The LINEARIZABILITY HIERARCHY
(From sequentiality to concurrency)

Michel RAYNAL

Univ Rennes (IRISA, CNRS, Inria) France
Table of Contents

From sequential to concurrent specifications

- At the very beginning (the sixties)
- Linearizability (1986, 1991)
- Set-linearizability (1994)
- Interval-linearizability (2018)
- Underlying theory (2018)
At the very beginning
From structured programming to objects

Once upon a time... sequential computing

- Simula: an algol-based simulation language.
  by O.-J. Dahl and K. Nygaard
  *Communications of the ACM, 9(9):671-678 (1966)*

- Go To statement considered harmful.
  by E.W. Dijkstra

Structured programming.
*Academic Press, 220 pages (1972)*

- Proof of correctness of data representation.
  by C.A.R. Hoare
  *Acta Informatica, 1:271-281 (1972)*

- Nondeterminacy and formal derivation of programs.
  E.W. Dijkstra
  *Communications of the ACM, 18(8):453-457 (1975)*

- Programming: sorcery or science?
  by C.A.R. Hoare

- **Pre/post conditions** (Hoare's logic)
  Pre-condition `{ statement }` Post-condition
- **Weakest pre-condition, Predicate transformer** (EWD)
Once upon a time... the advent of concurrency

- Solution of a problem in concurrent programming control. 
  E.W. Dijkstra  
  *Communications of the ACM*, 8(9):569 (1965)

- Cooperating sequential processes. 
  E.W. Dijkstra  
  *Programming Languages (Genuys Ed.)*, Academic Press, pp. 43-112 (1968)

- Monitors: an operating system structuring concept. 
  C.A.R. Hoare  

Basically reduces concurrency to sequentiality (mutex)

Mastering concurrent computing through sequential thinking. 
S. Rajsbaum & M. Raynal  
*Communications of the ACM*, 83(1):78-87 (2020)  
(explorers the deep continuity from mutex to consensus)
Where is the problem?

- A sequential execution of a queue object

- A concurrent execution of a queue object
On the definition of time: citations

Time is what makes that all does not arrive at the same time.

Time is what is measured by clocks.
What is a specification?

- Asynchronous processes, crash failures

- **Sequential object:**
  all the traces of object operations capturing all the correct behaviors

- **Concurrent objects:**
  Description of all the traces of object operations capturing all the correct behaviors

Partial orders, How to break atomicity (at most one operation at a point of the time line? why to break it? etc.)

(BTW, A question is only the formatting of its answer!)
Concurrency: What is a **consistency condition** (1)?

- Define the (limits on the) way concurrency is allowed to impact an execution
- (Always respect process order)
Let us consider a concurrent run $R$ involving an object $O$ defined by a specification (e.g. a seq. spec.)

A consistency condition is a mapping from the operations on the object produced by the run $R$ to the specification of the object.

- If (for example) the specification is sequential the consistency condition must produce a trace belonging to the specification.
- If no such mapping can be produced, the run does not satisfy the consistency condition.

Linearizability, sequential consistency, serializability, ..., are consistency conditions.
A guided visit to the linearizability hierarchy

- Linearizable objects
- Set-linearizable objects
- Interval-linearizable objects
Linearizability
Atomicity, Linearizability, etc.

The masters of time (concurrency)

To synchronize or not to synchronize, that is the question and what to synchronize?
Basic articles

• **Solution of a problem in concurrent programming control**  
  E.W. Dijkstra  
  *Communications of the ACM*, 8(9):569 (1965)  
  First article on concurrency

• **On interprocess communication, Part I: basic formalism, Part II: algorithms**  
  L. Lamport  
  This article analyzes the nature of what is atomic, and what is not

• **Linearizability: a correctness condition for concurrent objects**  
  M.P. Herlihy and J.M. and Wing J.M.  
  *ACM Transactions on Progr. Languages and Systems*, 12(3):463-492 (1990)  
  This article introduced linearizability and its properties

