```
R"""An automata class for timed automata with input and output

Author : Anders Andersen
Created On : Mon Jul 10 02:26:32 1998
Last Modified By: Anders Andersen
Last Modified On: Thu Apr 29 10:59:57 1999
Status : Unknown, Use with caution!

Copyright © 1998, 1999 Lancaster University, UK and NORUT Information Technology Ltd., Norway. See
COPYING for details.

This module implements an automata class Automata. This is a class for timed automata with input- and
ouput-events that configure itself from a description in the FC2 common format for transition systems ("FC2:
Reference Manual, Version 1.1", E. Madelaine and R. de Simone, 1993). It uses the two modules fc2 and
fc2string to interpret the automaton description. A more detailed description is found with the implementation
of the Automata class.
"""


# String splitting/matching/searhing
from string import split, replace
import re

# Writing to stdout
import sys

# Random numbers
from whrandom import whrandom

# We need to do some timing
import time

# Timers are checked with an internal thread
import thread


# The FC2 common format parser and the string parser
from fc2 import *
from fc2string import *


# A random object
random = whrandom()

# Regular expression used to match values in automata expressions
re_exp_valu = re.compile(r'values\.([a-zA-Z]\w*)')

# Regular expression to match the type of a declared variable
re_exp_type = re.compile(r'^\s*(int|chan|clock)\s+(.+)$')


class AutomataError(Exception):
    R"""Exception specific for this module

    An AutomataError is thrown when error or exceptions specific for this module is generated.
    """
    pass


class Values:
    R"""A class for the name space of an automaton

    This is a container (name space) for all the values found in expressions in a given automaton. It includes
    two set of values, timers and other values. The distinction is needed because setting and getting the value
    of a timer must be synchronised with the real clock.

    """

    def __init__(self):

        # Initialize the name space
        self.__dict__["__timers__"] = {}
        self.__dict__["__values__"] = {}

    def __newtimer__(self, name):
        R"""Create a new timer

        Create a new timer and initialize it to current time.

        """
        self.__timers__[name] = time.time()

    def __getvalues__(self):
        R"""Get value names

        Get the name of all values stored in this name space.

        """
        return self.__values__.keys()

    def __gettimers__(self):
        R"""Get timer names

        Get the name of all timers stored in this name space.

        """
        return self.__timers__.keys()

    def __setattr__(self, name, val):
        R"""Set the value of an attribute

        All attributes of this object (name-space) is stored in either the __timers__ or the __values__
        dictionary. The timers have to be set relative to current time.

        """
        if self.__timers__.has_key(name):
            self.__timers__[name] = time.time() - val
        else:
            self.__values__[name] = val

    def __getattr__(self, name):
        R"""Get the value of an attribute

        All attributes of this object (name-space) is stored in either the __timers__ or the __values__
        dictionary. Timers are relative to current time.

        """
        if self.__timers__.has_key(name):
            return time.time() - self.__timers__[name]
        elif self.__values__.has_key(name):
            return self.__values__[name]
        else:
            raise NameError


class Automata:
    R"""A class for timed automata with input and output
```

You can use this class and Pythonized FC2 common format description (from the `fc2` module) to create a timed automaton that reacts to input signals (events) and internal timed events and can produce output signals (events).

The automaton must be initialised with a `send_event` method and a automaton description. The `send_event` method is called once for every output signal (message) that is produced. The `install` method can be used to install an automaton description.

The automaton is sarted withe the `run` method and stopped with the `stop` method. The `print_state` method is used to print the current state of the automaton and the `new_event` method is used to send a new event to the automaton. Every move and produced event (output message) generates an output to stdout if the `print_info` attribute is true (1).

```
    """

    # Set this to 1 to print debug information
    DEBUG = 0

    def __init__(self, send_event=None, fc2py=None):
        R"""Initialise the automaton
```

Initialise some start values and install the automaton description if it is given. Also save the reference to the `send_event` method if it is given. This is the method called when the automaton produces an (output) event.

```
        """
        self.name = ""
        self.vertice = {}
        self.current = ""
        self.namespace = {"values": Values()}
        self.print_info = 1
        self.running = 0
        self.event_lock = thread.allocate_lock()
        if fc2py:
            self.install(fc2py)
        if send_event:
            self.send_event = send_event

    def _debug(self, info, eol="\n"):
        R"""Print debug information
```

