Schedule checker for concurrency control

Data Management

Facoltà di ingegneria dell'informazione, informatica e statistica
Engineering in Computer Science
Sapienza, Rome, Italy

Andrea Panceri 1884749, Francesco Sudoso 1808353

Academic Year 2022/2023

Contents

List of Figures

1 Introduction on concurrency control

Concurrency control is a critical aspect of database management systems (DBMS) that addresses the challenge of ensuring the correct behavior of concurrently executing transactions. The problem arises due to the potential interleaving of transactions, where multiple transactions are executed simultaneously, possibly leading to undesirable outcomes such as data inconsistencies or violations of the ACID properties.

In the context of the ACID properties mentioned, concurrency control aims to maintain these properties even in the presence of concurrent transaction execution. Let's delve deeper into the ACID properties and the problem of concurrency control:

- **Atomicity:** This property ensures that either all the actions within a transaction are completed, or none of them take effect. In the event of a failure, such as a system crash or an error, the DBMS should ensure that partially completed transactions are rolled back to maintain consistency.
- **Consistency:** Each transaction execution should bring the database from one consistent state to another. This means that the integrity constraints and business rules defined for the database should not be violated by any transaction. Concurrency control must prevent situations where interleaved transactions result in a database state that violates these constraints.
- **Isolation:** Transactions should be executed as if they are the only ones running in the system. This isolation property prevents interference between transactions. Concurrent transactions should not interfere with each other's intermediate states or outcomes. The result of concurrent transaction execution should be the same as if they were executed serially.
- **Durability:** Once a transaction is successfully committed, its effects should be permanently recorded in the database. Even in the face of system failures, committed changes should not be lost.

The main challenge in achieving these ACID properties under concurrent execution lies in preventing conflicts between transactions. Conflicts can occur when multiple transactions try to access and modify the same data simultaneously. These conflicts include:

- Read-Write Conflicts ri(x) ... wj(x): One transaction reads data while another modifies it.
- Write-Write Conflicts wi(x) ... wj(x): Two transactions attempt to modify the same data simultaneously.
- Read-Write Conflicts wi(x) ... rj(x): A transaction reads data while another is modifying it.

To address these conflicts and ensure correct concurrent execution, concurrency control mechanisms are employed. These mechanisms may include locks, timestamps, serialization schedules, and optimistic concurrency control. These techniques manage access to data and coordinate transaction execution to maintain the ACID properties.

In summary, the problem of concurrency control is about enabling high throughput and efficient execution of transactions while preserving the ACID properties. It involves careful synchronization and coordination of concurrent transactions to prevent conflicts and inconsistencies in the database. Various strategies and techniques are used to ensure that the system behaves as if transactions were executed serially, even when multiple transactions are executed simultaneously.

Schedules are crucial concepts in concurrency control. They define the order in which actions from different transactions are executed and determine how the database evolves over time. There are various types of schedules:

- **Partial Schedule:** A partial schedule is a sequence of actions from a subset of transactions. It doesn't necessarily include all the actions from all transactions. Partial schedules can arise in systems where not all transactions are executed concurrently or are interleaved. These schedules are generally not as interesting in the context of database concurrency control because they don't capture the full picture of concurrent execution.
- **Total (or Complete) Schedule:** A total schedule is a sequence of actions that includes all the actions of all transactions involved. In other words, it is a schedule where every action from every transaction is present. Total schedules are of primary interest in concurrency control because they represent the complete execution of the concurrent transactions.
- **Serial Schedule:** A serial schedule is a type of total schedule in which the actions of each individual transaction are executed consecutively without any interleaving with actions from other transactions. In a serial schedule, the actions of one transaction are completed before the actions of another transaction begin. This guarantees isolation between transactions and makes it easy to reason about the order of execution. However, serial schedules do not provide high concurrency, as transactions are executed one after the other.

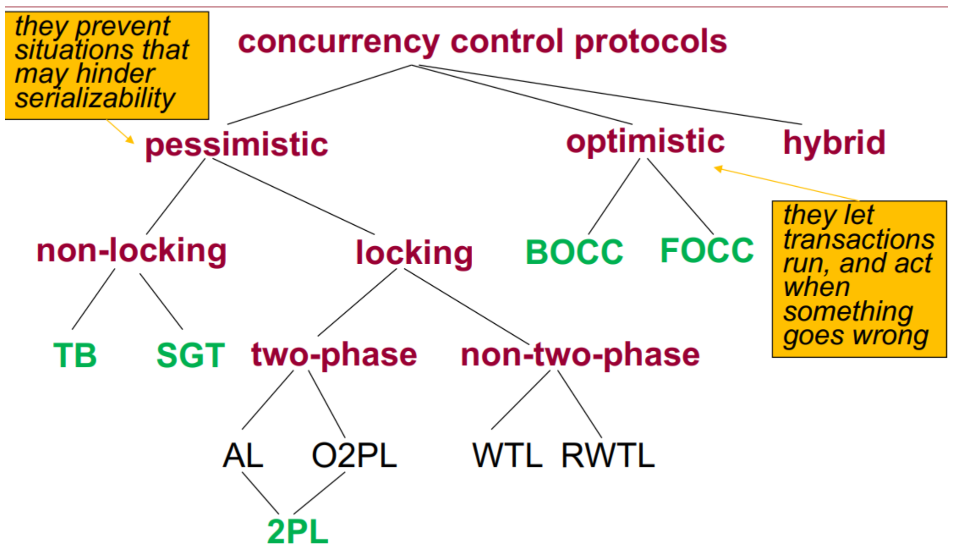

they prevent situations that may hinder serializability
concurrency control protocols
pessimistic
optimistic
hybrid
they let transactions run, and act when something goes wrong
non-locking
locking
BOCC
FOCC
TB
SGT
two-phase
non-two-phase
AL
O2PL
WTL
RWTL
2PL

Figure 1: Protocols

The web application we developed tackles the concurrency control by identifying whether a user-given schedule falls within the following classes of schedules:

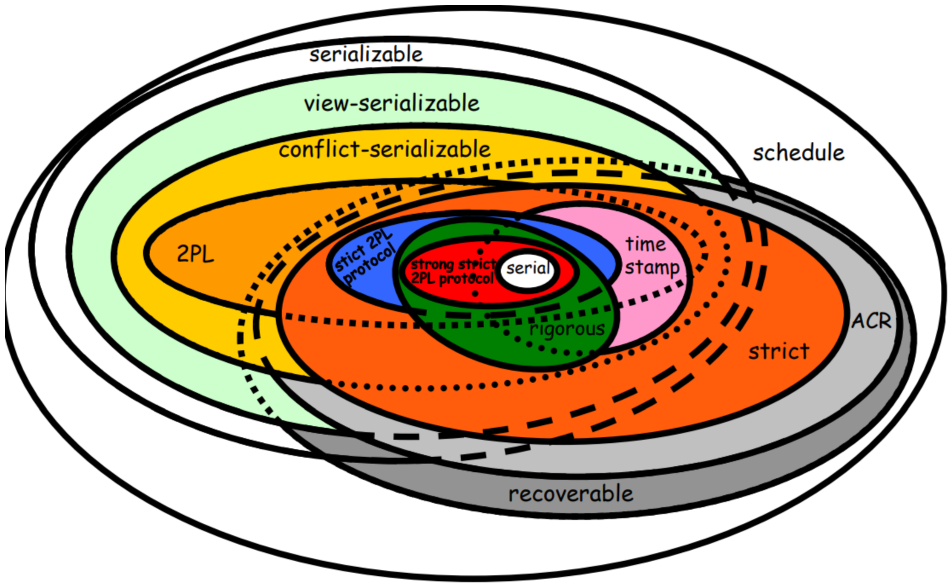


Figure 2: Classes of schedules

2 Frameworks used

Several frameworks and libraries were used in the development of our application. The most important among them are:

- **Bootstrap**, that simplifies the process of creating responsive and visually appealing web applications by providing a foundation of reusable components and styles. Provides a set of pre-designed HTML, CSS, and JavaScript components, as well as a responsive grid system, that makes it easier to create consistent and modern user interfaces.
- **Flask**, that is a lightweight Python web framework and provides the core functionality needed for web development without imposing a lot of constraints or adding unnecessary layers of complexity. For our project it was mainly used to handle Routing, HTTP Request Handling and HTTP Response Generation.
- **D3.js**, that is a JavaScript library that is used for creating dynamic and interactive data visualizations in web browsers. Provides layout algorithms for creating common data visualizations, such as pie charts, bar charts, line charts, scatter plots, and more. In our case it was used to reproduce the precedence graph image in the case of conflict serializability.

3 Serializability

Serializability is a fundamental concept in concurrency control that ensures the correctness of concurrent execution of transactions. In essence, it guarantees that the outcome of a concurrent schedule is equivalent to the outcome of some serial execution of the same transactions. This equivalence ensures that even though transactions are executed concurrently, the final database state remains consistent as if they were executed in some sequential order.
In the context of your provided example, let's discuss serializable and non-serializable schedules:

- **Serializable Schedule:** In a serializable schedule, the final outcome is equivalent to some serial execution of the transactions involved. This means that the final state of the database, as well as any intermediate local states, is the same as it would be in a serial execution.

- **Non-Serializable Schedule:** On the other hand, a non-serializable schedule doesn't guarantee such equivalence with any serial execution. In this case, the interleaving of actions can lead to different outcomes compared to a serial execution.

In summary, serializability is a key property that ensures that concurrent schedules maintain the same consistency and correctness as if the transactions were executed serially. It provides a strong foundation for designing concurrency control mechanisms that prevent conflicts and guarantee ACID properties, even in a highly concurrent environment.

3.1 View Serializability

View-serializability is a concept that focus on the order of read and write operations in schedules to determine if two schedules are equivalent in terms of their outcomes. This concept provide a way to ensure that concurrent schedules maintain the same effects as if they were executed serially, even though the specific order of execution might differ.

- **View-Equivalence:** Two schedules, S1 and S2, are considered view-equivalent if they share the same **READS-FROM** relation and the same **FINAL-WRITE** set. The READS-FROM relation captures the order in which read operations read their values from preceding write operations. The FINAL-WRITE set includes the last write action on each data item in the schedule.

- **View-Serializability:** A schedule S is view-serializable if there exists a serial schedule S′ that is view-equivalent to S. In other words, a view-serializable schedule can be transformed into a serial schedule through a rearrangement of actions, while preserving the order of read and write operations and their final effects.

This concept allow us to reason about the correctness of concurrent schedules by comparing their READS-FROM relations and FINAL-WRITE sets. If two schedules are view-equivalent, it means that they produce the same final outcomes, even though they might execute reads and writes in different interleaved orders. View-serializability guarantees that the concurrency introduced by executing transactions concurrently doesn't affect the final database state as long as the order of read and write operations is properly managed. We report the algorithm that we implement for check if a schedule is view serializable:

```python
def SolveViewSerializability(schedule):
    isViewSerializable = False

    transactions = set(action.id_transaction for action in schedule)

    read_from_schedule = read_from(schedule)
    last_write_schedule = last_write(schedule)

    for ordering in permutations(transactions):
        reordered_schedule = []
        for transaction_id in ordering:
            transaction_action = [
                action for action in schedule if action.id_transaction ==
    transaction_id]
            reordered_schedule.extend(transaction_action)
        if (Counter(read_from(reordered_schedule)) == Counter(read_from_schedule) and last_write(reordered_schedule) == last_write_schedule):
            res_equivalent = ''
            for id in ordering:
                res_equivalent += 'T' + str(id)
            return True, res_equivalent

    return isViewSerializable, None


def last_write(schedule):
    last_write = {}
    for action in schedule:
        if action.action_type == 'READ' or action.action_type == 'COMMIT':
            continue
        else:
            last_write[action.object] = action.id_transaction

    return dict(last_write)


def read_from(schedule):
    read_from = []
    for first in range(len(schedule)):
        firstAction = schedule[first]
        if firstAction.action_type == 'READ' or firstAction.action_type == 'COMMIT':
            continue

        for second in range(first + 1, len(schedule)):
            secondAction = schedule[second]

            if secondAction.action_type == 'READ' and firstAction.id_transaction != secondAction.id_transaction and firstAction.object == secondAction.object:
                read_from.append((secondAction, firstAction))
    return read_from
```

The code works by checking all possible permutations of transaction orderings and comparing the READS-FROM relations and the LAST-WRITE set of the original schedule with those of the reordered schedules. Let's break down the code step by step:

- SolveViewSerializability(schedule): This is the main function that takes a schedule as input and aims to determine if it's view-serializable.
  - transactions is a set that stores the unique IDs of transactions present in the schedule.
  - read_from_schedule is computed by calling the read_from function, which generates the READS-FROM relations for the input schedule.
  - last_write_schedule is computed by calling the last_write function, which generates the LAST-WRITE set for the input schedule.

