DataManagement Project : Report

Design and implement a software that emulates a scheduler based on a specific protocol or strategy for concurrency control

Andrea Nardocci - 1709323

October 2023

Contents

Introduction

Creating a software scheduler for concurrency control involves designing and implementing a system that can handle multiple concurrent data access requests efficiently. Serializability, view-serializability and conflict-serializability are extremely important in the theory of concurrency, since they represent the basic notions for characterizing the correctness of concurrency control. The goal of the scheduler is to analyze the input schedule resulting from the requested concurrent execution of multiple transactions, and to output a corresponding schedule (the sequence of actions that are really executed), according to a specific strategy.

In this example I will show how to implement **concurrency control strategy via timestamp**. It is based on the idea that each transaction T has an associated timestamp $ts(T)$ that is unique among the active transactions, and is such that $ts(Tj) < ts(Th)$ whenever transaction Tj arrives at the scheduler before transaction Th. In what follows, we assume that the timestamp of transaction Ti is simply $ts(Ti) = i$. So at each action execution, the scheduler checks whether the involved timestamps violates the serializability condition according to the order induced by the timestamps. Then since we use this strategy we need to take into account the rules that characterize the concurrency control process.

Futhermore unfortunately, the method based on timestamps does not avoid the risk of deadlock (although the probability is lower than in the case of other concurrency control like the one based on lock/unlock strategy). We recall that the deadlock occurs when two transactions $T1$ and $T2$ have the use of two elements A and B, and each of them is waiting for the commit of the others transaction, and therefore no one can proceed. Of course, a suitable technique for deadlock management must then be implemented. For the purpose of our project, we chose to adopt a technique of "deadlock recognition and solution" . So we need to implement a method in which we recognize deadlock and define how we can solve the situation.

Below, I'll outline the steps to design and implement a simple scheduler based on a common concurrency control strategy via timestamp.

1.1 Requirements

We need to define the requirements for our algorithm. Here are some of the key requirements:

- **Schedule**: How we can represent the scheduler (sequence of actions) through the use of a programming language.
- **Concurrency Control Strategy**: In this example, we'll use concurrency control through timestamp, and we need to outline the properties of the chosen one.
- **Rule**: We need to define methods and variables in order to replicate the rules that characterize concurrency control which are : *read ok , read to late , write ok , write too late, thomas rule.*
- **DeadLock Management** : Implement a mechanism to recognize deadlock and also to solve deadlock.

1.2 Algorithm

In the implementation of the algorithm, we have to take into account several steps, such as acquiring the information necessary for the correct execution of the code, like the scheduler sequence. In addition, there is the implementation of the concurrency control through timestamp. As a programming language, I chose python because of its versality and the fact that it provides a good basis for using GUI. So here are written the most crucial and important steps in the algorithm.

- ***Input Inizialization*** : we need to declare and assign variable useful for the algorithm, like the data that describe the status of each resources and transaction, we need to get from keyboard input a sequence of actions (scheduler) and some other internal variable useful for the ongoing of the algorithm. For example we represent the schedule as a list of tuple like where the tupla is in the following format : (transactionID, actionType, resource) so the user need to be able to insert the action in the correct way and so on.
- ***Apply Timestamp concurrency*** Core function of the program is to apply the strategy in the best way, the most crucial part is the implementation of the rules. In order to represent the associated values for each resources, we create the class ResourceInfo, which contains and keep track of the variables $(rts, wts, wts - c, cb)$ used by the rules of the strategy. Futhermore we also have a class for the transaction simply called Transaction, in which we keep track of the status of transaction (it can change from active, waiting, rollback)
- ***DeadLock Management*** : for this purpose since in the algorithm there's a list of the action that bring the transaction in waiting, we create a simply check by using this list in order to recognize deadlock. When a deadlock event appears, the algorithm create a random number for each different transaction belonging to the schedule in this way. So we need to check the priority number associated to the transaction involved in the deadlock event, and simply kill the transaction with less priority.
- ***Output*** Since we have a GUI, we need to find a good solution in order to display the results into good formatting, in order to be fully understandable for the users.

Implementation

Now let's get in the deep of the implementation of the above steps.

2.1 Define the Input Format

This are the class for the representation of the variable associated to the resources and the status of the transaction.

```python
class Transaction:
    def __init__(self, name):
        self.name = name
        self.active = True
        self.rollback = False
        self.waiting = False

class ResourceInfo:
    def __init__(self,name):
        self.name = name
        self.rts = 0
        self.wts = 0
        self.wts_c = 0
        self.cb = True
```

This are our usefull variable, used by the algorithm, in which :

```python
scheduler = []
ignored_actions = []
rollback_transaction = []
deadlock_detector = []
resource_info = []
transaction_info = []
deadlock_f = False
deadlock_s_exec = False
scheduler_solution = []
```

- **scheduler** = [ ] : is a list in which data schedulers are saved, in the order in which users enter actions into the program. The action is saved as a tuple of three elements and and respectively are : (transactionID, action_name, resource) i.e. ("T2","read","x")
- **ignored_actions** = [ ] : in this list we save the action that we need to ignore for the algorithm logic, and we store the ignored tuples when the transaction to which the analyzed action belongs is in a waiting or rollback state
- **deadlock_detector** = [ ] : Actions that have changed the status of the membership transaction to pending or rollback are saved in this list and are the ones that need to be kept track of, because they can generate deadlocks.
- **resource_info** = [ ] & **transaction_info** = [ ] : in this two list, we save the object class related to the resource and transaction class.

- **deadlock_f** : flag to denote if a deadlock event occour in the scheduler.
- **deadlock_s_exec** : flag to denote if in the application of the concurrency control we want to solve deadlock event.
- **scheduler_solution** = [ ] is a list in which we build the new scheduler obtained from the solution of the deadlock.

2.2 Process the Input

This is the way on how we handle the input written by the users in the dedicated section, so basically whenever you insert an action of the schedule it adds the action ,formatted as the tupla defined earlier, in the scheduler list.

```python
    def process_input(self, user_input):
        parts = user_input.split()
        if len(parts) == 2:
            action_type, transaction_id = parts
            if action_type not in ['commit','rollback']:
                #Message error
                ...

            else:
                resource = None
                transaction_id = "T"+str(transaction_id)
                scheduler.append((transaction_id, action_type, resource))
                self.display_scheduler()

        elif len(parts) == 3 :
            action_type, transaction_id, resource = parts
            if action_type not in ['read', 'write']:
                #Message error
                ...

            else:
                transaction_id = "T"+str(transaction_id)
                scheduler.append((transaction_id, action_type, resource))
        else:
            #Message error
            ...