Print the debug information given in `info` if the `DEBUG` flag is set.

```
        """
        if self.DEBUG:
            sys.stdout.write(info + eol)
            sys.stdout.flush()

    def _num_timers(self, string):
        R"""Count number of timers
```

Count the number of timers in `string`.

```
        """
        num = 0
        pos = 0
        while pos < len(string):
            if re_exp_valu.search(string[pos:]):
                m = re_exp_valu.search(string[pos:])
                if m.group(1) in self.namespace["values"].__gettimers__():
                    num = num + 1
                pos = pos + m.end()
            else:
                return num
        return num

    def _install_vtests(self, name, tests):
        for (type, val) in tests:
            if type == "testop":
                if self._num_timers(jointest((type, val))) == 1:
                    self._debug("    Timer %s" % (jointest((type, val)),))
                    self.vertice[name]["timers"].append((val[1], val[2]))

    def _main_subnets(self, main):
        if isit(main, "label", "struct", "infix3"):
            infix3_list = getit(main, "label", "struct", "infix3")
            for (s, (e1, e2)) in infix3_list:
                if s == "<" and e1[0] == "constant":
                    if e1[1] == '_' and islist(e2, "ref"):
                        return map(lambda x: x[1], getlist(e2))
        return []

    def _addto_namespace(self, config_entry):
        ce_match = re_exp_type.match(config_entry)
        if ce_match:
            if ce_match.group(1) in ["int", "clock"]:
                for var in re_exp_comm.split(ce_match.group(2)):
                    if ce_match.group(1) == "clock":
                        self._debug("  A timer: %s" % (var,))
                        self.namespace["values"].__newtimer__(var)
                    else:
                        self._debug("  An int: %s" % (var,))
                        self.namespace["values"].__setattr__(var, 0)

    def _new_vertex(self, info):
        name = ""; tests = []
        for (type, value) in split_string(info, "values."):
            if type == "name":
                if not name:
                    name = value
            elif type == "test":
                tests.append(value)
        if name:
            self._debug("  Found vertex %s %s" % (name, map(jointest, tests)))
            self.vertice[name] = {}
            self.vertice[name]["test"] = map(jointest, tests)
            self.vertice[name]["edges"] = {}
            self.vertice[name]["timers"] = []
            self._install_vtests(name, tests)

    def _initial(self, automaton):
        astructs = automaton["tables"]["structs"]
        if isit(automaton, "label", "logic", "infix3"):
            for infix3 in getit(automaton, "label", "logic", "infix3"):
                (op, (exp1, exp2)) = infix3
                if op == ">" and exp1[0] == "string":
                    if exp1[1] == "initial" and exp2[0] == "ref":
                        (type, value) = astructs[exp2[1][1]]
                        if type == "string":
                            vlist = split_string(value, "values.")
                            if vlist[0][0] == "name":
                                self._debug(
                                    "  Automaton initial %s" %(vlist[0][1],))
                                return vlist[0][1]
        return ""

    def _vertex_name(self, vertex, automaton):
        if isit(vertex, "label", "struct"):
            exp = getit(vertex, "label", "struct")[0]
            if exp[0] == "ref":
                (type, value) = automaton["tables"]["structs"][exp[1][1]]
                if type == "string":
                    vlist = split_string(value, "values.")
                    if vlist[0][0] == "name":
                        return vlist[0][1]
        return ""

    def _edge_behavs(self, edge):
        behavs = []
        if isit(edge, "label", "behav"):
            exp_list = getit(edge, "label", "behav")
            for (type, value) in exp_list:
                if type == "ref":
                    behavs.append(value[1])
        return behavs

    def _edge_target(self, edge, automaton):
        try:
            expr = edge["target_vertice"]
            if expr[0] == "ref":
                (type, value) = automaton["tables"]["structs"][expr[1][1]]
                if type == "string":
                    vlist = split_string(value, "values.")
                    if vlist[0][0] == "name":
                        return vlist[0][1]
        except KeyError:
            pass
        return ""

    def _install_event(self, name, target, behavs, behav_list):
        events = []
        behav = {}
        behav["test"] = []
        behav["stmt"] = []
        behav["mesg"] = []
        behav["target"] = target
        for bi in behavs:
            for (type, value) in behav_list[bi]:
                if type == "name" or type == "evnt":
                    events.append(value)
                elif type == "test":
                    behav[type].append(jointest(value))
                else:
                    behav[type].append(value)