  The function then iterates through all permutations of transaction orderings and checks if the READS-FROM relations and LAST-WRITE set match with the original schedule's relations and set. If a match is found, the ordering of transactions that resulted in a view-serializable schedule is returned. If no match is found, the function returns False for non-view-serializability.

- last_write(schedule): This function takes a schedule as input and computes the LAST-WRITE set, which maps each object to the ID of the transaction that last wrote to it.

This information is stored in the last_write dictionary.

- read_from(schedule): This function computes the READS-FROM relations between actions in the schedule. It checks for READ actions that are reading from a preceding WRITE action on the same object in a different transaction. The relations are stored as pairs in the read_from list.

3.2 Conflict Serializability

Conflict serializability is a crucial concept in the field of database management and concurrency control. It's a property of transaction schedules that ensures that the final outcome of a set of transactions is equivalent to some serial execution of those transactions. This property is essential for maintaining data consistency and integrity in multi-user database systems.
Conflict serializability is established by analyzing the relationship between actions of different transactions that operate on the same data elements. These actions are typically read and write operations, and they can potentially conflict with each other, leading to non-deterministic outcomes if not managed properly.

- **Conflict-Equivalence:** This is a fundamental concept that describes the ability to transform one schedule of actions into another through a series of swaps, where the swaps follow the commutativity rule. The commutativity rule states that if two actions from different transactions can be executed in either order without violating data integrity, then they are conflict-equivalent.

- **Conflict-Serializability:** This property ensures that a given schedule of transactions is equivalent to some serial order of those transactions that preserves the integrity of data. In other words, a conflict-serializable schedule ensures that the final result is the same as if the transactions were executed one after the other in some order.

To check if a schedule is conflict-serializable, you can analyze its precedence graph (also known as the conflict graph). This is the algorithm that we use:

```python
def SolveConflictSerializability(schedule):
    transactions = set([op.id_transaction for op in schedule])

    graph = {tx: set() for tx in transactions}

    for op1, i in zip(schedule, range(len(schedule))):
        if op1.action_type == 'COMMIT':
            continue
        for op2 in schedule[i+1:]:
            if op1.object != op2.object:
                continue
            if op1.id_transaction == op2.id_transaction:
                continue
            if op1.action_type == 'WRITE' or op2.action_type == 'WRITE':
                graph[op1.id_transaction].add(op2.id_transaction)

    def DFS(node_current, node_start, visited):
        for children in graph[node_current]:
            if children == node_start:
                return True
            if not children in visited:
                visited.add(children)
                is_cycle = DFS(children, node_start, visited)
                if is_cycle:
                    return is_cycle
        return False

    cycle_on_node = map(lambda n: DFS(n, n, set()), graph)

    return not any(cycle_on_node)
```

- **Input:** The function takes a schedule as its input. This schedule represents a sequence of operations performed by different transactions. Each operation has attributes like id_transaction (transaction identifier), action_type (action type such as 'READ' or 'WRITE'), and object (the object being read or written).

- **Creating Transaction Set:** The code starts by extracting the unique transaction identifiers from the schedule and stores them in a set called transactions.

- **Creating the Graph:** A directed graph (graph) is created using a dictionary where each transaction identifier is a key, and the value is a set of transaction identifiers. This graph represents the potential conflicts between transactions based on their actions on the same object.

- **Building the Graph:** The code then iterates through the schedule to build the graph. For each operation (op1), it checks if the operation is a COMMIT; if yes, it skips it. For non-COMMIT operations, it checks subsequent operations (op2) in the schedule to identify potential conflicts. Conflicts arise when operations on the same object are performed by different transactions and at least one of the operations is a WRITE operation. In this case, a directed edge is added from the transaction of op1 to the transaction of op2 in the graph.

- **DFS for Cycle Detection:** The function defines a Depth-First Search (DFS) helper function called DFS to check for cycles in the graph. It starts the DFS from a given node_current (transaction identifier), keeping track of visited nodes to prevent revisiting the same node. The DFS explores the graph by recursively traversing child transactions of the current node. It aims to detect whether there's a path from the node_current back to the node_start. If such a path is found, it indicates the presence of a cycle, which would mean that the transactions are not conflict serializable.

- **Checking for Cycles on Nodes:** The map function applies the DFS function to each transaction in the graph, starting a DFS from each transaction's node. The results (True if a cycle is found, False otherwise) are collected into a cycle_on_node list.

- **Final Decision:** The function returns True if there are no cycles detected in any of the transactions indicating that the schedule is conflict serializable. If there's at least one cycle detected, it returns False, indicating that the schedule is not conflict serializable.

3.2.1 Precedence Graph

The **precedence graph** is a visual representation of the conflicts between transactions in a schedule. Nodes in the graph correspond to transactions, and edges represent conflicts between transactions' actions on the same data element. If an edge exists between transaction Ti and Tj, it indicates that there are conflicting actions between these transactions.
By analyzing the structure of the precedence graph, you can determine if a schedule is conflict-serializable. If the graph is acyclic, it implies that the transactions can be reordered without causing any conflicts, making the schedule conflict-serializable. Now we report our code for compute and render the precedence graph:

```python
def ComputePrecedenceGraph(schedule):
    precedence_graph = set()  # Using a set to avoid duplicates

    for first in range(len(schedule)):
        firstAction = schedule[first]
        if firstAction.action_type == 'COMMIT':
            continue

        for second in range(first + 1, len(schedule)):
            secondAction = schedule[second]

            if firstAction.id_transaction != secondAction.id_transaction and firstAction.object == secondAction.object:
                if (firstAction.action_type == 'WRITE' and secondAction.action_type == 'WRITE') or \
                   (firstAction.action_type == 'READ' and secondAction.action_type == 'WRITE') or \
                   (firstAction.action_type == 'WRITE' and secondAction.action_type == 'READ'):
                    precedence_graph.add(
                        (firstAction.id_transaction, secondAction.id_transaction))

    return list(precedence_graph)
```

The function performs the following steps:

- Initializes an empty set called precedence_graph to store the unique pairs of transaction IDs that represent precedence relationships.
- Iterates through each action in the schedule using two nested loops. The outer loop considers the "first" action, and the inner loop considers the "second" action.
- Skips actions of type 'COMMIT', as they don't affect the precedence relationships.
- Compares the IDs of the transactions associated with the "first" and "second" actions. If they are different and the objects they interact with are the same, a potential precedence relationship is identified.
- Checks for specific conditions to determine if a precedence relationship exists between the two actions:
  - If both actions are of type 'WRITE'.
  - If the first action is a 'READ' and the second action is a 'WRITE'.
  - If the first action is a 'WRITE' and the second action is a 'READ'.
- If a valid precedence relationship is identified, a tuple containing the transaction IDs of the two actions is added to the precedence_graph set.
- After processing all possible pairs of actions, the function returns a list containing the unique precedence relationships in the form of tuples.

```javascript
function drawPrecedenceGraph(precedenceGraph) {

    // Update the title with the specified text
    $("#precedence-graph-container").html("<h4>Precedence Graph:</h4>");
    // Extract unique IDs from the precedenceGraph

    if (precedenceGraph.length > 0) {
        const uniqueIds = [...new Set(precedenceGraph.flat())];

        // Create SVG
        const svgWidth = 400;
        const svgHeight = 200;
        const svg = d3.select("#precedence-graph-container")
            .append("svg")
            .attr("width", svgWidth)
            .attr("height", svgHeight);

        // Add arrow marker
        svg.append("defs").append("marker")
            .attr("id", "arrowhead")
            .attr("refX", 6)
            .attr("refY", 2)
            .attr("markerWidth", 6)
            .attr("markerHeight", 4)
            .attr("orient", "auto")
            .append("path")
            .attr("d", "M0,0 V4 L6,2 Z") // Arrowhead path
            .style("fill", "#000000");

        // Create links
        const links = [];
        for (let i = 0; i < precedenceGraph.length; i++) {
            const [source, target] = precedenceGraph[i];
            links.push({ source, target });
        }

        const linkElements = svg.append("g")
            .selectAll("line")
            .data(links)
            .enter().append("line")
            .attr("x1", d => Math.cos((uniqueIds.indexOf(d.source) / uniqueIds.length) * 2 * Math.PI) * 50 + svgWidth / 2)
            .attr("y1", d => Math.sin((uniqueIds.indexOf(d.source) / uniqueIds.length) * 2 * Math.PI) * 50 + svgHeight / 2)
            .attr("x2", d => Math.cos((uniqueIds.indexOf(d.target) / uniqueIds.length) * 2 * Math.PI) * 50 + svgWidth / 2)
            .attr("y2", d => Math.sin((uniqueIds.indexOf(d.target) / uniqueIds.length) * 2 * Math.PI) * 50 + svgHeight / 2)
            .attr("marker-end", "url(#arrowhead)")
            .style("stroke", "#000000")
            .style("stroke-width", 2);

        const nodeElements = svg.append("g")
            .selectAll("circle")
            .data(uniqueIds)
            .enter().append("circle")
            .attr("r", 20)
            .attr("cx", d => Math.cos((uniqueIds.indexOf(d) / uniqueIds.length) * 2 * Math.PI) * 70 + svgWidth / 2)
            .attr("cy", d => Math.sin((uniqueIds.indexOf(d) / uniqueIds.length) * 2 * Math.PI) * 70 + svgHeight / 2)
            .style("fill", "#66ccff")
            .style("stroke", "#333")
            .style("stroke-width", 2)
            .style("pointer-events", "none");

        const labelElements = svg.append("g")
            .selectAll("text")
            .data(uniqueIds)
            .enter().append("text")
            .text(d => d)
            .attr("x", d => Math.cos((uniqueIds.indexOf(d) / uniqueIds.length) * 2
    * Math.PI) * 70 + svgWidth / 2)
            .attr("y", d => Math.sin((uniqueIds.indexOf(d) / uniqueIds.length) * 2
    * Math.PI) * 70 + svgHeight / 2 + 4)
            .style("text-anchor", "middle")
            .style("fill", "#333");

        // Center the SVG element within its container
        const containerWidth = $("#precedence-graph-container").width();
        const leftMargin = (containerWidth - svgWidth) / 2;
        $("svg").css("margin-left", `${leftMargin}px`);
    } else {
        $("#precedence-graph-container").append("<p>There are no conflicting
    actions, the Precedence graph is empty!")
    }
}
```

This JavaScript function uses the D3.js library to visualize a precedence graph based on the provided precedence relationships.
The function performs the following steps:

- Checks if the precedenceGraph has any precedence relationships. If it does, it proceeds with visualization; otherwise, it appends a message indicating that there are no conflicting actions.
- Extracts unique transaction IDs from the precedenceGraph and stores them in the uniqueIds array.
- Creates an SVG element with specified dimensions (svgWidth and svgHeight) using D3.js and appends it to the HTML container.
- Defines an arrow marker using SVG's (marker) element for indicating the direction of precedence relationships.
- Creates links between nodes in the SVG:
  - Loops through each precedence relationship in precedenceGraph to create an array of links.
  - Calculates positions for the source and target nodes based on their indices in the uniqueIds array and adds these to the links array.
- Appends SVG (line) elements to represent the links between nodes, using trigonometry to position the endpoints of the lines around a circle.
- Appends SVG (circle) elements for each unique ID to represent nodes in the graph, positioned using trigonometry.
- Appends SVG (text) elements to display text labels within the circles, indicating the transaction IDs.