```

2.3 Implement the Concurrency Control through Timestamp

Now there's the implementation of the rules that characterize concurrency control via timestamp. Remember that :

- **rts(X)**: the highest timestamp among the active transactions that have read X
- **wts(X)**: the highest timestamp among the active transactions that have written X (this coincides with the timestamp of the last transaction that wrote X)
- **wts-c(X)**: the timestamp of the last committed transaction that has written X
- **cb(X)**: a bit (called commit-bit), that is false if the last transaction that wrote X has not committed yet, and true otherwise.

Also remember the effect of the action over this properties whenever they are executed :

- **read action** : set rts(X) as the maximum Timestamp of the transaction that belongs to the read action, and rts(X).

- **write action** :set set wts(X) as the Timestamp of the transaction that belongs to the write action, and set cb(X) to false
- **commit action** : for each element X written by the transaction that perform commit action, set cb(X) = True and the wts_c = timestamp transaction
- **rollback action** : for each element X written by the transaction that perform rollback, set wts(X) = wts-c(X) and cb(X) = True i.e. se to the timestamp of the last transaction that write the resources and has surely commit.

Here is the implementation of the **rule** related to the **read**.

```python
    if action_type == "read":
        if (transaction_ts >= resource_info[resource_index].wts):
            if (resource_info[resource_index].cb == True) or (transaction_ts  ==
            ↪ resource_info[resource_index].wts):
                ...
                resource_info[resource_index].rts = max(transaction_ts,
                ↪ resource_info[resource_index].rts)
            else:
                ...
                #add the action that generate waiting into deadlock_list_detector
                deadlock_detector.append(( transaction , action_type, resource ))
                transaction_info[transaction_index].waiting = True
                ignored_actions.append(elem)
                self.check_deadlock(( transaction , action_type, resource ))

        else:
            ...
            #add the transaction that need to be rollbacked
            rollback_transaction.append(transaction)
            transaction_info[transaction_index].rollback = True
            self.rollback(transaction_ts) #execute rollback
```

Here is the implementation of the **rule** related to the **write**.

```python
    if action_type == "write":
        if (transaction_ts >= resource_info[resource_index].rts) and (transaction_ts
        ↪ >= resource_info[resource_index].wts):
            if resource_info[resource_index].cb == True:
                resource_info[resource_index].wts = transaction_ts
                resource_info[resource_index].cb = False
                ...

            else:
                ...

                #add the action that generate waiting into deadlock_list_detector
                deadlock_detector.append(( transaction , action_type, resource ))
                transaction_info[transaction_index].waiting = True
                ignored_actions.append(( transaction , action_type, resource ))
                self.check_deadlock(( transaction , action_type, resource ))

        elif (transaction_ts >= resource_info[resource_index].rts) and (transaction_ts
        ↪ < resource_info[resource_index].wts):
            if resource_info[resource_index].cb == True:
                # Ignore the action
                ...
            else:
                ...
                #add the action that generate waiting into deadlock_list_detector
                deadlock_detector.append(( transaction , action_type, resource ))
                transaction_info[transaction_index].waiting = True
                ignored_actions.append(( transaction , action_type, resource ))
                self.check_deadlock(( transaction , action_type, resource ))
        else:
            ...
            #add the transaction that need to be rollbacked
            rollback_transaction.append(transaction)
            transaction_info[transaction_index].rollback = True
            self.rollback(transaction_ts) # execute rollback
```

Here is the implementation of the behaviour of the scheduler when it encounter the **commit** action.

```python
    if action_type == "commit":
        ...
        # We can check in the array of the resourceinfo objects, if there's resources
        ↪  where the last write have the ts = to the ts of the commit, it means is the
        ↪  last transaction that wrote in this element,
        # so we need to set cb to true
        resource_to_check = None
        for index, elem in enumerate(resource_info):
            if elem.wts == transaction_ts :
                elem.cb = True
                elem.wts_c = transaction_ts
                resource_to_check = elem.name
        # Now we need to check if some other actions are in waiting for this commit.
        self.check_waiting(resource_to_check)
```

The following is the implementation of the function that is checking for some other actions that are in waiting for this commit. It means, that we need to check each transaction that is waiting for cb(X) become to True or for the rollback of the transaction that has the last to write X, allowing them to proceed.

```python
def check_waiting(self,resource_to_check):
    global deadlock_detector

    # Get the Transaction that is in waiting list in which the resource is set to True
    transaction = None
    for elem in deadlock_detector:
        resource = elem[2]
        if resource == resource_to_check:
            transaction = elem[0]

    if transaction is not None :
        # Create a new list that contain all the tuple of the Transaction in the
        ↪  waiting list .
        new_schedule = [tupla for tupla in ignored_actions if transaction in tupla]
        # Remove the element in the deadlock_list because now it is processed
        deadlock_detector = [tupla for tupla in deadlock_detector if resource_to_check
        ↪  not in tupla]

        # Remove waiting status from the transaction
        for index, elem in enumerate(transaction_info):
            if elem.name == transaction:
                elem.waiting = False

        # Process the related action
        for elem in new_schedule:
            #print(elem)
            self.apply_rules(elem)
            ignored_actions.remove(elem)
```

Here is the implementation of the behaviour of the scheduler when it encounter the **rollback** action.

```
    if action_type == "rollback":
        ...
        # Now we need to check if some other actions are in waiting for this commit.
        self.rollback(transaction_ts)

```

Essentially we need to check each element X that are written by the Transaction that is going to be rollbacked, and we need to set wts(X) to be wts_c(X). It means that wts(X) will be timestamp of the transaction Tj that wrote X before Ti and has surely committed, so we put also cb(X) to true. Then also in this case we need to check if some other action are waiting for cb(X) to became true so we need to call the check_waiting function explained earlier.

```
def rollback(self,transaction_ts):
    resource_to_check = None
    for index, elem in enumerate(resource_info):
        if elem.wts == transaction_ts :
            elem.wts = elem.wts_c
            elem.cb = True
            resource_to_check = elem.name
            # Now we need to check if some other transaction is in waiting , each time
            ↪  we put to True a new variable
            self.check_waiting(resource_to_check)
```

2.4 Deadlock Management

Remember that the method based on timestamps does not avoid the risk of deadlock. So we need a way in order to detect deadlock in our scheduler. The implementation of the algorithm for handle deadlock problem is based on the technique *"deadlock recognition and solution"*. So basically first we need a way in order to recognize a deadlock and then we need to apply a solution to be able to continue the execution of the program.
In our program we have the possibility to choose if you want to apply a technique for deadlock management or to only have a recognition of the event. This is done through the UI by the interaction with a checkbox.

Figure 2.1: Scheduler Output.

So the behaviour of the deadlock management depends on this checkbox. The recongnition will be the same, changes are found only on the event solution.

