Pràctica 5: Simulador de circuits digitals

Tecnologia de la Programació — iTIC

Aleix Llusà-Serra Sebastià Vila-Marta Marta Tarrés-Puertas

11 d'abril de 2016

Índex

Índex

1 Introducció

1.1 Objectius

Els objectius d'aquesta pràctica són:

- Prendre contacte amb els programes estructurats en base a classes d'objectes i, particularment, amb relacions d'herència.
- Consolidar l'ús de les eines de test i el disseny basat en tests.
- Consolidar l'ús de les eines de documentació de programari i de les eines de gestió de versions.

La pràctica té com a context la simulació de circuïts digitals i el seu fi és la implementació d'un simulador de circuïts combinacionals.

1.2 Condicions

- La pràctica té una durada de dues sessions de laboratori.
- Cal fer la pràctica amb l'equip de treball.
- Cal fer el desenvolupament usant control de versions amb subversion sobre el dipòsit. L'ús de l'eina serà part de l'avaluació de la pràctica.

1.3 Lliurament

Caldrà lliurar el resultat de la pràctica a través de l'activitat escaient d'Atenea. El lliurament haurà d'incloure:

- El codi font del projecte.
- Els doctests associats al codi font.
- La documentació del projecte escrita amb `Sphinx` que ha d'incloure una taula del temps de dedicació de cada persona de l'equip.
- El dibuix del circuit corresponent a la tasca 10.

2 Context

Un circuït digital combinacional està format per triports, elements de tres ports com ara portes AND i OR i nodes que permeten connectar un port de sortida amb un o més ports d'entrada. Amb aquests dos tipus d'elements es poden definir circuits amb una certa facilitat. Per representar aquests conceptes usarem dues classes principals Triport i Node i algunes subclasses que especialitzaran les anteriors com ara And o bé Or.

Els nodes tenen un estat (alt, baix o indefinit) que pot canviar al llarg del temps. Els triports saben consultar l'estat dels nodes als que estan connectats les seves entrades, fer els càlculs escaients i informar al node connectat a la seva sortida del resultat.

Amb les classes anteriors no n'hi ha prou per simular un circuit. Cal a més una certa infraestructura per poder simular el seu comportament. A tal efecte farem servir la classe Supervisor. Una instància de supervisor és un objecte que:

1. Coneix tots els nodes i triports del circuit a simular.
2. Rep missatges dels nodes en cada vegada que un node canvia d'estat.
3. Envia missatges als triports per tal que consultin els nodes d'entrada i actualitzin el node de sortida.

3 Arquitectura

L'aplicació consistirà en un conjunt de classes i un programa principal. Les classes són:

Estat La classe estat representa l'estat lògic d'un node (conductor). Segueix una lògica de tres estats: Alt, Baix i Indefinit.

Node La classe node representa un conductor que es troba en un cert estat. Un node pot connectar una sortida a una o més entrades.

Triport La classe triport representa un element actiu de tres ports. Essencialment sap consultar els nodes connectats a les entrades, calcular el resultat i informar al node connectat a la sortida.

And És una especialització de Triport que calcula l'AND lògic.

Or És una especialització de Triport que calcula l'OR lògic.

Entrada És un Node al qual no s'hi connecten dispositius d'entrada. S'usa per modelar les entrades del circuit.

Sortida És un Node al qual no s'hi connecten dispositius de sortida. S'usa per modelar les sortides del circuit.

Supervisor És la classe encarregada d'activar els triports d'un circuit de la forma adequada per tal de simular el seu comportament.

Més a més, l'aplicació tindrà un mòdul `main.py` que encabirà el programa principal.
Aquestes classes es relacionen seguint el diagrama UML de la figura 1.

4 Preparació de l'entorn de treball

Aquesta pràctica cal desenvolupar-la usant `subversion` i generar la documentació emprant `Sphinx`. Per treballar amb comoditat és molt important establir una estructura de directoris còmoda i raonable.

Tasca 1 Creeu un directori per desenvolupar la pràctica. A tal efecte, creeu un nou directori de nom `p5` dins el directori que teniu sota control de `subversion`. Afegiu-la subversion i feu commit.

Tasca 2 Afegiu ara l'estructura necessària per fer la pràctica.

1. Dins el directori `p4`, creeu un un subdirectori anomenat `src` en que hi anireu escrivint els fitxers font del projecte.

2. Dins de `p4` i usant l'ordre de `Sphinx sphinx-quickstart` creeu un directori per la documentació anomenat `doc` en el que escriureu la documentació usant les eines que ja coneixeu.

