CS132: Software Engineering

Validation

Banking System

Group 15

Author: 贾舜康 祝宇航 王芸飞

Catalogue

1. Unit Test

This section provides information of unit tests we made for every main function with statement coverage, branch coverage and condition coverage criteria. Testing cases with runnable test functions are provided in every test, you can find in corresponding files.

1.1 Server (Backend of the system)

1.1.1 Process request: create account

```
def process_request(self, address: str, message: str):
    cleaned_string = message.replace("(", "").replace(")", "")
    parts = re.split('[@#]', cleaned_string)
    print(parts)
    # parts = message.split('@')
    command = parts[0]
    params = parts[1:]

    if command == "create_account":
        account_id = params[0]
        print(len(account_id))
        password = params[1]
        print(len(password))
        if not len(account_id) == 10:···
        if self.account_exists(account_id):
            self.send_string(address, "failed@A@Account already exists")
            return
        if not len(password) == 6:
            self.send_string(address, "failed@B@Password must consist of 6 digits")
            return
        self.create_account(account_id, password)
        self.send_string(address, "success@Account created successfully")
```

- Coverage Criteria: Branch Coverage

-TestFunction:src/test/unitttest_server.py/test_001,test_002,test_003,test_004

- Test Case

Test Case	T1.1.1.1	T1.1.1.2	T1.1.1.3	T1.1.1.4
Input	"create_account@1234567890#000000"	“create_account@145670#000000”	“create_account@5876476543#00000”	“create_account@2024888888#000000”
Coverage Item	Tcover1.1.1.1	Tcover1.1.1.2	Tcover1.1.1.3	Tcover1.1.1.4
State	This account does not exist	-----	-----	This account already exist
Expected Output	"success@Account created successfully"	"failed@A@Account ID must consist of 10 digits"	"failed@B@Password must consist of 6 digits"	“failed@A@Account already exists”
Test Result	Passed	Passed	Passed	Passed

- Test Coverage: 4/4 = 100%

1.1.2 Process request: log in

```
elif command == "log_in":
    account_id = params[0]
    password = params[1]
    if not self.account_exists(account_id):
        self.send_string(address, "failed@A@Invalid account ID")
        return
    if not self.verify_password(account_id, password):
        self.send_string(address, "failed@B@Invalid password")
        return
    self.send_string(address, "success@Log in successful")
```

- Coverage Criteria: Branch Coverage
- TestFunction:src/test/unitttest_server.py/test_005,test_006,test_007
- Test Case

Test Case	T1.1.2.1	T1.1.2.2	T1.1.2.3
Input	log_in@2024888888#000000	log_in@2024888889#000000	log_in@2024888888#000001
Coverage Item	Tcover1.1.2.1	Tcover1.1.2.2	Tcover1.1.2.3
State	The account exist	The account does not exist	The account password is 000000
Expected Output	" success@Log in successful"	"failed@A@Invalid account ID"	"failed@B@Invalid password"
Test Result	Passed	Passed	Passed

- Test Coverage: 3/3 = 100%

1.1.3 Process request: Insert card

```
elif command == "insert_card":
    account_id = params[0]
    password = params[1]
    if not self.account_exists(account_id):
        self.send_string(address, "failed@A@Invalid account ID")
        return
    if not self.verify_password(account_id, password):
        self.send_string(address, "failed@B@Invalid password")
        return
    self.send_string(address, "success@Insert card successful")
```

- Coverage Criteria: Branch Coverage

- TestFunction:src/test/unitttest_server.py/test_008,test_009,test_010
- Test Case

Test Case	T1.1.3.1	T1.1.3.2	T1.1.3.3
Input	insert_card@2024888888#000000	insert_card@2024888889#000000	insert_card@2024888888#000001
Coverage Item	Tcover1.1.3.1	Tcover1.1.3.2	Tcover1.1.3.3
State	The account exist	The account does not exist	The account password is 000000
Expected Output	"success@Insert card successful"	"failed@A@Invalid account ID"	"failed@B@Invalid password"
Test Result	Passed	Passed	Passed

- Test Coverage: 3/3 = 100%

1.1.4 Process request: Deposit cash

```
elif command == "deposit_cash":
    account_id = params[0]
    amount = float(params[1])
    if not self.account_exists(account_id):
        self.send_string(address, "failed@Invalid account ID")
        return
    if amount <= 0 or amount > 50000:
        self.send_string(address, "failed@Deposit amount must be between $0.01 and $50000.00")
        return
    starting_balance, ending_balance = self.deposit_cash(account_id, amount)
    self.send_string(address, f"success@${amount:.2f} deposited successfully. Balance: {starting_balance} -> {ending_balance}")
```