2.4.1 Deadlock Event

When a deadlock event occurs, the algorithm checks the behaviour chosen by the users, in fact if we don't need to handle a solution, it will only display a message error, and stop the execution of the program, and giving as output the last status of the variables associated to each resources and the status of transaction. The message error it will be displayed through the GUI as a pop-up message. Here is the code for handling that.

```
    if flag_deadlock_solution == True:
        # Apply solution
    else:
        self.show_error_popup(message_error)
```

This is how the pop-up looks like when an event deadlock occur:

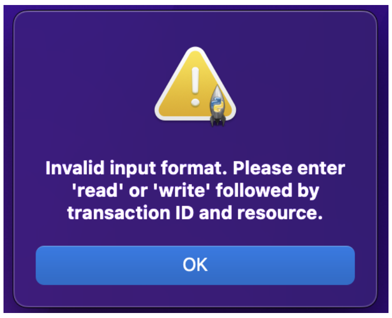

Invalid input format. Please enter 'read' or 'write' followed by transaction ID and resource.

Figure 2.2: Scheduler Output.

If we need to handle the behaviour with solution we need to check the priority (assign it in some way), build a new scheduler and reapply the strategy.
For both cases, the algorithm sets a useful deadlock flag in order to guide the algorithm's execution flow to the correct behavior.

2.4.2 Deadlock Recognition

We essentially make use of our **deadlock_detector list**, in fact, whenever an action generates a waiting state, that is, it puts itself on hold for a commit or rollback to occur that will change the status of the commit variable cb(X) related to the same resource, it is added to this list. Next we go to check if the **Timestamp** of the last transaction that wrote the same element that the current action attempts to use is not inside the **deadlock_detector list**. If it is true, it means that , the current action that is waiting for the last transaction that wrote to that element perform a commit or rollback, but if it is inside this list it means that it is waiting for another transaction commit or rollback. So in this case you check if the last one is not in waiting of the current transaction. If this occurs we have a deadlock, otherwise we proceed with execution of the code. Here is the code that handle this situation :

```python
def check_deadlock(self,elem):
    #Invoked whenever an action is added to the deadlock_list
    transaction_w, action_w, resource_w = elem[0],elem[1],elem[2]
    # Get the transaction ID in which the current action is waiting for
    transactionID_in_conflit = None
    for index, elem in enumerate(resource_info):
        if elem.name == resource_w:
            transactionID_in_conflit = elem.wts
            resource_index = index
            break
    if transactionID_in_conflit != None :
        transactionID_in_conflit = "T"+str(transactionID_in_conflit)
        if(any(transactionID_in_conflit in tupla for tupla in deadlock_detector)):
            transactionInDeadLockList_ID = transactionID_in_conflit
            if(any(transactionInDeadLockList_ID in elem for elem in
            ↪  deadlock_detector)):
                self.setDeadLock()
                if flag_deadlock_solution == True:
                    self.deadlock_solution(transactionInDeadLockList_ID,transaction_w)
                else:
                    self.show_error_popup(message_error)
```

2.4.3 Deadlock Solution

Since we use the technique *deadlock recognition and solution* the solution part is based to the fact that each transaction have a priority number, and when a deadlock event occur, you need to kill the transaction with the lowest priority among all the transaction involved in the event. Then in order to achieve that in the algorithm we simply generate a random number over the different transaction, but it is generated in a way that it can't create duplicate so we are sure that all the transaction must have different numbers of priority. We then rebuild the scheduler without the transaction chosen to be killed and recall the concurrency control strategy over this new scheduler. Here is the code that is used for this purpose:

```python
# For each transaction generate ad assign a priority number.
priority_dictionary = {}
generated_numbers = set()

list_transaction = set([tupla[0] for tupla in scheduler])
for elem in list_transaction:
    priority_dictionary[elem] = random.randint(1, 100)
    while True:
        random_num = random.randint(1, 100)
        # we are sure that the generated number is not equal to others
        if random_num not in generated_numbers:
            generated_numbers.add(random_num)
            priority_dictionary[elem] = random_num
            break

# Remove the text from UI
self.resources_status_text.clear()
self.actions_text.clear()
self.scheduler_output.clear()

if( priority_dictionary[transactionID_1] > priority_dictionary[transactionID_2] ):
    # print("Kill transaction :",trans1)
    deadlock_solution = [tupla for tupla in scheduler if transactionID_1 not in tupla]
    deadlock_s_exec = True
    self.display_scheduler(deadlock_solution)
    self.apply_timestamp(deadlock_solution)
else:
    # print("Kill transaction :",transactionID_2)
    deadlock_solution = [tupla for tupla in scheduler if transactionID_2 not in tupla]
    deadlock_s_exec = True
    self.display_scheduler(deadlock_solution)
    self.apply_timestamp(deadlock_solution)
```

Implementation of the GUI

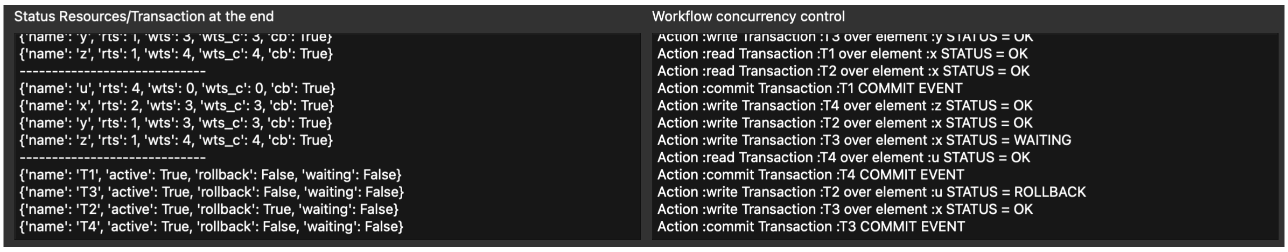
3.1 Introduction

The implementation of the GUI was done through the PyQt, a set of cross-platform C++ libraries that implement high-level APIs for accessing many aspects of modern desktop and mobile systems. These include some features like location and positioning services, multimedia, NFC and Bluetooth connectivity, a Chromium based web browser, as well as traditional UI development. There if you want to go in details i will leave you the link of the **official documentation**.

3.2 How to display the data

Concurrency Results

When you execute the concurrency control via timestamp, this two columns are filled with their dedicated data. The first column, on the left shows the final status of the variable related to each resources at each step of the execution of the actions and the final status of the transaction. Whereas in the second column, each time an action is processed its execution is written and displayed.


