Universitat Autònoma de Barcelona

escola d'enginyeria

2022: EINA GRÀFICA PER LA CREACIÓ D'AUTÒMATS PER A LA REPRESENTACIÓ DEL LLENGUATGE NATURAL

Memòria del Projecte Fi de Carrera
d'Enginyeria en Informàtica
realitzat per
J. Oriol Aguadé Estivill
i dirigit per
Marc Ortega Gil
Bellaterra, 21 de Juny de 2010

Agraïments

Volia agrair a Marc Ortega les ganes i l'esforç que ha utilitzat en ajudar-me a portar a terme el desenvolupament d'aquest projecte. Ha estat un excel·lent tutor i m'ha permès aprendre molt d' ell.

Índex

Capítol 1
Introducció

Aquest PFC es centra en la generació d'una eina gràfica multiplataforma de creació i edició de gramàtiques electròniques per representar el Llenguatge Natural com un transductor.

Per computar el significat del LN primer s'ha de representar per poder-lo analitzar. Una representació àmpliament estesa és convertir la seva estructura en un autòmat per a continuació ser processat amb programes d'anàlisis d'autòmats. El projecte tracta l' implementació d'una eina per la tasca de representació d'aquests autòmats.

1.1 Objectius

Com s'ha introduït s'ha d'implementar un eina capaç de representar i crear la definició d' autòmats que representen el LN també coneguts com gramàtiques electròniques.

L'eina a desenvolupar ha de ser una interfície gràfica multiplataforma capaç de crear i modificar transductors. S'han de poder representar autòmats finits afegint, canviant de posició i esborrant estats i transicions. Els estats i transicions utilitzaran l'estructura determinada per la sintaxi pròpia del Spanish FraneNEt Project.

L'eina ha de ser capaç de guardar i obrir els autòmats creats per l'usuari. El format a utilitzar per guardar serà el Format Autòmat. Per aconseguir una eina més polivalent es permetrà l' importació i exportació d'autòmats a partir d'Expressions Regulars.

L' eina va dirigida a un públic amb coneixements bàsics d'informàtica per tant és una prioritat crear una interfície còmode i intuïtiva per l'usuari. S'organitzarà el contingut amb una estructura senzilla perquè l'usuari no tingui cap problema alhora de treballar amb ella. Tot i així s'inclourà un manual d'usuari perquè l'usuari tregui tot el rendiment possible al programa. Aquesta eina va destinada a lingüistes per tant s'utilitzarà la representació en el format de "caixes" per representar els autòmats d'una forma més visual que l'estàndard. Un cop finalitzat el projecte l'eina serà utilitzada en el marc del projecte de recerca Spanish FrameNet Project.

L'objectiu principal de l'eina és la creació i edició d'autòmats però també és premia l' implementació d'opcions variades que proporcionin a l'eina funcionalitats útils per l'usuari.

1.2 Organització de la memòria

La memòria està organitzada de tal manera que el pròxim capítol tracta els temes referents a la descripció del projecte. Aquest apartat està dividit en fonaments teòrics, disseny i planificació.

El següent capítol fa referència a l' implementació de l'eina. El tractament de l' implementació inicialment es centra en la representació de l'escena i els elements que s'hi representen. L'escena és el marc de treball de la nostra eina un cop coneguda es passa a explicar les aplicacions que la complementen i que proporcionen les eines necessàries per treballar. Aquest anàlisis s' enfoca des del marc més general del programa a cada component específic que engloba l'aplicació.

L'últim capítol són les conclusions del projecte on s'analitzarà el treball realitzat, la funcionalitat del programa final i les seves aplicacions, el destí de l'aplicació i les possibles millores.

Capítol 2
Descripció del projecte

2.1 Fonaments teòrics

Per entendre l'objectiu del projecte s'ha de saber que una gramàtica electrònica és un transductor (un autòmat finit que quan reconeix una cadena d'entrada determinada pot retornar una sortida). Les gramàtiques electròniques permet el reconeixement d'estructures sintàctiques i marcar-les o manipular-les una vegada reconegudes mitjançant la funció de sortida del transductor.

Partint que una frase en Llenguatge Natural (LN) té una estructura determinada i les seves paraules tenen un significat el qual només és complet quan es relaciona amb la resta de paraules de la frase, fem una distinció entre els elements de la frase que poden ser fixes o opcionals.

Per exemple, la frase "estar al carrer" té:

- Elements fixes: estar/al/carrer
- Elements opcionals: elements interns com adverbis. Estar *sempre* al carrer

L'avantatge de tractar el Llenguatge Natural és que podem representar la frase com una Expressió Regular i, per tant, també com un autòmat finit.

Com s'ha introduït abans, per reconèixer o representar aquestes estructures s'utilitzen els transductors. Mitjançant una combinació d'estats i transicions és possible representar els elements fixes i opcionals de la frase que s'està tractant a més de poder associar una sortida de manera que si una cadena d'entrada arriba a estat final, rebrem la sortida que havíem associat al recorregut dels estats pels que ha passat.

2.1.2 SFN

Aquest projecte vol crear una aplicació que serà utilitzada en el marc del projecte de recerca Spanish FrameNet Project (SFN)[1].

SFN és un projecte que es desenvolupa a la Universitat Autònoma de Barcelona cooperant amb l'International Computer Science Institute[2] (Berkeley, CA) en cooperació amb el FrameNet Project[3].

Aquest projecte té com a finalitat la creació d'una base d'informació lèxica on-line basada en la teoria de marcs conceptuals (frame semantics) de Fillmore (Fillmore 1976, 1977, 1992).

La idea bàsica dels marcs conceptuals és que no es pot entendre el significat d'una sola paraula sense tenir accés a tot el coneixement essencial que es relaciona amb aquesta paraula. Per exemple, no es pot entendre la paraula "vendre" sense conèixer el context que inclou la situació que es dóna ,entre altres aspectes, amb el venedor, el comprador, la mercaderia, els diners, la relació entre el venedor i el producte, etc.

Les paraules han de tenir un marc amb un coneixement semàntic associat segons el concepte a que es refereix. La definició de marc conceptual consisteix en que una estructura coherent de conceptes relacionats de tal manera que, sense coneixement de tots ells, un no pot entendre el significat complet d'un concepte en concret.

SFN té un conjunt finit de tags per definir les paraules en la seva base d'informació lèxica. Coneixent aquests tags els implementarem en la nostra aplicació perquè la creació de transicions per part dels usuaris sigui lo més ràpida i còmode possible oferint-los les diferents opcions disponibles.

2.1.3 Estat de l'art

Hi ha publicacions i eines que treballen sobre autòmats aplicats a la representació del Llenguatge Natural un d'ells per exemple és el de M. Ortega amb "Teoría de Autómatas aplicada a la Lingüística Informática". *Facultat de Ciències. Secció d'Enginyeria Informàtica,* Universitat Autònoma de Barcelona, 1997.

---

[1] http://gemini.uab.es/SFN
[2] http://www.icsi.berkeley.edu/
[3] http://framenet.icsi.berkeley.edu/

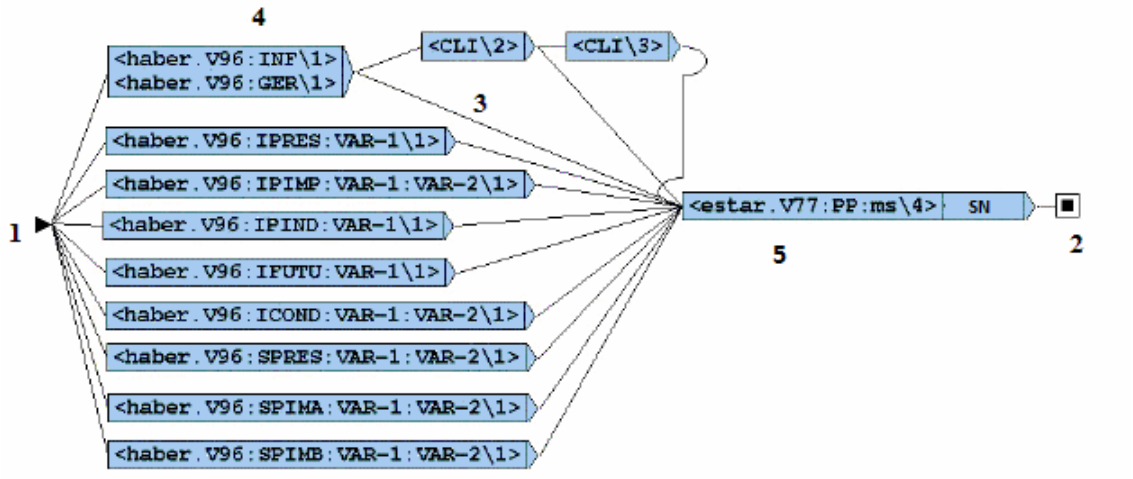
L'eina que crearem tracta de renovar i millorar aquestes eines, per tal proporcionar una aplicació amb una interfície gràfica a l'ordre del dia que estigui organitzada adequadament permetent treballar amb ella còmodament.

La nostra aplicació incorpora els diferents tags d'informació lèxica i altres novetats per tal de proporcionar una eina per lingüistes el més complerta possible.

2.1.4 Representació dels autòmats en format de "caixes"

Els autòmats que la nostra aplicació utilitzen una representació més visual que té unes característiques determinades. Les transicions són uns elements formats per una caixa que contenen l' informació del token que produeix la transició entre estats absoluts. Els estats absoluts no es veuen representats a simple vista en l'edició però s'han d'analitzar per guardar i obrir els autòmats. Els estats del nostre autòmat estan representats per arestes que uneixen les diferents transicions.

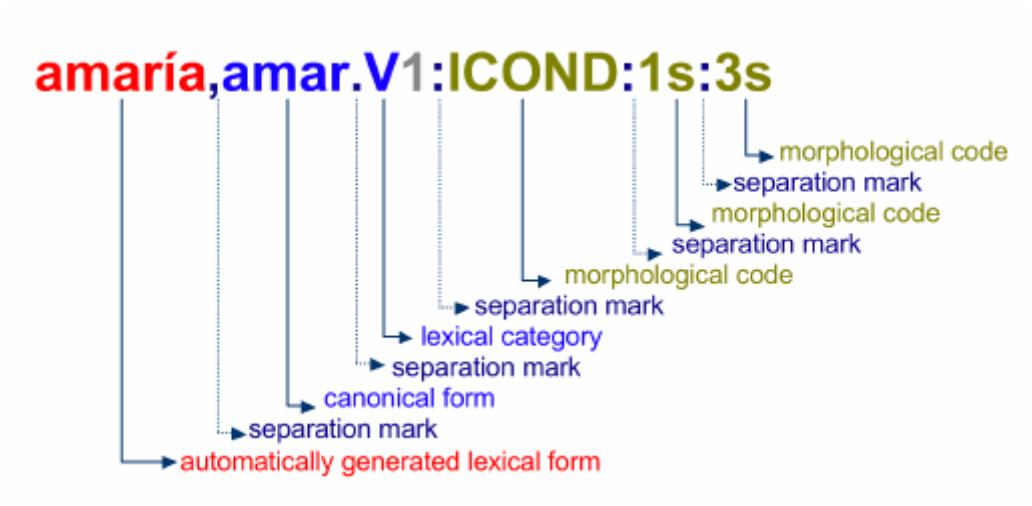
A continuació es mostra un exemple d'autòmat representat segons aquest format per observar exactament quins elements s'han de representar.



1) El triangle horitzontal representa l'estat inicial
2) El quadrat negre que es troba dins del quadrat blanc representa l'estat final
3) Les arestes són els estats
4) Les capses són les transicions.
5) Observem que algunes transicions estan formades per dos rectangles contigus. En el primer rectangle s'hi especifica l' informació lèxica que defineix cada paraula. En el segon rectangle s'hi guarda la sortida del transductor.

2.1.5 Sintaxi pròpia del SFN

Cada paraula inclosa en el corpus del Spanish FrameNet Project està definida per una sintaxis determinada :



Aquesta es pot simplificar en:

<forma . categoria : informació_morfològica>[sortida]

Com hem mencionat el tags de categoria i d'informació morfològica són finits i estan definits en la pàgina web de SFN (http://gemini.uab.es:9080/SFNsite/taggers-chunkers)

Fent un anàlisis de requeriments a l'hora de definir les paraules observem que quan volem definir un token o paraules no és necessari especificar la informació de tots els seus camps. Per guardar correctament l'autòmat es demanarà a l'usuari al crear una transició que especifiqui com a mínim la categoria gramatical.

2.2 Disseny

El punt fort de l'aplicació és la creació de l' interfície gràfica multiplataforma i amb possibilitat de que aquesta sigui web.

Una opció seria fer una aplicació flash ja que és multiplataforma i la l' implementació de l' interfície gràfica seria senzilla de crear.

Una altra opció per crear una aplicació multiplataforma seria implementar-la en Java. Java ens proporciona dos potents llibreries per GUI's: Swing i AWT. Un avantatge d'escollir aquest llenguatge de programació és la gran quantitat de documentació i exemples disponibles en llibres i Internet que facilitaran la recerca d'informació de com fer l' implementació.

L' API de Java proporciona una gran quantitat de classes que seran molt útils per facilitar-nos la nostra feina com a programadors.

A l'hora d'implementar la interfície gràfica, es necessiten finestres, quadres de diàleg, botons, llistes desplegables, menús i altres elements que estem acostumats a trobar en les aplicacions que utilitzem i que es troben en les llibreries Swing i AWT.

Aquestes dos llibreries contenen totes les classes necessàries pel desenvolupament d'Interfícies gràfiques per l'usuari. Les interfícies gràfiques implementades amb aquestes llibreries tenen una aparença i es comporten de forma semblant en totes les plataformes en les que s'executa.

La seva estructura bàsica gira en torn a "components i contenidors". Els contenidors contenen components i son components a la mateixa vegada, de manera que el tractament amb els diferents elements és bastant semblant.

Els primers components gràfics en Java pel desenvolupament de GUIs es trobaven en la llibreria Abstract Window Toolkit (AWT). Aquesta llibreria és adequada per a interfícies senzilles.

Al cap d'un temps ha aparegut la llibreria Swing que és més robusta, versàtil i flexible i utilitza components de la llibreria AWT per complementar-se. Aquesta última llibreria és la que utilitzarem per fer l' implementació de la nostra eina.

2.2.1 Què és el JFC i Swing?

L'abreviatura de JFC correspon a Java Foundation Classes. JFC es un entorn gràfic per construir interfícies gràfiques per usuaris multi plataforma basades en Java.

JFC es composa d'Abstract Windows Toolkit (AWT), Swing i Java 2D que ens proporcionen les eines adequades per construir una interfície d'usuari consistent per programes Java, tant si es executat sobre una plataforma Windows, Mac OS X o Linux.

AWT és l'API va ser la primera API d'interfícies que es va crear, va ser fortament criticada per ser poc més que un mascara sobre les capacitats gràfiques natives de la plataforma amfitrió.

En AWT els components amb els quals l'usuari interactua amb d'interfície gràfica o widgets depenen de les capacitats del widgets de la plataforma nativa.

Més endavant es va crear una API de gràfics alternatius anomenada Internet Foundation Classes que va ser desenvolupada per Netscape. Sun va barrejar la IFC amb altres tecnologies sota el nom de "Swing". Els elements de l'API de Swing comencen amb "javax.swing", i afegeixen la capacitat per un look&feel que permet als programes Swing mantenir la base del codi independent de la plataforma però pot imitar l'aparença de l'aplicació nativa.

L'API de Swing es presenta en el JDK 1.2 i en la JFC 1.1 i pot ser descarregada gratuïtament des de la pàgina web de Sun[4].

2.3 Planificació

Al principi, tasques simples porten més temps del necessari perquè a part de les tasques que puguin sorgir en tot moment s'està fent una recerca d'informació sobre l'API de Java i altres manuals sobre la programació de GUIs en Java.

S'han previst els punts negres del projecte: L' implementació de l' interfície gràfica és preveu que serà lenta degut a l'actual desconeixement i dels possibles problemes que puguin sortir. Per no malgastar temps s'anirà fent una recerca d'informació que ajudi en l' implementació de la següent tasca paral·lelament a l' implementació de la tasca en curs.

[4] http://java.sun.com/javase/downloads/index.jsp

Tasques a implementar:

- Recerca sobre manuals de Java sobre la programació de GUI's i crear entorn bàsic
    - Durada: una setmana
- Afegir un objecte en forma de rectangle que es pugui moure per la pantalla arrossegant-lo amb el ratolí
    - Durada: una setmana
- Crear les classes Transició (caixa) i estat (aresta) versió inicial
    - Durada: un dia
- Afegir a un lateral dos botons que permetin afegir estats i transicions.
    - Durada: una setmana
- Implementar opció perquè les línies dels estats puguin ser línies rectes o corbes
    - Durada: tres dies
- Als anteriors botons els hi assignem una icona
    - Durada: un dia
- Controlar posicions vàlides en el desplaçament dels rectangles
    - Durada: dos dies
- Crear els diferents tipus de transicions: estàndard, inicial, final i connector.
    - Durada: una setmana
- Creació dels menús bàsics
    - Durada: tres dies
- Control bàsic d'events
    - Durada: tres dies
- Crear menú amb informació lèxica per crear transicions
    - Durada: quatre dies

- Afegir estats entre transicions i eliminar-los.
    - Durada: tres dies
- Actualitzar posició d'estats segons moviment de les transicions
    - Durada: tres dies
- Crear finestra per modificar informació lèxica transicions.
    - Durada: quatre dies
- Implementar l'opció de guardar el transductor en format autòmat
    - Durada: una setmana
- Implementar l'opció de llegir un transductor en format autòmat
    - Durada: una setmana
- Implementar l'opció de llegir un transductor en format d'ER
    - Durada: una setmana
- Implementar l'opció de guardar el transductor en format d'ER.
    - Durada: una setmana
- Test i correcció d'errors
    - Durada: deu dies.
- Convertir l'aplicació en applet de Java
    - Durada: una setmana
- Afegir noves funcionalitats a l'eina
    - Durada: Temps restant

Capítol 3
Implementació

3. 1 Representació de l'escenari de disseny

Per començar tractarem el mòdul encarregat de la representació i edició de l'escena. Aquest mòdul està compost per les classes *ZonaDibuix.java, Aresta.java* i *Caixa.java*.

Una interfície gràfica es basa en una finestra que conté varis elements amb els quals podem interaccionar. Java ens proporciona la classe *JFrame*, una classe contenidor que necessita un gestor de Layout que s'encarregui de l'organització dels elements que conté. Aquest gestor pot canviar la posició dels components de la nostra aplicació però no permet dibuixar directament sobre ella per tant s'ha d'afegir un component sobre el qual dibuixar. El component que utilitzarem per aquesta finalitat és de la classe *JComponent*, que a continuació veurem que és una classe molt potent i que ens proporciona molta llibertat per representar els diferents elements.

ZonaDibuix.java és la classe que s'encarrega de la representació dels elements en l'escena i de la gestió dels events que es produeixen en ella. Per una banda aquesta classe és una extensió de la classe *JComponent*, que ens facilita en gran mesura la labor de representar per pantalla els estats i transicions d'una forma agradable i còmode. Per altra banda aquesta classe implementa les interfícies *MouseListener* i *MouseMotionListener* per donar cobertura al control i gestió d'events del ratolí perquè l'usuari pugui treballar i editar el contingut amb gran comoditat.

3.1.1 Extensió de la classe JComponent:

Com s'acaba d'introduir per poder dibuixar en una interfície gràfica s'ha de crear una classe que hereta de la classe *Canvas* (la traducció al castellà és "lienzo") o *JComponent* i entre altres coses s'ha de definir el mètodes **paint** i **update** (en el cas de la classe *Canvas*). Aquests mètodes reben un objecte de la classe *Graphics* sobre el qual es dibuixa l'escena que es vol mostrar i a continuació es mostra per pantalla. Perquè el nostre programa sigui més potent, aquest objecte és convertit a un objecte *Graphics2D*. Aquesta classe és una extensió de la classe *Graphics* per tant té tots els mètodes d'aquesta classe i d'altres que ens permeten canviar la forma en que dibuixem.

Per entendre la representació dels elements s'ha de conèixer la classe *Graphics*. La classe en qüestió és el sistema bàsic de totes les operacions gràfiques, és una classe abstracta que té l'objectiu de conformar el context gràfic (encapsular l' informació necessària que es vol mostrar per pantalla) i proporcionar els mètodes necessàries per dibuixar per pantalla. Per dibuixar es necessita una context gràfic vàlid que el rebem com un objecte de la classe *Graphics*. El problema es que al ser una classe abstracta, aquesta no es pot instanciar directament i per tant li hem de passar el context gràfic al programa a través dels mètodes *paint()* o *update()* o amb el mètode *getGraphics*() de la classe Component.

3.1.2 Pintar l'escena:

Els mètodes *paint(Graphics g)* i *update(Graphics g)* són els mètodes que podem definir en la nostra classe i juntament amb el mètode ***repaint()*** són els que s'encarreguen de pintar l'escena.

El mètode *repaint()* es crida en codi quan es necessita que un component es repinti, aquestes sol·licituds s'anomenen App-triggered Painting Operation i és el programador qui decideix repintar. Si és un component normal (JTable, JButton, JLabel, etc) no és necessari fer la crida ja que si es produeix algun canvi sobre algun d'aquests components automàticament ja es repinten. A diferencia dels mètodes paint i update que no poden ser cridats en codi, si podem cridar el mètode repaint(). Per aquesta propietat és de vital importància donat que permet repintar l'escena quan el programador vol mostrar canvis permeten definir el tractament desitjat de l'escena. De tal manera, quan canviem l' informació d'un element del *Canvas*, cridarem aquest mètode perquè ens cridi al mètode paint i ens la dibuixi amb els canvis actualitzats. El mètode *repaint*() únicament avisa a la màquina virtual de Java que aquell component necessita ser repintat. El mètode en sí mateix no borra ni dibuixa res, és la màquina virtual que posa la petició a la cua d'events de pantalla juntament amb els events de teclat i ratolí. Quan la màquina virtual decideix repintar el component , el que fa és cridar al seu mètode *update(Graphics g)* . L' implementació per defecte d'aquest mètode borra l'escena actual repintant-la amb el color de fons definit (Background) i crida al mètode paint per pintar de nou l'escena.

El mètode *paint(Graphics g)* és el responsable de dibuixar les imatges. Normalment és el sistema operatiu que el crida per dibuixar la pantalla com a resposta a una System-triggered Painting Operation, una operació on el sistema operatiu decideix que el component o una part d' ell ha de ser repintat per algun motiu, els motius més típics són: el component es fa visible per primer cop, el component és redimensionat o que

el component ha estat tapat per una altra finestra i ara el component torna a estar a la vista.

3.1.3 Efecte Flickering:

En un primer moment la classe *ZonaDibuix.java* va ser implementada com una extensió de la classe *Canvas* que és molt utilitzada en diferents manuals i programes per programar aquest tipus d'entorns. Un cop ja teníem implementat el dibuix d'un rectangle en l'escena i el podíem moure, ens vam trobar amb l'efecte anomenat flickering o parpelleig que es dona al esborrar-se i tornar-se a pintar l'escena des de zero amb massa refresc. Aquest efecte és molt molest donat que al moure un element de l'escena tota l'escena pampallugueja produint que treballar amb l'aplicació fos inviable. Una llarga exposició a aquest efecte produiria mal de cap i d'ulls al usuari.

En un primer moment es va resoldre parcialment el problema utilitzant la crida del mètode repaint(int x, int y, int w, int h) enlloc de repaint() on x, y, w i h defineixen la zona que volem repintar. L 'únic que vam aconseguir amb aquesta solució va ser que el pampallugueges només a la zona de la pantalla que repintàvem, encara que si arrossegàvem un elements al llarg de la pantalla de manera molt ràpida, a vegades la zona de redibuix era insuficient i quedaven parts del element per la pantalla que no s'esborraven fins que tornaves a passar un altre element a prop seu.

Aquest problema es va solucionar totalment d'una manera molt més eficient utilitzant la tècnica del Doble Buffer. Aquesta tècnica requereix per una banda definint el mètode update(Graphics g) de manera que controlem que no esborri la pantalla, sinó que directament cridi al mètode paint(Graphics g). Per altra banda s'han d'omplir tots els píxels del component dibuixant les noves dades sobre les antigues. Per aconseguir que el mètode paint(Graphics g) implementi la tècnica del Doble Buffer hem d'utilitzar dos buffers. Un d'ells conté els píxels en pantalla i l'altre és un objecte BufferedImage. El que es fa és dibuixar sobre un dels buffers (el BufferedImage) evitant dibuixar sobre la pantalla. Quan ja hem dibuixat tots els elements sobre el BufferedImage, l'enganxem directament sobre el component. Al no esborrar el component i dibuixar la imatge a sobre directament eliminem l'efecte flickering.

Per si queda algun dubte en el següent codi es pot observar la manera correcta de definir els elements per representar l'escena.

```
Public class NouCanvas extends Canvas
{
        Public void paint (Graphics g) {

                BufferedImage imatge = ( BufferedImage) createImage(width, height);
                (1)
                imatge.draw(....);      (2)
                g.drawImage(imatge,0,0,this);          (3)
        }

        Public void tractamentElements(){
                ....
                //quan ens interessa actualitzar el component
                repaint(); (4)
        }
        Public void update(Graphics g) { paint (g); }  (5)
}
```

1) Creem *una imatge de la mateixa mida que l'escena*
2) Dibuixem sobre l' imatge.
3) Enganxem la imatge sobre el component
4) Es crida al mètode repaint() perquè faci un update de l'escena
5) Es crida directament el mètode paint evitant que s'esborri l'escena.

En els *JComponents*, la tècnica del Doble Buffer la tenim implementada per defecte i només l'hem d'activar de la següent manera*: unJComponent.setDobleBuffered(true);*

Aquest va ser un dels motius pels quals la classe *ZonaDibuix.java* va passar a ser una extensió de *JComponent* enlloc de *Canvas*. Amb aquest canvi no ens hem de preocupar de definir el mètode *update* ni d'utilitzar un objecte BufferedImage on dibuixar l'escena per evitar l'efecte flickering.

L'altre motiu que ens va fer decidir migrar a *JComponent* va ser perquè la classe *Canvas* ve de la llibreria AWT mentre que la classe *JComponent* pertany a la llibreria Swing, per tant la classe Canvas com alguns d'altres components del package java.awt.* són components antics (de Java 1.0) basats en una estructura més simple (Light Weight) que Swing i no funcionen igual. Aquest fet provocava que es produïssin glitches i la zona de disseny es mostres per sobre de la barra de menú tapant la visibilitat dels seus ítems.

Com que la classe *Canvas* i *JComponent* tenen un comportament tant semblant la migració va ser molt senzilla i només es va tenir que canviar la classe de la que es deriva. Per millorar el rendiment s'ha esborrat codi innecessari com la redefinició del mètode *update* i l' implementació manual del Doble Buffer.

3.1.4 Que es pinta?

Cada escena disposa d'un objecte de classe *Format* classe que té com a única finalitat guardar els colors de tots els elements que es poden dibuixar.

Primer de tot es pinta el fons per netejar l'escena i a continuació es pinten la resta d'elements segons la profunditat que els volem donar. Els següents elements que pientem són les transicions que les tenim guardades en una llista (*llistaCaixes*). Per cada transició a part de dibuixar la seva figura, se li dibuixa a sobre l' informació lèxica que ha de tenir cada un dels seus camps. Per saber on ha d'anar el text s'aprofiten les coordenades de la transició i una variable de desplaçament pròpia de cada transició que marca on està el separador en cada cas.

Amb motiu d' optimitzar els recursos en la mesura del possible es va decidir que enlloc de que cada estat pintés el seu rectangle que marca l'origen i el destí de les arestes siguin les transicions qui controlin si tenen algun estat que arriba o surt de la transició amb un comptador. En cas que els comptador siguin més grans que zero, es pintarà un rectangle a l'extrem de la transició corresponent.

Per últim es pinten les arestes. S'ha tingut en compte l'ordre de dibuix dels elements per resoldre un conflicte de comoditat en casos en que es treballi sobre un autòmat que té molts estats i transicions que es creuen. Pot ser molt complicat veure l'origen i destí de les arestes si hi ha transicions que les tapen pel camí. Per aquest motiu s'ha decidit que els estats tenen més prioritat que les transicions i per tant es dibuixen en última instancia per resoldre la superposició d'elements. Abans de sortir es pinta una bora al voltant de l'escena per motius d'estètica.

3.1.5 Control d'events:

Cada vegada que l'usuari tecleja un caràcter o fa clic en un botó del ratolí, es produeix un event. Perquè la nostra classe faci un correcte tractament els events,s'ha d'implementar l' interfície apropiada i ser registrada com un "oient d'event" de la "font d'event" corresponent.

Cada event està representat per un objecte que ofereix informació sobre l'event i identifica la font. La font dels events normalment són components, però altres tipus d'objectes també poden ser fonts d'events. Cada font d'events pot tenir varis oients registrats i a l'inversa, un únic oient pot registrar-se en varies fonts d'events.

Els components Swing poden generar moltes classes d'event per tant primer de tot s'ha d'escollir quines events són els que ens interessa detectar i quin tipus d'oient hem d'implementar:

Per una banda necessitem l'oient *MouseListener* per tractar els events que sorgeixen cada vegada que l'usuari fa clic amb el ratolí sobre un component. L'utilitzarem per seleccionar els element de l'escena.

Per altra banda també necessitem l'oient *MouseMotionListener*, per tractar els events que sorgeixen quan l'usuari mou el cursor. L'utilitzarem per moure un element per pantalla seguint el recorregut del ratolí.

Per utilitzar aquests oients la classe ZonaDibuix.java els ha d' implementi en la definició de la classe. A continuació els elements que volem controlar els hem de registrar al oient. En aquest cas és tota l'àrea de l'escena que volem que sigui escoltada per tant en el constructor de la classe li assignem els oients utilitzant els mètodes : ***addMouseMotionListener (this)*** i ***addMouseListener(this)***.

Per últim hem d'implementar els mètodes de l' interfície de l'oient:
public void mousePressed(MouseEvent e) i **public void mouseReleased(MouseEvent e)** pel *MouseListener* i **public void mouseDragged(MouseEvent e)** pel *MouseMotionListener*.

D'ara en endavant al parlar de la gestió dels events referents a les transicions i estats es farà en tres contexts: ratolí premut, ratolí arrossegat i ratolí alliberat i estarem fent referència al codi implementat en cadascun dels tres mètodes que acabem de mencionar.

3.1.6 Buscar elements per l'escena:

Per optimitzar la cerca de transicions i estats d'una manera centralitzada, assegurar-nos una bona assignació de les variables globals i eliminar codi duplicat s'ha creat un mètode que es crida cada vegada que es vol comprovar si en unes coordenades es troba algun component.

Aquest mètode s'anomena **buscarComponent(int x, int y**) i retorna un booleà que tindrà valor *True* en cas que s'hagi trobat un element en aquella posició.

