 # File kcv9b.py

```
# Author: G.Doeben-Henisch
# First date: September 6, 2020
# Last date: October 22, 2020, 12:55 am

################################


# CLASS DEFINITIONS

class Start:
        def __init__(self):
                self.menulist = ['START','EDIT P and V','EDIT S', 'EDIT
X','SIMULATION','EVALUATION','STOP']

        def menushow(self):
                i=0  # Counter for menu-loop
                for state in self.menulist:
                        i=i+1
                        message="str(i)+' is '+state"
                        pub.useroutput(eval(message))

        def badoption(self,opt):
                if int(opt)<1 or int(opt)>7:
                        message='!!You have selected a bad option'
                        pub.useroutput(message)

                if int(opt)>0 and int(opt)<8:
                        message='!!You have selected the state :\n'+self.menulist[int(opt)-1]
                        pub.useroutput(message)

#################################################################

class Actor:
        def __init__(self,inp):
                self.message=inp

#################################################################

class Publish():

        def show(self,other):
                print(other.message)

        def useroutput(self,message):
                print(message)

        def userinput(self,message):
                self.opt=input(message)
```

```
#####################################################################
# CLASS PROBLEM and VISION
'''
MAIN IDEA

A main window W1 with a menu showing all possible questions for the problem and the vision  to be nswered.

(a) See a list of all topics to be covered
(b) See a list of all problem-vision documents so far
(c) Enter a name for a pv-document
(d) Describe the problem P: What is given and what is the intended future state?
(e) Describe a vision
(f) Describe the intended real part of the world (space).
(g) Describe the time model T : which time period, which cycles.
(h) Which kinds of actors are seen as being important for the problem and its  future?
'''


class Problem(Actor):

        def menushow(self):
                self.menulist = ['PROBLEM','VISION','REGION', 'TIME','PERSONS (Individuals or Roles)']
                i=0  # Counter for menu-loop
                for state in self.menulist:
                        i=i+1
                        message="str(i)+' is '+state"
                        pub.useroutput(eval(message))

        def getpvlist(self):
                self.problemList = st.openstoragepv()
                self.kl=list(st.dpv.keys())
                self.message='\n Here is a list of all Problem-Vision documents so far:\n'+str(self.kl)
                pub.useroutput(self.message)


        def getpname(self,inp):
                self.problemName = inp
                message='Feedback Problem Name :\n'+self.problemName
                pub.useroutput(message)

        def getproblem(self,inp):
                self.problemNow = inp
                message='Feedback Problem Now :\n'+self.problemNow
                pub.useroutput(message)

        def getvision(self,inp):
                self.problemFuture = inp
                message='Feedback Problem Future :\n'+self.problemFuture
```

```
                pub.useroutput(message)

        def getregion(self,inp):
                self.problemRegion = inp
                message='Feedback Problem Region :\n'+self.problemRegion
                pub.useroutput(message)

        def gettime(self,inp):
                self.problemTime = inp
                self.problemTM = self.problemTime.split(',')
                message='Feedback Problem TimeModel :\n'+str(self.problemTM)
                pub.useroutput(message)

        def getperson(self,inp):
                self.problemPerson = inp
                self.problemPRS = self.problemPerson.split(',')
                message='Feedback Problem Persons :\n'+str(self.problemPRS)
                pub.useroutput(message)

        def problemTotal(self):
                self.problemAll=[]
                self.problemAll.append(self.problemNow)
                self.problemAll.append(self.problemFuture)
                self.problemAll.append(self.problemRegion)
                self.problemAll.append(self.problemTime)
                self.problemAll.append(self.problemPerson)
                self.problemAll
                ########################################
                # Now mixing the inputs with categories
                self.pvcats=['problem','vision','region','time','persons']
                self.pv=dict(zip(self.pvcats,self.problemAll))
                self.message='\n The final  problem document with the name\
n'+str(self.problemName)+'\n is the following one:\n'+str(self.pv)
                pub.useroutput(self.message)
                ################################################
                # Store document permanently
                st.dpv[self.problemName]=self.pv
                st.closestoragepv()

######################################################################
# CLASS S(tate Description)
'''
IDEA:
This state should allow in the final version the editing of the texts S and X in parallel. Additionally
one should be able to call from within this state(s) the simulation mode to test whether the actual
texts are working.

