An Introduction to the Implementation of Concurrent Objects

Michel RAYNAL
raynal@irisa.fr

Institut Universitaire de France
IRISA, Université de Rennes, France

Summary

- Concurrent objects
- Safety: Linearizability vs sequential consistency
- Lock-based implementations
- Mutex-free implementations
- Liveness: Progress conditions
- Hybrid implementations
- Conclusion

Source

- The content of these slides are from chapters 2, 5 and 6 of the book (composed of 17 chapters)

Concurrent Programming: Algorithms, Principles and Foundations,
Computation Model

- A set $\Pi$ of $n$ asynchronous processes $p_1, \ldots, p_n$
- A shared memory made up of atomic read/write registers
- Failure model: process crash model
  - Terminology:
    - Correct = a process that never crashes
    - Faulty = a process that crashes
  - $t =$ upper bound on the nb of faulty processes
  - Failure-free: $t = 0$
  - Wait-free model $t = n - 1$
  - $t$-resilient model: $1 \leq t < n$

Concurrent Object

An object accessed by \textit{concurrent} processes

\[ p_1 \quad \ldots \quad p_i \quad \ldots \quad p_n \]

Non seq. specification: the \textbf{NBAC} example

Each process is assumed to vote \textit{yes} or \textit{no}

- \textbf{Termination}. A process that does not crash decides
- \textbf{Agreement}. No two processes decide differently
- \textbf{Validity}. A decided value is \textit{abort} or \textit{commit}
  - \textbf{Justification}. \textit{commit} decided $\Rightarrow$ all processes have voted \textit{yes}
  - \textbf{Obligation}. No process crashes and all processes vote \textit{yes} $\Rightarrow \textit{commit}$ is decided
Object considered here

- Sequential specification
- With *total operations*:

  An operation can always return a result (no blocking imposed by the spec)
  
  * E.g., `pop()` on an empty stack returns `empty`
  * E.g., `enqueue()` on a full bounded queue returns `full`

Hence, (if any) blocking is due to the implementation, not to the spec

Sequential vs Concurrent (1)

**SEQUENTIAL:**

```
Enq (a) Enq (c) Enq (b) Deq (a) Deq (c)
```

**CONCURRENT:**

```
P1  Enq (a) Enq (b) Deq (a|b|c) ?

P2    Enq (c) Deq (a|b|c) ?
```

Sequential vs Concurrent (2)

```
P1  Enq (a) Enq (b) Deq (a|b|c) ?

P2  Enq (c) Deq (a|b|c) ?
```

This “history” belongs to the sequential specification

Sequential vs Concurrent (3)

```
P1  Enq (a) Enq (b) Deq (a|b|c) ?

P2  Enq (c) Deq (a|b|c) ?
```

This “history” belongs to the sequential specification
Part II

On the SAFETY side:
Consistency conditions

The aim is here to answer the question:
what is a correct execution involving a set of objects?

Atomicity vs Linearizability

• Atomicity first introduced for read/write registers
  • Lamport L., On interprocess communication, Part I: basic formalism, Distributed Computing, 1(2):77-85, 1986
  • Lamport L., On interprocess communication, Part II: algorithms, Distributed Computing, 1(2):77-101, 1986
• Linearizability extends Atomicity to any object with a sequential specification
• Hence, Atomicity and Linearizability can be considered as synonymous

Linearizability

• a history is linearizable if
  • each operation appears as if it has been executed instantaneously at some point of the time line between its start event and its end event
  • no two operations appear at the same point of the time line
  • the corresponding sequence belongs to the specification of the objects


Another consistency condition: seq consistency

• Similar to Linearizability without requiring agreement with real time

\[ p_1 \xrightarrow{Q, eq(a)} \]

\[ p_2 \xrightarrow{Q, eq(i) \rightarrow Q, deq()} \]

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

Seq consistency is more interesting in message-passing systems
The fundamental difference: composability

- Locality property: A property $P$ is **local** if a set of objects as a whole satisfies $P$ whenever each object satisfies $P$.
- Locality = modularity
  
  independent implementations compose for free
- Linearizability is a local property
- Sequential consistency is a not local property

Seq consistency is not a local property

- $p_1$:
  - $Q,\text{enq}(a)$
  - $Q',\text{enq}(b')$  
  - $Q',\text{deq}() \rightarrow b'$

