About Me

Omer KORKMAZ

COMPUTER ENGINEER / RESEARCH ASSITANT at SABANCI UNIVERSITY

E-mail: omer.korkmaz.95@windowslive.com

Website: sites.google.com/site/omeerkorkmazz

Linkedin: omerkorkmazz | **Github**: omeerkorkmazz

Stackoverflow: omeerkorkmazz | **Medium**: @omeerkorkmazz

Table of Contents

1.1. Explanation of Decision Table-Based Testing

Decision table-based testing is one of the testing approaches in software world. The aim of this the project is to find inconsistent and redundant pairs of rules from given decision table (data file) and write test suites for the rules if they are satisfiable.

In decision table-based testing, there are three parts that we can explain. First one of these parts is conditions. Conditions are boolean expressions in our program such as (c1: v1&v2), (v3| -v4). In programming, conditions can consist of many parameters such as (v1, v2, v3, v4, …, vn). We need to assume these variables as the parameters of conditions. For example;

c4 = (o1 & o2) | o3

In this expression, it is easily seen that c4 is a condition and o1, o2 and o3 are the parameters (variables) of c4 condition. The combination of parameters is an expression of c4 condition. Second part is regarding rules. Rules are the values of given conditions. Most importantly, the values of rules ought to be known such as ***True*** or ***False***. For instance, if there is a condition with three parameters (x1, x2, x3), a sample rule could be x1 = T, x2 = F, x3 = T, which means rule1 = (TFT). So, we can see many rules like an example, but the total number of rules should be equal to $2^{\#\ of\ conditions}$ Last part is actions. Actions are the results of the rules. In other words, they are behaviors of conditions for each rule. For instance, if c1 = T and c2 = T and c3 = F for rule1, the action is action1. If c1 = F and c2 = F and c3 = T for rule2, the action is action2 and it continues like this process. Now, we might express the terminologies regarding decision table-based testing to be able to understand the working mechanism of this testing style.

Inconsistent Pair: Inconsistent pair, is a pair whose rules are same but whose actions are different. In other meaning, if the values of two rules are equal respectively, their actions must be equal. If not, they are inconsistent pair.

Redundant Pair: Redundant pair, is a pair whose rules are same, also whose actions are same. In other meaning, if the values of two rules are equal respectively, and their actions are same, they are redundant pair.

Satisfiability: The terminology of satisfiability is significant factor for decision table-based testing. Finding inconsistent and redundant pairs is not enough for our aim. To be able to write a test case for a rule as we said before, that rule should be satisfiable. So, a satisfiable rule emphasizes that a rule can be satisfiable if and only if the values of parameters for each condition of a candidate rule makes that rule **True**.

1.2. Information About Used Technologies

In this project, we have provided to use many technologies to be able to solve the given problem properly. Firstly, and most significantly, we need to emphasize that the technologies and development environments are totally depend on the operating system which you use for development progress. For instance, while developing an application, we need a SAT solver which provides to solve boolean expressions and find the satisfiability of given expression. There are many sat solver libraries for many kinds of programming languages. So, you need to install ***Minisat*** so that you can use SAT solver. However, the installation process of Minisat might be changeable depends on the operating system and programming language you use. For this project, we have generally used Windows OS (I have also used MacOS rarely). The technologies, programming languages or platforms can be listed like that;

- Python (version 3.7.1) for both Window OS and macOS
- Pycharm and Spyder (IDE) for both Windows OS and macOS
- Satispy Library (for SAT solver) for both Windows OS and macOS
- Cygwin (to install Minisat) for only Windows OS
- Homebrew or Brew (to install Minisat) for only macOS
- Pyinstaller Library (for executable file) for only Windows OS

We have chosen Python programming language because of two reasons. First one is about SAT solver. According to our research and given information, there is a nice-working sat solver library for python called ***Satispy***. I have tried this library and it is easy to use CNF and CNF from string properly. Another reason is totally regarding python. It is a powerful and beneficial programming language for developers. It is compatible for operating systems such as Windows OS, macOS, Linux, etc. Also, it has huge and strong libraries for solving mathematical problems, machine learning, data mining problems, etc.