        if not events:
            events = [""]
        for event in events:
            if not self.vertice[name]["edges"].has_key(event):
                self.vertice[name]["edges"][event] = []
            self.vertice[name]["edges"][event].append(behav)
            self._debug("     <%s:%s> " % (
                event, self.vertice[name]["edges"][event]))

    def install(self, fc2py):
        R"""Install automaton

        Install an automaton description in this automaton. We try to do this in a way that makes the running of the automaton efficient (and not the installing). The automaton description is given in a Pythonized FC2 format (see the fc2 module).

        """

        # Can not install in a running automaton
        self.event_lock.acquire()
        if self.running:
            self.event_lock.release()
            raise AutomataError, "Can not install in a running automaton"
        if self.vertice:
            self.event_lock.release()
            raise AutomataError, "Automaton allready installed"
        self._debug("Install ", "")

        # Find name
        if isit(fc2py["net_table"], "label", "struct", "string"):
            str_list = getit(fc2py["net_table"], "label", "struct", "string")
            self.name = str_list[0]
        self._debug(self.name)

        # Find main (or not)
        main = {}
        if isit(fc2py["net_table"], "label", "hook", "infix3"):
            infix3_list = getit(
                fc2py["net_table"], "label", "hook", "infix3")
            for (s, (e1, e2)) in infix3_list:
                if s == ">" and e1[0] == "string":
                    if e1[1] == "main" and e2[0] == "ref":
                        main = fc2py["net_table"]["net_list"][e2[1][1]]
                        self._debug("Found main")
                        break
        if not main:
            raise AutomataError, "No main net found inf fc2 structure"

        # Goto main and find automaton and synchronisation vectors
        automaton = {}; synch_vectors = []
        if isit(main, "label", "hook", "string", "automaton"):
            automaton = main
            self._debug("Main is automaton")
        elif isit(main, "label", "hook", "string", "synch_vector"):
            self._debug("Main is synch_vector")
            for ni in self._main_subnets(main):
                if isit(fc2py["net_table"]["net_list"][ni],
                        "label", "hook", "string", "automaton"):
                    if not automaton:
                        self._debug("Found an automaton")
                        automaton = fc2py["net_table"]["net_list"][ni]
                elif isit(fc2py["net_table"]["net_list"][ni],
                          "label", "hook", "string", "synch_vector"):
                    self._debug("Found a synch_vector")
                    synch_vectors.append(fc2py["net_table"]["net_list"][ni])
        if not automaton:
            raise AutomataError, "No automaton found"

        # Find config information in the synchronisation vectors
        self._debug("Interpret synch_vectors")
        cl = []
        for net in synch_vectors:
            if isit(net, "tables", "structs", "string", "Config"):
                if isit(net, "tables", "behavs", "string"):
                    cl = cl + getit(net, "tables", "behavs", "string")
        for config in cl:
            for config_entry in split(config, ";"):
                self._addto_namespace(config_entry)

        # Find the vertice of the automaton
        self._debug("Interpret automaton")
        if isit(automaton, "tables", "structs"):
            for (type, info) in automaton["tables"]["structs"]:
                if type == "string":
                    self._new_vertex(info)

        # Find the behaviours of the automaton (used later in the edges)
        behav_list = []
        if isit(automaton, "tables", "behavs"):
            for (type, behav) in automaton["tables"]["behavs"]:
                if type == "string":
                    self._debug("  Found automaton behaviour: %s" % (behav,))
                    behav_list.append(split_string(behav, "values."))

        # Find the initial of the automaton
        self.current = self._initial(automaton)
        if not self.current:
            raise AutomataError, "No initial state found"

        # Find the edges of the automaton
        if automaton.has_key("vertice_table"):
            self._debug("  Automaton edges:")
            for vertex in automaton["vertice_table"]:
                name = self._vertex_name(vertex, automaton)
                if name and vertex.has_key("edge_table"):
                    self._debug("    Vertex %s: " % (name,))
                    for edge in vertex["edge_table"]:
                        behavs = self._edge_behavs(edge)
                        target = self._edge_target(edge, automaton)
                        if behavs and target:
                            self._install_event(
                                name, target, behavs, behav_list)
        self.event_lock.release()

    def _print_state(self):
        R"""Print current state
```