- $p_2$:
  - $Q',\text{enq}(a')$
  - $Q,\text{enq}(b)$
  - $Q,\text{deq}() \rightarrow b$

The benefit of linearizability

- $Q,\text{enq}()$  
  - $Q,\text{deq}()$

- $Q,\text{enq}2()$  
  - $Q,\text{deq}2()$

- $Q_1,\text{enq}()$  
  - $Q_1,\text{deq}()$

- $Q_2,\text{enq}()$  
  - $Q_2,\text{deq}()$

- $Q_\text{enq}$  
  - $Q_\text{deq}$

- $\text{Module I1}$  
  - $\text{Module I2}$  
  - $\text{Queue Q2}$

- Module I implementing the object $Q$

Part III

Lock-based Implementations
Classical approaches

- Lock = Mutual exclusion
- Lock from read/write registers
- Low level locks: Semaphores
- Imperative language: monitors (Hoare, Brinch Hansen)
- Declarative language: path expressions (Campbell)

From deadlock-free lock to starvation-free lock

Such a construction is based on

- An SWMR array $FLAG[1..n]$ with an entry per process (init to $[\text{down}, \ldots, \text{down}]$)
- A MWMR register $TURN$ which contains a proc identity
- A deadlock-free lock $DF\ _LOCK$ (e.g., Lamport's fast mutex algorithm)


On the liveness side: liveness conditions

- **Deadlock-freedom:**
  At least one operation invocation always terminates
- **Starvation freedom:**
  All operation invocations terminate

The construction

```
operation acquire_SF_lock(i) is
   FLAG[i] ← up;
   wait [(TURN = i) \lor (FLAG[TURN] = down)];
   DF\_LOCK.acquire_DF\_lock(i);
   return()
end operation.

operation release_DF\_mutex(i) is
   FLAG[i] ← down;
   if (FLAG[TURN] = down)
      then TURN ← (TURN mod n) + 1
   end if;
   DF\_LOCK.release_DF\_lock(i);
   return()
end operation.
```
Reminder

From a computability point of view

- Mutex can be implemented in crash-free systems from atomic read/write registers
- \( b \)-valued atomic read/write registers can be built from safe bits
- Mutex can be implemented directly from safe registers


Part IV

Mutex-free Implementations

Drawbacks of lock-based implementations

- In a lock-based solution: one process at a time can access a given object
- Make the progress of processes depends on the ones from the others
  - Deadlock-prone
  - Cannot cope with the net effect of
    - Asynchrony
    - And failures
  - Process scheduling, swapping

Drawback due to lock granularity

Example of a double-ended queue

Operations accessing from the left side:
- left_enq()
- left_deq()

Operations accessing from the right side:
- right_enq()
- right_deq()

Left side of the queue

Right side of the queue
Mutex-free implementation

Do not use lock (implicitly or explicitly)

No code is protected by a critical section (lock)
- Peterson G.L., Concurrent reading while writing, *ACM TOPLAS*, 5:46-55, 1983

Progress (liveness) conditions

- **Obstruction-freedom** (is wrt concurrency)
- **Non-blocking** (~ deadlock-freedom)
- **Wait-freedom** (~ starvation-freedom)
  * Finite wait-freedom
  * Bounded wait-freedom

These progress conditions cope naturally with any asynchrony and crash pattern (while lock-based deadlock-freedom and starvation-freedom do not), i.e., they implicitly consider $t = n - 1$ (wait-free model)

<table>
<thead>
<tr>
<th>Lock-based implementation</th>
<th>Mutex-free implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadlock-freedom</td>
<td>Obstruction-freedom</td>
</tr>
<tr>
<td>Starvation-freedom</td>
<td>Non-blocking</td>
</tr>
<tr>
<td></td>
<td>Wait-freedom</td>
</tr>
</tbody>
</table>

A simple theorem

- Context:
  * One-shot objects
  * Bounded nb of processes
- Theorem: Non-blocking $=$ Wait-free
Boosting obstruction-freedom

- From Obstruction-freedom to non-blocking
- From Obstruction-freedom to wait-freedom
- Failure detector-based contention managers


A very simple wait-free object: the Splitter (1)

- Validity. Value returned by direction() is right, left, or stop
- Concurrent execution. If \( x \) processes invoke direction():
  * At most \( x-1 \) processes obtain the value right
  * At most \( x-1 \) processes obtain the value left
  * At most one process obtains the value stop
- Termination. Any invocation of direction() terminates

A very simple wait-free object: the Splitter (2)
A very simple wait-free object: the Splitter (3)

operation $SP.direction(i)$ is

LAST $\leftarrow i$;
if ($DOOR = closed$)
then return(right)
else ($DOOR \leftarrow closed$;
if ($LAST = i$)
then return(stop)
else return(left)
end if
end if
end operation.