El cos d'aquest mètode és el següent:
Primer s'entra en un bucle que recorre la llista de transicions de final a inici (en cas de transició superposades la del final de la llista és la superior i té prioritat al ser seleccionada) fins que trobem una caixa que té les coordenades del MouseEvent. Si ens trobem amb una caixa que té les coordenades es faran les següents accions:

- Assignem a valor True a **element_trobat** que és la variable de retorn. Aquesta assignació ens fa saltar una condició de parada del bucle per no continuar recorrent la llista.
- Assignem el valor True a la variable global **caixa_seleccionat**. Aquesta variable un cop fora del mètode se li comprovarà el seu valor per saber quin tipus d'element s'ha seleccionat.
- Assignem el valor de la posició de l'element en la llista a la variable global **index_llista**.

En cas que s'hagi recorregut la llista de transicions sense resultat, es fa el mateix procés sobre llista d'estats (**llistaArestes**) per comprovar si s'ha fet clic sobre el rectangle de menú de l'estat. En cas de trobar un estat es produeixen les mateixes accions excepte que el valor de *caixa_seleccionat* és False.

3.1.7 Moure elements per l'escena:

A continuació parlarem en deteniment del desplaçament dels estats i transicions per la pantalla però aquest apartat està dedicat al mètode compartit pels dos elements que realment serà l'encarregat generar el seu moviment. El mètode del que estem parlant és **updatePosicio(MouseEvent e)** i es cridat pel gestor d'events del ratolí.

La manera de funcionar d'aquest mètode és la següent: si es té una transició seleccionada s'actualitzen les seves coordenades sumant-li el desplaçament determinat per les coordenades del MouseEvent. La variable global "*index_llista*" instanciada en el moment de seleccionar la transició, determina quina és la transició a tractar evitant haver de fer una cerca en cada refresc .

La nova posició de la transició es validada amb el mètode **checkPosicio()** que corregeix les seves coordenades si una transició surti del límit superior o lateral esquerra, eliminant així el risc de que surti de la zona de disseny i no es pugui recuperar.

A continuació en el cas que hi hagi estats que depenguin de la transició que s'està desplaçant, aquests actualitzaran el seu inici i/o fi (segons el tipus de relació que mantinguin amb la transició) , el seu control i menú.

Per altra banda si el que es té seleccionat és un estat, el desplaçament del *MouseEvent* és sumat a les coordenades del rectangle de control de l'aresta que és el responsable de la seva curvatura . Es comprova que sigui una posició vàlida (no surti límits) i s'actualitza el menú de l'estat perquè estigui sobre la línea.

Abans de sortir del mètode es produeix una crida al mètode *repaint()* que s'encarregarà d'ensenyar per pantalla els canvis produïts.

3.1.8 Reset:

S'ha implementat un mètode per fer un reset de l'escena el qual borra tots els estats i transicions de les seves llistes i retorna la mida de l'àrea de l'escena al seu valor inicial.

3.1.9 Solució a la limitació de la mida de l'escena:

Es van valorar les necessitats de l'usuari final i es va observar que l'escena hauria de ser de mida dinàmica per tal de permetre representar tant autòmats petits com autòmats d'una mida considerable amb una gran quantitat d'estats i transicions.

El problema és que la mida de l'escena que és finita, per tant si es vol treballar amb moltes transicions la mida de les caixes ha de ser petita perquè hi càpiguen totes però en conseqüència el text que contenen serà il·legible. Per altra banda si la mida de les caixes és gran l'autòmat no cap per pantalla.

Com que la comoditat de l'usuari final és una prioritat s'han implementat dos solucions: una barra de desplaçament i un zoom.

3.1.9.1 Barra de desplaçament:

La classe *Escena.java* ha estat implementada per crear i inicialitzar les estructures necessàries per treballar després amb un objecte *ZonaDibuix*.

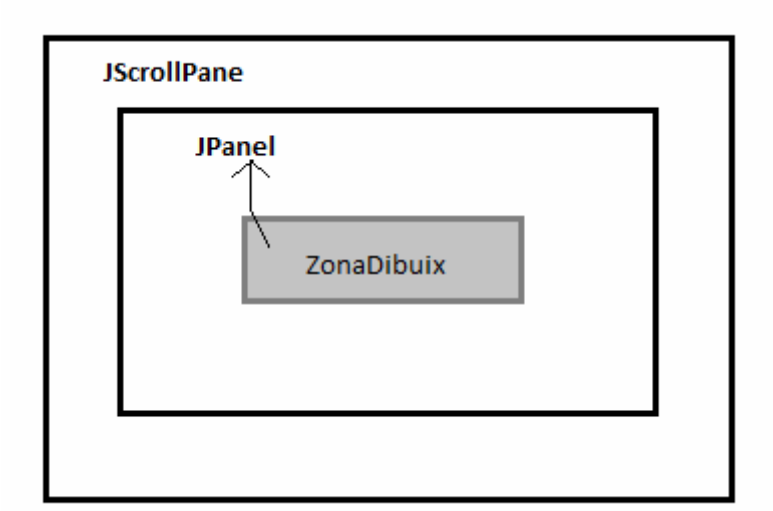
Java ens proporciona la classe *JScrollPane* per afegir una barra de desplaçament a la nostra GUI.

Un objecte *JScrollPane* proporciona una vista desplaçable sobre un component lleuger a través de barres de desplaçament per mostrar un component que no cab a la pantalla degut a la seva mida. Aquest component s'encarrega de crear les barres de desplaçament quan són necessàries i repinta l'escena quan l'usuari fa ús de la barres.

La inicialització de la variable contenidor és la següent:

- Creem un objecte JPanel anomenat panellDibuix i el fem visible.
- Creem un objecte ZonaDibuix passant-li com a paràmetre el JPanel anterior per guardar la referència.
- panellDibuix se li afegeix l'objecte ZonaDibuix.
- Creem l'objecte contenidor JScrollPane i li passem per paràmetre panellDibuix. També fem les inicialitzacions de mida i flags necessaris.

En la següent imatge es mostra el resultat de l' inicialització on s'observen les classes que són contenidores i les que són contingudes. També podem observar que des de ZonaDibuix tenim una referència al objecte JPanel necessària per cridar els mètodes que faran possible l'actualització de la barra de desplaçament.



Un cop feta la inicialització la resta d'operacions es produeixen en la classe ZonaDibuix cada vegada que fem zoom o movem una transició. Quan es mou una transició de lloc es crida el mètode *checkPosicioRectangle()* per comprovar que no es una posició incorrecta(negativa), també comprova si la nova posició està més enllà de l'àrea de

l'escena actual, si es surt per la part dreta i/ o inferior, augmentem l'àrea perquè s'inclogui la transició i es criden els mètodes **panellDibuix.setPreferredSize(area)** i **panellDibuix.revalidate()** en aquest ordre. Aquests mètodes s'encarreguen d'avisar al seu contenidor *JScrollPane* que comprovi la nova àrea i que en cas que sigui necessari faci aparèixer o actualitzar la barra de desplaçament.

3.1.9.2 Zoom:

El zoom s'ha implementat per suplir la necessitat de treballar l'escena amb perspectiva. La barra de desplaçament ens permet tenir en l'escena autòmats de qualsevol mida. La combinació amb el zoom permet treballar des del punt de vista més còmode per l'usuari.

Per implementar el zoom s'utilitza una variable global a la classe *ZonaDibuix* anomenada **factorEscala**.
El paint s'ha modificat de tal manera que es dibuixa el fons i a continuació s'escala l'escena amb 1/factorEscala per representar la resta d'elements.

Així doncs, a partir d'aquest moment cada vegada que es treballi sobre unes coordenades de l'escena aquestes són multiplicades pel *factorEscala*. Per exemple per comprovar si les coordenades on s'ha fet clic al ratolí pertanyen a una transició és faria la següent comparació:
If(Transicio.contains(x*factorEscala, y*factorEscala)) { OPERACIONS}

Per recrear les accions d'apropar i allunyar el zoom s'han afegit dos botons al a barra d'eines. Aquests botons criden els mètodes zoomIn() i zoomOut() que multiplicar *factorEscala* per una constant, modifiquen l'àrea de l'escena i criden els mètodes panellDibuix.setPreferredSize(area) i panellDibuix.revalidate() per avisar al JScrollPane que la mida de l'àrea de dibuix ha variat.

3.2 Transicions

Per representar les transicions s'ha creat la classe *Caixa.java* específicament per guardar tota la informació rellevant de les transicions. Tot el seu tractament es porta a terme en la classe *ZonaDibuix*

3.2.1 Què conté la classe Caixa.java?

- Cada transició té associada un identificador de tipus enter. Aquest se li assigna un valor en el constructor utilitzant el mètode static getNewIdperTransicio() de la classe estàtica GeneradorDeIDs.java. Aquest mètode ens retorna un valor d'ID diferent per cada crida.
- Dos cadenes de caràcters on es guardarà la informació lèxica i la sortida del transductor.
- Un objecte GeneralPath el qual utilitzem per mostrar per pantalla la transició. S'actualitzaran les seves coordenades per fer un desplaçament de la transició per la pantalla.
- Un enter que identifica de quin tipus és la transició (1=Normal, 2= Inicial, 3= final i 4 = connector).
- Tenim dos variables nArestesIniciTransicio i nArestesFiTransicio que són comptadors de la quantitat d'arestes que tenim al inici i al final de la transició.

Els mètodes d'aquest classe són únicament pel tractament d'aquestes dades per instanciar o demanar un valor, i per traslladar l'objecte . En el constructor de la classe s'inicialitzen tots els camps.

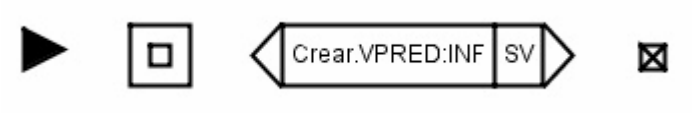
3.2.2 Tipus transicions:

Tenim quatre tipus de transicions:

1. La transició inicial és representada amb un triangle. Determina on comença l'estat inicial de l'autòmat. Només n'hi pot haver una d'aquest tipus.
2. La transició final és representa amb un quadrat dins un quadrat més gran. Determina els estats finals de l'autòmat.
3. La transició normal és representada per figura resultant de dos rectangles que contenen informació dins seu i dos triangle als extrems. En el primer dels rectangles s'hi conté l' informació lèxica de la transició i en el segon rectangle la

sortida del transductor. També ens podem trobar amb el cas que no tenim sortida del transductor i la transició només té un rectangle.

4. La transició connector és representada per un quadrat amb dos segments a les seves diagonals. Aquest tipus de transició no conté informació lèxica. S'utilitza per poder simplificar representacions d'autòmat i representar e-transicions.



3.2.3 Gestió de les transicions:

La gestió de les transicions es desenvolupa integrament en la classe *ZonaDibuix*. En aquest classe tenim una LinkedList parametritzada a tipus Caixa. En aquesta llista tindrem totes les transicions que s'estan representant en l'escena. S'ha utilitzat la classe LinkedList per aprofitar els mètodes de gestió de dades que afegeixen i treuen dades de la llista d'una manera molt simple i eficient. S'ha parametritzat per eliminar les operacions de cast del tipus de dades cada vegada que es treballa amb aquesta llista.

Encara que tinguem quatre tipus diferents de transicions, la classe és la mateixa l'únic que varia és la forma que s'ha de dibuixar en cada cas. Per la resta el codi s'ha implementat de tal manera que el tractament de les transicions es fa a partir de la rectangle que les envolta. Així s'utilitza el mateix codi tant per moure les transicions com per afegir-li estats o qualsevol altre acció.

3.2.4 Construir les Transicions:

Un dels requisits del projecte és representar els estats i transicions de la manera que marca l'estàndard abans mencionat. Recordem que aquesta representació consisteix en una aresta pels estats i per les transicions s'utilitza la representació que s'acaba de mencionar.

Per representar una transició de tipus normal es podria haver utilitzat dos rectangles i triangles contigus, però aquesta representació hauria suposat que per realitzar la

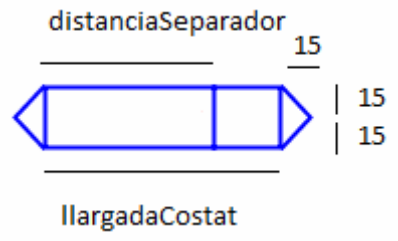
translació s'hagués de fer per cada un dels objectes, igual que al comprovar si s'ha picat sobre la transició s'hauria de comprovar sobre l'àrea que formen les tres figures.

Per reduir operacions s'ha optat per utilitzar un sol objecte *GeneralPath* . Aquesta classe hereta de la classe *Shape* i ens permet representar figures definint la seqüència de traços utilitzant els mètodes **moveTo(x,y)** i **lineTo(x,y)**.

Per representar les transicions primer fem un *moveTo(x,y)* a la coordenada de la zona de disseny on volem que estigui la cantonada superior esquerra de la transició. A partir d'aquí se li dona la forma desitjada com si fos un joc infantil d'unió de punts. S'utilitza el mètode *lineTo(x,y)* per recórrer el camí mínim que passa pels punts característics de la transició.

Per saber els punts característics s'utilitzen dos variables instanciades en el constructor **distanciaSeparador** i **llargadaCostat**. La variable *distanciaSeparador* és l'amplada que ocupa el text de l' informació lèxica que va en el primer rectangle de la transició marcant-nos on acaba el primer rectangle. La variable *llargadaCostat* ve definida per la distanciaSeparador més la llargada de la informació de sortida del transductor. L'amplada que ocupen els 2 textos marca la referència on acaba el segon rectangle.

Amb l' informació d'aquestes dos variables i un valor constant per la posició de la punta ja es suficient per crear la transició.



Per introducció d'informació lèxica dins la figura s'encarrega el mètode *paint* de *ZonaDibuix* amb el mètode **drawString(String,x,y)**. La coordenada y té el mateix valor per les dos cadenes de text determinat pel valor de **centerY()** de l'objecte. Per les coordenades x, utilitzarem el valor de la coordenada X de l'objecte i per dibuixar segona cadena de text dins el segon rectangle se li suma el desplaçament corresponent al valor de la variable distanciaSeparador.

Per les transicions de tipus inicial, final i connector també s'utilitza un objecte de la classe *GeneralPath* però els hi donarem una forma diferent per cada cas. L'única diferència important respecte les transicions de tipus normal és que aquests no contenen informació en el seu interior i s'ha de tenir present en el mètode paint() per no produir un error.

3.2.5 Anàlisis de les accions que es duen a terme amb les transicions:

Per explicar el tractament resultant de la gestió d'events que tenen a veure amb les transicions parlarem en tres contexts introduïts anteriorment: ratolí premut, ratolí arrossegat, ratolí alliberat.

3.2.5.1 Acció afegir transició:

Ratolí premut:

Aquesta acció només té repercussió en aquest context per tant al arrossega el ratolí i al alliberar-lo, no es produirà cap altra acció.

S'afegeix la transició en el punt de l'escena on s'ha fet clic. Per donar servei a aquesta acció es fa una crida al mètode **afegirTransicio(MouseEvent e)**. Aquest mètode era massa complex i es va crear la classe **CreadorTransicions.java** perquè s'encarregués de les operacions de creació i actualització del contingut de les transicions. S'utilitza el mètode **novaTransicio(escena, x,y)** d'aquesta classe que s'encarrega de crear i mostrar per pantalla una finestra emergent que ajuda a l'usuari a especificar els camps de la transició.

Les coordenades x i y determinen la posició on es col·locarà la transició. Els seus valors venen determinats per les coordenades del MouseEvent a través dels mètodes **e.getX()** i **e.getY()**. El paràmetre **escena** és el propi objecte ZonaDibuix per configurar la finestra emergent i es mostri en tot moment davant de l'escena fins que s'acabi d'afegir la transició o es cancel·li l'operació.

El valor retorn esperat del mètode *novaTransicio(escena,x,y)* és un objecte de la classe caixa inicialitzat segons la configuració escollida per l'usuari. En cas de que es produeixi algun problema en la creació de la transició perquè l' informació no és correcta o es cancel·li l'operació el retorn és NULL.

Per tant si tenim no tenim NULL afegim la Caixa a la llista de transicions i repintem l'escena perquè es mostri la nova transició.

L' informació sobre la classe *CreadorTransicions.java* i els seu contingut se'n tracta exhaustivament en un futur apartat de la memòria. La raó d'aquest aplaçament és per organitzar millor el contingut de la memòria i no perdre el fil actual sobre les accions que es duen a terme amb les transicions.

Acció eliminar transició:

Ratolí premut:

Si l'opció de esborrar elements està activa i s'ha fet clic sobre una transició, es fa una crida al mètode **borrarElement(MouseEvent e)** que s'encarregarà d'eliminar-la.

Al eliminar una transició també s'ha d'eliminar tots els estats que tenen el seu origen i/o destí.

A part dels estats que tenen el seu origen i destí sobre una mateixa transició, la resta d'estats relacionen dos transicions. El problema recau que s'ha de buscar cada un dels estats que tenen relació en algun dels seus extrems amb la transició que es vol eliminar i s'ha d'avisar a la transició de l'altre extrem perquè decrementin el seu comptador d'estats.

Acció moure transició:

Ratolí premut:

En aquest cas es fa una crida al **mètode iniciarDesp(MouseEvent e)**.

En el mètode *iniciarDesp(MouseEvent e)* es produeixen una sèrie d'operacions per inicialitzar la informació necessària que s'haurà d'actualitzar durant el desplaçament.

El primer que es fa es comprovar si s'ha seleccionat un element o si s'ha fet clic sobret el buit. Es crida al mètode *buscarComponent(x,y)* i el seu retorn es guarda en la variable booleana *element_seleccionat* .

En cas que s'hagi trobat algun element en aquelles coordenades el valor *element_seleccionat* serà True. Quan ens trobem en aquest cas es passa a comprovar si s'ha fet clic sobre un estat o una transició. Si és una transició el valor de la variable global *caixa_seleccionat*, que ha estat instanciat dins el mètode *buscarComponent(x,y)* té valor True.

Si realment s'ha fet clic sobre una transició la seva posició dins la llistaCaixes és el valor de la variable global *index_llista*.

A continuació fem les següents operacions:

- S'assigna el valor a les variables "x" i "y" a partir de les coordenades de la transició menys les del MouseEvent. En aquestes variables es guarda el valor de referència de les coordenades. És a dir on s'ha produït l'últim desplaçament.
- Es busca quins estats tenen relació amb la transició que s'està traslladant:
    - S'Assigna el valor de la ID de la transició a una variable local id_caixa.

  - Es recorre la llista d'estats comprovant si valor de id_caixa és igual al valor de id_inici i/o id_fi. En cas positiu, segons el tipus de relació de l'estat amb la transició es guarda la seva posició en la llista dins una de les següents llistes: ArestesModificarInici , ArestesModificarFi o ArestesModificarIniciIFi.
- Es crida el mètode *updatePosicio(MouseEvent e)*.

Ratolí arrossegat i ratolí alliberat:

Si ens trobem en el cas que estem movent una transició simplement fem una crida al mètode *updatePosicio(MouseEvent e).*

3.3 Estats

Els estats en la nostra aplicació es representen en forma d'arestes. Per representar els estats s'ha creat la classe *Aresta.java* específicament per guardar tota la informació rellevant dels estats. Tot el seu tractament es porta a terme en la classe ZonaDibuix.

3.3.1 Què conté la classe Aresta.java?

- Un objecte QuadCurve2D.Double que serà la representació de la nostra aresta .
- Quatre objectes Rectangle anomenats inici, fi, control i menú.
- Dos variables enteres per guardar els identificadors de les transicions origen i destí.
- Una variable booleana per determinar si es vol mantenir la curvatura.
- La mida dels rectangles que utilitzem per senyalitzar l' inici, fi i menú de l'estat està definida per dos constants.
- Una cadena de caràcters que s'utilitza com etiqueta. Per defecte la cadena és buida però a través del menú de l'estat se li canvia el valor.

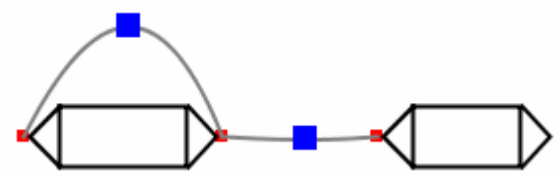
3.3.2 Gestió de les transicions:

La gestió de les transicions es desenvolupa integrament en la classe ZonaDibuix. En aquest classe tenim una LinkedList parametritzada a tipus Aresta on es guarden totes les arestes que s'estan representant en l'escena. Al igual que en el cas de les transicions s'ha utilitzat la classe LinkedList per aprofitar els mètodes de gestió de dades. S'ha parametritzat per eliminar les operacions de cast del tipus de dades cada vegada que es treballa amb aquesta llista.

3.3.3 Construir els Estats:

Tothom que ha treballat alguna vegada amb programes de creació de d'autòmats o diagrames d'estats s'ha trobat que a vegades és útil que les línees que uneixen els elements es puguin corbar. Per aquest motiu s'ha escollit que les arestes que representen els estats siguin objectes *QuadCurve2D.Double* enlloc de *Line2D* per poder aconseguir corbar els estats. També és necessari corbar els estats pels casos en que comencen i acaben en la mateixa transició (loop).

Per pantalla les arestes es visualitzen de la següent manera:

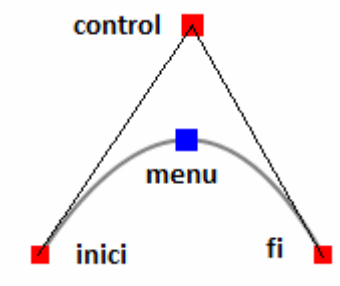


La classe QuadCurve2D permet construir un segment corbat basat en equacions matemàtiques. La corba generada també rep el nom de corba quadràtica de Bézier. Aquesta corba es basa en l'idea de d'establir dos punts que defineixen els extrems del segment, i un tercer punt anomenat punt de control que s'encarrega de determinar el grau de la curvatura.

El constructor de la classe QuadCurve2D.Double ha d'instanciar les coordenades dels punts d'inici, fi i control. Un cop té aquestes tres coordenades la forma de l'aresta pren la següent formula:

$$Aresta(t) = (1-t)^2 * inici + 2 * t * (1-t) * control + t^2 * fi$$

Notem que per t= 0 es compleix que Aresta(t=0) = inici i quan t=1, Aresta(t=1) = fi, que són els extrems.



Al mostrar els estats per pantalla només pintem l'objecte QuadCurve2D.Double i el rectangle de menú situat al punt mig de l'aresta. Aquest rectangle serà el punt d'interacció de l'estat, que ens permet corbar l'estat arrossegant-lo per la pantalla. El menú també ens permet accedir a les seves característiques fent clic amb el botó dret del ratolí.

Per corbar un estat tot i que es fa arrossegant el rectangle menú, el desplaçament que realitzem amb el ratolí es suma sobre les coordenades del punt de control de l'aresta. A continuació es fa una cridar al mètode **checkPosicioControl()** que ens corregeix les coordenades en cas que es surti del límits. Un cop les noves coordenades de control estan validades, s'actualitza l'aresta perquè adopti la nova curvatura i s'actualitza la posició de menú perquè torni a estar al mig de l'aresta.

3.3.4 Restriccions:

- Un estat no pot tenir com a destí la transició inicial.
- Un estat no pot tenir com a origen la transició final.

Les transicions de tipus inicial i final tenen com a únic propòsit marcar els extrems de l'autòmat. Si es vol fer una de les anteriors connexions s'hauria d'iniciar/finalitzar l' inserció de l'estat a una transició connector que estigui connectada amb l'estat en qüestió. Semànticament és equivalent.

3.3.5 Anàlisis de les accions que es duen a terme amb els estats:

Per explicar el tractament resultant de la gestió d'events que tenen a veure amb els estats parlarem en tres contexts introduïts anteriorment: ratolí premut, ratolí arrossegat, ratolí alliberat.

Acció afegir estat:

Ratolí premut:

Primer de tot es fa una cerca sobre l'escena amb el mètode *buscarComponent(x, y)* per comprovar si s'ha fet clic sobre una transició. En cas afirmatiu i en cas de no ser la

transició final, cridem el mètode **novaArestaInici()** de la transició per augmentar el seu comptador d'estats que neixen en aquella transició. Guardem la referencia d'aquesta transició en una variable global auxiliar per utilitzar el seu valor en cas que l'aresta no sigui vàlida perquè és repetida (té el mateix origen i destí) o no es selecciona una transició de destí, decrementar el comptador d'estats que neixen de la transició. S'afegeix una nou objecte Aresta a la llista Arestes. Aquest nou objecte guarda la ID de la transició origen i inicialitza els camps d'inici, control i fi de l'objecte QuadCurve2D. L' inici de la transició ve definit per la coordenada més gran de la transició. El fi fins que no acabem d'afegir l'estat les seves coordenades són les que apunta el ratolí.

Ratolí arrossegat:

Es fa una crida al mètode *updateEstat(MouseEvent e)* el qual actualitza la coordenada de fi de l'aresta. Per poder observar en temps real l'aresta que estem construint.

Ratolí alliberat:

Per posar punt i final a l' inserció d'un nou estat es crida el mètode **finalitzarInsertarEstat(MouseEvent e)** quan s'allibera el botó del ratolí.

Aquest mètode comprova si en el moment que s'ha deixat de fer clic el ratolí és troba sobre una transició que no sigui la inicial. En cas negatiu s'elimina l'estat de la llista i es decrementa el comptador d'estats que neixen de la transició origen per desfer l'operació d'inserció.

En altre cas es guarda la ID de la transició destí . S'ha de comprovar si les transicions d'origen i destí són les mateixes ja que en aquell cas es corba l'aresta per sobre la transició. També es comprova que sigui un estat no repetit (en cas de que en la llista d'estats ja tinguem un altre estat que té el mateix origen i destí es procedeix a eliminar l'aresta i decrementar els comptadors d'estats de les transicions afectades). Per finalitzar s'actualitzen les coordenades de fi de l'aresta així com el seu control i menú.

Acció eliminar estat:

Ratolí premut:

Si l'opció de esborrar elements està activa i s'ha fet clic sobre un estat, es procedeix a eliminar-lo. A part d'eliminar l'estat en qüestió s'ha de recorre la llista de transicions fins trobar-nos amb la/les transicions que tenen la mateixa ID que nosaltres tenim guardada en els caps id_inici i id_fi . Quan trobem aquestes transicions les avisem que han d'actualitzar el seu comptador d'arestes que arriben/surten de la transició.

Acció moure estat:

Quan tenim l'opció moure element seleccionada de la barra d'eines, si seleccionem el rectangle definit pel menú de l'estat i l'arrosseguem, el que realment es fa es moure la coordenada de control de l'objecte QuadCurve2D és a dir de l'aresta. Amb aquesta acció el que s'aconsegueix és modificar la curvatura de l'estat.

Per moure el control s'encarrega el mètode *updatePosicio(MouseEvent e)*. Les operacions a realitzar són les mateixes que s'aplicaven per moure una transició al igual que també es fa una comprovació de posició per assegurar-nos que és sigui vàlida. També s'actualitza quan és necessari la barra de desplaçament. L'únic diferència respecte al tractament de les transicions és que s'ha d'anar actualitzant la posició del menú de l'aresta per observar el resultat.

El desplaçament en si de l'aresta es produeix indirectament al moure una transició amb qui té relació. Pels estats que se'ls ha de mantenir la curvatura no se'ls actualitza les coordenades de control l'aresta evitant que es converteixi en una recta.

3.4 Classe Main

La classe *Main.java* és la classe principal de l'aplicació. En ella es produeixen operacions molt importants com és el llançament del programa o la gestió dels diferents events d'acció llençats pels menús. En aquesta classe també implementem la característica de preguntar a l'usuari si es vol guardar el treball abans de tancar l'aplicació així com múltiples pestanyes per diferents escenes per poder treballar amb la millor comoditat. Tot això sense mencionar que aquesta és la classe arrel del programa la qual conté tots els components que es visualitzen per pantalla.

3.4.1 Mètode estàtic main:

Tota aplicació necessita que existeixi aquest mètode per poder executar un programa. Aquest mètode és realment simplement, tot el que fa és crear un objecte tipus *Main.java* el qual dins el seu constructor crearà un objecte tipus *JFrame* que és la finestra que ens apareix a la pantalla i sobre la qual treballem.

3.4.2 Constructor de la classe Main:

Com s'acaba de mencionar, és aquí on es crea l'objecte *JFrame* que serà la cara del nostre programa. En el constructor instanciem tots els seus paràmetres: li definim el nom de l'aplicació, li definim un Layout tipus BorderLayout, definim la seva operació de tancament, li afegim la barra de menú amb el mètode setJMenuBar, la barra d'eines, la zona de disseny i el label de notificacions. A continuació es crida el mètode **pack()** que fa que la finestra prengui la mida més petita possible que permet veure tots els components. Per acabar fem que la finestra sigui visible amb el mètode **setVisible(true).**

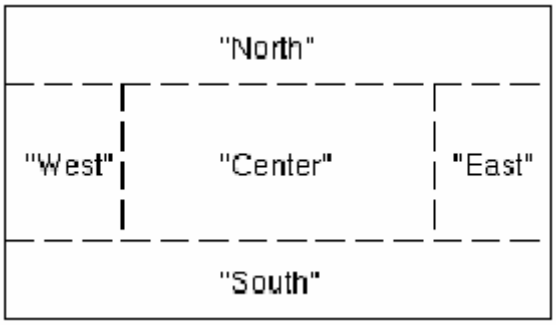
3.4.3 BorderLayout:

La portabilitat de Java a diferents plataformes i diferents sistemes operatius necessita flexibilitat a l'hora de situar els Components (Buttons, Labels, Canvas, etc.) en un Contenidor (JFrame, JPanel, etc.)..

La classe JFrame és una classe tipus Container que s'encarrega de contenir els diferents elements del programa. S'anomena components als elements que estan continguts dins un container.

Un Layout Manager és un objecte que controla com els Components es situen en un Contenidor.

Existeixen varis Layout Managers però per la nostra aplicació hem escollit el BorderLayout. La peculiaritat d'aquest LayoutManager és que divideix el container en cinc zones: North,South, East,West i Center.



S'ha escollit aquest *LayoutManager* entre les diferents opcions perquè és la que ens posicionava millor els elements per oferir una aparença agradable. S'ha situat una barra d'eines a un lateral (WEST), un Label de notificacions a la part inferior (SOUTH) i el centre està reservat pel component que ens mostrarà la zona de disseny (CENTER).

Un altre avantatge d'utilitzar *BorderLayout* és que la barra d'eines que és un component *JToolBar* el qual se'l pot arrossegar i col·locar-lo a una altra posició lliure. L'avantatge de col·locar la barra de menú amb el mètode de la classe JFrame **setJMenuBar(menú)** enlloc d'utilitzar el LayoutManager i assignar-li la posició de North és que d'aquesta manera aquesta posició queda lliure. Per tant la barra d'eines es pot situar als dos costats i a la part superior, justament els llocs més comuns on s'acostumen a situar aquest tipus d'elements en les aplicacions, oferint la possibilitat de personalitzar l'aplicació a gust de l'usuari que l'utilitzi situant la barra d'eines en la posició que li és més còmode.