2. How Does It Work?

In this section, we will explain the working process (build, execute, compile) of the application we have developed. Firstly, we want to clarify the process for Windows Operating System. Note that, using Windows OS is highly recommended if you would like to use executable file in your system. Morever, keep in mind that python and Minisat should be installed whatever you are using as an operating system.

There are two ways to run the application (program) for Windows Environment. These ways will be explained below:

- **Run executable file:** There is a file (.exe) which you can use easily in your Windows System. You just need to open command prompt (cmd) and move your location into the location of executable file. (use cd command). Then, you can run the program by writing this command:

```
C:\Users\Omer\Desktop>dectblproc C:\Users\Omer\Desktop\cs539\dt5
```

C:\Users\Omer\Desktop shows the location of my executable file called ***dectblproc.*** "dectblproc" is the name of executable file. Second part of the command is the value of argument. In this project, the data file will be given by the user. So, you need to give the program the path of the data file which you want to use. (If you do not give the path, it will not work!

After this command, you can see the output of the application. It will be similar with given example like that:

```
C:\Users\Omer\Desktop>dectblproc C:\Users\Omer\Desktop\cs539\dt5
Processing File: dt5
Is table complete? 87% complete
Is table redundant? Yes
    Redundant pairs of rules:  (r7, r8)
Is table inconsistent? Yes
    Inconsistent pairs of rules:  (r5, r7)
Testsuite
=========
+----+-------+-------+-------+-------+-------+
|    | o1    | o2    | o3    | o4    | o5    |
+====+=======+=======+=======+=======+=======+
| r1 | False | False | True  | True  | False |
+----+-------+-------+-------+-------+-------+
| r2 | True  | False | True  | True  | False |
+----+-------+-------+-------+-------+-------+
| r5 | False | False | False | True  | False |
+----+-------+-------+-------+-------+-------+
| r6 | True  | False | False | True  | False |
+----+-------+-------+-------+-------+-------+
| r7 | False | False | False | True  | False |
+----+-------+-------+-------+-------+-------+
| r8 | False | False | False | False | False |
+----+-------+-------+-------+-------+-------+
```

- **<u>Run Python File:</u>** If the executable file does not work because of any reason, you can still run the application easily by using python file of program. We assume that you have installed python and minisat. You need to use the command which will be shown below:

```
C:\Users\Omer\Desktop>python 26070-OmerKORKMAZ.py C:\Users\Omer\Desktop\cs539\dt1
```

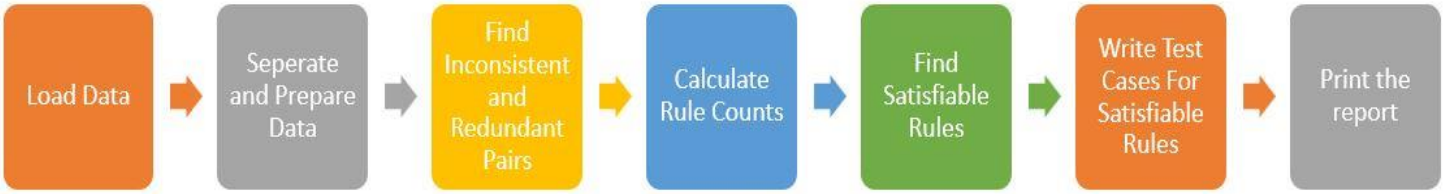
The running process of python file is similar with executable file. You again need to move your location into the location of python file(.py). However, you should use ***python*** command to be able to run the python file. Second part is again the value of argument (parameter) which shows the path of data file which you want to test.

