar si el algoritmo es linealizable hasta el momento. Además podría ser posible desplegar un mecanismo de corrección de errores.

1.3. Objetivos

El objetivo de este trabajo es exponer un análisis teórico de la verificación de la linealizabilidad utilizando la técnica de verificación en tiempo de ejecución, lo cual permite:

- Comprender el contexto de la verificación de la linealizabilidad para situar la técnica de verificación de la linealizabilidad en tiempo de ejecución: usos, ventajas, desventajas y soluciones planteadas.
- Construir una formalización del problema de la verificación de la linealizabilidad en tiempo de ejecución para plantear las soluciones.
- A partir de la demostración de imposibilidad de la verificación de la linealizabilidad en tiempo de ejecución, comprender que este problema no es soluble incluso cuando se utilicen operaciones primitivas con número de consenso infinito.
- A partir de la demostración de imposibilidad de la verificación de la linealizabilidad en tiempo de ejecución, evadir este problema al formalizar un problema más relajado que sí es soluble.

1.4. Justificación

La verificación de la linealizabilidad es un problema muy significativo, porque la linealizabilidad en una condición de corrección sumamente utilizada en la industria, en parte porque tiene las propiedades de ser composicional y no bloqueante. Estas permiten diseñar sistemas de forma modular, lo cual es importante en sistemas de gran envergadura. Aún más, existen librerías en lenguajes de programación como *JAVA* que tienen implementaciones linealizables de estructuras de datos como colas y pilas [26].

Verificar la linealizabilidad con la técnica de prueba de teoremas, es decir, de forma manual, puede incurrir en errores humanos y de un análisis particular y riguroso de cada algoritmo que se quiera verificar.

Cuando se verifica la linealizabilidad con la técnica de verificación de modelos, se verifican todas las posibles ejecuciones de un modelo que representa el sistema a verificar [31]. Verificar la linealizabilidad de forma automática es indecidible cuando no se acota el número de procesos [11]; aún más, verificar siquiera una historia de una ejecución cuando no se acota el número de procesos es **NP**-*completo* [1].

Las técnicas de verificación de modelos y de prueba de teoremas se suelen referir a un modelo del sistema real que se está verificando y no se pueden aplicar directamente a la implementación real. Un modelo de un sistema refleja los aspectos más relevantes del sistema; sin embargo, debido a la complejidad de los algoritmos concurrentes, la implementación del sistema puede comportarse diferente a lo que se espera del modelo.

Debido a esto, no podemos estar completamente seguros de que un sistema es realmente linealizable a pesar de que se haya verificado su modelo. Esto evidencia la importancia de la técnica de verificación en tiempo de ejecución, que permite verificar el comportamiento real del sistema sin tener información previa del mismo, como en la técnica de testeo. Además puede permitir desplegar mecanismos de detección y de corrección de errores [10, 31].

Ya existen soluciones al problema de la verificación de la linealizabilidad en tiempo de ejecución. Sin embargo, algunas soluciones presentan resultados inconsistentes e inexplicables [18]; otras soluciones se apoyan en relojes globales que interfieren con el progreso del sistema a verificar [17]; y otras se apoyan en que son correctas si se cumplen oráculos que eligen un veredicto que representa lo que realmente sucedió en la ejecución actual [33].

1.5. Contribución

El problema de la verificación de la linealizabilidad en tiempo de ejecución es imposible de resolver incluso cuando se utilicen operaciones primitivas con un número de consenso infinito (teorema 8).

La imposibilidad de este problema se relaciona con la imposibilidad de distinguir el orden de ciertos eventos que ocurren en una ejecución [30]. Por ello no importa si un sistema verificador resuelve o no el consenso ni si es centralizado o distribuido, de cualquier forma no es posible saber si la ejecución

detectada es en realidad la ejecución que sucedió.

Formalizar este problema y demostrar que es imposible de resolver nos brinda herramientas para relajarlo y proponer una solución. Además de una perspectiva del porqué la verificación de la linealizabilidad en tiempo de ejecución no es soluble, resolviendo el por qué un mismo algoritmo sobre una misma ejecución da resultados inconsistentes [18], por qué es más factible solucionarlo en un sistema síncrono [17] y por qué se necesita de un oráculo [33].

1.6. Organización de la Tesis

En el capítulo 2 se presenta el modelo de cómputo concurrente en el que tiene lugar este trabajo. En el capítulo 3 se muestra un análisis del estado del arte de la verificación de la linealizabilidad; algunas técnicas de verificación sobre la verificación de la linealizabilidad junto con sus trabajos, ventajas y desventajas. Se profundiza en la verificación de la linealizabilidad en tiempo de ejecución y se abordan algunos desafíos y soluciones propuestas.

La contribución de este trabajo se desarrolla en los capítulos 4 y 5. En el capítulo 4 se formaliza el problema de la verificación de la linealizabilidad en tiempo de ejecución y se presenta una prueba de imposibilidad a este problema. En el capítulo 5 se desarrolla un problema relajado al problema de la verificación de la linealizabilidad en tiempo de ejecución (VLTE): el problema de la verificación débil de la linealizabilidad en tiempo de ejecución (VDLTE), el cual consiste en relajar la propiedad de completez de la formalización de la VLTE; además se presenta un algoritmo que resuelve la VDLTE. Finalmente en el capítulo 6 se presentan las conclusiones del trabajo.

Capítulo 2

Modelo de cómputo concurrente

La verificación de la linealizabilidad de algoritmos concurrentes se puede analizar en un modelo de cómputo concurrente. En este modelo se pueden diseñar algoritmos concurrentes y analizar su corrección a partir de las condiciones de progreso y de corrección. En este trabajo nos enfocamos en la linealizabilidad como condición de corrección. La verificación básicamente consiste en comprobar que los algoritmos concurrentes cumplen con la linealizabilidad.

En este capítulo se define un modelo de cómputo secuencial el cual es un antecedente al modelo de cómputo concurrente. Posteriormente se presenta la definición formal de un modelo de cómputo concurrente, se definen los algoritmos concurrentes y se profundiza en su corrección: la condición de corrección de la linealizabilidad y las condiciones de progreso bloqueantes y no bloqueantes. Después se define un modelo básico de memoria compartida, el registro, y se presenta un ejemplo de este modelo: el Snapshot atómico. Por último, se describen las operaciones de sincronización y su poder de sincronización.

2.1. Modelo de cómputo secuencial

Los fundamentos del cómputo secuencial fueron establecidos en la década de 1930 por Alan Turing y Alonzo Church. Ambos, de forma independiente, formularon lo que se conoce como la *Tesis Church-Turing*. La cual estipula que si un problema no puede ser resuelto por una *Máquina de Turing* o por el *Calculo Lambda de Church*, se considera que universalmente el problema

no tiene solución. Esta es una tesis y no un teorema porque involucra las nociones de *computabilidad* y de *calculabilidad efectiva*, y esta última no puede ser definida de forma rigurosamente matemática [26].

Una máquina de Turing es, a grandes rasgos, la intuición de que la computación consiste en aplicar reglas de forma mecánica para manipular números, en la que la persona o máquina que realiza la manipulación utiliza un cuaderno de notas para realizar los cálculos [4]. Una implementación de la máquina de Turing es la arquitectura presentada por Von Neumann en [39]. En ella el cómputo se realiza en la unidad de procesamiento según la unidad de control mientras que los datos y el resultado del cómputo se encuentra en una unidad de almacenamiento.

2.1.1. Objetos secuenciales

En una arquitectura de computadora secuencial, el cómputo se realiza secuencialmente; es decir, existen secuencias de operaciones que modifican los datos en la memoria. En esta propuesta representaremos a los datos en la memoria como *objetos secuenciales*.

Cada objeto tiene un tipo, el cual define un conjunto de valores posibles y un conjunto de métodos que proveen las únicas herramientas para crear y manipular al objeto. El tipo de un objeto es análogo a una clase en lenguajes de programación orientados a objetos.

La implementación de un objeto es un programa o algoritmo secuencial que cumple con las especificaciones de un tipo de objeto determinado. Podemos describir a una implementación como un objeto en concreto y una especificación como un objeto abstracto.

En este modelo las operaciones son invocadas por un solo proceso a la vez, lo cual implica que las operaciones, el significado de ellas, puedan tener precondiciones y postcondiciones [27]. Una operación se define como una *llamada a un método* del objeto. Una secuencia, finita o infinita, de operaciones se denota como una *ejecución*.

2.1.1.1. Objeto de tipo cola

Un ejemplo de un tipo de objeto secuencial es una *cola*. Una cola es una colección ordenada de elementos que provee dos métodos principales: $enq()$ y $deq()$. Los cuales satisfacen la propiedad de *FIFO*, esta propiedad implica que el primer elemento en entrar a la cola es el primer elemento en salir de

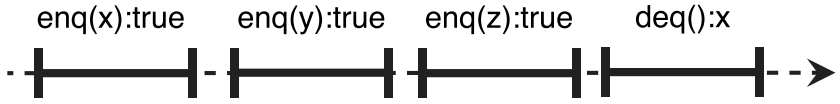
Algoritmo 1 Implementación de un objeto de tipo cola

```
1: Objeto Nodo(nuevoValor, nuevoAnterior):
2:     S valor = nuevoValor
3:     Nodo anterior = nuevoAnterior
4: Método inicializacion():
5:     Nodo tail = Nodo(null, null)
6: Método enq(nuevoValor) :
7:     If no está inicializado then
8:         inicializacion()
9:     Nodo nuevoNodo = Nodo(nuevoValor, tail.anterior)
10:    tail.anterior = nuevoNodo
11:    return true
12: Método deq() :
13:    If la cola no está vacía then
14:        While nodo.anterior.anterior ≠ null
15:            nodo = nodo.anterior
16:        valor = nodo.anterior.valor
17:        nodo.anterior = null
18:        return valor
```

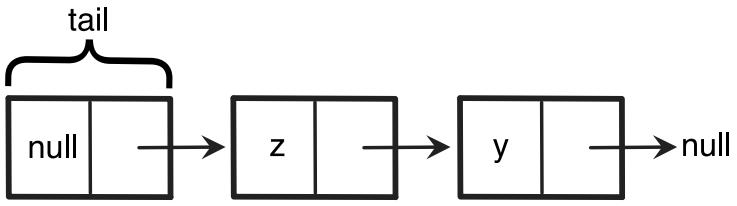
ella. De tal forma que el método $enq(x)$ consiste en insertar un elemento x al final de la cola y el método $deq()$ consiste en eliminar el elemento que se encuentra al inicio de la cola.

Los objetos secuenciales tienen un *estado bien definido*. Por ejemplo, en el algoritmo 1 se muestra una implementación de un objeto de tipo cola; en el caso de una cola su estado bien definido es la secuencia de elementos ordenados en ella. Si una ejecución es la siguiente secuencia ordenada de operaciones: $enq(x)$, $enq(y)$, $enq(z)$ y $deq()$, como se muestra en la subfigura 2.1a, entonces su estado bien definido al finalizar la ejecución se muestra en la subfigura 2.1b.

La *precondición* de un objeto describe el estado de objeto antes de una llamada a un método. La *postcondición* de un objeto describe el estado del objeto después de la llamada al método junto con el valor que regresa esa llamada. Por ejemplo, si el estado de la cola es el que se muestra en la figura 2.2 (precondición), entonces llamar a la última operación de la ejecución $deq()$ deja el estado de la subfigura 2.1b (postcondición); además se denota como



(a) Ejecución de la cola



(b) Estado bien definido de la cola después de la ejecución

Figura 2.1: Ejemplo de una ejecución de una implementación de un objeto de tipo cola

un *efecto secundario* que el elemento con el valor x se elimine de la cola.

A este estilo de documentación del objeto se le conoce como *especificación secuencial*, esta consiste en describir el estado del objeto antes y después de cada operación, es una forma de argumentar si una historia secuencial es *válida*.

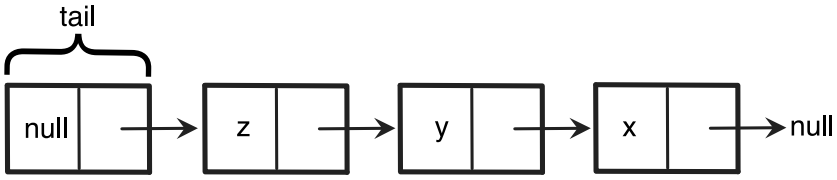


Figura 2.2: Estado bien definido antes de la llamada a *deq*(): precondición

Definir objetos así solo es posible en el modelo secuencial, ya que si en un modelo concurrente un conjunto de procesos aplican operaciones sobre un mismo objeto concurrente, entonces las operaciones se pueden traslapar y puede no ser posible obtener el estado del objeto entre cada par de operaciones.

De hecho, una especificación secuencial de un objeto concurrente es simplemente el conjunto de historias secuenciales *válidas* para el objeto.

2.1.2. Modelo de cómputo concurrente

Los conceptos de esta sección fueron tomados del libro de Herlihy, M y Shavit, N: *The art of multiprocessor programming* en [26].

Un modelo concurrente de memoria compartida está conformado por procesos que se comunican a través de estructuras de datos compartidas llamadas objetos compartidos. Cada proceso tiene su propio programa secuencial o algoritmo local. A la colección de algoritmos locales se le denota como *algoritmo* o *implementación concurrente*.

El que los procesos sean secuenciales quiere decir que cada proceso aplica una secuencia de operaciones a los objetos, alternando la invocación y la respuesta a la invocación. Los procesos son asíncronos, esto implica que corren a diferentes velocidades y que cualquier proceso puede detenerse en cualquier momento por un intervalo de tiempo impredecible. Esta noción de asincronía refleja la realidad de los sistemas concurrentes en los que los retrasos son impredecibles desde microsegundos (fallos de caché), a milisegundos (fallos de página), a segundos (programación interrumpida).

2.2. Comportamiento de los algoritmos concurrentes

En contraste con la especificación secuencial de los algoritmos secuenciales, los algoritmos concurrentes tienen dos aspectos: la seguridad que garantiza que *nada malo suceda*, y la viveza que garantiza que eventualmente *algo bueno sucederá*. Al término que garantiza la viveza de los algoritmos se le denota como *condición de progreso*, y al termino que garantiza la seguridad de los algoritmos se le denota como *condición de corrección* [36].

Por ejemplo, para realizar una implementación concurrente de una cola, necesitamos coordinar los procesos para garantizar el progreso de los procesos y mantener las especificaciones secuenciales de un objeto abstracto de tipo cola.

2.2.1. Condiciones de corrección

El enfoque estándar para argumentar las propiedades de seguridad es especificar secuencialmente las propiedades del objeto y encontrar una manera

de asignar las ejecuciones concurrentes a ejecuciones secuenciales "correctas" o válidas. Existen varias condiciones de corrección las cuales son apropiadas para diferentes algoritmos; algunas de ellas son la *consistencia secuencial*, *la serializabilidad*, la *linealizabilidad*, entre otras [36].

En este trabajo se utilizará la linealizabilidad como noción de corrección.

2.2.1.1. Linealizabilidad

Antes de definir la linealizabidad se explicarán algunos conceptos fundamentales del modelo de cómputo concurrente.

En el modelo concurrente una llamada a un método se define como un intervalo que inicia con un *evento de invocación* y termina con un *evento de respuesta*. Una *ejecución* de un algoritmo concurrente es una secuencia, finita o infinita, de eventos en la cual se empieza con una configuración inicial y cada proceso aplica eventos y cambia su estado de acuerdo con su algoritmo. Una configuración describe el estado del sistema concurrente, el valor de cada objeto compartido y el estado de cada proceso.

Una ejecución de un algoritmo concurrente se modela por una *historia*, la cual es una secuencia finita de eventos de invocación y respuesta de métodos. El evento de invocación de un método lo denotamos como $\langle X.m(a^*)p\rangle$; donde X es un objeto compartido, m es el nombre del método, a^* es una secuencia de argumentos y p es un proceso.

El evento de respuesta de un método lo denotamos como $\langle X : t(r^*)p\rangle$; donde X es un objeto compartido, t es la confirmación o el nombre de una excepción, r^* es una secuencia de resultados y p es un proceso.