- Coverage Criteria: Branch Coverage
- TestFunction:src/test/unitttest_server.py/test_011,test_012,test_013,test_014
- Test Case

Test Case	T1.1.4.1	T1.1.4.2	T1.1.4.3	T1.1.4.4
Input	deposit_cash@2024888888#200	deposit_cash@2024888889#200	deposit_cash@2024888888#50001	deposit_cash@2024888888#-78.90
Coverage Item	Tcover1.1.4.1	Tcover1.1.4.2	Tcover1.1.4.3	Tcover1.1.4.4
State	The account exist	The account does not exist	-----	-----
Expected Output	"success@$200.00 deposited successfully. Balance: 10000.0 -> 10200.0"	"failed@Invalid account ID"	"failed@Deposit amount must be between $0.01 and $50000.00"	“failed@Deposit amount must be between $0.01 and $50000.00”

Test Result	Passed	Passed	Passed	Passed

- Test Coverage: 4/4 = 100%

1.1.5 Process request: Return card

```
elif command == "return_card":
    self.send_string(address, "success@Card returned successfully")
```

- Coverage Criteria: Statement Coverage
- TestFunction:src/test/unitttest_server.py/test_015
- Test Case

Test Case	T1.1.5.1
Input	“return_card”
Coverage Item	Tcover1.1.5.1
State	-----
Expected Output	“success@Card returned successfully”
Test Result	Passed

- Test Coverage: 1/1 = 100%

1.1.6 Process request: Log out

```
elif command == "log_out":
    self.send_string(address, "success@Logged out successfully")
```

- Coverage Criteria: Statement Coverage
- TestFunction:src/test/unitttest_server.py/test_016
- Test Case

Test Case	T1.1.6.1
Input	“log_out”
Coverage Item	Tcover1.1.6.1
State	-----
Expected Output	“success@Logged out successfully”
Test Result	Passed

- Test Coverage: 1/1 = 100%

1.1.7 Process request: Change password

```
elif command == "change_password":
    account_id = params[0]
    new_password = params[1]

    old_password = self.get_password(account_id)
    if new_password == old_password:
        self.send_string(address, "failed@New password cannot be the same as the old password")
        return
    if not len(new_password) == 6 or not new_password.isdigit():
        self.send_string(address, "failed@Password must consist of 6 digits")
        return
    self.change_password(account_id, new_password)
    self.send_string(address, "success@Password changed successfully")
```

- Coverage Criteria: Branch Coverage
- TestFunction:src/test/unitttest_server.py/test_017,test_018,test_019,test_020
- Test Case

Test Case	T1.1.4.1	T1.1.4.2	T1.1.4.3	T1.1.4.4
Input	change_password@2024888888#000000	change_password@2024888888#abcefg	change_password@2024888888#12345	change_password@2024888888#123456
Coverage Item	Tcover1.1.4.1	Tcover1.1.4.2	Tcover1.1.4.3	Tcover1.1.4.4
State	-----	-----	-----	-----
Expected Output	"failed@New password cannot be the same as the old password"	"failed@Password must consist of 6 digits"	"failed@Password must consist of 6 digits"	“success@Password changed successfully”
Test Result	Passed	Passed	Passed	Passed

- Test Coverage: 4/4 = 100%

1.1.8 Process request: Transfer Money

```
elif command == "transfer_money":
    sender_id = params[0]
    receiver_id = params[1]
    amount = float(params[2])

    if sender_id == receiver_id:
        self.send_string(address, "failed@Can't tranfer to your own")
        return

    if not receiver_id.isdigit() or len(receiver_id) != 10:
        self.send_string(address, "failed@Receiver's account ID must consist of 10 digits")
        return

    if not self.account_exists(receiver_id):
        self.send_string(address, "failed@Invalid receiver account ID")
        return

    if amount <= 0 or amount > 50000:
        self.send_string(address, "failed@Transfer amount must be between $0.01 and $50000.00")
        return

    if not self.has_sufficient_balance(sender_id, amount):
        self.send_string(address, "failed@Insufficient account balance for transfer")
        return

    sender_starting_balance, sender_ending_balance, receiver_starting_balance, receiver_ending_balance = self.transfer_money(s
    self.send_string(address, f"success@${amount:.2f} transferred successfully. Sender balance: {sender_starting_balance} -> {
```