If you want to run the program in macOS, Linux (except Windows), you can not use executable file. The reason of this problem is about pyinstaller library. This library provides to create executable file for your python scripts, but it totally creates this file as a compatible file for your operating system which you use "pyinstaller." For instance, as we said before, we have developed this application in Windows OS. While creating executable file, we have used again Windows. So, the executable file of project is only compatible and runnable for Windows. The only way you might use to run the program is ***"running python file"*** like we have explained above. The running process is similar with running python file on Windows. The only difference could be regarding command prompt of operating system you want to use.

3. Information About Codes

In this section, we will give detail information about the general codes, methods, algorithms which have been used and the working mechanism of program.

There are many methods we have used in the code and all of them is in a one file. In the beginning, we would like to show the working process of our application.



1. Get the path of data file from user and load the data into the program. Then, read each line of data respectively.
2. Separate the prepared data according to the given information such as conditions, condition values, condition expressions, rules, actions and action values.
3. Find inconsistent and redundant pairs from separated data.
4. Calculate the count of each rule by checking uniqueness of pairs.

5. Find the satisfiable rules by checking the satisfiability process.
6. Write suitable test cases for satisfiable rules.
7. Print the report of intended information to the screen.

PreviewTable Method

```
# it takes all seperated data and just previews how they are seen.
def PreviewTable(conditions, conditionExpressions, conditionValues, outputs, outputValues):
    for i in range(0, len(conditions)):
        print(conditions[i], " ", conditionExpressions[i])

    for i in range(0, len(conditions)):
        print(conditions[i], " ", conditionValues[i])

    for i in range(0, len(outputs)):
        print(outputs[i], " ", outputValues[i])
```

This method was developed at the request of the developer. It previews the given data on the screen. In other words, we can check whether if the data file is loaded correctly or not.

PrepareToSeperate Method

```
# Prepares the our data for seperation. fileName is dt0 as a default.
# It takes the document name and opens document and starts to read lines respectively.
# Then, creates a list of lines and returns the list
def PrepareToSeperateData(fileName):
    isExcept = False
    try:
        pathInput = fileName.split("\\")
    except:
        isExcept = True

    with open(r"" + fileName, 'r') as f:
        content = f.readlines()
    content = [x.strip() for x in content]

    if(isExcept):
        return content, fileName
    else:
        return content, pathInput[len(pathInput) - 1]
```

This method loads the given data into the program. Then, it reads each line of the data. At the end, it returns the data as a list and the name of file. If the file name is not proceeded, it will return the path of data as a file name.

SeperateData Method

```python
#It takes content which is returned by PrepareToSeperateData()
#It seperates document as Conditions, ConditionExpressions, ConditionVaues,Outputs and OutputValues
def SeperateData(content):
    conditions = []
    conditionExpressions = []
    conditionValues = []
    outputs = []
    outputValues = []

    seperateIndex = content.index("##")

    for i in range(0, seperateIndex):
        text = content[i]
        if (text.startswith("c")):
            condition, expression = text.split(":")
            conditions.append(condition)
            conditionExpressions.append(expression)

    for i in range(seperateIndex, len(content)):
        text = content[i]
        if (text.startswith("c")):
            condition, conditionValue = text.split(" ")
            conditionValues.append(list(conditionValue))
        elif (text.startswith("a")):
            output, outputValue = text.split(" ")
            outputs.append(output)
            outputValues.append(list(outputValue))

    return conditions, conditionExpressions, conditionValues, outputs, outputValues
```

This method seperates the loaded data and returns necessary parts like conditions, condition values, condition expressions, outputs, output values. (Output means action and condition values means the values of rules)