A very simple wait-free object: the Splitter (4)

LAST $\leftarrow i$  DOOR $\leftarrow closed$  LAST $= i$

No process has modified LAST

Opstruction-free counter (1)

Weak timestamp generator which provides processes with a single operation denoted get_timestamp() which returns an natural integer

- Validity. No two invocations of get_timestamp() return the same value
- Consistency. Let $gt_1()$ and $gt_2()$ be two distinct invocations of get_timestamp(). If $gt_1()$ returns before $gt_2()$ starts, the timestamp returned by $gt_2()$ is greater than the one returned by $gt_1()$
- Termination. Obstruction-freedom

Obstruction-free counter (2)

- NEXT: value of the next timestamp, initialized to 1
- LAST: unbounded array of atomic registers

A process $p_i$ deposits its index $i$ in LAST[$k$] to indicate it is trying to obtain the timestamp $k$

- COMP: unbounded array of atomic Boolean registers initialized to false

A process $p_i$ sets COMP[$k$] to true to indicate that it is competing for the timestamp $k$
Obstruction-free counter (2)

operation get_timestamp(i) is
    k ← NEXT;
    repeat forever
        LAST[k] ← i;
        if (¬ COMP[k])
            then COMP[k] ← true;
                if (LAST[k] = i)
                    then NEXT ← NEXT + 1; return(k)
            end if;
    end repeat
end operation.

How do processes communicate?

Shared memory models

- Base read/write model
- Base read/write model enriched with specific operations
  - Swap (local/shared), Test&Set, Fetch&Add, etc.
  - Compare&Swap, LL/SC, etc.
  - Herlihy’s Hierarchy on the synchro power of base operations define a hierarchy of shared memory models

Compare&Swap: definition

X.compare&swap(old, new) is
    if (X = old)
        then X ← new; return(true)
    else return(false)
end if.

Using Compare&Swap

statements;
old ← X;
any sequence of statements possibly involving accesses to the shared memory;
if X.compare&swap(old, new)
    then statements S1
    else statements S2
end if;
statements.
Compare\&Swap: the ABA problem

- Initially $X = a$
- At time $\tau_1$: $p_i$ reads $a$ from $X$
- At time $\tau_2 > \tau_1$:
  - $p_j$ successfully executes $X.C&S(a, b)$ ($X = b$)
- At time $\tau_3 > \tau_2$:
  - $p_j$ successfully executes $X.C&S(b, a)$ ($X = a$)
- At time $\tau_4 > \tau_3$:
  - $p_i$ successfully executes $X.C&S(a, b)$ and erroneously believes that $X$ has not been modified by another process in the interval $[\tau_1..\tau_4]$

Solving the ABA problem

- Associate a new sequence number with every $X.C&S$
- $X$ is now a pair $(a, sn)$
- At time $\tau_1$: $p_i$ reads $(a, sn)$ from $X$
- At time $\tau_2 > \tau_1$:
  - $p_j$ successfully executes $X.C&S((a, sn), (b, sn + 1))$
- At time $\tau_3 > \tau_2$:
  - $p_k$ successfully executes $X.C&S((b, sn + 1), (a, sn + 2))$
- At time $\tau_4 > \tau_3$:
  - when $p_i$ executes $X.C&S((a, sn), (c, sn + 1))$, the write into $X$ fails and returns false to $p_i$

Non-blocking objects based on Compare\&Swap

- Non-Blocking Queue Based on Read/Write Registers and Compare\&Swap:
  - This implementation was included in the standard Java Concurrency Package
- Non-Blocking Stack Based on Compare\&Swap Regist-

A wait-free stack (1)

- Based on Fetch\&Add and Swap operations
- Uses:
  - $REG[0..\infty]$: array of atomic registers which contains the elements of the stack.
    - $REG[0]$ contains always the value $\perp$ (used only to simplify the description of the algorithm)
  - $NEXT$: atomic register containing the index of the next entry where a value can be deposited, initialized to 1

A wait-free stack (2)

operation push(v) is
  in ← NEXT.fetch&add() − 1;
  REG[in] ← v;
  return()
end operation.

operation Q.pop() is
  last ← NEXT − 1;
  for x from last to 0 do
    aux ← REG[x].swap(⊥);
    if (aux ≠ ⊥) then return(aux) end if
  end for;
  return(empty)
end operation.