- Coverage Criteria: Branch Coverage
- TestFunction:src/test/unitttest_server.py/test_021,test_022,test_023_1,test_023_2,test024_1,test024_2,test_024_3
- Test Case

Test Case	T1.1.8.1	T1.1.8.2	T1.1.8.3	

Input	transfer_money@2024888888#2024000000#5000	transfer_money@2024888888#2024000001#5000	transfer_money@2024888888#2024000000#50000.01	transfer_money@2024888888#2024000000#40000.0
Coverage Item	Tcover1.1.8.1	Tcover1.1.8.2	Tcover1.1.8.3	Tcover1.1.8.7
State	-----	-----	-----	----
Expected Output	"success@$5000.00 transferred successfully. Sender balance: 10200.0 -> 5200.0"	"failed@Invalid receiver account ID"	" failed@Transfer amount must be between $0.01 and $50000.00"	failed@Insufficient account balance for transfer
Test Result	Passed	Passed	Passed	Passed
Test Case	T1.1.8.4	T1.1.8.5	T1.1.8.6	-----
Input	transfer_money@2024888888#2024000000#-78.90	transfer_money@2024888888#20248888888#5000	transfer_money@2024888888#2024#5000	-----
Coverage Item	Tcover1.1.8.4	Tcover1.1.8.5	Tcover1.1.8.3	------
State	-----	-----	-----	----
Expected Output	"failed@Transfer amount must be between $0.01 and $50000.00"	"failed@Can't tranfer to your own"	"failed@Receiver's account ID must consist of 10 digits"	-----
Test Result	Passed	Passed	Passed	

- Test Coverage: 7/7 = 100%

1.1.9 Process request: Withdraw cash

```
elif command == "withdraw_cash":
    account_id = params[0]
    amount = float(params[1])
    if amount <= 0 or amount > 50000:
        self.send_string(address, "failed@Withdrawal amount must be between $0.01 and $50000.00")
        return

    if not self.has_sufficient_balance(account_id, amount):
        self.send_string(address, "failed@Insufficient account balance for withdrawal")
        return

    starting_balance, ending_balance = self.withdraw_cash(account_id, amount)
    self.send_string(address, f"success@${amount:.2f} withdrawn successfully. Balance: {starting_balance} -> {ending_balance}")
```

- Coverage Criteria: Branch Coverage
- TestFunction:src/test/unitttest_server.py/test_025,test_027,test_028
- Test Case

Test Case	T1.1.9.1	T1.1.9.2	T1.1.9.3
Input	withdraw_cash@2024888888#5200	withdraw_cash@2024888888#50000	withdraw_cash@2024888888#-78.90
Coverage Item	Tcover1.1.9.1	Tcover1.1.9.2	Tcover1.1.9.3
State	-----	-----	-----
Expected Output	"success@$5200.00 withdrawn successfully. Balance: 5200.0 -> 0.0"	"failed@Insufficient account balance for withdrawal"	"failed@Withdrawal amount must be between $0.01 and $50000.00"
Test Result	Passed	Passed	Passed

- Test Coverage: 3/3 = 100%

1.1.10 Process request: Cancel account

```
elif command == "cancel_account":
    account_id = params[0]

    if not self.has_zero_balance(account_id):
        self.send_string(address, "failed@Account balance must be zero to cancel the account")
        return
    self.cancel_account(account_id)
    self.send_string(address, "success@Account canceled successfully")
```

- Coverage Criteria: Branch Coverage
- TestFunction:src/test/unitttest_server.py/test_029,test_030
- Test Case

Test Case	T1.1.10.1	T1.1.10.2
Input	cancel_account@2024888888	cancel_account@2024000000
Coverage Item	Tcover1.1.10.1	Tcover1.1.10.2
State	-----	-----
Expected Output	"success@Account canceled successfully"	" failed@Account balance must be zero to cancel the account"

Test Result	Passed	Passed

- Test Coverage: 2/2 = 100%

1.1.11 Process request: Get balance

```
elif command == "get_balance":
    account_id = params[0]
    balance = self.get_balance(account_id)
    self.send_string(address, f"balance@{balance}")
```

- Coverage Criteria: Statement Coverage
- TestFunction:src/test/unitttest_server.py/test_031
- Test Case

Test Case	T1.1.11.1
Input	“get_balance@2024000000”
Coverage Item	Tcover1.1.11.1
State	-----
Expected Output	“balance@15000.0”
Test Result	Passed