FindPairs Method

```python
def FindPairs(conditions, conditionExpressions, conditionValues, outputs, outputValues):
    length = len(conditionValues[0])
    InconsistentPairs = []
    RedundantPairs = []

    for i in range(0, length - 1):
        state = True

        # check condiitons whether if they are equal or not.
        # if state is true, it will check actions for redundant and inconsistent situations.
        # if state is false, it means there is no redundancy or inconsistency.
        for k in range(i + 1, length):
            for j in range(0, len(conditions)):
                if (conditionValues[j][i] == "-" or conditionValues[j][k] == "-"):
                    state = True
                elif (conditionValues[j][i] == conditionValues[j][k]):
                    state = True
                elif (conditionValues[j][i] != conditionValues[j][k]):
                    state = False
                    break

            # check actions whether if they are same or not. It clarifies inconsistent and redundant rules.
            if (state == True):
                outState = True
                for m in range(0, len(outputs)):
                    if (outputValues[m][i] != outputValues[m][k]):
                        outState = False
                        break
                    else:
                        outState = True

                if (outState == False):
                    # print("Inconsistent pair rule ",i+1, "and rule ",k+1)
                    InconsistentPairs.append((i, k, "r" + str(i + 1), "r" + str(k + 1)))
                else:
                    # print("Redundant pair rule ",i+1, "and rule ",k+1)
                    RedundantPairs.append((i, k, "r" + str(i + 1), "r" + str(k + 1)))

    return InconsistentPairs, RedundantPairs
```

This method finds inconsistent and redundant pair of rules from given data. Then, it returns the inconsistent and redundant pairs.

CalculateRuleCounts Method

```python
# This method calculates the total number of unique rules
def CalculateRuleCounts(InConsistentPairs, RedundantPairs):
    combineList = InconsistentPairs + RedundantPairs
    deletedList = []
    uniqueList = []

    for i in range(0, len(combineList)):

        firstIndex = combineList[i][0]
        secondIndex = combineList[i][1]
        firstRuleCount = 0
        secondruleCount = 0

        # find the total rule numbers of each pair
        for c in range(0, len(conditions)):
            if (conditionValues[c][firstIndex] == "-"):
                firstRuleCount += 1
            if (conditionValues[c][secondIndex] == "-"):
                secondruleCount += 1

        # if # of rules of first pair is greater than second pair or they are equal
        if (firstRuleCount > secondruleCount or firstRuleCount == secondruleCount):
            isSeconPairUnique = secondIndex in uniqueList
            isSecondPairDeleted = secondIndex in deletedList
            # if second pair is presence in uniqueList before, delete it from uniqueList and add it to deletedList.
            if (isSeconPairUnique == True):
                uniqueList.remove(secondIndex)
            elif (isSecondPairDeleted == False):
                deletedList.append(secondIndex)

                # first pair is added into uniqueList, if it is not presence in the list before and it is not in deletedList.
            isFirstPairDeleted = firstIndex in deletedList
            isFirstPairUnique = firstIndex in uniqueList

            if (isFirstPairDeleted == False):
                if (isFirstPairUnique == False):
                    uniqueList.append(firstIndex)

        # if # of rules of second pair is greater than first pair
        if (secondruleCount > firstRuleCount):
            isFirstPairUnique = firstIndex in uniqueList
            isFirstPairDeleted = firstIndex in deletedList
            # if first pair is presence in uniqueList before, delete it from uniqueList and add it to deletedList
            if (isFirstPairUnique == True):
                uniqueList.remove(firstIndex)
                if (isFirstPairDeleted == False):
                    deletedList.append(firstIndex)
            elif (isFirstPairDeleted == False):
                deletedList.append(firstIndex)

                # second pair is added into uniqueList, if it is not presence in the list before and it is not in deletedList.
            IsSecondPairUnique = secondIndex in uniqueList
            IsSecondPairDeleted = secondIndex in deletedList
            if (IsSecondPairDeleted == False):
                if (IsSecondPairUnique == False):
                    uniqueList.append(secondIndex)

            elif (isFirstPairDeleted == False):
                deletedList.append(firstIndex)

    # finds the other unique rules except inconsistent and redundant pairs
    for r in range(0, len(conditionValues[0])):
        isExistInUnique = r in uniqueList
        isExistInDeleted = r in deletedList
        if (isExistInDeleted == False and isExistInUnique == False):
            uniqueList.append(r)

    # finds the total rule count by checking unique rules
    rCounts = []
    for m in range(0, len(uniqueList)):
        ruleCount = 0
        for c in range(0, len(conditions)):
            if (conditionValues[c][uniqueList[m]] == "-"):
                ruleCount += 1
        rCounts.append(math.pow(2, ruleCount))

    return sum(rCounts)
```