Status Resources/Transaction at the end
Workflow concurrency control

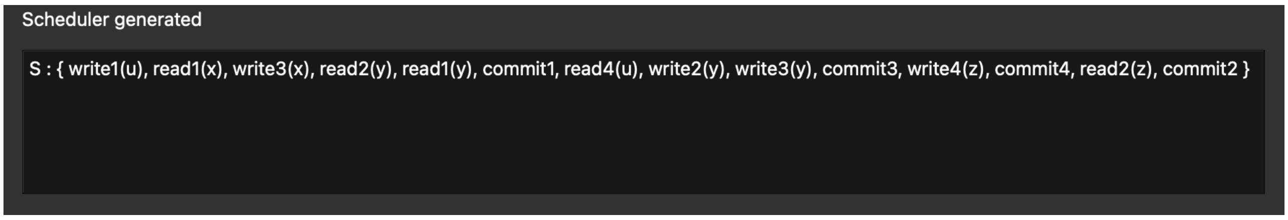
```
{'name': 'y', 'rts': 1, 'wts': 3, 'wts_c': 3, 'cb': True}
{'name': 'z', 'rts': 1, 'wts': 4, 'wts_c': 4, 'cb': True}
----------------------------
{'name': 'u', 'rts': 4, 'wts': 0, 'wts_c': 0, 'cb': True}
{'name': 'x', 'rts': 2, 'wts': 3, 'wts_c': 3, 'cb': True}
{'name': 'y', 'rts': 1, 'wts': 3, 'wts_c': 3, 'cb': True}
{'name': 'z', 'rts': 1, 'wts': 4, 'wts_c': 4, 'cb': True}
----------------------------
{'name': 'T1', 'active': True, 'rollback': False, 'waiting': False}
{'name': 'T3', 'active': True, 'rollback': False, 'waiting': False}
{'name': 'T2', 'active': True, 'rollback': True, 'waiting': False}
{'name': 'T4', 'active': True, 'rollback': False, 'waiting': False}
```

```
Action :write Transaction :T3 over element :y STATUS = OK
Action :read Transaction :T1 over element :x STATUS = OK
Action :read Transaction :T2 over element :x STATUS = OK
Action :commit Transaction :T1 COMMIT EVENT
Action :write Transaction :T4 over element :z STATUS = OK
Action :write Transaction :T2 over element :x STATUS = OK
Action :write Transaction :T3 over element :x STATUS = WAITING
Action :read Transaction :T4 over element :u STATUS = OK
Action :commit Transaction :T4 COMMIT EVENT
Action :write Transaction :T2 over element :u STATUS = ROLLBACK
Action :write Transaction :T3 over element :x STATUS = OK
Action :commit Transaction :T3 COMMIT EVENT
```

Figure 3.1: Output Concurrency control through Timestamp.

Scheduler

Each time you add correctly an action the following scheduler is generated and displayed over the GUI.



```
S : { write1(u), read1(x), write3(x), read2(y), read1(y), commit1, read4(u), write2(y), write3(y), commit3, write4(z), commit4, read2(z), commit2 }
```

Figure 3.2: Scheduler Output.

Here is the code for the output of the scheduler

```python
def display_scheduler(self,input_scheduler):
    scheduler_for_output = list(input_scheduler)
    scheduler_str = "S : {"
    for action in scheduler_for_output:
        transaction_id, action_type, resource = action
        if action_type in ["read","write"]:
            scheduler_str += f" {action_type}{transaction_id[1]}({resource}),"
        else :
            scheduler_str += f" {action_type}{transaction_id[1]},"
    scheduler_str = scheduler_str.rstrip(',')  # Remove the trailing comma
    scheduler_str += " }"
    #Display over the UI
    self.scheduler_output.setText(scheduler_str)
```

3.3 Error Handling

Since we use GUI, all the error are handled by displaying some message error via a pop-up like this one, once the error is occurred, all the needed variable used for the algorithm will be reset to default in order to avoid some problem.

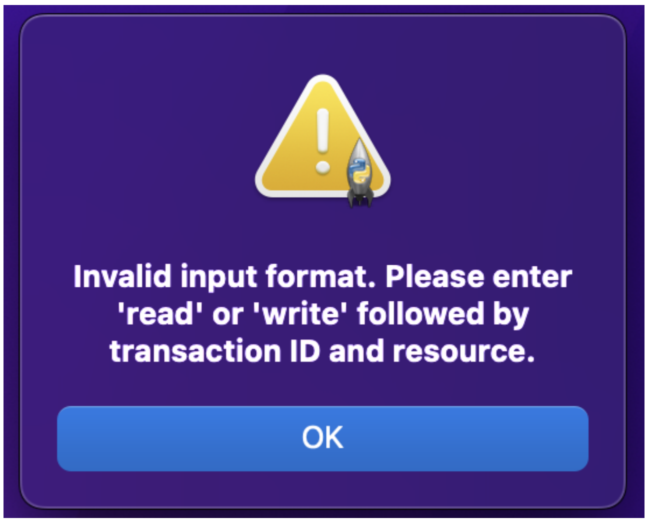

Invalid input format. Please enter 'read' or 'write' followed by transaction ID and resource.

Figure 3.3: ErrorMessage PopUp.

Here is the piece of code for handling that :

```python
# Definition
def show_error_popup(self, message):
    error_popup = QMessageBox()
    error_popup.setWindowTitle("Error")
    error_popup.setIcon(QMessageBox.Icon.Critical)
    error_popup.setText(message)
    error_popup.exec()
# Usage
self.show_error_popup("Message Error")
```

3.4 GUI Visualization

Here is the full visualization of the GUI developed for the purpose of this project