Object operations vs events

Asynchronous processes, crash failures

Physical (or logical time) line of an external omniscient observer

© The linearizability hierarchy
Linearizability: definition

From sequential specifications to concurrent executions

- Linearizability considers objects defined by a sequential specification on total operations
- An execution of an object is linearizable if it is possible to totally order all the operations on the object in such a way that this order respects real-time order

(if an operation on the object $op_1$ terminated before an operation $op_2$ started, $op_1$ appears before $op_2$ in the total order)

Remarks:
- total operation: always returns a result
- always respects process order

© The linearizability hierarchy
An object $SN$ containing pairs with two operations

- $SN$.write($v$): adds the pair $\langle i, v \rangle$ to $SN$ and suppress the previous pair $\langle i, - \rangle \in SN$ if any

- $SN$.snapshot(): returns the “current” set of pairs

 Linearizability example: snapshot object (2)

- Internally (implementation): concurrency
- Externally (spec. for users): sequentiality
Fundamental properties of linearizability

- **Non-blocking:**
  To complete an object operation does not need to wait for another to terminate

- **Composability:**
  Linearizable objects compose for free
Composability: example

Module $I_1$

$Q_1$.deq()

$Q_1$.enq()

Module $I_2$

$Q_2$.enq()

$Q_2$.deq()

Queue $Q_1$

Queue $Q_2$

$Q$.enq1()

$Q$.deq1()

$Q$.enq2()

$Q$.deq2()

Module $I$ implementing the object $Q$
Sequential consistency

- How to make a multiprocessor computer that correctly executes multiprocess programs. L. Lamport
  *IEEE Transactions on Computers, C28(9):690-691 (1979)*

- Definition:
  an execution of an object is sequentially consistent if it is possible to totally order all the operations on the object while respecting each process order.

- The “witness” total order is “physical” in linearizability and “logical” only in sequential consistency.

- Seq. consistent objects do not compose for free!
Example of sequential consistency

\[ Q . \text{enq}(a) \quad Q' . \text{enq}(b') \quad Q' . \text{deq}() \rightarrow b' \]

\[ Q' . \text{enq}(a') \quad Q . \text{enq}(b) \quad Q . \text{deq}() \rightarrow b \]
A lot of works relaxing linearizability

Many++ works investigated weakening of linearizability
Relaxing linearizability: a few examples

- **A scalable lock-free stack algorithm**
  D. Hendler, N. Shavit and L. Yerushalmi

- **Quasi-linearizability: relaxed consistency for improved concurrency**
  Afek Y., Korland G., and Yanovsky E.
  
  Idea: Each run is at a bounded distance of a linearizable run

- **Data structures in the multicore age**
  Shavit N.,
  *Communications of the ACM*, 54(3):76-84 (2011)

- **Local linearizability for concurrent container-type data structures**
  
  Introduced the notion of **container object** (RW is not a container)
Relaxing linearizability: a few examples: Cont’d

- The computability of relaxed data structures: queues and stacks as examples
  Shavit N. and Taubenfeld G.,

- Distributionally linearizable data structures
  Alistarh D., Brown T., Kopinsky J., Li J. and Nadiradze G.,

- Intermediate value linearizability: a quantitative correctness condition
  Rinberg A. and Keidar I.,
  *Proc. 34th DISC*, LIPICs 179, 17 pages (2020)

- Relaxed queues and stacks from read/write operations
  A. Castañeda, S. Rajsbaum, M. Raynal.
  *Proc. 24th OPODIS*, LIPICs 184, 19 pages (2020)

- Upper and lower bounds for deterministic approximate objects
  Hendler D., Khattabi A., Milani A., and Travers C.,
  *Proc. 41st IEEE ICDCS*, LIPICs, pp. 438-448 (2021)
Set-linearizability
• Introduced by Gil Neiger:
  Set linearizability.
  *Proc. 13th ACM symposium on Principles of distributed computing (PODC’94),
  Brief announcement, ACM Press, page 396 (1994)*

• Later investigated in:

  * Hemed N., Rinetzky N., and Vafeiadis V.,
    Modular verification of concurrency-aware linearizability.