Por ejemplo, en la figura 2.3 se muestra una ejecución concurrente de una implementación de una cola de dos procesos, p y q, sobre un objeto compartido Q. La primera operación del proceso p se compone de un evento de invocación, $\langle Q.enq(y)\, p\rangle$, y un evento de respuesta, $\langle Q : true\, p\rangle$.

Una llamada a un método en una historia H es un par de eventos que consiste de un evento de invocación y un evento de respuesta *coincidentes* en H. Se dice que un evento de invocación y un evento de respuesta son coincidentes en H si comparten el mismo objeto y el mismo proceso. Por ejemplo, los eventos $\langle Q.enq(y)\, p\rangle$ y $\langle Q : true\, p\rangle$ son coincidentes.

Por el contrario, decimos que una llamada a un método está *pendiente* si el evento de invocación no tiene un evento de respuesta que coincida. Una

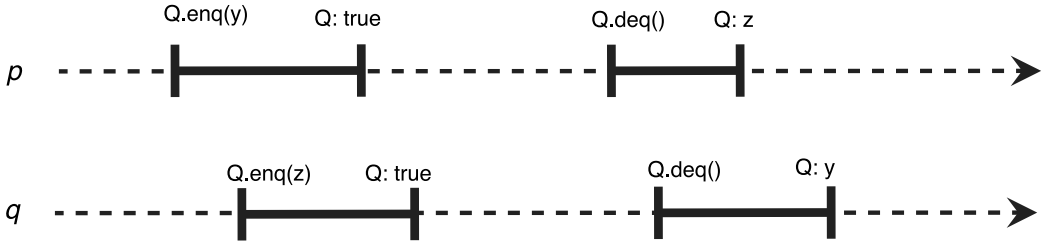


Figura 2.3: Ejecución concurrente de una implementación de una cola: $\langle Q.enq(y)p\ \rangle \ \rightarrow \langle Q.enq(z)q\ \rangle \ \rightarrow \langle Q\ :\ true\ \ p\rangle \ \rightarrow \langle Q\ :\ true\ \ q\rangle \rightarrow \langle Q.deq()\ \ p\rangle \rightarrow \langle Q.deq()\ \ q\rangle \rightarrow \langle Q.z\ \ p\rangle \rightarrow \langle Q.y\ \ q\rangle$

extensión de H se construye agregando eventos de respuesta a cero o más invocaciones pendientes de H. Una historia $completa(H)$ es una subsecuencia de H que consiste de todas las invocaciones y respuestas coincidentes; se construye a partir de ignorar todos los eventos de invocación pendientes.

En algunas historias las llamadas pueden traslaparse. Una historia H es *secuencial* si el primer evento de H es una invocación y a cada invocación, excepto posiblemente la última, le sigue inmediatamente su respuesta coincidente. Por ejemplo, la historia de la figura 2.1a es secuencial.

También nos podemos enfocar en procesos u objetos: una *subhistoria de un proceso*, $H|p$, es la historia H de la subsecuencia de todos los eventos en H cuyos nombres de proceso son p. Por ejemplo, en la figura 2.3 la historia del proceso p es $H|p = \langle Q.enq(y) : true\ \ p\rangle \ \rightarrow_{H_Q} \langle Q\ :\ deq()\ :\ z\ \ p\rangle$. Una *subhistoria de un objeto*, $H|X$, es la historia H de la subsecuencia de todos los eventos en H cuyos nombres de objeto son X.

Dos historias son *equivalentes* si para cada proceso p, $H|p = H'|p$.

Una historia está *bien formada* si cada subhistoria de proceso es secuencial. Todas las historias que se consideran en este trabajo son bien formadas.

Una historia secuencial H es válida si cada subhistoria del objeto concurrente es válida para el objeto concurrente.

Para entender cómo se relacionan las llamadas a métodos en la historia, se define un *orden parcial* denotado como "$\rightarrow$" y un *orden total* denotado como "$<$ ".

Un orden parcial $\rightarrow$ en un conjunto U es una relación irreflexiva y transitiva. De tal forma que es posible que dados distintos r, s y t en U nunca se cumple que $s \rightarrow s$ y cuando $s \rightarrow r$ y $r \rightarrow t$, entonces $s \rightarrow t$. Además es

posible que no se cumpla el orden $r \to s$ ni $s \to r$. En cambio, un orden total $<$ en U es un orden parcial tal que para todos los distintos r y s en U se cumple que $r < s$ o $s < r$.

Cualquier orden parcial puede ser extendido a un orden total de la siguiente forma: si $\to$ es un orden parcial en U, entonces existe un orden total $<$ en U tal que si $r \to s$, entonces $r < s$.

Decimos que cualquier llamada a un método m_0 precede a otra llamada a un método m_1 en la historia H en el orden parcial, lo cual es equivalente a $m_0 \to_H m_1$, si el evento de respuesta de m_0 denotado como e_{res_0} termina antes que el evento de invocación de m_1 denotado como e_{inv_1}, lo cual es equivalente a $e_{res_0} \to e_{inv_1}$ en el orden parcial. Como el orden parcial $m_0 \to_H m_1$ es una historia secuencial, entonces también es un orden total, $m_0 <_H m_1$. Sin embargo si $e_{inv_1} \to e_{res_0}$, entonces no se cumple ninguna relación de precedencia entre m_0 y m_1 en el orden parcial, lo cual implica que son concurrentes.

Vamos a abreviar una llamada a un método uniendo los eventos de invocación y de respuesta de la siguiente forma: dado el orden parcial de los eventos que componen una llamada $\langle Q.enq(y)\ p\rangle \quad \to \quad \langle Q\ :\ true\ p\rangle$, la denotaremos como $\langle Q.enq(y) : true\ p\rangle$.

Si modelamos la ejecución de la figura 2.3 como una historia H_Q, podemos construir un orden parcial en los eventos de la historia H_Q: $\langle Q.enq(y)p\rangle \to \langle Q.enq(z)q\rangle \to \langle Q : truep\rangle \to \langle Q : trueq\rangle \to \langle Q.deq()p\rangle \to \langle Q.deq()q\rangle \to \langle Q.z\ p\rangle \to \langle Q.y\ q\rangle$. Sin embargo, este orden parcial en los eventos no permite construir un orden parcial en las llamadas a métodos porque los eventos de invocación de las operaciones $\langle Q.enq(z) : true\ \ q\rangle$ y $\langle Q.deq() : y\ \ q\rangle$ preceden a los eventos de respuesta de las operaciones $\langle Q.enq(y) : true\ \ p\rangle$ y $\langle Q : deq() : z\ \ p\rangle$, respectivamente.

La idea básica detrás de la linealizabilidad es que toda historia concurrente es equivalente, de alguna forma, a una historia secuencial. A grandes rasgos, la regla consiste en que si una llamada a un método precede a otra, entonces la llamada que sucedió antes debe *tomar efecto* antes de la llamada que sucedió después. En cambio si dos llamadas a métodos se traslapan, entonces su orden es ambiguo y pueden ordenarse de la manera que más convenga.

Al momento en el que una llamada a un método toma efecto de forma instantánea se le conoce como *punto de linearización*.

De manera más formal,

Definición 1. *Una historia H es linealizable si tiene una extensión H' y existe una historia secuencial válida S tal que*

1. *completa(H') es equivalente a S, y*
2. *si una llamada a un método m_0 precede a otra llamada a un método m_1 en H, entonces la misma relación de precedencia se cumple en S.*

S es una linearización de H, siendo que H puede tener múltiples linerizaciones. De manera informal, extender H a H' captura la idea de que algunas invocaciones pendientes pueden tomar efecto aunque sus respuestas no existan en H.

Por ejemplo, la historia H_Q es linealizable, a continuación se muestran dos linearizaciones de H_Q, S_1 y S_2, las cuales están representadas en la figura 2.4.

$S_1 = \langle Q.enq(y) : true\ p\rangle \rightarrow_{H_Q} \langle Q.enq(z) : true\ q\rangle \rightarrow_{H_Q} \langle Q.deq() : y\ q\rangle \rightarrow_{H_Q} \langle Q : deq() : z\ p\rangle$, y

$S_2 = \langle Q.enq(z) : true\ q\rangle \rightarrow_{H_Q} \langle Q.enq(y) : true\ p\rangle \rightarrow_{H_Q} \langle Q.deq() : z\ p\rangle \rightarrow_{H_Q} \langle Q : deq() : y\ q\rangle$.

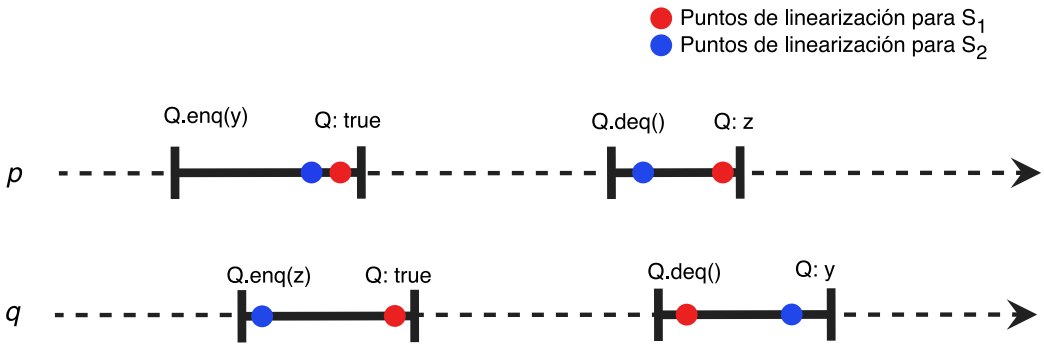


Figura 2.4: Puntos de linearización para obtener las linearizaciones S_1 y S_2

La linealizabilidad es composicional en relación a los objetos (Teorema 1) y a las ejecuciones (Teorema 2):

Teorema 1. *H es linealizable si y solo si para cada objeto X, $H|X$ es linealizable.*

Teorema 2. *H es linealizable si y solo si todas sus ejecuciones son linealizables.*

Esta noción permite que los sistemas concurrentes sean diseñados y desarrollados de forma modular, de modo que los objetos linealizables pueden ser implementados, verificados y ejecutados de manera independiente.

2.2.2. Especificaciones de progreso

Existen diferentes especificaciones o condiciones de progreso, algunas de estas son bloqueantes y algunas son no bloqueantes.

2.2.2.1. Bloqueantes

En las condiciones de progreso bloqueantes el retraso inesperado de cualquier proceso puede retrasar a otros; es decir, impedir que los demás procesos progresen.

Los métodos que tienen una condición de progreso bloqueante suelen estar basados en *candados*, los cuales funcionan a partir de la propiedad de la *exclusión mutua*. Esta propiedad, a grandes rasgos, implica que un bloque de código, *sección crítica*, pueda ser ejecutado por solo un proceso a la vez. De tal forma que un candado es una sección crítica que se puede adquirir y liberar. Por ejemplo, en una implementación concurrente bloqueante de una cola, si un proceso p ejecuta el método $enq(y)$, adquiere el candado y se detiene; después, si un proceso q ejecuta el método $deq()$, entonces este proceso q no será capaz de adquirir el candado y se detendrá mientras el proceso p se detenga, tal como se muestra en la figura 2.5 en donde ambas llamadas están pendientes.

Dos ejemplos de condiciones de progreso bloqueantes que utilizan este principio son *libre de punto muerto* (*deadlock-free*) y *libre de inanición* (*starvation-free*).

La condición de libre de punto muerto consiste en que si algún proceso trata de adquirir el candado, entonces algún proceso tendrá éxito en adqui-

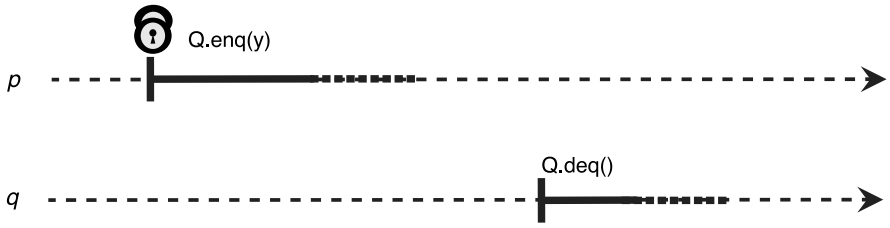


Figura 2.5: Ejemplo de una ejecución de una implementación concurrente bloqueante de una cola: $\langle Q.enq(y)p \ \rangle \ \rightarrow \langle Q.deq() \ q \rangle$

rirlo. La condición de libre de inanición consiste en que cada proceso que trate de adquirir el candado eventualmente tendrá éxito en adquirirlo.

Estas condiciones tienen la propiedad de ser *condiciones de progreso dependientes*; esto consiste en que el progreso solo ocurre (existe) si y solo si el sistema sobre el cual funciona el algoritmo concurrente, provee ciertas garantías. Por ejemplo, en un sistema operativo estas propiedades son útiles cuando el sistema garantiza que eventualmente cada proceso deja la sección crítica de *manera oportuna*.

La forma usual en la que se determina si un algoritmo concurrente es linealizable, es identificando el punto de linearización donde cada método del algoritmo tiene efecto. En los algoritmos bloqueantes, las secciones críticas suelen ser los puntos de linearización.

2.2.2.2. No bloqueantes

En las condiciones de progreso no bloqueantes, el retraso inesperado de un proceso no retrasa a los demás procesos. Por ejemplo, en una implementación concurrente no bloqueante de una cola, un proceso p ejecuta el método $enq(y)$ y se detiene, después un proceso q ejecuta el método $deq()$ y devuelve *null*; esto implica que el retraso inesperado del proceso p no afecta la ejecución del proceso q, tal como se muestra en la figura 2.6 en donde la llamada de p está pendiente.

Dos ejemplos de condiciones bloqueantes son *libre de esperas* (*wait-free*) y *libre de candados* (*lock-free*).

Un método es libre de esperas si garantiza que cada llamada a un método termina de ejecutarse en un número finito de pasos; decimos que un algorit-

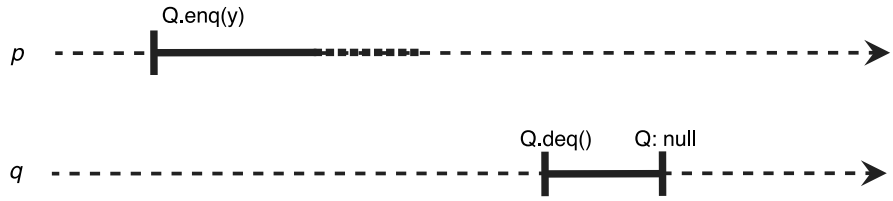


Figura 2.6: Ejemplo de una ejecución de una implementación concurrente no bloqueante de una cola: $\langle Q.enq(y)p\ \rangle\ \rightarrow \langle Q.deq()\ q\rangle\ \rightarrow \langle Q:\ null\ q\rangle$

mo es libre de esperas si todos sus métodos lo son. En otras palabras, esta propiedad garantiza que cada proceso, siempre que quiera, eventualmente progresa.

Un método es libre de candados si garantiza que infinitamente con frecuencia alguna llamada a un método termina de ejecutarse en un número finito de pasos.

Cualquier método libre de esperas es también libre de candados pero no al revés. Los métodos libre de esperas pueden llegar a ser ineficientes, es por ello que una propiedad menos fuerte como libre de candados es muy útil.

La linealizabilidad es una propiedad no bloqueante (Teorema 3) porque una invocación pendiente de un método nunca necesita esperar a que se complete otra invocación pendiente.

Teorema 3. *Sea $inv(m)$ una invocación de un método total. Si $\langle x\ inv\ \boldsymbol{P}\ \rangle$ es una invocación pendiente en una historia linealizable H, entonces existe un respuesta $\langle x\ res\ \boldsymbol{P}\ \rangle$ tal que $H \cdot \langle x\ res\ \boldsymbol{P}\ \rangle$ es linealizable.*

A diferencia de los algoritmos bloqueantes, identificar el punto de linearización en los algoritmos no bloqueantes es típicamente un paso (por ejemplo, puede ser una línea de código), donde los efectos de las llamadas se vuelven visibles para los demás procesos con sus respectivas operaciones.

2.3. Modelos de memoria compartida

En la teoría de computabilidad secuencial existen diferentes modelos de cómputo; un ejemplo de ellos son autómatas de estados finitos y las máquinas de Turing. En el cómputo concurrente también existen diferentes modelos;

por ejemplo, el modelo de cómputo de memoria compartida más simple se construye únicamente con registros de lectura/escritura.