3. Creeu un capítol de la documentació en el que començareu a anotar el temps que dediqueu individualment a cada tasca.

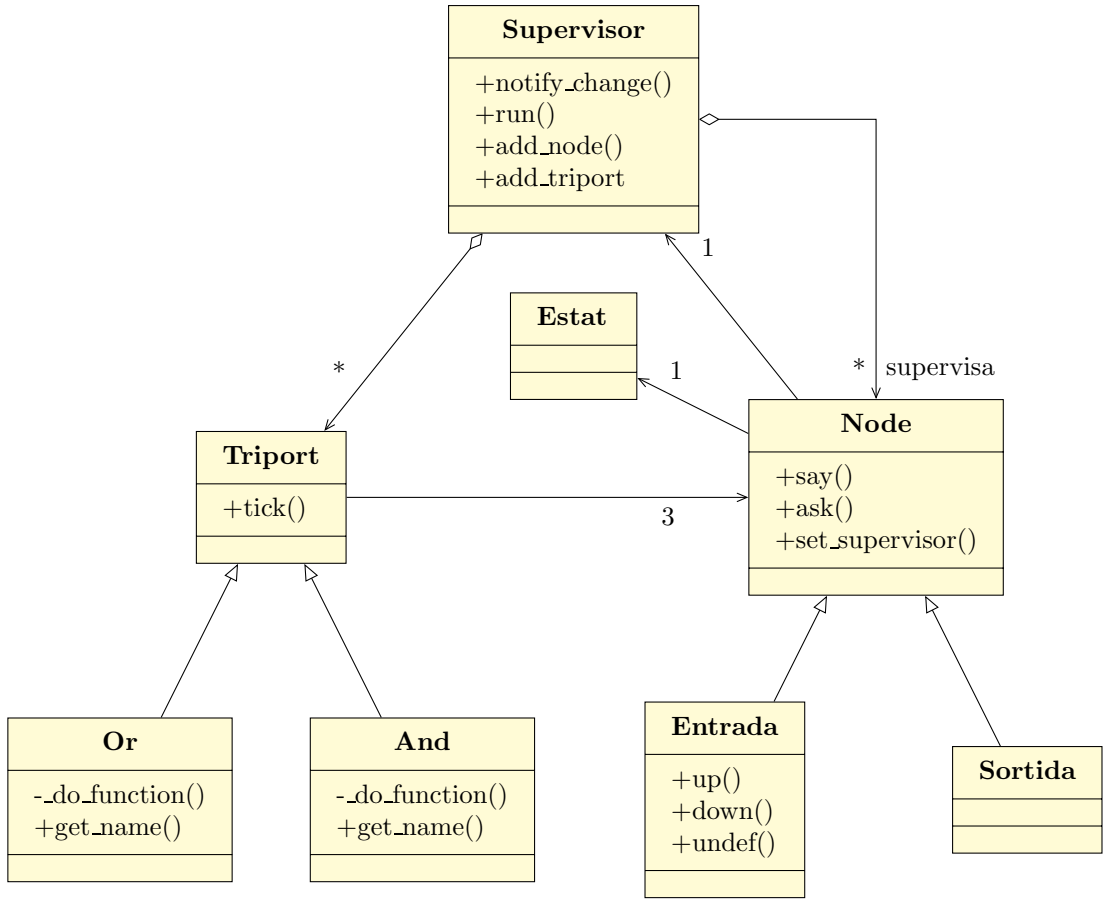

Figura 1: Diagrama de classes de la pràctica

Quan es treballa amb un sistema de gestió de versions com, per exemple, `subversion`, existeix una regla d'or que cal complir escrupulosament:

> En cap cas cal emmagatzemar en el sistema de versions fitxers que s'obtenen automàticament a partir d'altres fitxers del projecte. Per exemple, no han d'emmagatzemar-se els fitxers `*.pyc`, ja que s'obtenen automàticament a partir dels fitxers `*.py`. Tampoc han d'emmagatzemar-se els fitxers `pdf` que s'han obtingut a partir d'un document en format `rst` usant l'ordre `rst2pdf`.

