Apêndice A

Manual Técnico do PETRILab

Este capítulo tem como intuito apresentar a arquitetura base do PETRILab, de modo a permitir a implementação de novas funções ou correção de *bugs* por futuros desenvolvedores. O programa não é trivial, e é todo orientado a objetos, portanto o desenvolvedor deve estar bem familiarizado com esse tipo de programação. Também é recomendado um amplo conhecimento do módulo de interface gráfica do Pyhon, o *Tkinter*. A estrutura de cada módulo será explicada, assim como a relação entre eles.

O PETRILab consiste basicamente de 5 módulos, cada um contido em um arquivo separado no diretório principal. São eles:

- O módulo *CIPN*, que contém a classe da RPIC em si, e classes para temporizadores
- O módulo *ladder*, que contém as classes correspondentes a cada elemento do diagrama Ladder
- O módulo *conversao*, responsável por fazer a conversão de uma RPIC em um diagrama Ladder
- O módulo *diagrama*, responsável por desenhar o diagrama gerado na área de desenho
- O módulo *petrilab*, que contém toda a interface gráfica do programa

Este capítulo está estruturado da seguinte forma: Na seção A.1 será apresentado o módulo *CIPN.py*; na seção A.2 será apresentado o módulo *ladder*; na seção A.3 será apresentado o módulo *conversao*; na seção A.4 será apresentado o módulo *diagrama*; por fim, na seção A.5 será apresentado o módulo *petrilab*.

A.1 Módulo CIPN

O módulo CIPN está contido no arquivo *CIPN.py* no diretório principal do PETRILab. Sua principal classe é a classe CIPN, que cria um objeto correspondente a uma RPIC. Como mostra a Figura A-1, seus argumentos de criação são basicamente os da RPIC apresentada na seção 2.4, com exceção de *parent*, que, em caso de uso da simulação da interface gráfica, precisa conter uma referência ao objeto *Programa* do programa principal.

```
class CIPN():
    def __init__(self, Pc, Tc, Prec, Postc, x0c, Inc, C, Ec, lc, D, ld, A, la, parent=None):
        self.parent = parent
        self.x0c = x0c
        self.Pc, self.Tc = Pc, Tc
        self.Prec, self.Postc, self.Inc = Prec, Postc, Inc
        self.C, self.Ec, self.lc, self.D = C, Ec, lc, D
        self.ld, self.A, self.la = ld, A, la
        self.x, self.toFire = x0c, [False]*Tc
        self.timers = []
        for t in range(Tc):
            self.timers.append([])
        self.eimpulse = [1]*len(Ec)

    def setImpulse(self, e, x):...
    def isFireable(self, t):...
    def setFireable(self):...
    def setCondition(self, c, b):...
    def initialize(self):...
    def fire(self, t):...
    def event(self, e):...
    def run(self):...
    def check_timers(self, t):...
    def __str__(self):...
    def __repr__(self):...
```

Figura A-1: Classe CIPN

A função *initialize()* serve apenas para verificar se alguma ação deve ser acionada assim que a rede é criada. Já a função *setImpulse()* tem como objetivo definir o tipo de borda de um determinado evento: subida ou descida.

Tendo sido criado o objeto da rede, o estado da rede evoluirá na ocorrência de eventos ou mudança de condições, que são representados pela ocorrência das funções *event()* e *setCondition()*, respectivamente. Basicamente, na ocorrência de uma das duas funções supracitadas, o programa modifica uma *tag* correspondendo a ocorrência de um evento *e* ou uma condição *c*, e chama a rotina *setFireable()* para definir quais transições estão aptas a disparar após essa mudança, utilizando a função *isFireable()* para os testá-las individualmente. Feito isso, o programa chama a função *fire()* para realizar o disparo

das transições, modificando o vetor de estados x, e para acionar as ações que devem ser ativadas. A Figura A-2 mostra a função *isFireable()* expandida. Note que a função é bem simples, sendo puramente um teste das condições de habilitação e disparo de uma transição *t*.

```
def isFireable(self, t):
    for i in range(self.Pc):
        if self.x[i]<self.Prec[i][t]:
            return False
        if self.Inc[i][t]:
            if self.x[i]>=self.Inc[i][t]:
                return False
    if not self.C[self.lc[t][0]] or not self.Ec[self.lc[t][1]]:
        return False
    return True
```