3.2.2 OCSR

Order Preserving Conflict Serializable (OCSR) schedules are a type of schedule that aim to achieve both conflict serializability and maintain the relative order of conflicting operations as they appear in the original transactions. This means that if two transactions have conflicting operations (e.g., a read followed by a write) on the same data item in their original order, the execution of these conflicting operations in the concurrent schedule should maintain this order.
In order to check if a schedule is OCSR, we first check if the schedule S is conflict equivalent to a serial schedule S′. If there exists this serial schedule, then we calculate the order that this serial schedule should follow in order to be OCSR. If the order is respected by the serial schedule, then, S is OCSR. The order that S′ should follow is given by the following rule: for each transaction t,t′ in S, if t completely precedes t′ then the same should happen in S′.
The following algorithm checks if a schedule is OCSR:

```python
def get_initial_order(schedule):
    order_to_preserve = []
    for action in schedule:
        if action.id_transaction not in order_to_preserve:
            order_to_preserve.append(action.id_transaction)
    return order_to_preserve

def completely_precedes(transaction_1, transaction_2, schedule,
    commit_as_last_action):
    first_index_T1 = 0
    last_index_T1 = 0
    already_updated = False
    for action in schedule:
        if action.id_transaction == transaction_1:
            if action.action_type == 'COMMIT' or (action.isLastAction and not
    commit_as_last_action):
                last_index_T1 = schedule.index(action)
            elif not already_updated:
                first_index_T1 = schedule.index(action)
                already_updated = True

    for action in schedule[first_index_T1 : last_index_T1]:
        if action.id_transaction == transaction_2:
            return False

    return True

def is_schedule_containing_commits(schedule):
    for action in schedule:
        if action.action_type == 'COMMIT':
            return True
    return False

def get_all_possible_serial_schedule(schedule):
    transactions = set(action.id_transaction for action in schedule)

    read_from_schedule = read_from(schedule)
    last_write_schedule = last_write(schedule)
    serial_schedules = []
    for ordering in permutations(transactions):
        reordered_schedule = []
        for transaction_id in ordering:
            transaction_action = [
                action for action in schedule if action.id_transaction ==
    transaction_id]
```

```
43             reordered_schedule.extend(transaction_action)
44         if (Counter(read_from(reordered_schedule)) == Counter(read_from_schedule)
    and last_write(reordered_schedule) == last_write_schedule):
45             possible_order = []
46             for id in ordering:
47                 possible_order.append(id)
48             serial_schedules.append(possible_order)
49
50     if len(serial_schedules) > 0:
51         return serial_schedules
52     else:
53         return None
54
55 def is_ocsr(serial_schedule, must_comes_before):
56     for transactions in must_comes_before:
57         transaction_comes_before = transactions[0]
58         transaction_comes_after = transactions[1]
59         # ordine rispettato se indice di comes_before < indice di comes_after
60         if serial_schedule.index(transaction_comes_before) > serial_schedule.index
    (transaction_comes_after):
61             return False, f"because T{transaction_comes_before} must come before T
    {transaction_comes_after}"
62     serial_schedule_to_print = [f"T{transaction}" for transaction in
    serial_schedule]
63     return True, f"<br>Serial Schedule Conflict Equivalent to S and OCSR: <strong
    >{''.join(serial_schedule_to_print)}</strong>"
64
65 def solveOCSR(schedule):
66     order = get_initial_order(schedule)
67     commit_as_last_action = is_schedule_containing_commits(schedule)
68     must_comes_before = []
69     for index, transaction_1 in enumerate(order):
70         for transaction_2 in order[index+1:]:
71             if completely_precedes(transaction_1, transaction_2, schedule,
    commit_as_last_action):
72                 must_comes_before.append([transaction_1, transaction_2])
73
74     possible_serial_schedules = get_all_possible_serial_schedule(schedule)
75     if possible_serial_schedules != None:
76         for serial_schedule in possible_serial_schedules:
77             result, msg = is_ocsr(serial_schedule, must_comes_before)
78             if result:
79                 return result, msg
80         return result, msg
81
82     return False, "because not exist any serial schedule"
```

First of all, through lines 66-72 of the **solveOCSR** function, the order that the serial schedule(if any) should follow is calculated. This is done through the **completely_precedes** function, which, given *transaction_1* and *transaction_2* as input, checks whether all actions of *transaction_1* completely precede the first action of *transaction_2*. At the end of the execution of line 72 and at the beginning of line 73, we will have populated the variable **must_comes_before** which will be a list of arrays of two elements(transactions), of which the first must come before the second. The serial schedule should follow the order of transaction contained in this variable.
After that, the algorithm computes all possible serial schedules that are conflict-equivalent to the initial schedule (line 74). The code used by the function, is very similar to the code used earlier for checking view-serializability.
Obviously, if there is no serial schedule, the schedule cannot be OCSR either. Whereas, if

there is one or more serial schedule, then the algorithm, for each of them checks whether the order of the transactions in the serial schedule follows that contained in the *must_comes_before* variable, and it does this through the **is_ocsr** function. It will return True if they have the same order of transactions, otherwise False.

3.2.3 COCSR

Commit Order Preserving Conflict Serializable (COCSR) schedules are a type of schedule that aim to achieve that transactions are executed in a way that preserves the order in which they commit while also maintaining conflict serializability, which guarantees that the final state of the database is equivalent to some serial execution of the transactions. It can also be said that a schedule is COCSR if there is a serial schedule S' conflict-equivalent to S and for each transaction t1, t2, in S we have that t1 precedes t2 in S' if and only if the commit c1 of t1 precedes the commit c2 of t2 in S.
In order to check if a schedule is COCSR, we first check if the schedule S is conflict equivalent to a serial schedule S'. If there exists this serial schedule, then we calculate the order that this serial schedule should follow in order to be COCSR. If the order is respected by the serial schedule, then, S is COCSR.
The following algorithm checks if a schedule is COCSR:

```python
def get_commit_order(schedule, commit_as_last_action):
    commit_order = []
    for action in schedule:
        if action.action_type == 'COMMIT' or (action.isLastAction and not
    commit_as_last_action):
            commit_order.append(action.id_transaction)
    return commit_order

def is_cocsr(serial_schedule, commit_order):
    print(serial_schedule, commit_order)
    for transaction_serial_schedule, transaction_commit_order in zip(
    serial_schedule, commit_order):
        if transaction_serial_schedule != transaction_commit_order:
            return False, f"because any possible serial schedule, doens't follow
    the commit order of S"
    serial_schedule_to_print = [f"T{transaction}" for transaction in
    serial_schedule]
    return True, f"<br>Serial Schedule Conflict Equivalent to S and is COCSR: <
    strong>{''.join(serial_schedule_to_print)}</strong>"

def solveCOCSR(schedule):
    # get che commit order transactions of S
    commit_as_last_action = is_schedule_containing_commits(schedule)
    commit_order = get_commit_order(schedule, commit_as_last_action)
    # get, if exist, all possible serial schedules that is conflict-equivalent to
    schedule
    possible_serial_schedules = get_all_possible_serial_schedule(schedule)
    # check that the commit order transaction of S is equal to the order of
    transactions in the serial schedule
    if possible_serial_schedules != None:
        for serial_schedule in possible_serial_schedules:
            result, msg = is_cocsr(serial_schedule, commit_order)
            if result:
                return result, msg
        return result, msg

    return False, "because not exist any serial schedule"
```

The algorithm is the same as that of OCSR. The only thing that changes are the function for getting the order that the serial schedule should follows, **get_commit_order**, and the function **is_cocsr** that checks for validity of the order of the serial schedule accordingly to the order obtained by the *get_commit_order* function. Specifically, the order of the transactions returned by the function is given by the order in which the transactions perform their last action(implying that they are followed by the commit) or by the commit itself.
Obviously, if there is no serial schedule, the schedule cannot be COCSR either. Whereas, if there is one or more serial schedule, then the algorithm, for each of them checks whether the order of the transactions in the serial schedule follows that contained in the *commit_order* variable, and it does this through the **is_cocsr** function. It will return True if they have the same order of transactions, otherwise False.

4 Recoverability of transactions

Recoverability is a key concept in database systems that addresses the ability to restore the database to a consistent state after a failure occurs. It ensures that transactions are executed in such a way that the changes they make to the database are durable and can be properly recovered even in the presence of system failures. Recoverability is crucial for maintaining the integrity and reliability of the database.

4.1 Recoverable

If in a schedule S, a transaction Ti that has read from Tj commits before Tj, the risk is that Tj then rollbacks, so that Ti leaves an effect on the database that depends on an operation (of Tj) that never existed.
A schedule S is recoverable if no transaction in S commits before all other transactions it has “read from”, commit.
This is our algorithm:

```python
def SolveRecoverability(schedule):
    isRecoverable = True
    ConflictPair = None

    transactions = set(action.id_transaction for action in schedule)

    commitedTransactions = set()
    for action in schedule:
        if action.action_type == 'COMMIT':
            commitedTransactions.add(action.id_transaction)

    for id in transactions:
        read_from_id = read_from(schedule, id)
        if len(read_from_id) > 0:
            ended = False
            for action in schedule:
                if ended:
                    if action.id_transaction in read_from_id:
                        return False, (id, action.id_transaction)
                else:
                    if action.id_transaction == id and (action.action_type == '
    COMMIT' or (action.isLastAction and not (action.id_transaction in
    commitedTransactions))):
                        ended = True
    return isRecoverable, ConflictPair

```

```python

def read_from(schedule, transaction_id):
    read_from_id = set()
    for first in range(len(schedule)):
        firstAction = schedule[first]
        if firstAction.action_type == 'READ' or firstAction.action_type == 'COMMIT
    ' or firstAction.id_transaction == transaction_id:
            continue

        for second in range(first + 1, len(schedule)):
            secondAction = schedule[second]

            if secondAction.action_type == 'READ' and secondAction.id_transaction
    == transaction_id and firstAction.object == secondAction.object:
                read_from_id.add(firstAction.id_transaction)
    return read_from_id
```

- Extracting Transaction IDs: The unique transaction IDs are extracted from the schedule and stored in the transactions set.
- Extracting Committed Transactions: The transaction IDs of the committed transactions are extracted from the schedule and stored in the commitedTransactions set.
- Checking Recoverability:
  - The code loops through each transaction ID in transactions.
  - For each transaction ID, it identifies transactions that have read from this transaction. The read_from function is called to find such transactions.
  - If there are transactions that have read from the current transaction, the code iterates through the schedule to analyze their order.
  - It checks whether the current transaction (id) either commits or ends, and whether the transactions that read from it commit before or after it.
  - If there is a violation of recoverability (read before the reading transaction commits), the function returns False for isRecoverable and a tuple containing the conflicting transaction IDs for ConflictPair.
- read_from Function:
  - The read_from function takes the schedule and a transaction_id as parameters.
  - It iterates through the schedule to find transactions that have performed a READ operation from the specified transaction_id.
  - If a matching READ operation is found, the firstAction (the reading action) and the secondAction (the read-from action) are compared.
  - If the object being read is the same in both actions, the firstAction transaction ID is added to the read_from_id set.
- Returning Results: After processing all transactions, the function returns the isRecoverable boolean and the ConflictPair containing the conflicting transaction IDs (if any).

The code's purpose is to check whether a given schedule of transactions satisfies recoverability constraints. It identifies if there are cases where transactions are read from before the transactions they read from have committed. If such cases are found, the schedule is considered not recoverable, and the conflicting transaction pair causing the violation is reported.

4.2 ACR

S is ACR, i.e., avoids cascading rollback, if no transaction "reads from" a transaction that has not committed yet.

```python
def SolveACR(schedule):
    isACR, ConflictPair = read_from_ACR(schedule)
    return isACR, ConflictPair


def read_from_ACR(schedule):
    commitedTransactions = set()
    for action in schedule:
        if action.action_type == 'COMMIT':
            commitedTransactions.add(action.id_transaction)

    for first in range(len(schedule)):
        firstAction = schedule[first]
        if firstAction.action_type == 'READ' or firstAction.action_type == 'COMMIT
    ':
            continue

        haveCommit = False
        if firstAction.isLastAction and not (firstAction.id_transaction in
    commitedTransactions):
            haveCommit = True
        for second in range(first + 1, len(schedule)):
            secondAction = schedule[second]
            if not haveCommit and ((secondAction.action_type == 'COMMIT' and
    secondAction.id_transaction == firstAction.id_transaction) or (secondAction.
    isLastAction and secondAction.id_transaction == firstAction.id_transaction and
     not (secondAction.id_transaction in commitedTransactions))):
                haveCommit = True

            if secondAction.action_type == 'READ' and secondAction.id_transaction
    != firstAction.id_transaction and firstAction.object == secondAction.object:
                if not haveCommit:
                    return False, (secondAction.id_transaction, firstAction.
    id_transaction)
                else:
                    break

    return True, None
```

- SolveACR(schedule) Function: This function serves as the main interface. It takes a schedule as input and returns whether the schedule satisfies the ACR property and any conflicting transaction pair (if applicable). It calls the read_from_ACR function to perform the ACR check and returns its result.

- read_from_ACR(schedule) Function: This function performs the ACR check on the given schedule. It begins by creating a set called commitedTransactions to keep track of transaction IDs that have committed.

- Checking for ACR Violations: For each non-COMMIT action (firstAction) in the schedule, it checks if it's the last action for its transaction and if the transaction hasn't committed yet. It then iterates through the remaining actions in the schedule to look for actions that commit the same transaction as firstAction or the last action of the same transaction that hasn't committed yet. If such an action is found, it implies that the transaction has committed or is guaranteed to commit before any further reads from it. The haveCommit flag is set to True.

- Checking Reads: For each firstAction, it continues iterating through the remaining actions to check for potential ACR violations.

- After processing all actions, the function returns True if no ACR violations are found, indicating that the schedule satisfies the ACR property. If an ACR violation is found, it returns False and the conflicting transaction pair.

