Universitat de Girona
Escola Politècnica Superior

Treball final de grau

Estudi: Grau en Enginyeria Informàtica

Títol: Plataforma de suport docent a l'estudi d'autòmats.

Document: Memòria

Alumne: Josep Aragonès i Bargalló

Tutor: Dr. Jaume Rigau Vilalta

Departament: d'Informàtica, Matemàtica Aplicada i Estadística

Àrea: Gilab

Convocatòria (mes/any): 01/16

Índex

Índex de Descripcions:

Índex de Figures:

1. Introducció.

Actualment a la Universitat de Girona s'hi cursa l'assignatura de "Fonaments de Computació" dins el grau d'Enginyeria Informàtica. El seu objectiu és introduir els alumnes en els llenguatges formals i disseny d'autòmats, alhora que oferir-los els primers conceptes sobre la teoria de la decidibilitat i la complexitat.

L'objectiu d'aquest projecte és desenvolupar una eina de caràcter docent on-line per posar a disposició dels estudiants i professors involucrats a l'assignatura prèviament esmentada o bé de similars en el seu mateix context. L'anomenarem *Plataforma de suport docent a l'estudi d'autòmats.*

Més concretament l'aplicació serà dissenyada per ser una eina didàctica que ens permeti dissenyar qualsevol tipus d'autòmat finit bàsic, de pila o màquina de Turing, i tanmateix la seva execució. També ha de permetre que s'enregistrin codificacions personals de qualsevol dels autòmats esmentats, i computar-les d'acord a les diferents opcions que ofereix la plataforma.

Dins del apartat dels autòmats finits, l'aplicació permetrà emprar un conjunt de rutines habituals en el contexts de treball d'aquesta tipologia d'autòmats, com són: la determinització, minimització, obtenció de l'expressió regular i obtenció d'un autòmat finit a partir d'una expressió regular.

La motivació per dur a terme aquest projecte ha estat la possibilitat d'aplicar coneixements d'aquesta matèria del grau especialment conceptuals interseccionant amb uns de més pragmàtics dins la programació com ha estat la utilització dels Java Server Faces, no explicats a la carrera i alhora aprendre sobre el seu funcionament. Tanmateix, la interdisciplenariatat del projecte en el que respecte a disseny, programació i xarxa són un handicap que a nivell personal m'interessava a molt portar a terme.

El treball m'ofereix la possibilitat doncs, de posar en pràctica molts conceptes i tecnologies estudiades a la carrera. A nivell de disseny posar en pràctica els conceptes adquirits, i poder assolir els objectius assolits.

2. Estudi de viabilitat

Aquest projecte s'ha pogut realitzar a partir dels coneixements adquirits prèviament al llarg de la carrera.

L'objecte primari d'estudi involucrat són els autòmats finits que s'han analitzat a l'assignatura obligatòria de "Fonaments de Computació". Els altres aspectes relacionats amb el treball s'han desenvolupat a les assignatures del grau corresponents a disseny i programació.

A data d'avui l'assignatura utilitza una eina bàsica a nivell de linea de comandes i que esta ubicada al servidor BAS, de l'Escola Politècnica Superior. Tot hi que compleix els seus objectius, no està excent de diferents handicaps: els usuaris s'han de connectar als servidors de la universitat (fet que a vegades i per culpa d'atacs d'internet s'ha hagut de tallar l'accés), dificultats de comunicació, limitacions d'espai, espai multicompartit, etc.

Dit això, hi ha l'existència de molt bones eines amb objectius similars i enfocades directament a l'aprenentatge dels autòmats com seria l'eina "RACSO" desenvolupada per en Guillem Godoy de l'UPC, amb una altre línia més especifica existeix el "JFLAP", el "automaton", Cadascuna d'aquestes eines permet treballar autòmats des d'una perspectiva concreta. L'objectiu final de la nostre plataforma seria conjuntar tots aquest enfocs d'una manera simple i adequada als estudiants del grau de l'Enginyeria Informàtica d'aquí a la UdG.