Types of hybrid implementations

- **Static hybrid**
  - Some operation implementations are wait-free, other are lock-based
  - Example: a concurrent set

- **Dynamic hybrid** (context sensitive)
  - Define a notion of favorable circumstances (wrt failures, concurrency, etc.)
  - And the implementation of the operations must not use locks in favorable circumstances

Hybrid Implementations

The aim is here to design object implementations merging locks and mutex freedom

Static hybrid set

- Operations
  - *S.add(v)* adds v to the set S and returns true if v was not in the set; Otherwise it returns false
  - *S.remove(v)* suppresses v from S and returns true if v was in the set; Otherwise it returns false
  - *S.contain(v)* returns true if v ∈ S and false otherwise

- Static hybridism
  - *S.add()* and *S.remove()*: lock-based but deadlock-free
  - *S.contain()*: mutex-free and wait-free

internal Representation

- linked list pointed to by \textit{HEAD}
- A cell of the list (say \textit{NEW\_CELL}) is made up of
  \begin{itemize}
  \item \textit{NEW\_CELL.val} which contains a value (element of the set).
  \item \textit{NEW\_CELL.out}: Boolean set to true when the corresponding element is suppressed from the list
  \item \textit{NEW\_CELL.lock}: lock used to ensure mutual exclusion (when needed) on the cell
  \item \textit{NEW\_CELL.next}: pointer to the next cell.
  \end{itemize}

---

\textbf{Operation} \texttt{S.remove(v): behavior}

\begin{center}
\begin{tikzpicture}
\node [draw] (A) at (0,0) {\texttt{S1}};
\node [draw] (B) at (1,0) {\texttt{S2}};
\node [draw] (C) at (2,0) {\texttt{v \downarrow \uparrow}};
\node [draw] (D) at (3,0) {\texttt{\ldots}};
\node [draw] (E) at (4,0) {\texttt{\ldots}};
\node [draw] (F) at (5,0) {\texttt{\ldots}};

\draw [->] (A) -- (B);
\draw [->] (B) -- (C);
\draw [->] (C) -- (D);
\draw [->] (D) -- (E);
\draw [->] (E) -- (F);
\end{tikzpicture}
\end{center}

---

\textbf{Initial state}

- The set is organized as a sorted linked list
- All operation algorithms are based on list traversal

\begin{center}
\begin{tikzpicture}
\node [draw] (A) at (0,0) {\texttt{HEAD}};
\node [draw] (B) at (1,0) {\texttt{\ldots}};
\node [draw] (C) at (2,0) {\texttt{\ldots}};

\draw [->] (A) -- (B);
\draw [->] (B) -- (C);
\end{tikzpicture}
\end{center}

---

\textbf{Operation} \texttt{S.remove(v): algorithm}

\begin{verbatim}
operation S.remove(v) is
  pred ← HEAD; curr ← (HEAD \downarrow).next;
  while ((curr \downarrow).val < v)
    do pred ← curr; curr ← (curr \downarrow).next end while;
  ((pred \downarrow).lock).acquire_lock(); ((curr \downarrow).lock).acquire_lock();
  valid ← false;
  if validate(pred, curr)
    then valid ← true; pres ← ((curr \downarrow).val = v);
      if (pres) then (curr \downarrow).out ← true;
        (pred \downarrow).next ← (curr \downarrow).next
      end if
    end if;
  ((pred \downarrow).lock).release_lock(); ((curr \downarrow).lock).release_lock();
  if (valid) then return(pres) else restart the operation end if
end operation.
\end{verbatim}
Validation predicate

internal predicate validate(pred, curr) is
let res = (¬ ((pred \downarrow).out)
∧ ¬ ((curr \downarrow).out)
∧ (pred \downarrow).next = curr);
return(res)
end internal predicate.