In summary, the code's purpose is to check whether a given schedule of transactions satisfies the Access-Committed-Read (ACR) property. The ACR property ensures that no transaction reads data from another transaction before the latter has committed, preventing uncommitted data from being read. If an ACR violation is found, the conflicting transaction pair causing the violation is reported.

4.3 Strict

The strictness condition for a schedule is related to ensuring that transactions preserve the consistency of the database while allowing for concurrency. A schedule is considered **strict** if every transaction reads only values written by transactions that have already committed, and writes only on transactions that have already committed. In other words, if transaction T1 reads a value produced by another transaction T2, transaction T1 cannot commit until transaction T2 has committed. This condition ensures that a transaction can commits only if it has a consistent and up-to-date view of the database, which helps maintain data integrity and prevents issues like dirty reads.
In order to check if a schedule is Strict, we get all possible pair of conflicting action, such as read-from and write-on pair. Then for each read-from pair, formed by $< r, w >$ we check that the transaction that wrote the object has committed before that the transaction that reads the object. We check the same for each pair of write-on pair, formed by $< w, w >$.
The following algorithm checks if a schedule is Strict:

```python
def get_all_commits(schedule):
    committed_transaction_index = {}
    for action in schedule:
        if action.action_type == 'COMMIT' or action.isLastAction:
            committed_transaction_index[action.id_transaction] = schedule.index(
action)
    return committed_transaction_index

def read_from(schedule):
    read_from = []
    for first in range(len(schedule)):
        firstAction = schedule[first]
        if firstAction.action_type == 'READ' or firstAction.action_type == 'COMMIT
':
            continue

        for second in range(first + 1, len(schedule)):
            secondAction = schedule[second]

            if secondAction.action_type == 'READ' and firstAction.id_transaction
!= secondAction.id_transaction and firstAction.object == secondAction.object:
                read_from.append((secondAction, firstAction))
    return read_from

```

```python
def write_on(schedule):
    write_on = []
    for first in range(len(schedule)):
        firstAction = schedule[first]
        if firstAction.action_type == 'READ' or firstAction.action_type == 'COMMIT':
            continue

        for second in range(first + 1, len(schedule)):
            secondAction = schedule[second]

            if secondAction.action_type == 'WRITE' and firstAction.id_transaction != secondAction.id_transaction and firstAction.object == secondAction.object:
                write_on.append((secondAction, firstAction))
    return write_on

def check_read_from_committed_transaction(schedule, conflicting_pair, indexes_of_commits):
    #calcolo posizione nella schedule del commit della transazione che fa la write,
    #calcolo posizione nella schedule della read
    #se posizione della read successiva alla posizione del commit della write -> ok, altrimenti NO STRICT
    read_action = conflicting_pair[0]
    write_action = conflicting_pair[1]
    index_of_commit_of_the_write = indexes_of_commits[write_action.id_transaction]
    index_of_read = schedule.index(read_action)
    if index_of_read < index_of_commit_of_the_write:
        return False, f" because <strong>{read_action}</strong> read from transaction <strong>{write_action}</strong> that has not yet committed"
    else:
        return True, None

def check_write_on_committed_transaction(schedule, conflicting_pair, indexes_of_commits):
    #calcolo posizione nella schedule del commit della transazione che ha fatto l'ultima write sull'oggetto
    #calcolo posizione nella schedule della nuova write
    #se posizione della nuova write successiva alla posizione del commit della precedente write -> ok, altrimenti NO STRICT
    write_action_1 = conflicting_pair[0]
    write_action_2 = conflicting_pair[1]
    index_of_commit_of_the_write_action_2 = indexes_of_commits[write_action_2.id_transaction]
    index_of_write_action_1 = schedule.index(write_action_1)
    if index_of_write_action_1 < index_of_commit_of_the_write_action_2:
        return False, f" because <strong>{write_action_1}</strong> write on transaction <strong>{write_action_2}</strong> that has not yet committed"
    else:
        return True, None

def SolveStrict(schedule):
    indexes_of_commits = get_all_commits(schedule)
    read_from_actions = read_from(schedule)
    write_on_actions = write_on(schedule)
    for conflicting_pair in read_from_actions:
        read_from_committed_transaction, msg = check_read_from_committed_transaction(schedule, conflicting_pair, indexes_of_commits)
        if read_from_committed_transaction is not True:
            return False, msg
```

```
70
71      for conflicting_pair in write_on_actions:
72          write_on_committed_transaction, msg = check_write_on_committed_transaction
    (schedule, conflicting_pair, indexes_of_commits)
73          if write_on_committed_transaction is not True:
74              return False, msg
75      return True, " because every transaction reads-from and writes-on transactions
     that have already committed"
```

First of all, through lines 63-65 of the **solveStrict** function, we calculate all the conflicting pair actions and all the indexes of the position of the commit actions in the schedule. This is done through the **read_from** function, which, given the schedule as input, it returns a list of array of two elements(actions), that are respectively $< read\_action, write\_action >$, which means that the read_action reads from the write_action (the last action that wrote the object that the read_action is reading). In addition, the **write_on** function, does the same, but in this case it returns a list of array of two elements(actions), that are respectively $< write\_on, write\_last >$, which means that the write_on action writes on the write_last (the last action that wrote the object that the write_on is writing). Also, the indexes of the position of the commit actions is returned by the **get_all_commits**.
Given these three variables, we can check if the schedule is Strict. Indeed, through lines 66-75 we check for Strictness by scanning for each conflicting pair, if there is a commit between these two actions. This is done by the functions **check_read_from_committed_transaction** and **check_write_on_committed_transaction**.
At the end, if for all conflicting pair there is a commit action (of the transaction involved in the conflicting pair), then the schedule is Strict.

4.4 Rigorousness

A schedule is considered **rigorous** if for each pair of conflicting actions of transaction T_i and T_j in S, the commit command of transaction T_i appears in S between the conflicting actions.
In order to check if a schedule is Rigorous, we get all possible pair of conflicting action for each object used by the transactions in the schedule. Then for each object, we scan all the conflicting pairs and check that between the actions involved in the conflicting pair there is a commit action.
The following algorithm checks if a schedule is Rigorous:

```
def SolveRigorousness(schedule):
    confliction_actions_dict = get_conflicting_actions(schedule)
    for object in confliction_actions_dict:
        for action_1 in confliction_actions_dict[object]:
            id_transaction_1 = action_1[0]
            action_type_1 = action_1[1]
            index_1 = action_1[2]
            for action_2 in confliction_actions_dict[object]:
                id_transaction_2 = action_2[0]
                action_type_2 = action_2[1]
                index_2 = action_2[2]
                if index_1 == index_2 or id_transaction_1 == id_transaction_2 or (
    action_type_1 == 'READ' and action_type_2 == 'READ') or index_1 > index_2:
                    continue
                else:
                    commit_operation = 'c'+id_transaction_1+'(None)'
                    commit_between_conflicting_actions, msg = check_commit_between
    (index_1, commit_operation, index_2, schedule)
                    if not commit_between_conflicting_actions:
                        return False, msg
```

```
19      return True, f"because for every pair of conflicting action there is a commit between them"
20
21  def check_commit_between(index_1, commit_operation, index_2, schedule):
22      portion_schedule = schedule[index_1 + 1: index_2]
23      index_last_action_of_transaction_1 = get_index_of_last_action(schedule, schedule[index_1].id_transaction)
24      if commit_operation in portion_schedule or schedule[index_1].isLastAction or index_last_action_of_transaction_1 < index_2:
25          return True, None
26      return False, f"because <strong>{commit_operation[:2]}</strong> <strong>is not </strong> between <strong>{schedule[index_1]}</strong> and <strong>{schedule[index_2]}</strong>"
27
28  def get_index_of_last_action(schedule, id_transaction_1):
29      for action in schedule:
30          if action.id_transaction == id_transaction_1 and action.isLastAction:
31              return schedule.index(action)
32
33  def  get_conflicting_actions(schedule):
34      confliction_actions = {}
35      # Setup the dict with all object used by transactions
36      for action in schedule:
37          if action.object != 'None' and action.object not in confliction_actions:
38              confliction_actions[action.object] = []
39
40      # For each object, we have an array of actions that use that object
41      for i in range(len(schedule)):
42          action = schedule[i]
43          if action.action_type == 'COMMIT':
44              continue
45          else:
46              confliction_actions[action.object].append([action.id_transaction, action.action_type,i])
47
48      return confliction_actions
```

Through line 2 of the **SolveRigorousness** function, we calculate all object used by the schedule and the corresponding conflicting pairs of actions.
This is done through the **get_conflicting_actions** function, which, given the schedule as input, it returns a dict that has as key the object, and as value an array constituted by the transaction, the type of action (such as read or write) and the index of the action in the schedule.
Given this dictionary, for each object inside it, we check if the action that use the object are in conflict. Indeed, this is done through lines 3-19. In particular, from line 4 to line 13 we have the case in which *action_1* and *action_2* are not in conflict, while from line 15 to line 19 the two actions are in conflict (two different transaction use the same object). In this last case, we check if there is a commit between these two actions through the **check_commit_between** function, that takes in input the indexes of the two actions in the schedule and the *commit_operation* that should be present between them.
At the end, if for all conflicting pair the *commit_between_conflicting_action* is true, then the schedule is Rigorous.

5 Concurrency control through locks

Concurrency control through locks ensure that multiple transactions can access and modify a shared database concurrently while maintaining data consistency and integrity. Lock-based concurrency control uses locks to regulate access to objects, preventing conflicting operations from occurring simultaneously. There are two types of locks:

- **Shared Lock (sl)**: allows multiple transactions to hold the lock simultaneously for reading purposes. Transactions with shared locks can't modify the locked data item.
- **Exclusive Lock (xl)**: allows exclusive access to an object, preventing other transactions from reading or writing to it while the lock is held.

Once a transaction has finished using a data item, it releases the lock it holds on that object. Releasing a lock allows other transactions to acquire it and access the data. In this case the action is the **Unlock (u)**.

5.1 2PL Compliance, S2PL and SS2PL

Two-Phase Locking (2PL) is a concurrency control protocol used to ensure serializability and prevent conflicts among transactions. It divides the execution of a transaction into two distinct phases: the growing phase and the shrinking phase:

- **Growing Phase:** during this phase, a transaction is allowed to acquire locks on data items but cannot release any locks. Once a lock is released during the shrinking phase, it cannot be reacquired.
- **Shrinking Phase:** during this phase, a transaction is allowed to release locks but cannot acquire any new locks. Once a lock is released, it is considered "unlocked," and other transactions can then request locks on that data item.

A schedule follows the 2PL protocol if in every transaction T_i of S, all lock operations (exclusive or shared) precede all unlocking operations of T_i. The main idea behind the 2PL protocol is to ensure that transactions acquire locks on data items before accessing them and to release those locks only when the transaction is complete, thereby preventing conflicts and ensuring proper synchronization between transactions. With the 2PL protocol conflicts and deadlocks can be prevented. A deadlock occurs when two or more transactions are waiting for locks held by each other, leading to a situation where none of them can proceed. It's important to note that the 2PL protocol can be implemented in different variations, each with varying levels of strictness in lock management and release. The choice of the specific 2PL variant depends on the database system's requirements and the desired trade-offs between concurrency and performance. In particular we have:

- **Strict 2PL (S2PL)**: A schedule S is S2PL if it follows the 2PL protocol, and all exclusive locks of every transaction T are kept by T until either T commits or rollbacks.
- **Strong Strict 2PL (SS2PL)**: A schedule S is SS2PL if it follows the 2PL protocol, and all locks (both exclusive and shared) of every transaction T are kept by T until either T commits or rollbacks.