2.3.1. Registros

Un registro de lectura/escritura, o simplemente registro, es un objeto que encapsula un valor que puede ser observado por un método de lectura $read()$ y modificado por un método de escritura $write()$.

Existen diferentes implementaciones de registros y cada una tiene diferentes características, podemos analizarlas en tres dimensiones:

- Por el tipo de dato que pueden encapsular. El cual puede ser un valor booleano, entero, una referencia a un objeto, o cualquier otro valor que pueda ser codificado. Además puede ser multivaluado o binario.
- Por la condición de corrección: pueden ser seguros, regulares o atómicos. Cualquier implementación de un registro con cualquiera de estas condiciones define un orden total en las llamadas al método de escritura. Sin embargo, para registros seguros y regulares el orden de escritura es trivial porque permiten que un solo proceso a la vez realice llamadas al método de escritura. Para los registros atómicos, las llamadas en general tienen un orden de linearización.
- Por la cantidad de procesos que pueden leer y escribir un registro. Por ejemplo, se utiliza el término $SRSW$ para referirse a un registro que un solo proceso puede leer y escribir. El término $MRSW$ se refiere a un registro que un solo proceso puede escribir pero que muchos procesos pueden leer. El término $MRMW$ se refiere a un registro que muchos procesos pueden leer y escribir.

Un ejemplo de una implementación con registros atómicos es un *snapshot atómico*.

2.3.2. Snapshots

El snapshot atómico es un objeto que permite leer los valores de múltiples registros de forma instantánea, construyendo una vista instantánea de un arreglo de registros atómicos. Estos objetos pueden ser útiles para realizar respaldos o analizar puntos de control.

De forma más específica, se trata de un arreglo de registros atómicos MRSW, donde a cada proceso le corresponde un registro; por lo tanto, todos los procesos pueden leer todos los registros del arreglo y cada proceso solo puede escribir en el registro que le corresponde.

Consta de dos métodos principales. El método *scan*() construye una vista instantánea del arreglo de registros, mientras que el método *update*() escribe un valor v en el registro del proceso que realiza la llamada. La propiedad clave de este objeto secuencial es que el método *scan*() regresa una colección de valores que representan el estado del sistema, donde cada *scan*() corresponde a la última actualización, es decir, a la última llamada a *update*().

Se han desarrollado implementaciones de snapshots atómicos. Por ejemplo, en [2] presentan una implementación libre de esperas que requiere $O(n^2)$ operaciones de lectura y escritura en registros atómicos. Esta implementación consiste, en resumen, en que si ambos métodos *scan*() y *update*() se ejecutan al mismo tiempo entonces el método *update*() ayuda al método *scan*(). El método *update*() toma una foto del arreglo de registros antes de modificar su registro, de esta forma la ejecución del método *scan*() que falla repetidamente al tomar una foto estática del arreglo puede tomar la foto de uno de los procesos que ejecutaron el método *update*() de forma concurrente a su ejecución.

En [7] presentan una implementación libre de esperas en la cual cada operación (*scan*() y *update*()) requiere $O(n\ log\ n)$ operaciones en registros atómicos de lectura/escritura, en donde, relacionan un objeto snapshot con el problema de decisión de *lattice agreement*, la implementación que presentan permite que los procesos obtengan valores distribuidos de la cadena o red (*lattice*), a diferencia de la implementación en [2], en donde cada proceso que ejecuta cualquiera de los dos métodos almacena una foto, es decir, un arreglo de registros.

2.4. Operaciones primitivas de sincronización

Las *operaciones primitivas* son instrucciones de sincronización que permiten a los procesos leer, modificar y escribir un valor en un objeto en un solo paso de hardware atómico, es decir, indivisible.

Estas operaciones se pueden analizar como objetos que exportan métodos y que a su vez ellas mismas son instrucciones; se pueden denotar también como *primitivas de sincronización*.

Se identifican y clasifican dependiendo de qué tan poderosas son para resolver problemas de sincronización y qué tan eficientes son al hacerlo. De hecho, existe una jerarquía infinita de primitivas de sincronización. De tal forma que ninguna primitiva en un nivel más bajo puede ser utilizada para una implementación no bloqueante de cualquier primitiva en un nivel mayor.

2.4.1. Operaciones RMW

Muchas de las operaciones de sincronización pueden ser expresadas como operaciones *read-modify-write* (RMW), también llamadas *registros read-modify-write* si son vistas como objetos. De hecho, cualquier operación de sincronización puede ser transformada de forma trivial a un método RMW.

Sea un registro RMW que encapsula valores enteros y sea $\mathcal{F}$ un conjunto de funciones de enteros a enteros. Un registro es un RMW para el conjunto de funciones $\mathcal{F}$ si de forma atómica reemplaza el valor actual del registro v con $f(v)$ para algún $f \in \mathcal{F}$ y regresa el valor original v.

Un ejemplo trivial de un método RMW es un método de lectura *get*() donde la función es $f(v) = v$. Algunos otros ejemplos de métodos *RMW* son: *swap()*, *test&set()*, *fetch&add()*, *fetch&inc()*, *compare&swap()*, entre otras.

Un método RMW es no trivial si tiene un conjunto de funciones que incluye al menos una función que no es una función identidad. A esta clase de registros RMW se le denota como *Common*2 RMW operaciones. Un ejemplo de esta clase es *swap()* porque sobreescribe el registro y devuelve el valor anterior ($f(x) = v$).

2.4.2. Número de consenso

La idea básica de la jerarquía es que cada clase de primitivas en la jerarquía está asociada a un *número de consenso* el cual es el número máximo de procesos para el que una primitiva de sincronización puede resolver el problema del consenso.

Un objeto de consenso provee un solo método *decide*(), cada proceso llama al método con su valor v al menos una vez. El método debe regresar un valor cumpliendo la condición de *consistencia*: todos los procesos deben decidir el mismo valor y la condición de *validez*: el valor común decidido es la entrada de un algún proceso.

A cualquier clase que implemente un objeto de consenso libre de esperas se le denomina como *protocolo de consenso*. Se han analizado diferentes implementaciones utilizando varios objetos de sincronización:

Teorema 4. *Los registros atómicos tienen un número de consenso 1.*

Teorema 5. *Las colas FIFO tienen un número de consenso de* 2*.*

Teorema 6. *Cualquier registro RMW en Common2 tiene un número de consenso de* 2*.*

Teorema 7. *Cualquier registro que provee métodos compare&swap() y get() tiene un número de consenso infinito.*

En la tabla 2.1 se muestra la jerarquía de las primitivas de sincronización según su número de consenso.

Número de consenso	Primitivas
1	registros, snapshots, ...
2	colas, pilas, *fetch&add()*, *fetch&inc()*, ...
⋮	⋮
∞	*compare&swap()*, *get()*, ...

Cuadro 2.1: Jerarquía de las primitivas de sincronización

Capítulo 3

Estado del arte de la verificación de la linealizabilidad

Verificar que un algoritmo concurrente cumple con determinadas condiciones de corrección y de progreso es importante debido a que estas condiciones describen al algoritmo, revelando su capacidad para tolerar fallas, para ser escalable, para cumplir con cierta disponibilidad y, en general, para resolver el problema para el cual fue desarrollado.

Este trabajo se centra en la verificación de la condición de corrección de la linealizabilidad. En este capítulo se presenta el estado de arte de la verificación de la linealizabildad a través de cuatro técnicas fundamentales de verificación: *prueba de teoremas*, *verificación de modelos*, *testeo* y *verificación en tiempo de ejecución* [31]. Además se describe con mayor detalle la técnica de verificación en tiempo de ejecución: su estructura general y algunas dificultades en su desarrollo e implementación. Por último se comparan las técnicas de verificación de la linealizabilidad con la técnica de verificación en tiempo de ejecución.

3.1. Verificación de la linealizabilidad

Debido la creciente demanda del alto rendimiento, de la escalabilidad y de la disponibilidad en los sistemas modernos de cómputo, se presenta un aumento en la sofisticación de implementaciones concurrentes linealizables.

Dados estos requerimientos los algoritmos deben probar ser correctos, es decir, realmente linealizables.

Existen métodos para crear algoritmos concurrentes libres de esperas y linealizables. En [23] se muestra un método universal, también llamado *construcción universal*, para transformar cualquier objeto secuencial a un objeto concurrente libre de esperas y linealizable. En esta construcción se utilizan primitivas con número de consenso infinito como *compare&swap*. Si todos los algoritmos concurrentes se diseñaran con este método, se podrían construir por defecto algoritmos concurrentes linealizables y libres de esperas; sin embargo, la implementación que se obtiene no es eficiente y la eficiencia es un requisito indispensable para muchos problemas en la actualidad.

Existe una gran variedad de métodos y técnicas de verificación de la linealizabilidad; sin embargo, la técnica de verificación que se desarrolló en un principio fue la de prueba de teoremas.

3.2. Verificación de la linealizabilidad en prueba de teoremas

Esta técnica consiste en mostrar la linealizabilidad de un algoritmo de forma manual; es decir, con papel y lápiz, de forma similar a probar la correctez de un teorema en las matemáticas.

Se han desarrollado una gran cantidad de métodos que utilizan nociones como *refinamiento*, *análisis de formas*, *reducción*, entre otras. La mayoría de los métodos utilizan la noción de refinamiento para relacionar de alguna forma objetos concretos y abstractos. Por ejemplo, varios métodos consisten en caracterizar los puntos de linearización del algoritmo a verificar [16].

Los métodos actuales continúan luchando con la sofisticación de los algoritmos concurrentes; como resultado, solo se ha verificado formalmente la linealizabilidad de unos cuantos algoritmos. En [16] presentan una relación de cuáles algoritmos han sido verificados con qué métodos.

Algunas desventajas de esta técnica es que cualquier método no puede verificar cualquier algoritmo. Esto sumado a que, debido a la complejidad de los algoritmos, se han descubierto errores en algoritmos que supuestamente eran linealizables, como el algoritmo *Snark* [15].

Algunos de los algoritmos presentados en [16] tienen herramientas automáticas para reducir el potencial error humano. Estas herramientas tienen

puntos de linearización fijos por lo que muy pocos algoritmos pueden ser verificados con estas herramientas.

3.3. Verificación de la linealizabilidad en verificación de modelos

La verificación de modelos es una técnica de verificación automática que se aplica más comúnmente en sistemas de estados finitos. En estos sistemas todos los cómputos pueden ser enumerados de forma exhaustiva. Este problema consiste básicamente en describir el problema de determinar si dados un modelo $\mathcal{M}$ y una especificación de corrección φ, en todos los cómputos $\mathcal{M}$ satisface φ, siendo φ la linealizabilidad [31].

Cuando se aplica en sistemas de estados finitos, desde un enfoque teórico de autómatas, la propiedad de la linealizabildad φ puede ser transformada a un autómata $\mathcal{M}_{\neg\varphi}$ que acepta todas las ejecuciones que incumplen con φ [31].

El problema de la verificación de la linealizabildad de forma automática es indecidible[1] para un número no acotado de procesos [11].

Verificar todas las ejecuciones de una implementación con un número acotado de procesos es **EXPSPACE**-*completo*[2] [3].

En [1] muestran que verificar la linealizablidad de siquiera una historia de una ejecución para un número no acotado de procesos es **NP**-*completo*[3]. Además presentan un algoritmo exponencial con respecto al número de procesos, de tal forma que el problema se resuelve de forma eficiente para un número muy reducido de procesos.

Un ejemplo concreto de esta técnica en donde resuelven el problema en tiempo y espacio exponencial para un número acotado de procesos, se pre-

[1] Un problema es indecidible si no existen algoritmos que lo solucionen [37].

[2] La clase **EXPSPACE** consiste de todos los problemas de decisión que pueden ser resueltos con un algoritmo de espacio $O(2^{p(n)})$, donde $p(n)$ es una función polinomial sobre n. Un problema B es **EXPSPACE**-*completo* si $B \in$ **EXPSPACE** y todo problema $A \in$ **EXPSPACE** es reducible en tiempo polinomial a B. [11]

[3] La clase **NP** consiste de todos los lenguajes que pueden ser verificados con un algoritmo en tiempo polinomial, los problemas **NP**-*completos* son los problemas más difíciles en **NP**, lo cual significa que existe un algoritmo en tiempo polinomial que resuelve un problema **NP**-*completo* si y solo si **P** = **NP** [4]

senta en [11]. Modelan el sistema distribuido como una gráfica. Esta gráfica se detiene y decide únicamente cuando el algoritmo es linealizable; pero si el algoritmo no es linealizable, la gráfica se queda ciclada y nunca se detiene.

Otro ejemplo se desarrolla en [3], donde reducen el problema de la verificación de la linealizabilidad con un número acotado de procesos al problema de determinar si un lenguaje es un lenguaje regular. Este problema de pertenencia de un lenguaje se soluciona con un algoritmo de tiempo y espacio exponencial que consiste en comprobar si la especificación del objeto que implementa el algoritmo a verificar es un lenguaje regular; en otras palabras, comprueba si el objeto concreto que implementa el objeto abstracto es un lenguaje regular.

Un ejemplo en donde plantean una solución eficiente[4] se presenta en [12]. Esta consiste en modelar un conjunto de reglas que conservan la linealizabilidad de determinadas implementaciones de objetos, de tal forma que se construyen autómatas finitos deterministas que aceptan si y solo si se incumplen las reglas; es decir, aceptan si y solo si las implementaciones no son linealizables. Sin embargo, las únicas implementaciones de objetos que se pueden verificar con este método son las colas, las pilas y los registros.

El veredicto de la verificación de las técnicas de prueba de teoremas y de verificación de modelos, se suele referir a un modelo o abstracción del sistema real que se está verificando, porque aplicar estas técnicas directamente a la implementación real sería imposible o intratable. Un modelo de un sistema refleja los aspectos más relevantes del sistema; sin embargo, la implementación del sistema puede comportarse un poco diferente a lo que se espera del modelo. Por ello la importancia de las técnicas de testeo y de verificación en tiempo de ejecución que permiten verificar la implementación real del sistema.

3.4. Verificación de la linealizabilidad en testeo

La técnica de testeo consiste, a grandes rasgos, en considerar un conjunto finito de secuencias de entrada/salida para formar un *banco de pruebas* y

[4]Un problema se puede resolver de forma eficiente si puede ser resuelto por una Máquina de Turing en tiempo polinomial [4].

posteriormente comprobar si la salida del sistema concuerda con la salida esperada [31].

Esta técnica, a diferencia de las dos previamente presentadas, no considera todos las posibles ejecuciones del sistema sino únicamente un conjunto de finito de ellas; de hecho comparte esta cualidad con la técnica de verificación en tiempo de ejecución.

Un modelo de testeo tiene como objetivo comprobar la linealizabilidad de una implementación concurrente de un objeto con respecto a un objeto secuencial dado a partir de un mecanismo de detección, posteriormente se pueden aplicar algoritmos de testeo genéricos para descubrir errores predefinidos [9].

Por ejemplo, en [9] se presenta un modelo para crear un programa de testeo adaptable a implementaciones de diferentes tipos de objetos. El mecanismo de detección que se plantea consiste en que cada proceso escribe su propio archivo de detección, en donde se acompaña cada operación que realiza con una marca de tiempo. Además se plantean algoritmos de testeo genéricos, los cuales pueden complementar a otros algoritmos que verifiquen la linealizabilidad con esta técnica.

3.5. Verificación de la linealizabilidad en tiempo de ejecución

Varios conceptos de esta sección fueron tomados del artículo de Bonakdarpour, Fraigniaud, Rajsbaum y Travers: *Challenges in fault-tolerant distributed runtime verification* en [10].

La técnica de verificación en tiempo de ejecución es una disciplina que consiste en el estudio, desarrollo y aplicación de aquellas técnicas de verificación que permiten verificar en tiempo de ejecución un sistema o algoritmo que satisface o no la linealizabilidad [31].

Se utiliza para verificar una implementación que ya ha sido desplegada, para determinar si una ejecución en tiempo de ejecución se encuentra en un estado válido o no con respecto a la linealizabilidad.

3.5.1. Estructura general

La estructura general de esta técnica de verificación se compone esencialmente de dos sistemas: el sistema a verificar y el sistema verificador.