En el cas de la documentació generada amb `Sphinx`, hi ha fitxers que són originals i altres que són generats. Només cal sotmetre al control de versions aquells fitxers que són originals i en cap cas la resta. La figura 2 mostra quins fitxers cal sotmetre normalment al control de versions quan es tracta d'una documentació `Sphinx`. Els fitxers i directoris que tenen el marc vers cal que siguin gestionats pel sistema de versions. Els grisos no. Tots els documents `rst` que s'aniran afegint al documentar en el directori `doc/source` cal que estiguin sota el control de `subversion`. En canvi, com es veu a la figura, cap dels fitxers o directoris que es troben dins de `src/build` han d'estar sota el control de `subversion` ja que es generen automàticament a partir del que hi ha a `doc/source`.

Tasca 3 tenint en compte el que s'acaba d'explicar, afegiu a subversion l'estructura del projecte inclosa la de la documentació `Sphinx`. Amb aquest pas, ja teniu l'entorn a punt per començar a

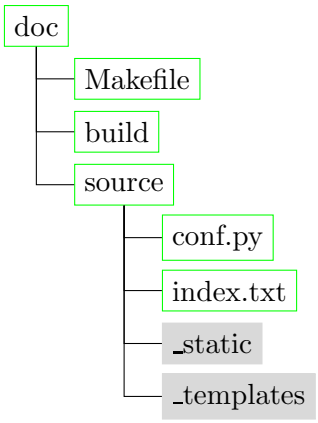

Figura 2: Configuració d'una documentsció `Sphinx` sota control de versions.

treballar en equip.

5 Mòdul estat

El mòdul `estat.py` conté la classe Estat. La classe és immutable i ha de tenir els següents atributs:

e [**int**, Privat] És el valor de l'estat: Baix(0), Alt(1), Indefinit(-1).

La classe disposarà d'un constructor que, per omissió, construirà un estat amb valor Indefinit. A més, usant mètodes especials, el dotarem de les operacions: and lògic (&), or lògic (|), negació lògica (~), i igualtat. Busqueu els operadors en la documentació de Python i implementeu-los per a la classe Estat amb el significat apropiat. Per tractar el valor Indefinit, considereu que qualsevol operació amb un o més operands indefinits, ha de retornar Indefinit.

També implementarà els mètodes següents:

- undef(self)

  que retorna **True** si self representa l'estat indefinit.

- __nonzero__(self)

  que retorna **True** so self està ben definit (Alt o Baix). Consulteu el significat d'aquest mètode a la documentació de Python.

- __repr(self)__

  Consulteu el paper d'aquest mètode especial a la documentació de Python i obreu de manera conseqüent. En què es diferencia de __str__?

Tasca 4 Implementeu la classe Estat. A continuació definiu els doctests corresponents, assegureu-vos que són complets i funcionen correctament. Finalment documenteu el codi correctament i annexeu aquesta documentació a la documentació del projecte usant `Sphinx`.

6 Mòdul node

6.1 Classe Node

El mòdul `node.py` implementa la classe Node. La classe ha de tenir els següents atributs:

e [Estat, Privat] Manté l'estat del node.

n [**str**, Privat] Manté el nom del node. És un atribut de caire informatiu.

s [Supervisor, Privat] Representa al instància de supervisor que cal informar si el node canvia d'estat.

La classe disposarà dels següents mètodes:

1. __init__(self,n)

   Inicialitza la instància donant-li el nom n i l'estat inicial a Indefinit.

2. say(self,e)

   El mètode s'usa per modificar l'estat del node self. Si el nou estat e és diferent de l'estat que tenia el node, aleshores cal informar al supervisor del canvi.

3. ask(self)

   Retorna l'estat del node.

4. set_supervisor(self, s)

   Informem al node que el seu supervisor és s.

5. __repr__(self)

   Retorna un **str** que representa el node convenientment.

TASCA 5 Definiu els doctests corresponents a la classe Node. Implementeu la classe i assegureu-vos que els tests són complets i funcionen correctament. Finalment documenteu el codi correctament i annexeu aquesta documentació a la documentació del projecte usant `Sphinx`.

6.2 Classes Entrada i Sortida

El mateix mòdul `node.py` conté les classes Entrada i Sortida, que són una especialització de la classe Node. Aquestes classes representen nodes del circuit amb la funció específica de fer d'entrada o de sortida.

Una i altra classes no afegeixen atributs però si afegeixen o redefineixen mètodes.

En el cas de la classe Entrada es defineixen els mètodes següents:

1. up(self)

   Força el node a estat alt.

2. down(self)

   Força el node a estat baix.

3. undef(self)

   Força el node a estat indefinit.

4. __repr__(self)

   Retorna un **str** que representa el node convenientment, deixant clar que es tracta d'una entrada.