This method is one of the longest methods in the codes because it needs to check many situations to be able to find the rule counts. It takes inconsistent and redundant pairs as parameters and returns the total rule count.

WriteTestCases Method

```python
def WriteTestCases(conditionValues, conditions):
    generalOutput = ""
    dontCareIndexes = []
    testCaseExpressions = []
    dontCareConditionsIndexes = []
    for t in range(0, len(conditionValues[0])):
        isDontCare = False
        dontCaresCount = 0
        for i in range(0, len(conditions)):
            conditionText = ""
            # we need to check each condition as true
            # if condition value is false, we can use negation(-) for that rule to be able to get true result from sat solver
            if (conditionValues[i][t] == "T"):
                conditionText = "(" + conditionExpressions[i] + ")"

            elif (conditionValues[i][t] == "F"):
                conditionText = "-(" + conditionExpressions[i] + ")"

            # if the value of condition of tth rule is dont care, check for both false and true.
            # if one of them says not satisfiable, we should approve that one value.
            elif (conditionValues[i][t] == "-"):
                dontCaresCount+=1
                isDontCare = True
                dontCareConditionsIndexes.append((i, t))
            # combines the conditions
            if (i == 0):
                generalOutput = conditionText
            else:
                generalOutput = generalOutput + " & " + conditionText

        if (isDontCare != True):
            testCaseExpressions.append((str(t), generalOutput))
        else:
            dontCareIndexes.append((t, dontCaresCount))

    return testCaseExpressions, dontCareIndexes, dontCareConditionsIndexes
```

This method takes condition values and conditions as parameters and writes test cases by combining the expressions according to the conditions. Then, it returns test cases, dontcare indexes and dontcareConditionIndexes because we need to use another method for dontcare situations.

WriteTestCasesForDontCares Method

```
def WriteTestCasesForDontCares(conditionValues, dontCareIndexes, dontCareConditionIndexes, conditionExpressions):
    dontCareSuites=[]
    for r in range(0, len(dontCareIndexes)):
        dCount = dontCareIndexes[r][1]
        dIndex = dontCareIndexes[r][0]
        possibleSamples = list(itertools.product([True, False], repeat=dCount))
        conditionText = ""
        globalText = ""
        for p in range(0, len(possibleSamples)):
            dCare = 0
            IsDontCare = False
            for c in range(0, len(conditions)):
                if (conditionValues[c][dIndex] == "T"):
                    conditionText = "(" + conditionExpressions[c] + ")"

                elif (conditionValues[c][dIndex] == "F"):
                    conditionText = "-(" + conditionExpressions[c] + ")"

                elif (conditionValues[c][dIndex] == "-"):
                    IsDontCare = True
                    if (possibleSamples[p][dCare] == True):
                        conditionText = "(" + conditionExpressions[c] + ")"
                    elif (possibleSamples[p][dCare] == False):
                        conditionText = "-(" + conditionExpressions[c] + ")"

                if (c == 0):
                    globalText = conditionText
                else:
                    globalText = globalText + " & " + conditionText

                if(IsDontCare):
                    dCare+=1
                    IsDontCare = False

            dontCareSuites.append((dIndex, globalText))

    return dontCareSuites
```

This method writes test cases for dontcare rules. For dontcare situation, we need to think two cases for the value of True and the value of False. The condition which has dontcare should take both True and False. Then, we need to check whether if that rule is satisfiable or not. At the end, it returns dontcare test cases.