Figura A-2: Função isFireable() da classe CIPN

Em certos casos, as transições de uma rede podem ser disparadas em sequência. Isso ocorre quando elas não estão associadas a eventos, e foram habilitadas depois do disparo de outra transição. Por esse motivo, o programa não pode executar apenas um ciclo de evolução de estados, sendo necessário checar repetidamente se existem transições a serem disparadas após a ocorrência de um evento. Isso é alcançado através da função *run()*, a função que de fato é chamada quando um evento ocorre na interface. A função pode ser vista na Figura A-3.

```
def run(self):
    contador=0
    while self.toFire != [0]*self.Tc:
        contador += 1
        if contador > 2000:
            abortar = askquestion('Looping infinito detectado',
                                  'usuário.
                                  'Deseja ab
            if abortar=='yes':
                self.parent.parar()
                contador=0
                return
            else:
                contador=0
        for t in range(self.Tc):
            if self.toFire[t] and self.D[self.ld[t]]==0:
                self.fire(t)
            elif self.toFire[t] and self.D[self.ld[t]] !=0:
                self.timers[t]+=[Delay(self, t)]
                self.timers[t][-1].start()
```

Figura A-3: Função run() da classe CIPN

Para a simulação dos temporizadores, são criados objetos da classe *Delay*. Essa classe é uma subclasse de *Thread*, a classe do Python que gerencia novas tarefas. Sendo assim, o programa se torna multitarefa, ficando muito mais propenso a travamentos. É sempre boa prática finalizar as tarefas não utilizadas para evitar esse problema. A classe *TimerAct*, também derivada de *Thread*, é apenas um delay para controlar o tempo que a luz indicadora de ação acionada fica acesa na interface gráfica.

A.2 Módulo ladder

O módulo *ladder* é o responsável por criar as classes dos elementos do diagrama Ladder, e está contido no arquivo *ladder.py* no diretório principal. Os objetos dessa classe são criados com parâmetros referentes ao tipo de elemento e os rótulos necessários para sua exibição.

Todos os objetos das classes desse módulo contém a função *draw()*, que tem como argumentos um objeto de *Canvas* – correspondente a uma área de desenho no módulo de interface gráfica do Python – e as coordenadas *x* e *y* que o objeto deve ser desenhado. A Figura A-4 mostra essa função expandida para um objeto correspondente a um bloco COMP.

```
class Comp:
    def __init__(self, l, t, q):...
    def __str__(self):...
    def draw(self, c, xy):
        R = 20
        c.create_rectangle(xy[0],xy[1]-R,xy[0]+3*R,xy[1]+R)
        c.create_text(xy[0]+1.5*R,xy[1]-R/2., text=self.s)
        c.create_text(xy[0]+1.5*R,xy[1]+R/2., text=self.s2, font=('Purisa',8))
        return [xy[0]+3*R,xy[1]]
    def size(self):...
```

Figura A-4: Classe COMP do módulo ladder

A função *size()* retorna as dimensões do bloco, para posterior utilização pelo módulo *diagrama*, responsável pelo deseho. A função *__str__()* representava os objetos na forma de texto, sendo utilizada em versões anteriores do PETRILab, e pode ser ignorada ou apagada.

A.3 Módulo conversao

O módulo conversão é responsável por realizar a conversão de RPIC em diagrama Ladder proposta no Capítulo 4, e está contido no arquivo *conversao.py*.. Ele consiste basicamente de uma única função, *convert()*, que leva como argumentos um objeto CIPN do módulo CIPN, e uma lista com os rótulos dos elementos, no formato mostrado no comentário acima da função. A Figura A-5 mostra esse comentário, além da primeira parte da função, que faz a criação do *Módulo dos Eventos Externos*.

```
# labels = [[plabels],[tlabels],[elabels],[clabels],[alabels]]

def convert(pn, labels=None):
    # Modulo de eventos externos
    m1 = Ladder()
    l = Linha()
    for i in range(len(pn.Ec)-1):
        if True in map(lambda x: x[1]==i, pn.lc):
            if labels:
                l += Contato(labels[2][i], 1)
            else:
                l += Contato('S'+str(i), 1)
            if pn.eimpulse[i] == 1:
                l += Borda(1)
                if labels:
                    l += Bobina(labels[2][i]+'r', 1)
                else:
                    l += Bobina('S'+str(i)+'r', 1)
            else:
                l += Borda(0)
                if labels:
                    l += Bobina(labels[2][i]+'f', 1)
                else:
                    l += Bobina('S'+str(i)+'f', 1)
            m1 += l
            l = Linha()
```

Figura A-5: Função *convert()* do módulo *conversao*

Note que a conversão segue exatamente o esquema proposto no Capítulo 4. Primeiramente, o programa cria uma instância de um diagrama Ladder e de uma linha. Para cada evento da rede, o programa checa se ele está associado a alguma transição, para então começar a inserção de elementos em sua linha correspondente. Um contato é então inserido, seguido de um contato de borda, e finalizando com uma bobina.