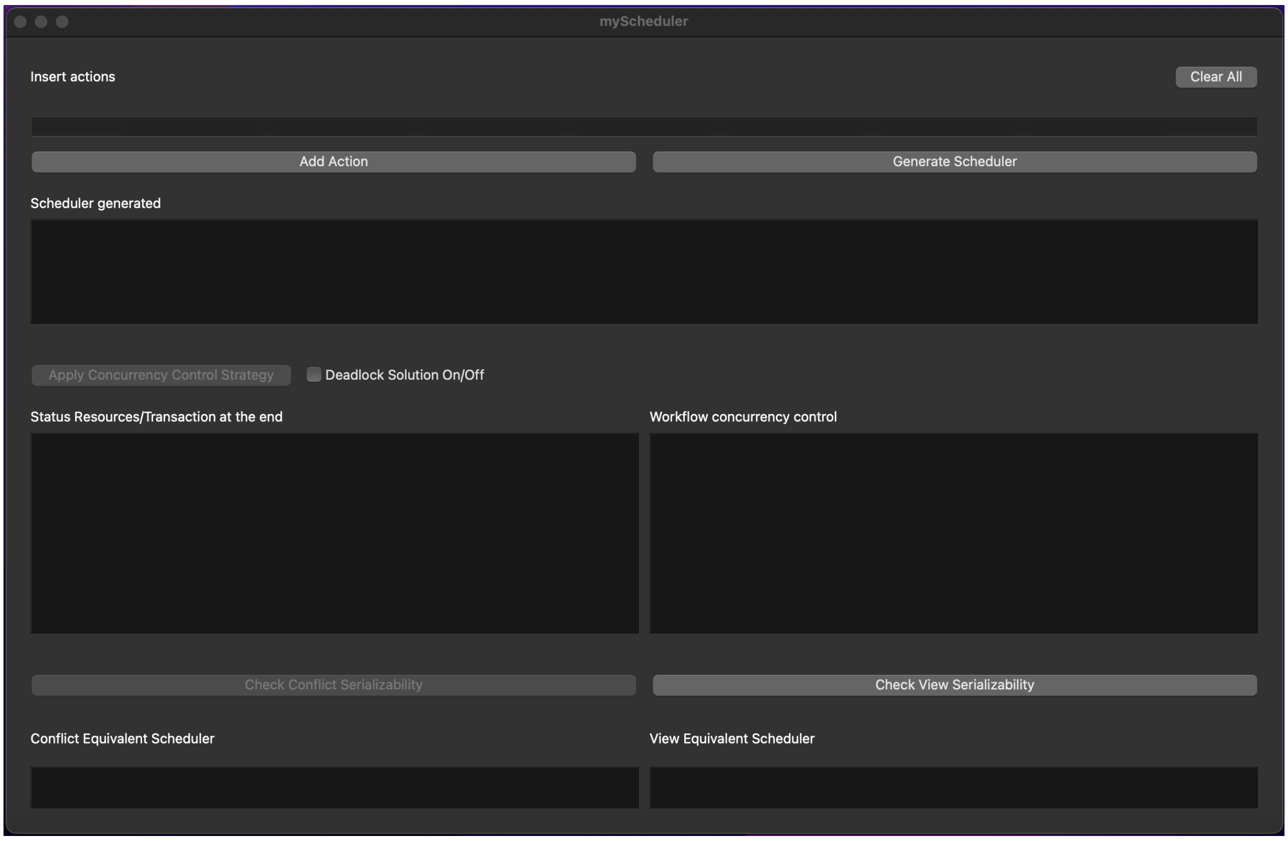

myScheduler
Insert actions
Clear All
Add Action
Generate Scheduler
Scheduler generated
Apply Concurrency Control Strategy
Deadlock Solution On/Off
Status Resources/Transaction at the end
Workflow concurrency control
Check Conflict Serializability
Check View Serializability
Conflict Equivalent Scheduler
View Equivalent Scheduler

Figure 3.4: Final GUI.

Test and Execution

4.1 Setup & Start

If you want to test the project, you can download it from the **repository on github**

Once you download it, you need execute the following command in order to install all the dependencies needed:

```
#!/bin/bash
pip3 install -r requirements.txt
```

Once you install the dependencies you need to run the program

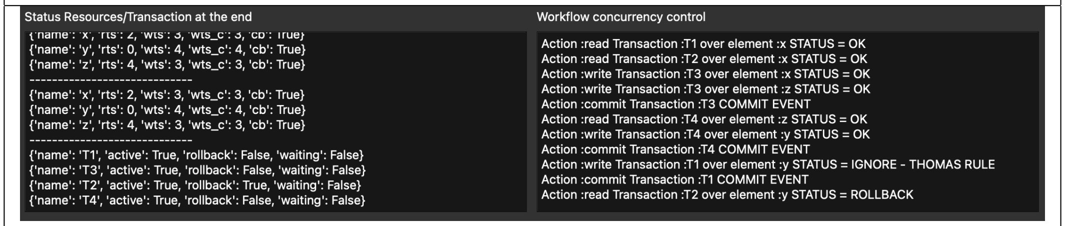
```
#!/bin/bash
python3 my_scheduler_gui.py
```

Then the dedicated window appear on the screen.

4.2 Sample Testing

In order to be able to verify the correctness of the algorithm and execution of these, we took as example references some schedulers provided by the professor during the lectures. In this way we can double check the outcome of the applied strategy. So below I have reported some tests that were performed to validate the implemented concurrency control strategy via timestamp. All the following test are made on a scheduler in which we assume, that,initially, $rts(\alpha) = wts(\alpha) = 0, andwtsc(\alpha) = cb(\alpha) = true$ for each element α of the database and assuming that the subscript of each action denotes the timestamp of the transaction executing such action.

$$Scheduler = [\ r1(x)r2(x)w3(x)w3(z)c3r4(z)w4(y)c4w1(y)c1r2(y)c2\ ]$$



Status Resources/Transaction at the end

```
{'name': 'x', 'rts': 2, 'wts': 3, 'wts_c': 3, 'cb': True}
{'name': 'y', 'rts': 0, 'wts': 4, 'wts_c': 4, 'cb': True}
{'name': 'z', 'rts': 4, 'wts': 3, 'wts_c': 3, 'cb': True}
-----------------------------
{'name': 'x', 'rts': 2, 'wts': 3, 'wts_c': 3, 'cb': True}
{'name': 'y', 'rts': 0, 'wts': 4, 'wts_c': 4, 'cb': True}
{'name': 'z', 'rts': 4, 'wts': 3, 'wts_c': 3, 'cb': True}
-----------------------------
{'name': 'T1', 'active': True, 'rollback': False, 'waiting': False}
{'name': 'T3', 'active': True, 'rollback': False, 'waiting': False}
{'name': 'T2', 'active': True, 'rollback': True, 'waiting': False}
{'name': 'T4', 'active': True, 'rollback': False, 'waiting': False}
```

Workflow concurrency control

```
Action :read Transaction :T1 over element :x STATUS = OK
Action :read Transaction :T2 over element :x STATUS = OK
Action :write Transaction :T3 over element :x STATUS = OK
Action :write Transaction :T3 over element :z STATUS = OK
Action :commit Transaction :T3 COMMIT EVENT
Action :read Transaction :T4 over element :z STATUS = OK
Action :write Transaction :T4 over element :y STATUS = OK
Action :commit Transaction :T4 COMMIT EVENT
Action :write Transaction :T1 over element :y STATUS = IGNORE - THOMAS RULE
Action :commit Transaction :T1 COMMIT EVENT
Action :read Transaction :T2 over element :y STATUS = ROLLBACK
```

$r_1(x) \rightarrow \text{OK} \rightarrow \text{rts}(x) = 1$

$r_2(x) \rightarrow \text{OK} \rightarrow \text{rts}(x) = 2$

$w_3(x) \rightarrow \text{OK} \rightarrow \text{wts}(x) = 3,\ \text{cb}(x) = \text{false}$

$w_3(z) \rightarrow \text{OK} \rightarrow \text{wts}(z) = 3,\ \text{cb}(z) = \text{false}$

$c_3 \rightarrow \text{OK} \rightarrow \text{wts-c}(x) = 3,\ \text{wts-c}(z) = 3,\ \text{cb}(x) = \text{true},\ \text{cb}(z) = \text{true}$

$r_4(z) \rightarrow \text{OK} \rightarrow \text{rts}(z) = 4$

$w_4(y) \rightarrow \text{OK} \rightarrow \text{wts}(y) = 4,\ \text{cb}(y) = \text{false}$