- Test Coverage: 1/1 = 100%

1.1.12 Process request: Query

```
elif command == "query":
    account_id = params[0]
    account_info, transactions = self.query_account(account_id)
    transactions_text = f"Password: {account_info[0]}\nBalance: ${account_info[1]:.2f}\n\nTransactions:\n"
    for transaction in transactions:
        transactions_text += (f"{transaction[2]} - {transaction[0]}: ${transaction[1]:.2f} "
                              f"(Starting Balance: ${transaction[3]:.2f}, Ending Balance: ${transaction[4]:.2f})\n")
    self.send_string(address, f"success@{transactions_text}")
```

- Coverage Criteria: Statement Coverage
- TestFunction:src/test/unitttest_server.py/test_032
- Test Case

Test Case	T1.1.12.1
Input	“query@2024000000”
Coverage Item	Tcover1.1.12.1
State	-----
Expected Output	String
Test Result	Passed

- Test Coverage: 1/1 = 100%

1.1.13 Account exist

```python
def account_exists(self, account_id: str) -> bool:
    conn = sqlite3.connect('bank.db')
    cursor = conn.cursor()
    cursor.execute('SELECT 1 FROM accounts WHERE id = ?', (account_id,))
    exists = cursor.fetchone() is not None
    conn.close()
    # print("whether_Exist==========",exists)
    return exists
```

- Coverage Criteria: Statement Coverage
- TestFunction:src/test/unitttest_server.py/test_033
- Test Case

Test Case	T1.1.13.1
Input	“2024000000”
Coverage Item	Tcover1.1.13.1
State	-----
Expected Output	True
Test Result	Passed

- Test Coverage: 1/1 = 100%

1.1.14 Verify password

```python
def verify_password(self, account_id: str, password: str) -> bool:
    conn = sqlite3.connect('bank.db')
    cursor = conn.cursor()
    cursor.execute('SELECT password FROM accounts WHERE id = ?', (account_id,))
    result = cursor.fetchone()
    conn.close()
    if result is None:
        return False
    return result[0] == password
```

- Coverage Criteria: Statement Coverage
- TestFunction:src/test/unitttest_server.py/test_034
- Test Case

Test Case	T1.1.14.1
Input	"2024000000", "000000"
Coverage Item	Tcover1.1.14.1
State	-----
Expected Output	True
Test Result	Passed

- Test Coverage: 1/1 = 100%

1.1.15 Get balance

```python
def get_balance(self, account_id: str):
    conn = sqlite3.connect('bank.db')
    cursor = conn.cursor()
    cursor.execute('SELECT balance FROM accounts WHERE id = ?', (account_id,))
    result = cursor.fetchone()
    conn.close()
    # print("get success !!!!!!")
    return result[0]
```

- Coverage Criteria: Statement Coverage
- TestFunction:src/test/unitttest_server.py/test_035
- Test Case

Test Case	T1.1.15.1
Input	"2024000000"
Coverage Item	Tcover1.1.15.1
State	-----
Expected Output	15000
Test Result	Passed

- Test Coverage: 1/1 = 100%

1.1.16 Has sufficient balance

```python
def get_balance(self, account_id: str):
    conn = sqlite3.connect('bank.db')
    cursor = conn.cursor()
    cursor.execute('SELECT balance FROM accounts WHERE id = ?', (account_id,))
    result = cursor.fetchone()
    conn.close()
    # print("get success !!!!!!")
    return result[0]
```

- Coverage Criteria: Statement Coverage
- TestFunction:src/test/unitttest_server.py/test_035
- Test Case

Test Case	T1.1.14.1
Input	"2024000000", 10000.0
Coverage Item	Tcover1.1.16.1
State	-----
Expected Output	15000
Test Result	Passed

- Test Coverage: 1/1 = 100%

1.2 Controller (frontend)

1.2.1 Init

- Coverage Criteria: Statement Coverage
- TestFunction: src/test/unitttest_controller.py/test_001_initUI

- Test Case

Test Case	T1.2.1.1
Input	---
Coverage Item	Tcover1.2.1.1
State	---
Expected Output	Main window appears with label 'Banking System Controller'
Test Result	Passed

- Test Coverage: 1/1 = 100%

1.2.2 Open app

- Coverage Criteria: Statement Coverage
- TestFunction: src/test/unitttest_controller.py/ test_002_open_app
- Test Case

Test Case	T1.2.2.1
Input	---
Coverage Item	Tcover1.2.2.1
State	---
Expected Output	num_apps = 1
Test Result	Passed