En el cas de la classe Sortida es defineixen els mètodes següents:

1. __repr__(self)

   Retorna un **str** que representa el node convenientment, deixant clar que es tracta d'una sortida.

Noteu que en un i altre cas, el mètode __repr__ redefineix el de la superclasse.

TASCA 6 Definiu els doctests corresponents a les classes Entrada i Sortida. Implementeu les classes i assegureu-vos que els tests són complets i funcionen correctament. Finalment documenteu el codi correctament i annexeu aquesta documentació a la documentació del projecte usant `Sphinx`.

7 El mòdul triport

7.1 La classe Triport

El mòdul `triport.py` conté la classe Triport. La classe ha de tenir els següents atributs:

in1,in2 [Node, Privat] Són les instàncies de node als que estan connectades les entrades del triport.

out [Node, Privat] És el node al que està connectada la sortida del triport.

La classe disposarà dels següents mètodes:

1. __init__(self, i1, i2, o)

   Constructor. Crea un triport amb els node connectats a les entrades i sortida corresponents.

2. tick(self)

   Quan s'invoca el mètode, el triport fa un cicle de càlcul: llegeix l'estat dels nodes d'entrada, calcula la seva funció i notifica el resultat al node de sortida. El calcul el delega al mètode _do_function(self, i1, i2) que cada subclasse de Triport haurà de definir convenientment.

3. __repr__(self)

   Retorna un **str** que representa el triport convenientment.

TASCA 7 Definiu els doctests corresponents a la classe Triport. Implementeu la classe i assegureu-vos que els tests són complets i funcionen correctament. Finalment documenteu el codi correctament i annexeu aquesta documentació a la documentació del projecte usant `Sphinx`.

7.2 Les classes And i Or

En el mateix mòdul `triport.py` implementeu les classes And i Or, que són una especialització de la classe Triport. Una i altra classe difereixen només en la funció booleana que implementen.

No tenen atributs propis. Únicament tenen els atributs heretats. Disposen dels següents mètodes:

1. _do_function(self,e1,e2)

   És el mètode que fa el càlcul de la funció booleana corresponent. e1 i e2 són objectes de la classe Estat. La funció retorna el estat corresponent a operar e1 amb e2 mitjançant la funció booleana corresponent: and o or.

2. get_name(self)

   Retorna un **str** que representa el nom de la funció, i.e.: "And" o bé "Or".

Mireu d'entendre bé el paper del mètode _do_function, que s'usa mitjançant l'estratègia de delegació.

TASCA 8 Definiu els doctests corresponents a les classes And i Or. Assegureu-vos que són complets. Implementeu les classes i assegureu-vos que els tests passen correctament. Finalment documenteu el codi correctament i annexeu aquesta documentació a la documentació del projecte usant `Sphinx`.

8 El mòdul supervisor

Un supervisor és un objecte que està informat de tots els nodes i triports que formen un circuit. La seva funció principal és «donar vida» a un circuit. El principi de funcionament és molt simple: el supervisor fa una ronda notificant a tots els triports del circuit, en un ordre arbitrari, que facin un «tick». Una vegada acabada la ronda, es comprova si algun node ha notificat canvi d'estat. En cas que sigui així, s'itera i s'inicia una nova ronda de ticks. Finalment, quan durant una ronda cap node canvia, la simulació acaba.

El mòdul `supervisor.py` conté la classe Supervisor. La classe ha de tenir els següents atributs:

nodes [**list**, Privat] Emmagatzema els nodes del circuit.

triports [**list**, Privat] Emmagatzema els triports del circuit.

changed [**bool**, Privat] Indica si algun node ha canviat d'estat durant la simulació.

La classe disposarà també dels següents mètodes:

1. __init__(self)

   Constructor. Crea un supervisor buit, sense nodes ni triports.

2. add_node(self,n)

   Afegeix el node n a la llista de nodes controlats pel supervisor. Addicionalment, s'informa al node a través del seu mètode set_supervisor de qui és el seu supervisor.

3. add_triport(self,t)

   Afegeix el triport `t` a la llista de triports controlats pel supervisor.

4. notify_change(self)

   Notifica al supervisor que un node ha canviat d'estat. Principalment utilitzada pels nodes.

5. run(self, log=**False**)

   Fa funcionar el circuit aplicant l'estratègia que s'ha explicat prèviament. Si log és **True**, llavors escriu per la pantalla un missatge cada vegada que provoca un tick en un triport seguint el següent esquema:

   ```
   Tick -> <Representació del triport>
   ```

TASCA 9 Definiu els doctests corresponents i assegureu-vos que són complets. Implementeu la classe i comproveu que els tests funcionen correctament. Finalment documenteu el codi correctament i annexeu aquesta documentació a la documentació del projecte usant `Sphinx`.