FOR NOW:
In this first experimental version one has to work either with the stae S or with the state X
separatedly. Simulation would be a follow up state.

TASK:
```

Input all data which are necessary for the S-state (including sectioning
and extended texts with details)

ACTORS:
Human experts.

SYSTEM INTERFACE:
A main window W1 offering the editing of a text consisting of individual
statements. Every statement can be edited separately and repeatedly.

ACTIONS:
Select either a given statement for editing or edit a new statement or
stop.

IMPLEMENTATION:
Using the set class of python to collect expressions. Set operations are very convenient e.g. to unify different sets to one set, and more.

'''
```
class AState(Actor):

	def __init__(self):
		self.stateAll = set()

	def getStateAll(self,inp):
		self.stateAll=inp

	def emptydocs(self):
		self.stateAll = set()

	# Get list of all state descriptions so far
	def getslist(self):
		st.openstorage()
		self.kl=list(st.d.keys())
		self.message='\n Here is a list of all state descriptions so far:\n'+str(self.kl)
		pub.useroutput(self.message)

	def getsname(self,inp):
		self.stateName = inp
		message='Feedback STATE Name :\n'+self.stateName
		pub.useroutput(message)

	# Load a document S from file
	def getdocs(self,filen):
		self.stateAll = set()
		self.stateAll=st.d[filen]

	def getexpression(self,inp):
		self.expression = inp
		self.stateAll.add(inp)
		message='Feedback Your last expression :\n'+str(self.expression)
		pub.useroutput(message)
```

```
            message='Feedback Your document S so far :\n'+str(self.stateAll)
            pub.useroutput(message)

            ################################################
            # Store document permanently
    def storeSdocument(self):
            st.d[self.stateName]=self.stateAll
            st.closestorage()



##########################################################################
# CLASS X (Change Rules)
'''
IDEA:
The change rules X are described in the requirements paper cited in the beginning of the main
program text. The principal idea of the change rules X is to allow changes to an actual state S if
certain conditions are fulfilled (satisfied). These changes will be executed during the state called
simulation.

FOR NOW:
Because a complete implementation of the theoretically possible change rules is nearly an infinite
task this version of the change rules X called X01 is limited to the simplest possible case. This
contains the following simple structure:

IF Condition C THEN with Probability Pr realize the Effect E- and E+.

While this is already the case without any actor all the parts (C,E+,E+) are additionally limited to
one expression each. Thus we start with the format:

IF Condition C(1) THEN with Probability Pr [0,1] realize the Effect E-(1) and E+(1).

The strategy is to extend all these limits stepwise in the next versions.

TASK:
Input all rules   for the X-state

ACTORS:
Human experts.

SYSTEM INTERFACE:
A main window W1 offering the editing of a text consisting of individual
rules. Every statement can be edited separately and repeatedly.

ACTIONS:
Select either a given statement for editing or edit a new statement or
stop.

IMPLEMENTATION:
Using the list-construct of python to collect expressions, because lists are ordered and mutable and
allow many intersting operations.