- Test Coverage: 1/1 = 100%

1.2.3 Close App

```
def close_app(self):
    dialog = QInputDialog(self)
    dialog.setInputMode(QInputDialog.IntInput)
    dialog.setWindowTitle('Close App')
    dialog.setLabelText('Enter app ID:')
    # dialog.setIntRange(1, 99)  # Set range for the input
    dialog.setIntValue(1)  # Set default value
    self.test_dict["d_dialog"]=dialog

    ok = dialog.exec_()
    if ok:
        app_id = str(dialog.intValue())
        if app_id in self.app_instances:
            app_instance = self.app_instances[app_id]
            if app_instance.logged_in == False:
                app_instance.close()
            else:
                QMessageBox.warning(self, "Warning", "Please log out before closing the app.")
        else:
            QMessageBox.warning(self, "Error", "App with specified ID is not open.")
```

- Coverage Criteria: Branch Coverage
- TestFunction: src/test/unitttest_controller.py/ test_003_close_app, test_004_close_app_logged_in, test_005_close_app_notopen
- Test Case

Test Case	T1.2.3.1	T1.2.3.1	T1.2.3.1
Input	---	---	---
Coverage Item	Tcover1.2.3.1	Tcover1.2.3.2	Tcover1.2.3.3

State	---		
Expected Output	---	"Warning", "Please log out before closing the app."	"Error", "App with specified ID is not open."
Test Result	Passed	Passed	Passed

- Test Coverage: 3/3 = 100%

2. Integrated Functional Test

This section provides information on the functional tests we have done for Banking system as well as common use cases. They are integrated into three groups: 1. Functional tests on ATM, including all functionalities concerning ATM with single user and multiple user. 2. Functional tests on APP with single user, with only single user and tests all functionalities except for transferring. 3 Functional tests on both ATM and APP, with multiple users transferring to each other.

2.1 Integrated Functional tests on ATM

- Test Function: src/test/functionalTestATM.py/test_001~031
- Test Case

Test Case	Operation	State	Expected Output	Test Result
T2.1.001	Press create account button; Input 2024123456 as id; Input password 123456	---	MessageBoxText: Account created successfully and enter main page	Passed
T2.1.002	Press deposit button; Input 10 as number of 100 cash and confirm	---	MessageBoxText: $1000.00 deposited successfully	Passed
T2.1.003	Press deposit button; Input 0 and cancel	---	MessageBoxText: Deposit amount must be between $0.01 and $50000.00	Skipped
T2.1.004	Press deposit button; Input -1 and cancel	---	MessageBoxText: Deposit amount must be between $0.01 and $50000.00	Skipped
T2.1.005	Press deposit button; Input 501 and cancel	---	MessageBoxText: Deposit amount must be between $0.01	Skipped

			and $50000.00	
T2.1.006	Press withdraw button; Input 5 as number of 100 cash and confirm	---	MessageBoxText: $500.00 withdrawn successfully	Passed
T2.1.007	Press withdraw button; Input 0 and cancel	---	MessageBoxText: Withdrawal amount must be between $0.01 and $50000.00	Skipped
T2.1.008	Press withdraw button; Input -1 and cancel	---	MessageBoxText: Withdrawal amount must be between $0.01 and $50000.00	Skipped
T2.1.009	Press withdraw button; Input 501 and cancel	---	MessageBoxText: Withdrawal amount must be between $0.01 and $50000.00	Skipped
T2.1.010	Press withdraw button; Input 20 and cancel	---	MessageBoxText: Insufficient account balance for withdrawal	Passed
T2.1.011	Press return card button	---	MessageBoxText: Card returned successfully	Passed
T2.1.012	Press log in button; Input 2024987654 as id; Input password 123456	---	MessageBoxText: Invalid account ID	Passed
T2.1.013	Press log in button; Input 123451 as id; Input password 654321	---	MessageBoxText: Invalid account ID	Passed
T2.1.014	Press log in button; Input 2024123456 as id; Input password 654321	---	MessageBoxText: Invalid password	Passed
T2.1.015	Press log in	---	MessageBoxText:	Passed

	button; Input 2024123456 as id; Input password 123456		Insert card successfully	
T2.1.016	Press change password button; Input j12re7 as password and cancel	---	MessageBoxText: Password must consist of 6 digits	Skipped
T2.1.017	Press change password button; Input 123456 as password and cancel	---	MessageBoxText: New password cannot be the same as the old password	Passed
T2.1.018	Press change password button; Input 654321 as password and confirm	---	MessageBoxText: Password changed successfully; Card returned successfully	Passed
T2.1.019	Press create account button; Input 1234O6789 as id; Input password 654321	---	MessageBoxText: Account ID must consist of 10 digits	Passed
T2.1.020	Press create account button; Input 2024654321 as id; Input password jsk1!-=234	---	MessageBoxText: Password must consist of 6 digits; Back to previous page	Passed
T2.1.021	Press create account button; Input 2024123456 as id; Input password 654321	---	MessageBoxText: Account already exists	Passed