El objetivo del sistema verificador es comprobar en tiempo de ejecución si el sistema a verificar cumple con la especificación de corrección de la linealizabilidad.

El tiempo de ejecución se entiende como una posible secuencia de estados, conjunto de ejecuciones, del sistema a verificar. El sistema verificador comprueba si la ejecución actual del sistema a verificar en tiempo de ejecución, cumple con la linealizabilidad. Para lograr su objetivo el sistema verificador suele utilizar un monitor o un conjunto de monitores.

De manera formal,

Definición 2. *Un monitor es un dispositivo que lee una ejecución finita y produce un determinado veredicto sobre esa ejecución [31].*

Un veredicto suele ser un valor de verdad sobre algún dominio. Este puede ser binario, verdadero/falso o adaptarse a un tipo de probabilidad con la que se satisface la linealizabildad [31].

Un monitor puede ser una máquina, un programa, un proceso o una mezcla.

La relación entre los dos sistemas se establece mediante una *interface de comunicación*. Esta comunicación suele consistir en que el sistema verificador detecta de forma continua el estado del sistema a verificar sin afectar el desarrollo del sistema a verificar.

El sistema verificador, además de comprobar la linealizabilidad del sistema a verificar, puede realizar un diagnóstico de los errores y ejecutar métodos para resolverlos.

3.5.2. Sistema verificador distribuido

Esta técnica de verificación se puede aplicar de forma centralizada o distribuida sobre algún sistema a verificar.

De forma centralizada se tiene el escenario más sencillo: el sistema a verificar es distribuido y el sistema verificador es centralizado.

El sistema verificador puede consistir de un único monitor que detecta y verifica las ejecuciones que obtiene del sistema a verificar, de tal forma que por cada ejecución emite un veredicto. La desventaja de este modelo es que no es tolerante a fallas ya que si este único monitor se detiene o si no logra capturar la concurrencia del sistema a verificar, el sistema verificador no logra su objetivo.

De forma distribuida ambos sistemas son distribuidos. El sistema verificador distribuido puede consistir en desplegar sobre el sistema a verificar un conjunto de monitores que se comunican entre sí para realizar la detección de la ejecución; es decir, construir de alguna forma la historia de la ejecución para que cada monitor pueda verificarla de forma local.

3.5.3. Desafíos y dificultades de un sistema verificador distribuido

Dos grandes dificultades en el desarrollo de un sistema verificador distribuido son que la estructura general está desacoplada y que los monitores deben obtener la misma ejecución actual del sistema; es decir, deben resolver el problema del consenso para obtener el mismo estado global del sistema a verificar.

3.5.4. Estructura general desacoplada

Que la estructura general esté desacoplada quiere decir que los dos sistemas están desacoplados. El sistema a verificar es diseñado y desplegado sin considerar el sistema verificador que se construye después.

Con la idea de tratar este problema en [17] introducen un lenguaje especifico de dominio denotado como *Psync*, el cual es capaz de unificar modelar, programar y verificar algoritmos distribuidos tolerantes a fallas. La ventaja es que los programas que se diseñan y se despliegan en este lenguaje tienen características que les permiten verificarse; sin embargo, en la práctica, la verificación la realizan con un motor de ejecución sobre la ejecución del algoritmo. Esto implica que los sistemas permanecen desacoplados.

3.5.5. Sistemas verificadores distribuidos que resuelven el consenso

El concepto del estado global de un sistema en ejecución se presenta en el algoritmo de Lamport y Chandy en [14]. Este tiene como objetivo generar fotos globales de un sistema distribuido.

Detectar el estado global de un sistema que se está ejecutando ayuda a resolver problemas como la verificación en tiempo de ejecución. Así, dada

la misma foto global, cada monitor puede comprobar si el estado actual del sistema cumple con la linealizabilidad.

La forma más intuitiva para obtener el estado global del sistema es resolviendo el problema del consenso.

En un sistema distribuido cada monitor conoce solo una parte de este estado global, de forma que para obtener el estado global completo, los monitores asíncronos se comunican entre ellos utilizando ciertas herramientas de sincronización y tolerando diferentes tipos de fallas con el objetivo de resolver el consenso sobre el estado global. Por ejemplo, se pueden comunicar en un modelo de memoria compartida o en un modelo de paso de mensajes.

Sin embargo, el problema del consenso no se puede solucionar en sistemas asíncronos con una sola falla en un modelo de paso de mensajes [19]. Esta imposibilidad también se extiende a modelos de memoria compartida [9]. Además existen cotas inferiores en el número de rondas de comunicación necesarias para resolver el consenso como se muestra en [9].

3.5.5.1. Ejemplos de sistemas verificadores que resuelven el consenso

En [17, 33, 35] se resuelve el consenso del estado global del sistema.

Para lograrlo en [35] despliegan m monitores que se comunican en un modelo de paso de mensajes para construir cada uno un vector de conocimiento, el cual se actualiza con la comunicación y representa una aproximación al estado global del sistema. En [33] construyen un conjunto de m monitores que se comunican en un modelo de paso de mensajes y detectan estados globales posibles utilizando una adaptación de la técnica de *lattice-theoretic*, de forma que cada monitor tiene un conjunto de veredictos. Para elegir un solo veredicto construyen un oráculo que dicta el orden total de los eventos. A este orden total le corresponde un veredicto del conjunto de veredictos de cada monitor.

En [17] despliegan monitores en un modelo de memoria compartida. Realizan suposiciones sobre la ejecución y sobre el tipo de fallas que pueden suceder utilizando un reloj global para aproximarse al estado global del sistema a verificar.

En general estos tres ejemplos resuelven el consenso utilizando herramientas de sincronización que permiten que los monitores obtengan el mismo

estado actual del sistema a verificar.

3.5.6. Sistemas verificadores distribuidos que no resuelven el consenso

Se han estudiado sistemas verificadores distribuidos en donde los monitores en el sistema verificador no son capaces de resolver el consenso.

Un modelo para este tipo de sistemas se presenta en [10]. Este consiste en que dada una ejecución $\alpha = s_0 s_1 \cdots s_k$ de k estados globales, un conjunto de monitores $M = \{M_1, M_2, \ldots, M_n\}$ inspecciona la condición de corrección φ de α. Los monitores se comunican entre ellos por medio de registros de lectura/escritura de memoria compartida. Ejecutan el algoritmo siguiente:

Por cada $j \in [0, k-1]$, entre s_j y s_{j+1}, cada monitor M_i:

1. toma una muestra que resulta en una observación parcial de s_j, denotada como $S_i(s_j)$;
2. repetidas veces comunica su muestra con otros monitores a través de la memoria compartida, y
3. emite un veredicto sobre la corrección de α.

Si las muestras $S_i(s_j)$ logran construir de forma colectiva toda la información de s_j, entonces cada monitor puede construir s_j y evaluar φ de forma local.

En el paso 2 los monitores se comunican tanto como quieran, pero en un número finito de veces utilizan N operaciones de escritura y una de lectura. Este modelo corresponde a un modelo de memoria compartida de lectura/escritura asíncrono por capas y libre de esperas, en un modelo así los monitores no pueden llegar a un consenso debido a la imposibilidad de resolver el consenso en [23].

3.5.6.1. Ejemplo de un sistema verificador que no resuelve el consenso

Este modelo lo utilizan en [20]. La diferencia es que añaden un punto 4 al modelo, este permite evaluar una vista parcial o incompleta de un estado del sistema s_j; es decir, el monitor M_i es capaz de emitir un veredicto basado

en su sola vista parcial sobre s_j o sobre un conjunto de vistas parciales. Para ello se utiliza una fórmula o función de extensión que permite completar la vista parcial al estado completo s_j que pudo haber ocurrido en el sistema. Las vistas parciales se pueden completar pero no se puede llegar al consenso si al menos una vista parcial falta.

3.5.7. Veredictos

Los monitores en un sistema verificador pueden emitir un conjunto de veredictos; si el sistema verificador resuelve el consenso, todos los monitores eligen de ese conjunto el mismo veredicto. Sin embargo, si el sistema verificador no resuelve el consenso, cada monitor puede producir un veredicto diferente.

Los monitores pueden evaluar un conjunto de veredictos con lógicas temporales lineales (LTL_K) porque permiten especificar propiedades de ejecuciones infinitas.

En sistemas verificadores que resuelven el consenso los monitores evalúan sus veredictos para emitir el mismo veredicto. Por ejemplo, en [35] los monitores utilizan una lógica temporal distribuida (DLT), la cual es capaz de modelar el tiempo pasado. En [33] utilizan una semántica de tres opiniones sobre la lógica temporal lineal (LTL_3), de tal forma que cada monitor tiene un autómata para inspeccionar cada propiedad de LTL_3.

En [20] se desarrolla un ejemplo de cuando los monitores no resuelven el problema del consenso. En este caso se propone una lógica temporal multivaluada. De forma concreta, se proponen $2k + 4$ valores lógicos para $k \geq 0$, denotada como LTL_{2k+4}, que intuitivamente representa un *grado de certeza* de que la formula se satisface.

En [19] prueban una cota inferior en el número de veredictos por monitor requeridos para distinguir las ejecuciones correctas de las incorrectas. Además argumentan que entre más "compleja" sea la especificación a verificar, más veredictos diferentes se deben utilizar.

3.6. Relación de las técnicas de verificación con la técnica de verificación en tiempo de ejecución

La técnica de verificación de modelos utiliza varias nociones de la técnica de prueba de teoremas pero de forma automática, esto es conveniente debido a la complejidad y a la sofisticación de los algoritmos a verificar.

La técnica de verificación en tiempo de ejecución tiene su origen en la técnica de verificación de modelos porque el problema clave de generar monitores es similar a la generación de autómatas en la verificación de modelos. Sin embargo, los monitores solo comprueban ejecuciones que son generadas por el sistema real en ejecución, mientras que en la verificación de modelos se verifican todas las posibles ejecuciones sobre un modelo que representa el sistema real.

La técnica de verificación en tiempo de ejecución y la técnica de testeo tienen similitudes debido a que ambas comprueban ejecuciones reales del sistema; de hecho, la verificación en tiempo de ejecución puede entenderse como una forma de "testear por siempre".

El comportamiento de un sistema en el que se ejecuta un algoritmo no siempre puede ser descrito de forma precisa por lo que puede no existir información suficiente para testear el sistema de forma adecuada y las pruebas formales de corrección de la verificación de modelos y de prueba de teoremas solo funcionan bajo suposiciones del comportamiento del sistema en el que se desarrolla el algoritmo.

En este escenario la verificación en tiempo de ejecución supera el testeo clásico, y en el modelado de los monitores, se pueden añadir pruebas formales de corrección de la verificación de modelos y de la prueba de teoremas.

En general la importancia de cada técnica radica en su capacidad de verificar a los algoritmos desde diferentes perspectivas.

Se puede realizar un análisis teórico utilizando las técnicas de verificación de modelos y de prueba de teoremas. También, se puede comprobar la implementación de los algoritmos en la práctica, lo cual permite detectar y corregir errores del sistema real utilizando las técnicas de testeo y de verifi-

cación en tiempo de ejecución. Por esta razón las cuatro técnicas se pueden complementar para verificar la linealizabilidad de un algoritmo concurrente.

Capítulo 4

Verificación de la linealizabililidad en tiempo de ejecución

El problema de la verificación en tiempo de ejecución denotado como "VLTE" es imposible de resolver incluso si se utilizan primitivas de sincronización con número de consenso infinito (teorema 8).

En este capítulo se define formalmente el problema de la VLTE para después presentar una demostración de imposibilidad al problema de la VLTE.

4.1. Formalización del problema de la VLTE

Este problema consiste en monitorear y comprobar de forma distribuida y en tiempo de ejecución la linealizabilidad de cualquier algoritmo que se especule es linealizable. La verificación es local porque cada proceso ejecuta el mecanismo que detecta la historia de la ejecución y decide si la historia es linealizable.

Sea A un algoritmo concurrente que se presume es linealizable con respecto a un objeto O.

El modelo de este problema consiste en que un conjunto de procesos se comunica sobre una memoria compartida para detectar la ejecución actual del algoritmo A y emitir un veredicto sobre si es linealizable o no con respecto al objeto O. A todo el trabajo que los procesos desempeñan desde comunicarse entre ellos hasta emitir el veredicto le denotaremos como sistema o algoritmo verificador V.

El algoritmo A se representa como una caja negra, lo que quiere decir que su contenido no se puede inspeccionar, alterar o controlar las ejecuciones: sus invocaciones y/o respuestas.

Como el algoritmo A y el sistema verificador V están desacoplados, su diseño e implementación se realiza de forma independiente.

Un sistema verificador V se puede analizar en dos sistemas:

1. Sistema de detección.

   Es un algoritmo concurrente cuyo objetivo es que los procesos involucrados en la ejecución del algoritmo A colaboren para obtener una descripción abstracta de la ejecución actual del algoritmo A que se desarrolle en tiempo de ejecución.

   Cada vez que un proceso p se involucre en la ejecución del algoritmo A, debe comunicar de alguna forma en el sistema verificador V, en el número de operaciones que necesite la parte de la ejecución que él realiza sobre el algoritmo A. Así, por cada operación que un proceso realice en el algoritmo A, realiza un conjunto de N operaciones sobre el sistema verificador V. Debido a que cada proceso está bien formado solo pueden ejecutar las operaciones del sistema verificador V antes o después o antes y después del punto de linearización de cada operación del algoritmo A, tal como se muestra en la siguiente expresión 4.1 en donde la primera o segunda ejecución del sistema de detección pueden ser vacías.

   $$\rightarrow \text{Sistema de detección}(\ldots)\ p \rightarrow A(\ldots)\ p \rightarrow \text{Sistema de detección}(\ldots)\ p \tag{4.1}$$

   En la expresión 4.1 se muestra cómo el sistema verificador V envuelve de alguna forma a la ejecución del algoritmo A, debido a esto el sistema verificador puede bloquear o disminuir el progreso de los procesos del algoritmo A. Es por ello que solo nos interesa el caso en el que V es libre de esperas, ya que si V es libre de candados, puede ser que determinados procesos no puedan compartir sus ejecuciones de A en V.

2. Sistema de verificación de la linealizabilidad.

   Es un algoritmo local cuyo objetivo es que cada proceso pueda, en cualquier momento, obtener la historia abstracta que se construye en

el sistema de detección y resolver el problema de la verificación de la linealizabilidad. Este sistema de verificación de la linealizabilidad es una caja negra porque existen varios algoritmos que pueden resolver este problema [16]. La salida que produce este sistema es un veredicto $falso$ si la historia abstracta en V, no es linealizable con respecto a A y *verdadero* si la historia abstracta en V, es linealizable con respecto a A junto con su linearización.

El algoritmo V soluciona el problema de la VLTE si se cumple lo siguiente cada vez que un proceso utilice el sistema de verificación de la linealizabilidad:

1. Precisión: si la historia en tiempo de ejecución del algoritmo A es linealizable con respecto al objeto O, entonces el algoritmo V siempre computará *verdadero*.

2. Completez: Si el algoritmo V computa *verdadero*, entonces la historia en tiempo de ejecución del algoritmo A es linealizable con respecto al objeto O.

Estas dos propiedades implican que si A es linealizable con respecto a O, entonces V mantiene o preserva la linealizabilidad en la historia abstracta que detecta de A con respecto a O.

Si el sistema verificador V cumple con las dos propiedades, entonces puede detectar las ejecuciones en tiempo real del algortimo A y puede obtener un certificado que determine que el algoritmo A no es linealizable, a este certificado lo denotaremos como *historia causante*. Una historia causante es la primera historia de una ejecución del algoritmo A que el sistema verificador V detectará como no linealizable.

Si el sistema verificador V detecta una historia causante, entonces el algoritmo A es *permanentemente no linealizable*.

Si el sistema verificador V no ha detectado una historia causante, entonces el algoritmo A es *linealizable hasta el momento*.

Estas dos son propiedades que pueden caracterizar al algoritmo A porque la linealizabilidad es composicional (teoremas 1 y 2). Si A es permanentemente linealizable, no es posible, aunque todas las demás ejecuciones sean linealizables, que el algoritmo lo sea. En cambio, si es linealizable hasta el momento, existe la posibilidad de que el algoritmo A no lo sea pero hasta el momento no se ha encontrado una historia causante.