Per afegir un component a la nostra finestra primer s'ha hagut d'especificar el tipus de Layout Manager. La resta és una tasca ben senzilla, tan sols s'ha d'especificar localització en el segon paràmetre del mètode add(Component, arg).

3.4.4 Menús i la seva gestió:

La barra de menús que ens permet accedir a una gran varietat d'opcions d'una manera organitzada. Aquest component és fonamental en qualsevol aplicació perquè els usuaris estan acostumats a la seva presencia i estan familiaritzats amb la seva organització. En la barra de menús trobem les opcions més comuns de programes de disseny.

Tenim una altra barra de menú o barra d'eines a un lateral que s'encarrega de proporcionar les utilitats bàsiques per dissenyar un autòmat amb totes les comoditats que han estat possibles d' implementar.

Els menús estan compostos per objectes de les classes JButton o JMenuItem aquests elements al ser pressionats llencen un event ActionEvent que ha de ser capturat per una classe que implementa l' interfície ActionListener. La pròpia classe Main és l'encarregada d'implementar aquesta interfície i atendre les peticions dels menús.

En un futur apartat de la memòria es tracta més exhaustivament el tema de la creació d'aquests menús i la seva gestió.

3.4.5 Pestanyes:

En un moment determinat del desenvolupament de l'aplicació es va tenir que decidir que fer quan s'estava editant un autòmat i es volia obrir un document per crear un nou autòmat.

La primera idea que es va considerar va ser que cada vegada que s'obrís una finestra nova crear un nou objecte de tipus *Main*. Encara que aquesta idea era ben vàlida alhora d'implementar es va decidir per una altre camí.

L'opció que s'ha escollit ha estat una que inclou l' utilització d'una barra de pestanyes, on cada pestanya té la seva escena. S'ha escollit aquesta opció perquè per una banda evitem crear masses components repetits com el menú , la barra d'eines o tota la classe *Main*. El que realment l' interessa a l'usuari quan vol obrir un nou document és tenir una nova superfície sobre la qual dissenyar, en el nostre cas, un nou component *ZonaDibuix*, i això és el que s'ha proposat. L'altre raó és per motius d'organització, l'usuari té tota l' informació controlada enlloc del caos que suposa tenir varies aplicacions obertes.

La barra de pestanyes és un objecte de la classe *JTabbedPane* que com havíem mencionat inicialment és el component que està situat al centre del BorderLayout. Això significa que és el Component responsable d' ensenyar-nos per pantalla sobre quina escena estem treballant.

Un objecte de la classe JTabbedPane és un contenidor com el JFrame o el JScrollPane amb la peculiaritat que ens organitza el seu contingut per pestanyes. A més a més, és el propi objecte qui autònomament tracta els canvis de pestanya realitzats amb el ratolí, mostrant en cada cas el contingut de la pestanya seleccionada.

L' objecte JTabbedPane que utilitzem és una variable global a la classe, afegida al JFrame en el constructor. S'ha de decidir la política del Layout de les pestanyes que no caben a la pantalla. Per determinar-ho s'utilitza el mètode setTabLayoutPolicy(int). La resposta que hem considerat més agradable per estètica ha estat SCROLL_TAB_LAYOUT que com el seu nom indica apareixeran dos botons de desplaçament per navegar entre les pestanyes.

Per afegir una nova pestanya al JTabbedPane fem el següent procés:

- Definim un String que té l'estructura "Archivo "+un enter que fa d'identificador i que ens ve proporcionat per la nostra classe GeneradorDeIDs.
- Creem un nou objecte de tipus Escena.
- La classe Main disposa d'una variable global que és una LinkedList parametritzada per objectes de classe ZonaDibuix. En aquesta llista afegim l'objecte ZonaDibuix que conté l'Escena que acabem de crear.
- A continuació afegim una nova pestanya amb el mètode **addTab(nom,icona,JComponent)**. El primer paràmetre és el nom que identificarà aquella pestanya. Si volem es podria posar una icona posant-la en el segon paràmetre, i en el tercer paràmetre que ens demanen un JComponent, li passem el JScrollPane que forma part de la classe Escena. Quan un usuari faci clic en aquesta pestanya el JTabbedPane farà visible aquest component, per tant amb el mètode addTab crea el vincle entre les diferents pestanyes i els JScrollPanes que contenen les zones de disseny.
- A continuació procedim a editar la pestanya en si. L'objectiu es afegir un botó en forma de creu a cada pestanya que al ser pressionat ens elimini la pestanya. Per editar la pestanya utilitzem el mètode **setTabComponentAt(int,Component)**. Pel primer paràmetre li passem una variable que utilitzem com comptador de les escenes (o pestanyes) que tenim disponibles. El segon component és un objecte del a classe PestanyaTab.java que és una classe derivada de la classe JPanel. A continuació s'explica com es construeix i el que fa es afegir la creu per tancar la pestanya.
- Utilitzem el mètode **setSelectedComponent(Component)** on li passem per paràmetre el component de l'última pestanya per aconseguir que la pestanya seleccionada sigui la nova que acabem de crear.
- Utilitzem una variable entera anomenada selectedTab per saber sobre quina escena de totes les que estan disponibles s'han d'habilitar les opcions que es seleccionin en els menús.

Per tancar una pestanya s'ha de seleccionar el botó en forma de creu que li hem afegit. Aquest botó el fem de color transparent dient que la seva àrea no ha de ser omplerta, li creem una bora encara que no la fem visible, i l'assignem a un oient d'events de

ratolí i a un oient d'events d'acció. Definim el mètode **paintComponent(Graphics g)** el qual es crida cada vegada que s'ha de pintar el botó i que determinem que el que s'ha de pintar és una creu amb dos línees que entravessen el botó diagonalment.

Abans de continuar és important saber que el constructor de la classe PestanyaTab té dos paràmetres, ja hem dit que un és una String pel nom. L'altre és l'objecte de la classe Main perquè necessitem guardar la seva referència per després poder-lo avisar quan es tanqui la pestanya. Tot el JPanel es assignat a un oient d'events de ratolí que hem implementat nosaltres anomenat tabListener i que serà comú per totes les pestanyes.

tabListener és una classe derivada de la classe MouseAdapter. Java ens proporciona ajudes per definir els mètodes de les interfícies Listener. Una de les seves ajudes són els classes Adapter, que existeixen per cada una de les interfícies Listener que tenen més d'un mètode. El seu nom correspon al nom de l' interfície però substituint "Listener" per "Adapter". Les classes Adapter deriven d' Object i són classes predefinides que contenen definicions buides per tots els mètodes de l' interfície. Per crear un objecte que respongui a un determinat event, enlloc de crear una classe que implementi l 'interfície Listener, és suficient creant una classe que derivi de la classe Adapter corresponent i definir només els mètodes que ens interessen. En el nostre cas tabListener li hem definit que faci les següents accions pels següents events de ratolí:

- Quan passem per sobre del botó en formà de creu (event mouseEntered) fem que el botó habiliti l'opció de pintar la seva bora. D'aquesta manera subtil farem saber a l'usuari que està sobre el botó que tancarà la pestanya.
- Quan deixem d'estar sobre del botó (event mouseExited), deshabilitem el pintat de la bora del botó.
- Quan ens trobem que s'ha fet clic a sobre (event mousePressed), comprovem que l'origen no és el botó per tant vol dir que s'ha fet clic sobre un altre part de la pestanya. Utilitzem la referència al Main que havíem guardat al constructor per cridar el mètode **actualitzarTab(nom)**. Aquest mètode el que fa és buscar en quina posició de la llista hi ha un component que li coincideix el nom (el qual degut a l' intervenció en el seu moment del GeneradorDeIDs és únic). Amb el mètode **setSelectedComponent(int)** avisem al JTabbedPane que ara volem veure el component que es troba en la posició que acabem de trobar.

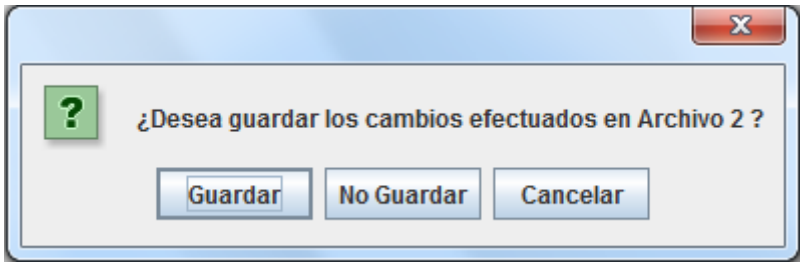
Per acabar amb el tema de les pestanyes només que quan es captura un ActionEvent significa que el que es vol fer es tancar la pestanya per tant es crida al mètode **eliminarTab(nom)** de la classe Main. En aquest mètode esborrem tant la pestanya del JTabbedPane i l'objecte ZonaDibuix de la llista d'escenes. En el cas que només queda una pestanya i es prem el botó per tancar-la el que fem és resetejar l'objecte ZonaDibuix que ho borra tot enlloc fer l'eliminació per no crear un conflicte d'aplicació.

3.4.6 Tancament de l'aplicació:

L'opció per defecte d'un JFrame és tancar l'aplicació quan es tanca la finestra, però si ens posem en la pell d'un usuari que utilitza la nostra aplicació, li seria de gran utilitat que li proporcionéssim el recurs una finestra preguntant-li si vol guardar els canvis realitzats abans de tancar l'aplicació. És més seria lamentable pensar en la possibilitat de la quantitat de feina perduda per fer clic involuntàriament sobre el botó de tancament d'una pestanya.

S'ha afegit com a mesura de seguretat una finestra emergent que avisa que s'està tancat una pestanya i pregunta si vol guardar els canvis, no vol guardar o si vol cancel·lar l'operació i aquella pestanya no es tancarà. Aquesta mesura de seguretat està implementada en el mètode *eliminarTab()* per fer la consulta abans d'eliminar alguna cosa important.

Però que passa si enlloc de tancar una pestanya es tanca directament tota la finestra per la creu superior? També s'ha controlat aquest cas. Primer de tot s'ha definir l'acció per defecte de tancament de la finestra perquè sigui no fer res (*DO_NOTHING_ON_CLOSE*) i que sigui un *WindowAdapter* qui gestioni les peticions de tancament de la finestra. Per aquest motiu la classe Main és una classe derivada de WindowAdapter. En la classe Main capturarem els events de tancament de finestra i davant d'ells es crida al mètode *eliminarTab()* per cada pestanya. Si s'han eliminat totes les pestanyes es tanca l'aplicació.



La gràcia d'aquesta mesura de seguretat és la finestra flotant per tant procedim a explicar com s'ha implementat. L'objectiu és demanar a l'usuari la seva confirmació per realitzar l' acció.

Java ens proporciona la classe JOptionPane que es de gran utilitat per construir finestres destinades a donar cobertura per consultes habituals. JOptionPane té dos modalitats de finestra, les d'avís i les de confirmació que és la que utilitzem. La modalitat de confirmació ens proporciona una finestra que la podem omplir amb un objecte com un JPanel. El que mostrarà per escena és el panell que li passem més els uns botons per acceptar o cancel·lar l'operació a la seva part inferior.

Com que en aquest cas no surt a compte crear una panell perquè contingui únicament un missatge d'advertència, la finestra emergent la crearem d'una manera més compacta utilitzant el mètode **OptionPane.showOptionDialog()**. Aquest mètode ens ajuda a crear la finestra a través d'uns paràmetres configurable evitant haver de crear un panell .

Els paràmetres que li hem de passar són:

- parentComponent: A partir d'aquest component es determina quina és la finestra que fa de pare del JOptionPane. D'aquesta manera la finestra de validació es mostra davant la nostra finestra principal i no permetrà continuar treballant fins que s'escull una opció.
- Message: Passem una String amb el missatge a mostrar. Nosaltres preguntem "¿Desea guardar los cambios efectuados en "+nom+" ?" on nom és el nom de la pestanya per evitar confusions.
- Title: Se li passa una cadena de caràcters que determina el títol de la finestra
- optionType: Demana un enter que indica quina opció volem que tingui la finestra. Els valors possibles són les constants definides en JOptionPane: DEFAULT_OPTION, YES_NO_OPTION, OK_CANCEL_OPTION i la que hem escollit nosaltres YES_NO_CANCEL_OPTION.
- messageType: Demana un enter que indica quin tipus de missatge estem ensenyant
- Icon: una icona per mostrar. Si posem NULL es mostra l' icona determinada pel messageType.
- Options: Se li passa una array d'objectes que determina les possibles opcions. Si se li passa null es posen els botons per defecte. Nosaltres li passem {"Guardar", "No Guardar", "Cancelar"}
- initValue: Determina la selecció per defecte. Ha de ser un dels objectes que li hem passat en el paràmetre options o null.

La crida a *JOptionPane.showOptionDialog()* retorna un enter que representa la opció que ha seleccionat l'usuari. La primera de les opcions de l'array és 0. Si es tanca la finestra amb la creu de el mètode retorna -1.

El que fem és fer un Switch d'aquest retorn. Si el valor és -1 o 3 (Cancelar) no fem res. Si el valor es 0 (Guardar) guardem l'autòmat i anem al següent case que és el del valor 1 (No guardar) que tanca la pestanya i selecciona la pestanya anterior a la que ha estat tancada perquè sigui la nova pestanya seleccionada.

3.4.7 Diagrama de classes

Diagrama de classes de l'aplicació general:

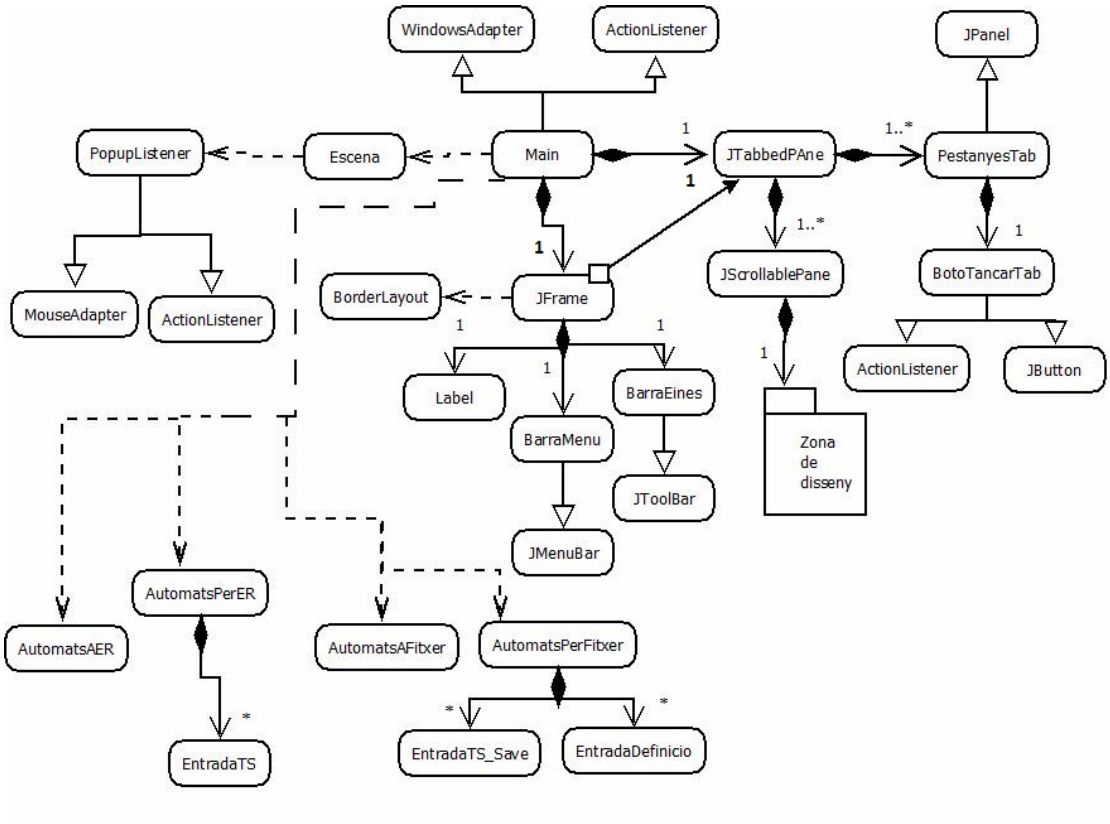
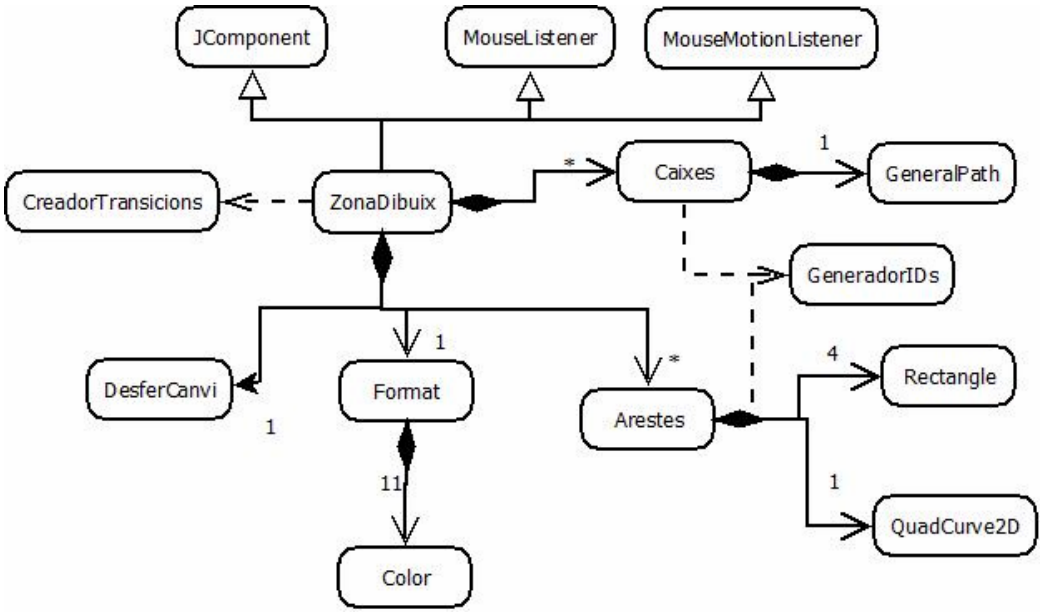


Diagrama de classes de la Zona de disseny:



3.5 Barra d'eines

Per implementar la barra d'eines s'ha utilitzat la classe JToolBar. Un objecte d'aquesta classe es pot veure com un contenidor que espera que l'omplim amb diferents ítems, majoritàriament objectes de la classe JButton . Els avantatges d'aquesta classe davant d'altres mètodes per implementar barres d'eines és que la col·locació dels diferents ítems és gestionada automàtica pel gestor de Layout propi del propi objecte. Al afegir diferents ítems a una barra d'eines aquests es col·loquen un al costat de l'altre d'esquerra a dreta s si la barra d'eines és horitzontal o un a sota de l'altre si la barra en el cas que sigui vertical.

Un objecte d'aquesta classe es afegit dins un contenidor com el que tenim implementat que utilitza el layout BorderLayout i té un component al centre(ZonaDibuix) ens proporciona un grau més de flexibilitat i dinamisme donat que ens permet moure la barra d'eines per la pantalla. En un primer moment la col·loquem a l'esquerra de la pantalla però si l'usuari que està fent servir el programa decideix que en un moment donat li aniria millor que estigues col·locada a la part superior de la pantalla, o per simple comoditat perquè està acostumat a treballar amb programes amb aquesta organització, tant sols ha d'arrossegar la barra d'eines cap aquella posició. També es pot col·locar la barra d'eines a la part dreta o inferior de la pantalla o inclús la utilitzar com a menú flotant si l'arrosseguem fora de la finestra .

Com abans s'ha introduït els ítems que s'afegeixen dins l'objecte JToolBar són tants components JButton com eines volem tenir en el menú.

A l'usuari final li agrada treballar amb programes que tenen una interfície atractiva i per tant s'ha dissenyat icones pels botons per aconseguir un millor acabat i impressió que si s'hagués implementat amb botons omplerts per text. Les icones són de mida reduïda perquè l'aplicació no ocupi més espai del necessari. S'han creat amb l'editor d'imatges paint basant-nos en un disseny minimalista. La icona assignada a cada ítem mostra una imatge representativa del que l'acció que porta a terme. Però per si algun usuari té dubtes, passant el ratolí per sobre seu apareix un missatge informatiu sobre l'acció que representa i quan es selecciona una eina s'actualitza el Label principal de notificacions informant-nos breument de com s'utilitza.

També és important mencionar que l'aplicació s'ha implementat de tal manera que només tenim una única barra d'eines per totes les escenes (en cas que tinguem varies pestanyes). A més a més quan ens movem entre escenes es manté l'opció que es tenia seleccionada.

3.5.1 Control de les accions:

Quan es prem un botó es produeix un event d'acció o ActionEvent, perquè el nostre programa capturi aquests events quan es produeixen s'ha d'associar el botó a un oient d'aquets tipus d'events. Aquest oient en Java s'anomena ActionListener i els events són tractats en el mètode **actionPerformed (ActionEvent e)**.

Quan s'està construint la barra d'eines abans d'afegir els botons al contenidor JToolBar, sel's ha d'assignar l' ActionListener.

Per tractar els ActionEvent's que és produeixen quan es prem un botó hi ha dos alternatives. Una alternativa és assignar un ActionListener diferent per cada o assignar tots els botons a una mateix ActionListener (opció implementada). D'aquesta manera cada vegada que es selecciona una opció del menú l'event és capturat pel mateix ActionListener facilitant-nos la lectura i organització del codi.

Per disposar d'un oient ActionListener hem de crear una classe que implementi ActionListener, i definir el mètode *actionPerformed (ActionEvent e)* . En el nostre programa aquesta classe és la classe Main.java que tractarà els events d'acció produïts tant per la barra d'eines com per la barra de menú.

Quan ens trobem editant l'escena s'ha de tenir un control sobre quina acció s'està portant a terme quan ens trobem que s'ha fet clic amb el ratolí, ja que tots els botons són capturats pel mateix mètode de la mateixa classe.

Quan un event d'acció és capturat primer comprovem que l'origen de l'event és un botó (JButton) per diferenciar de la barra de menú que són ítems de menú (JMenuItem).

Amb el mètode **getSource()** del objecte ActionEvent s'aconsegueix el botó que ha estat pressionat. En un primer moment es guardava com a variables globals a la classe cada botó i simplement es feia una comparació. Però no vam tardar en veure que el nombre de botons creixia considerablement i es va buscar una manera de no tenir que guardar tantes dades.

Una altra manera era guardar totes les Strings que formaven el nom de les eines i cridar el mètode **getLabel()** per fer una comparació entre Strings enlloc de entre JButtons. Tot i així cada vegada en el pitjor dels casos s'havia de fer tantes comparacions com botons. A part per raons d'estètica es va suprimir el camp del Label perquè per pantalla només es veiés la icona.

La solució que es va trobar per no haver de guardar tants botons ni fer tantes comparacions va resultar ser ben esnzilla. En la classe Main.java definim les següents constants:

```
static final char RES = 0;
static final char AFEGIR_ESTAT = 1;
static final char AFEGIR_TRANSICIO = 2;
static final char MOURE_TRANSICIO = 3;
static final char ZOOMIN = 4;
static final char ZOOMOUT = 5;
static final char ELIMINAR = 6;
static final char DESFERCANVI = 7;
```

Quan es crea cada botó s'instanciarà el seu camp nom amb el mètode **setName(String)** passant-li com String un dels anteriors caràcters com si fos un identificador. Quan un botó sigui premut el seu ActionEvent és capturat pel ActionListener. Amb el mètode **getSource()** tenim l'instancia del botó i per tant podem accedir al seu camp nom amb **getName()**. Utilitzem el primer caràcter del nom per entrar-lo com a paràmetre d'un Switch on cada Case serà una de les anteriors constats i així saber quina acció l'usuari vol portar a terme sense tenir que recórrer a múltiples comparacions.

Un factor rellevant és que el nom de les constants es bastant clarificador cosa que ens facilita la feina de lectura del programa.

Segons en quina Case del Switch s'entra es crida un mètode o un altre de la classe ZonaDibuix de l'escena actual de la pantalla. També s'actualitza l' informació del Label de notificacions per interactuar amb l'usuari i que ell pugui validar que ha fet clic sobre l'opció que ell desitjava.

Per les opcions RES, AFEGIR_ESTAT, AFEGIR_TRANSICIO, MOURE_TRANSICIO I ELIMINAR, els mètodes que les tracten simplement assignen el valor de la constant a la variable global opció de la classe ZonaDibuix que també utilitza un Switch cada vegada que es produeix un event de ratolí per decidir quina acció fer segons l'opció que està habilitada.
La resta d'opcions són més elaborades i explicades individualment en un altre apartat de la memòria.

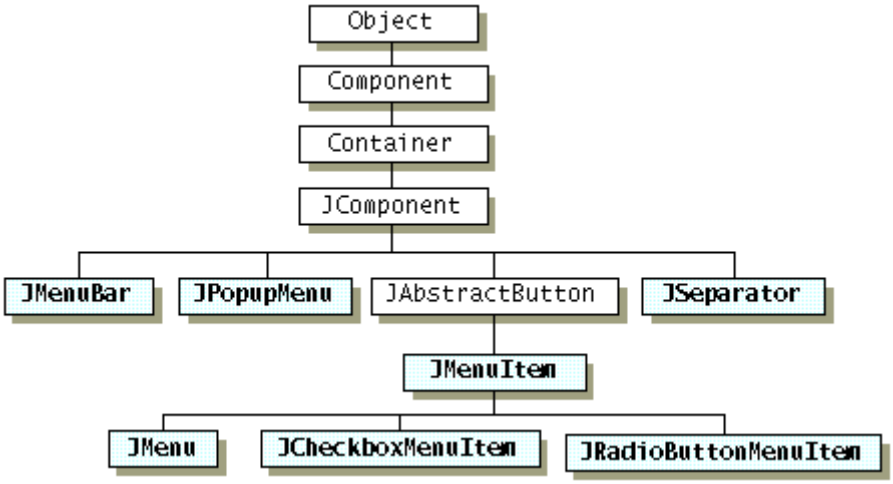
3.6 Barra menú

La barra de menús és la manera més comú que utilitzen la majoria d'aplicacions per distribuir les seves característiques en menús desplegables.

La majoria de funcionalitats es criden a través de la barra de menú perquè sigui fàcil de treballar amb el nostre programa i l'usuari tingui l' informació organitzada de la manera que està més acostumat a treballar.

La creació de la barra de menú és molt semblant a la creació de la barra d'eines. Per la seva implementació es necessària l' utilització d'un objecte de classe JMenuBar.

Aquesta és l'herència de la jerarquia dels menús relacionats amb les classes:



Podem observar que els elements del menú (incloent els menús) són simplement botons. La diferència que tenen respecte un JButton és que quan s'activa un menú , automàticament apareix un popup menú que ensenya els elements del menú.

Tornant a l'objecte JMenuBar, és interessant saber que aquest objecte és un contenidor igual que quan utilitzàvem el JToolBar per la barra d'eines. En aquest cas enlloc d'afegir-li botons, l'omplirem amb objectes JMenu ,per crear els diferents menús.

Cada un d'aquests objectes JMenu se l'omple amb objectes JMenu per construir un sub-menú dins el menú. Els objectes JMenuItem representen cada opció que es mostrar al menú. Quan el programa està en execució i es sel·lecciona és produeix un ActioEvent.

En aquesta captura s'il·lustra el que s'acaba de dir, Inicio, Edición, Ayuda i Exportar són objectes JMenu. Els tres primers són afegits al JMenuBar com a menus principals mentre que el menú Exportar és un sub-menú. La resta d'elements són JMenuItem i estan continguts dins un objecte JMenu.

Cada ítem de menú se'l construeix passant-li una String pel nom. Aquesta String que li passem al constructor es guarda en el seu camp Label, i és el text que es veu en l'aplicació al obrir el menú per aquell ítem.

Opcionalment a alguns ítems se'ls hi ha afegit una icona com la de guardar o obrir un fitxer. Altres ítems se'ls ha definit un accelerador, és a dir definir una combinació de tecles equivalent a premer l' ítem del menú.

Per definir l'accelerador de tecles s'utilitza el mètode de l'ítem **setAccelerator(KeyStroke key)**. El KeyStroke podia ser una única tecla, però s'ha considerat que quan una persona està treballant si tenim moltes lletres assignades el risc de prémer una tecla involuntàriament amb conseqüències imprevisibles és massa elevat. La millor opció és definir dos tecles, una d'elles actua com una màscara especificada com un ActionEvent constant. En la nostra eina per utilitzar els acceleradors haurem de prémer la tecla corresponent alhora que la tecla Ctrl que és la màscara.

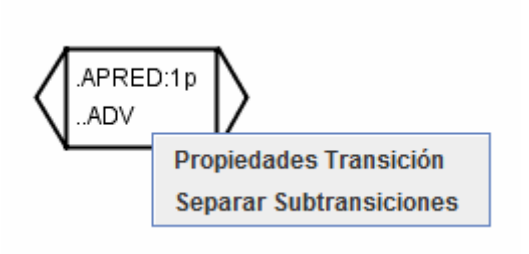
Les combinacions de tecles definides són les següents:

- Ctrl + N per crear una nou document.
- Ctrl + O per obrir un fitxer en format autòmat.
- Ctrl + G per guardar un fitxer en format autòmat.
- Ctrl + F per canviar el format.
- Ctrl + Z per desfer últim canvi.

Per la resta els ítems del menú se'ls tracta igual que els botons de la barra d'eines, també se'ls ha d'assignar a l'ActionListener de l'objecte Main. També utilitzen el mateix algorisme per evitar guardar tantes dades i fer una gran quantitat de compracions cada vegada que el ActionListener captura un event. Se'ls defineix una constant com a nom i quan l'ActionListener captura un ActionEvent, comprova si és un JMenuItem i procedeix a fer ús d'un Switch utilitzant un altre cop com a paràmetre el primer caràcter del seu nom per saber quina resposta a l'event ha de realitzar.

3.7 Menú emergent

Per accedir a les propietats individuals de cada estat i transició s'ha decidit que la manera més còmode és a través d'un menú emergent. El botó esquerra del ratolí s'utilitza per editar l'escena agregant, movent i esborrant elements. El botó dret del ratolí s'utilitzarà per obrir un menú emergent si es fa clic sobre un estat o transició.



3.7.1 Crear el menú emergent:

Els menús emergents són gestionats per un objecte de la classe *PopupListener.java* que és una extensió de la classe MouseAdapter i implementa la interfícies ActionListener.

Cada escena ha de tenir el seu propi gestor de menús emergents per tant en la classe Escena.java a cada objecte de tipus ZonaDibuix que creem se li assigna com a oient d'events del mouse una nou PopupListener.