Operation \texttt{S.add}(v): algorithm

operation \texttt{S.add}(v) is
pred ← HEAD; curr ← (HEAD \downarrow).next;
while ((curr \downarrow).val < v)
do pred ← curr; curr ← (curr \downarrow).next end while;
((pred \downarrow).lock).acquire_lock(); valid ← false;
if validate(pred, curr)
then valid ← true; to_add ← ((curr \downarrow).val ≠ v);
if (to_add) then \texttt{S.add\_new\_cell()} end if
end if;
((pred \downarrow).lock).release_lock();
if (valid) then return(to_add) else restart the operation end if
end operation.

Internal operation \texttt{S.add\_new\_cell}(): algorithm

internal operation \texttt{S.add\_new\_cell}() is
\texttt{NEW\_CELL} ← new\_cell();
\texttt{NEW\_CELL}.out ← false;
\texttt{NEW\_CELL}.val ← v;
\texttt{NEW\_CELL}.next ← curr;
\texttt{NEW\_CELL}.lock ← open;
(pred \downarrow).next ← (↑ new\_cell)
end internal operation.
Operation $S$.contain($v$): behavior

\[
\begin{array}{c}
\text{HEAD} \\
\text{pred} \quad \text{curri} \quad b \quad c \quad d \quad f \quad \Rightarrow \text{head} \\
\end{array}
\]

operation $S$.contain($v$) is
\[
curr \leftarrow \text{HEAD}; \\
\text{while } ((\text{curr}.\text{val} < v) \text{ do } curr \leftarrow (curr).\text{next} \text{ end while}; \\
\text{let } res = ((\text{curr}.\text{val} = v) \land \neg(\text{curr}.\text{out})); \\
\text{return } (res) \text{ end operation.}
\]

A dynamic hybrid consensus object

- Consensus object
  - Validity. A decided value is a proposed value
  - Agreement. No two processes decide different values
  - Termination. Any invocation of propose() terminates
  - Binary consensus: only 0 and 1 can be proposed

- Favorable circumstances: when there is no concurrency or the participating processes propose the same value

- Taubenfeld G., Contention-sensitive data structure and algorithms. Proc. 23th Intl Symposium on Distributed Computing (DISC’09), Springer Verlag, LNCS #5805, pp. 157-171, 2009

Underlying implementation objects

- PROPOSED[0..1], which is an array of two Boolean registers, both initialized to false. The atomic register $PROPOSED[v]$ is set to true to indicate that a process has proposed value $v$.
- DECIDED: atomic register whose domain is $\{\bot, 0, 1\}$, initialized to $\bot$, it eventually contains the value that is decided (and never the value which is not decided)
- AUX: atomic register whose domain and initial value are the same as for DECIDED
- LOCK: starvation-free lock
Dynamic hybrid implementation of binary consensus

operation \( C . \text{propose}(v) \) is
\[
\begin{align*}
\text{PROPOSED}[v] & \leftarrow \text{true}; \\
\text{if } (& \text{AUX } = \bot) \text{ then AUX } & \leftarrow v \text{ end if;}
\end{align*}
\]
\[
\begin{align*}
\text{if } & (\neg \text{PROPOSED}[1 - v]) \\
\text{then DECIDED } & \leftarrow v \\
\text{else if } & (\text{DECIDED } = \bot) \\
\text{then LOCK.acquire_lock();} \\
\text{if } & (\text{DECIDED } = \bot) \\
\text{then DECIDED } & \leftarrow \text{AUX} \\
\end{align*}
\]
\text{end if;}
\text{LOCK.release_lock();}
\text{return(DECIDED)}
\text{end operation.}

Abortable objects

Concurrency abortable object

- Any invocation of an object operation
  - Returns after a bounded number of steps (shared memory accesses) and
  - is allowed to return the default value \( \bot \) in presence of concurrency (then the object has not been modified)
- Can be generalized: An operation is allowed to return \( \bot \) only in “unfavorable circumstances”

Illustrating space-time diagram
A non-blocking abortable bounded stack (1)