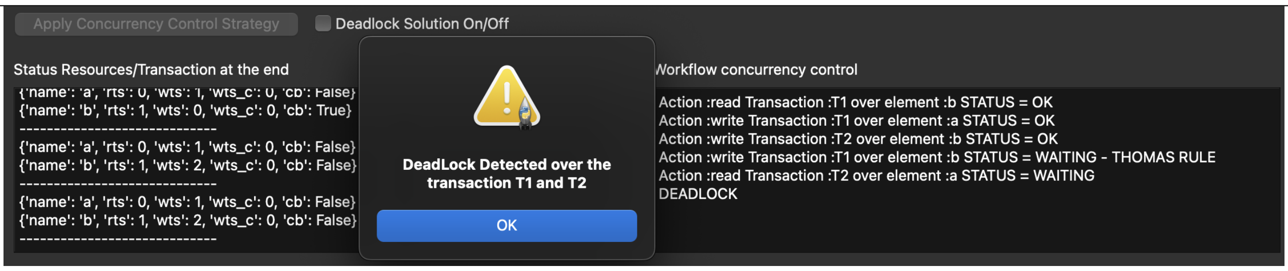
$c_4 \rightarrow \text{OK} \rightarrow \text{wts-c}(y) = 4,\ \text{cb}(y) = \text{true}$

$w_1(y) \rightarrow \text{OK (Thomas rule)}$

$c_1 \rightarrow \text{OK}$

$r_2(y) \rightarrow \text{read too late} \rightarrow T_2 \text{ aborted}$

$$Scheduler = [\ r1(B)w1(A)w2(B)w1(B)r2(A)\ ]$$



Status Resources/Transaction at the end

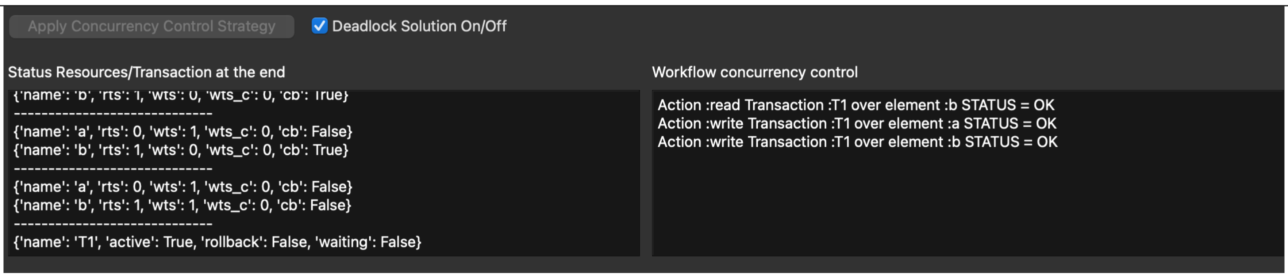
```
{'name': 'a', 'rts': 0, 'wts': 1, 'wts_c': 0, 'cb': False}
{'name': 'b', 'rts': 1, 'wts': 0, 'wts_c': 0, 'cb': True}
-----------------------------
{'name': 'a', 'rts': 0, 'wts': 1, 'wts_c': 0, 'cb': False}
{'name': 'b', 'rts': 1, 'wts': 2, 'wts_c': 0, 'cb': False}
-----------------------------
{'name': 'a', 'rts': 0, 'wts': 1, 'wts_c': 0, 'cb': False}
{'name': 'b', 'rts': 1, 'wts': 2, 'wts_c': 0, 'cb': False}
-----------------------------
```

DeadLock Detected over the transaction T1 and T2

Workflow concurrency control

```
Action :read Transaction :T1 over element :b STATUS = OK
Action :write Transaction :T1 over element :a STATUS = OK
Action :write Transaction :T2 over element :b STATUS = OK
Action :write Transaction :T1 over element :b STATUS = WAITING - THOMAS RULE
Action :read Transaction :T2 over element :a STATUS = WAITING
DEADLOCK
```

r1(B) → ok → ts(T1)=1, rts(B)=1 -- because ts(T1)>= wts(B), cb(B)=true and rts(B)=max(ts(T1),rts(B))
w1(A) → ok → wts(A)=1 and cb(A)=false -- because ts(T1)>= wts(A), cb(A)=true and ts(T1)>= rts(A)
w2(B) → ok → ts(T2)=3, wts(B)=3 and cb(B)=false -- because ts(T2)>= wts(B), cb(B)=true and ts(T2)>= rts(B)
w1(B) → T1 waiting for the commit or rollback of T2 -- because cb(B)=false, ts(T1) = rts(B) and ts(T1)<wts(B)
r2(A) → T2 waiting for the commit or rollback of T1 because cb(A)=false and ts(T2)>wts(A) ==> **DEADLOCK!**


Apply Concurrency Control Strategy
✓ Deadlock Solution On/Off

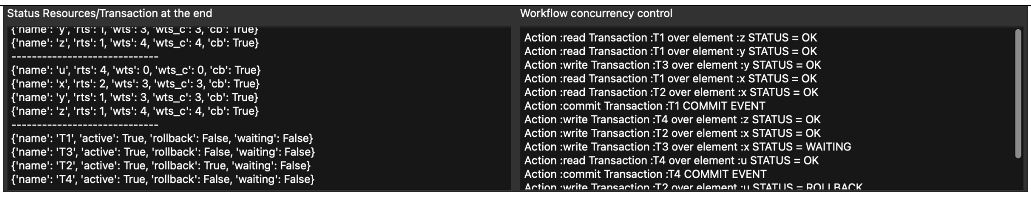
Status Resources/Transaction at the end

```
{'name': 'b', 'rts': 1, 'wts': 0, 'wts_c': 0, 'cb': True}
-----------------------------
{'name': 'a', 'rts': 0, 'wts': 1, 'wts_c': 0, 'cb': False}
{'name': 'b', 'rts': 1, 'wts': 0, 'wts_c': 0, 'cb': True}
-----------------------------
{'name': 'a', 'rts': 0, 'wts': 1, 'wts_c': 0, 'cb': False}
{'name': 'b', 'rts': 1, 'wts': 1, 'wts_c': 0, 'cb': False}
-----------------------------
{'name': 'T1', 'active': True, 'rollback': False, 'waiting': False}
```

Workflow concurrency control

```
Action :read Transaction :T1 over element :b STATUS = OK
Action :write Transaction :T1 over element :a STATUS = OK
Action :write Transaction :T1 over element :b STATUS = OK
```

$$Scheduler = [\ r1(z)r1(y)w3(y)r1(x)r2(x)c1w4(z)w2(x)w3(x)c3r4(u)c4w2(u)c2.\ ]$$