  * Castañoed a A., Rajsbaum S., and Raynal M.,
    Unifying concurrent objects and distributed tasks: interval-linearizability.
    *Journal of the ACM*, 65(6), Article 45, 42 pages (2018)
Why set-linearizability?

• Motivation example: $k$-set agreement object
  ★ Each process proposes a value and decides a value
  ★ a decided value is a proposed value
  ★ at most $k$ different values are decided

• Linearizability:
  ★ cannot capture the full generality of $k$-set agreement (and many other objects)
  ★ Due to its very definition: restricted to seq. spec.

• need to free from the “burden of the (seq.) past”
What does set-linearizability add

<table>
<thead>
<tr>
<th>Linearizability</th>
<th>Set-linearizability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomicity</td>
<td>Atomicity + simultaneity</td>
</tr>
<tr>
<td>User level: specification</td>
<td>Sequential</td>
</tr>
<tr>
<td>Implementation level</td>
<td>FT + Concurrent</td>
</tr>
</tbody>
</table>

- Due to its very definition: linearizability $\leftrightarrow$ seq. spec.

- Set-linearizability
  - allows to capture simultaneity of operations
  - captures the notion of point contention

- Suited to a class of concurrent object specification

\[
\text{Set-lin} = \text{linearizability} + \text{simultaneity}
\]
Set-lin example: Immediate snapshot object


- A **snapshot** object with **concurrent** specification

- A single operation denoted \( \text{im\_snapshot}(v) \)

- When a process \( p_i \) invokes \( \text{im\_snapshot}(v_i) \)
  - it **deposits** the pair \( \langle i, v_i \rangle \) in the object
  - and **returns** a set of pairs denoted \( \text{view}_i \)
Set-LIN: Immediate snapshot object

- **Termination.** If a process invokes `im_snapshot()` and does not crash, its invocation terminates.

- **Self-inclusion.**
  \[
  im\_snapshot(v_i) \text{ returns } view_i \text{ to } p_i \Rightarrow (\langle i, v_i \rangle \in view_i)
  \]

- **Global inclusion (Containment).**
  - Invocation of `im_snapshot(v_i)` by \( p_i \) returns \( view_i \) and invocation of `im_snapshot(v_j)` by \( p_j \) returns \( view_j \) ⇒ \( \text{view}_i \subseteq \text{view}_j \) or \( \text{view}_j \subseteq \text{view}_i \)

- **Immediacy.**
  \[
  (\langle i, v_i \rangle \in \text{view}_j) \land (\langle j, v_j \rangle \in \text{view}_i) \Rightarrow (\text{view}_i = \text{view}_j)
  \]

Immediacy ⇒ simultaneity

© The linearizability hierarchy
Set-lin: immediate snapshot algorithm

Shared registers:

- $MEM[1..n]$ init to $[\bot, \ldots, \bot]$
- $LEVEL[1..n]$ init to $[(n+1), \ldots, (n+1)]$

operation im_snapshot($v$)is

% code for process $p_i$

1. $MEM[i] \leftarrow v$
2. repeat $LEVEL[i] \leftarrow LEVEL[i] - 1$
   (L3) for each $j \in \{1, \ldots, n\}$ do $level_i[j] \leftarrow LEVEL[j]$ end for;
   set $i \leftarrow \{x \mid level_i[x] \leq level_i[i]\}$
3. until $(|set_i| \geq level_i[i])$ end repeat;
   (L6) let $view_i = \{ \langle x, MEM[x]\rangle \mid x \in set_i \}$;
4. return($view_i$)
end operation.
Immediate snapshot example of an execution