Per tal d'implementar la nova eina s'utilitza per la part gràfica el Java Server Faces, i més concretament la suite de components, Primefaces, ja que en permet una bona interacció entre client i servidor i permet l'estudi d'un nou llenguatge i un nou tipus de programació seguint el patró MVC.

Durant la meva formació acadèmica ja he vist i he estudiat amb anterioritat els patrons de dissenys dins l'assignatura "Enginyeria del Software 2" i un nivell avançat de HTML i CSS3, adquirit en els múltiples projectes efectuats durant la meva estada al CSIC i a Comexi, dins l'assignatura optativa de 4t de les "Estades a l'Entorn Laboral".

Finalment, com que és una aplicació web, es necessita un lloc a on allotjar-la i l'elecció és Amazon Web Service. Aquesta plataforma proveeix de múltiples serveis i un d'ells és la possibilitat d'hostejar una WebApp durant tot un any de forma gratuïta. També permet la redirecció a un nom de domini propi i no el proporcionat per ells.

Per tant, la viabilitat del projecte en quant a coneixements, tecnologia i economia per part meva eren factibles i el cost el considero de maquinari i software els considero

mínims(es poden considerar zero), donat que el hardware és propi o gratuït i el software és de propietat personal.

En conclusió el projecte és totalment viable, tant per temps com per costos. L'aspecte temps es refereix els crèdits associats directament al projecte, ja que com veurem a l'apartat de Treball Futur, ampliacions diverses sempre serien possibles.

3. Metodologia

La metodologia de treball emprada per desenvolupar aquest treball s'ha basat en utilitzar una de les múltiples metodologies àgils de desenvolupament, l'Scrum. Tanmateix s'escau perfectament, perquè la vam utilitzar en l'assignatura de "Projecte de Desenvolupament del Software" del grau.

3.1.Scrum

D'acord amb el llibre "The New New Product Development Game"[1], Scrum es va definir per primera vegada el 1986 per Hirotaka Takeuchi i Ikujiro Nonaka com una estratègia flexible i com una aproximació holística al desenvolupament de productes, on un equip de desenvolupadors treballen com una unitat per aconseguir un objectiu comú. Per la creació de Scrum els seus autors es van basar en casos d'estudi d'empreses d'automoció, fotocopiadores i impressió. Durant el seu estudi, Nonaka i Takeuchi van comparar la nova forma de treball amb equip amb l'avanç en formació de la melé (scrum en inglés) dels jugadors de rugbi, per aquest motiu es va escollir el terme "scrum" per referir-se a aquesta metodologia de treball.

A principis del 1990, Ken Schwaber va utilitzar el que es convertiria Scrum a la seva companyia, Mètodes Avançats de Desenvolupament, i Jeff Sutherland, amb en John Scumniotales i Jeff McKenna, van desenvolupar un enfocament similar a Easel Corporation, i van ser els primers a referir-se a ell amb la paraula Scrum. El 1995, Sutherland i Schwaber van fer la primera presentació publica de Scrum on van presentar conjuntament un document on descrivien la metodologia de treball de Scrum en el disseny d'objectes de negoci i la implementació en el marc de la programació orientada a objectes, sistemes, idiomes i aplicacions '95 a Austin (Texas). Durant els següents anys Sutherland i Schwaber van treballar conjuntament per fusionar els escrits anteriors, les seves experiències i millorar les pràctiques de la industria fins a obtenir el resultat final de Scrum que és tal com el coneixem actualment.

L'Scrum l'objectiu del qual és desenvolupar i crear un producte en un període de temps determinat on un equip format per diferents persones treballen conjuntament per arribar a un objectiu comú.

Scrum és un model de referència que defineix un conjunt de pràctiques, on cada persona participant assumeix un rol (Scrum Master, Product Owner, Equip de desenvolupament, ...), fet que permet adaptar-se a les necessitats i preferències de cada equip o organització.

El primer que es fa és definir els objectius del projecte, separar-los per tasques a realitzar i assignar-los un temps necessari per realitzar aquestes tasques.