Print information about the current state. This includes the current state itself and all the timers and the values.

```
        """
        sys.stdout.write("State %s: %s" % (self.name, self.current))
        for var in self.namespace["values"].__getvalues__():
            sys.stdout.write(
                ", %s=%d" % (var, self.namespace["values"].__getattr__(var)))
        for var in self.namespace["values"].__gettimers__():
            sys.stdout.write(
                ", %s=%f" % (var, self.namespace["values"].__getattr__(var)))
        sys.stdout.write("\n")
        sys.stdout.flush()

    def _print_move(self, event):
        R"""Print a move

        Print a move including the evnent that produced the move and the new state.

        """
        if self.print_info:
            sys.stdout.write("%s moved: -- %s --> %s.\t" % (
                self.name, event, self.current))
            sys.stdout.flush()
            self._print_state()

    def _print_mesg(self, mesg):
        R"""Print an (output) event

        Print an (output) event produced by this automaton.

        """
        if self.print_info:
            sys.stdout.write("%s mesg: %s\n" % (self.name, mesg))
            sys.stdout.flush()

    def print_state(self):
        R"""Print current state

        A public available (and thread safe) method that prints the current state.

        """
        self.event_lock.acquire()
        self._print_state()
        self.event_lock.release()

    def _init_timers(self):
        R"""Initialise all timers

        Initalise all the timers in the automaton to zero.

        """
        for var in self.namespace["values"].__gettimers__():
            self.namespace["values"].__setattr__(var, 0)

    def run(self):
        R"""Run the automaton

        Run (start) the automaton. This initalise all the timers to zero before the automaton is set in a
        running state.

        """
        self.event_lock.acquire()
        if not self.running:
            self.running = 1
            self._init_timers()
            self.event_lock.release()
            self.new_event()
        else:
            self.event_lock.release()
            raise AutomataError, "Automaton allready running"

    def stop(self):
        R"""Stop the automaton

        Set the automaton in a non-running state. New events will now produce an exception.

        """
        self.event_lock.acquire()
        if self.running:
            self.running = 0
        self.event_lock.release()

    def _test(self, test_exps):
        R"""Test if all tests are true

        Test if all tests in test_exps are true.

        """
        for test in test_exps:
            if not eval(test, self.namespace):
                return 0
        return 1

    def _select_edge(self, edge_list):
        R"""Select a valid edge randomly

        Select an edge randomly from the list of valid ones (with valid guards). Returns None if no valid
        edge is avilable.

        """
        valid_list = []
        for edge in edge_list:
            if self._test(edge["test"]):
                valid_list.append(edge)
        if valid_list:
            return valid_list[random.randint(0, len(valid_list) - 1)]
        else:
            return None

    def _checktimer(self, timers):
        R"""Produce an event when the next timer change

        This is started in a seperate thread an will try to produce an vent when the next timer goes of. The
        current implementation has some limitations (the timer expressions can not be to complex).

        """
        sleep_time = -1.0
        for (tl, tr) in timers:
            diff = abs(eval(tl, self.namespace) - eval(tr, self.namespace))
            if sleep_time < 0.0 or diff < sleep_time:
                sleep_time = diff
        time.sleep(sleep_time)
        self.new_event()

    def _send_mesg(self, mesg):
        R"""Produce an output event

        The automaton call this method when an output event (mesg)is produced. The only thing it does
        is to call the registered send_event method. If no method is registered, the event is ignored.

        """
        self._print_mesg(mesg)
        try:
            self.send_event(mesg)
        except AttributeError:            # self.send_event without value
            pass

    def new_event(self, event=""):
        R"""A new event for the automaton

        A new input event for the automaton. Each step (or move) in the automaton is the result of an event
        (either an input event or a timed event).

        """
        self.event_lock.acquire()
        if not self.running and event:
            self.event_lock.release()
            raise AutomataError, "Automaton not running"
        if not event:
            if self._test(self.vertice[self.current]["test"]):
                if self.vertice[self.current]["timers"]:
                    thread.start_new_thread(
                        self._checktimer,
                        (self.vertice[self.current]["timers"],))
                self.event_lock.release()
                return
        try:
            edge_list = self.vertice[self.current]["edges"][event]
        except KeyError:
            self.event_lock.release()
            return
        edge = self._select_edge(edge_list)
        if not edge:
            self.event_lock.release()
        else:
            self.current = edge["target"]
            for stmt in edge["stmt"]:
                exec(stmt, self.namespace)
            self._print_move(event)
            for mesg in edge["mesg"]:
                self._send_mesg(mesg)
            self.event_lock.release()
            self.new_event()
```