La classe *PopupListener.java* és derivada de la classe MouseAdapter perquè com que només ens interessa tenir control sobre l'event de fer clic sobre un element de la zona de dibuix. És a dir ens interessa el mètode *mousePressed(MouseEvent e)* de l'interfície MouseListener i definint la classe com una derivada de la classe MouseAdapter tenim l'avantatge que no estem obligats a implementar tots els mètodes de l' interfície MouseListener. Al seu constructor li passa com a paràmetre l'objecte de la classe ZonaDibuix per guardar la referència sobre la qual treballarà

Aleshores cada vegada que es detecta un MouseEvent es comprova si el botó amb el que s'ha produït la selecció és el dret. També utilitzant la referència que tenim sobre l'objecte ZonaDibuix per comprovar si en les seves coordenades de l'event tenim algun element.

En cas que es compleixin les dos condicions es crida al mètode **mostrarPopupCorresponent(MouseEvent)** que segons si l'element seleccionat és un estat o una transició mostrarà un menú o un altre. El menú resultant fa ús del mètode **show(Component,x,y)** al qual li passem com a paràmetre les coordenades on volem que apareix-hi.

S'han implementat dos mètodes mètodes **MenuTransicio(boolean composta)** i **MenuEstat()** encarregats de crear els menús que es mostren quan es fa clic sobre un estat o una transició. Més endavant hem afegit una opció complementaria per crear transicions compostes, el paràmetre del primer mètode determina si n'és el cas per afegir una opció més al menú.

Per crear els menús emergents s'utilitza un objecte de la classe JPopupMenu. Aquesta classe té el mateix comportament que la classe JMenu, és un contenidor i l'omplim amb tants ítems com ens interessa. La diferència recau en que la classe JPopupMenu es mostra com menú flotant en les coordenades que li assignem enlloc de tenir una posició fixa.

Per poder controlar quan es produeix una selecció sobre un ítem del menú, necessitem un oient d'ActionEvents és a dir necessitem una classe que implementa d'interfície ActionListener. Per tenir tot el codi del tractament dels menús emergents junt la classe PopupListener.java hem fet que també implementi l' interfície ActionListener. Per cadascun dels ítems dels menús se'ls assigna com oient la pròpia classe.

Aleshores cada vegada que un event d'acció és capturat per la classe el tractament és el mateix que fèiem per la barra d'eines o la barra de menú. S'utilitza un switch del primer caràcter del camp Name de l'ítem que ha estat seleccionat per escollir l'acció a fer.

Algunes de les accions que es porten a terme requereixen canviar el contingut dels menús. Per exemple quan seleccionem l'opció "Habilitar mantenir la curvatura" del menú d'un estat, a part de fer el tractament sobre l'estat per modificar-li el valor de la variable booleana que determina si la curvatura és mante, ens interessa canviar el contingut de l'ítem. És a dir cridem una funció perquè actualitzi l' informació del label perquè la pròxima vegada que apareix-hi el menú emergent l'usuari vegi l'opció de "Deshabilitar mantenir la curvatura".

3.7.2 Menú per les transicions:

L' opció "Propiedades Transición" quan és seleccionada fa una crida al el mètode **actualitzarTransicio(Caixa)** de la classe CreadorTransicions.java.

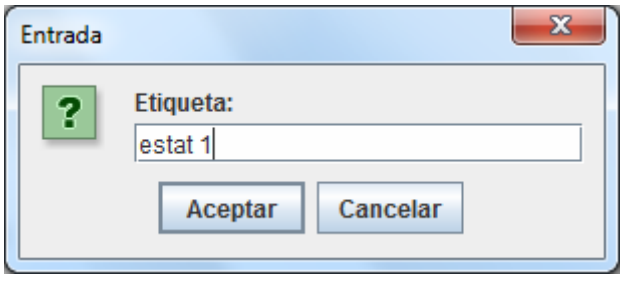
Bàsicament el que fa és mostrar una finestra emergent on s'observa l' informació actual de la transició i dona la possibilitat de canviar-la.

L'altra opció que apareix al menú només si és una transició composta és "Separar Subtransiciones". En aquest cas la transició és reduïda a varies transicions simples situades en posicions contigües. Per cada nova transició es creen nous estats perquè disposi de les mateixes connexions que la transició original.

3.7.3 Menú pels estats:

En el menú dels estats trobem l'opció "Habilitar mantener curvatura" o "Deshabilitar mantener curvatura" . El seu nom depenent del valor de la variable booleana corbatDesactivat de l'objecte de la classe Aresta. Aquesta variable determina si al moure una transició les arestes que en depenen són rectes o mantenen la curvatura. Cada vegada que es selecciona l'opció del menú el seu valor és negat.

L'altra opció del menú es "Cambiar etiqueta". Aquesta opció permet canviar el valor de l'etiqueta d'aquell estat. Utilitzem el mètode **showInputDialog(label, text per defecte)** de la classe JOptionPane. Aquesta classe ja l'hem utilitzat en la classe Main per mostrar de forma eficient una finestra emergent. En aquest cas, el mètode mostra per pantalla una finestra amb un quadre de text que conté l'actual valor de l'etiqueta.



Un cop introduït el nou valor de l'etiqueta si es selecciona el botó de confirmació s'actualitza el valor de l'etiqueta de l'estat en qüestió pel text que s'ha acabat d'introduir.

3.8 Creador de Transicions:

En el nostre programa utilitzem finestres emergents com a forma d'interactuar amb l'usuari perquè faci la seva selecció de les diferents opcions que li ensenyem.

En un primer moment vam utilitzar una finestra emergent com a mesura de seguretat davant del tancament de finestres en la classe Main.java. Aquella finestra era molt simple i simplement era configurada per paràmetres. En aquest apartat parlarem de finestres emergents amb un contingut més complex.

A continuació parlarem de les diferents classes i mètodes que creen finestres emergents, començant per la classe *CreadorTransicions.java.*

En una transició l'usuari ha d'especificar el valor dels camps que la conformen. Per cada transició s'ha d'especificar l' informació de la forma canònica, la categoria lèxica, l' informació morfològica i la sortida del transductor. Aquesta classe s'encarrega principalment de crear una transició per primer cop i també s'encarregarà de mostrar i actualitzar la seva informació.

3.8.1 Crear una nova transició:

Una transició a part de ser del tipus estàndard és a dir la típica caixa que conté l' informació lèxica, també pot ser connector, final o inicial. Els dos últims tipus de transicions actuen com estats inicials i finals, és a dir marquen on comença i acaba l'autòmat per tant només podem tenir una transició inicial i final. Totes les arestes que tenen com a destí la transició de tipus final són els estats finals.

Per donar cobertura a l' inserció de les transicions el que volem aconseguir és una finestra on es mostrin totes les opcions disponibles per construir la transició i a peu de la finestra dos botons un per acceptar i l'altre per cancel·lar l'operació.

Per crear aquesta finestra farem ús de la classe JOptionPane. En aquest cas la finestra emergent és complexa per tant hem de crear nosaltres un panell perquè es mostri juntament amb els botons d'acceptar i cancel·lar que afegeix el JOptionPane a la part inferior.

3.8.2 Construint el panell d'informació:

L'usuari escull el tipus de transició, i en cas de que sigui una transició de tipus normal, també ha de escollir entre les diferents opcions d'inicialització dels camps.

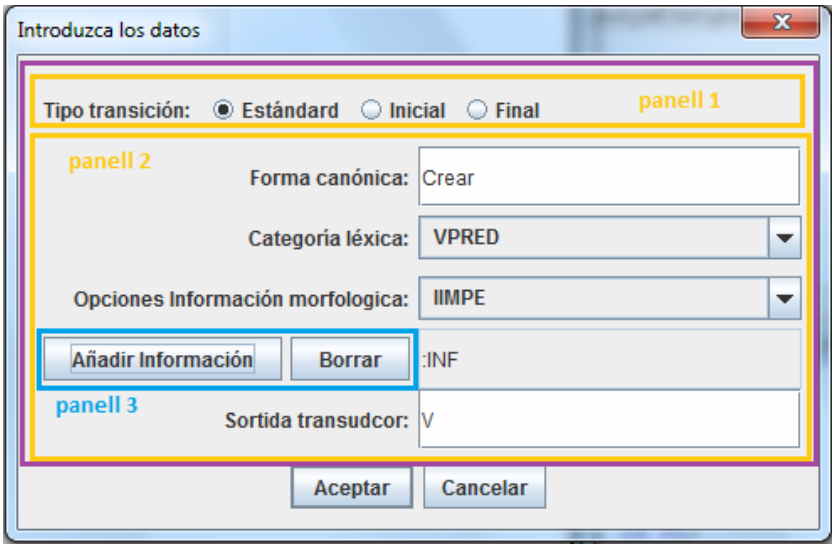
Un objecte de la classe JPanel és un contenidor de propòsit general per components lleugers. El JPanel que estem construint contindrà dos panells. En un dels panells mostrarà les opcions perquè l'usuari esculli quin tipus de transició vol afegir. En l'altre panell es mostraran les diferents opcions d'informació lèxica que li podem assignar a la transició.

Com que una transició pot ser només d'un sol tipus les opcions entre les que pot escollir són excloents per tant la millor manera d'implementar-ho ha estat amb un panell que conté quatre botons de la classe JRadioButton. Aquests botons han d'estar continguts dins d'un mateix objecte ButtonGrup que s'encarregarà de desactivar l'anterior opció habilitada cada vegada que l'usuari seleccioni l'opció d'un altre JRadioButton del mateix grup de botons.

La classe ZonaDibuix té un variable booleana anomenat noTenimInici i noTenimFinal que ens diu si tenim una transició inicial o final. En cas de tenir-ne no s'afegeix el JRadioButton que dona l'opció a crear una transició d'aquell tipus per tant l'usuari no podrà escollir-la.

Quan tenim seleccionat l'opció d'una transició de tipus connector, inicial o final, no volem que es mostri l' informació lèxica per tant implementem l'interfície ItemListener perquè escolti els ItemEvent's que generen els JRadioButtons. Quan l'opció no sigui tipus estàndard farem que el panell de selecció d'informació no sigui visible. L'avantatge d'aquesta implementació és que no es perd l' informació que es defineix en el panell de selecció d'informació per tant si l'usuari defineix una transició i per error selecciona el JRadioButton "inicial", si torna a seleccionar el botó "Estándard" no haurà perdut les dades introduïdes anteriorment.

En la següent imatge es mostra tot l'organització del JPanel:



El rectangle més exterior de color lila és el panell general que se li passa per paràmetre al JOptionPane. S'utilitza el BorderLayout com a manager de Layout. Dins el panell general tenim els dos panells marcats en color taronja. En el panell superior s'escull l'opció i quan aquesta és diferent a la primera opció el panell inferior se li treu la propietat de visible.

L'anterior imatge ens serveix de guia per comentar els elements del segon panell o panell de selecció d'informació.

El LayoutManager del panell de selecció d'informació s'anomena GridLayout. Aquest manager se li passa en el seu constructor el nombre d'elements que es vol contenir per files i columnes en el panell. Els elements per defecte s'afegeixen d'esquerra a dreta i de dalt a baix però la política d'inserció es pot canviar fàcilment. En l'anterior imatge podem observar que la posició (4,1) s'ha afegit un panell amb dos botons per tenir en una mateixa ubicació dos elements i així obtenir una millor organització del contingut permetent que a la seva dreta es pugui situar el quadre de text sobre el qual treballen sense que això provoqui una desorganització de la resta del contingut.

Com es pot observar a l'esquerra situem tots els Labels amb informació que guiaran a l'usuari sobre quins camps està seleccionant l' informació.

Pel cas de l' informació sobre la forma canònica i la sortida del transductor, s'afegeix un JTextField perquè l'usuari entri l' informació per teclat.

En el cas de l' informació sobre la categoria gramatical com que el número d'opcions possibles és finit s'ha decidit utilitzar un objecte de tipus JComboBox amb les diferents opcions disponibles. Un JComboBox no és altra cosa que una llista desplegable.

Aquesta llista està inicialitzada amb diferents valors segons quines opcions se li puguin assignar a cada camp.

L'avantatge d'utilitzar els objectes de la classe JComboBox i JTextField és que accedir al seu contingut és realment senzill. En un cas hem de fer **JComboBox.getSelectedItem().toString()** i en l'altre cas hem de fer **JTextField.getText()**.

En quant a l' informació morfològica el número d'opcions possibles també es finit però a diferència de la categoria gramatical, se li ha de poder assignar tantes com l'usuari cregui convenient.

Per fer possible afegir-li informació sense saber la seva llargada a priori utilitzem per una banda un JComboBox on es mostren a l'usuari totes les opcions disponibles. Per altra banda tenim un JTextField on s'anirà afegint l' informació de manera progressiva. Per controlar que l'usuari no introdueixi informació incorrecta s'ha deshabilitat l'opció d'omplir el quadre de text per teclat. L'única manera d'introduir informació morfològica és a través dels dos botons situats a la seva esquerra que els seus events són tractats per la pròpia classe que implementa també l'interfície ActionListener. El botó "Añadir información" afegeix ":"+l' informació que conté l'ítem seleccionat del JComboBox a l'String del quadre de text. L'única restricció que s'ha de complir al crear l' informació morfològica és que no es repeteixi informació per tant es treu del JComboBox l'ítem que conté l' informació que s'acaba d'afegir. El botó "Borrar" fa un reset de l' informacó morfològica i al JComboBox li torna a afegir totes les opcions possibles.

3.8.3 Creant la finestra emergent:

Un cop hem creat el panell que desitgem mostrar per pantalla només falta posar els botons d'acceptar o cancel·lar l'operació. Com ja havíem introduït podem utilitzar la classe JOptionPane i així evitem haver de crear els botons i implementar l' interfície ActionListener per gestionar la seva selecció. Utilitzant el mètode **JOptionPane.showConfirmDialog()** creem aquesta finestra que d'una manera senzilla i eficaç en unes poques línees de codi. Per crear-la hem de donar valor als paràmetres del mètode que són els següents:

- parentComponent: determina el component sobre el qual es mostrarà la finestra emergent. No es pot continuar amb el parentComponent fins que es tanca la finestra. En el nostre cas el parentComponent és l'objecte ZonaDibuix que vol crear la transició. D'aquesta manera s'evita poder obrir varies finestres de creació de transició.

- Message: aquest és l'objecte que mostrarà la finestra emergent, en el nostre cas ensenyarem el JPanel.
- Title: Ha de ser una cadena de caràcters que determinarà el títol de la finestra.
- optionType: Demana un enter que indica quina opció volem que tingui la finestra. Els valors possibles són: :YES_NO_OPTION, YES_NO_CANCEL_OPTION, or OK_CANCEL_OPTION . Nosaltres hem triat OK_CANCEL_OPTION.
- messageType: Demana un enter que indica quin tipus de missatge estem ensenyant. Aquest tipus determina quina icona es veurà. Les diferents opcions són: ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, PLAIN_MESSAGE i la que nosaltres hem escollit PLAIN_MESSAGE.

La crida d'aquest mètode retorna un enter que representa l'opció que s'ha seleccionat. Nosaltres comprovem que s'hagi seleccionat l'opció de confirmació. Si el valor de retorn es correspon amb el valor de JOptionPane.OK_OPTION vol dir que l'usuari ha omplert tots els camps referents al contingut de la transició i ha confirmat la seva creació premut el botó d'OK.

En cas de que s'hagi cancel·lat l'operació el que retornem és null. En altre cas retornem un objecte de tipus Caixa del tipus corresponent en les coordenades on s'havia fet clic amb el ratolí. Si el tipus caixa és normal al constructor li hem de passar l' informació que contenen els diferents camps del panell.

3.8.4 Actualitzar la transició:

Aquest mètode s'encarregarà d'oferir a l'usuari la possibilitat de canviar el contingut d'una transició de tipus estàndard un com està afegida a l'escena. D'aquesta manera si l'usuari en un moment donat vol canviar l' informació d'una transició no es veu obligat a eliminar-la per tornar-la a afegir amb el contingut canviat. Encara que en un primer moment no se li pugui donar gaire importància a aquest mètode és molt útil per casos amb transicions que tenen molts estats perquè al eliminar la transició també s'eliminaven tots ells.

Amb el mètode **actualitzarTransicio(Caixa),** mostrarem el mateix panell de l'informació lèxica de la mateixa manera que ho havíem fet en la creació de les transicions però amb dos petits detalls diferents. L'únic que es permet canviar és el contingut de la transició, no el tipus per tant no es mostrarà el panell de selecció del tipus. L'altra diferència es que s'inicialitzen els camps perquè mostrin l'informació que en aquell moment conté la transició.

Per fer aquesta inicialització primer de tot s'ha de separar l'informació que s'havia comprimit en un única cadena de caràcters en els seus 3 camps utilitzant el mètode

lastIndexOf(“.”) que retorna -1 en cas que no tinguem informació sobre la forma canònica. En altre cas l’informació és la sub-cadena de caràcters entre la posició 0 i l’índex. La resta de caràcters contenen l’ informació sobre la categoria gramatical i l’informació morfològica que els separa el primer caràcter “:” en cas d’existir. Per fer-ho en aquest cas utilitzem el mètode **indexOf(“:”)** que ens retorna la posició on es troba el separador o -1 en cas de no trobar-ho.

Un cop ja tenim aquesta informació ja podem instanciar els JTextFields referents a la forma canònica, l’ informació morfològica i la sortida del transductor (la caixa la guarda en una variable a part).

Pels JComboBox tractem dos casos diferents: pel cas de la categoria inicialitzem el JComboBox amb totes les opcions possibles. A continuació busquem quina de les opcions es correspon amb la que conté la transició per seleccionar aquella opció.

El JCombobox de l’ informació morfològica se l’inicialitza amb totes les opcions possibles per a continuació eliminar les opcions que ja es troben en el JTextField. Com havíem fet anteriorment per extreure l’informació d’una única cadena de caràcters utilitzem el mètode *lastIndexOf(“:”)* recursivament. Per cada opció busquem amb quin ítem del JComboBox es correspon i s’elimina.

Un cop ja tenim el panell que volem mostrar configurat el mostrem cridant el mètode **JOptionPane.showConfirmDialog (...)**. Si el retorn del JOptionPane és correspon a l’opció d’OK, s’actualitzarà l’objecte Caixa amb la nova informació.

S’ha de tenir en compte que a part d’assignar la nova informació a les variables que la guarden. La nova informació pot tenir una mida diferent que l’ anterior i per tant s’ha de recalcular el valor dels separadors perquè l’informació es mostri centrada dins la transició. Per fer això cridem el mètode **actualitzarTamanySeparadors()** de l’objecte Caixa que s’encarrega de definir els valors dels nous separador segons la llargada de la nova informació de la mateixa manera que ho havíem fet quan construirem la transició per primer cop.

El fet de que la mida de la transició hagi variat afecta a la posició dels estats de la transició els quals ara no estaran situats als laterals de la transició. Per col·locar-los al seu lloc utilitzem de la classe ZonaDibuix.java el mètode **actualitzarExtremsEstat(IDtransicio,xIniciTransicio,xFiTransicio,yCentreTransicio)**.
Aquest mètode busca els estats que tenen en algun extrem la transició i actualitza les coordenades del extrem a la seva nova posició.

3.9 Llegir fitxer en Format Autòmat

La nostra aplicació ha de ser capaç de guardar la feina feta així com recuperar-la per continuar treballant. El format que s'utilitza per emmagatzemar l' informació és el format autòmat (FA).

3.9.1 Format autòmat:

```
4 4   (*1)
 %<VPRED\3>%<ADV>%<PALABRA\4>[S]%<PALABRA\5>%       (*2)
: 0 2 -1     (*3)
t 1 2 2 3 3 4  -1  (*4)
t 3 4  -1   (*5)
t  -1 (*6)
f   (*7)
```

(*1) el primer nombre especifica el # de tokens i el segon el # d'estats

(*2) El separador de tokens és el símbol '%' que marca el inici i el final de cada token. Els token representa l' informació lèxica dels diferents transició que es poden utilitzar per crear l'escena. Cada token té una id implícita segons l'ordre d'aparició: id = 0,1,2,3. Si una transició té associada informació de sortida del transductor, aquesta apareixerà entre claudàtors com podem observar en el token id=2.

A continuació de la capçalera i dels diferents tokens ens trobem la definició dels estats:

Cada sentencia es la definició d'un estat començant per l'estat 1. La fi de sentencia o fi de definició de l'estat ve especificada per l'element '-1'.

L'estructura de la definició d'estat és la següent:

Primer tenim un caràcter que determina si l'estat és final ('t') o no (':'). L' informació següent fins el fi de sentencia s'ha de tractar per parelles. Cada parella d'enters especifica "amb quin token" anem a "# estat".

(*3) estat 1 (estat inicial) el token id = 0 porta al estat 2

(*4) estat 2 el token amb id= 1 porta a estat 2. El token amb id = 2 porta a l'estat 3. El token amb id = 3 porta a esat 4

(*5) estat 3

(*6) estat 4

(*7) el caràcter ‘f’ determina el final de la definició de l’autòmat

3.9.2 Crear escena a partir fitxer en FA:

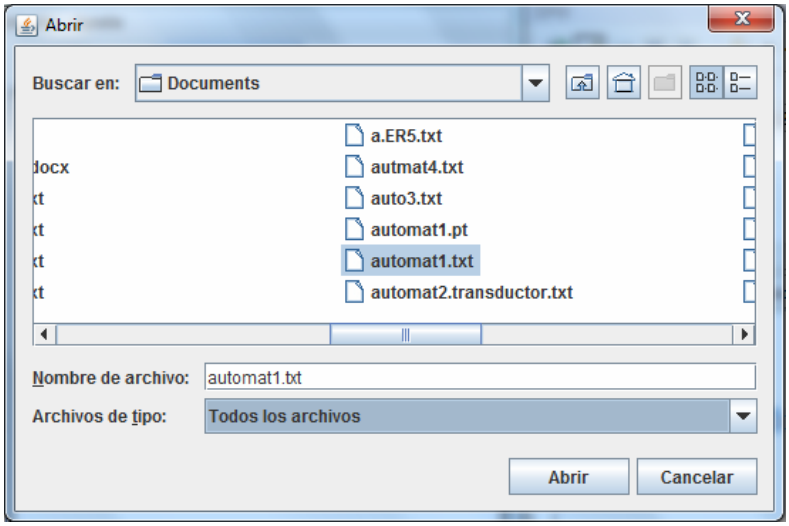
Per obrir un nou document a partir d'un fitxer en FA autòmat s'ha de seleccionat l'opció "Abrir" del menú "Inicio". Quan la classe Main.java captura aquest event crea una nova pestanya (una nova escena on treballar), crea un objecte de la classe AutomatsPerFitxer.java i afegeix l'autòmat a l'escena amb el mètode **llegirAutomatDeFitxer()**. Aquest mètode retorna un booleà amb valor false si s'ha produït algun problema al llegir el fitxer, en aquest cas s'elimina la nova pestanya.

3.9.3 Classe AutomatsPerFitxer:

S'ha creat una classe específicament encarregada de definir els elements d'una escena a partir de l'informació d'un fitxer en FA. En el constructor de la classe s'ha de passar per paràmetre l'escena (objecte ZonaDibuix) sobre la que es representarà el nou autòmat.

Per escollir el fitxer s'utilitza un objecte de la classe JFileChooser. Aquesta classe ens crea fàcilment un selector de fitxers en una finestra emergent. Els selectors de fitxers s'utilitzen normalment per presentar una llista de fitxers que poden ser "oberts" per l'aplicació i per introduir la ruta i el nom d'un fitxer a guardar.

La classe JFileChooser ni obre ni guarda fitxers, únicament ens presenta una GUI per escollir un fitxer i ens permet navegar per les diferents carpetes del nostre PC. Podem definir-li el mode selecció si volem que només obri fitxers, com és el nostre cas, o si volem que també obri carpetes.



Per crear la finestra emergent utilitzem el mètode **showOpenDialog(Object)**en que té les mateixes propietats que els mètodes *showXXXXDialog(Object)* de la classe JOptionPane ja comentats.

Aquest mètode ens retorna un valor enter segons si s'ha seleccionat un fitxer (JFileChooser.APPROVE_OPTION) o no. En cas afirmatiu el fitxer l'obtenim amb el mètode **getSelectedFile()**. Un cop ja disposem de la ruta al fitxer utilitzem un objecte de la classe FileInputStream(File) per anar llegint el fitxer caràcter a caràcter.

La lectura la dividirem en tres etapes: llegir la capçalera, llegir els diferents tokens i llegir els estats. Si ens trobem amb algun error en alguna etapa s'atura la lectura i avisem a la classe Main que elimini la nova pestanya perquè l'autòmat no es pot representar.

3.9.4 Llegir capçalera:

Hem de llegir dos enters per instanciar el nombre de tokens que hem de llegir i el nombre d'estats. Hem de comprovar que aquests nombres són diferent de zero.

3.9.5 Llegir tokens:

Cada token que llegim el guardem com una transició on el seu valor de ID li assignen la posició segons l'ordre de lectura. Com que no s'eliminen transicions l'ID de cada transició es correspon en la posició en la llista de transicions. Aquest detall ens serà de gran utilitat al llegir els estats perquè ens evitarà haver de buscar la transició que es correspon amb una determinada ID.

Al llegir els tokens s'ha de tenir en compte que aquests venen delimitats pel símbol '%'. També sabem que la seva informació lèxica ve definida dins els simbols '<' i '>' i que opcionalment pot tenir definida la sortida del transductor entre '[' i ']'.

El fitxer no determina la posició de les transicions per tant utilitzarem la variable booleana seleccionat de cada transició per saber si una transició ja se li ha assignat una posició o no. El valor serà true si en cas que encara no se li hagi assignat.

Un cop llegits tots els tokens afegim a la llista de transicions la transició inicial i final, i guardem en una variable global el valor de la ID d'aquesta. D'aquesta manera quan un estat sigui final ja sabrem la posició de la transició final.

La posició de les transicions se li assignarà dinàmicament al construir l'autòmat exceptuant la transició inicial que se li assigna la seva posició al ser creada. Quan hem d'afegir un estat entre dos transicions comprovem la transició destí, si el valor de la seva variable seleccionat és true vol dir que encara no ha estat situada i se li assigna la posició a la dreta de la transició origen. Si ja hi ha alguna transició en aquella posició se la situa sota seu.

Si un usuari estava treballant amb un autòmat i el guarda, la posició de les seves transicions serà diferent al ser obert de nou el fitxer. Per solucionar aquest problema al guardar l'autòmat també es guardarà en el mateix directori un arxiu amb extensió ".pt" que contindrà les Posicions de les Transicions. Per tant un cop s'han llegit tots els tokens es comprova si en el directori on esta el fitxer es troba el fitxer amb el mateix nom però amb l'extensió mencionada. Si existeix es llegeix i s'assigna a cada transició la seva posició.

El fitxer ".pt" té la següent estructura:

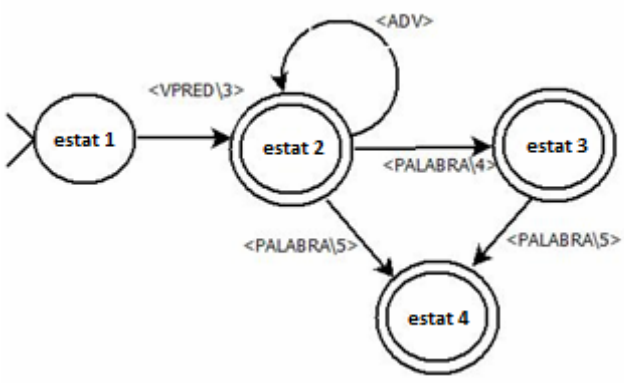
3 (*1)
0 44 33 (*2)
1 250 33
2 200 99
10 10 (*3)
400 260 (*4)

(*1) El primer element determina el nombre de transicions del fitxer
(*2) El primer valor correspon la posició de la transició segons l'ordre d'aparició del fitxer, els següents dos valors enters determinen les coordenades.
(*3) i (*4) Coordenades transició inicial i final

3.9.6 Llegir els estats:

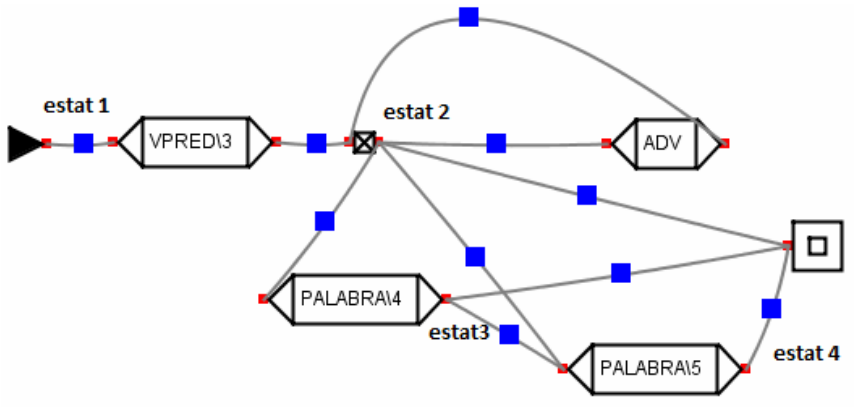
Un estat del fitxer en FA a l'hora de representar-lo hem de fer una traducció per saber on els trobem en la nostra representació per anem a representar l'autòmat determinat pel fitxer:

4 4
%<VPRED\3>%<ADV>%<PALABRA\4>%<PALABRA\5>%
: 0 2 -1
t 1 2 2 3 3 4 -1
t 3 4 -1
t -1
f



La seva representació en un AFD és la següent:

La conversió a la nostra representació és la següent:



Les transicions se'ls connecten per l'esquerra amb un estat (aresta) que representa l'estat d'es d' on arriba el token. A la seva dreta trobem en l'estat resultant d'haver-nos trobat el token que simbolitza la transició.

Per tenir una referència al llegir del nostre fitxer en FA els estats correspondran al node que apareix a la banda dreta de les transicions o a una transició de tipus connector.

Un cop entès el concepte procedim amb aquest apartat que forma el nucli de la construcció de l'autòmat. Es basa en un bucle que finalitza quan després d'una definició d'estat ens trobem amb el caràcter '-1' o si és 'f' que vol dir que s'ha arribat al final de fitxer.

S'utilitza una variable entera anomenada estatActual la qual en cada iteració del bucle s'incrementa en 1. Cada iteració del bucle equival a analitzar una definició d'estat. Per tant la variable estatActual està instanciada en tot moment amb el valor de l'estat que s'està processant. S'utilitza per saber quina és la transició origen dels estats que s'hagin d'afegir. D'ara endavant els estats del fitxer els anomenarem estats absoluts per diferenciar dels estats que afegim a les transicions.

Per guardar la relació entre estats absoluts i transicions que representen aquests estats s'ha creat una variable local anomenada taulaDeSimbols. Aquesta variable és una llista d'elements entradaTS. Aquests elements contenen la següent informació: estat que representen, ID de la transició que representa l'estat i una variable booleana que ens diu si la transició és de tipus connector o no. La taula de símbols és inicialitzada amb l'entrada estat = 1, transicio = ID transicio inicial, esConector =false.

Accions a portar a terme:
1: En cada iteració primerament es recorre tota la definició de l'estat i es guarda en una llista els diferents valors que s'han llegit. En les posicions parelles de la llista es troba l' ID de la transició i en les posicions parelles l'identificador a l'estat que s'ha d'afegir l'aresta. Aquesta llista l'anomenarem llistaDefinicio.