Status Resources/Transaction at the end

```
{'name': 'y', 'rts': 1, 'wts': 3, 'wts_c': 3, 'cb': True}
{'name': 'z', 'rts': 1, 'wts': 4, 'wts_c': 4, 'cb': True}
-----------------------------
{'name': 'u', 'rts': 4, 'wts': 0, 'wts_c': 0, 'cb': True}
{'name': 'x', 'rts': 2, 'wts': 3, 'wts_c': 3, 'cb': True}
{'name': 'y', 'rts': 1, 'wts': 3, 'wts_c': 3, 'cb': True}
{'name': 'z', 'rts': 1, 'wts': 4, 'wts_c': 4, 'cb': True}
-----------------------------
{'name': 'T1', 'active': True, 'rollback': False, 'waiting': False}
{'name': 'T3', 'active': True, 'rollback': False, 'waiting': False}
{'name': 'T2', 'active': True, 'rollback': True, 'waiting': False}
{'name': 'T4', 'active': True, 'rollback': False, 'waiting': False}
```

Workflow concurrency control

```
Action :read Transaction :T1 over element :z STATUS = OK
Action :read Transaction :T1 over element :y STATUS = OK
Action :write Transaction :T3 over element :y STATUS = OK
Action :read Transaction :T1 over element :x STATUS = OK
Action :read Transaction :T2 over element :x STATUS = OK
Action :commit Transaction :T1 COMMIT EVENT
Action :write Transaction :T4 over element :z STATUS = OK
Action :write Transaction :T2 over element :x STATUS = OK
Action :write Transaction :T3 over element :x STATUS = WAITING
Action :read Transaction :T4 over element :u STATUS = OK
Action :commit Transaction :T4 COMMIT EVENT
Action :write Transaction :T2 over element :u STATUS = ROLLBACK
```

$r_1(z) \rightarrow$ OK,	rts(z)=1
$r_1(y) \rightarrow$ OK,	rts(y)=1
$w_3(y) \rightarrow$ OK,	wts(y)=3, cb(y)=`false`
$r_1(x) \rightarrow$ OK,	rts(x)=1
$r_2(x) \rightarrow$ OK,	rts(x)=2
$c_1 \rightarrow$ OK,	
$w_4(z) \rightarrow$ OK,	wts(z)=4
$w_2(x) \rightarrow$ OK,	wts(x)=2, cb(x)=`false`
$w_3(x) \rightarrow$ OK,	transaction 3 suspended
$r_4(u) \rightarrow$ OK,	rts(u)=4
$c_4 \rightarrow$ OK,	wts-c(z)=4, cb(z)=`true`
$w_2(u) \rightarrow$ write too late,	transaction 2 rollbacks
$w_3(x) \rightarrow$ OK,	wts(x)=3
$c_3 \rightarrow$ OK,	wts-c(x)=3, cb(x)=`true`

Extra features

5.1 Conflict Serializable

We also implement the possibility to check whether the given scheduler is conflict serializable or not. We remember that in order to check if a schedule is conflict, we need to analyze the precedence graph (also called conflict graph) associated to a schedule. Given a schedule S on $T1, ..., Tn$, the precedence graph $P(S)$ associated to S is defined as follows:

- the nodes of $P(S)$ are the transactions $T1, ..., Tn$, of S
- the edges E of $P(S)$ are as follows: the edge $Ti \rightarrow Tj$ is in E if and only if there exists two actions $Pi(A), Qj(A)$ of different transactions Ti and Tj in S operating on the same object A such that
  - $Pi(A) < Qj(A)$ (i.e., $Pi(A)$ appears before $Qj(A)$ in S)
  - at least one between $Pi(A)andQj(A)$ is a write operation

Then we build the precedente graph just for this theoreme : A schedule S is conflict- serializable if and only if the precedence graph P(S) associated to S is acyclic. So if the precedence graph is acyclic the schedule is conflict serializable, otherwise not.

5.1.1 Precedent Graph

So in order to check if the given scheduler is conflict first of all we need to build the precedente graph, and since the graph is represented through a list of adjacencies we need to build a specific implementation. Here there's the implementation :

```python
    def check_serializability(self):
        # Set our scheduler well.
        scheduler_for_conflict = [elem for elem in scheduler if "commit" not in elem
        ↪  and "rollback" not in elem ]
        # Init precedent graph
        precedence_graph = {}
        # Scan the scheduler in order to find the conflict pair for adding edges
        for i in range(len(scheduler_for_conflict)):
            transaction_i, action_i, element_i = scheduler_for_conflict[i]
            if transaction_i not in precedence_graph:
                precedence_graph[transaction_i] = set()

            for j in range(i + 1, len(scheduler_for_conflict)):
                transaction_j, action_j, element_j = scheduler_for_conflict[j]
                # check conflict pair in order to add edges
                if (transaction_i != transaction_j and element_i == element_j and
                ↪  (action_i == "write" or action_j =="write")):
                    precedence_graph[transaction_i].add(transaction_j)

        # Check if the graph is acyclic
        visited = {transaction: False for transaction in precedence_graph}
        rec_stack = {transaction: False for transaction in precedence_graph}
        topological_order = []

        for transaction in precedence_graph:
            if not visited[transaction]:
                if has_cycle(transaction, visited, rec_stack,topological_order):
                    return False, [] # Schedule isn't conflict-serializable

        return True,topological_order  # Schedule is conflict serializable
```

5.1.2 DFS For detecting cycle

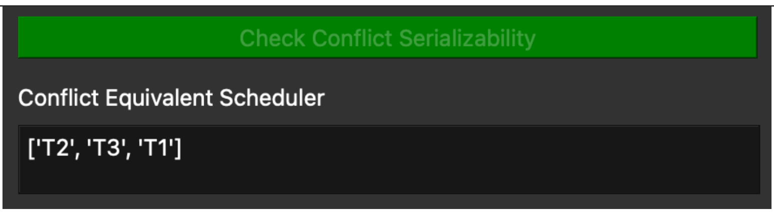
In order to perform the check on the precedence graph to determine whether it has cycles or not is done through the use of an algorithm based on the DFS algorithm. Depth-first search (DFS) is an algorithm for traversing or searching graph data structures and since we have a graph it is usefull for our purpose.

```python
# Function for checking if there'are cycle (use DFS search)
def has_cycle(node, visited, rec_stack, order):
    visited[node] = True
    rec_stack[node] = True

    for neighbor in precedence_graph.get(node, []):
        if not visited[neighbor]:
            if has_cycle(neighbor, visited, rec_stack,order):
                return True
        elif rec_stack[neighbor]:
            return True

    rec_stack[node] = False
    order.append(node)
    return False
```

5.1.3 Conflict Visualization

Then when we apply the function for checking the conflict serialization, we have that the function returns us two values, which correspond to a flag and a list. The flag simply tells us whether the schedule is conflict or non conflict serializable (True/False) while the list is the list of any topological order present. We will only need these two values to output the result of the operation.