O teste *if labels*, realizado diversas vezes nessa função pode confundir o desenvolvedor. Ocorre que, originalmente, existia a opção de não fornecer nenhum rótulo dos elementos à função *convert()*; nesse caso, a função gerava automaticamente rótulos com numeração ordenada para eles. Com a criação da interface gráfica, esse caso nunca irá ocorrer, portanto os testes não são necessários.

Todos os outros módulos do diagrama a ser construído seguem o mesmo esquema de criação: criam-se as linhas necessárias, e adicionam-se os elementos, todos provenientes do módulo *ladder*. A função retorna então uma lista contento as linhas de todos os módulos do diagrama convertidos.

A.4 Módulo *diagrama*

O módulo *diagrama* é responsável por fazer o desenho do diagrama Ladder gerado em um *Canvas*. e está contido no arquivo *diagrama.py*. Ele contém uma classe *Diagrama*, que ao ser inicializada, cria uma janela e um *Canvas* através do módulo de interface *Tkinter*; e uma função *gerar()*, que é a responsável por fazer de fato o desenho do diagrama nesse *Canvas.*

A função *gerar()* leva como argumentos um objeto de RPIC, um objeto da classe *Programa*, presente na interface gráfica, e os rótulos dos elementos do diagrama Ladder. Internamente ele chama a função *convert()*, do módulo *conversao* apresentado na seção A.3, que retorna uma lista com os elementos do diagrama Ladder de cada módulo criado. A Figura A-6 mostra o trecho da função que desenha o Módulo da Inicialização.

```
# Desenho do Módulo 4

if len(b[3].x[0].x[1].e)>1:
    xy = app.xy
    xy = b[3].x[0].x[0].draw(app.c, xy)
    app.c.create_line(xy[0], xy[1], larg+5-b[3].x[0].x[1].size()[0], xy[1])
    xy = [larg+5-b[3].x[0].x[1].size()[0], xy[1]]
    xy = b[3].x[0].x[1].draw(app.c, xy)
    app.xy=xy
    if b[4].x:
        app.newline(70)
```

Figura A-6: Trecho da função de desenho do diagrama Ladder

Na Figura A-6, *b* representa a lista com os elementos de cada módulo. Sendo assim, *b[3]*, o quarto elemento da lista, corresponde ao objeto da classe *Ladder* – do módulo *ladder* – correspondente ao Módulo da Inicialização. A propriedade *x* dessa classe armazena as linhas do módulo, então *x[0]* acessa sua primeira linha – um objeto da classe *Linha*, do módulo *ladder* – que também tem uma propriedade *x* que armazena seus elementos. Acessando então o item *x[1]* dessa linha, obtemos o segundo elemento da linha, que, no caso do Módulo da Inicialização, é uma associação em paralelo, representada por um objeto da classe *Paralelo* do módulo *ladder*. A propriedade *e* desse objeto armazena os elementos associados em paralelo. Portanto, a função o teste *len(b[3].x[0].x[1].e)>1* testa se a quantidade de elementos em paralelo no Módulo da Inicialização é maior que um, pois se ela fosse igual a um – a bobina Set que está sempre presente – não haveria necessidade do desenho desse módulo!

As linhas restantes desse trecho de código basicamente chamam as funções *draw()* dos elementos do módulo, além de desenhar a linha que os une, através da função de desenhar linha do módulo *Tkinter*. A variável *xy* contém uma tupla que representa a posição atual de desenho do módulo no *Canvas*.

O restante do código segue o mesmo esquema do trecho mostrado, sendo necessária uma boa análise para entender o que de fato está sendo testado ou inserido. Vale notar que números os inteiros com valores aleatórios distribuídos pelo código representam as dimensões de alguns componentes, e foram descobertos a partir de tentativa e erro.

A.5 Módulo petrilab

O módulo *petrilab*, contido no arquivo *petrilab.pyw* é responsável pela criação da interface gráfica do programa. É um módulo extenso, contendo pouco mais de 2.400 linhas de código; por esse motivo, recomenda-se o uso de alguma interface de desenvolvimento que permita a minimização de funções e classes, de forma a facilitar a navegação pelo código.