2: Es recorre la llista llistaDefinició buscant si hi ha un bucle sobre aquell estat. Perquè hi hagi un bucle s'ha de buscar si en alguna posició parella de la llista el seu valor es correspon amb el valor de l'estatActual.

En el cas que es tingui un bucle aquell estat ha d'estar representat per una transició tipus connector. Si no n'és el cas s'ha de crear un nou estat tipus connector per poder reproduir el bucle. De la taula de símbols s'ha de canviar la transició que representava aquell estat per la ID de la nova transició. S'ha d'actualitzar totes les arestes que apuntaven a la transició que anteriorment representava aquell estat perquè apuntin a la nova transició i s'ha d'afegir una transició entre elles dos.

3: Es busca a la taula de símbols quina és la transició que representa l'estat actual i que serà l'origen de les arestes que s'afegeixin.

4: Es recorre la llista llistaDefinició per treballar amb el parell de valors que ens determinen l'estat següent i la transició per arribar a aquell estat.

A continuació es busca en la taula de símbols la transició que representa l'estat següent. Si no es troba en la taula de símbols s'afegeix una nova entrada amb aquest parell de valors i s'afegeix una aresta entre la transició origen i la transició que defineix el nou estat.

Si es troba en la taula de símbols es comprova si la transició és tipus connector o no. En cas afirmatiu s'afegeix una aresta entre la transició origen i la transició de la llistaDefinicio i una aresta entre aquesta última transició i el connector.

En cas que l'estat sigui representat per una transició normal s'ha de comprovar si la transició que actualment representa l'estat i la que es vol utilitzar per accedir aquell estat són la mateixa. Si no ho són s'ha de crear un connector perquè totes les transicions que vulguin estar en a aquell estat connectin amb ella.

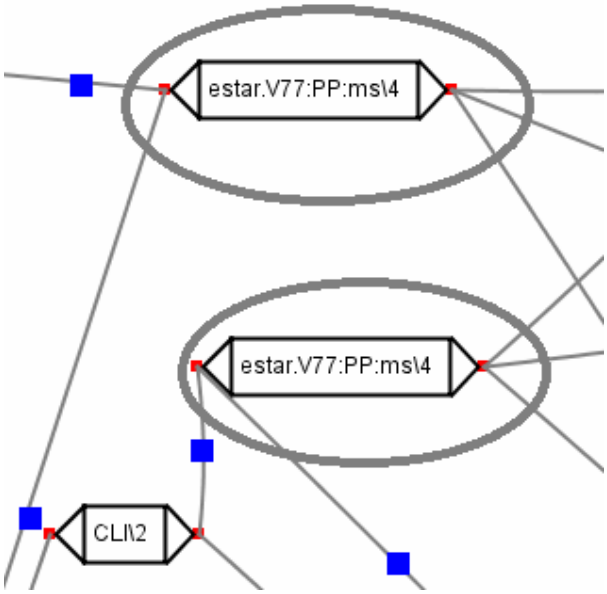
6: Si l'estat actual és un estat final s'afegeix una aresta entre la transició origen i la transició final.

Un cop ja s'ha processat tots els estats afegim la llista de transicions i llista d'estats a l'objecte ZonaDibuix que representa l'escena creada. D'aquesta manera es mostra el contingut del fitxer per pantalla.

3.9.7 Casos especials:

E-transicions: Per definir que des d'un estat absolut es pot anar a un altre estat absolut sense cap transició, el valor que s'utilitza com a token és el -2. Quan l'algorisme es troba amb aquest cas afegeix una aresta entre l'estat origen i la transició que representa l'estat destí. En cas que no existeixi l'última transició es crea un connector.

Utilitzar un mateix token per accedir a diferents estats:

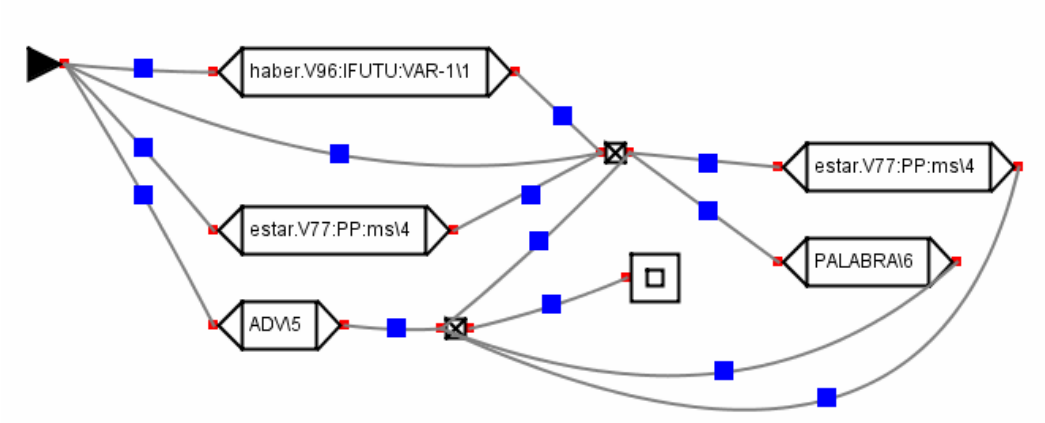


Els tokens els guardem en transicions. Un cop s'ha utilitzat una transició per accedir a un estat absolut totes les arestes que apunten a ella apunten a un mateix estat absolut. En el cas de la imatge ens trobem que es vol utilitzar la mateixa informació lèxica d'una transició transició però per accedir a un estat absolut diferent. Quan ens trobem amb aquesta situació es clona la transició que conté aquell token i s'afegeix a la taula de símbols el nou estat absolut i la ID de la nova transició clonada. Per comprovar si ens trobem en aquest cas s'utilitza el mètode **Checkduplicat.(estat, transicio, TS).**

També s'ha de controlar que podem estar davant d'un cas de duplicat encara que no s'accedeixi a un nou estat sinó a un estat existent.

Si analitzem la següent definició d'estat:

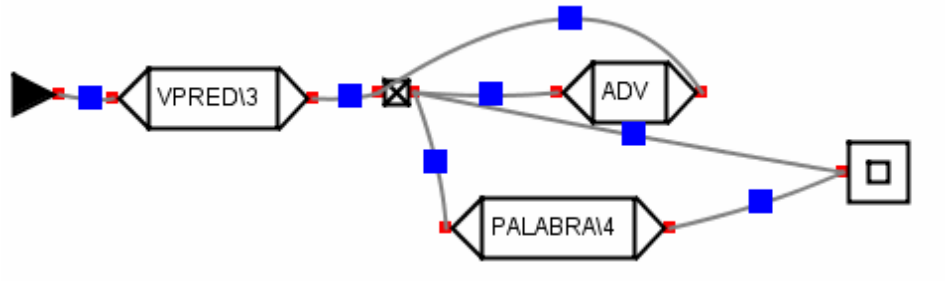
: 0 2 1 2 -2 2 2 3 -1
: 1 3 3 3 -2 3 -1
t -1
f



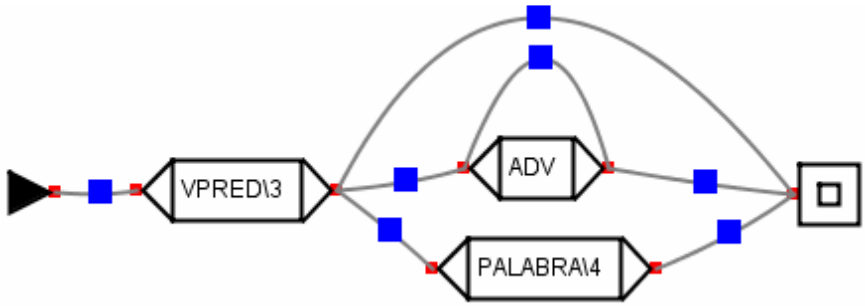
En aquest autòmat podem observar una E-transició entre l'estat 2 i 3 (els dos connectors).
També ens trobem en un cas de duplicat amb la transició <estar.v77:PP:ms\4>
Per resoldre aquestes situacions es fa ús del mètode *checkduplicat(estat,transició,TS)* durant l'anàlisi de les connexions. Aquest mètode a part de buscar una coincidència amb les transicions de la taula de símbols, busca si la transició original té una aresta amb un connector. En aquest cas a no ser que els estats absoluts coincideixin vol dir que ens trobem davant un cas de duplicitat.

Bucles: Ens podem trobar dos casos diferents:
Cas 1:
: 0 2 -1
t 1 2 2 3 -1
t -1
f

La seva ER seria equivalent a <t0>(<t1><t1>*+<t2>) i es representa de la següent manera:



Cas 2 :
: 0 2 -1
t 1 3 2 4 -1
t 1 3
t -1
f
La seva ER equivalen es correspon amb: <t0>(<t1>*+<t2>) i es representa de la següent manera:



Per saber de quin cas es tracta a l'hora de comprovar si és un bucle comprovar si la transició d'origen i final és la mateixa. Si és la mateixa o equivalent en cas que sigui un duplicat, vol dir que ens trobem davant d'un cas 2. En aquest cas no s'ha de crear la transició connector que creàvem en l'apartat 2. En cas de trobar-nos un bucle de cas 1 s'ha d'afegir una aresta entre la transició origen (connector) i la nova transició i la seva inversa per tancar el bucle.

3.10 Guardar fitxer en Format Autòmat (FA)

Tot el que es representa a l'escena s'ha de poder guardar per poder continuar des del mateix punt en un altre moment. Per guardar les dades utilitzem un fitxer en FA i el seu associat .pt per guardar l'autòmat que s'està editant.

Aquest procés s'encarrega la classe *AutomatsAFitxer.java*. Aquesta classe està dedicada a recollir l' informació de l'escena per guardar-la en l'estructura definida dels fitxers en FA.

Per escollir el directori on guardar fitxer i el seu nom s'utilitza un objecte de la classe JFileChooser. Aquesta vegada s'invoca el mètode **showSaveDialog(Object)** per mostrar la finestra emergent. Aquest mètode en retorna si s'ha escollit guardar o cancel·lar l'operació. En cas que s'hagi guardat ens proporciona la ruta on guardar el fitxer.

3.10.1 Requisits:

Abans de fer qualsevol operació s'ha de comprovar que l'autòmat té definit una transició inicial. L'autòmat que guardarem està composat pel conjunt d'estats i transicions els quals són accessibles des de l'estat inicial. En cas que l'escena no disposi de la transició inicial és mostra per pantalla una notificació d'error amb *JOptionPane.showMessageDialog(..)* definint-li per paràmetre que és un missatge d'error (JOptionPane.ERROR_MESSAGE).

3.10.2 Implementació del procés de guardat del fitxer FA:

El fitxer en FA té dos apartats importants: la llista de tokens i la definició d'estats.

3.10.2.1 Recopilació dels tokens:

Per una banda es crea la llista de tokens que disposa l'autòmat. Recorrem totes les transicions de tipus estàndard i guardem en una llista la seva informació lèxica. En cas que sigui una transició complexa i tingui més d'una informació és concatenen <info1><info2><info3>[SortidaTransductor]. Com que els tokens seran separats pel símbol '%' quan es carregui aquest fitxer ja s'encarregarà de construir correctament la transició complexa.

S'ha de realitzar primer aquest procés perquè quan estiguem definint els estats el que s'especifica és la posició del token en la llista.

3.10.2.2 Definició d'estats:

Per definir els estats per una banda necessitem una taula de símbols que és una llista d'objectes EntradaTS_Save. Cada entrada de la taula de símbols representa un estat absolut del nostre autòmat. Els objectes EntradaTS_Save guarden l' informació referent a l'estat que representen, la transició que els representa, si són un connector i si l'estat es final.

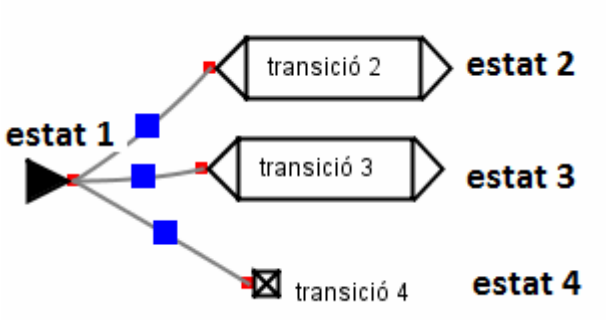
En la taula de símbols tindrem emmagatzemats els diferents estats absoluts de l'autòmat i la transició que el representa. La informació referent a quin token s'ha d'especificar per anar a un altre estat la guardem en TaulaDefinició. TaulaDefinició és una llista d'objectes EntradaDefinició. En aquests objectes guardarem l'estat origen, l'estat destí i la transició que produeix aquest canvi d'estat i el token d'aquella transició.

Inicialment afegim a la taula de símbols la transició inicial. A continuació es crida el mètode **guardarEstats(estatActual, transicioRepresentaEstatActual)**. Aquest mètode recursiu analitza la transició definida pel segon paràmetre de la funció, busca els seus següents i va creant els estats de la taula de símbols i les entrades de la TaulaDefinició a mesura que va analitzant.

Per guardar correctament la definició d'estats s'ha de preveure quines connexions ens podem trobar en l'autòmat i interpretar-les correctament. Per cada iteració el primer paràmetre del mètode ens determina quin estat absolut s'està tractant, el segon paràmetre és l'identificador de la transició que representa aquell estat. Els estats absoluts són una transició de tipus connector o una transició estàndard. L'interpretació de que és per nosaltres un estat absolut sobre el nostre autòmat es correspon amb el significat d'arribar a un estat amb un token determinat.

A continuació s'especifiquen els diferents casos possibles amb el seu tractament corresponent:

Cas base:



Taula de símbols :

Estat	ID estat inicial	EsConnector
1	Transició inicial	false

(estat 1, ID estat inicial, no és connector)

Pel cas base busquem els següents de la transició transicioRepresentaEstatActual. Per cadascun d'aquestes si la seva ID no està en la taula de símbols vol dir que ens trobem amb un nou estat absolut i per tant l'afegim a la TS i a la taula de Definició. Es fa una crida recursiva del mètode **guardarEstats()** amb cadascun dels següents.

Taula de símbols :

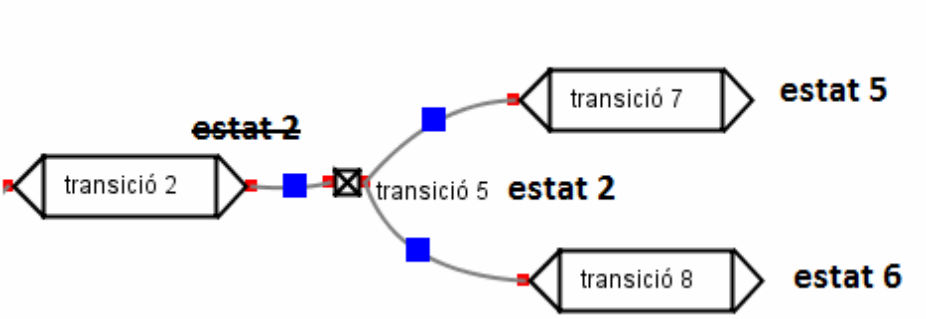
Estat	ID estat inicial	EsConnector
1	Transició inicial	False
2	Transició 2	False
3	Transició 3	False
4	Transició4	True

Taula de Definició:

Estat origen	Estat destí	Id Transició
1	2	Transició 2
1	3	Transició 3
1	4	-2

Pel tractament del cas d'una E-transició entre estats absoluts, és a dir anar d'un estat absolut a un connector (que no té token) s'utilitza la constant -2.

Cas actualitzar la taula de símbols:



Quan analitzem una transició si aquesta no és un connector i ens trobem davant del cas que la transició només té una transició següent i aquesta és un connector s'ha d'actualitzar la taula de símbols perquè el verdader estat absolut està en la transició connector.

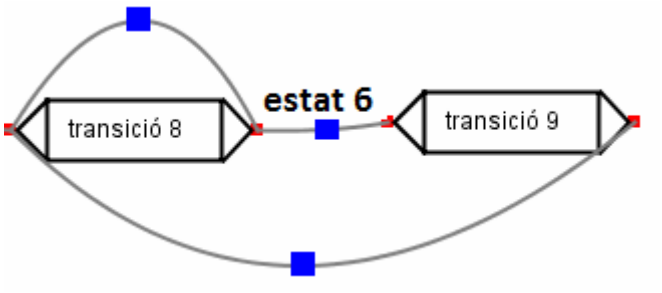
Taula de símbols :

Estat	ID estat inicial	EsConnector
2	~~Transició 2~~ transició 5	True
5	Transició 7	False
6	Transició 8	False

Taula de Definició:

Estat origen	Estat destí	Id Transició
1	2	Transició 2
2	5	Transició 7
2	8	Transició 8

Cas loop sobre estat absolut:



El cas de tenir un bucle unitari o loop sobre un estat absolut pot ser representat de dos maneres diferents. Pel cas d'un bucle sobre una mateixa transició no hi ha cap problema en reconèixer-ho. Per identificar el cas d'un bucle amb una altra transició per cada un dels següents de la transició que s'està analitzant s'ha comprovar si aquella transició només té un següent i si aquest coincideix amb la transició actual. En aquest casos

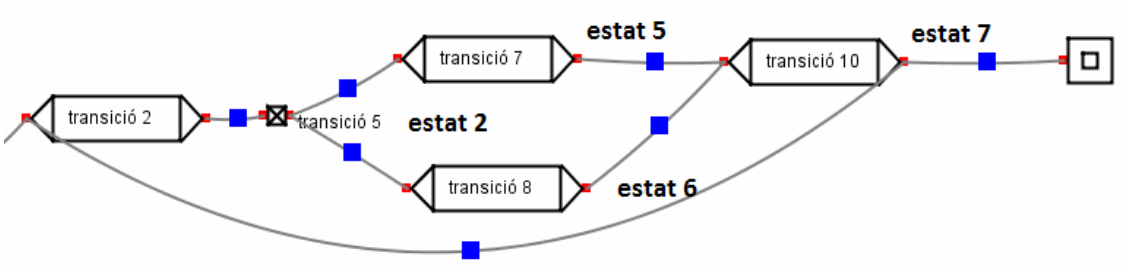
Taula de símbols :

Estat	ID estat inicial	EsConnector
6	Transició 8	False

Taula de Definició:

Estat origen	Estat destí	Id Transició
6	6	Transició 8
6	6	Transició 9

Cas parada:



El moment que parem de cridar recursivament el mètode a part del cas dels loops és quan ja s'han analitzat les transicions. Per exemple si estem analitzant la transició 8 ens trobem que el seu següent és la transició 10, però aquesta ja ha estat avaluada i està en la TS per tant simplement s'afegeix al a taula de Definició una nova transició.

Per altra banda quan analitzem la transició 10 veiem que el seu següent és la transició 2 però encara que ha estat analitzat aquest no està en la TS. Per tant l'altra comprovació per saber si un estat ha estat analitzat o no és buscar en la taula de Definicions si hi ha alguna transició entre estats absoluts que utilitzi aquella transició.

Si el següent d'una transició és la transició final vol dir que aquell estat absolut és final. Els elements entradaTS_Save tenen un paràmetre booleà anomenat EsFinal que es posa true en aquesta situació.

Taula de símbols :

Estat	ID estat inicial	EsConnector
2	Transició 5	False
5	Transició 7	False
6	Transició 8	False
7	Transició 10	False

Taula de Definició:

Estat origen	Estat destí	Id Transició
1	2	Transició 2
2	5	Transició 7
2	6	Transició 8
5	7	Transició 10
6	7	Transició 10
7	2	Transició 2

Un cop ja s'ha analitzat tot l'autòmat recorrem la taula de Definicions assignant el seu valor corresponent al camp token segons l' informació lèxica de la transició.

A continuació es guarda en un fitxer el nombre de tokens i el nombre d'estats (el nombre d'elements de la TS). A continuació es guarden els diferents tokens separats pel símbol '%' i a continuació es guarden els estats.

Per cada estat primer es busca en la TS i es mira si és final ('t') o no (':'). La resta d'informació necessària es troba en la taula de Definició. S'han de buscar les entrades que tenen com estat origen l'estat que s'està guardant per escriure al fitxer el token i

l'estat destí. S'escriu el símbol de fi de definició d'estat (-1). Quan ja s'han guardat tots els estats s'escriu el símbol de fi de definició de l'autòmat ('f').

3.10.3 Procés de guardat del fitxer de posicions (PT):

Al obrir un fitxer se'ls assigna dinàmicament la posició a cada transició però això pot no ser sempre útil per l'usuari final. L'usuari vol que quan carregui un fitxer les transicions estiguin en la mateixa posició que on les havia col·locat ell. Com que en el fitxer en FA no hi ha lloc per guardar aquesta informació al guardar en FA també es crea un fitxer amb el mateix nom però amb extensió ".pt". En aquest fitxer conté el nombre de tokens, i a cada fila hi ha tres enters corresponents a la posició d'aparició del token, i les seves coordenades x i y. A continuació hi ha 2 coordenades més corresponents a la transició inicial i la transició final.

Exemple fitxer de posicions:

```
4
0 90 120  //token 0 té coordenades (90,120)
1 181 120
2 272 120
3 181 190
30 120 //coordenades transició inicial
615 120 // coordenades  transició final
```

3.11 Llegir autòmat per ER

La nostre aplicació ha de ser capaç carregar unes dades amb les que treballar. Ja carrega arxius en FA però l'usuari a qui va destinada aquesta aplicació li interessa també que es representi un autòmat a partir de la seva Expressió Regular.

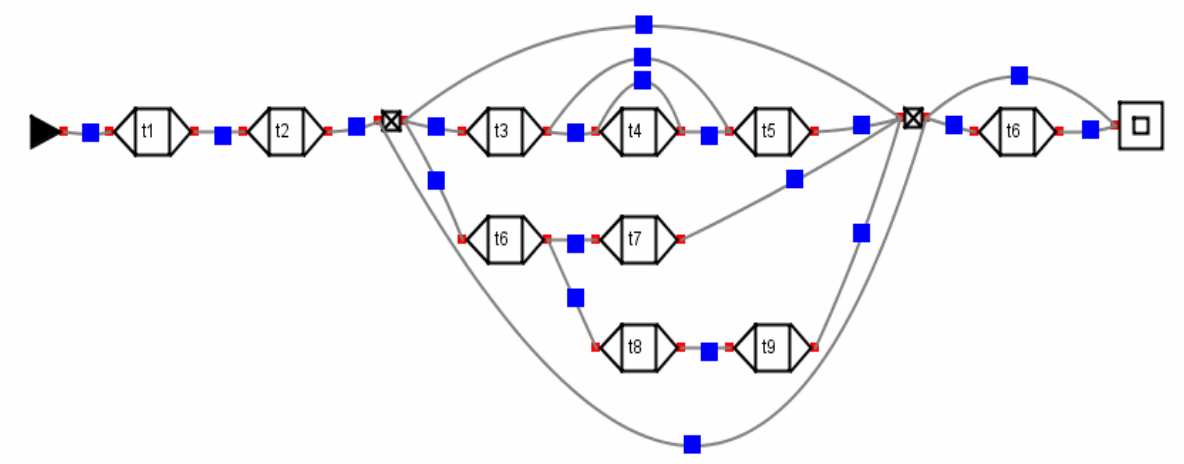
Un exemple d'expressió regular que ha de poder llegir seria el següent:

<t1><t2>(<t3><t4>*<t5>+<t6>(<t7>+<t8><t9>))*(<t6>+<E>)

La sintaxis de les ER ve definida pels següents patrons:

- L'operació AND entre dos elements s'escriuen un a continuació de l'altre.
- L'operació OR s'ha d'escriure les diferents opcions entre parèntesis i separades per l'operador '+'.
- Definir que un element pot repetir 0 o més vegades s'utilitza '*'.
- Definir que un element pot estar o no estar. Fer una operació OR amb l'element <E>

La nostre aplicació representa l'anterior ER de la següent manera:



Resposta a les diferents operacions:

- Quan ens trobem l'operació AND s'afegeix una aresta entre els dos elements
- Quan ens trobem l'operació OR s'afegeixen arestes entre l'anterior element i el primer element de cada una de les diferents opcions.
- Quan ens trobem l'operador de repetició '*':

- Si es sobre un element afegim una aresta sobre l'element i els anteriors de l'element se'ls connecta amb per mitjà d'arestes amb els següents de l'element.
- Si és un conjunt o element compost es creen dos transicions tipus connector per delimitar l'inici i fi del conjunt. Per una banda es connecten entre ells. Per l'altra els elements de dins el conjunt els primers es relacionen amb el connector que marca l'inici i els últims elements amb el connector que marca el final. L'element anterior al conjunt es connecta amb el connector inicial i els elements següents al conjunt es connectaran amb el connector final.

- Quan ens trobem un conjunt on una opció és <E> (que representa una E-transició) els elements anteriors al conjunt es connecten amb els elements següents del conjunt.

3.11.1 Implementació:

La classe encarregada de la càrrega del autòmat a partir d'un fitxer que conté una ER és la classe *automatsPerER.java*. Aquesta classe té implementats tots els mètodes i procediments necessaris per llegir el fitxer, interpretar-lo i crear l'autòmat.

Les expressions estan guardades en fitxers amb extensió ".txt" o ".er". Aquest fitxer és a seleccionat de la mateixa manera que s'ha explicat amb un objecte de la classe JFileChooser.

Sobre aquest fitxer se aplicarà un anàlisis seqüencial amb un parell d'etapes significatives:

Primer farem una fase de lectura de l'expressió regular i de traducció.

3.11.1.1Traducció:

Abans de començar la lectura creem una llista de transicions i afegim com a primer element i amb ID = 0 la transició inicial. Procedim a Llegir tota l'expressió regular i per cada element que trobem guardarem la seva informació lèxica en una transició. En una String auxiliar anem guardant l'ER que llegim del fitxer però per evitar tenir que fer una recerca cada vegada que es vol buscar una transició utilitzem el mateix mètode que quan vam llegir autòmats per fitxer. On abans hi havia un element entre els símbols '<' i '>' i opcionalment podia tenir informació sobre la sortida del transductor, ho canviarem per '<' #posició nova transició '>'. Si el seu contingut era E li assignarem el valor 0 per fer un tractament especial quan ens trobem aquests elements. Un cop s'ha acabat la lectura afegirem a la llista de transicions la transició final. El resultat final de la lectura serà cadena de caràcters anomenada definicioER formada pel contingut de la cadena de caràcters que s'ha acabat de llegir entre parèntesis i se li concatena "<"+ ID transició final+">"

D'aquesta manera la següent ER quedaria traduïda de la següent maenra:

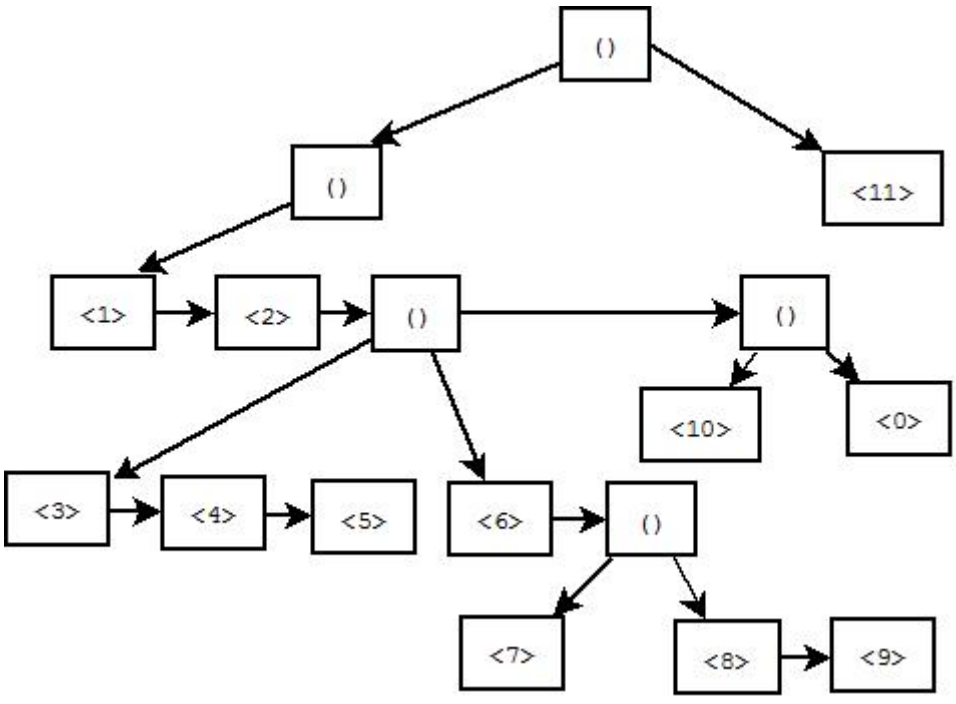
<t1><t2>(<t3><t4>*<t5>+<t6>(<t7>+<t8><t9>))*(<t6>+<E>)

(<1><2>(<3><4>*<5>+<6>(<7>+<8><9>))*(<10>+<0>))<11>

3.11.1.2 Construcció:

La segona fase és la de construcció i s'analitza el contingut de la cadena de caràcters definicioER amb el mètode recursiu **crearAutomat(informació, anteriors)**. El primer paràmetre és una cadena de caràcters amb l' informació a processar. Anteriors és una llista d'enters que conté les ID de les transicions anteriors a l'element que s'ha d'analitzar. En un primer moment el mètode es crida amb *crearAutomat(definicioER,[0])*. D'aquesta manera connectem l'expressió regular amb la transició inicial. Aquest mètode retorna una cadena de caràcters que consisteix en el resultat d'aplicar el mètode sobre el contingut del primer paràmetre.

Aquest mètode recursiu a mesura que va analitzant crea un arbre . En la següent imatge veiem l'arbre resultant de l'anàlisi de l'anterior ER. Els elements d'un mateix nivell se'ls ha separat amb una fletxa mostrant la direcció de l'anàlisi el qual creix en profunditat abans d'analitzar el següent element del mateix nivell.