Un cop estan definides les tasques a fer, el *Product Owner* les ordena per ordre d'importància i preferències, tot creant el que es coneix com a *Product Backlog*, que és, en resum, el conjunt de requeriments a alt nivell prioritzats que defineixen la feina a fer en un període determinat.

Un cop es té el *Product Backlog* redactat, l'equip fa una reunió per a planejar i prioritzar tasques. És en aquest moment que es fa la planificació de sprints. Els ***sprints*** són períodes de temps fix d'una durada definida -solen ser d'entre una i quatre setmanes- on l'equip realitza les tasques que tenia assignades al Product Blacklog. Durant aquest període l'equip ha de realitzar les tasques encomanades per tal d'assolir la fita dins del període marcat. El procés de planificació previ es coneix com a *Sprint Planning*.

Durant el temps que dura l'Sprint, l'única part que pot canviar del *Backlog* és l'equip de desenvolupament, si veu que aquest no pot completar les tasques dins del període establert.

El sistema Scrum permet la creació d'equips auto-organitzats impulsant la colocació de tots els membres de l'equip, i la comunicació verbal entre tots els membres i disciplines involucrades en el projecte.

Per tal de treballar utilitzant la metodologia Scrum s'ha estructurat la feina a fer en una sèrie de tasques principals que seguirien un ordre concret i seqüencial. A partir de l'esquema inicial, aquestes tasques es dividirien en tasques secundàries, per poder tenir una estructura organitzada.

D'altra banda, s'anava informant al tutor periòdicament sobre l'evolució del treball com ara els dissenys fets, les proves realitzades o les decisions que s'anaven prenent; tanmateix, per les decisions que no estaven massa clares, s'han pres conjuntament.

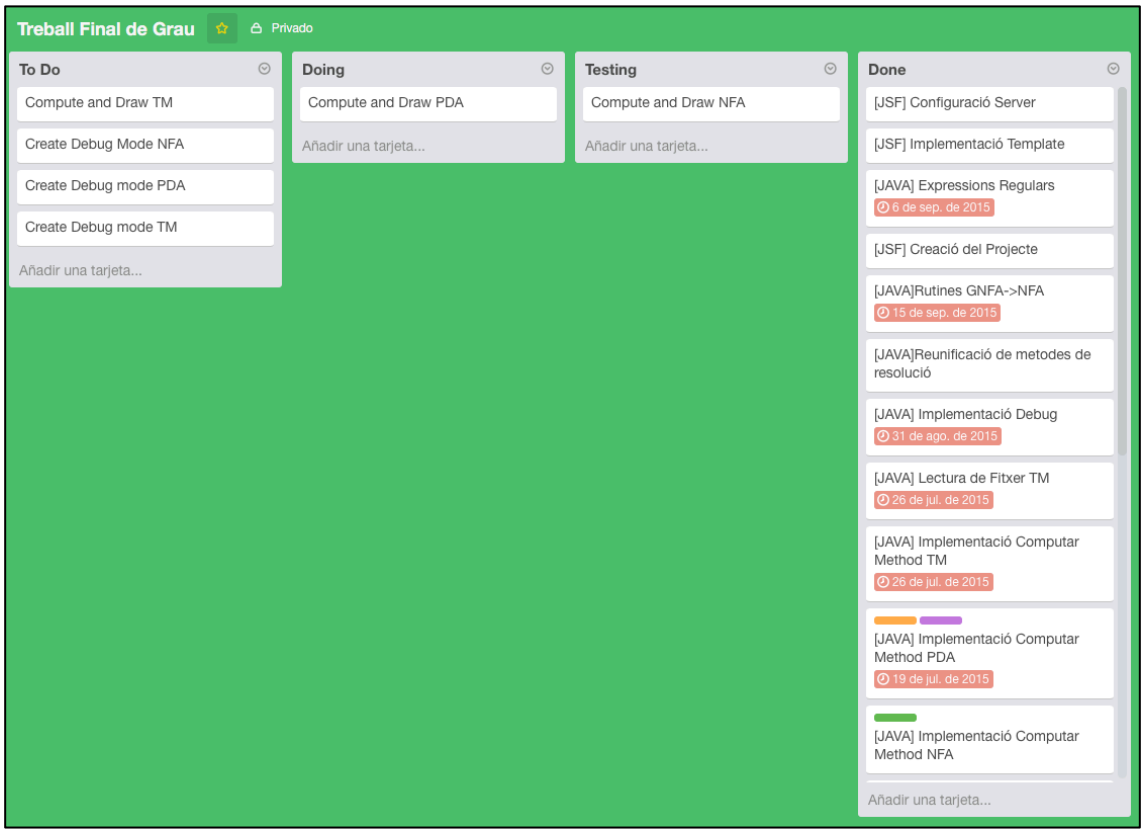
Així, el conjunt de tasques és el següent:

- **Estudi dels Conceptes Previs:** Estudi del funcionament dels autòmats Finits deterministes i indeterministes, Push-Down autòmats i les Màquines de Turing.
- **Estudi i comprensió de l'UTM:** Estudi i comprensió del projecte, implementat en C++, UTM i estructurant l'esquelet del nou projecte implementat en JAVA.
- **Estudi i comprensió dels Llenguatges Regulars:** Estudi i comprensió de la determinació, minimització, expressions regulars i isomorfisme de dos autòmats.

- **Implementació NFA:** Lectura des de fitxer d'un NFA-DFA, computació de el autòmat, implementació de diferents Flags per la computabilitat dels autòmats Finits.
- **Implementació PDA:** Lectura des de fitxer d'un PDA, computació de el autòmat, implementació dels Flags necessaris per la computabilitat del PushDown autòmats.
- **Implementació de TM:** Lectura des de fitxer d'un TM, computació de la TM i dels flags necessaris per a la computabilitat de la TM.
- **Determinització de NFA:** implementació de l'algoritme de determinització que incorpora en el llibre <ITC|Sipser>
- **Minimització de DFA:** implementació de l'algoritme de minimització que incorpora en el llibre <ITC|Sipser>
- **Reducció a Expressió Regular:** Implementació de l'algoritme de reducció de MFA a expressió regular.
- **Isomorfisme d'autòmats Finits:** Algoritme per tal de detectar si dos autòmats són equivalents o no.
- **Creació de l'entorn Web de l'Aplicació:** creació del projecte Web amb Java Server Faces i els Framework PrimeFaces.
- **Recerca i implementació de Layouts de Grafs:** Cerca de l'algoritme per dibuixar un graf de la a millor manera possible.
- **Creació de la Plana Principal:** creació del menú amb totes les utilitats i afegir la possibilitat d'exemples.
- **Creació d'un Petit tutorial:** Creació d'un petit tutorial per tal de fer més fàcil la comprensió de la Web.

4. Planificació

Per tal d'aplicar Scrum he utilitzat la plataforma Trello[2], la qual em permet fer el mur amb les etiquetes i les tasques a dur a terme cada dos setmanes. Vegem-ho a continuació:



Tal com és pot apreciar l'existència de 4 etiquetes, la primera "To Do", reflecteix el fet totes les feines que s'han de fer des de l'inici, estant ordenades de més prioritari a menys. La següent, ens indica el sprint actual quines tasques estem realitzant actualment. La tercera etiqueta serveix per tal de un cop implementat la tasca testejar-la un mateix o l'equip encarregat del mateix, el meu cas un parell de companys de la carrera.

I per últim l'etiqueta "Done" a la qual hi han d'arribar totes les tasques un cop han estat implementades i testejades correctament.

Per tal de d'aportar una visió temporal de la implementació del projecte, adjunto un diagrama de Gantt per tal de veure amb més claredat la planificació ideada al principi. Al tenir prop de 8 mesos per poder entregar el Treball, vaig decidir acabar tal com és veu en el diagrama 1 mes abans de l'entrega.