4.2. Prueba de imposibilidad

A continuación se presenta una demostración que exhibe que no es posible que un sistema verificador V verifique en tiempo de ejecución la linealizabilidad de cualquier algoritmo A que se presuma es linealizable incluso si V utiliza operaciones con número de consenso infinito como *compare&swap*.

Primero se presenta un caso ideal donde no se considera la asincronía de los procesos, después se presenta un caso general donde se considera la asincronía de los procesos y el algoritmo V no puede distinguir lo que sucede en A.

4.2.1. Caso ideal

Suponemos que existe un algoritmo V que soluciona la VLTE de un algoritmo A para cualquier número de procesos. En este ejemplo nos enfocaremos solo en dos procesos: p y q.

Suponemos que ambos algoritmos son libre de esperas. Suponemos que A es un algoritmo que se presume es linealizable e implementa una cola.

Los procesos p y q utilizan el algoritmo V de acuerdo a la estructura de la expresión 4.1. Le denotaremos como V_d al sistema de detección del algoritmo V y como V_{VL} al sistema de verificación de la linealizabildad del algoritmo V. El sistema V_d codifica de alguna forma la invocación y la respuesta del algoritmo A en la memoria compartida, de tal forma que se construyen las relaciones de precedencia en ella. Para lograrlo, V_d puede utilizar métodos de escritura/lectura, así como operaciones *Read-Modify-Write* como *compare&swap*, *swap()*, *get()*, entre otras.

Suponemos que se desarrollan dos ejecuciones:

- Ejecución 1.

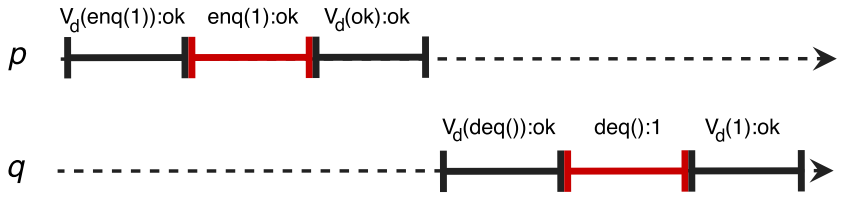


Figura 4.1: Historia de la ejecución 1

La historia de la ejecución del algoritmo A se muestra en rojo en la figura 4.1, es: $\rightarrow \langle enq(1) : ok\ p\rangle \rightarrow \langle\ deq() : 1\ q\rangle$, podemos determinar con facilidad que es una historia linealizable con respecto a una cola.

La historia de la ejecución del algoritmo V se muestra en la figura 4.1 en color negro, es: $\rightarrow \langle V_d(enq(1)) : ok\ p\rangle \rightarrow \langle V_d(ok) : ok\ p\rangle \rightarrow \langle V_d(deq(1)) : ok\ q\rangle \rightarrow \langle V_d(1) : ok\ q\rangle$.

Cada proceso a partir de la historia detectada invoca de forma local el método V_{VL}.

Supongamos que el proceso p invoca el método V_{VL} justo después de realizar su operación en V_d, entonces obtiene la historia $\rightarrow \langle V_d(enq(1)) : ok\ p\rangle \rightarrow \langle V_d(ok) : ok\ p\rangle$ y computa *verdadero* porque la historia por el algoritmo V es linealizable con respecto al algoritmo A.

Supongamos que el proceso q invoca el método V_{VL} justo después de realizar su operación en V_d, entonces obtiene la historia $\rightarrow \langle V_d(enq(1)) : ok\ p\rangle \rightarrow \langle V_d(ok) : ok\ p\rangle \rightarrow \langle V_d(deq(1)) : ok\ q\rangle \rightarrow \langle V_d(1) : ok\ q\rangle$ y computa *verdadero* porque la historia por el algoritmo V es linealizable con respecto al algoritmo A.

- Ejecución 2.

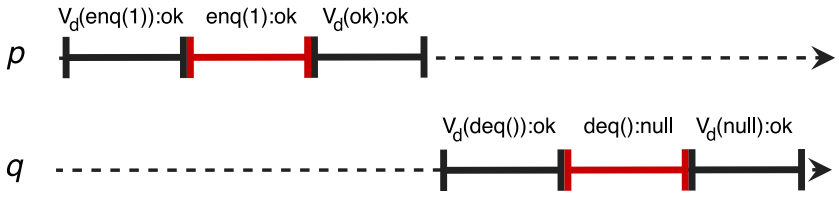


Figura 4.2: Historia de la ejecución 2

La historia de la ejecución del algoritmo A se muestra en rojo en la figura 4.2, es: $\rightarrow \langle enq(1) : ok\ p\rangle \rightarrow \langle deq() : null\ q\rangle$, podemos determinar con facilidad que esta historia no es linealizable con respecto a una cola.

La historia de la ejecución del algoritmo B se muestra en la figura 4.2 en color negro, es: $\rightarrow \langle V_d(enq(1)) : ok\ p\rangle \rightarrow \langle V_d(ok) : ok\ p\rangle \rightarrow \langle V_d(deq()) : ok\ q\rangle \rightarrow \langle V_d(null) : ok\ q\rangle$.

Cada proceso a partir de la historia detectada invoca de forma local el método V_{VL}.

Supongamos que el proceso p invoca el método V_{VL} justo después de realizar su operación en V_d, entonces obtiene la historia $\rightarrow \langle V_d(enq(1)) : ok\ p\rangle \rightarrow \langle V_d(ok) : ok\ p\rangle$ y computa *verdadero* porque la historia por el algoritmo V es linealizable con respecto al algoritmo A.

Supongamos que el proceso q invoca el método V_{VL} justo después de realizar su operación en V_d, entonces obtiene la historia $\rightarrow \langle V_d(enq(1)) : ok\ p\rangle \rightarrow \langle V_d(ok) : ok\ p\rangle \rightarrow \langle V_d(deq()) : ok\ q\rangle \rightarrow \langle V_d(null) : ok\ q\rangle$ y computa $falso$ porque la historia por el algoritmo V no es linealizable con respecto al algoritmo A.

En estos dos casos concretos el algoritmo V cumple con la propiedad de precisión porque siempre que la historia de la ejecución de A es linealizable con respecto a una cola, V computa *verdadero*. Además cumple con la propiedad de completez porque V computa *verdadero* cuando la historia de la ejecución de A es linealizable con respecto a una cola.

Por lo tanto, en este caso el algoritmo V computa correctamente el resultado del problema de la VLTE de A.

Sin embargo este es un caso ideal porque no se refleja la asincronía de los procesos.

4.2.2. Asincronía

Para lograr casos más apegados a lo que sucede en un sistema concurrente real consideraremos la asincronía.

La noción de asincronía puede generar que los procesos puedan detenerse o tardarse en ejecutar una operación del algoritmo A después de ejecutar una operación del algoritmo V y viceversa, a esta acción le denotaremos *retardo*. De forma que la longitud y la existencia de retardos es impredecible.

Por ejemplo, en la figura 4.3 el proceso p tiene dos retardos de longitud arbitraria e impredecible: r_i y r_j.

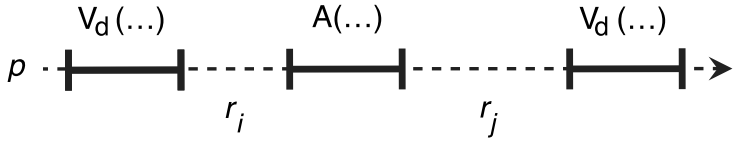


Figura 4.3: Estructura 4.1 con retardos

4.2.3. Caso general

Por contradicción suponemos que existe un algoritmo V que cumple con las dos propiedades del problema de la VLTE. El algoritmo V es libre de esperas y resuelve la VLTE de cualquier algoritmo concurrente A para cualquier número de procesos. Le denotaremos como V_d al sistema de detección del algoritmo V y como V_{VL} al sistema de verificación de la linealizabildad del algoritmo V.

Lema 1. *Suponemos que existe un algoritmo concurrente A no bloqueante con la siguiente propiedad: en toda ejecución en la que dos procesos p y q realicen las operaciones op_p: $\langle A.inv_p() : res_p\ p\rangle$ y op_q: $\langle A.inv_q() : res_q\ q\rangle$, respectivamente, las invocaciones de las operaciones y las respuestas de las operaciones son siempre las mismas.*

Entonces para todo algoritmo V que resuelve el problema de la VLTE de A con respecto a algún objeto secuencial O, existen tres ejecuciones en las que solo los dos procesos p y q participan y realizan op_p y op_q, respectivamente. A pesar de que en las tres ejecuciones las operaciones se ejecutan en ordenes distintos, los procesos no pueden distinguir el orden en el que sucedieron op_p y op_q. Por lo tanto, en las tres ejecuciones los dos procesos producen la misma respuesta en V_{VL}.

Las tres ejecuciones en donde aparecen op_p y op_q con órdenes distintos son α: sucede $op_p \to op_q$, β: sucede $op_q \to op_p$ y γ: sucede ni $op_p \to op_q$ ni $op_q \to op_p$, tal como se muestran en las figuras 4.4, 4.5 y 4.6, respectivamente.

La historia que detecta V_d es la misma en las tres ejecuciones:

$$\to \langle V_d(\ldots) : ok\ p\rangle \to \langle V_d(\ldots) : ok\ q\rangle \to \langle V_d(\ldots) : ok\ q\rangle \to \langle V_d(\ldots) : ok\ p\rangle.$$

Esto sucede debido a la existencia de retardos:

- En la ejecución α, figura 4.4, el proceso p tiene un retardo r_i y un retardo r_j antes y después, respectivamente, de ejecutar la operación

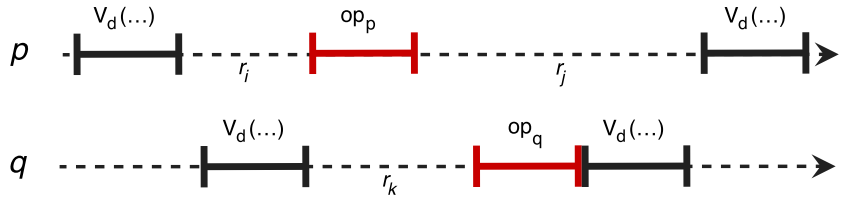


Figura 4.4: V detecta la ejecución α

op_p del algoritmo A . El proceso q tiene un retardo r_k antes de ejecutar la operación op_q del algoritmo A.

- En la ejecución β, figura 4.5, el proceso p tiene un retardo r_i y un retardo r_j antes y después, respectivamente, de ejecutar la operación op_p del algoritmo A. El proceso q un retardo r_k después de ejecutar la operación op_q del algoritmo A.

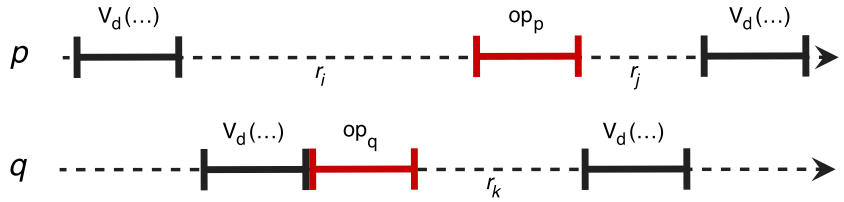


Figura 4.5: V detecta la ejecución β

- En la ejecución γ, figura 4.6, el proceso p tiene un retardo r_i y un retardo r_j antes y después, respectivamente, de ejecutar la operación op_p. De la misma forma el proceso q tiene un retardo r_k y un retardo r_l antes y después de ejecutar la operación op_q del algoritmo A.

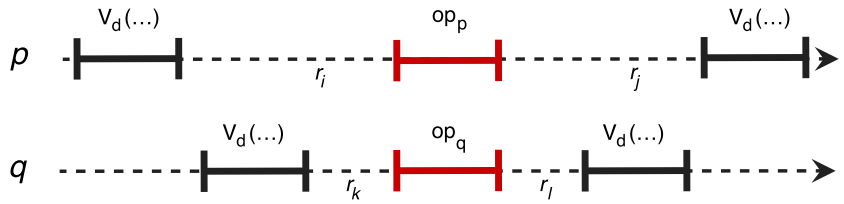


Figura 4.6: V detecta la ejecución γ

En las tres ejecuciones los procesos obtienen la misma historia al ejecutar localmente el método V_{VL}, por lo tanto cada proceso computa lo mismo en

V_{VL} independientemente del orden en el que sucedieron las operaciones op_p y op_q en A.

Corolario 1. *Consideremos la situación descrita en el lema 1 y supongamos que una de las historias secuenciales, $op_p \rightarrow op_q$ y $op_q \rightarrow op_p$, es válida y la otra no lo es para el objeto secuencial O, esto implica que A no es linealizable con respecto a O. Entonces V incumple alguna de las propiedades de precisión o completez de la definición de la VLTE.*

Si en las tres ejecuciones α, β y γ ambos procesos p y q computan $falso$ y al menos una de las ejecuciones es válida con respecto a O y al menos una no lo es, entonces alguna de las ejecuciones es linealizable con respecto a O y el algoritmo V no computa *verdadero*, entonces V no cumple la propiedad de precisión.

En cambio, si en las tres ejecuciones α, β y γ ambos procesos p y q computan *verdadero* y al menos una de las ejecuciones es válida con respecto a O y al menos una no lo es, entonces el algoritmo V computa *verdadero* cuando alguna de las ejecuciones no es linealizable con respecto a O, entonces V no cumple la propiedad de completez.

Teorema 8. *El problema de la VLTE es imposible incluso cuando en V se puedan utilizar operaciones con número de consenso infinito.*

Sea O un objeto secuencial de tipo cola y sea A_{cola} un algoritmo libre de esperas que implementa a O. El algoritmo A_{cola} implementa los métodos $enq(x)$ y $deq()$ como se muestra a continuación:

Algoritmo 2 Algoritmo A_{queue}

```
1: Método enq(nuevoValor) :
2:     return ok
3: Método deq() :
4:         return null
```

Dado un algoritmo V libre de esperas que utiliza operaciones con número de consenso infinito y que resuelve la VLTE de A_{cola} con respecto a un objeto secuencial O, existen tres ejecuciones en las que dos procesos p y q participan

y realizan las operaciones $\langle enq(x) : ok\rangle$ y $\langle deq() : null\rangle$, respectivamente, en ordenes distintos.

Estas tres ejecuciones son α_{cola}, β_{cola} y γ_{cola}, y se muestran en las figuras 4.7, 4.8 y 4.9, respectivamente.

A pesar de que la ejecución α_{cola} consiste de $\langle deq() : null\rangle \rightarrow \langle enq(x) : ok\rangle$, la ejecución β_{cola} consiste de $\langle enq(x) : ok\rangle \rightarrow \langle deq() : null\rangle$ y la ejecución γ_{cola} consiste de ni $\langle deq() : null\rangle \rightarrow \langle enq(x) : ok\rangle$ ni $\langle enq(x) : ok\rangle \rightarrow \langle deq() : null\rangle$. La historia que detecta V_{VL} es la misma en las tres ejecuciones:

$$\rightarrow \langle V_d(\ldots) : ok\ p\rangle \rightarrow \langle V_d(\ldots) : ok\ q\rangle \rightarrow \langle V_d(\ldots) : ok\ q\rangle \rightarrow \langle V_d(\ldots) : ok\ p\rangle.$$

Esto sucede debido a la existencia de retardos:

- En la ejecución α_{cola}, figura 4.7, el proceso p tiene un retardo r_i y un retardo r_j antes y después, respectivamente, de ejecutar la operación op_p del algoritmo A_{cola}. El proceso q tiene un retardo r_k antes de ejecutar la operación op_q del algoritmo A_{cola}.