L'algorisme que implementa el mètode és el següent:

crearAutomat(String definicio,int [] IDanteriros)

{

0. String Element = Llegim primer element

   String definicioSeguent = continuació informació

   1. Si és element simple
      a. Afegim estats entre els elements de la llista IDanteriors i l'identificador d'Element
      b. ultimsElements = [idntificador d'Element];
      c. Comprobar si el següent simbol de definicioSeguent == '*'
         i. Si ho és afegir a la llista transicionsLoop la l'identificador d'Element
         ii. Treure simbol '*' de definicioSeguent
         iii. ultimsElements = ultimsElements:: IDanteriors

      d. Si definicioSeguent != ""
         i. String infoRetorn = crearAutomat(definicioSeguent, ultimsElements).
      e. Return Element+infoRetorn
   2. Si és element compost
      a. String[] SubElements = buscarElements(Element)
      b. Comprobar si el següent simbol de definicioSeguent == '*'
         i. Crear connector inici i fi
         ii. Afegir estat entre inici i fi en les dos direccions
         iii. Afegim estats entre els elements de la llista IDanteriors i l'identificador d'inici
         iv. IDanteriros = [identificador d'inici]
         v. Cada element de llista SubElements concatenar pel final "<"+ID fi+">" d'aquesta manera quan s'analitzin connectaran amb fi.
         vi. EsBucle = true;
      c. Analitzar SubElements
         i. Si algun SubElement == "<0>"
            1. Borrar subElement
            2. Crear connector inici i fi
            3. Afegir aresta entre inici i fi

4. Afegir estats entre els elements de la llista IDanteriors i connector inici
5. IDanteriors = [identificador d'inici]
6. Cada element de llista SubElements concatenar pel final "<"+ID fi+">" d'aquesta manera quan s'analitzin connectaran amb fi

ii. subElement = crearAutomat (SubElement, IDanteriors)

d. Si definicioSeguent != ""
   i. Si el conjunt és bucle
      1. ultimsElements = [id fi]
   ii. Sino Per cada SubElement
      1. ultimsSubElements = buscarUltimsElements(SubElements)
      2. ultimsElements = ultimsElements::ultimsSubElements
      3. infoRetorn = Element
   iii. Si mantenirAntenirAnteriors == true
      1. UltimsElements = ultimsElements::IDanteriors
   iv. String infoRetorn = crearAutomat(definicioSeguent
e. Si el conjunt és un bucle
   i. Return "<"+ID fi+">"+ infoRetorn
f. Sinó
   i. Return Element+ infoRetorn

}

Un cop ja hem analitzat tota l'ER regular en la variable global transicionsLoop tenim les ID's de les transicions que poden repetir-se 0 o més vegades ('*'). Es recorre aquesta llista de final a principi afegint una estat loop sobre la transició i es comprova que els anteriors d'aquell estat estiguin connectats amb els següents d'aquell estat. Si falta algun estat s'afegeix.

A continuació es comenten alguns punts importants de l'algorisme i s'explica els mètodes desconeguts:

0. Llegir el primer element ens retorna un element simple o un element compost, segons quin és l'element de més a l'esquerra de la definició.

Per exemple el primer element de :

(<1><2>(<3><4>*<5>+<6>(<7>+<8><9>))*(<10>+<0>))<11>

És: <1><2>(<3><4>*<5>+<6>(<7>+<8><9>))*(<10>+<0>)

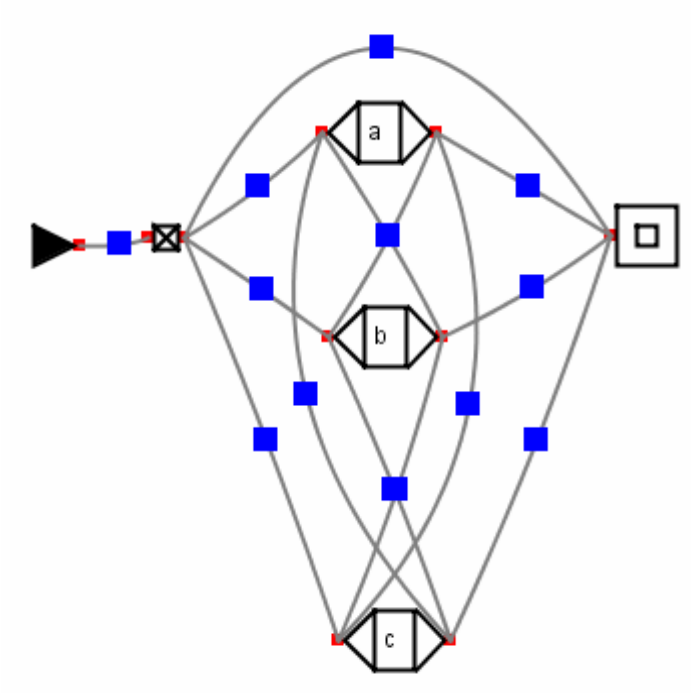
1b. En el cas d'un element simple l'anterior del seu següent és ell mateix, pel cas que el següent símbol sigui '*' (1.c) els anteriors del seu següent són ell mateix i els seus anteriors.

1e. En un cas simple sempre no es modifica la seva informació per tant retornem aquell element concatenat amb el resultat d'aplicar l'algorisme sobre la cadena que ve a continuació d'aquell element.

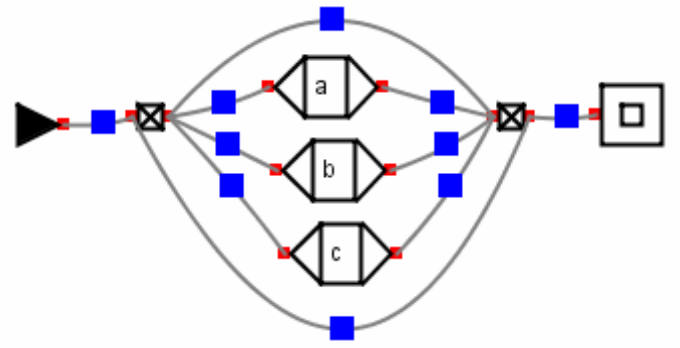
2a. El mètode **buscarElements(String)** aplicat sobre un element compost retorna una llista amb cada subElement.

2b. Si és un bucle sobre un conjunt per exemple en la ER (<a>+<b>+<c>)*

La primera opció seria representar-ho d'aquesta manera:

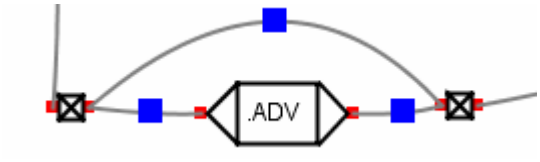


Per simplificar l'autòmat s'ha creat també el connector fi i s'ha concatenat a cada subElement (2.b.v.) per obtenir un resultat més òptim



2.c. quan ens trobem un SubElement “<0>” recordem que vol dir que ens trobem davant d’una E-transició per tant des de la transició actual s’ha de poder accedir a cada una de les diferents opcions o a cap d’elles. Per aquest motiu s’utilitzen dos transicions de tipus connector per crear un “pont” per simular el salt unidireccional.

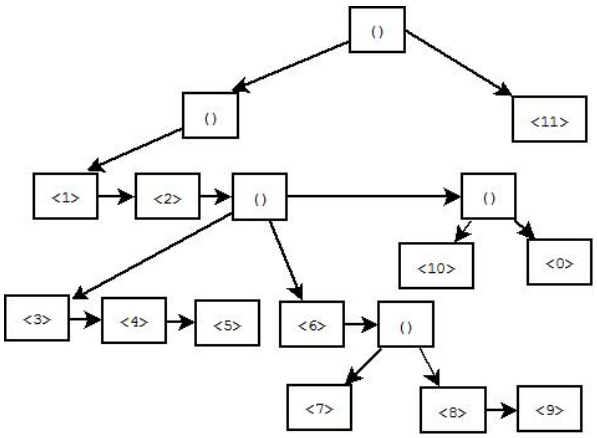
....(<ADV>+<E>).. →



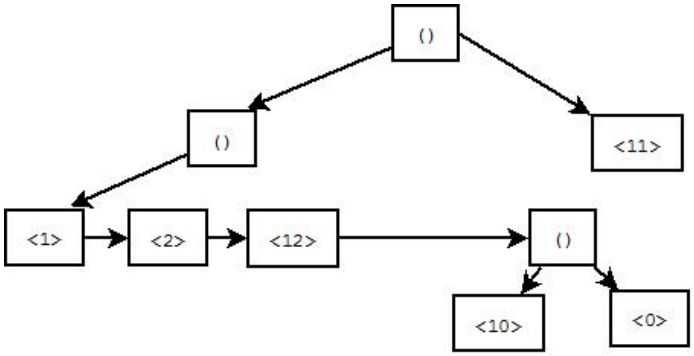
2.c.ii. Es crea l’arbre de cada subElement i s’actualitza el seu valor pel resultat que retorna la seva construcció. Que el mètode *crearAutomat* tingui com a retorn una cadena de caràcters és per actualitzar en aquest moment cada SubElement després de la seva construcció. Aquesta acció la portem a terme per resoldre fàcilment els casos en que un subElement és un conjunt amb repetició de 0 o més vegades (‘*’). Quan es crida el mètode *crearAutomat* sobre un d’aquest subElements el resultat de la crida és una cadena de caràcters de l’estil “<id>” on id és el valor de l’identificador del connector final d’aquell conjunt (2.e.i). En la resta de casos el retorn es correspon amb el SubElement analitzat i no afecta a la resta d’anàlisis (2.f.i).

Aquest detall ens simplifica l’arbre que es construïa en profunditat d’aquesta manera.

Per l’arbre analitzat anteriorment:



Queda simplificat d'aquesta manera:



Un cop s'ha creat el conjunt actual s'han de buscar els últims elements del conjunt per continuar analitzant el següent element. Si no s'apliqués l'actualització de l'arbre al conjunt '*' que tenim a continuació de la transició <2> es passaria per paràmetre com els elements anteriors de l'estat 10 les transicions [9,7,5,2] sense tenir en compte que algun d'aquests pogués ser un bucle. Fent aquesta actualització les transicions anteriors a <10> es correspon únicament a la transició connector final del conjunt <12> simplificant-nos significativament l'autòmat.

2.d.ii.1: el mètode **buscarUltimsElements(SubElement)** si el subElement analitzat és un element simple ens retorna el seu identificador, per altra banda si és un element compost es crida recursivament fins trobar els extrems. Aquest mètode també té en compte els conjunts amb E-transicions buscant els ultims elements de l'element anterior.

Si tenim de l'element compost:

(<a0><b0>+(<a1>+<b1>+<c1>)+<a2><b2>(<a3><b3>+<c3><d3>+<E>))

El resultat d'aplicar buscarUltimsElements( ) és:

[#id <b3>, #id <d3>, #id <b2>, #id<c1>,#id <b1>,#id <a1>,#id <b0>]

3.12 Guardar autòmats com ER

De les diferents opcions de guardar i llegir autòmats aquesta és la que es va decidir que tenia menys valor perquè ja hi ha altres programes que tradueixen de format autòmat a ER. Tot i així com que la planificació del projecte va ser l'adequada s'ha disposat de temps just per poder realitzar l' implementació del guardat d'autòmats lineals en forma d'ER.

3.12.1 Requisits per poder guardar l'autòmat:

S'ha de tenir com una transició inicial perquè només es guarda les transicions i estats de l'autòmat accessibles des de la transició inicial.

En cas que no hi hagi la transició inicial es mostra per pantalla una notificació del error.

3.12.2 Implementació:

La classe encarregada de guardar del autòmat com ER en un fitxer és la classe *automatsAER.java*. Aquesta classe té implementats tots els mètodes i procediments necessaris per llegir el fitxer, interpretar-lo i crear l'autòmat.

La ER s'aconsegueix amb cinc fases: creació l'ER , simplificació, substitució, eliminació d'elements redundants i descodificació.

3.12.3 Fase de creació de l'expressió regular:

Per crear l'ER inicialment es recorre recursivament l'autòmat en profunditat guardant les transicions que es troben en l'exploració de l'autòmat. S'ha implementat el mètode **crearER(id, anteriors)**. Aquest mètode retorna una cadena de caràcters que és l'ER. El primer paràmetre és l'identificador del a transició que avalua. El segon paràmetre són les transicions que ja ha visitat per detectar recursivitats i no caure en un bucle infinit.

Aquest mètode busca els estats dels quals la transició origen té la mateixa id que la transició que s'està analitzant (primer paràmetre) i es guarda les ID's de les transicions destí en una llista.

En cas que la transició actual sigui una transició estàndard o una transició de tipus connector la cadena de caràcters que retornem li afegeim "<"+id+">".

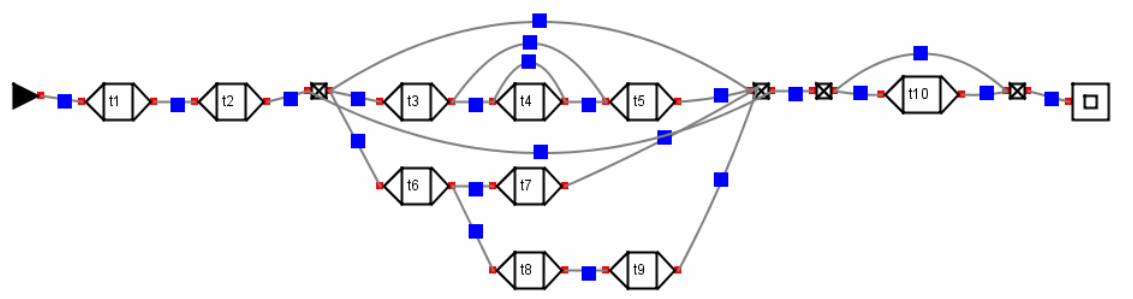
A continuació per cada element de la llista en cas que no hagi estat ja analitzat es fa una crida recursiva a aquest mètode, el seu retorn es concatenarà amb la cadena de caràcters del retorn. Si es té més d'un element en la llista el seu retorn es guarda entre parèntesis i separat per l'operador suma.

Si ens trobem un element que ja ha estat analitzat ens trobem davant d'un bucle. En aquest cas guardem l'identificador de l'inici del loop amb un asterisc ("<?"+id+">") per diferenciar del cas base. D'aquesta manera tenim el loop acotat i amb la referència amb l'identificador d' on comença.

Si l'element que ens trobem és la transició final s'afegeix el token "<E>"que s'utlitzarà pel tractament dels casos de transicions opcionals.

Per exemple el següent autòmat que equival l'ER:

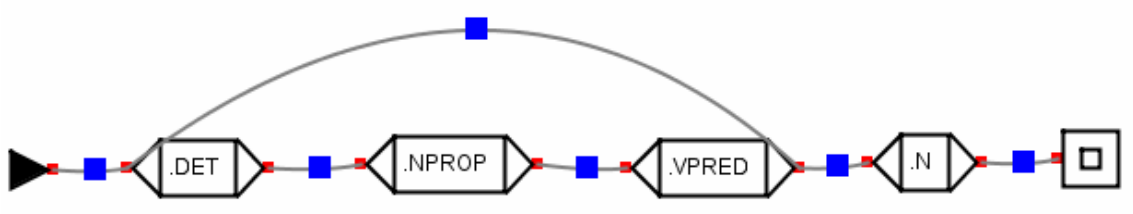
<t1><t2>(<t3><t4>*<t5>+<t6>(<t7>+<t8><t9>))*(<t10>+<E>)



L'ER resultant d'aplicar aquesta primera fase és la següent:

ER1:
<1><2><12>(<3>(<4><4>*<5><13>((<10>+<E>))+<5><13>((<10>+<E>)))+<6>(<7><13>((<10>+<E>))+<8><9><13>((<10>+<E>)))+<13>((<10>+<E>)))

Pel cas de (<DET><NPROP><VPRED>)(<DET><NPROP><VPRED>)*<N> :



L'ER resultant d'aplicar aquesta primera fase és la següent:

ER2: <1><2><3>(<4>+<*1>)

3.12.4 Simplificació:

El procés de simplificació correspon a treure el màxim factor comú possible de l'ER.

Aquest procés es porta a terme amb el mètode **simplificar(String)** el qual va recorrent l'ER i quan es troba un conjunt crida el mètode **PosarComu(llista SubElements).**

El mètode PosarComu calcula els últims elements de cada subElement. Els subElements que tenen la mateixa terminació els torna a posar en comú sense el seu últim element el qual queda aïllat. Aquest mètode també simplifica si tenim una repetició i una E-transició en un mateix conjunt. Als elements aïllats se'ls aplica el mètode simplificar sobre ells per simplificar tota l'ER d'una forma recursiva.

El mètode simplificar simplifica els elements en ordre d'aparició. Elements que en un primer moment no poden ser posats en comú al ser simplificats si que ho poden ser. Per aquest motiu l'ER inicial és simplificada tantes vegades com és necessari fins que el resultat de dos iteracions és identic.

Exemple d'aplicació del mètode posarComu(llista subElements):

Amb els subelements:
[<3>(<4><4>*+<E>)<5><13>(<10>+<E>), <6>(<7>+<8><9>)<13>(<10>+<E>)]

El resultat de posar-ho en comú és:

(<3>(<4><4>*+<E>)<5>+<6>(<7>+<8><9>))<13>(<10>+<E>)

Exemple d'aplicació del mètode simplificar(String):

ER1:
<1><2><12>(<3>(<4><4>*<5><13>((<10>+<E>))+<5><13>((<10>+<E>)))+<6>(<7><13>((<10>+<E>))+<8><9><13>((<10>+<E>)))+<13>((<10>+<E>)))

Primera iteració:

<1><2><12>(<3>(<4><4>*+<E>)<5><13>(<10>+<E>)<E>+<6>(<7>+<8><9>)<13>(<10>+<E>)<E>)

Segona iteració:

<1><2><12>(<3>(<4><4>*+<E>)<5>+<6>(<7>+<8><9>))<13>(<10>+<E>)<E>

3.12.5 Substitució:

La fase de substitució s'encarrega de substituir els connectors per bucles i d'eliminar els elements redundants.

La fase de substitució consta de dos parts: creació de bucles, tractament de connectors

3.12.5.1 Creació de bucles:

Inicialment al crear l'ER inicial quan ens trobàvem un bucle afegíem a la ER l'element <*id> per fer el seu tractament en aquest apartat. Per diferenciar una transició normal d'un bucle utilitzem el símbol '*', el valor de id correspon al identificador de la transició des d' on comença el bucle. Es recorre la cadena de caràcters buscant aquests elements i copiem la subcadena de caràcters corresponent entre l'element <*id> i <id> aquest últim inclòs. Aquesta subcadena conté tots els elements que es repeteixen dins el bucle. Li afegim els símbols "("+subcadena+")*" per guardar la repetició del conjunt.

S'ha de tenir en compte que els element <*id> poden formar part d'un conjunt. En aquests casos la subcadena és limitada per l'inici del conjunt on està inclòs <*id>.

Exemple de creació de bucles:

ER2: <1><2><3>(<4><E>+<*1>)

<*1> marca un bucle des de la transició <1> fins ella.

El resultat d'aplicar la substitució equival a:

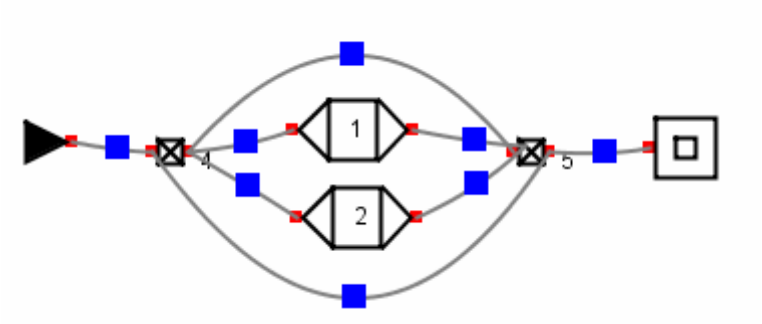
<1><2><3>(<1><2><3>)*(<4><E>)

3.12.5.2 Tractament de connectors:

Al crear inicialment l'ER regular quan s'analitzava un connector aquest es guardava per dos motius. Per una banda s'han de guardar per la part de creació de bucles per resoldre els casos que una transició crea un bucle connectant-se amb un connector.

L'altre motiu perquè els guardem és per crear la repetició de conjunts. La repetició de conjunts ve determinada per dos connectors connectats entre ells amb transicions que va d'un a l'altre.

La manera de representar (<t1>+<t2>)* és:



El resultat de crear l'ER i simplificar-la és: <4>(<1>+<2>)<5>

Les transicions <4> i <5> són els connectors que delimiten el conjunt que es repeteix. El mètode **actualitzarConnectors(String)** busca els connectors que representen aquesta estructura. Quan els troba seleccionen ER que hi ha entre ells dos i els hi afegeix el símbol de repetició ('*')

Resultat d'aplicar *actualitzarConnectors(String):* (<1>+<2>)*

Al convertir aquell conjunt en una repetició si un dels subElements és la E-transició (<E>) és redundant. Es fa una crida al mètode **treureERedundant(String**) que s'encarrega de retornar la cadena sense aquell subElement en cas de contenir-lo.

El resultat del tractament de ER1 és:

<1><2>(<3>(<4><4>*+<E>)<5>+<6>(<7>+<8><9>))*<14>(<10>+<E>)<15>

3.12.6 Eliminació d'elements Redundants:

Fins aquest punt al crear l'ER al trobar-nos amb una repetició l' informació a priori que sabíem era que es repetia 1 o més vegades. Aquestes altures de la simplificació l'ER ja és compacte i és simplifiquen els conjunts que representen la repetició d'un element cap o més vegades:

El mètode **buscarElementsSimplificar(Cadena)** busca la següent subcadena dins la cadena de caràcters:(<X><X>*+<E>). Les subcadenes trobades les redueix a la seva mínima expressió: <X>*

El resultat del tractament de ER1 és:

<1><2>(<3><4>*<5>+<6>(<7>+<8><9>))*<14>(<10>+<E>)<15>

3.12.7 Descodificació:

Aquesta fase rep l'ER ja construïda i simplificada per retornar-la traduïda per les transicions que representa. Per cada element que parseja <id> busca la transició que té associada aquella id. En cas que sigui un connector la seva utilitat era per guardar la referència per possibles bucles. L'informació que aporta és nul·la per tant no es substitueix per una cadena buida.

Quan la transició en qüestió és de tipus estàndard l'element se'l substitueix per la seva informació lèxica. El cas de les E-transició és invariant.

El resultat d'aplicar aquesta fase a la següent ER corresponent a

ER2: <1><2><3>(<1><2><3>)*<4>

Correspon a: <DET><NPROP><VPRED>(<DET><NPROP><VPRED>)*<N>

3.13 Opcions complementaries:

Una bona aplicació es caracteritza per portar a terme correctament l'objectiu pel qual ha estat implementada. La nostra aplicació permet crear i editar gramàtiques electròniques així com guardar l' informació i carregar-la . Els objectius del projecte ja estan assolits però s'han implementat unes quantes funcionalitats complementaries amb la finalitat de proporcionar a l'usuari final una eina més complerta.

S'ha analitzat les possibles optimitzacions que es podien implementar i el temps disponible per fer la selecció de les opcions que s'han afegit a l'aplicació.

3.13.1 Desfer l'últim canvi realitzat.

L'usuari que treballa amb l'aplicació pot realitzar una acció amb un resultat que no és el desitjat. S'ha considerat de gran utilitat oferir a l'usuari la possibilitat de desfer l'últim canvi realitzat. Aquesta aplicació per fer moltes operacions així com afegir i esborrar estats i transicions, modificar la seva informació lèxica o etiqueta, canviar de posició, etc.

Realment el que s'ha considerat primordial ha estat l'aspecte sobre el tema de poder tornar endarrere en el cas que l'usuari elimini un element per equivocació. El pitjor escenari seria el cas d'eliminar per equivocació una transició la qual té connectades varis estats, l'usuari a part d'afegir de nou la transició hauria de recordar-se de tots els estats que la connectaven amb altres transicions amb la conseqüència de possiblement oblidar-se algun. Per aquest motiu l'opció de desfer l'últim canvi només tracta el context d'inserció i eliminació d'elements i desfer l'últim canvi que s'ha desfet.

S'ha implementat la classe *Desfer.java* per encarregar-se d'emmagatzemar l'informació necessària i encarregar-se de fer els canvis sobre l'escena. Al constructor d'aquesta classe se li passa per paràmetre la referència a l'objecte ZonaDibuix (escena) sobre el qual treballarà.

Aquesta classe utilitza una variable entera anomenada "opció" que s'utilitzarà per determinar quina és l'opció que s'haurà de portar a terme en el cas que l'usuari vulgui desfer l'últim canvi. Aquesta variable pot prendre cinc valors diferents:

0 = no fer cap acció.
1 = Afegir transició i estats.

2 = Afegir estats.
3 = Eliminar transició i estats.
4 = Eliminar estats.

Per guardar l' informació necessària la classe té les següents variables internes: un objecte Caixa on es guarda l'última transició eliminada. Una llista d'Arestes on es guarden els estats eliminats en l'última acció, s'utilitza una llista perquè el cas d'eliminar una transició que té varis estats s'han de guardar tots. També tenim una variable entera que ens diu el nombre d'arestes que s'han afegit (NNousEstats). Com que els elements s'afegeixen al final de la llista només hem de saber el nombre d'estats que s'ha afegit per eliminar tots aquells elements.

La classe conté un únic mètode que s'encarrega d'escollir quina acció portar a terme segons el valor de la variable interna "opcio".

Pel cas "Afegir transició i estats" afegeix la transició i estats que tenim guardats en les variables internes de la classe a les seves respectives llistes de l'objecte zonaDibuix. Canviem el valor d'opció per 3 i NNousEstats se li assigna el valor d'estats que s'han afegit.

La resta de casos tenen un tractament semblant amb el propòsit d'afegir o eliminar l'últim estat eliminat o d'eliminar la transició. En el cas d'eliminar la transició només s'ha d'eliminar l'últim element de la llista de transicions (abans s'ha de copiar en la variable interna de la classe per poder tornar endarrere).

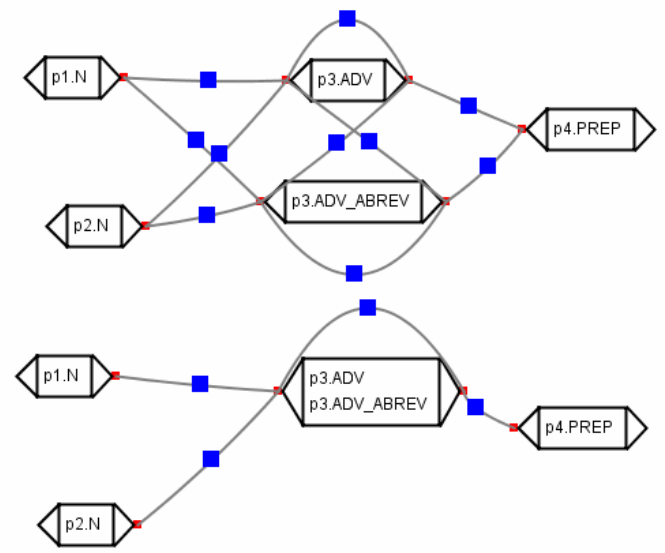
Cada objecte ZonaDibuix té una variable interna que és un objecte de la classe *Desfer.java*. Cada vegada que s'afegeix o s'elimina un element de l'escena es modifica el valor de la variable "opció". En el cas d'eliminar un estat o una transició aquests es guarden en l'objecte Desfer per tenir-lo preparat per desfer l'última acció en qualsevol.

S'ha afegit aquesta opció a la barra de menú perquè l'usuari la trobi fàcilment. Quan l'ítem és seleccionat es crida el mètode **desferCanvi()** de l'escena que està seleccionada el qual s'encarrega d'avisar a l'objecte DesferCanvi.

Transicions compostes

Fins aquest moment les transicions que permetíem representar eren simples doncs estaven definides per una única informació lèxica. Per anar d'un estat a un altre ens podíem trobar amb varies transicions que representaven la mateixa paraula o expressió però que tenien una informació lèxica diferent per suplir tots els casos possibles. L'objectiu de les transicions compostes es ajuntar totes aquestes transicions amb contingut específic en una transició més general. Les transicions compostes són una manera de simplificar l'autòmat que s'està editant. Per ser més precisos una

transició composta és una transició que dins seu conté varies informacions lèxiques resultants de fusionar el contingut de dos o més transicions en una mateixa transició. D'aquesta manera se'ns redueix enormement el nombre d'estats doncs al fusionar les transicions la transició composta resultant té l' intersecció dels seus estats. Dos estats es simplificaran si els dos tenen com origen la seva respectiva transició però el seu destí apunta a una mateixa transició, i a l'inversa.



3.13.2.1 Crear transicions compostes:

Primer de tot s'han hagut de fer canvis en la classe Caixa.java on hem hagut de canviar la cadena de caràcters que contenia l' informació lèxica, per una llista de cadenes de caràcters per poder emmagatzemar totes les informacions que pugui tenir la transició.

També hem creat el mètode **CheckAndAddSubtransicio (Caixa transicioSimple)**. Aquest mètode de la classe rep per paràmetre l'objecte tipus Caixa amb el qual es vol comprovar si es correcte permetre la fusió entre les dos transicions i retorna un booleà amb el valor true en cas positiu.

Perquè una fusió entre dos transicions sigui vàlida s'han de complir una sèrie de condicions: primer es comprova l' informació del camp forma canònica on en cas que les dos transicions tinguin un valor definit aquest ha de coincidir. En cas que només un de les dos transicions estigui instanciada se li assigna aquest valor l'altra transició.

L'altre transició que s'ha de duu a terme es comprovar que la sortida dels transductors és estricament igual. En aquest cas o les dos transicions no tenen el camp definit o sinó aquest ha de coincidir.

Aquest mètode també s'encarrega d'afegir l' informació de la transició passada per paràmetre a la transició des d' on es produeix la crida del mètode, en el cas que sigui vàlid fer la fusió.

Per poder fusionar transicions s'ha afegit un nou ítem a la barra d'eines per portar a terme aquesta operació. Aquest ítem treballa com la resta d'elements de la barra d'eines. Quan es seleccionat li assigna un nou valor a la variable "opció" de l'objecte ZonaDibuix de l'escena actual.

Amb aquesta nova opció quan es faci clic en l'escena buscarà si en aquella posició si troba una transició, en cas afirmatiu s'assigna el valor true al camp "seleccionat" de la transició perquè aquesta es vegi d'un color diferent i l'usuari observi fàcilment la seva selecció.

També es guarda la posició de la transició dins la seva llista en la variable index_juntar quan es selecciona el primer element i en la variable index_llista quan es fa la segona selecció.

En cas que s'hagi fet clic sobre el mateix element o les dos transicions no són de tipus "normal" el que fem es desseleccionar-les. En altre cas cridem el mètode *CheckAndAddSubtransicio(Caixa)* des de l'última transició seleccionada. Si el resultat és true eliminem l'altra transició i desseleccionem l'element. En cas contrari simplement desseleccionem les dos transicions.

3.13.2.2 Tractament dels estats al fusionar dos transicions simples en una transició composta:

Quan es va decidir que fer amb els estats que tenien relació amb alguna de les transicions involucrades en la fusió es va arribar a dos solucions. Una solució era no permetre la fusió entre dos transicions a no ser que com a mínim una dels dos no tingués cap estat. L'altre solució era permetre fusions de transicions que tenen varis estats.

Per raons de programació hauria estat molt més fàcil d'implementar la primera opció, però es va pensar cara l'usuari final i es va decidir la segona opció. La raó d'aquesta decisió va ser per millorar la comoditat de treball de l'usuari. D'aquesta manera podrà fusionar transicions tan bon punt són acabades de crear com si s'ha estat treballant amb elles (i poden tenir varis estats que en depenen) i ara s'ha donat compte que les vol ajuntar.