9 El mòdul main

El mòdul `main.py` s'usa per definir el circuit a simular, executar una simulació i escriure els resultats. Per exemple, si volguéssim simular un circuit que calcula una AND de quatre ports, el mòdul `main.py` podria ser:

```
from node import Node, Entrada, Sortida
from triport import And
from supervisor import Supervisor

# definim 4 entrades
e1 = Entrada('E1')
e2 = Entrada('E2')
e3 = Entrada('E3')
e4 = Entrada('E4')

# definim 1 sortida
s = Sortida('S1')

# definim el circuit
n1 = Node('N1')
a1 = And(e1, e2, n1)
n2 = Node('N2')
a2 = And(e3, e4, n2)
a3 = And(n1, n2, s)

# creem un supervisor i l'informem dels nodes i triports que ha de supervisar
sup = Supervisor()
sup.add_node(n1)
sup.add_node(n2)
```

```
sup.add_triport(a1)
sup.add_triport(a2)
sup.add_triport(a3)

# fixem les entrades i simulem
e1.up()
e2.up()
e3.down()
e4.up()
sup.run()

# escrivim resultats
print s
```

TASCA 10 Mireu d'entendre quin paper fa cada element en aquest exemple i quin circuit s'està simulant. Cal lliurar el dibuix del circuit que s'està simulant.

TASCA 11 Implementeu el mòdul `main.py`. Proveu després de simular alguns circuits de la vostra collita. Lliureu els circuits que heu simulat.

10 Comentaris

Aquesta pràctica té una certa connexió amb VHDL i les eines de simulació de circuits digitals. Fixeu-vos que el concepte de "node" es correspon aproximadament amb el concepte de "Signal" en VHDL. D'altra banda, un triport es correspondria amb un "Element" de tres ports, dos d'entrada i un de sortida. Per altra banda, la definició de un circuit que fem en el main, té una certa similitud amb l'assemblat de components que pot fer-se a VHDL per definir components més grans.

11 Ampliacions

1. Tal i com s'ha definit el significat de les operacions and i or en la classe Estat, retornen indefinit si alguna de les entrades és indefinida. Tot i que això simplifica les coses, en certs casos es pot afinar una mica més. Per exemple, quan estem calculant $a \cdot b$ i a és indefinit i b zero, sabem segur que la sortida serà zero.

   Mireu d'aprofitar aquests casos especials per refinar les operacions booleanes de la classe Estat. Amb això, l'algoritme de simulació pot convergir més de pressa en alguns casos.

2. Com l'objecte supervisor té constància dels nodes del circuit i pot saber quins són Entrades o Sortides, seria útil afegir al supervisor un nou mètode print_outputs(self) que en invocar-lo escrigués l'estat de totes les sortides del circuit. Implementeu-lo i modifiqueu el `main.py` convenientment.

3. Els mètodes per afegir nodes o triports del supervisor són millorables. Per un costat, seria interessant reduir-los a un únic mètode add que servís tant per nodes com per triports. Com faries aquesta simplificació?

Per altra banda, una vegada feta la simplificació anterior, una nova millora seria poder invocar el mètode add passant com argument una llista de nodes i/o triports indistintament o bé un únic node o triport. Com faries aquest nou canvi? Et simplifica la descripció del circuit?

4. Segurament has simulat un circuit simple. Seria interessant simular circuits més complexos. Te'n proposem dos:

   a) Un "ripple carry adder". Per conèixer el circuit consulta

      `http://en.wikipedia.org/wiki/Adder_(electronics)`.

   b) Un "carry look-ahead adder". Per conèixer el circuit consulta

      `http://en.wikipedia.org/wiki/Adder_(electronics)`.

   Quin triga més temps a simular-se? T'atreviries a modificar la classe Supervisor per tal que comptés el nombre d'iteracions que fa abans de donar per acabada la simulació i es pogués consultar amb un mètode? De què deu dependre aquest valor?

TASCA 12 Escolliu una de les tres primeres ampliacions i implementeu-la. Opcionalment implementeu-ne alguna altra.

TASCA 13 Recordeu a lliurar la documentació sphinx, que inclogui doctests, juntament amb un petit joc de proves per cada mòdul, que en demostri el seu funcionament.