T2.1.022	Press create account button; Input 2024654321 as id; Input password 654321; Press deposit button; Input 10 as number of 100 cash and confirm	---	MessageBoxText: Account created successfully; $1000.00 deposited successfully	Passed
T2.1.023	Press transfer button; Input 123456789 as id and confirm	---	MessageBoxText: Receiver's account ID must consist of 10 digits	Passed
T2.1.024	Press transfer button; Input 2024123456 as id; Input jqk0 as money and confirm	---	MessageBoxText: Transfer amount must be between $0.01 and $50000.00	Passed
T2.1.025	Press transfer button; Input 2024123456 as id; Input 9000 as money and confirm	---	MessageBoxText: Insufficient account balance for transfer	Passed
T2.1.026	Press transfer button; Input 2024654321 as id; Input 10 as money and confirm	---	MessageBoxText:Can't tranfer to your own	Passed
T2.1.027	Press transfer button; Input 2024987654 as id; Input 10 as money and confirm	---	MessageBoxText: Invalid receiver account ID	Passed
T2.1.028	Press transfer button; Input 2024123456 as id; Input 100	---	MessageBoxText: $100.00 transferred successfully	Passed

	as money and confirm			
T2.1.029	Press cancel account button	---	MessageBoxText: Account balance must be zero to cancel the account	Passed
T2.1.030	Press transfer button; Input 2024123456 as id; Input all the balance as money and confirm	---	MessageBoxText: ${balance:.2f} transferred successfully	Passed
T2.1.031	Press cancel account button	---	Account canceled successfully	Passed

2.2 Integrated Functional tests on APP

- Test Function: src/test/functionalTestApp.py/test_001~009
- Test Case

Test Case	Operation	State	Expected Output	Test Result
T2.2.001	Create account with ID 2024123456 and password 123456; Deposit $1000	---	MessageBoxText: Account created successfully; $1000.00 deposited successfully	Passed
T2.2.002	Open the app with ID 1	---	App opened successfully	Passed
T2.2.003	Skipped as appid is automatically assigned	---	---	Skipped
T2.2.004	Open the app with ID 2 and then close it	---	App opened successfully; App closed successfully	Passed
T2.2.005	Log in to app 1 with ID 2024123456 and password 123456	---	MessageBoxText: Log in successfully	Passed
T2.2.006	Open app with ID 3; Log in to	---	MessageBoxText: Log in	Passed

	app 3 with same account ID 2024123456 and password 123456		successfully, the former app is forced to log out	
T2.2.007	Change password on app 3 to 000666	---	MessageBoxText: Password changed successfully; Card returned successfully; Logged out successfully	Passed
T2.2.008	Log in to app 3 using new password 000666	---	MessageBoxText: Log in successfully	Passed
T2.2.009	Query account balance on app 3	---	Account balance displayed successfully	Passed

2.3 Integrated Functional tests on APP and ATM(transfers between multiple users)

- Test Function: src/test/functionalTestApp.py/test_001~009
- Test Case

Test Case	Operation	State	Expected Output	Test Result
T3.2.001	Create accounts with IDs 2024111111; 2024222222; 2024333333 and deposit $1000 each	---	MessageBoxText: Account created successfully; $1000.00 deposited successfully; Card returned successfully	Passed
T3.2.002	Open 3 apps with IDs 1; 2; 3	---	Apps opened and positioned successfully	Passed
T3.2.003	Log in to apps 1; 2; 3 with respective accounts and check balance	---	MessageBoxText: Log in successfully; Balance: $1000.00 for	Passed