O módulo implementa diversas classes, que contém instruções para o desenho, movimentação e remoção de diversos itens no *Canvas* principal. A Figura A-7 mostra a classe *Lugar*, com a função *mover()* expandida. Essa função supostamente é chamada ao

mover-se o mouse enquanto o botão esquerdo está pressionado sobre um lugar, portanto ela será chamada diversas vezes durante uma única movimentação. As linhas antes de código antes do *if* servem apenas para determinar a posição atual do ponteiro do mouse, levando em conta as barras de rolagem *sh* e *sv* da janela principal. As funções obscuras utilizadas para essa parte podem ser consultadas em alguma documentação do *Tk*, programa que originou o módulo *Tkinter* do Python. Em seguida, o programa apenas move os objetos relativos ao lugar no *Canvas*, chama a função *update_moving()* – que serve pra mover os textos das ações impulsionais associadas ao lugar – e a função *redraw()* de todos os arcos da rede, que os redesenha para acompanhar a movimentação do lugar.

```
class Lugar:
    def __init__(self, parent):...
    def follow(self, e,add=0):...
    def mover(self,add=0,xs=None,ys=None):
        self.parent.saved=False
        parent=self.parent
        a, b = self.parent.sh.get()
        perch = a/(1-b+a)
        a, b = self.parent.sv.get()
        percv = a/(1-b+a)
        xb = perch*(CANVASSIZEX-self.parent.c.winfo_width())
        yb = percv*(CANVASSIZEY-self.parent.c.winfo_height())
        x,y = parent.winfo_pointerx(), parent.winfo_pointery()
        x,y = x-parent.canv.winfo_rootx(), y-parent.canv.winfo_rooty()
        if xs is not None and ys is not None:
            x,y = xs+15, ys+15
            xb,yb = 0,0
        self.canvas.coords(self.oval, (x-15+xb,y-15+yb+add,x+15+xb, y+15+yb+add))
        self.canvas.coords(self.elabel, (x+xb, y+yb+add))
        self.canvas.coords(self.label, (x+xb, y-16+yb+add))
        self.update_moving()
        for i in self.parent.arcos+self.parent.inibidores:
            i.redraw()
```

Figura A-7: Classe *Lugar* do módulo *petrilab*

O programa principal, no entanto, está todo contido na classe *Programa*, que deriva da classe *Tk*, ou seja, representa uma janela. A método *__init__()* desse programa contém diversas linhas que contém: a definição das propriedades da janela, dos quadros de divisão de região dos botões (*Frames*) e das variáveis de controle; a criação de todos os menus *drop-down* e botões; e a associação das teclas de atalho à funções da classe. A Figura A-8 mostra um trecho que associa teclas de atalho a algumas funções.

```
# Teclas de atalho
self.bind('l', lambda x: self.lugar())
self.bind('t', lambda x: self.transicao())
self.bind('a', lambda x: self.arco())
self.bind('i', lambda x: self.inibidor())
self.bind('e', lambda x: self.evento())
self.bind('c', lambda x: self.condicao())
self.bind('k', lambda x: self.acao())
self.bind('g', lambda x: self.ladder())
self.bind('s', lambda x: self.simular())
```

Figura A-8: Trecho da criação de teclas de atalho da classe *Programa*

Como fica evidente na Figura A-8, a classe *Programa* tem diversas funções que implementam todas as ações de inserção e edição na interface principal. Tomemos como exemplo a função *inita()*, mostrada na Figura A-9. Ela é chamada ao clicar-se em qualquer lugar da área de desenho após clicar-se no botão de *Inserir Arco*, e leva como argumento uma tupla com as coordenadas atuais do mouse. As primeiras linhas do código desassociam o <*Duplo-Clique*>, <*Clique-Direito*> e <*Clique-Arrastado*> das suas funções originais de edição, para evitar edições acidentais ao inserir-se um arco. Em seguida, o programa chama uma função do *Canvas*, que detecta se o usuário clicou em cima de algum item. Em caso negativo, a variável *obj* conterá *None*, e a função terminará. Em caso positivo, a função então testa se o objeto clicado pertence a um lugar ou transição da rede; se pertencer, ele primeiramente testa se o arco é inibidor – terminando a função em caso positivo –, e, em seguida, armazena nas variáveis *ainit* e *arcstart* o objeto clicado. Por fim, ele muda o cursor para uma cruz menor, e associa o clique do mouse à definição do fim do arco, dada pela função *enda()*.

Muitas outras funções diferentes da do exemplo mostrado estão presentes nessa classe, mas não é o objetivo deste trabalho detalha-las. No entanto, todas elas seguem uma linha lógica não muito complicada, e o desenvolvedor será capaz de entende-las após estar familiarizado com o programa.

```
def inita(self, e):
    self.c.unbind('<Double-Button-1>')
    self.c.unbind('<Button-3>')
    self.unbind('<Button-3>')
    self.unbind('<Double-Button-1>')
    self.c.unbind('<B1-Motion>')
    obj = self.c.find_withtag(CURRENT)
    if obj:
        for i in self.transicoes+self.lugares:
            if obj[0] in i.items:
                if self.inserindoin and isinstance(i, Transicao):
                    return
                self.arcstart = i
                self.ainit=i
                self.configure(cursor='plus')
                self.c.bind('<Button-1>', self.enda)
                return
```

Figura A-9: Função *inita()* da classe *Programa*