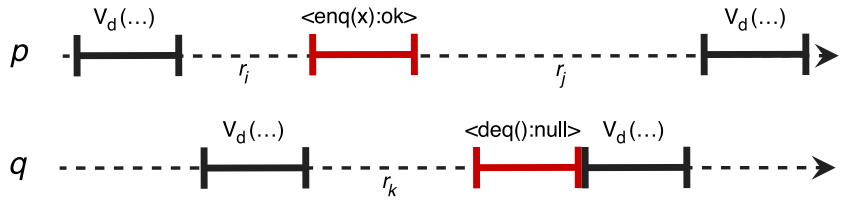


Figura 4.7: V detecta la ejecución α_{cola}

- En la ejecución β_{cola}, figura 4.8, el proceso p tiene un retardo r_i y un retardo r_j antes y después, respectivamente, de ejecutar la operación op_p del algoritmo A_{cola}. El proceso q tiene un retardo r_k después de ejecutar la operación op_q del algoritmo A_{cola}.

- En la ejecución γ_{cola}, figura 4.9, el proceso p tiene un retardo r_i y un retardo r_j antes y después, respectivamente, de ejecutar la operación op_p. El proceso q tiene un retardo r_k y un retardo r_l antes y después, respectivamente, de ejecutar la operación op_q del algoritmo A_{cola}.

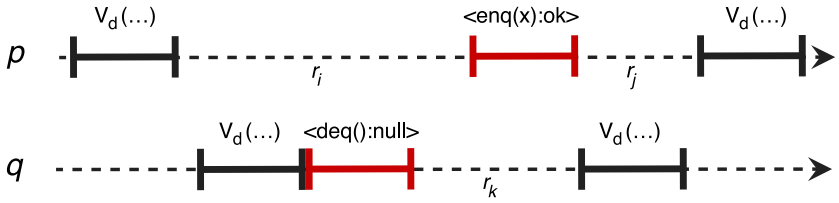


Figura 4.8: V detecta la ejecución β_{cola}

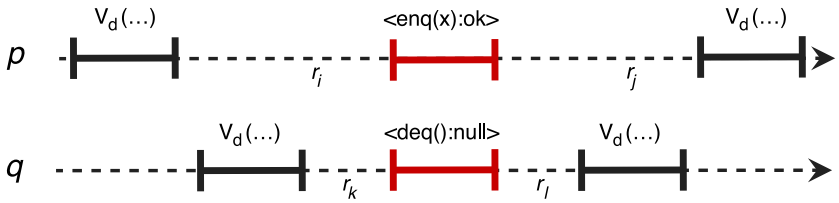


Figura 4.9: V detecta la ejecución γ_{cola}

En las tres ejecuciones los procesos obtienen la misma historia al ejecutar localmente el método V_{VL}, por lo tanto computan lo mismo independientemente del orden en el que sucedieron las operaciones op_p y op_q en A_{cola}. Esto sucede incluso si V utiliza operaciones con número de consenso infinito porque la existencia de retardos no está relacionada con la eficiencia de V sino con la asincronía de los procesos.

Se cumple lo que supone el lema 1 para el algoritmo A_{cola}: existen tres ejecuciones con las mismas operaciones en órdenes distintos que V no puede distinguir y por lo tanto V_{VL} devuelve el mismo resultado en las tres ejecuciones.

Se puede ver con facilidad que $\langle deq() : null\rangle \rightarrow \langle enq(x) : ok\rangle$ es una historia secuencial válida pero $\langle enq(x) : ok\rangle \rightarrow \langle deq() : null\rangle$ no lo es con respecto a O; esto implica que el algoritmo A_{cola} no es linealizable con respecto a O.

Dado que se cumple el lema 1 para A_{cola}, el algoritmo V devuelve el mismo resultado en las tres ejecuciones.

Existe al menos una ejecución que sí es válida con respecto a O, esta es la ejecución α_{cola}. Si ambos procesos computan $falso$ en las tres ejecuciones, entonces algún proceso que invoca V_{VL} de forma local computa $falso$, por lo tanto no se cumple la propiedad de precisión.

En cambio si en las tres ejecuciones ambos procesos computan *verdadero*, entonces algún proceso que invoca V_{VL} de forma local computa *verdadero*

cuando las ejecuciones β_{cola} y γ_{cola} no son válidas con respecto a O, por lo tanto V no cumple la propiedad de completez.

Se cumple lo que supone el corolario 1: V incumple con alguna de las propiedades de precisión y completez de la definición de la VLTE.

4.3. Discusión

El algoritmo V se supone libre de esperas para no afectar el progreso del algoritmo A. Por ejemplo, si V es libre de candados y A es libre de esperas, solo algunos procesos podrán participar en el sistema de detección del algoritmo V. Además si V utiliza una condición de progreso bloqueante puede perjudicar el progreso del algoritmo A de forma que si los procesos se detienen en V, también se detendrán en A.

Un gran desafío en este problema es que los algoritmos V y A están desacoplados, esto origina la existencia de retardos entre lo que sucede en el algoritmo A y lo que sucede en el algoritmo V, sin importar si V utiliza operaciones con número de consenso infinito. Una forma intuitiva de eliminar este problema es haciendo que los algoritmos A y V sean uno solo; sin embargo, esto perjudica las condiciones de corrección y de progreso del algoritmo A.

De hecho la imposibilidad de este problema ha sido mencionada de manera no formal en algunos trabajos donde desarrollan algoritmos para solucionar la VLTE. En ellos utilizan herramientas para tener la mejor aproximación al estado actual real, es decir que resuelven el consenso sobre la mejor aproximación al sistema. Por ejemplo, en [33] el consenso sobre el estado actual del sistema depende de un oráculo que elige el estado actual "real". En [35] afirman que el vector de conocimiento que construyen es la mejor aproximación al estado real actual del sistema. En [17] se apoyan de un reloj global que permite terminar las operaciones y hacer suposiciones sobre la indistinguibilidad del sistema real actual.

Esta demostración de imposibilidad comparte la noción de indistinguibilidad de la demostración de imposibilidad del consenso en un sistema asíncrono con fallas bizantinas presentada en [9], donde la clave de esta noción es el conjunto de decisiones a las que se puede llegar a partir de una configuración particular. Esto sucede debido a la naturaleza asíncrona de los sistemas. En

un sistema asíncrono un proceso que falla no puede ser distinguido de uno que no falla; esto en la demostración se traduce como que la existencia y la longitud de un retardo no se puede distinguir.

El problema de la VLTE es un problema particular del problema de observar el tiempo real. Observar el tiempo real es imposible debido a la indistinguibilidad [8].

Otro ejemplo del problema de observar el tiempo real, además de la VLTE, es el de la sincronización de relojes. Las limitaciones de este problema pueden ser explicadas a través de la imposibilidad de distinguir la ocurrencia de eventos en tiempo real [8, 30].

En [34] se ha estudiado que un evento pudo haber ocurrido en cualquier unidad de tiempo en un intervalo, es más, para cada unidad de tiempo existe una ejecución indistinguible para todos los procesos [8].

Debido a la imposibilidad de observar el tiempo real [8], el que un algoritmo V pueda o no resolver el consenso no implica que pueda resolver la VLTE porque el tiempo de ejecución de un algoritmo a verificar A es por sí solo un sistema asíncrono con fallas. De acuerdo con las imposibilidades en [9] existirían más posibilidades si se convirtiera a A en un sistema síncrono y si fuera posible garantizar que los procesos no fallaran. Sin embargo, si se realiza lo primero, como ya se describió, se afectarían las condiciones de corrección y de progreso del algoritmo A, y realizar lo segundo no es posible debido a la naturaleza asíncrona de los procesos.

Capítulo 5

Verificación débil de la linealizabilidad en tiempo de ejecución

El problema de la VLTE es imposible de solucionar. Sin embargo, se puede evadir este problema al solucionar una versión relajada del mismo.

En este capítulo se presenta una formalización del problema de la verificación débil de la linelizabilidad en tiempo de ejecución, al cual denotaremos como VDLTE. Después se presenta un algoritmo que soluciona el problema de la VDLTE.

5.1. Formalización del problema de la VDLTE

Este problema, al igual que el problema de la VLTE, consiste en monitorear y comprobar de forma distribuida la linealizabildad de cualquier algoritmo que se especula es linealizable. Sin embargo, a diferencia del problema de la VLTE, este problema permite que la verificación de la linealizabilidad sea más relajada, para ello se relaja la propiedad de completez y se mantiene la propiedad de precisión de la VLTE.

Sea A un algoritmo concurrente que se presume es linealizable con respecto a un objeto O. Al igual que el problema de la VLTE, el modelo de este problema consiste en que un conjunto de procesos se comunica sobre

una memoria compartida para detectar la ejecución actual del algoritmo A y emitir un veredicto, sobre si es linealizable o no con respecto al objeto O. A todo el trabajo que los procesos desempeñan desde comunicarse entre ellos hasta emitir el veredicto le denotaremos como sistema o algoritmo verificador V.

El algoritmo A se representa como una caja negra lo que quiere decir que su contenido no se puede inspeccionar, alterar o controlar las ejecuciones: sus invocaciones y/o respuestas.

Como el algoritmo A y el sistema verificador V están desacoplados su diseño e implementación se realizan de forma independiente.

Un sistema verificador V se puede analizar en dos sistemas, el sistema de detección y el sistema de verificación de la linealizabilidad, ambos son idénticos a los sistemas del problema de la VLTE.

El algoritmo V soluciona el problema de la VDLTE si se cumple lo siguiente cada vez que un proceso utilice el sistema de verificación de la linealizabilidad:

1. Precisión: si la historia en tiempo de ejecución del algoritmo A es linealizable con respecto al objeto O, entonces el algoritmo V siempre computará *verdadero*.

2. Completez relajada: si el algoritmo V computa *verdadero*, entonces

   a) la historia en tiempo de ejecución del algoritmo A es linealizable con respecto al objeto O, o

   b) la historia en tiempo de ejecución del algoritmo A no es linealizable con respecto al objeto O pero la historia de V sí es linealizable con respecto al objeto O.

El algoritmo V debe ser libre de esperas para no afectar el progreso del algoritmo A porque si V utiliza una condición de progreso bloqueante o es libre de candados, solo algunos procesos podrían participar en el sistema de detección.

Estas dos propiedades implican que si A es linealizable con respecto a O, entonces la historia abstracta que detecta V del algoritmo A es también linealizable con respecto a O. También implica que si A no es linealizable,

entonces la historia abstracta que detecta V del algoritmo A puede ser linealizable o no con respecto a O.

Si V cumple estas dos propiedades, entonces V, al detectar en tiempo de ejecución al algoritmo A, puede encontrar un certificado que determine que el algoritmo A no es linealizable. A este certificado lo denotaremos como *historia causante*.

Si el algoritmo V detecta una historia causante, el algoritmo A es *permanentemente no linealizable*.

Si el algoritmo V no detecta una historia causante, el algoritmo A es *presuntamente linealizable*.

Estas dos propiedades caracterizan al algoritmo A debido a que la linealizabilidad es composicional (teoremas 1 y 2). Si A es presuntamente linealizable, existe la posibilidad de que A no lo sea y V no haya detectado aún una historia causante.

5.2. Algoritmo de VDLTE

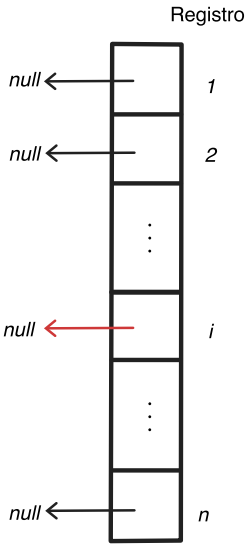
El algoritmo usa una implementación concurrente de un objeto de tipo snapshot atómico. Los procesos involucrados en la resolución de la VDLTE de cualquier algoritmo A, comparten cada operación que realizan sobre el algoritmo A en un objeto compartido para escribir la historia de la ejecución actual. Cualquier proceso involucrado puede obtener de forma local la historia de la ejecución actual y verificar su linealizabilidad.

A este algoritmo lo denotaremos como algoritmo W.

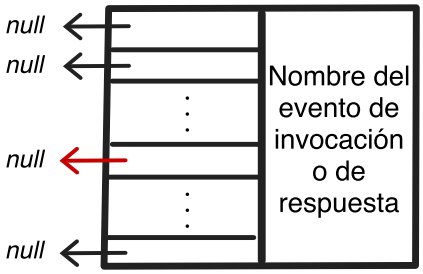
5.2.1. Descripción detallada del algoritmo W

El sistema de detección del algoritmo W lo denotaremos como W_d y al sistema de verificación de la linealizabildad lo denotaremos como W_{VL}.

Para resolver la VDLTE de cualquier algoritmo A que implemente un objeto O, los procesos utilizan el algoritmo de W_d en 3 para detectar la historia de la ejecución actual en tiempo de ejecución del algoritmo A. Después cada proceso de forma local puede obtener la historia detectada utilizando el algoritmo de W_{VL} en 4.



(a) Representación abstracta de la inicialización del arreglo compartido



(b) Representación abstracta del objeto de tipo nodo

Figura 5.1: Representación abstracta de los objetos del algoritmo W

La inicialización del arreglo compartido (línea 1) que se muestra con más detalle en las líneas 20 a 22, consiste en crear una tabla o arreglo de registros MRSW de tamaño del número de procesos involucrados e inicializar cada registro a *null*, como se muestra en la figura 5.1a. Cada proceso p_i puede escribir únicamente en su registro i.

Cada registro almacena un apuntador a un nodo como se muestra en las líneas 14 a 17. Un nodo (figura 5.1b) es un objeto que almacena el nombre de un evento de invocación o de respuesta y un arreglo de registros de tamaño del número de procesos involucrados tal como se muestra en las líneas 7 a 13. Cada registro i en este arreglo de registros apunta a un nodo creado por un proceso p_i o a *null* si no existen nodos creados por el proceso p_i. Este arreglo de registros representa el conjunto de eventos más recientes de cada proceso que preceden al evento de invocación o respuesta del nodo.

Este mecanismo de nodos y apuntadores está basado en la construcción de un grafo de precedencia que se presenta en [5]. Este mecanismo representa el orden parcial de las operaciones de alguna historia.

El algoritmo 3 de W_d consiste en que cada vez que un proceso p_i quiera realizar una operación del algoritmo A, debe guardar la invocación y la respuesta de la operación de A en su registro i (líneas 4 a 6) tal como se muestra en la expresión 5.1.

$$\langle update(invocación\ de\ A)\rangle \rightarrow \langle operación\ de\ A\rangle \rightarrow \langle update(respuesta\ de\ A)\rangle \tag{5.1}$$

El método $update(eventoNombre)$ (líneas 23 a 26) consiste en obtener de forma atómica, por medio de una operación $scan()$, un arreglo de registros de tamaño del número de procesos involucrados (línea 24); existen implementaciones libre de esperas con operaciones de lectura/escritura en [2, 7, 29]. Después se crea un nodo en el se guarda el nombre del evento de invocación o de respuesta y el resultado de la operación $scan()$ en el arreglo de registros del nodo (línea 25). Por último cada proceso p_i actualiza el valor de su registro i con la dirección del nuevo nodo. Un ejemplo de la expresión 5.1 se muestra en la figura 5.2, en la que un proceso p_i añade su operación sin que ningún otro proceso aparezca aún en la ejecución de la siguiente forma: en la subfigura 5.2a añade su evento de invocación y en la subfigura 5.2b añade su evento de respuesta.

Cada vez que un proceso quiera verificar la historia abstracta descrita por el algoritmo W_d, debe invocar al algoritmo 4 de W_{VL} de forma local. Primero obtiene la *historia completa*, la cual es la historia abstracta descrita (líneas 27 a 30), después en la línea 34 se utiliza un algoritmo de verificación de la linealizabilidad (algunos ejemplos se encuentran en [16]) para obtener un veredicto sobre si la historia es linealizable o no con respecto al objeto O.

Algoritmo 3 Algoritmo W_d

```
1: Se inicializa el snapshot con el numero de procesos involucrados.
2: W_d():
3:     if se quiere realizar una operación del algoritmo A do
4:         update(invocación de A)
5:         Realiza la operación de A
6:         update(respuesta de A)
7: Clase Nodo():
8:     Cadena evento
9:     Registro vista[]
10:    constructor(eventoNombre, vistaNueva[]):
11:        this.evento = eventoNombre
12:        for i de 0 a vistaNueva.tamaño, i++ do
13:            this.vista[i] = Registro(vistaNueva[i].nodo)
14: Clase Registro():
15:    Nodo nodo
16:    constructor(nuevoNodo):
17:        this.nodo = nuevoNodo
18: Clase ArregloCompartido():
19:    Registro tabla[]
20:    constructor(capacidad):
21:        for i de 0 a capacidad, i++ do
22:            this.tabla[i] = Registro(null)
23:    void update(Cadena eventoNombre):
24:        vistaNueva = scan()
25:        Nodo nodo = nuevo Nodo(eventoNombre, vistaNueva)
26:        tabla[miID].nodo = nodo
```

5.2.2. Ejemplo del algoritmo W

Para ejemplificar el funcionamiento del algoritmo W suponemos que A es un algoritmo que se presume es linealizable e implementa una cola.