- The stack is of size \( k \)
- Operation \( \text{push}(v) \)
  - \( \star \) returns \textit{full} if the stack is \textit{full}, otherwise
  - \( \star \) adds \( v \) to the top of the stack and returns \textit{done}
- Operation \( \text{pop}() \)
  - \( \star \) returns \textit{empty} if the stack is empty, otherwise
  - \( \star \) suppresses the value from the top of the stack and
    returns it

Stack representation (1)

- An array \( \text{STACK}[0..k] \) of atomic registers
- \( \forall x : 0 \leq x \leq k : \text{STACK}[x] \) has two fields
  - \( \star \) \( \text{STACK}[x].\text{val} \) contains a value
  - \( \star \) \( \text{STACK}[x].\text{sn} \) contains a seq number (used to prevent
    the ABA problem on this register)
    - It counts the nb of successful writes on \( \text{STACK}[x] \)
- \( \forall x : 1 \leq x \leq k : \text{STACK}[x] \) initialized to \( \langle \bot, 0 \rangle \)
- \( \text{STACK}[0] \) always stores a dummy entry (init to \( \langle \bot, -1 \rangle \))

In presence of concurrency

- Operation invocations may return \( \bot \) (abortable object)
- But at least one returns a non-\( \bot \) value (non-blocking)

Stack representation (2)

- A register \( \text{TOP} \) that contains the index of the top of the stack plus the corresponding pair \( \langle v, sn \rangle \)
- \( \text{TOP} \) initialized to \( \langle 0, \bot, 0 \rangle \)
- Both \( \text{STACK}[x] \) and \( \text{TOP} \) are modified with Compare&Swap
Principle: laziness + helping mechanism

- A push or pop operation
  * updates TOP, and
  * leaves to the next operation the corresponding update of the stack
  Hence it helps the previous (push or pop) operation by modifying the stack accordingly

Shafiei N.,
Non-blocking Array-based Algorithms for Stacks and Queues,
Proc. 13th Int'l Conference on Distributed Computing and Networking (ICDCN’09),
Springer Verlag LNCS #5408, pp. 55-66, 2009

Abortable push: weak_push()

operation weak_push(v):
  (index, value, seqnb) ← TOP;
  help(index, value, seqnb);
  if (index = k) then return(full) end if;
  sn_of_next ← STACK[index + 1].sn;
  newtop ← (index + 1, v, sn_of_next + 1);
  if TOP.C&S((index, value, seqnb), newtop)
     then return(done) else return(⊥) end if.

Abortable stack: help procedure

procedure help(index, value, seqnb):
  stacktop ← STACK[index].val;
  STACK[index].C&S((stacktop, seqnb - 1), (value, seqnb)).

Abortable pop: weak_pop()

operation weak_pop():
  (index, value, seqnb) ← TOP;
  help(index, value, seqnb);
  if (index = 0) then return(empty) end if;
  belowtop ← STACK[index - 1];
  newtop ← (index - 1, belowtop.val, belowtop.sn + 1);
  if TOP.C&S((index, value, seqnb), newtop)
     then return(value) else return(⊥) end if.
From an abortable to a non-blocking stack

operation non_blocking_push(v):
    repeat res ← weak_push(v) until res ≠ ⊥ end repeat;
    return(res).

operation non_blocking_pop():
    repeat res ← weak_pop() until res ≠ ⊥ end repeat;
    return(res).

From Non-blocking abortable to Starvation-freedom (1)

- Object operations: denoted $ABO.ab\_oper(par)$
- $CONTESTION$: atomic Boolean read/write register, initialized to false.
  Used to indicate that there is a process that has acquired the lock and is invoking $ABO.ab\_oper()$
- $LOCK$: a starvation-free lock

From Non-blocking abortable to Starvation-freedom (2)

operation oper(par) is
    if (~$CONTESTION$)
        then res ← $ABO.ab\_oper(par)$;
            if (res ≠ ⊥) then return(res) end if
    end if;
    $LOCK.acquire\_SF\_lock();$
    $CONTESTION ← true;$
    repeat res ← $ABO.ab\_oper(par)$ until res ≠ ⊥ end repeat;
    $CONTESTION ← false;$
    $LOCK.release\_SF\_lock();$
    return(res)
end operation.

Part VII

Conclusion
What do we have visited?

- Concurrent objects
- Different types of objects
- Safety vs liveness
- Lock-based vs mutex-free implementations
- Notion of a hybrid implementation
- Abortable objects
- Systematic transformations