A possible run of the previous algorithm

\[ IS.\text{im}_\text{snapshot}(1) \rightarrow \{\langle a, 1 \rangle\} \]

\[ IS.\text{im}_\text{snapshot}(b) \rightarrow \{\langle 2, b \rangle, \langle 1, a \rangle, \langle 3, c \rangle\} \]

\[ IS.\text{im}_\text{snapshot}(c) \rightarrow \{\langle 2, b \rangle, \langle 3, c \rangle, \langle 1, a \rangle\} \]

value of immediate-snapshot after the linearization points

\{\langle 1, a \rangle\}

\{\langle 1, a \rangle, \langle 3, c \rangle, \langle 2, b \rangle\}

External observer’s time line
Interval-linearizability
What does interval linearizability add

<table>
<thead>
<tr>
<th>Consist. cond.</th>
<th>Specification</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linearizability</td>
<td>Sequentiality</td>
<td>Concurrent</td>
</tr>
<tr>
<td>Set Lin.</td>
<td>Lin + simultaneity</td>
<td>Concurrent</td>
</tr>
<tr>
<td>Interval Lin.</td>
<td>Set Lin + time ubiquity</td>
<td>Concurrent</td>
</tr>
</tbody>
</table>

Int-lin: write-snapshot object (definition)

- Its is a snapshot object in which the two operations `write()` and `snapshot()` are pieced together into a single operation denoted `write_snapshot()`

- Properties:
  - Self-inclusion: \( (i, v_i) \in view_i \)
  - Containment: \( view_i \subseteq view_j \) or \( view_j \subseteq view_i \)

Reminder: Self-inclusion is not a property required by the base snapshot object (operations `write()` and `snapshot()`).
One-shot write-snapshot object: algorithm

operation write_snapshot(v) is

\[ \text{MEM}_{i} \leftarrow \langle i, v \rangle; \]
\[ \text{new}_i \leftarrow \bigcup_{1 \leq j \leq n} \{ \langle j, \text{MEM}[j] \rangle \text{ such that } \text{MEM}[j] \neq \bot \}; \]

repeat

\[ \text{old}_i \leftarrow \text{new}_i; \]
\[ \text{new}_i \leftarrow \bigcup_{1 \leq j \leq n} \{ \text{MEM}[j] \text{ such that } \text{MEM}[j] \neq \bot \} \]

until \( \text{old}_i = \text{new}_i \) end repeat;

return(\text{new}_i)

end operation
Write-snapshot: example of an execution

A possible run of the previous algorithm

\[ WS.write\_snapshot(a) \rightarrow \{(2, b), (1, a)\} \]

\[ WS.write\_snapshot(b) \rightarrow \{(2, b), (1, a), (3, c)\} \]

\[ WS.write\_snapshot(c) \rightarrow \{(2, b), (3, c), (1, a)\} \]

value of write-snapshot after the linearization points

\{⟨1, a⟩, ⟨2, b⟩\} \quad \{⟨1, a⟩, ⟨3, c⟩, ⟨2, b⟩\}

External observer’s time line

© The linearizability hierarchy
Interval linearizability: another example

Lattice agreement

• A set $L$ partially ordered by a binary relation $\sqsubseteq$ s. t. any pair $x, y \in L$ has a least upper bound called $\text{join}$

• A one-shot operation $\text{propose}(v)$
  with input $v \in L$, returns a value $v' \in L$, such that:

  * **Validity**: $v'$ is a join of some proposed values including $v$ and all values returned by previous $\text{propose()}$ operations
  * **Consistency**: returned values are ordered by $\sqsubseteq$

Used in distributed state reconciliation:
Many objects are defined by distributed tasks

Distributed tasks: no notion of “order” on operation execution

Vector of distributed inputs

\[ I = [in_1, \ldots, in_n] \]

Communicating processes

\[ p_1, \ldots, p_n \]

Vector of distributed outputs

\[ O = [out_1, \ldots, out_n] \]