In order to check if a schedule follows the 2PL protocol and then if it is S2PL or SS2PL, we check if for every transaction, its lock and unlock actions follows the requested 2PL protocol. The following algorithm checks if a schedule is 2PL, S2PL and SS2PL:

```python
states = {}
schedule = []
transactions = []
x_unlocks = []
unlocks = []
locks = []
use_xl_only = False

# Setup transactions and objects variables
def get_all_transactons_and_objects(schedule):
    transaction = []
    objects = []
    for action in schedule:
        if action.id_transaction not in transaction:
            transaction.append(action.id_transaction)
        if action.object not in objects:
            objects.append(action.object)
    return transaction, objects

# Represents if an object is available for locking/unlocking by the transaction.
    Values: 'BEGIN', 'sl', 'xl', 'u'
def setup_initial_states(transactions, objects):
    states = {}
    for transaction in transactions:
        if transaction not in states:
            states[transaction] = {}
        for object in objects:
            states[transaction][object] = 'BEGIN'
    return states

# Setup initial unlocks variables
def setup_initial_unlocks(transactions):
    x_unlocks = {} # true when the transaction has unlock the first exclusive lock
    unlocks = {} # true when the transaction has unlock the first lock (both
    shared or exclusive)
    for transaction in transactions:
        x_unlocks[transaction] = False
        unlocks[transaction] = False
    return x_unlocks, unlocks

# Store lock/unlock actions. An array [] for every transaction i in the schedule,
    +1 to add unlocks actions at the end
def setup_locks(schedule):
    ret = []
    for i in range(len(schedule)+1):
        ret.append([])
    return ret

# Given a 'state', it returns all the transactions with the object 'obj' in the
    state 'state'
def getTransactionsToUnlock(transactions, states, type_of_unlock_needed, obj,
    currentTransaction):
    transaction_to_unlock = []
    for type_of_unlock in type_of_unlock_needed:
        ret_unlock = []
        for transaction in transactions:
            if states[transaction][obj] == type_of_unlock:
                ret_unlock.append(transaction)
        transaction_to_unlock.append(ret_unlock)
    ret = mergeUnlocks(transaction_to_unlock, currentTransaction)
    return ret
```

```

# Returns a list with all the transactions having unlocks to be done(both shared
    and exlusive) on the same object, without the currentTransaction
def mergeUnlocks(transaction_to_unlock, currentTransaction):
    final_to_unlock_list = []
    for to_unlock_list in transaction_to_unlock:
        for transaction in to_unlock_list:
            if transaction not in final_to_unlock_list and transaction !=
    currentTransaction:
                final_to_unlock_list.append(transaction)
    return final_to_unlock_list

# Return lists of object that the currentTransaction will read or write in future
def get_object_to_read_and_write(remaining_schedule, currentTransaction):
    to_read = []
    to_write = []
    to_read_only = []
    # get to_write objects (for them we need the exclusive lock)
    for action in remaining_schedule:
        if action.id_transaction == currentTransaction and action.action_type == '
    READ':
            to_read.append(action.object)
        elif action.id_transaction == currentTransaction and action.action_type ==
     'WRITE':
            to_write.append(action.object)

    # get to_read_only objects (for them we only need the shared lock, not the
    exlusive)
    for elem in to_read:
        if elem not in to_write:
            to_read_only.append(elem)
    return to_read_only, to_write

# Unlock 'obj' for transaction 'currentTransaction'. We look in the future and
    check if, before unlocking an object, we need to acquire other locks.
def unlock(currentTransaction, obj, i):

    # The object is already unlocked or the transaction has just begun.
    if states[currentTransaction][obj] == 'u' or states[currentTransaction][obj]
    == 'BEGIN':
        return

    # We need to scan only the remaining schedule
    remaining_schedule = schedule[i+1:]

    # to_read_only contains objects that in the future will be readed-only by the
    transaction 'currentTransaction'. In this case we need only a shared lock
    # to_write contains objects that in the future will be written by the
    transaction 'currentTransaction'. In this case we need an exclusive lock
    to_read_only, to_write = get_object_to_read_and_write(remaining_schedule,
    currentTransaction)

    # for each object, we check if possible to acquire an exclusive lock ('xl').
    for obj_to_lock in to_write:
        ret = manageLocksAndState('xl', currentTransaction, obj_to_lock, i)
        if ret:
            return ret # Not 2PL

    # for each object, we check if possible to acquire a shared lock ('sl').
    for obj_to_lock in (to_read_only):
        ret = manageLocksAndState('sl', currentTransaction, obj_to_lock, i)
        if ret:
            return ret # Not 2PL

    # After all locks acquired, we can unlock the object 'obj' by the transaction 'currentTransaction'
    manageLocksAndState('u', currentTransaction, obj, i)

# Change the state of an 'obj' of 'transaction' to state 'target', adding the corresponding lock/unlock actions to solution list, and check if it is feasible.
def manageLocksAndState(target, currentTransaction, obj, i):

    if use_xl_only and target == 'sl':
        target = 'xl'

    if states[currentTransaction][obj] == target:
        return

    #if I want to do a shared lock, first I need to unlock the object from the transaction that has an exclusive lock
    #if I want to do a exclusive lock, then I need to unlock the object from the transaction that has any type of lock
    type_of_unlock_needed = ['xl'] if target == 'sl' else ['xl', 'sl']

    if target == 'sl' or target == 'xl':
        # Unlock transactions holding 'obj' in exclusive or shared lock before obtaining the desidered type of lock
        while True:
            transactions_to_unlock = getTransactionsToUnlock(transactions, states, type_of_unlock_needed, obj, currentTransaction)
            # Check if there are transactions to unlock
            if len(transactions_to_unlock) > 0:
                # If so, I take the last element in the list and start the unlock process for that object
                transaction_to_process = transactions_to_unlock.pop()
                # If the unlock is permitted, I will add the unlock in the locks array at position i
                ret = unlock(transaction_to_process, obj, i)
                if ret:
                    return ret  # Not 2PL
            else:
                break

    # strictness: check if the schedule unlocks an exclusive lock
    if states[currentTransaction][obj] == 'xl' and target == 'u':
        x_unlocks[currentTransaction] = True

    # strong strictness: check if the schedule unlocks any lock
    if (states[currentTransaction][obj] == 'xl' or states[currentTransaction][obj] == 'sl') and target == 'u':
        unlocks[currentTransaction] = True

    states[currentTransaction][obj] = target  # set target state

    # add the lock/unlock action to the solution
    tmp = Action(target, currentTransaction, obj)
    locks[i].append(tmp)

def solve2PL(schedule, use_xl_only_flag):
    global transactions
    global states
    global x_unlocks
    global unlocks
    global locks
    global use_xl_only

    ## SETUP VARIABLES ##
    use_xl_only = use_xl_only_flag
    transactions, objects = get_all_transactons_and_objects(schedule)
    states = setup_initial_states(transactions, objects)
    x_unlocks, unlocks = setup_initial_unlocks(transactions)
    is_strict, is_strong_strict = True, True
    locks = setup_locks(schedule)

    ## MAIN ##
    for i in range(len(schedule)):

        action = schedule[i]
        obj_state = states[action.id_transaction][action.object]

        if x_unlocks[action.id_transaction]: #The transaction has unlocked an
    exclusive lock and executes another action
            is_strict = False
        elif unlocks[action.id_transaction]: #The transaction has unlocked a lock
    and executes another action
            is_strong_strict = False

        # READ operation needs to get share lock
        if action.action_type == 'READ' and obj_state == 'BEGIN':
            ret = manageLocksAndState('sl', action.id_transaction, action.object,
    i)
            if ret:
                return ret # Not 2PL
        elif action.action_type == 'READ' and obj_state == 'u': #the action is
    trying to read from an unlocked object.
            return f'Not 2PL: <strong>{action}</strong> is trying to read from an
    unlocked object', None, None

        # WRITE operation needs to get exclusive lock
        if action.action_type == 'WRITE' and (obj_state == 'BEGIN' or obj_state ==
     'sl'):
            err = manageLocksAndState('xl', action.id_transaction, action.object,
    i)
            if err:
                return err
        elif action.action_type == 'WRITE' and obj_state == 'u': #the action is
    trying to write to an unlocked object
                return f'Not 2PL: <strong>{action}</strong> is trying to write to
    an unlocked object', None, None

    # Unlocks all the resources in use at the end of the schedule
    for currentTransaction in transactions:
            for obj in objects:
                state = states[currentTransaction][obj]
                if state == 'sl' or state == 'xl':
                    manageLocksAndState('u', currentTransaction, obj, len(schedule
    ))

    # get the final schedule with operation and locks/unlocks
    final_schedule = []
    for i in range(len(schedule)):
        final_schedule += locks[i] + [schedule[i]]
    final_schedule += locks[len(schedule)]
    final_schedule = formatSchedule(final_schedule)
    return final_schedule, is_strict, is_strong_strict
```

Through lines 1-7 we declare the global variables that we will need in order to take track of all locks and unlocks done by the transactions on every object. We setup all these variables through lines 165-170 of the **Solve2PL** function. In particular we have:

- **use_xl_only_flag** variable that indicate if the user wants only to use exclusive locks also for read operation, instead of shared locks.
- **transactions, objects** variables that are array initialized by the function **get_all_transactions_and_objects** which return two arrays respectively of all the transaction and all the objects involved in the schedule.
- **states** variables is obtained through the **setup_initial_states** function that returns a dict which represents if an object is available for locking/unlocking by the transaction. All possible values are *BEGIN*, *sl*, *xl*, *u*.
- **x_unlocks, unlocks** variables initialized by the **setup_initial_unlocks** function that returns two dict that have as key the transaction and as value a Boolean that is true for *x_unlocks* when the transaction has unlock the first exclusive lock and is true for *unlocks* when the transaction has unlock the first lock (both shared or exclusive). Initially, all value are False.
- **is_strict, is_strong_strict** variables that are flags initialized to True and will indicate if the schedule is respectively S2PL and SS2PL.
- **locks** variable initialized by the **setup_locks** function that returns all the lock/unlock actions needed by the schedule in order to follow the 2PL protocol. It is an array of length equals to the length of the schedule plus one, because we also need to insert the final unlocks at the end of the schedule. This variable will be used to compose the final schedule with all actions from the original schedule plus all locks and unlocks.

After the initiation of all the variables we will make use of, there is the main body of the function, that is, from the line 173 to line 197. In this for loop, for each action of the schedule we perform the following actions:

- lines 178-179 we check if the transaction has already performed an unlock on an exclusive block and performs another action. If this is the case, then the schedule is definitely not S2PL.
- lines 180-181 we check if the transaction has already performed an unlock on any lock and performs another action. If this is the case, then the schedule definitely is not SS2PL.

Next, depending on the type of action (read or write), we will need a specific type of lock (shared or exclusive). From line 184 to line 189 there is the case where the action is of type read, while from line 192 to line 197 there is the case where the action is of type write. In both cases, we will use the function **manageLocksAndState** in order to change the state of an object used by transaction to state *sl* (if the action type is read) or to state *xl* (if the action type is write), adding the corresponding lock/unlock actions to solution list, and checking if it change of state is feasible from the point of view of the 2PL protocol. Calling the previous function, we will also acquire all the locks that a specific transaction must require in order to follow the 2PL protocol (both S2PL and SS2PL). During the execution of this function, all

the locks and all the unlocks are appended to the *locks* variable in order to compose the final schedule at the end of the process.
After this process, will remain only to unlocks all the resources in use at the end of the schedule, and this is done through lines 200-204. At the end, we will compute the final schedule composed by all the action and locks/unlocks through lines 207-212, and we return if the schedule is S2PL or SS2PL with the respectively variables.

6 Concurrency control through timestamps

This approach utilizes timestamps associated with transactions to create a total order that dictates the execution order of transactions and ensures their serializability.
Each transaction is assigned a timestamp (ts) that reflects its order of arrival at the scheduler. Transactions that arrive earlier have lower timestamps.
Any schedule that respects the order defined by timestamps is conflict-serializable, meaning it is equivalent to a serial schedule.
The scheduler maintains various data for each data element X, including:

- **rts(X):** The highest timestamp among active transactions that have read X.
- **wts(X):** The highest timestamp among active transactions that have written X.
- **wts-c(X):** The timestamp of the last committed transaction that has written X.
- **cb(X):** A commit-bit that indicates whether the last transaction that wrote X has committed.