Pensant que en cas de sorgir algun problema, bug, etc. tenir un coixí de temps per tal de no patir pel temps. I el cas és que la tasca de minimització i la de reducció de NFA hi van sorgir problemes i per tant he acabat més tard del que havia previst feia 8 mesos abanss

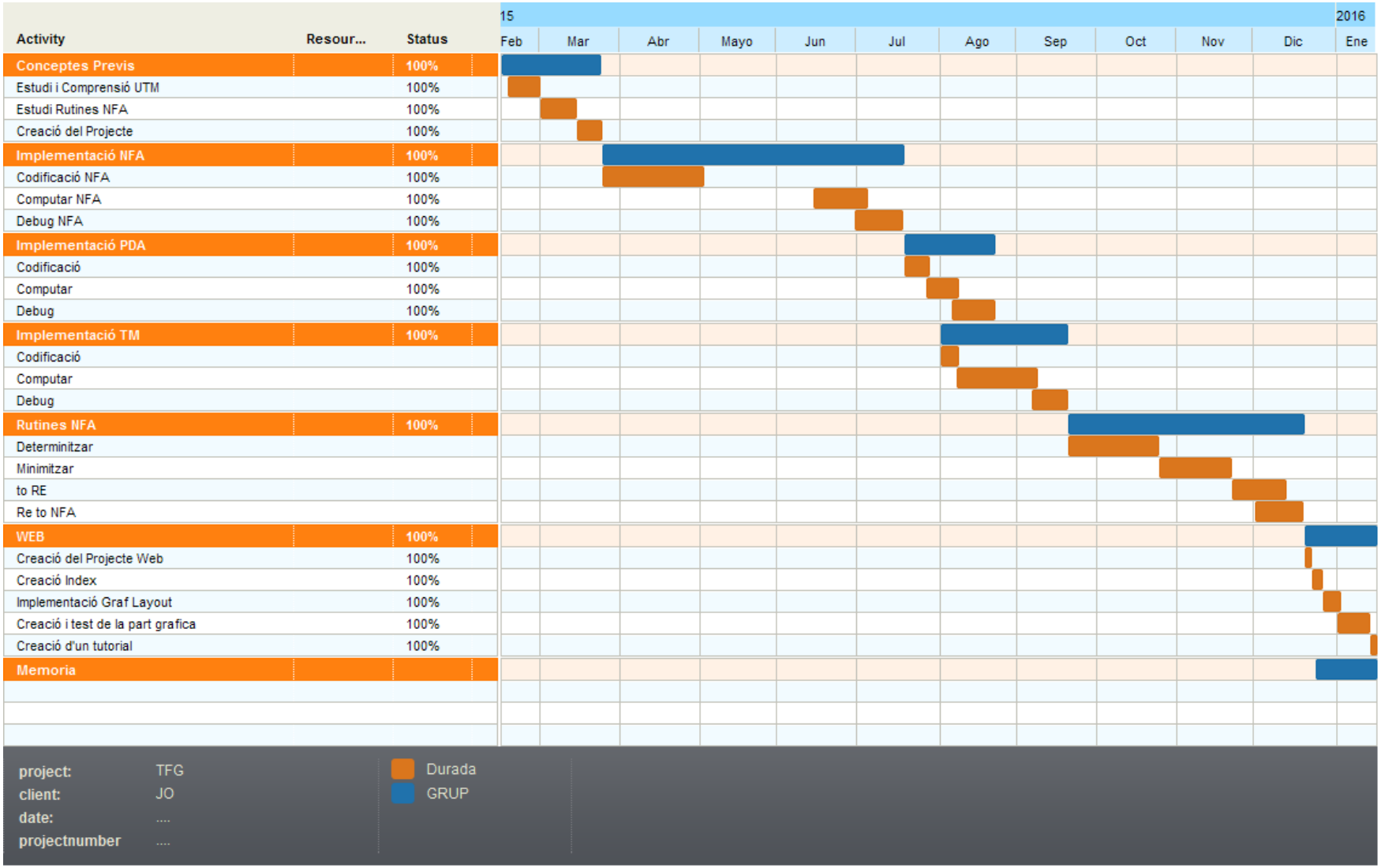


Figura 1 Diagrama de Gantt

5. Marc de treball i conceptes previs

Al ser una eina d'us didàctic s'han treballat per tal que la seva comprensió sigui del tot intuïtiva. Degut a la dificultat dels conceptes s'ha optat per explicar-los amb força enteniment.