PrintTestCaseTable Method

```
325 def PrintTestCaseTable(parametersValues, conditionValues):
326     headers = []
327     rules = []
328     testCases = []
329     table = []
330 
331     for t in range(0, len(parametersValues)):
332         parametersValues[t] = sorted(parametersValues[t])
333 
334     if (len(parametersValues) != 0):
335         for c in range(0, len(parametersValues[0])):
336             headers.append(parametersValues[0][c][0])
337 
338     for b in range(0, len(parametersValues)):
339         if (len(parametersValues[b]) != 0):
340             for s in range(0, len(parametersValues[0])):
341                 rules.append(parametersValues[b][s][1])
342             testCases.append(("r" + str(parametersValues[b][0][2]), rules))
343             rules = []
344 
345     for z in range(0, len(conditionValues[0])):
346         ruleInput = "r" + str(z + 1)
347         for k in range(0, len(testCases)):
348             if (ruleInput == testCases[k][0]):
349                 table.append(testCases[k])
350 
351     tempTable = []
352     testSuitesTable = []
353     for u in range(0, len(table)):
354         tempTable.append(table[u][0])
355         for n in range(0, len(table[u][1])):
356             tempTable.append(table[u][1][n])
357 
358         testSuitesTable.append(tempTable)
359         tempTable = []
360 
361     print("Testsuite")
362     print("=========")
363     print(tabulate(testSuitesTable, headers, tablefmt="grid"))
```

This method prints the test cases as a table. To be able to print them as a table, we have used python library called “tabulate”. It does not return anything. It just prints the results.

SATSolver Method

```
366 def SATSolver(dontCareSuites, testCaseExpressions):
367     FoundedIndex = -1
368     parameters = []
369     parametersValues = []
370 
371     for i in range(0, len(dontCareSuites)):
372         tempValues = []
373         if (FoundedIndex != dontCareSuites[i][0]):
374             # print(int(dontCareSuites[i][0]) + 1, " rule expression ----->", dontCareSuites[i][1])
375             # finds the satisfiable rules and writes test suite for them by using CnfFromString method
376             exp, symbols = CnfFromString.create(dontCareSuites[i][1])
377             solver = Minisat()
378             solution = solver.solve(exp)
379             if solution.success:
380                 FoundedIndex = dontCareSuites[i][0]
381                 # print("Rule :", int(dontCareSuites[i][0]) + 1)
382 
383                 for symbol_name in symbols.keys():
384                     # print("%s is %s" % (symbol_name, solution[symbols[symbol_name]]))
385                     parameters.append((symbol_name, solution[symbols[symbol_name]], (int(dontCareSuites[i][0]) + 1)))
386             else:
387                 # print("The expression cannot be satisfied")
388                 FoundedIndex = dontCareSuites[i][0]
389 
390             parametersValues.append(parameters)
391             parameters = []
392 
393     for i in range(0, len(testCaseExpressions)):
394         tempValues = []
395         # print("Rule ", (int(testCaseExpressions[i][0])+1))
396         # finds the satisfiable rules and writes test suite for them by using CnfFromString method
397         exp, symbols = CnfFromString.create(testCaseExpressions[i][1])
398         solver = Minisat()
399         solution = solver.solve(exp)
400         if solution.success:
401 
402             for symbol_name in symbols.keys():
403                 # print("%s is %s" % (symbol_name, solution[symbols[symbol_name]]))
404                 parameters.append((symbol_name, solution[symbols[symbol_name]], (int(testCaseExpressions[i][0]) + 1)))
405 
406         parametersValues.append(parameters)
407         parameters = []
408 
409     ##----------suitable test order operation ---------##
410     for m in range(0, len(parametersValues)):
411         parametersValues[m] = sorted(parametersValues[m])
412 
413     return parametersValues
```

This method provides to solve boolean expressions and decides whether if solved rule is satisfiable or not. If it is satisfiable, it appends the satisfiable values of conditions (rule) into a list. Then, that list is returned by SATSolver method.