rule pure =[ set Cond, number Pr, set E-, set E+]
```

```
Make a dict with CATs x rule pure
All rules = [D1, D2, ..., Dn]
Name for 'All rules'
'''

class Xrules(Actor):

	def __init__(self):
		self.rulesAll = []
		self.rule=[]
		self.rcat=['CONDITION', 'PROBABILITY', 'EFFECT+','EFFECT-']
		self.cond=set()		#Needs a set
		self.probab=0.0
		self.eminus=set()	#Needs a set
		self.eplus=set()	#Needs a set
		self.ruleDocName=''

	def AllToZero(self):
		self.rulesAll = []

	def RuleToZero(self):
		self.rule = []

	def CondToZero(self):
		self.cond=set()

	def ProbToZero(self):
		self.probab=0.0

	def EminusToZero(self):
		self.eminus=set()

	def EplusToZero(self):
		self.eplus=set()

	def greeting(self):
	   message="Your rule set at start :"+str(self.rulesAll)
	   pub.useroutput(message)
	   message="Your single rule buffer at start :"+str(self.rule)
	   pub.useroutput(message)

	# Get list of all rule documents so far
	def getxlist(self):
		st.openstorager()
		self.kl=list(st.dr.keys())
		self.message='\n Here is a list of all rule documents so far:\n'+str(self.kl)
		pub.useroutput(self.message)
		st.closestorager()

	# Get a name to select a document or make some new one
	def getxname(self,inp):
		self.ruleDocName = inp
```

```
        message='Feedback RULE document Name :\n'+self.ruleDocName
        pub.useroutput(message)

# Load a document X from DB
def getdocx(self,filen):
        st.openstorager()
        self.rulesAll = []
        self.rulesAll=st.dr[filen]
        st.closestorager()

# Get a condition
def getcond(self,inp):
        self.inp="
        self.inp = inp
        self.cond.add(self.inp)
        message="Your condition set so far : "+str(self.cond)
        pub.useroutput(message)

# Get a probability
def getprob(self,inp):
        self.probab=inp
        self.rule.append(self.probab) #Appends a probability
        message="Your single rule buffer : "+str(self.rule)
        pub.useroutput(message)



# Get the set EPlus
def geteplus(self,inp):
        self.inp="
        self.inp = inp
        self.eplus.add(self.inp)
        message="Your  set eplus so far : "+str(self.eplus)
        pub.useroutput(message)

# Get the set EMinus
def geteminus(self,inp):
        self.inp="
        self.inp = inp
        self.eminus.add(self.inp)
        message="Your  set eminus so far : "+str(self.eminus)
        pub.useroutput(message)

        ################################################
        # Store document permanently

def storeFinal(self,data):
        st.openstorager()
        st.dr[self.ruleDocName]=data
        st.closestorager()
```

```
##########################################################################
# CLASS SIMULATOR
############################
# FIRST SIMPLE SIMULATOR
'''
As a first simple simulator cycle the following schema has been proposed:
1. If there already exists a state file S and a rule file X load such files, otherwise edit two new files.
2. The simulator works in cycles.
3. Every cycle CYC the actual version of a state description S as well a rule set X will be loaded into the simulator.
4. If the set of applicable rules is not empty ¬∅ then applying satisfying change rules X∗ to the actual state Snow will change the state according to the schema: Si+1=Si− E− ∪E+.
5. After the simulation has finished the set S can have been changed. S can be stored permanently.
'''
'''
Using set operations:
https://docs.python.org/3.1/library/stdtypes.html#set
'''
################################################
# FORMAT OF A RULE WITHOUT ACTOR
#
# IF: CONDITION   THEN: PROBABILITY   - E-MINUS -     E-PLUS
#################################################################
#V0:. Expressions.....[0,1].....Expressions...Expressions
#

class Simulation():
        def __init__(self):
                self.s={}

        def xapply(self,s,x):
                for i in range(len(x)):
                        message='Set S given : \n'+str(s)
                        pub.useroutput(message)
                        r=x[i]
                        message="Actual rule :\n"+str(r)
                        pub.useroutput(message)
                        if r['CONDITION'].issubset(s):
                                delset=r['EFFECT-']
                                d=delset.pop()
                                if d in s:
                                        s.remove(d)
                                else:
                                        pass
                        message='Set S after Remove : \n'+str(s)
                        pub.useroutput(message)
                        s=s.union(r['EFFECT+'])
                        message='Set S after Union : \n'+str(s)
                        pub.useroutput(message)
                        aas.getStateAll(s)
```

```
#########################################################################
# CLASS DATA STORAGE
#
'''
All the files which will be created during a session can at the end of the session be stored at will in a
shelve. These can also be loaded in the beginning to continue editing. Every data set is marked by
an individual free index.