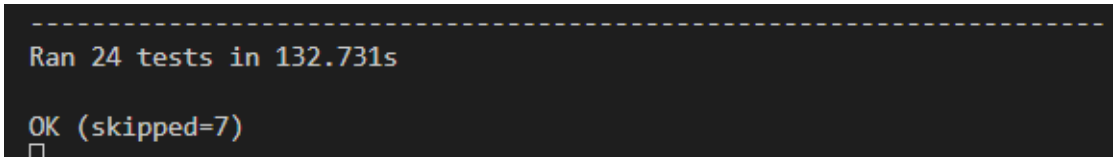
			each account	
T3.2.004	Transfer $100 from account 2024111111 to 2024222222	---	MessageBoxText: $100.00 transferred successfully; Balance for 2024111111: $900.00; Balance for 2024222222: $1100.00	Passed
T3.2.005	Transfer $200 from account 2024222222 to 2024333333	---	MessageBoxText: $200.00 transferred successfully; Balance for 2024222222: $900.00; Balance for 2024333333: $1200.00	Passed
T3.2.006	Transfer $300 from account 2024333333 to 2024111111	---	MessageBoxText: $300.00 transferred successfully; Balance for 2024333333: $900.00; Balance for 2024111111: $1200.00	Passed
T3.2.007	Transfer $100.5 from account 2024111111 to 2024222222	---	MessageBoxText: $100.50 transferred successfully; Balance for 2024111111: $1099.50; Balance for 2024222222: $1000.50	Passed
T3.2.008	Transfer $200.99 from account 2024222222 to 2024333333	---	MessageBoxText: $200.99 transferred successfully; Balance for 2024222222:	Passed

			$799.51; Balance for 2024333333: $1100.99	
T3.2.009	Transfer $1100.99 from account 2024333333 to 2024111111	---	MessageBoxText: $1100.99 transferred successfully; Balance for 2024333333: $0.00; Balance for 2024111111: $2200.49	Passed
T3.2.010	Query on all three apps	---	Transactions and balance are displayed correctly	Passed

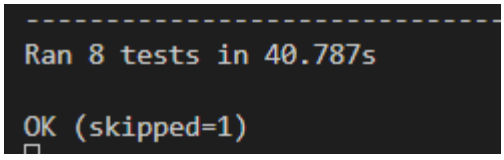
2.3 Test Results

You can run `python -m src.test.run_functionalTest` to test functional tests with UI about buttons' click conditions and corresponding constrains during the process.

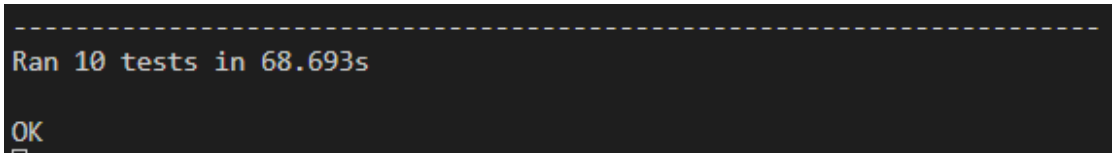
The result of functional tests on ATM.



The result of function tests on App.



The result of function tests on multiple user transfer



3. Model Checking

This section provides an abstract model built in UPPAAL for model checking purposes. You can find the source files in model checking/painkiller.xml and run it locally using an UPPAAL application .

3.1 Introduction

The Painkiller Injection System is divided into three main modules, a User with app and can use ATM as well, an ATM machine, and a Server.

3.2 Assumption

There are totally three user, with each carrying 10 cash initially. There is no balance at their account.

There is only one instance of ATM and one instance of Server.