Transactions execute without explicit protocols or locks. Instead, their execution order is determined by their timestamps. For each action of a transaction T executed at a physical time t, the scheduler checks whether the ordering of the action according to physical time is compatible with the logical time ts(T).
The scheduler ensures that the action's logical time (timestamp) is in line with its execution order in the schedule.
The logical time of an action is its timestamp, and the commit-bit is used to prevent the dirty read anomaly (reading uncommitted data).
Timestamp-based concurrency control eliminates the need for complex locking protocols, allowing transactions to execute naturally in timestamp order. It guarantees conflict serializability and avoids the dirty read anomaly.
However, it does not eliminate the possibility of deadlocks, which may still occur. We report our implementation:

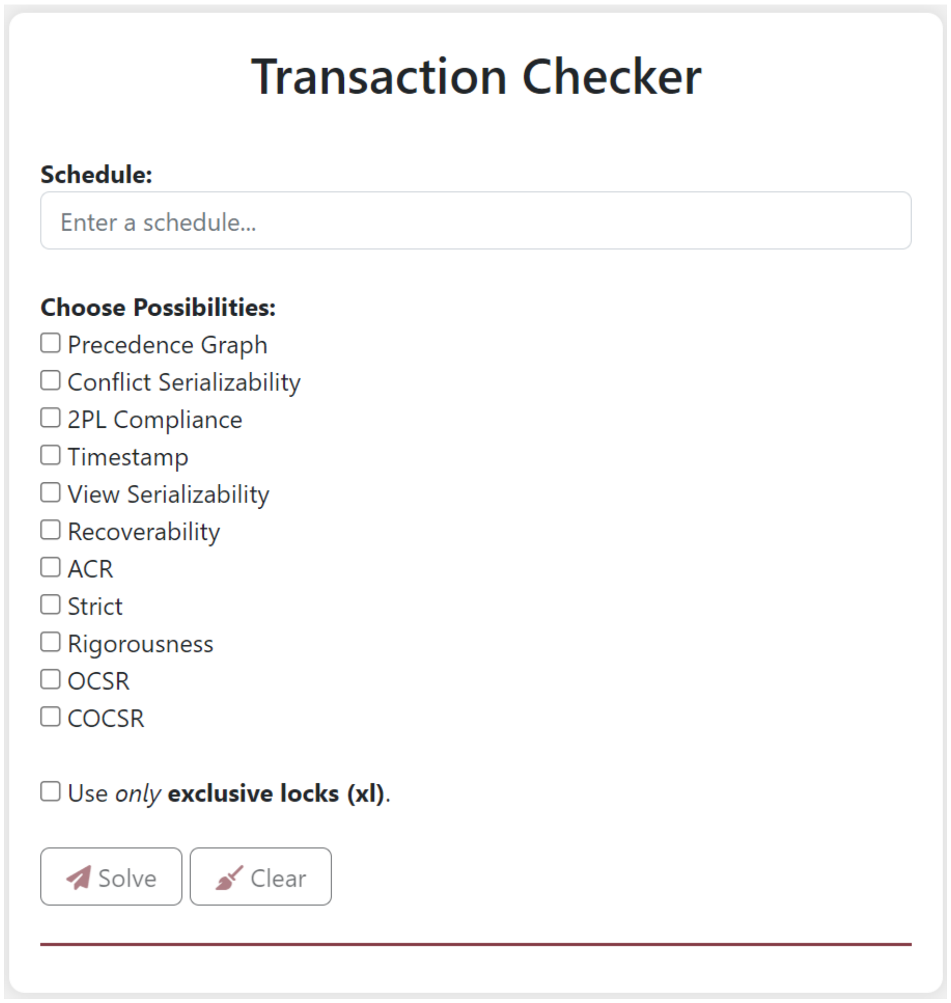
```python
rts = dict()

...

deadlock = dict()


def SolveTimestamps(schedule):
    objects = []
    transactions = []
    pending_action = []
    for action in schedule:
        if action.object not in objects:
            rts[action.object] = 0
            wts[action.object] = 0
```

```
                wts_c[action.object] = 0
                cb[action.object] = True
                last_write[action.object] = None
                objects.append(action.object)
            if action.id_transaction not in transactions:
                waiting[action.id_transaction] = (False, None)
                rollback[action.id_transaction] = False
                write_on[action.id_transaction] = set()
                transactions.append(action.id_transaction)

        formatted_schedule = format_schedule(schedule)
        deadlock['end'] = False

        ...

        solution += "<br>The system responds as follows:<br><ul class=\"list-group\">"

        clock_value = 1
        for action in formatted_schedule:
            isFirst = False
            if deadlock['end']:
                break
            elif waiting[action.id_transaction][0]:
                pending_action.append(action)
                solution += "<li class=\"list-group-item\">" + str(action) + " -->
    SKIP: because T" + \
                    action.id_transaction + " is in WAITING state, skipped for now."
            elif rollback[action.id_transaction]:
                solution += "<li class=\"list-group-item\">" + str(action) + \
                    " --> SKIP: because T" + action.id_transaction + " has done a
    rollback before."
            elif action.action_type == 'READ':
                if action.id_transaction not in timestamps:
                    isFirst = True
                    timestamps[action.id_transaction] = clock_value
                solution += read_action(action,
                                        action.id_transaction, action.object, isFirst)
            elif action.action_type == 'WRITE':
                if action.id_transaction not in timestamps:
                    isFirst = True
                    timestamps[action.id_transaction] = clock_value
                solution += write_action(action,
                                         action.id_transaction, action.object, isFirst
    )
            else:
                solution += commit_action(action.id_transaction)
            clock_value += 1

        solution += "</ul><br>"
        return solution


def read_action(action, transaction_id, object, isFirst):
    if timestamps[transaction_id] >= wts[object]:
        if cb[object] or timestamps[transaction_id] == wts[object]:
            rts[object] = max(timestamps[transaction_id], rts[object])
            action_solution = "<li class=\"list-group-item\">" + \
                str(action) + " --> ok --> "
            if isFirst:
                action_solution += "ts(T" + str(transaction_id) + ")=" + \
                    str(timestamps[transaction_id]) + ", "
            action_solution += "rts(" + str(object) + ")=" + str(rts[object])

        else:
            waiting[transaction_id] = (True, object)
            pending_action.append(action)
            if not waiting[last_write[object]][0]:
                action_solution = "<li class=\"list-group-item\">" + \
                    str(action) + " --> no: T" + \
                    str(transaction_id) + " must WAIT"
                if isFirst:
                    action_solution += " and ts(T" + str(transaction_id) + \
                        ")=" + str(timestamps[transaction_id])
            else:
                action_solution = "<li class=\"list-group-item\">" + \
                    str(action) + " --> no: T" + \
                    str(transaction_id) + " DEADLOCK"
                deadlock['end'] = True

    else:
        action_solution = rollback_action(action, transaction_id, isFirst)

    return action_solution


def write_action(action, transaction_id, object, isFirst):
    if timestamps[transaction_id] >= rts[object] and timestamps[transaction_id] >=
     wts[object]:
        if cb[object]:
            set_of_objects = write_on.get(transaction_id)
            set_of_objects.add(object)
            write_on[transaction_id] = set_of_objects

            wts[object] = timestamps[transaction_id]
            cb[object] = False
            last_write[action.object] = transaction_id

            action_solution = "<li class=\"list-group-item\">" + \
                str(action) + " --> ok --> "
            if isFirst:
                action_solution += "ts(T" + str(transaction_id) + ")=" + \
                    str(timestamps[transaction_id]) + ", "
            action_solution += "wts(" + str(object) + ")=" + \
                str(timestamps[transaction_id]) + \
                " and cb(" + str(object) + ")=False"

        else:
            waiting[transaction_id] = (True, object)
            pending_action.append(action)
            if not waiting[last_write[object]][0]:
                action_solution = "<li class=\"list-group-item\">" + \
                    str(action) + " --> no: T" + \
                    str(transaction_id) + " must WAIT"
                if isFirst:
                    action_solution += " and ts(T" + str(transaction_id) + \
                        ")=" + str(timestamps[transaction_id])
            else:
                action_solution = "<li class=\"list-group-item\">" + \
                    str(action) + " --> no: T" + \
                    str(transaction_id) + " DEADLOCK"
                deadlock['end'] = True

    else:
        if timestamps[transaction_id] >= rts[object] and timestamps[transaction_id] < wts[object]:
            if cb[object]:
                action_solution = "<li class=\"list-group-item\">" + str(action) + \
                    " --> SKIP: the action has been skipped for the THOMAS RULE"
                if isFirst:
                    action_solution += " and ts(T" + str(transaction_id) + \
                        ")=" + str(timestamps[transaction_id])

            else:
                waiting[transaction_id] = (True, object)
                pending_action.append(action)
                if not waiting[last_write[object]][0]:
                    action_solution = "<li class=\"list-group-item\">" + \
                        str(action) + " --> no: T" + \
                        str(transaction_id) + " must WAIT"
                    if isFirst:
                        action_solution += " and ts(T" + str(transaction_id) + \
                            ")=" + str(timestamps[transaction_id])
                else:
                    action_solution = "<li class=\"list-group-item\">" + \
                        str(action) + " --> no: T" + \
                        str(transaction_id) + " DEADLOCK"
                    deadlock['end'] = True

        else:
            action_solution = rollback_action(action, transaction_id, isFirst)
    return action_solution


def commit_action(transaction_id):
    set_of_objects = write_on.get(transaction_id)
    action_solution = "<li class=\"list-group-item\">" + \
        "c" + str(transaction_id) + " --> COMMIT: "
    for object in set_of_objects:
        action_solution += "cb(" + str(object) + ")=True, wts_c(" + \
            object + ")=" + str(timestamps[transaction_id]) + ", "
        cb[object] = True
        wts_c[object] = timestamps[transaction_id]
        for transaction in waiting:
            if waiting[transaction][0] and waiting[transaction][1] == object:
                waiting[transaction] = (False, None)
                action_solution += handle_pending_actions()

    return action_solution


def rollback_action(action, transaction_id, isFirst):
    set_of_objects = write_on.get(transaction_id)
    action_solution = "<li class=\"list-group-item\">" + str(action) + \
        " --> no: Rollback of T" + \
        str(transaction_id) + ", after the action set: "
    for object in set_of_objects:
        wts[object] = wts_c.get(object)
        cb[object] = True
        action_solution += "cb(" + str(object) + ")=True, wts(" + \
            object + ")=" + str(wts_c.get(object)) + ", "
        for transaction in waiting:
            if waiting[transaction][0] and waiting[transaction][1] == object:
                waiting[transaction] = (False, None)
                action_solution += handle_pending_actions()
    if isFirst:
        action_solution += " and ts(T" + str(transaction_id) + ")=" + \
            timestamps[transaction_id]

    rollback[transaction_id] = True
    return action_solution


def format_schedule(schedule):
    formatted_schedule = []
    commitedTransactions = set()
    for action in schedule:
        if action.action_type == 'COMMIT':
            commitedTransactions.add(action.id_transaction)

    for action in schedule:
        if action.isLastAction and not (action.id_transaction in
    commitedTransactions):
            formatted_schedule.append(action)
            current_action = 'COMMIT'
            ActionCurrent = Action(current_action, action.id_transaction, None)
            formatted_schedule.append(ActionCurrent)
        else:
            formatted_schedule.append(action)
    return formatted_schedule


def handle_pending_actions():
    actions_solution = ""
    for action in pending_action:
        isFirst = False
        if deadlock['end']:
            break
        elif waiting[action.id_transaction][0]:
            continue
        elif rollback[action.id_transaction]:
            actions_solution += "<li class=\"list-group-item\">" + str(action) + \
                " --> SKIP: because T" + action.id_transaction + " has done a
    rollback before."
        elif action.action_type == 'READ':
            actions_solution += read_action(action,
                                            action.id_transaction, action.object,
    isFirst)
        elif action.action_type == 'WRITE':
            actions_solution += write_action(action,
                                             action.id_transaction, action.object,
     isFirst)
        else:
            actions_solution += commit_action(action.id_transaction)
    return actions_solution
```

- Initialization: The code initializes various dictionaries to store information related to timestamps, waiting transactions, commit-bits, and other control variables.
- Preparing Data Structures: Before analyzing the schedule, the code prepares data structures like rts, wts, wts_c, cb, and others to keep track of relevant information for each object and transaction.

- Formatting the Schedule: The code formats the input schedule to ensure that every transaction ends with a COMMIT action. This is to handle cases where a transaction's last action is not a COMMIT.
- Solving Timestamps: The main function is SolveTimestamps(schedule), which takes a schedule of actions as input. The code initializes variables and structures to keep track of timestamps, pending actions, and deadlock states. It iterates through the schedule and processes each action according to the timestamp-based concurrency control rules.
- Action Processing: The code processes each action in the schedule based on its type (READ, WRITE, COMMIT). It checks conditions related to timestamps and the status of transactions to determine whether an action can be executed or not. Depending on the conditions, the code performs actions such as reading, writing, committing, or handling rollbacks. It handles situations where transactions are in a waiting state, and it also considers the "THOMAS RULE" to determine whether a WRITE action should be skipped.
- Handling Deadlocks:The code keeps track of deadlock situations and sets the deadlock['end'] flag to true if a deadlock is detected. Once this flag is set, processing stops.
- Generating Solution Steps: The code generates a textual solution that describes the outcome of each action processed. It includes details about the actions executed, their status (ok, no, SKIP), and the current values of variables like timestamps and commit-bits.