Els conceptes exposats aquí segueixen l'estructura del llibre que és segueix per dur a terme la docència de l'assignatura, el "Introduction to the Theory of Computation"[3] de l'autor, Michel Sipser.

5.1.Llenguatges Regulars

Un autòmat finit (AF) o màquina d'estats finits és un model matemàtic d'un sistema compost per estats, transicions i accions. Un estat emmagatzema informació del passat. Una transició indica un canvi d'estat i es descriu per la condició que és necessària acomplir per activar la transició. Una acció és una descripció d'una activitat que es realitza en un moment donat.

Més formalment, un autòmat finit és un ordinador amb una memòria molt minsa capaç de computar i dictaminar moltes coses. Llavors, un autòmat està format per un conjunt d'estats i directrius per anar d'un estat a un altre, en funció del símbol d'entrada. L'alfabet del input ens dictaminarà quins símbols pot llegir el AF. Conte un estat inicial i un conjunt d'estats d'acceptar. La definició formal diu que un AF està format per una llista de 5 elements: un conjunt d'estats, l'alfabet d'entrada, directrius de moviment, un estat inicial i un conjunt d'estats d'accepta. Matemàticament, direm que un AF està format per una 5-tuple.

Utilitzem algú denominat funció de transició, denotada δ, que ens defineix les directrius de moviment. Si un AF té una fletxa que va d'un estat x a un estat y amb un símbol d'entrada 1, voldrà dir que, si l'autòmat està a l'estat x mentre llegeix el símbol 1, l'autòmat s'ha de moure cap a l'estat y. Podem indicar la directriu com: $\delta(x, 1) = y$.

Vegem la definició formal d'un autòmat finit.

Un autòmat Finit(AFD) és una 5-tupla $(Q,\Sigma, \delta,q_o, F)$, on:

1. Q és un conjunt finit denominat states.
2. Σ és un conjunt finit denominat alfabet.
3. $\delta: Q \times \Sigma \rightarrow Q$ és la funció de transició. $\in$
4. $q_o \in Q$ és l'estat inicial.
5. $F \subseteq Q$ és un conjunt d'estats d'acceptació.

Vegem la representació gràfica d'un AF.

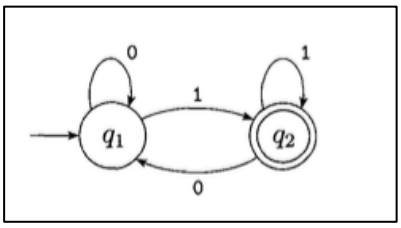


Figura 1 autòmat finit M

Si A és el conjunt de totes les paraules que la Màquina M accepta, llavors podem dir que A és el llenguatge de la Màquina M i ho podem denotar així: *L(M) = A.* Podem dir que M reconeix A o que M accepta A. A partir d'ara direm que un M reconeix A.

La màquina pot acceptar molts mots, però sempre reconeixerà un llenguatge. Si la màquina no accepta mots, llavors vol dir que accepta el llenguatge buit ∅.

Per exemple:

A= {w| w conté mínim un 1 i cap zero després de l'últim 1}

Llavors $L(M_1) = A$, o M_1 reconeix A.

5.2.Autòmat Finit Indeterminista

L'indeterminisme és un concepte molt potent i que té un gran impacte en la teoria de la computació. Quan la màquina està en un estat i llegeix un símbol d'entrada, es mou al altre estat això és determinista. Per tant, AF indeterminista és aquell que te múltiples opcions amb el mateix símbol.

L'indeterminisme és una generalització del determinisme qualsevol AF Determinista és automàticament un AF indeterminista.