Suponemos que se desarrollan dos ejecuciones con solo dos procesos, p y q, cada uno con su respectivo registro.

Algoritmo 4 Algoritmo W_{VL}

```
27: scanCompleto():
28:         Inicializa una historia completa
29:         Guarda cada nodo de cada registro en la historia completa
30:         return historia completa
31: W_VL():
32:         historiaCompleta = scanCompleto()
33:         veredicto = Verifica la historia completa
34:         return veredicto
```

- Ejecución 1.

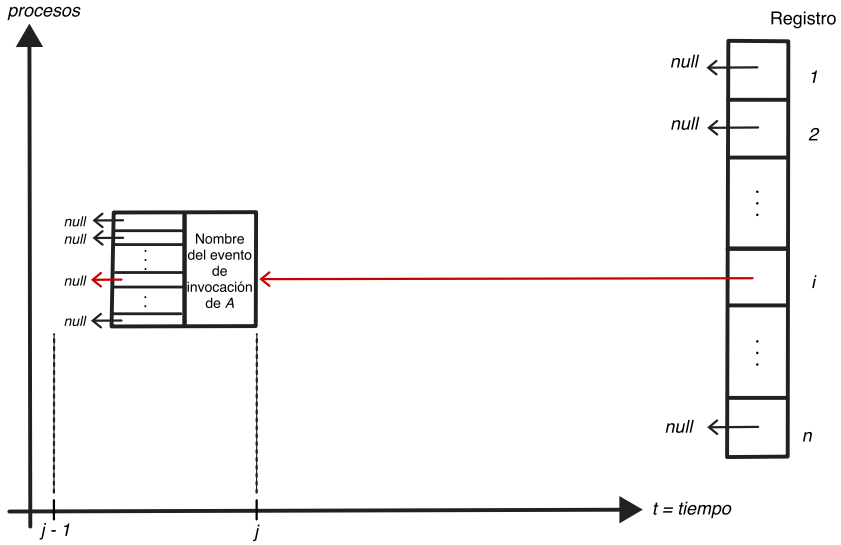
  La historia de la ejecución del algoritmo A que se muestra en rojo en la figura 5.3 es: $\rightarrow \langle enq(x) : ok\ p\rangle \rightarrow \langle deq() : x\ q\rangle$, podemos determinar con facilidad que es una historia linealizable con respecto a una cola.

  La historia de la ejecución del algoritmo W que se muestra la figura 5.3 en color negro es: $\rightarrow \langle update(enq(x)) : ok\ p\rangle \rightarrow \langle update(ok) : ok\ p\rangle \rightarrow \langle update(deq()) : ok\ q\rangle \rightarrow \langle update(x) : ok\ q\rangle$. Esta historia se puede representar de forma abstracta en el snapshot como se muestra en la figura 5.4.

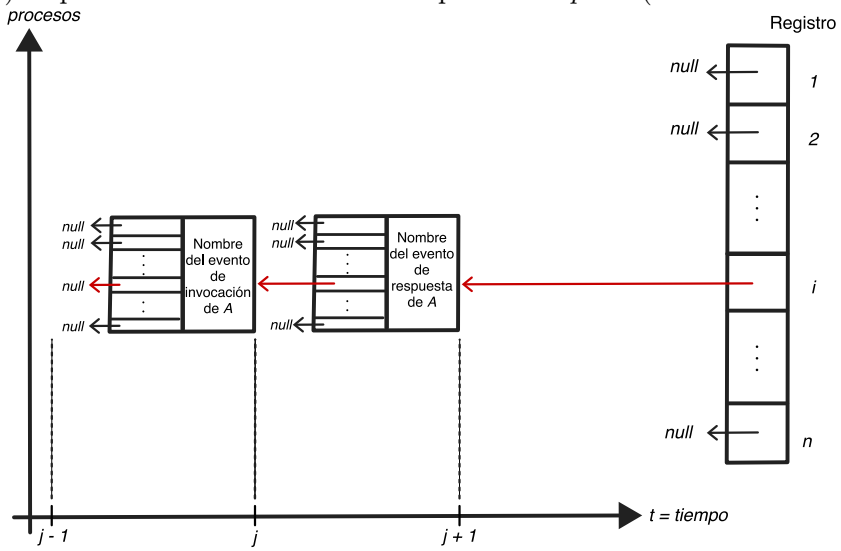
  Cada proceso a partir de la historia detectada invoca de forma local el método W_{VL}.

  Supongamos que el proceso p invoca el método W_{VL} justo después de realizar su operación $\langle update(ok) : ok\rangle$ (en el tiempo 3 de la figura 5.4), entonces obtiene la siguiente *historia completa* definida por los nodos del registro p que son el 1 y el 2. Si al nodo 1 no le precede ninguno y al nodo 2 solo le precede el nodo 1, se construye la historia $\rightarrow \langle enq(x) : ok\ p\rangle$, y W_{VL} computa *verdadero* porque la *historia completa* es linealizable con respecto al objeto O.

  Supongamos que el proceso q invoca el método W_{VL} justo después de realizar su operación $\langle update(x) : ok\rangle$ (en el tiempo 4 de la figura 5.4), entonces obtiene la siguiente *historia completa* definida por los nodos del registro p que son el 1 y el 2, y los nodos del registro q que son el 3 y el 4. Si al nodo 3 le precede el nodo 2 de respuesta del proceso p, y si al nodo 4 le precede el nodo 3, entonces se construye la historia $\rightarrow \langle enq(x) : ok\ p\rangle \rightarrow \langle deq() : x\ q\rangle$, y W_{VL} computa *verdadero* porque



(a) Representación abstracta de la operación *update*(*invocación de A*)



(b) Representación abstracta de la operación *update*(*respuesta de A*)

Figura 5.2: Representación de la expresión $\langle update(invocación\ de\ A)\rangle \rightarrow \langle operación\ de\ A\rangle \rightarrow \langle update(respuesta\ de\ A)\rangle$ por el proceso p_i en el registro i

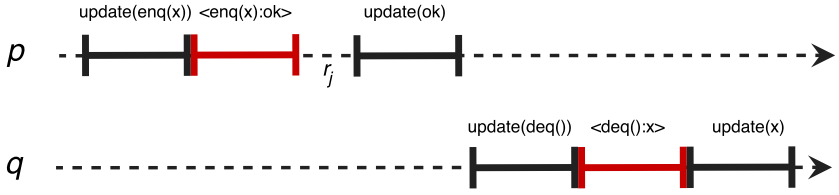


Figura 5.3: Historia de la ejecución 1

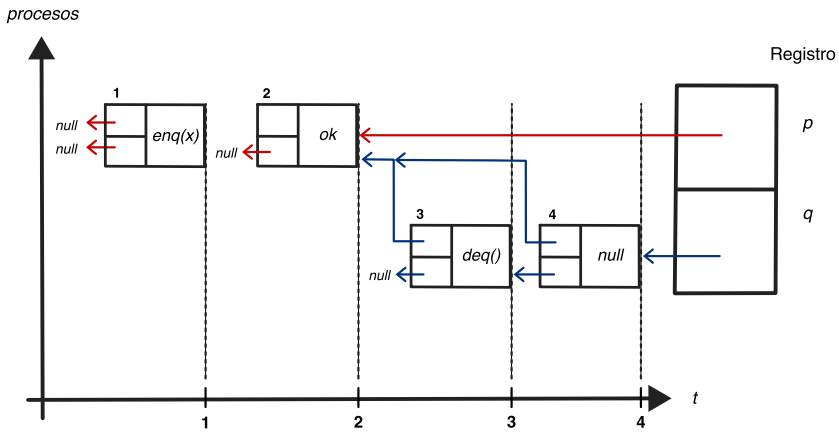


Figura 5.4: Representación la ejecución 1: ejemplo del algoritmo W

es linealizable con respecto al objeto O.

En este caso el algoritmo W resuelve la VDLTE del algoritmo A porque cuando la ejecución 1 es linealizable el algoritmo W computa verdadero, por lo tanto se cumple la propiedad de precisión. Cuando el algoritmo W computa verdadero la ejecución de A es linealizable, por lo tanto se cumple la propiedad de completez relajada.

- Ejecución 2.

  La historia de la ejecución del algoritmo A que se muestra en rojo en la figura 5.5 es: $\rightarrow \ \langle enq(x) : ok\ p\rangle \ \rightarrow \ \langle deq() : null\ q\rangle$. Podemos determinar con facilidad que esta historia no es linealizable con respecto a una cola.

  La historia de la ejecución del algoritmo W que se muestra la figura 5.5 en color negro es: $\rightarrow$ ($\langle update(enq(x)) : ok\ p\rangle$ y $\langle update(deq()) : ok\ q\rangle$ en orden indistinto) $\rightarrow \langle update(null) : ok\ q\rangle \ \rightarrow \langle update(ok) : ok\ p\rangle$.

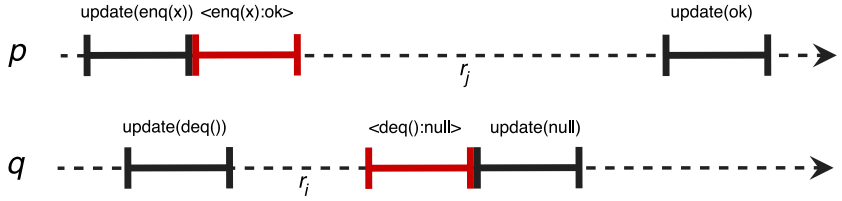


Figura 5.5: Historia de la ejecución 2

Esta historia se puede representar de forma abstracta en el snapshot como se muestra en la figura 5.6.

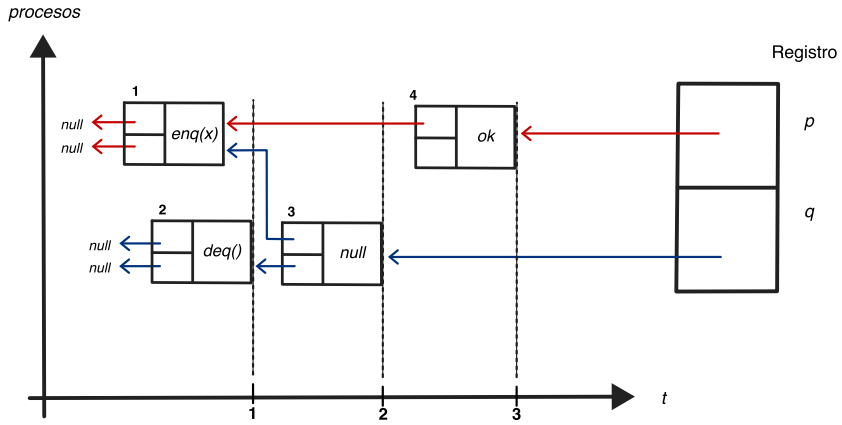


Figura 5.6: Representación la ejecución 2: ejemplo del algoritmo W

Cada proceso, a partir de la historia detectada, invoca de forma local el método W_{VL}.

Supongamos que ambos procesos, p y q, invocan el método W_{VL} justo después de la operación $\langle update(ok) : ok\ p\rangle$ (en el tiempo 3 de la figura 5.6), entonces ambos obtienen la misma *historia completa* definida por los nodos del registro p que son el 1 y el 4, y por los nodos del registro q que son el 2 y el 3. Si a los nodos 1 y 2 no les precede ninguno, si al nodo 3 le precede el 1 y el 2, y si al nodo 4 le precede el 1 y el 3, entonces se construye la historia parcial ni $\rightarrow \langle enq(x) : ok\ p\rangle \rightarrow \langle deq() : null\ q\rangle$ ni $\rightarrow \langle deq() : null\ q\rangle \rightarrow \langle enq(x) : ok\ p\rangle$, y cada proceso utiliza el método W_{VL} para computar de forma local *verdadero* porque $\rightarrow \langle deq() : null\ q\rangle \rightarrow \langle enq(x) : ok\ p\rangle$ es linealizable con respecto al

objeto O.

En este caso el algoritmo W resuelve la VDLTE del algoritmo A. Aunque W no es capaz de detectar que la historia de A no es linealizable con respecto al objeto O, la historia en el algoritmo W sí es linealizable con respecto al objeto O y debido a esto W computa verdadero. Por lo tanto se cumple la propiedad de completez relajada.

5.2.3. Corrección del algoritmo W

Lema 2. *El algoritmo W es libre de esperas.*

El algoritmo W es libre de esperas porque el método $update(x)$ es libre de esperas. Debido a que suponemos que el método $scan()$ (línea 24) es atómico porque utiliza una implementación libre de esperas en la literatura [2, 7, 29]. Además cada proceso puede crear un nodo (línea 25) y actualizar su registro en el arreglo compartido (línea 26) sin interferir con otros procesos debido a que el método $scan()$ es libre de esperas y los registros son de MRSW.

Lema 3. *El algoritmo W siempre codifica una historia bien formada.*

Suponemos que la historia en el algoritmo A está bien formada, entonces el algoritmo W codifica una historia bien formada de A porque el método $update(x)$ se ejecuta antes y después de cada operación del algoritmo A como se muestra en la expresión 5.1.

En cada ejecución del método $update(x)$ cada proceso p_i añade un nuevo nodo $nodo_j$ que apunta a su último nodo $nodo_{j-1}$.

Para ello el método $scan()$ devuelve de forma atómica un arreglo con las direcciones de los últimos nodos creados por cada proceso p_i (línea 24), entonces la dirección del último nodo $nodo_{j-1}$ por p_i se encuentra en este arreglo. En la línea 25 esta dirección se guarda en el arreglo del nuevo nodo $nodo_j$ de p_i. Por último en la línea 26 este nuevo nodo $nodo_j$ se convierte en el último nodo creado por p_i. Este mecanismo ya se ha utilizado para codificar historias bien formadas [5].

De tal forma que en la expresión 5.1, por cada operación de A, el primer nodo que se añade codifica el evento de invocación de la operación de A y el segundo nodo que se añade codifica el evento de respuesta de la misma operación de A. Por lo tanto si la historia en A está bien formada, cada proceso

p_i codifica una historia secuencial de A en W.

Teorema 9. *El algoritmo W cumple con las propiedades de precisión y completez relajada*

Debido a que el algoritmo W codifica una historia bien formada del algoritmo A (lema 3), el intervalo de tiempo de las operaciones $update(x)$ de la expresión 5.1 codifica el intervalo del evento de invocación al evento de respuesta de cada operación de A en el algoritmo W.

Al intervalo de cada operación k de A codificado en W le denotaremos como $T_{W,k}$. Al intervalo de tiempo de cada operación k de A le denotaremos como $T_{A,k}$. Entonces para toda operación k siempre se cumple que $T_{W,k} \geq T_{A,k}$, es decir que el intervalo de cada operación de A codificado en W envuelve o contiene el intervalo de cada operación de A tal como se muestra en la figura 5.7.

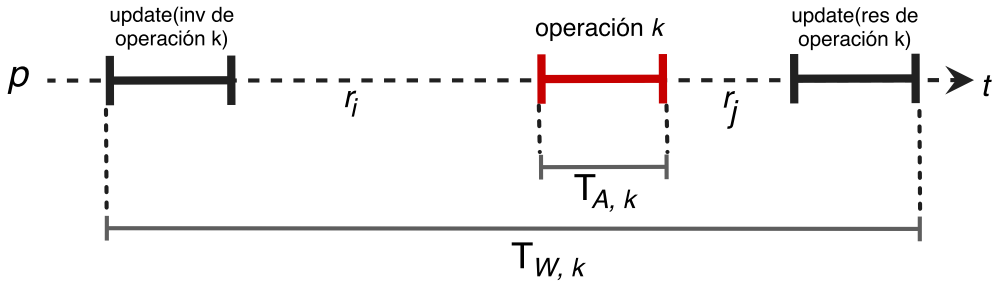


Figura 5.7: El intervalo de la operación k codificado en W, $T_{W,k}$, contiene al intervalo $T_{A,k}$ de la operación k en la ejecución del algoritmo A

Sea S_{VL} el conjunto de todas las posibles linearizaciones con respecto a O de la codificación de A en W y sea S_A el conjunto de todas las linearizaciones con respecto a O de la ejecución de A. Debido a que cada $T_{W,k}$ contiene a cada $T_{A,k}$, pueden suceder dos casos:

1. El orden de las operaciones de la ejecución de A codificadas en W es el mismo que el orden de las operaciones de la ejecución del algoritmo A. Por lo tanto $S_{VL} = S_A$.