7 Final Product



Transaction Checker

Schedule:

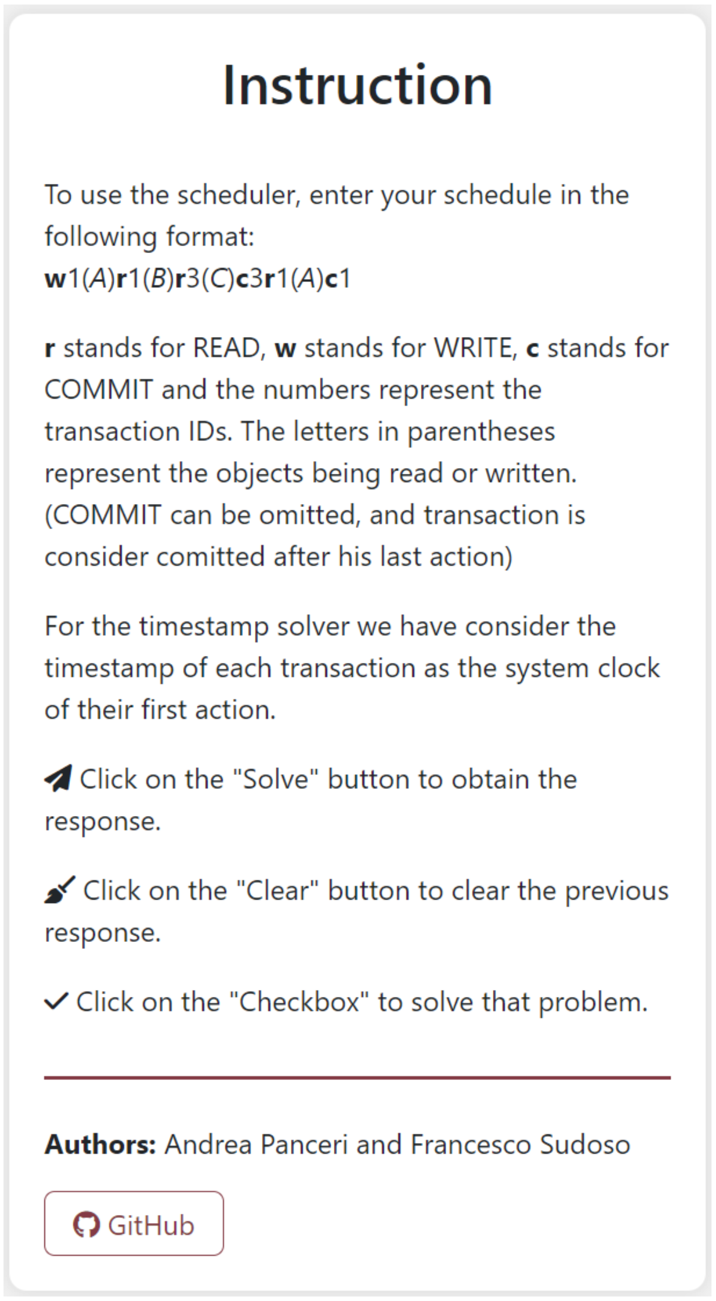
Enter a schedule...

Choose Possibilities:

- ☐ Precedence Graph
- ☐ Conflict Serializability
- ☐ 2PL Compliance
- ☐ Timestamp
- ☐ View Serializability
- ☐ Recoverability
- ☐ ACR
- ☐ Strict
- ☐ Rigorousness
- ☐ OCSR
- ☐ COCSR

☐ Use *only* **exclusive locks (xl)**.

Figure 3: Homepage



Instruction

To use the scheduler, enter your schedule in the following format:
w1(*A*)**r**1(*B*)**r**3(*C*)**c**3**r**1(*A*)**c**1

r stands for READ, **w** stands for WRITE, **c** stands for COMMIT and the numbers represent the transaction IDs. The letters in parentheses represent the objects being read or written. (COMMIT can be omitted, and transaction is consider comitted after his last action)

For the timestamp solver we have consider the timestamp of each transaction as the system clock of their first action.

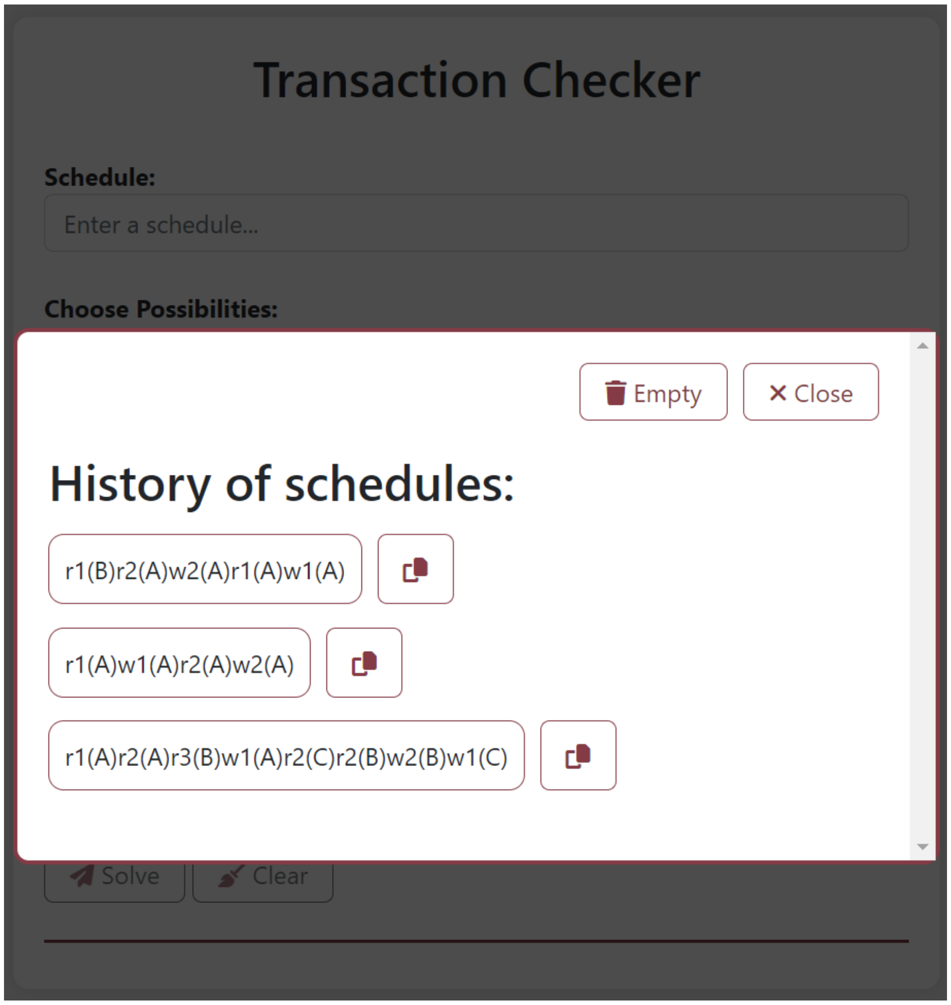
Click on the "Solve" button to obtain the response.

Click on the "Clear" button to clear the previous response.

✓ Click on the "Checkbox" to solve that problem.

Authors: Andrea Panceri and Francesco Sudoso

Figure 4: Instruction



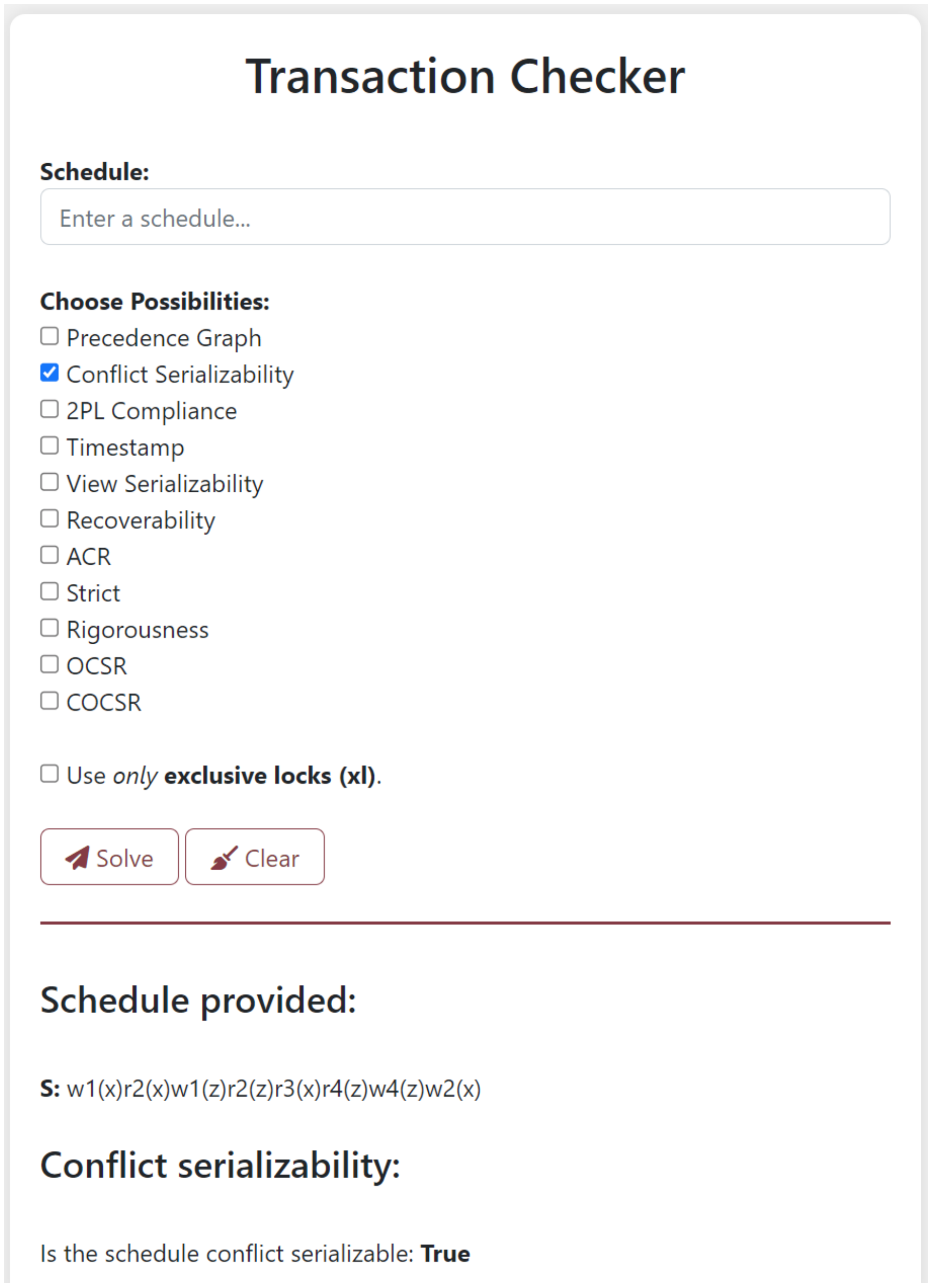
History of schedules:

r1(B)r2(A)w2(A)r1(A)w1(A)

r1(A)w1(A)r2(A)w2(A)

r1(A)r2(A)r3(B)w1(A)r2(C)r2(B)w2(B)w1(C)

Figure 5: History



Schedule:

Enter a schedule...

Choose Possibilities:

- [ ] Precedence Graph
- [✓] Conflict Serializability
- [ ] 2PL Compliance
- [ ] Timestamp
- [ ] View Serializability
- [ ] Recoverability
- [ ] ACR
- [ ] Strict
- [ ] Rigorousness
- [ ] OCSR
- [ ] COCSR

- [ ] Use *only* **exclusive locks (xl)**.