PrintResults Method

```python
# This method prints the expected outputs to the user
def PrintResults(fileName, InconsistentPairs, RedundantPairs, conditionValues):
    isRedundant = False
    isInConsistent = False
    totalRuleNumbers = math.pow(2, len(conditionValues))
    completeTablePercent = 100

    if (len(InconsistentPairs) > 0):
        isInConsistent = True

    if (len(RedundantPairs) > 0):
        isRedundant = True

    totalRuleCount = CalculateRuleCounts(InconsistentPairs, RedundantPairs)

    # if there is any redundant or inconsistent pairs, calculates the percentage of table complete
    # if not, it means the table is %100 complete, as I defined it initally as 100
    if (isRedundant or isInConsistent):
        completeTablePercent = (totalRuleCount / totalRuleNumbers) * 100

    redundantAnswer = "No"
    inconsistentAnswer = "No"
    if (isRedundant):
        redundantAnswer = "Yes"
    if (isInConsistent):
        inconsistentAnswer = "Yes"

    rPairs = []
    rtext = ""
    for r in range(0, len(RedundantPairs)):
        rPairs.append((RedundantPairs[r][2], RedundantPairs[r][3]))
        if (r == len(RedundantPairs) - 1):
            rtext = rtext + "(r" + str(RedundantPairs[r][0] + 1) + ", r" + str(RedundantPairs[r][1] + 1) + ")"
        else:
            rtext = rtext + "(r" + str(RedundantPairs[r][0] + 1) + ", r" + str(RedundantPairs[r][1] + 1) + "), "

    itext = ""
    for r in range(0, len(InconsistentPairs)):
        rPairs.append((InconsistentPairs[r][2], InconsistentPairs[r][3]))
        if (r == (len(InconsistentPairs) - 1)):
            itext = itext + "(r" + str(InconsistentPairs[r][0] + 1) + ", r" + str(InconsistentPairs[r][1] + 1) + ")"
        else:
            itext = itext + "(r" + str(InconsistentPairs[r][0] + 1) + ", r" + str(InconsistentPairs[r][1] + 1) + "), "

    # print output process
    print("Processing File: %s" % fileName)
    print("Is table complete? " + str(int(completeTablePercent)) + "% complete")
    print("Is table redundant? " + redundantAnswer)
    if (isRedundant):
        print("    Redundant pairs of rules: ", rtext)
    print("Is table inconsistent? " + inconsistentAnswer)
    if (isInConsistent):
        print("    Inconsistent pairs of rules: ", itext)
```

This method takes necessary parameters and prints the report of given data file. It prints the file name, inconsistent and redundant pairs, percentage of completeness.

Main Part

```
# ------------------ MAIN PART --------------------------


#get the path of file from user (arg)
pathInput = str(sys.argv[1])

#prepares file for seperating data (read lines respectively)
content, fileName = PrepareToSeperateData(pathInput)

conditions, conditionExpressions, conditionValues, outputs, outputValues = SeperateData(content)
InconsistentPairs, RedundantPairs = FindPairs(conditions, conditionExpressions, conditionValues, outputs, outputValues)
testCaseExpressions, dontCareIndexes, dontCareConditionsIndexes = WriteTestCases(conditionValues, conditions)
dontCareSuites = WriteTestCasesForDontCares(conditionValues, dontCareIndexes, dontCareConditionsIndexes, conditionExpressions)
parametersValues = SATSolver(dontCareSuites, testCaseExpressions)


PrintResults(fileName, InconsistentPairs, RedundantPairs, conditionValues)
PrintTestCaseTable(parametersValues, conditionValues)
```

This is just information from main section of codes. “sys.argv” provides to get value from the user as a parameter. We have used an argument for the path of given file.