Task \( T: O = \Delta(I) \)
Two important theorems

- Concurrent specifications: beyond linearizability
  Goubault E., Ledent J., and Mimram S.,
  22nd OPODIS, LIPIcs 125, 16 pages (2018)

  Theorem:
  
  Every concurrent specification is interval-linearizable

- Unifying concurrent objects and distributed tasks: interval-linearizability
  Castañeda A., A., Rajsbaum S., and Raynal M.,

  Theorem:
  
  interval-linearizable objects and (refined) tasks have
  the same expressive power (both are complete in the
  sense they are able to specify any prefix-closed set of
  well-formed executions)
On progress conditions

On the progress in the presence of failures
(Net effect of asynchrony and failures: mutex is irrelevant)

• 1991: Wait-freedom: If a process does crash (while executing an object operation) it terminates
• 1990: Non-blocking $\sim$ no deadlock
• 2005: Obstruction-freedom: if a process executes alone during a long enough period (and does not crash) it terminates its operation

(All these properties are due to M. Herlihy and co-authors)
On the interplay between safety and liveness

Queue in the consensus number (CN) 1 and 2 worlds

i.e., with the help of the weakest computability/synchronization power in the presence of asynchrony and crashes

Additional results

This execution fragment is linearizable.
This execution fragment is set-linearizable
(See work-stealing for idempotent jobs)
Additional results

- $\text{enq}(a)$
- $\text{enq}(c)$
- $\text{enq}(b)$
- $\text{deq}() \rightarrow a$
- $\text{deq}() \rightarrow b$
- $\text{deq}() \rightarrow c$

Physical time line:
- $b$
- $b, a$
- $a$
- $a, c$
- $c$

External observer's

© The linearizability hierarchy
<table>
<thead>
<tr>
<th>Base object</th>
<th>Liveness</th>
<th>Safety</th>
</tr>
</thead>
</table>
| CN = 1      | enqueue(): wait-freedom  
dequeue(): non-blocking | enqueue(): linearizability  
dequeue(): set-linearizability |
| CN = 1      | enqueue(): wait-freedom  
dequeue(): wait-freedom | enqueue(): linearizability  
dequeue(): interval-linearizability |
| CN = 2      | enqueue(): wait-freedom  
dequeue(): non-blocking | enqueue(): linearizability  
dequeue(): linearizability |
| CN = 2      | enqueue(): wait-freedom  
dequeue(): wait-freedom | enqueue(): linearizability  
dequeue(): interval-linearizability |
Additional results cont’d

Stack in the consensus number (CN) 1 and 2 worlds

<table>
<thead>
<tr>
<th>Base object</th>
<th>Liveness</th>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>CN = 1</td>
<td>push(): wait-freedom&lt;br&gt;pop(): wait-freedom</td>
<td>push(): linearizability&lt;br&gt;pop(): set-linearizability</td>
</tr>
<tr>
<td>CN = 2</td>
<td>push(): wait-freedom&lt;br&gt;pop(): wait-freedom</td>
<td>push(): linearizability&lt;br&gt;pop(): linearizability</td>
</tr>
</tbody>
</table>
### THE GLOBAL PICTURE

<table>
<thead>
<tr>
<th>Consistency condition</th>
<th>User layer specification</th>
<th>Implementation layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linearizability</td>
<td>Sequential</td>
<td>FT + Concurrent</td>
</tr>
<tr>
<td>Set-linearizability</td>
<td>concurrent: <strong>simultaneity</strong></td>
<td>FT + Concurrent</td>
</tr>
<tr>
<td>Int-linearizability</td>
<td>concurrent: <strong>time-ubiquity</strong></td>
<td>FT + Concurrent</td>
</tr>
</tbody>
</table>

As Lin, Set lin and Int lin are composable for free!
A look at the underlying theory
Two articles

• Introduced in:

Unifying concurrent objects and distributed tasks: interval-linearizability
Castañeda A., A., Rajsbaum S., and Raynal M.,