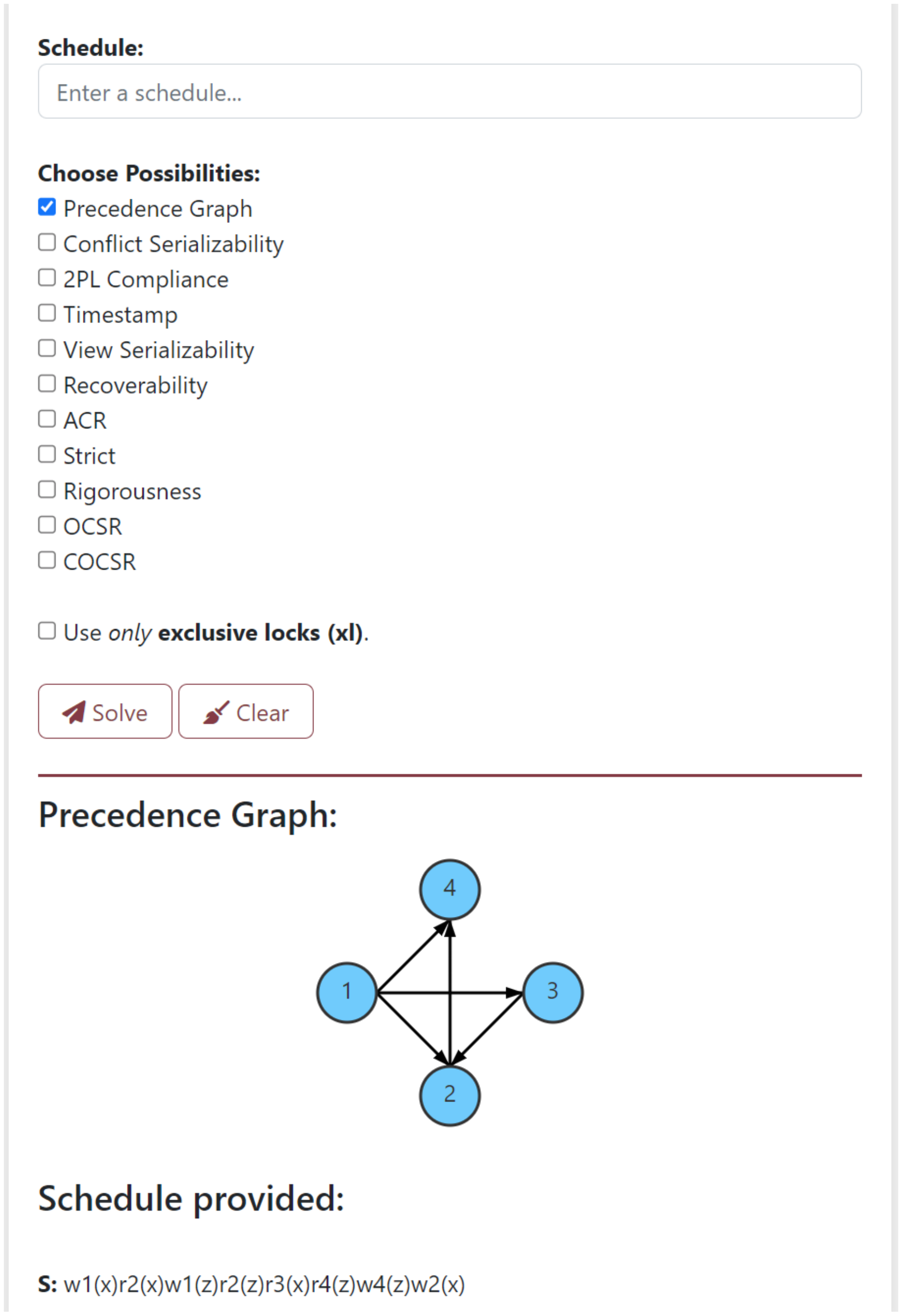
Schedule provided:

S: w1(x)r2(x)w1(z)r2(z)r3(x)r4(z)w4(z)w2(x)

Conflict serializability:

Is the schedule conflict serializable: **True**

Figure 6: Checker for Conflict Serializability



Schedule:

Enter a schedule...

Choose Possibilities:

- [x] Precedence Graph
- [ ] Conflict Serializability
- [ ] 2PL Compliance
- [ ] Timestamp
- [ ] View Serializability
- [ ] Recoverability
- [ ] ACR
- [ ] Strict
- [ ] Rigorousness
- [ ] OCSR
- [ ] COCSR

- [ ] Use *only* **exclusive locks (xl)**.

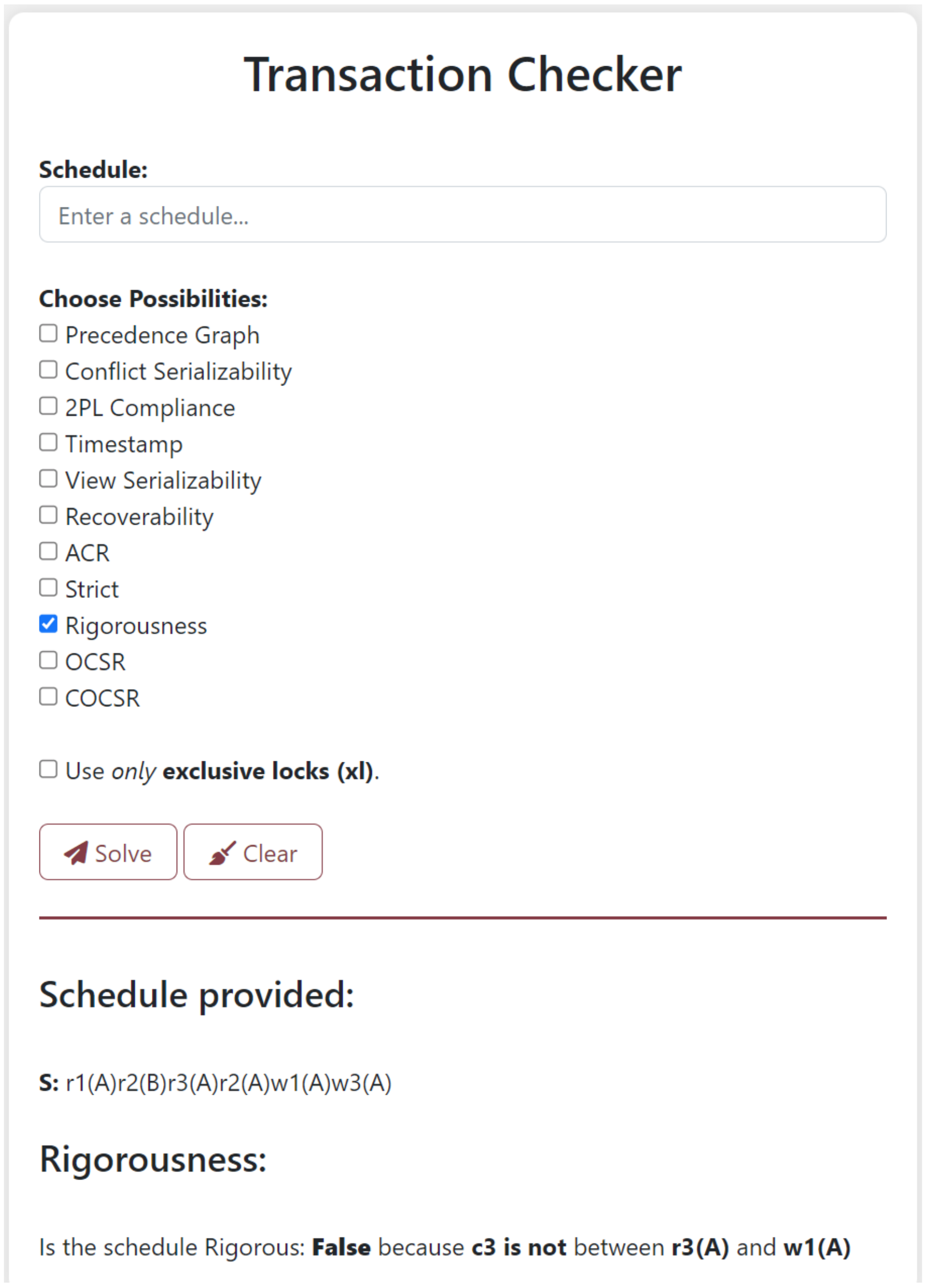
Precedence Graph:

$$\begin{tikzcd} & 4 & \\ 1 \arrow[ur] \arrow[rr] \arrow[dr] & & 3 \arrow[dl] \\ & 2 \arrow[uu] & \end{tikzcd}$$

Schedule provided:

S: w1(x)r2(x)w1(z)r2(z)r3(x)r4(z)w4(z)w2(x)

Figure 7: Precedence Graph



Transaction Checker

Schedule:

Enter a schedule...

Choose Possibilities:

- [ ] Precedence Graph
- [ ] Conflict Serializability
- [ ] 2PL Compliance
- [ ] Timestamp
- [ ] View Serializability
- [ ] Recoverability
- [ ] ACR
- [ ] Strict
- [✓] Rigorousness
- [ ] OCSR
- [ ] COCSR

- [ ] Use *only* **exclusive locks (xl)**.

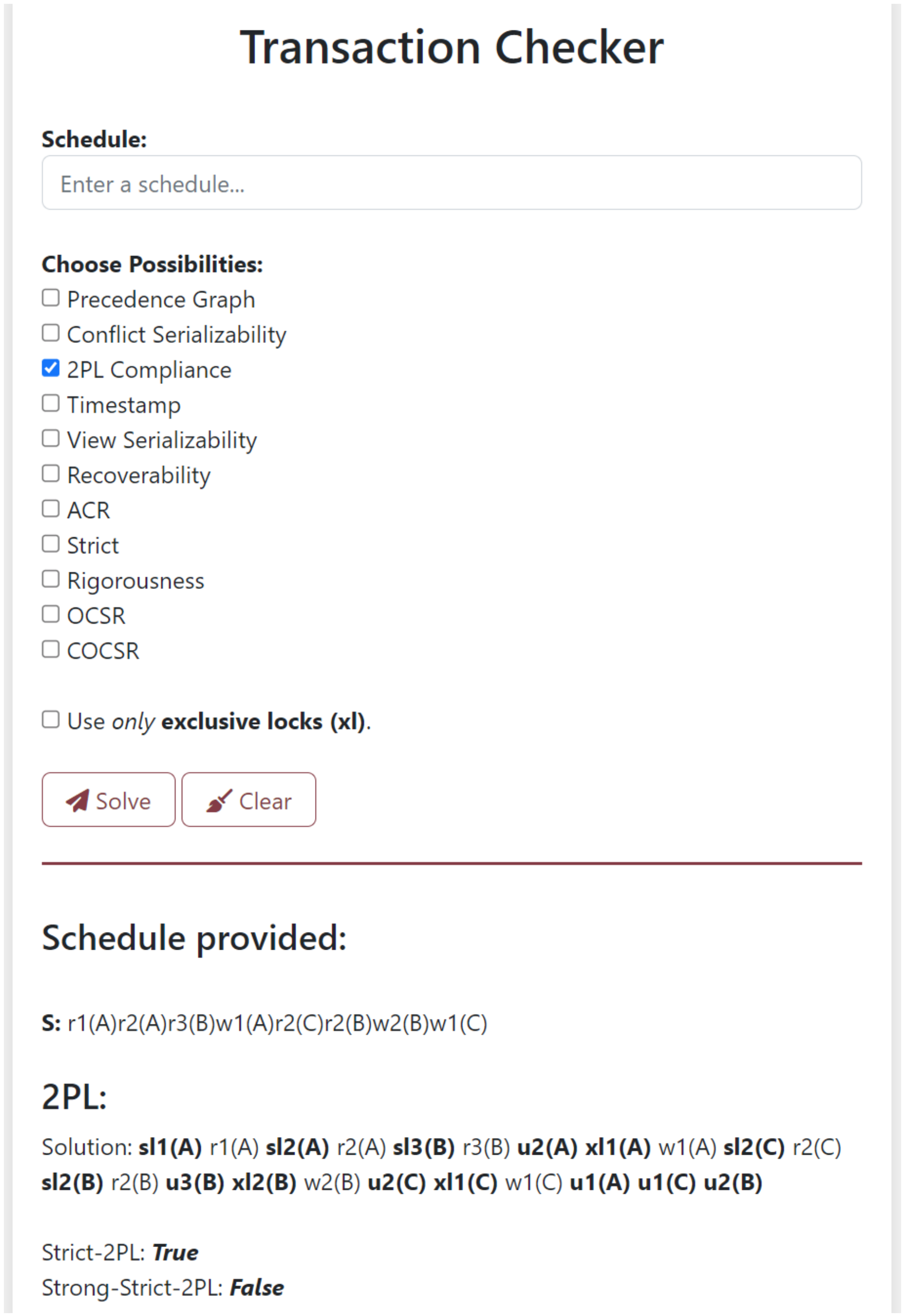
Schedule provided:

S: r1(A)r2(B)r3(A)r2(A)w1(A)w3(A)

Rigorousness:

Is the schedule Rigorous: **False** because **c3 is not** between **r3(A)** and **w1(A)**

Figure 8: Checker for Rigorousness


Transaction Checker

Schedule:

Enter a schedule...

Choose Possibilities:

- [ ] Precedence Graph
- [ ] Conflict Serializability
- [x] 2PL Compliance
- [ ] Timestamp
- [ ] View Serializability
- [ ] Recoverability
- [ ] ACR
- [ ] Strict
- [ ] Rigorousness
- [ ] OCSR
- [ ] COCSR

- [ ] Use *only* **exclusive locks (xl)**.

Schedule provided:

S: r1(A)r2(A)r3(B)w1(A)r2(C)r2(B)w2(B)w1(C)

2PL:

Solution: **sl1(A)** r1(A) **sl2(A)** r2(A) **sl3(B)** r3(B) **u2(A)** **xl1(A)** w1(A) **sl2(C)** r2(C) **sl2(B)** r2(B) **u3(B)** **xl2(B)** w2(B) **u2(C)** **xl1(C)** w1(C) **u1(A)** **u1(C)** **u2(B)**

Strict-2PL: ***True***

Strong-Strict-2PL: ***False***

Figure 9: Checker for 2PL Compliance