'''
# IMPORTS

import shelve  #To store and reload data
        #See: https://docs.python.org/3.1/library/shelve.html

#############################################################

class Storage():
        def __init__(self,storageID,rlID,pvID):
                self.storename=storageID # database for states
                self.rulename=rlID     # databse for rules
                self.pvname=pvID      # database for problem-vision documents

        def openstorage(self):
                self.d=shelve.open(self.storename,writeback=True)

        def openstorager(self):
                self.dr=shelve.open(self.rulename,writeback=True)

        def openstoragepv(self):
                self.dpv=shelve.open(self.pvname,writeback=True)

        def closestorage(self):
                self.d.close()

        def closestorager(self):
                self.dr.close()

        def closestoragepv(self):
                self.dpv.close()

        def showkeys(self):
                pub.useroutput('Keys from state DB\n')
                self.d=shelve.open(self.storename,writeback=True)
                keylist=list(self.d.keys())
                pub.useroutput(keylist)
                self.d.close()
                pub.useroutput('Keys from rule DB\n')
                self.dr=shelve.open(self.rulename,writeback=True)
                keylistr=list(self.dr.keys())
                pub.useroutput(keylistr)
                self.d.closer()
                pub.useroutput('Keys from problem-vision DB\n')
```

```
        self.dpv=shelve.open(self.pvname,writeback=True)
        keylistpv=list(self.dpv.keys())
        pub.useroutput(keylistpv)
        self.dpv.close()

def storedata(self,doctype,docc):
        message="Do You want to store your state-description? [Y,N] \n"
        pub.userinput(message)
        inp=pub.opt
        if inp == 'Y':
                self.openstorage()
                message="Here is the list of all stored state-descriptions so far :\n"
                pub.useroutput(message)
                klist=list(self.d.keys())
                pub.useroutput(klist)
                message="Enter a name for the state description to be stored."
                pub.userinput(message)
                fname=pub.opt
                self.closestorage()

def storedatar(self):
                self.openstorager()
                self.dr[axx.ruleDocName]=axx.rulesAll
                self.closestorager()

def loaddata(self):
                self.openstorage()
                message="Here is the list of all stored state descriptions so far :\n"
                pub.useroutput(message)
                klist=list(self.d.keys())
                pub.useroutput(klist)
                message="Enter a name for the document you want to load : \n"
                pub.userinput(message)
                fname=pub.opt
                docs=self.d[fname]
                self.closestorage()
                aas.stateAll=docs

def loadndata(self):
                self.openstorage()
                message="Here is the list of all stored state descriptions so far :\n"
                pub.useroutput(message)
                klist=list(self.d.keys())
                pub.useroutput(klist)
                aas.stateAll=set()
                SLoop='Y'
                while SLoop == 'Y':
                        message="Enter a name for a state description you want to load : \n"
                        pub.userinput(message)
                        fname=pub.opt
                        docs=self.d[fname]
                        aas.stateAll=aas.stateAll.union(docs)
```

```
                        message='Your State Description document is as follows :\n'
                        pub.useroutput(message)
                        pub.useroutput(aas.stateAll)
                        message="Do You want to load another document (Y,N)? :\n"
                        pub.userinput(message)
                        SLoop=pub.opt

                self.closestorage()


        def loaddatar(self):
                self.openstorager()
                message="Here is the list of all stored rule documents so far :\n"
                pub.useroutput(message)
                klist=list(self.dr.keys())
                pub.useroutput(klist)
                axx.rulesAll=[]
                SLoop='Y'
                while SLoop == 'Y':
                        message="Enter a name for the rule document you want to load : \n"
                        pub.userinput(message)
                        fname=pub.opt
                        docx=self.dr[fname]
                        axx.rulesAll.extend(docx)
                        message='Your Rule document is as follows :\n'
                        pub.useroutput(message)
                        pub.useroutput(axx.rulesAll)
                        message="Do You want to load another rule document (Y,N)? :\n"
                        pub.userinput(message)
                        SLoop=pub.opt

                self.closestorager()


########################################################################
# CLASS INSTANCES

ast=Start()

ap=Actor("Here you can describe your problem and your vision with regard to different questions.")
app=Problem("")

ass=Actor("Here You can describe an actual state S related to your problem.")
aas=AState()
ax=Actor("Here You can edit some change rules X to apply to an actual state S.")
axx=Xrules()
asim=Actor("Here You can run a simulation SIM to check what happens with your initial state S
when the change rules X will be applied repeatadly on the state S.")
assim=Simulation()

aev=Actor("Here some advice will be given how to organize an evaluation EVAL of a realized
simulation SIM.")
```

```
astp=Actor("This will stop the whole program.")

pub=Publish()

#######################################
# The following names for three data-bases can be change freely.
# They have no special meaning, only mnemonic reasons

st=Storage('STAT2','RULE2','PV2')

################################
# In case of windows 10 operating system you have to change the last line
# above as follows:
'''
st=Storage('STAT2.dat','RULE2.dat','PV2.dat')
'''
```