2. En la codificación de las operaciones de la ejecución de A en W se traslapan operaciones que no se traslapan en la ejecución del algoritmo A.

> Entonces el algoritmo W_{VL} puede ordenar las operaciones que preceden a otras en la ejecución de A en orden indistinto en la codificación de A en W. Por lo tanto $S_A \subset S_{VL}$.

Suponemos que la ejecución del algoritmo A es linealizable con respecto al objeto O.

Si sucede el primer caso, entonces $S_{VL} = S_A$; además, como la ejecución del algoritmo A es linealizable, entonces $S_A \neq \emptyset$. Por lo tanto $S_{VL} \neq \emptyset$ y W_{VL} computa verdadero.

Si sucede el segundo caso, entonces $S_A \subset S_{VL}$; además, como la ejecución de A es linealizable, entonces existe al menos una linearización $s_A \in S_A$ tal que $s_A \in S_{VL}$ por lo tanto $S_{VL} \neq \emptyset$ y computa verdadero.

Como en ambos casos W_{VL} computa verdadero entonces, siempre que una ejecución del algoritmo A es linealizable, el algoritmo W computa verdadero, por lo tanto cumple con la propiedad de precisión.

Suponemos que W_{VL} computa verdadero dada la historia de A codificada en W.

Si sucede el primer caso, entonces $S_{VL} = S_A$. Solo si $S_A \neq \emptyset$ entonces $S_{VL} \neq \emptyset$. Por lo tanto la codificación de A en W y la ejecución de A son linealizables. Entonces se cumple el caso a) de la propiedad de completez relajada.

Si sucede el segundo caso, entonces $S_A \subset S_{VL}$. Si $S_A \neq \emptyset$, entonces existe una linearización $s_A \in S_A$ tal que $s_A \in S_{VL}$. Por lo tanto la codificación de A en W y la ejecución de A son linealizables, debido a esto se cumple el caso a) de la propiedad de completez relajada. Si $S_A = \emptyset$, entonces puede existir una linearización $s_{VL} \in S_{VL}$ tal que $s_{VL} \notin S_A$. Por lo tanto la codificación de A en W es linealizable pero la ejecución de A no es linealizable con respecto a O, debido a esto se cumple el caso b) de la propiedad de completez relajada.

W puede ordenar las operaciones que se traslapan en la codificación de A en W que no se traslapan en la ejecución de A. Entonces W_{VL} puede computar verdadero cuando la historia de la ejecución de A no es linealizable con respecto al objeto O, por lo tanto se cumple el caso b) de la propiedad de completez relajada.

5.3. Discusión

Los snapshots son muy útiles porque permiten obtener estados globales consistentes de la memoria mientras se actualiza la memoria de forma concurrente [6].

En el algoritmo W el estado global consistente representa la historia de la ejecución que detecta W de cualquier algoritmo A que se verifique.

El algoritmo W permite resolver el problema de la VDLTE. Sin embargo, este algoritmo no puede resolver la VLTE porque no cumple con la propiedad de completez. Al relajar la propiedad de completez se permite que los algoritmos que resuelven el problema de la VDLTE tengan falsos positivos, es decir que computen verdadero cuando la ejecución del algoritmo A no es linealizable con respecto al objeto O pero la codificación del algoritmo A sí lo es.

Existen varios algoritmos concurrentes que tratan de resolver la VLTE. Por ejemplo, en [18] realizan un experimento práctico. En este experimento ejecutan tres algoritmos en JAVA para resolver la VLTE de un algoritmo que implementa un objeto de tipo cola; sin embargo, en las mismas ejecuciones cada uno de los tres algoritmos devuelve diferentes veredictos sin ningún patrón. Esto se debe a que en realidad los algoritmos no resuelven el problema de la VLTE sino un problema más relajado.

A diferencia de estos algoritmos implementados en JAVA [18], el algoritmo W tiene un análisis de corrección que permite comprender lo que representan los veredictos que devuelve como resultado. Debido a que el algoritmo W codifica una historia bien formada del algoritmo A (lema 3), el intervalo de tiempo de las operaciones $update(x)$ de la expresión 5.1 codifica el intervalo del evento de invocación al evento de respuesta de cada operación de A en el algoritmo W.

Capítulo 6

Conclusiones

Para entender el problema de la verificación de la linealizabilidad, en el capítulo 2 se presenta un modelo de cómputo concurrente en el cual se profundiza sobre las condiciones de corrección y progreso de los algoritmos concurrentes. Como en este trabajo se utiliza a la linealizabilidad como noción de corrección, la importancia de la verificación de la linealizabilidad radica en que un algoritmo es correcto solo si es linealizable.

El problema de la verificación de la linealizabilidad ya ha sido estudiado y se reveló como un problema computacionalmente difícil [1,3,11] que puede ser abordado con diferentes técnicas [31], siendo una de ellas la técnica de la verificación en tiempo de ejecución.

Al comparar la técnica de la verificación de la linealizabilidad en tiempo de ejecución con las demás técnicas, se muestra que esta técnica es capaz de verificar el algoritmo una vez esté desplegado sin la necesidad de conocer previamente su comportamiento. Esta característica también permite incorporar sistemas de detección y corrección de errores sobre el sistema a verificar.

A partir del análisis realizado en el artículo [10] podemos analizar los algoritmos que tratan de resolver la VLTE en dos grupos: los que resuelven el problema del consenso y los que no. Cada proceso puede tener un veredicto diferente sobre la ejecución actual cuando no lo resuelven, de hecho existe una cota inferior en el número de veredictos necesarios para poder distinguir una ejecución válida [21]. En los algoritmos que resuelven el problema del consenso sobre la ejecución actual, los procesos se ponen de acuerdo en un solo veredicto; sin embargo, resolver el problema del consenso no se puede solucionar en sistemas asíncronos con una sola falla [9,19], además para resol-

verlo en sistemas síncronos se requiere un determinado número de rondas [9].

Debido a que, en [10], una forma de resolver el problema de la VLTE es resolviendo el problema del consenso sobre la ejecución actual en tiempo de ejecución, se construyó el algoritmo W basado en el grafo de precedencia de [5], como se abordó en el capítulo 5. Este algoritmo permite que los procesos construyan órdenes parciales de forma concurrente para que, de forma local, cada proceso pueda verificar la historia.

Sin embargo al realizar el análisis teórico de corrección del algoritmo se desveló que el algoritmo W no resuelve en todos los casos el problema de la VLTE, debido a que no hay forma de saber si la ejecución detectada es la ejecución que sucedió en realidad en el sistema a verificar. Es más, se descubrió que ningún algoritmo es capaz de resolver el problema de la VLTE incluso si se utilizan operaciones primitivas con un número de consenso infinito, como se muestra en el capítulo 4.

La noción de indistinguibilidad de la demostración de imposibilidad del consenso en un sistema asíncrono con fallas bizantinas [9] permite comprender que el problema de la VLTE no puede ser resuelto debido a que los procesos son incapaces de distinguir la duración, el tamaño y la existencia de los retardos que suceden debido a la asincronía, así como los procesos son incapaces de distinguir procesos bizantinos en un sistema asíncrono.

6.1. Problemas relacionados con el tiempo

Los problemas basados en el tiempo dependen de cuantificar la indistinguibilidad del tiempo real [8].

En [30] Lamport establece la relación *precedió a* (*happened before*), la cual consiste en describir que un determinado evento sucedió antes que otro evento. Lamport define esta relación porque no se pueden utilizar relojes físicos debido a que no son precisos. Sin embargo, esta relación es solo un orden parcial porque el orden en el que ciertos eventos ocurren en el tiempo es impredecible debido a la relación con la teoría de la relatividad.

Las limitaciones del problema de la sincronización de relojes y la imposibilidad de la VLTE se deben también a la imposibilidad de distinguir la ocurrencia de los eventos en una ejecución.

De hecho, cada evento en una ejecución sucede en un intervalo de indistinguibilidad, es decir que cada evento en realidad pudo haber ocurrido en cualquier tiempo en ese intervalo de tiempo [8].

6.2. Contribución

La formalización del problema de la VLTE y su prueba de imposibilidad permiten comprender que el problema de la VLTE no puede ser resuelto incluso cuando se utilicen primitivas con un número de consenso infinito. Esto debido a que no es posible distinguir la ocurrencia de ciertos eventos en una ejecución por la existencia de retardos inesperados y de longitud impredecible. Esta indistinguibilidad afecta el veredicto sobre si la ejecución es o no linealizable y por lo tanto pueden no cumplirse las propiedades de completez y de precisión.

Además, dado el capítulo 4, se pueden definir problemas más relajados para evadir el problema de la VLTE que sí puedan ser resueltos y permitan desarrollar soluciones consistentes con su definición. Un ejemplo de ello se presenta en el capítulo 5, en el cual se formaliza un problema relajado de la VLTE, el problema de la VDLTE. Este problema puede ser resuelto con un algoritmo libre de esperas que utiliza operaciones de lectura/escritura.

Este trabajo permite comprender por qué en trabajos que tratan de resolver el problema de la VLTE [17,33,35] se utilizan herramientas como relojes globales para eliminar la concurrencia del algoritmo a verificar u oráculos para decidir lo que sucedió en "realidad" en el algoritmo a verificar.

Al eliminar la concurrencia, la verificación se realiza siempre en un sistema síncrono y se puede resolver el consenso en un determinado número de rondas; sin embargo, esto ocasiona que se modifique el comportamiento del sistema a verificar y resolver el consenso no implica resolver la VLTE.

Decidir de alguna forma lo que sucede en la "realidad" no es posible debido a la indistinguibilidad de la ocurrencia de eventos [30], por lo tanto se incumplen las propiedades de precisión o de completez.

Este trabajo también explica por qué dada la misma ejecución del mismo algoritmo a verificar tres algoritmos diferentes obtienen diferentes veredictos sin ningún patrón, como se desarrolla en [18]. En realidad estos tres algoritmos resuelven problemas más relajados de la VLTE porque la VLTE no es

posible de solucionar; sin embargo, no se tiene un análisis de los problemas que sí resuelven.

6.3. Trabajo a futuro

Se considera implementar el algoritmo W utilizando una implementación eficiente de un snapshot atómico como en [29] en un lenguaje de programación que permita la programación multihilo como JAVA y realizar una evaluación experimental del comportamiento del algoritmo en la práctica.

Plantear otro problema relajado al problema de la VLTE que permita crear algoritmos con mecanismos de auto corrección utilizando herramientas del cómputo distribuido a través de la topología combinatoria [25]. La topología combinatoria permite expresar todos los posibles estados de un sistema a verificar, de esta forma se puede simplificar el análisis en tiempo de ejecución.

También, a partir de la imposibilidad de la VLTE, realizar un análisis teórico de algunas herramientas de rastreo distribuido en aplicaciones de microservicios, como *Jaeger* o *OpenTracing* [28], o de plataformas de monitoreo en aplicaciones de microservicios como *Prometheus* [38].

Bibliografía

[1] Testing shared memories. *SIAM journal on computing*, 26:1208–1244, 1997.

[2] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, sep 1993.

[3] R. Alur, K. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. *Information and Computation*, 160(1):167–188, 2000.

[4] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[5] J. Aspnes and M. Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Second Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 340–349, July 1990.

[6] H. Attiya, F. Ellen, and P. Fatourou. The complexity of updating snapshot objects. *Journal of Parallel and Distributed Computing*, 71(12):1570–1577, 2011.

[7] H. Attiya and O. Rachman. Atomic snapshots in o (n log n) operations. *SIAM Journal on Computing*, 27(2):319–340, 1998.

[8] H. Attiya and S. Rajsbaum. Indistinguishability. *Commun. ACM*, 63(5):90–99, apr 2020.

[9] H. Attiya and J. Welch. *Fault-Tolerant Consensus*, chapter 5, pages 91–124. John Wiley Sons, Ltd, 2004.

[10] B. Bonakdarpour, P. Fraigniaud, S. Rajsbaum, and C. Travers. Challenges in fault-tolerant distributed runtime verification. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, pages 363–370, Cham, 2016. Springer International Publishing.

[11] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Verifying concurrent programs against sequential specifications. In M. Felleisen and P. Gardner, editors, *Programming Languages and Systems*, pages 290–309, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[12] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. On reducing linearizability to state reachability. *Information and Computation*, 261:383–400, 2018. ICALP 2015.

[13] B. Brown. Facebook's catastrophic blackout could cost $90 million in lost revenue, 2019.

[14] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, feb 1985.

[15] S. Doherty, D. L. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and G. L. Steele. Dcas is not a silver bullet for nonblocking algorithm design. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, page 216–224, New York, NY, USA, 2004. Association for Computing Machinery.

[16] B. Dongol and J. Derrick. Verifying linearizability: A comparative survey. *CoRR*, abs/1410.6268, 2014.

[17] C. Drăgoi, T. A. Henzinger, and D. Zufferey. Psync: A partially synchronous language for fault-tolerant distributed algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 400–415, New York, NY, USA, 2016. Association for Computing Machinery.

[18] A. El-Hokayem and y. Falcone. Can We Monitor All Multithreaded Programs? In *RV 2018 - 18th International Conference on Runtime Verification*, pages 1–24, Limassol, Cyprus, Nov. 2018.

[19] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, apr 1985.

[20] P. Fraigniaud, B. Bonakdarpour, S. Rajsbaum, D. Rosenblueth, and C. Travers. Decentralized Asynchronous Crash-Resilient Runtime Verification. In *27th International Conference on Concurrency Theory (CONCUR)*, Québec, Canada, 2016.

[21] P. Fraigniaud, S. Rajsbaum, and C. Travers. On the number of opinions needed for fault-tolerant run-time monitoring in distributed systems. In B. Bonakdarpour and S. A. Smolka, editors, *Runtime Verification*, pages 92–107, Cham, 2014. Springer International Publishing.

[22] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, jun 2002.

[23] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[24] M. Herlihy. Blockchains from a distributed computing perspective. *Commun. ACM*, 62(2):78–85, jan 2019.

[25] M. Herlihy, D. Kozlov, and S. Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann Publishers Inc., Boston, USA, 2014.

[26] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[27] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, jul 1990.

[28] J. Höglund. *An nalysis of a distributed tracing systems effect on performance Jaeger and OpenTracing API*. PhD thesis, 2020.

[29] M. Inoue, T. Masuzawa, W. Chen, and N. Tokura. Linear-time snapshot using multi-writer multi-reader registers. In G. Tel and P. Vitányi, editors, *Distributed Algorithms*, pages 130–140, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[30] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, jul 1978.

[31] M. Leucker and C. Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

[32] C. McCaffrey. The verification of a distributed system. *Commun. ACM*, 59(2):52–55, jan 2016.

[33] M. Mostafa and B. Bonakdarpour. Decentralized runtime verification of ltl specifications in distributed systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 494–503, 2015.

[34] B. Patt-Shamir and S. Rajsbaum. A theory of clock synchronization (extended abstract). In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '94, page 810–819, New York, NY, USA, 1994. Association for Computing Machinery.

[35] T. Scheffel and M. Schmitz. Three-valued asynchronous distributed runtime verification. In *2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 52–61, 2014.

[36] N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, mar 2011.

[37] M. Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

[38] N. Sukhija, E. Bautista, O. James, D. Gens, S. Deng, Y. Lam, T. Quan, and B. Lalli. Event management and monitoring framework for hpc environments using servicenow and prometheus. In *Proceedings of the 12th International Conference on Management of Digital EcoSystems*, MEDES '20, page 149–156, New York, NY, USA, 2020. Association for Computing Machinery.

[39] J. von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.