• Analyzed in:

Concurrent specifications: beyond linearizability
Goubault E., Ledent J., and Mimram S.,
22nd OPODIS, LIPIcs 125, 16 pages (2018)
Notations (1)

- \( n \) processes \( p_1, \ldots, p_n \)
- \( \mathcal{V} \): values (integers) exchanged by the processes
  - \( \text{inv}^x_i \): invocations of the object by \( p_i \) with input \( x \)
  - \( \text{resp}^y_j \): responses of the object to \( p_j \) with output \( y \)
- \( \mathcal{A} \): set of all the actions (events) on the object
- execution trace: finite seq of actions (events)
- \( \mathcal{T} = \mathcal{A}^* \): set of possible traces
- \( \epsilon \): empty trace
- \( T \cdot T' \): trace concatenation
• $\pi_i(T)$ trace obtained by removing all the actions of the processes $p_j \neq p_i$

• Alternating trace: $\pi_i(T)$ is empty or alternates between invocations and responses

• If $\pi_i(T)$ terminates with an invocation: pending inv.

• Complete trace: no pending invocation
Definition

A concurrent specification $\Sigma$ is a subset of $\mathcal{T}$ satisfying the following eight properties

- **Alternating**: every $T \in \Sigma$ is alternating
- **Prefix-closed**: if $T \cdot T' \in \Sigma$ then $T \in \Sigma$
- **non-empty**: $\epsilon \in \Sigma$
- **receptive**: if $T \in \Sigma$ and $p_i$ has no pending invocation, then $T \cdot \text{inv}_{x}^{x} \in \Sigma$ for any $x$
• **Total:**
  if $T \in \Sigma$ and $p_i$ has a pending invocation, then it exists $x \in \mathcal{V}$ such that $T \cdot \text{resp}_i^x \in \Sigma$

• **Commuting invocations:**
  if $T \cdot \text{inv}_i^x \cdot \text{inv}_j^y \cdot T' \in \sigma$ then $T \cdot \text{inv}_j^y \cdot \text{inv}_i^x \cdot T' \in \sigma$

• **Commuting responses:**
  if $T \cdot \text{resp}_i^x \cdot \text{resp}_j^y \cdot T' \in \sigma$ then $T \cdot \text{resp}_j^y \cdot \text{resp}_i^x \cdot T' \in \sigma$

• **Closure under expansions:**
  if $T \cdot \text{resp}_j^y \cdot \text{inv}_i^x \cdot T' \in \sigma$ then $T \cdot \text{inv}_i^x \cdot \text{resp}_j^y \cdot T' \in \sigma$
Meaning of “an algo implements an conc. object”

- Consider an automaton-based representation of a prog. language
- Decision function $\delta()$ : defines which object the program will call
- Transition function $\tau()$ : defines the next state of the object
- An algorithm $A$ (concurrent program) is defined by a set of automata $A_i$, each one associated with a process $p_i$
- $A$ implements a concurrent specification $\Sigma$ if all the traces it generates belong to $\Sigma$
Two important theorems

• Concurrent specifications: beyond linearizability
  Goubault E., Ledent J., and Mimram S.,
  22nd OPODIS, LIPIcs 125, 16 pages (2018)

  Theorem:
  Every concurrent specification is interval-linearizable

• Unifying concurrent objects and distributed tasks: interval-linearizability
  Castañeda A., A., Rajsbaum S., and Raynal M.,

  Theorem:
  interval-linearizable objects and (refined) tasks have
  the same expressive power and both are complete in
  the sense that they are able to specify any prefix-
  closed set of well-formed executions
Conclusion
A visit to

• Concurrent objects
• Specification of concurrent objects
• Linearizability hierarchy

Important:
Int-LIN ⇒ Composability (for free) of concurrent objects
Is there and to the story?

Colorín colorado,
est cuento NO se ha acabado...