Per implementar això s'ha implementat el mètode **canviarEstatsdeTransicio (oldID, newID, xIniciTransicio, xFiTransicio, y)**. Igual que anteriorment havíem ajuntat tota

l'informació en una mateixa transició, en aquest cas totes les transicions de la primera transició seleccionada ("Old")se li afegeixen a la segona transició ("new"). Els estats que compartien entre elles dos són eliminats així com els estats que si es canvia algun dels extrems es converteix en un estat repetit.

Per fer aquestes operacions recorrem amb un bucle la llista llistaArestes de final a principi per no tenir problemes d'indexació al eliminar estats per alguna de les raons anteriors. Per cada estat mirem si algun dels extrems té la ID de la transició "old". En cas de coincidència es comprova que l'altre extrem de la transició no sigui de la transició "new". També comprovem amb el mètode **checkArestaRepetida(IDinici, IDfi)** si actualitzem l'extrem que li tocaria si estaríem creant una aresta repetida. Aquest mètode recorre tota la llista llistaArestes buscant la coincidència amb una aresta que tingui les mateixes transicions per extrems. En cas de passar les dos comprovacions el valor de la ID d'inici o de fi (segons l'extrem involucrat) se l'assigna el valor de la ID "new". Amb els paràmetres xIniciTransicio, xFiTransicio i y situem l'extrem al seu lloc, i a continuació actualitzem les posicions del rectangle de control i menú de l'estat.

En cas de no passar alguna de les comprovacions es procedeix a eliminar l'estat de la llista però abans amb els mètodes **borrarArestaInici(aresta.getIDInici())** i **borrarArestaInici(aresta.getIDInici())** ja utilitzats en l'eliminació d'estats i transicions, es decrementa el comptador d'estats dels extrems de la transició que es troba a l'altre extrem de l'estat que s'està a punt d'esborrar.

En cas que l'aresta no tingui en els extrems la transició "old" es mira si té la transició "new" en aquest cas actualitzem la posició d'inici i fi de la transició per situar correctament els extrems de l'estat ja que ara la transició és més llarga.

3.13.2.3 Separar transicions composta:

Al decidir que es dona a l'usuari l'opció de ajuntar transicions en una amb més informació, també es va decidir que seria interessant permetre fer l'operació complementaria. L'opció de separar transicions s'accedeix fent clic amb el botó dret sobre una transició. Amb aquesta acció apareix el menú emergent que vam explicar al qual li hem afegit la nova opció.

Quan el oient d'Events de la classe PopupListener.java captura l'event llençat per la selecció d'aquesta opció, fa les següents operacions: primer guarda la caixa seleccionada en una variable local i comprova si la llista d'informació té més d'un element. Si n'és el cas es recorre la llista creant un objecte de tipus Caixa per cada informació de la llista menys el primer que s'aprofita la transició existent. Perquè no es superposin totes les noves transicions guardem les coordenades x i y de la transició

original i per cada transició nova li assignem aquelles coordenades sumades a una constant. Aquestes transicions que s'acaben de crear les afegim a la llista on estan la resta de transicions .

Un cop ja hem creat les noves transicions es creen nous estats per proporcionar a cada nova transició la mateixa connectivitat que tenien quan estaven agrupades. Coneixem el número de noves transicions, ja que correspon al nombre d'elements que tenia la llista d'informació de la transició menys 1 (NnousEstats). El mètode que crea aquests nous estats és el següent: **AfegirArestesSeparar(ID, NnousEstats)** on ID és el valor de l'identificador de la transició inicial.

Aquest mètode recórrer la llista d'Estats buscant transicions que en algun dels extrems es trobi la transició amb la que estem treballant. Per cada estat que trobem recorrem un bucle de 0 a N-1 iteracions on N és el número de nous estats (segons el valor del segon paràmetre de la funció). En cada iteració seleccionem la transició que tenim en la posició que concorda amb el valor de la iteració començant pel final de la llista de transicions. Aquestes transicions corresponen a les transicions que havíem creat noves i les hi assignem a l'extrem del nou estat en el qual l'estat original s'hi trobava la transició inicial. En tots els casos hem d'incrementar els comptadors d'estats de les transicions que es troben als extrems dels nous estats.

Per evitar fer una iteració innecessària en l' inserció d'estats recordem que de la transició original havíem buidat la llista d'informació i li havíem deixat únicament un element. Com que la transició ha canviat de mida s'ha de situar els seus estats al nou centre de la transició. Per fer aquesta operació utilitzem de la classe ZonaDibuix el mètode
actualitzarExtremsEstat(IDtransicio,xIniciTransicio,xFiTransicio,yCentreTransicio).
Aquest mètode busca els estats que tenen en algun extrem la transició i actualitza les coordenades del extrem a la seva nova posició.

3.13.2.4 Modificacions en la finestra de Propietats de la transició:

En els casos que la llista d'informació de la transició només té un element no hi ha cap canvi respecte el que ja s'ha escrit, la cadena de caràcters que s'analitza per omplir els camps de la finestra emergent és l'únic element de la llista.

El dilema era com mostrar els diferents elements d'una transició composta en el panell d'actualització. En un darrer moment es va decidir que la manera més òptima era preguntant a l'usuari amb una finestra emergent quin element vol modificar.

Per mostrar la finestra emergent una vegada fem ús de la classe JOptionPane i en aquest cas el panell que mostrem simplement conté un Label i un JComboBox amb N ítems on N és la mida de la llista d'informació.

Un cop feta la selecció del JComboBox s'utilitza el valor de l'índex de l'opció seleccionada per determinar quina element de la llista d'informació de la transició és el que es separa per omplir els camps de la finestra emergent que mostrem.

En cas de que l'usuari realitzi canvis tenim present que la forma i la sortida del transductor són comunes per totes els elements de la transició per tant si es produeixen canvis també se'ls assigna el nou valor d'aquests camps a la resta d'elements.

3.13.2.5 Permetre transicions heterogènies:

Nosaltres hem posat la restricció que si dos transicions no tenen la mateixa forma canònica i la mateixa sortida de transductor no es poden fusionar. S'ha escollit aquesta restricció perquè quan es fusionen dos transicions és perquè són la mateixa paraula (mateixa forma canònica) però la resta d'informació és diferent i es vol simplificar l'autòmat.

Tot i així s'ha implementat l'opció de permetre transicions heterogènies (forma canònica diferent) perquè sigui l'usuari final qui tingui el criteri de fusionar transicions coherentment.

Una variable booleana determina el grau si fem una fusió estricte és a dir que no permet transicions heterogènies o no. En cas que es permeti és salta la part de codi referent a la comprovació de la compatibilitat del a forma canònica.

S'ha afegit un nou ítem a la barra de menú anomenat "Transiciones heterogeneas: Des/Activado". Cada vegada que es selecciona és nega el valor de la variable booleana que determina el tipus de fusió.

En cas que estigui aquesta opció activada, si canviem el camp forma canònica d'un element d'una transició complexa la forma canònica de la resta d'elements de la transició no són actualitzats.

3.13.3 Guardar autòmat com imatge

L'usuari pot voler guardar l'autòmat que ha dissenyat en un format visual per poder-lo afegir en presentacions o informes. Per aquest motiu s'ha implementat l'opció que permet a l'usuari guardar l'autòmat com una imatge.

A l'usuari se li mostrarà la finestra emergent perquè determini el nom del fitxer i la ubicació on el vol guardar. A continuació se li mostrarà una altra finestra perquè esculli en quin format vol guardar l' imatge. L'usuari podrà escollir entre els formats de codificació d'imatges més comuns: bmp, gif, jpg, jpeg i png.

S'ha afegit el mètode **guardarImatge(fitxer , format)** a la classe ZonaDibuix. Aquest mètode s'encarrega de guardar la representació de l'escena en el fitxer que havia determinat l'usuari en el format seleccionat.

El procés es ben senzill, es crea una variable local de tipus BufferedImage :

BufferedImage imatge = new BufferedImage(width,height,
BufferedImage.TYPE_INT_RGB);

El width i el height venen determinats per l'àrea de l'escena, el tercer paràmetre indica com volem que sigui la imatge, en el nostre cas utilitzem 8 bits en RGB.

Aquest objecte es converteix en un Graphic2D

Graphics2D g = (Graphics2D)imatge.getGraphics();

Sobre el qual es representa l'escena recreant la mateixa estructura i procediment implementats en el mètode paint. Es a dir representem l'escena però no sobre l'objecte Graphics que utilitzem per mostrar per pantalla sinó sobre el que hem creat.

Per últim guardem la imatge en el fitxer utilitzant el mètode d'escriptura de la classe *ImatgeIO : ImageIO.write(g, format, fitxer);*

3.13.4 Escollir colors dels elements de l'escena:

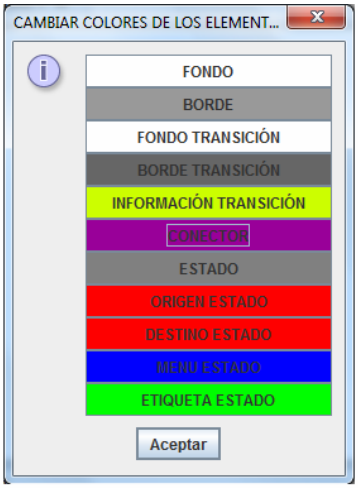
S'ha decidit oferir a l'usuari la possibilitat de canviar el colors dels elements de l'escena com el fons, el color dels estats i transicions, el text, etc. Perquè puguin personalitzar els seus autòmats de la manera que més els agradi.

S'ha creat una classe anomenada *Format.java* que conté com a variables internes diferents colors pels diferents elements de l'escena. S'ha afegit un objecte d'aquesta classe com a variable interna de la classe ZonaDibuix.java.

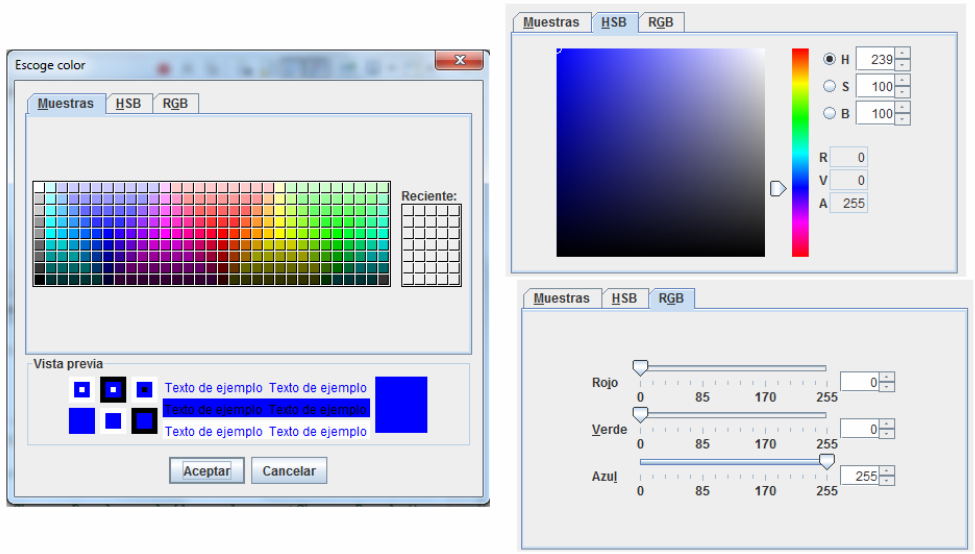
En el mètode paint abans de dibuixar cada element es busca el color que el representa de l'objecte Format. D'aquesta manera cada escena que tenim oberta en pestanyes diferents té els seus propis colors.

L'avantatge d'haver centralitzat tots els colors en una classe és la comoditat per fer modificacions. Per canviar un color d'una escena només hem de modificar el valor de les variables internes de l'objecte Format de l'escena.

Hem implementat la classe *CanviarColors.java* que s'encarregarà de mostrar les diferents opcions de canvi de colors i aplicar els canvis. Se li passa al constructor l'objecte ZonaDibuix sobre el qual treballarà.
Aquesta classe s'encarrega de mostrar un panell amb tants botons com diferents elements de l'escena que se'ls pot canviar el color. El color de fons de cada botó és el color que té assignat aquell element.



L'API de Java proporciona gran quantitat de classes interessants amb mètodes molt útils. En aquesta situació disposem de la classe JColorChooser que ens ofereix un magnífic panell de selecció de colors.



El tractament amb aquesta classe és molt intuïtiu. Només s'ha de controlar que l'opció seleccionada sigui la d'Acceptar i en aquell cas el mètode **getColor()** retorna el color seleccionat per l'usuari. L'únic que hem d'implementar nosaltres és un control de quin botó ha seleccionat l'usuari i assignar el color desitjat a la variable corresponent de la classe Format.

L'avantatge d'haver passat per paràmetre l'objecte ZonaDibuix al constructor és que després de canviar cada color actualitzem l'escena actual d'aquesta manera es van visualitzant els canvis a mesura que es van fent enlloc d'haver d'esperar fins haver acabat.

El panell que proporciona la classe JColorChooser és molt complert. S'ha trobat innecessari i poc estètic la part inferior del panell referent a vista prèvia perquè quan seleccioni un color l'usuari ja observarà els canvis. També s'ha considerat redundant la tercera pestanya (RGB) perquè l'usuari no es posarà a escollir el color de les transicions segons el seu valor RGB sinó que escollirà millor un color dels altres dos panells. El panell de vista prèvia i la pestanya RGB s'han eliminat perquè es vol oferir a l'usuari un panell de selecció de colors útil, senzill i còmode. Si donés el cas que l'usuari volgués seleccionar uns colors RGB determinats tampoc és indispensable la pestanya 3 perquè pot introduir els valors en la pestanya HSB.

La classe JColorChooser és molt interessant i molt flexible perquè així com podem treure-li algun panell si no ens interessa també permet afegir-ne d'altres creats per nosaltres.

3.13.5 Afegir etiquetes als estats:

Les transicions es diferencien entre elles perquè cada una se li defineix una informació lèxica determinada. Per altra banda els estats són tots iguals i només se'ls reconeix per la connexió entre transicions que realitzen. S'ha afegit l'opció d'assignar una etiqueta a cada estat.

La funcionalitat d'aquestes etiquetes és simplement perquè l'usuari s'orienti mentre està treballant. Amb les etiquetes pot guardar informació que l'ajudi a l'hora de crear l'autòmat com una mena de recordatoris o notes.

Fent clic amb el botó dret sobre el menú de l'estat s'ha afegit una nova opció al seu menú per instanciar i modificar el valor de l'etiqueta.

3.13.6 Manual d'usuari

S'ha afegit a l'aplicació un manual d'usuari per resoldre els possibles dubtes de funcionament de l'aplicació que puguin tenir els usuaris.

Després de varies idees s'ha decidit que la manera més elegant i ràpida de crear un element que contingui aquesta informació seria a través d'un document html. Els documents web són fàcils de crear amb molta flexibilitat en quan a l'organització del contingut.

Un cop creat el document html s'han valorat diferents maneres de mostrar-lo al usuari. Per utilitzar altres elements de l'ordinador Java ens proporciona la classe Runtime. Aquest mètode depèn de la plataforma sobre la que treballa el programa i els navegadors que l'usuari té instal·lats. S'ha decidit fer una implementació diferent per assegurar-nos que cap usuari no es quedi sense poder accedir al manual d'usuari.

S'ha utilitzat la classe JEditorPane que és un editor de text capaç d'editar varis tipus de contingut. Aquest component utilitza l' implementació d'un objecte EditorKit per treballar correctament amb diferents tipus de contingut. Els diferents tipus de contingut que coneixen per defecte són text pla, documents en format rtf i html.

Amb JEditorPane combinat amb EditorKit amb poques línees de codi tenim l'aplicació preparada per mostrar per pantalla un panell amb el nostre document html.

El JEditorPane per defecte permet editar el document que obre però en aquest cas no ens interessa per tant en deshabilitem l'opció.

Per carregar el document html s'utlitza el mètode **setPage(String url)**. El manual d'usuari és relativament breu però tot i així ocupa més d'una pàgina. L'editor no proporciona una barra de desplaçament per tant no es veu per pantalla tot el contingut del document a no ser que posem l'objecte JEditorPane dins un JScrollPane el qual s'encarrega d'afegir i gestionar les barres de desplaçament.

3.13.7 Conversió a Applet

Un dels objectius secundaris del projecte demanava l'opció de convertir l'aplicació en un applet de Java perquè es pogués afegir a un document web. Com que el nostre codi estava molt ben organitzat fer aquesta implementació ha estat realment senzilla utilitzant una classe que fa d'interfície.

Per crear un applet de Java s'ha de crear una classe que sigui derivada de la classe Applet. Aquesta classe ha tenir com a mínim el mètode **init()**. Aquest mètode realitza una crida al constructor de la classe Main per obrir el programa en un frame.

Aquesta és la part referent a la codificació. S'ha d'exportar el nostre projecte com un arxiu JAR. Per afegir l' Applet a una pàgina web s'ha d'afegir el següent tag al body del document html.

```
<APPLET
 CODEBASE = "."
  CODE     = "GeneradorApplet.class"
  ARCHIVE =  "applet24.jar"
  NAME     = "Generador Gramaticas electronicas"
  WIDTH    = 400
  HEIGHT   = 300
  HSPACE   = 0
  VSPACE   = 0
  ALIGN    = middle
>
</APPLET>
```

Els diferents atributs del tag <APPLET> tenen el següent significat:

- CodeBase: Determina la direcció des d' on es pot descarregar la classe de Java que va a carregar l'applet, la URL de la classe. Si no s'especifica és l' ubicació actual de la pàgina web.
- Code: Fa referència a la classe de l'arxiu JAR que és derivada de la classe Applet i que té el mètode *init()*. La direcció és relativa al CodeBase.
- Archive: És la llista de classes separades per comes que han de ser carregades a la caché per l'usuari local abans de poder executar-se.
- Name: Estableix un nom únic per l' Applet
- Width i height: determinen la mida en píxels de l'amplada i l'altura.
- HSpace i WSpace: estableix els marges laterals, superiors i inferiors en píxels.
- Align: Determina l'alineació respecte els altres elements.

Quan la pàgina web es carregada, s'executa l' Applet i apareix la nostra aplicació com una finestra emergent. El problema de l' Applet és que si es tanca el navegador l'Applet es tanca brutament sense donar la possibilitat de guardar els canvis. Per aquest motiu l'usuari ha d'anar en compte si treballa via navegador web i es recomana descarregar l'aplicació.

Capítol 4
Conclusions

El resultat final d'aquest projecte és una eina gràfica multiplataforma de creació i edició de gramàtiques electròniques per representar el Llenguatge Natural.

L'eina fusiona comoditat i utilitat amb la perspectiva de satisfer a tots els usuaris. L' interfície gràfica està organitzada amb l'estructura més òptima perquè l'usuari es trobi còmode treballant-hi. S'han implementat les millors opcions possibles per proporcionar a l'usuari fluïdesa i comoditat a l'hora de crear autòmats. També s'han assolit els objectius d'emmagatzemament dels autòmats creats en Format Autòmat (FA) i Expressió Regular (ER) així com la carrega de documents en aquest format.

4.1 Destí de l'aplicació:

És una eina destinada a ser utilitzada per lingüistes per tant se li ha donat especial importància a crear una interfície gràfica de fàcil ús per gent amb pocs coneixements informàtics.

És una eina transparent per l'usuari ja que el programa en tot moment els guia en les diferents tasques de generació de l'autòmat. S'ha utilitzat una variant de la representació estàndard d'autòmats per un format de representació més visual on les transicions són "caixes" que contenen informació lèxica. Els lingüistes estan més familiaritzats amb aquesta representació perquè l' informació és plasmada d'una manera més gràfica i no han de tenir gaires coneixements sobre l'estructura dels d'autòmats per representar les gramàtiques electròniques.

Aquesta aplicació serà utilitzada en el marc del projecte de recerca Spanish FrameNet Project en la generació i edició de diferents transductors per representar diferents estructures del Llenguatge Natural.

També serà utilitzada en els laboratoris de Lingüística informàtica del departament de Filologia Espanyola de la UAB amb el mateix propòsit.

4.2 Edició d'autòmats:

Per una banda l'usuari disposa de les eines necessàries per crear i modificar autòmats. S'ha afegit a la barra d'eines les principals opcions que l'usuari necessitarà alhora de treballar com icones representatives de la seva funcionalitat per oferir una aparença més elegant. En cas de dubte sobre l' utilitat de cada eina mantenint el cursor sobre apareix una breu descripció. L'usuari en qualsevol moment pot recórrer al manual d'usuari que porta incorporat l'aplicació on trobarà la resposta a qualsevol dubte sobre l' utilització i les capacitats de l'aplicació.

Les opcions de la barra d'eines proporcionen un gran potencial a l'aplicació perquè estan orientades a l'edició de gramàtiques electròniques. Quan s'afegeix una transició l'usuari no ha preocupar-se de respectar l'estructura que han de tenir les diferents paraules del corpus. L'eina proporciona a l'usuari un panell on escollir l' informació lèxica de cada camp de forma excloent per eliminar errors de sintaxi. Com que els camps que especifiquen una paraula són finits l'usuari només ha d'escollir l'opció que desitja d'una llista desplegable eliminant errors d'ortografia. Amb aquest procés tan guiat l'usuari ja no perdrà el temps depurant problemes que tenia amb els seus autòmats per errors de sintaxi o ortogràfics produïts en la seva especificació. S'ha tingut en compte que l'usuari en un moment determinat pot voler canviar l' informació lèxica d'una paraula. Per evitar obligar-lo a eliminar la transició i afegir una nova amb el contingut actualitzat comportant possibles problemes d'omissió d'estats que canviarien l'autòmat per culpa del comú oblit, es proporciona a l'usuari l'opció d'accedir a un panell amb tota l' informació lèxica estructurada en els seus diferents camps perquè la pugui modificar còmodament sobre la mateixa transició.

La figura de l'estat en aquests autòmats és tant simple com una aresta entre transicions. S'han afegit petits rectangles als seus extrems per oferir una millor visualització de l'autòmat. També se'ls ha proporcionat un rectangle situat al centre del segment per poder seleccionar millor l'estat. Aquest rectangle és de gran utilitat entre altres coses per modificar la curvatura de l'estat ja que permet seleccionar-lo fàcilment enlloc de perdre temps intentant seleccionar una línea que té menys superfície. Aquest rectangle central també s'utilitza per mostrar el menú d'aquell estat quan es seleccionat amb el botó dret del ratolí. S'ha estudiat les necessitats de l'usuari per oferir-li funcionalitats innovadores. Per una banda li oferim la possibilitat de guardar la curvatura de l'estat. D'aquesta manera als estats que se'ls vol donar forma de paràbola per no passar per sobre d'altres elements s'estalvien la feina repetitiva de tornar a corbar-los cada vegada que l'usuari mou alguna de les transicions dels seus extrems. Activant aquesta opció al moure una d'aquestes transicions la curvatura de l'estat es manté al desplaçament. L'altra opció que oferim a l'usuari és afegir una

etiqueta als estats com a recordatori. De manera que l'ajudi a situar-se o recordar parts d'un autòmat durant l'edició.

La zona de disseny creix dinàmicament a mesura que l'usuari va expandint l'autòmat per suportar autòmats tant grans com l'usuari desitgi. Per ampliar l'àrea l'usuari simplement ha de moure un element fora de l'escena i l'aplicació expandeix els seus límits per incloure-la. D'aquesta manera la zona de disseny té la mida amb la que l'usuari està treballant en tot moment evitant haver de tenir molta més zona de la necessària, fet que pot alentir l'aplicació sobre ordinadors poc potents. Quan l'autòmat té una mida que cap en la finestra apareixen dos barres de desplaçament per seleccionar la zona on volem treballar. L'usuari disposa d'un zoom en la barra d'eines per veure l'autòmat amb perspectiva. L'usuari pot allunyar-se tant com necessiti per poder veure l' imatge global de l'autòmat o aproximar-se i veure clarament l' informació lèxica de cada transició.

En tot moment la perspectiva que s'ha tingut durant l' implementació ha estat enfocada a l'usuari final. D'aquesta manera a part de crear una aplicació amb una funcionalitat determinada hem aconseguit que l'usuari treballi còmodament amb ella. Una de les millores que li hem afegit al programa és l'opció de desfer l'últim canvi referent a l' inserció o eliminació d'un element de l'escena. Aquesta opció salvarà a l'usuari de situacions on inconscientment esborri un element. En una primera instància pot semblar una opció poc important però les coses canvien quan l'usuari elimina per error una transició i no recorda com estava connectada.

També s'ha proporcionat a l'usuari l'opció de simplificar el seu autòmat amb la fusió de transicions. Dos transicions ja siguin simples o compostes es poden fusionar en una sola transició composta. Això comporta l' unificació de les dos transicions en una sola caixa i posar en comú els seus estats. D'aquesta manera transicions que representaven diferents especificitats d'una mateixa paraula i que per tant tenien les mateixes connexions duplicades per suplir cada cas és fusionen en una sola transició eliminant tots els estats redundants. Perquè l'aplicació fos més complerta també se li proporciona a l'usuari l'opció de descompondre transicions complexes en simples i permetre transicions compostes heterogènies.

Les transicions compostes homogènies tenen la mateixa forma canònica i la mateixa sortida del transductor. Hem deixat a judici de l'usuari l'opció de fusionar transicions amb diferent corma canònica (transicions compostes heterogènies) per si es troba en una situacions que li seria útil d'utilitzar.

L'eina es totalment personalitzable fins al extrem que l'usuari disposa de l'opció d'escollir el color de cada element de l'escena. Cada usuari treballarà amb els colors que més còmode es senti i els podrà anar canviant per diferents autòmats.

Per últim s'ha volgut proporcionar a l'usuari seguretat. S'ha implementat com a mesura de seguretat la confirmació del tancament d'una pestanya o de tota l'aplicació. Quan un usuari tanca una pestanya per assegurar-nos que no ha estat un accident i evitar la pèrdua de treball que suposaria, se li mostra una finestra emergent on pot escollir entre guardar els canvis, continuar amb el tancament o cancel·lar l'operació. Pel cas que es tanqui tota l'aplicació a l'usuari se li demanarà la confirmació de tots els documents oberts.

4.3 Generació d'autòmats:

S'ha donat molta importància a l'edició dels autòmats perquè l'usuari es sentis còmode amb l'aplicació però també s'ha aconseguit un programa potent. Capaç de tenir oberts més d'un autòmat alhora i tenir-los organitzats per pestanyes.

És essencial que el treball realitzat per l'usuari pugui ser guardat per reprendre'l en un altre moment o que pugui obrir autòmats creats per altres usuaris. S'ha treballat com a base amb el FA per emmagatzemar l' informació essencial de l'autòmat i per recuperar-la. Aquest format no guarda la posició dels diferents elements per tant per cada document se li ha creat un document de posicions (PT) encarregat d'emmagatzemar la posició de les diferents transicions.

Els usuaris que utilitzen aquesta aplicació poden estar acostumats a treballar amb ER per tant l'usuari té l'opció d'importar autòmats a partir d'ER o exportar-los en aquest format.

Per complementar les diferents opcions d'emmagatzemament de l' informació l'usuari pot guardar tot l'autòmat com una imatge en un dels següents formats: "bmp", "gif","jpg","jpeg","png".

4.4 Multiplataforma:

Un objectiu era crear una eina plataforma perquè un major públic pogués utilitzar-la.

L'aplicació ha estat implementada amb Java perquè pugui ser executada sobre qualsevol plataforma que tingui instal·lat Java Platform SE 6 o superior (ha d'incorporar la llibreria Swing).

Java platform SE 6 es pot descarregar i instal·lar gratuïtament des de la pàgina web de Java (http://java.sun.com/javase/downloads/index.jsp). En cas que el SO que utilitza l'usuari no estigui disponible en aquesta web com és el cas de Mac OS s'ha de buscar la versió equivalent a Java SE 6 en aquest cas Mac OS X 10.4.7 release 4 i descarregar-la des de la web del fabricant:
http://www.apple.com/downloads/macosx/apple/application_updates/javaformacosx105update4.html

S'han donat problemes per instal·lar aquesta actualització per tant es recomana instal·lar SnowLeopard a partir de la versió 10.6.

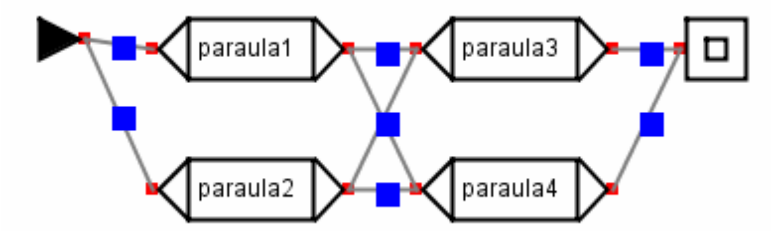
Com que el programa ha estat implementat amb Java ha estat senzill crear una interfície i convertir-lo en Applet de Java per poder ser afegit en qualsevol document web.

4.5 Anàlisis de resultats:

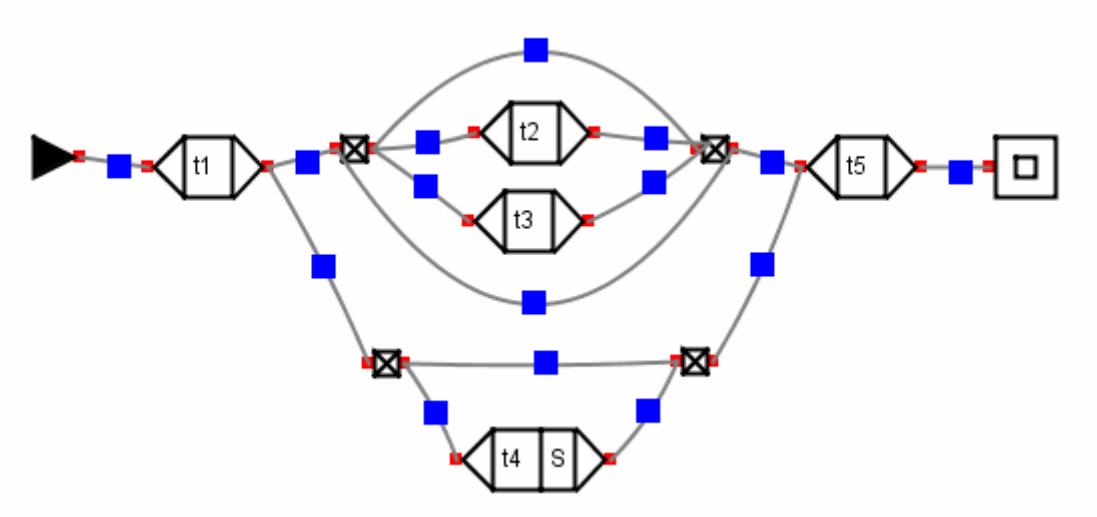
A continuació és mostraran una sèrie d'autòmats resultants de carregar diferents fitxers i els fitxers resultants de guardar altres autòmats.