Fig.1 : UserWithAPP

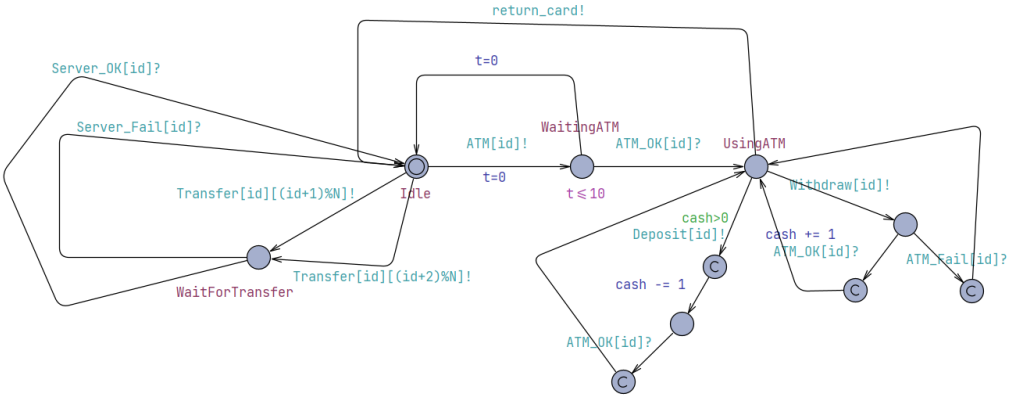


Fig.2: ATM machine

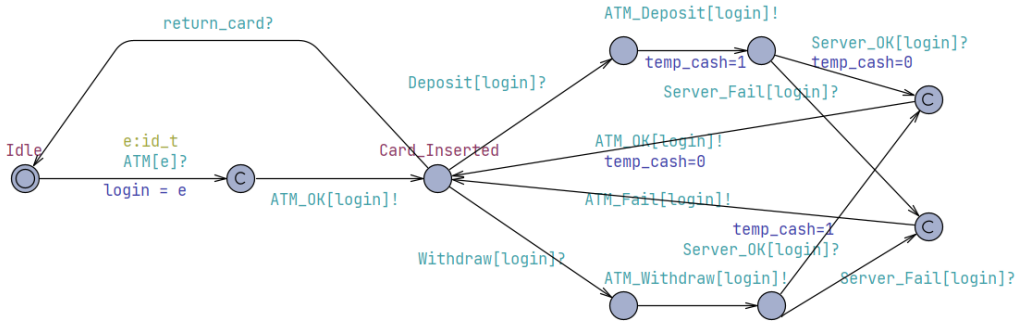
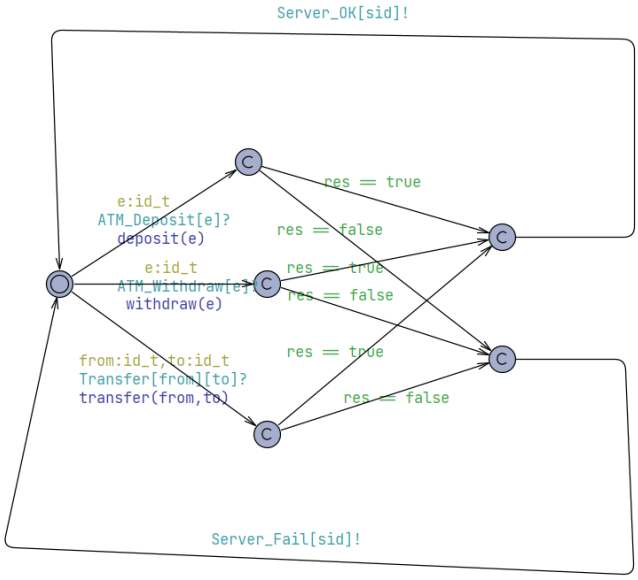


Fig.3: Server



3.3 Properties Validation

3.3.1 No deadlock

Property	A[] not deadlock
Description	The whole system has no deadlock.
Result	Passed

3.3.2 Balance in server will always greater equal 0

Property	A[] Server.balance[0] >= 0 and Server.balance[1] >= 0 and Server.balance[2] >= 0
Description	Balance of every user in the server will always greater equal 0
Result	Passed

3.3.3 User who wants to transfer, always finishes the transfer process.

Property	A<> (UserWithApp(0).WaitForTransfer imply UserWithApp(0).Idle)
Description	User in the process of waiting for transfer always eventually ends up return to idle state
Result	Passed

3.3.4 The cash held by user and their balance at bank server always add up to a constant.

Property	A[] ((UserWithApp(0).Idle and UserWithApp(1).Idle and UserWithApp(2).Idle) imply (Server.balance[0]+Server.balance[1]+Server.balance[2]+UserWithApp(0).cash+UserWithApp(1).cash+UserWithApp(2).cash+ATMachine.temp_cash == 30))
Description	When user are not in intermediate state, the cash they have plus their balance in bank server always add up to the same constant.
Result	Passed

4. Risk Management

4.1.1 Risks related to invalid input: non-digit and invalid length

In terms of the account input and password input, although a input line is used instead of specially made keyboard, but a regular expression of digit is enforced on the edit line, so user can only input digits within range to it.

4.1.2 Risks related to invalid input: pressing the button that is not needed.

There is a button hierarchy so that user will not press the button or input that are not related to the event they are doing. For example, user is not able to click on the cancel account button before they insert card into atm, because cancel account button only appears after log in.

4.1.3 Risks of transaction to be not atomic and problems cause by excessive amount of input at short time.

A sqlite database is used in this project to ensure ACID. Also, there is only one server thread dealing with all the requests from the clients, and there is a message queue to receive input so that no request will be omitted. The client will wait for the return message from server to continue. As a result, if massive input comes at same time, they may wait for a long time. However, in our implementation of frontend(app and atm interface), a pop-out window blocks other input, so exactly one user will be able to input at one time, so there not be such problem exists.