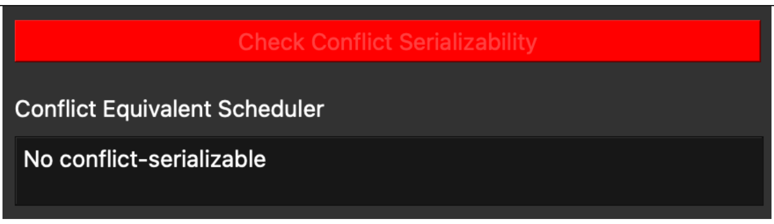
$Scheduler = [\ r1(a)r3(c)w3(b)r2(a)w1(b)r2(c)w3(c)r3(a)w2(d)\ ]$



Conflict Equivalent Scheduler

['T2', 'T3', 'T1']

$Scheduler = [\ r1(a)r3(c)w3(b)r2(a)w1(b)w1(a)w2(a)r1(c)w3(c)r3(a)r2(d)\ ]$



Conflict Equivalent Scheduler

No conflict-serializable

5.2 View Serializable

We remind that a schedule S on $T1, ..., Tn$ is view-serializable if there exists a serial schedule S' on $T1, ..., Tn$ that is view-equivalent to S. View equivalent means that the serial schedule S' have the same set of READ-FROM FINAL-WRITE.

- In a schedule S, we say that $ri(x)$ READS-FROM $wj(x)$ if $wj(x)$ preceeds $ri(x)$ in S, and there is no action of type $wk(x)$ between $wj(x)$ and $ri(x)$. The READS-FROM relation associated to S is the set of pair $< ri(x), wj(x) >$ such that $ri(x)$ READS-FROM $wj(x)$.
- In a schedule S, we say that wi(x) is a FINAL-WRITE if wi(x) is the last write action on x in S. The FINAL-WRITE set associated to S is the set of actions wi(x) such that wi(x) is the last write action on x in S.

5.2.1 Find READ-FROM & FINAL-WRITE sets

As a first step, we need to write a function that given a scheduler calculates over it the set of READFROMs and FINALWRITE. To do this, we simply scan the scheduler and differentiate the behavior in case we encounter a read or write action. In the case where it is a read we iterate the scheduler going backwards ,until we have the initial element,in order to be able to find an action that matches the read-from pair definition. So in the list of READFROMs we save the as a tupla the action pair in relation. In the case where it is a write we simply add it to our final-write set, if it is already there we update the transaction , otherwise a new element is added.

```
    def extract_read_from_final_write(self,schedule):
        read_from = []
        final_write = []
        for i, (transaction_id, action_name, resource) in enumerate(schedule):
            if action_name == "read":
                j = i - 1
                for j in range(i - 1, -1, -1):
                    #print("Nostra azione : ",(transaction_id, action_name, resource))
                    #print("Azione dello scheduler
                    ↪  :",(schedule[j][0],schedule[j][1],schedule[j][2]))
                    if (schedule[j][2] == resource and schedule[j][1] == "write" and
                    ↪  schedule[j][0] != transaction_id ):

                        ↪  read_from.append((transaction_id,action_name,resource,schedule[j][0],s
            if action_name == "write":
                j = i - 1
                if(schedule[j][2] == resource):
                    #create a new list without the tupla with the resource ==
                    ↪  schedule[j][2]
                    final_write = [(resource, transaction_id) for resource, _ in
                    ↪  final_write if resource != schedule[j][2]]
                    #add element
                    final_write.append((resource,schedule[j][0]))
                else:
                    final_write.append((resource,transaction_id))
        return read_from, final_write
```

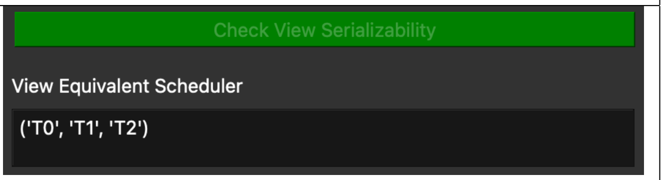
5.2.2 Check View-Equivalence

Now the only thing the algorithm has to do is to check if there is a serial program that has the same set of READFROM and FINALWRITE. The problem with this part of the code is how we are able to create the various permutations of the transactions to create a serial schedule: for example, if I have T1 and T2 transactions, the possible permutations are T1 - T2 and T2 - T1. In this case we rely on itertools.permutation, which allows us to have all possible orderings without any repeated elements. Then for each available permutation we are going to construct the relevant serial schedule, and then go on to compute the dedicated sets. Obviously if they are equal to the set of the original schedule, the original schedule is view-serializable and it will be printed in the output.

```python
    def check_view_serializability(self):
        # Set our scheduler well.
        scheduler_for_conflict = [elem for elem in scheduler if "commit" not in elem and "rollback" not in elem ]
        # Calculate the set of the current scheduler
        read_from, final_write = self.extract_read_from_final_write(scheduler_for_conflict)
        # Create a set without duplicate of all the transaction_id in the scheduler
        list_transaction = set([tupla[0] for tupla in scheduler_for_conflict])
        # Generate all permutation possible with the different transaction_id present in the schedule
        all_permutations = itertools.permutations(list_transaction)
        # iterate over all the permutation, we will stop at the first occurency
        for permuted_transactions in all_permutations:
            permuted_schedule = []
            # Create new scheduler in base of the current permutation of transaction
            for t_id in permuted_transactions:
                t_actions = [action for action in scheduler_for_conflict if action[0] == t_id]
                permuted_schedule.extend(t_actions)

            #calculate the set of the read from and final write serial scheduler obtained from the combination of permutation
            permuted_read_from, permuted_final_write = self.extract_read_from_final_write(permuted_schedule)

            read_from_to_check = set(read_from)
            final_write_to_check = set(final_write)
            read_from_permuted = set(permuted_read_from)
            final_Write_permuted = set(final_write)

            if read_from_to_check == read_from_permuted and final_write_to_check == final_Write_permuted:
                # View
                break
            else:
                #Not view

```

5.2.3 View Visualization

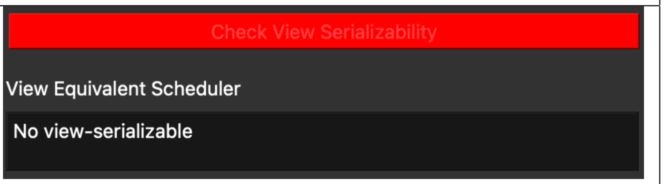
$$Scheduler = [\ w0(x)r1(x)w1(x)w2(z)w1(z)\ ]$$



View Equivalent Scheduler

('T0', 'T1', 'T2')

$$Scheduler = [\ w1(x)r2(x)w2(y)r1(y)\ ]$$



View Equivalent Scheduler

No view-serializable