4.5.1 Importació i exportació d' Expressions Regulars:

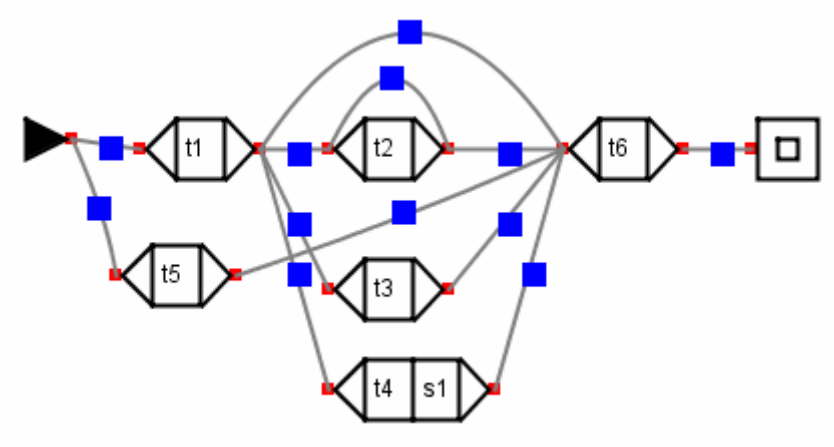
ER1: (<paraula1>+<paraula2>)(<paraula3>+<paraula4>)



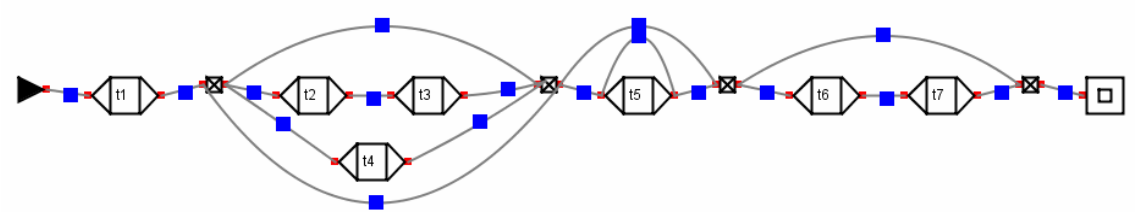
ER2: <t1>((<t2>+<t3>)*+(<t4>[S]+<E>))<t5>



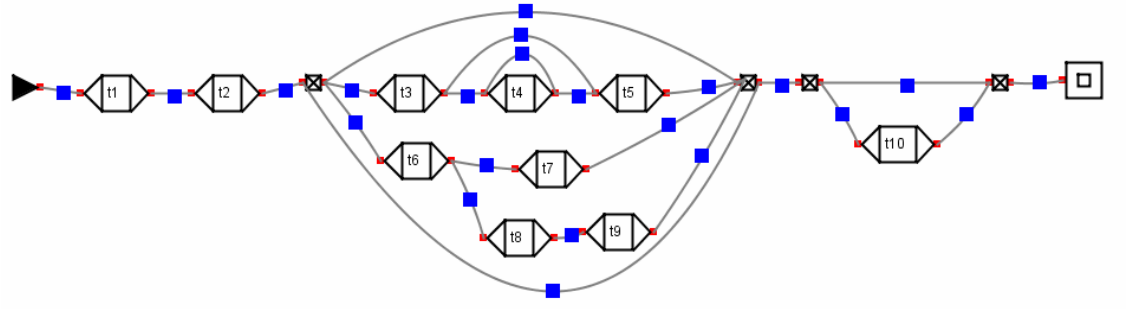
ER3: (<t1>((<t2>*+<t3>)+<t4>[s1])+<t5>)<t6>



ER4: <t1>(<t2><t3>+<t4>)*<t5>*(<t6><t7>+<E>)



ER 5: <t1><t2>(<t3><t4>*<t5>+<t6>(<t7>+<t8><t9>))*(<t10>+<E>)

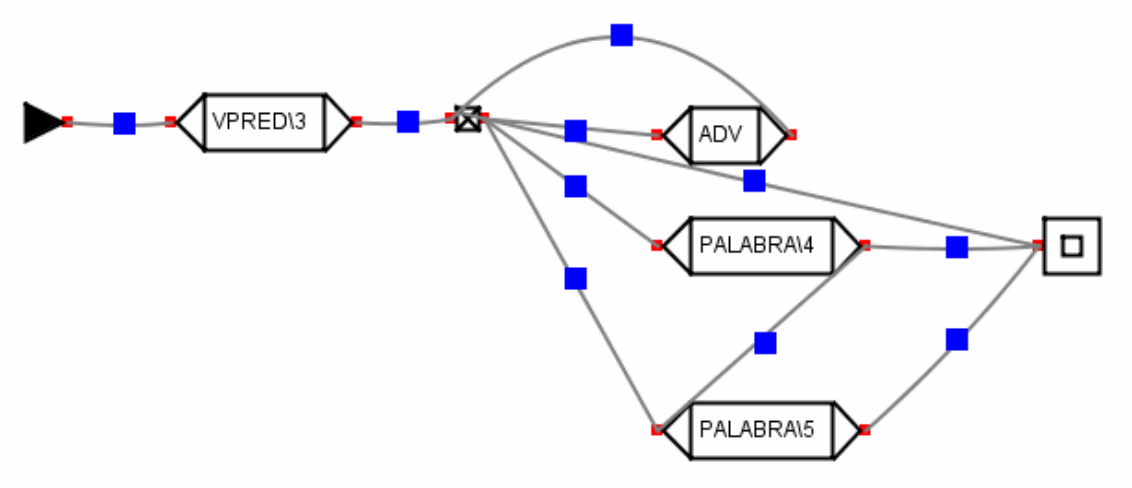


S'ha comprovat que un cop carregat cada arxiu si s'exporta com ER concorden amb l'ER original. A excepció del cas de ER3 on l'ER exportada és la seva versió simplificada: (<t1>(<t2>*+<t3>+<t4>[s1])+<t5>)<t6>.

4.5.2 Obrir i guardar documents en FA:

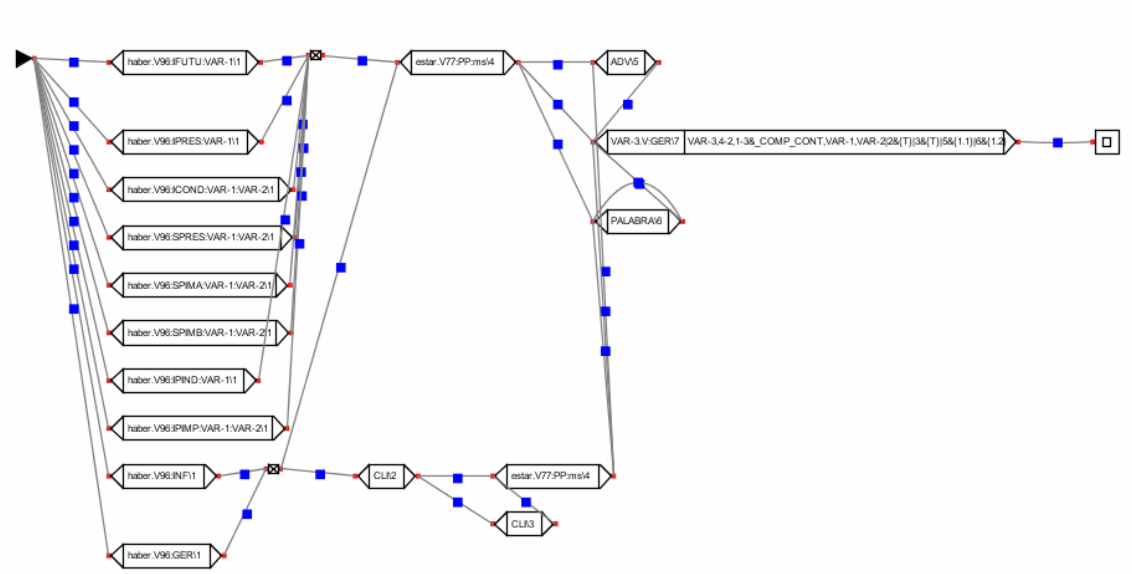
Autòmat 1:

```
4 4
%<VPRED\3>%<ADV>%<PALABRA\4>%<PALABRA\5>%
: 0 2 -1
t 1 2 2 3 3 4 -1
t 3 4 -1
t -1
f
```



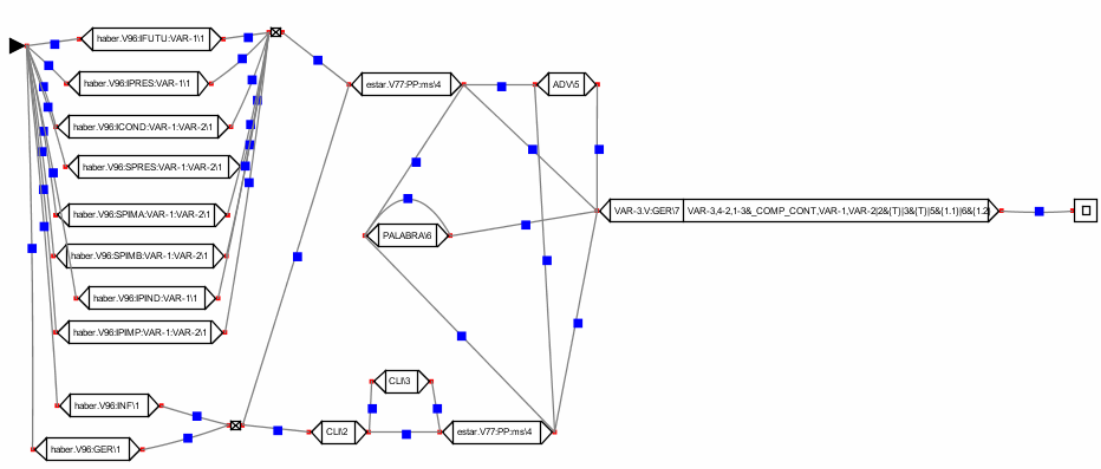
Autòmat 2:

16 10
%<haber.V96:IFUTU:VAR-1\1>%<estar.V77:PP:ms\4>%<ADV\5>%<VAR-
3.V:GER\7>[VAR-3,4-2
,1-3&_COMP_CONT,VAR-1,VAR-
2|2&{T}|3&{T}|5&{1.1}|6&{1.2}]%<PALABRA\6>%<haber.V96:
IPRES:VAR-1\1>%<haber.V96:ICOND:VAR-1:VAR-2\1>%<haber.V96:SPRES:VAR-1:VAR-
2\1>%<
haber.V96:SPIMA:VAR-1:VAR-2\1>%<haber.V96:SPIMB:VAR-1:VAR-
2\1>%<haber.V96:IPIND:
VAR-1\1>%<haber.V96:IPIMP:VAR-1:VAR-
2\1>%<haber.V96:INF\1>%<haber.V96:GER\1>%<CL
I\2>%<CLI\3>%
: 0 2 5 2 6 2 7 2 8 2 9 2 10 2 11 2 12 3 13 3 -1
: 1 4 -1
: 1 4 14 5 -1
: 2 6 3 10 4 7 -1
: 1 8 15 9 -1
: 3 10 -1
: 3 10 4 7 -1
: 2 6 3 10 4 7 -1
: 1 8 -1
t -1
f



Igual que en el cas anterior s'ha comprovat que els autòmats es guardin i es carreguin en la mateixa posició.

Abans de guardar l'autòmat 2 es va reorganitzar les transicions per aconseguir una aparença més estètica.



A diferència de quan guardàvem les ER en aquest cas el programa recalcula els estats absoluts i és típic que la definició d'estats no coincideixi amb l'arxiu original.

<u>L'autòmat 2</u> guardat a fitxer queda de la següent manera:

16 10

%<haber.V96:IFUTU:VAR-1\1>%<estar.V77:PP:ms\4>%<ADV\5>%<VAR-3.V:GER\7>[VAR-3,4-2,1-3&_COMP_CONT,VAR-1,VAR-

2|2&{T}|3&{T}|5&{1.1}|6&{1.2}]%<PALABRA\6>%<haber.V96:IPRES:VAR-1\1>%<haber.V96:ICOND:VAR-1:VAR-2\1>%<haber.V96:SPRES:VAR-1:VAR-2\1>%<haber.V96:SPIMA:VAR-1:VAR-2\1>%<haber.V96:SPIMB:VAR-1:VAR-2\1>%<haber.V96:IPIND:VAR-1\1>%<haber.V96:IPIMP:VAR-1:VAR-2\1>%<haber.V96:INF\1>%<haber.V96:GER\1>%<CLI\2>%<CLI\3>%

: 0 2 5 2 6 2 7 2 8 2 9 2 10 2 11 2 12 7 13 7 -1
: 1 3 -1
: 2 4 3 5 4 6 -1
: 3 5 -1
t -1
: 3 5 4 6 -1
: 1 3 14 8 -1
: 1 9 15 10 -1
: 2 4 3 5 4 6 -1
: 1 9 -1
f

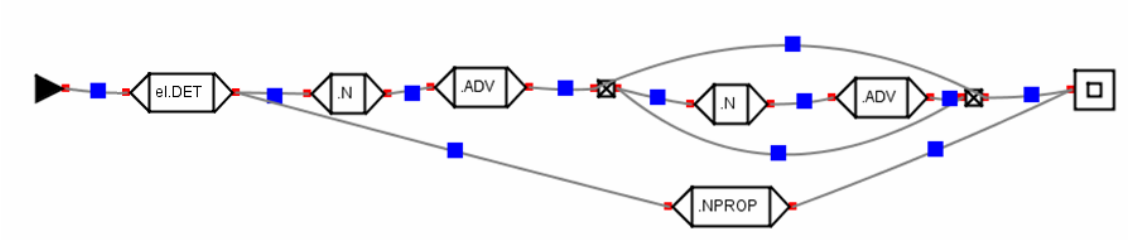
El seu fitxer de posicions té el següent contingut:

16
0 170 80
1 527 140
2 773 140
3 855 307
4 548 340
5 153 139
6 139 196
7 152 249
8 139 313
9 136 367
10 166 423
11 139 468
12 141 565
13 109 623
14 474 600
15 557 533
75 94
1484 307

S'ha comprovat que es carregui l'autòmat i que les transicions estiguin en la mateixa posició que se'ls havia assignat al guardar l'autòmat.

4.5.3 Creant un autòmat des de zero:

S'ha creat el següent autòmat



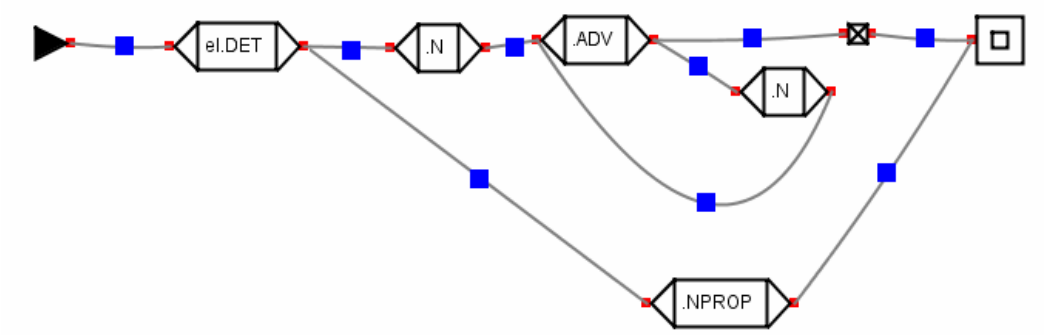
Si exportem com ER obtenim:

<el.DET>(<N><ADV>(<N><ADV>)*+<NPROP>)

S'ha guardat en FA:

```
4 7
%<el.DET >%<N>%<ADV>%<NPROP>%
: 0 2 -1
: 1 3 3 7 -1
: 2 4 -1
: 1 5 -2 6 -1
: 2 6 -1
t -1
t -1
f
```

Observem que l'autòmat que ha guardat és la seva versió simplificada. Carregant aquest document tenim el següent autòmat:



Aquesta és una petita part d'un extens conjunt de proves que s'ha realitzat per controlar tots els casos possibles a l'hora de guardar i carregar els autòmats.

4.6 Possibles millores:

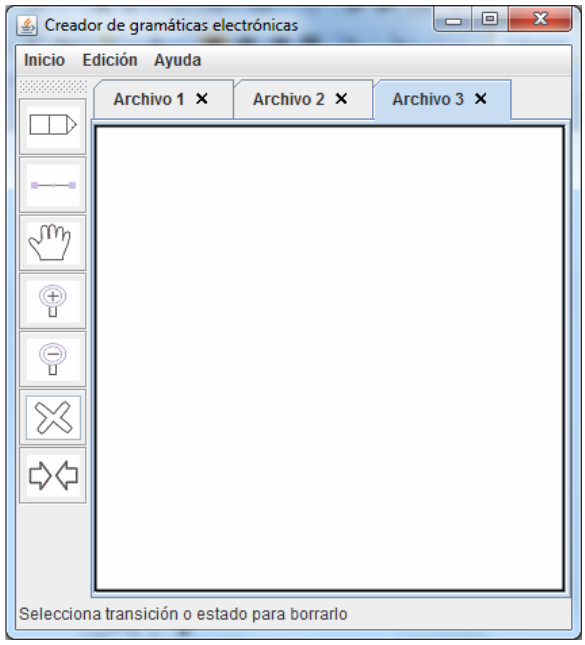
Durant el transcurs del projecte s'han anat incorporant noves idees per millorar el programa però que al final no s'ha disposat de temps suficient per portar-les a terme. Les ampliacions que es volien realitzar eren les següents:

- Fer una selecció de varis estats i transicions per poder-los copiar i enganxar.
- Moure els elements per teclat i eliminar transicions amb el botó suprimir un cop seleccionades.
- Traduir l'aplicació a varis idiomes.
- Afegir un camp de recerca que seleccioni transicions que contenen una determinada informació lèxica.
- Llegir varis autòmats d'un mateix fitxer
- Executar l'autòmat com a test. Permetre que l'usuari introdueixi un text d'entrada i observar el recorregut pels diferents estats de l'autòmat.

Annex

Annex
Manual d'usuari

S'ha afegit a l'aplicació un manual d'usuari perquè tots els usuaris puguin consultar-lo per aprofitar totes les opcions de l'eina.



Elements de l'aplicació:

En la part central de l'aplicació es troba la zona de disseny on es representen els autòmats. Per expandir l'àrea l'usuari ha de moure una transició fora dels seus límits lateral dret o inferior.

La barra de menú està situada a la part superior de l'aplicació i la barra d'eines està col·locada verticalment a l'esquerra tot i que es pot canviar de posició arrossegant-la.

Sota la barra de menú es troben diferents pestanyes per cadascun dels diferents documents oberts.

A la part inferior van apareixent notificacions per guiar a l'usuari en les diferents tasques que porta a terme.

Barra d'eines:

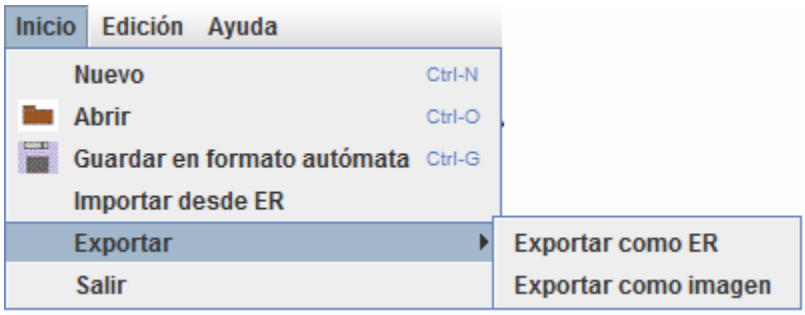
Les eines que treballen amb estats o transicions per seleccionar-los s'ha de fer clic amb el botó esquerra del ratolí.

1. Afegir transició: Selecciona la posició de la zona de disseny on afegir la transició. A continuació s'obrirà una finestra emergent per especificar d'informació lèxica de la transició.
2. Afegir un estat: Seleccionar la transició origen. Sense deixar de prémer el ratolí situar el cursor sobre la transició destí. Alliberar el ratolí en aquella posició per afegir l'estat.
3. Aquesta opció permet canviar de posició els diferents elements de l'escena. Un cop seleccionats s'arrosseguen fins la

seva nova localització. Per canviar la curvatura d'un estat s'ha d'arrossegar el seu rectangle de menú (el rectangle que es troba en la seva posició central).

4. i 5 . Realitzen un zoom sobre l'escena. Ítem 4 per aproximar-se i ítem 5 per allunyant-se.
6. Eliminar element: Seleccionar un estat o transició per eliminar-lo.
7. Fusionar transicions: Seleccionar dos transicions compatibles per convertir-les en una única transició. Útil per simplificar l'autòmat. Dos transicions són compatibles si tenen la mateixa forma canònica i la mateixa sortida de transductor.

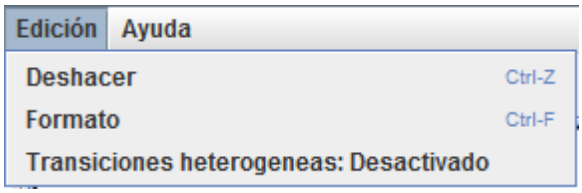
Barra de menú



Opcions del menú "Inicio":

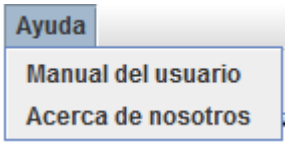
- Nuevo: Crear un nou document en blanc.
- Abrir: Carrega l'autòmat d'un fitxer en Format Autòmat (.aut).
- Guardar en formato autómata: Guarda l'autòmat de l'escena sobre la que s'està treballant en un fitxer en FA.
- Importar desde ER: Carrega en un nou document l'autòmat que representa l'expressió regular del document seleccionat. Els documents que s'importen han de tenir extensió ".txt" o ".er". En aquell document només pot haver una ER correctament escrita. En altre cas es carregarà erròniament.
- Exportar:
  - Exportar com ER: Guarda l'ER que representa l'autòmat de l'escena.
  - Exportar como imagen: Guarda l'escena com una imatge. Es pot escollir entre els següents formats: "bmp", "gif","jpg","jpeg" i"png".
- Sortir: Sortir de l'aplicació.

Opcions del menú "Edición":



- Deshacer: Desfà l'últim canvi si està relacionat amb l' inserció o eliminació d'un estat o transició.
- Formato: Mostra un menú emergent on es poden canviar els colors dels diferents elements de l'escena.
- Transiciones heterogeneas: Per defecte dos transicions es poden fusionar només si tenen la mateixa sortida de transductor i el camp forma canònica de les seves informacions lèxiques es correspon. Si s'activa aquesta opció es permet que dos transicions amb forma canònica diferent es puguin fusionar.

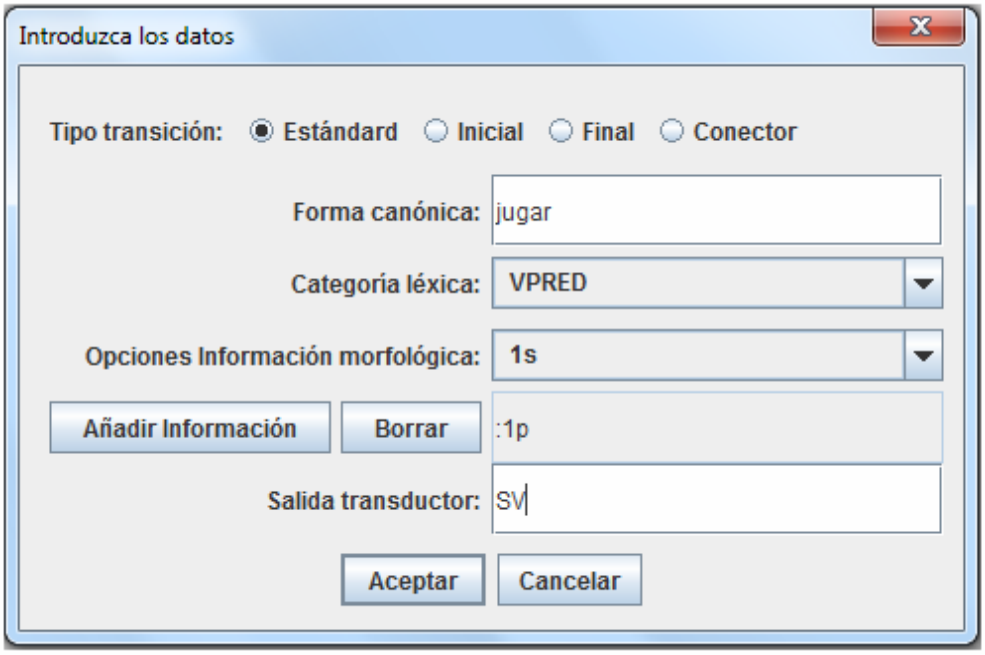
Opcions del menú "Ayuda":



Aquest menú és únicament informatiu.

L'opció manual d'usuari mostra en una finestra emergent aquest manual d'usuari.

Creació de transicions:

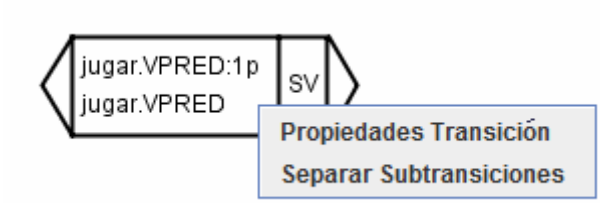


Aquesta finestra apareix a continuació d'haver seleccionat la posició de l'escena on afegir la transició. En la part superior es selecciona el tipus de transició. En cas de ser una transició estàndard es mostren les opcions necessàries per completar l' informació lèxica de la transició.

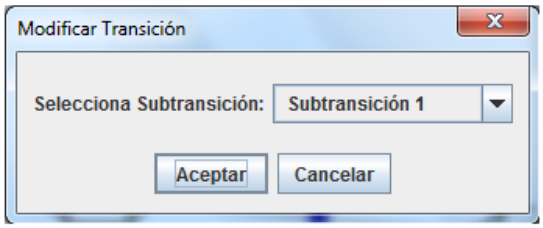
Menús individuals dels elements:

Per accedir als menús individuals de les transicions i dels estats s'ha de fer clic amb el botó dret del ratolí sobre seu.

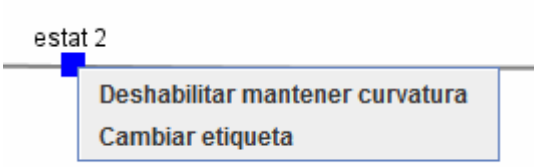
Menú de les transicions:



- Separar Substransiciones: En cas de seleccionar una transició composta s'observa aquesta opció que permet descompondre-la en transicions simples.
- Propiedades Transición: Podem canviar l' informació lèxica de la transició. En cas d'una transició composta es preguntarà sobre quin subelement es vol actualitzar l' informació. En cas que l'opció "Transicions heterogeneas" estigui desactivat si es modifica el camp forma canònica de la transició es modifica per tots els elements de la transició composta.



Menú dels estats:



- Habilitar mantener curvatura: Quan es mouen les transicions la posició dels estats que depenen d'ells s'actualitzen formant una línea recta entre la transició origen i destí. En cas de tenir habilitada aquesta opció al moure les transicions es conserva la curvatura de l'estat. Per tant en cas de tenir estats que volem que tinguin forma de paràbola, evitem haver de canviar la curvatura dels estats cada vegada que movem una de les transició dels seus extrems.
- Cambiar etiqueta: Canvia el valor de l'etiqueta de l'estat. Per defecte està buit.

Bibliografia

Charles J. Fillmore, “Frame semantics and the nature of language”. In Annals of the New York Academy of Sciences: Conference on the Origin and Development of Language and Speech, Volume 280: 20-32, 1976

Charles J. Fillmore, “The need for a frame semantics in linguistics”. In Karlgren, Hans (Ed.): Statistical Methods in Linguistics 12: 5-29, 1977.

Charles J. Fillmore and B. T. S. Atkins (), “Towards a frame-based lexicon: The semantics of RISK and its neighbors”. In Lehrer, A and E. Kittay (Eds.) Frames, Fields, and Contrast: New Essays in Semantics and Lexical Organization. Hillsdale: Lawrence Erlbaum Associates, 75-102, 1992.

J. E. Hopcroft, J. D. Ullman, “*Introduction to Automata Theory, Languages and Computation”,* Addison-Wesley, Reading, Mass., 1979.

M. Mohri, “Finite-state transducers in language and speech processing”, *Computational Linguistics,* vol. 23(2) , pág. 269-311, 1997.

M. Ortega, “Teoría de Autómatas aplicada a la Lingüística Informática”. *Facultat de Ciències. Secció d'Enginyeria Informàtica,* Universitat Autònoma de Barcelona, 1997.

M. Silberztein, “Finite State Language Processing with INTEX”, en *Pre-Conference Tutorial. COLING-ACL-98,* Université de Montreal. Montréal, Québec, Canada. http://www.ladl.jussieu.fr/INTEX/Tutorial.htm, 1998.

Noelia Méndez i José Luis García, “Programación con Swing”, Departamento de Informática i Autonomía, Universidad de Salamanca, 2005.

W. Brauer, “On Minimizing Finite Automata”, *Bulletin of the European Association for Theoretical Computer Science, EATCS,* vol. 32 , pág. 113-116, 1988.

Webgrafia:

http://java.sun.com/docs/books/tutorial/uiswing/

http://www.programacion.com/articulo/graficos_con_java_2d_111/10

http://www.javahispano.org/

Firmat: J. Oriol Aguadé Estivill
Bellaterra, 21 de Juny de 2010

Resum

Aquest projecte tracta la implementació d'una eina gràfica multiplataforma de creació i edició de gramàtiques electròniques per representar el Llenguatge Natural. És una eina per lingüistes i projectes com Spanish FrameNet Project amb la quan poden representar fàcilment transductors en un format més visual, les transicions es representen en forma de "caixes", i guardar els resultats. S'han implementat varies opcions per crear una eina còmode i personalitzable per l'usuari amb funcionalitats enfocades a les seves necessitats com importar/exportar autòmats des d'una Expressió Regular. Es tracta l' implementació de tots els components que s'han necessitat per crear la GUI així com la seva funcionalitat.

Resumen

Este proyecto trata la implementación de una herramienta gráfica multiplataforma de creación y edición de gramáticas electrónicas para representar el Lenguaje Natural. Es una herramienta para lingüistas y proyectos como Spanish FrameNet Project con la cual puedan representar fácilmente transductores en un formato visual, las transiciones se representan en forma de "cajas", y guardar el resultado. Se han implementado varias opciones para crear una herramienta cómoda y personalizable para el usuario con funcionalidades destinadas a sus necesidades como importar/exportar autómatas desde una Expresión Regular. Se trata de la implementación de todos los componentes que han sido necesarios para la GUI así como su funcionalidad.

Abstract

This project is implementing a multi-platform graphical tool for creating and editing electronic grammars to represent the natural language. Linguists and projects such as Spanish FrameNet Project could use this tool to easily represent transducers in a visual format, where transitions are represented with a "box" shape, and save the results. Several options have been implemented to create a comfortable and customizable tool for the user with functionality for his needs as import/export from a Regular Expression. This is the implementation of all components that have been necessary for the GUI and functionally.