La diferència entre un AF determinista, abreviat DFA(en anglès), i un AF indeterminista, abreviat NDA(en anglès), és immediatament aparent. Primer, per cada estat d'un DFA sempre hi ha exactament una única transició per cada símbol de l'alfabet.

Segon, un DFA, les etiquetes de les fletxes de transició son símbols de l'alfabet. Però en el NFA també hi ha el ε. En general, als NFA poden tenir transicions amb membres de l'alfabet o ε. Cap, una o moltes transicions poden ser adjacents a l'estat amb l'etiqueta ε.

Per tant, Com computa un NFA? Suposem que estem executant un NFA amb una input i estem en un estat amb múltiples possibilitats de moviment. Per exemple, direm que estem en el estat q1 en el NFA1 i que el símbol d'entrada és 1. Després de llegir aquest símbol, la màquina es divideix en múltiples còpies d'ella mateixa per tal de seguir totes les possibles vies. Si després ens trobem en el mateix cas, automàticament, fem el mateix. Si la màquina es troba en un cul de sac, o sigui que amb el símbol d'entrada no pot fer res, la màquina mort. Finalment, si alguna de les còpies de la màquina arriba a l'estat d'acceptació i a la fi de l'input, vol dir que aquest NFA accepta el mot d'entrada.

La definició formal d'un autòmat finit indeterminista és la següent:

Un autòmat finit és una 5-tupla $(Q,\Sigma, \delta,q_o, F)$, on:

1. Q és un conjunt finit denominat states.
2. Σ és un conjunt finit denominat alfabet.
3. $\delta: Q \times \Sigma \rightarrow Q$ és la funció de transició. ∈
4. $q_o \in Q$ és l'estat inicial.
5. $F \in Q$ és un conjunt d'estats d'acceptació.

La definició formal de la computació d'un NFA és similar a un DFA. Per tant, i tal com dèiem abans podem dir que un NFA N accepta un mot si podem escriure el mot com una successió de símbols que siguin membres de Sigma i la seqüència d'estats $r_0,r_1,...,r_m$ existeixi a Q i compleixi aquestes 3 condicions:

1. $r_0=q_0$,
2. $r_{i+1} \in \delta(r_i, y_{i+1})$, per i=0,...,m-1, i
3. $r_m \in$ a F.

La condició 1 diu que la màquina comença en l'estat inicial. La condició 2 que per cada estat r i+1 existeix una funció de transició que l'estat destí és membre de Q. Finalment, la condició 3 diu que l'últim estat de la seqüència està contingut en el conjunt d'estats finals.

Els NFA i els DFA són equivalent perquè reconeixen la mateixa classe de llenguatges. Aquesta equivalència és sorprenent i molt útil. És sorprenent perquè els NFA tenen més poder que els DFA, perquè els NFA reconeixen un major nombre de llenguatges. És útil perquè en la programació d'un NFA per certs llenguatges a vegades és més fàcil que la creació d'un DFA per el mateix llenguatge.

Per tant, direm que dos màquines són equivalents si reconeixen el mateix llenguatge.

Podem assegurar llavors que per cada NFA hi ha el seu equivalent DFA.

Això ens porta a explicar el següent apartat.

5.2.1. Determinització

La idea principal és que, si un llenguatge és reconegut per un NFA, podem assegurar de l'existència d'un DFA que també el reconegui. La idea es convertir un NFA en el seu equivalent DFA que simuli el NFA.

Com podem simular un NFA si volem que sigui un DFA? Quin camí ha seguit el NFA per processar el mot? En el exemple un NFA està dividit en moltes branques de computació per cada cruïlla de cada estat. Hem d'actualitzar la simulació unint, afegint i eliminant transicions tal com el NFA opera. El que necessitem, és trobar el conjunt d'estats per tal de mantenir-los.

Si k és el nombre d'estat de NFA, el 2^k és el subconjunt d'estats del DFA resultant. Cada subconjunt correspon a un possible moviment que el DFA ha de tenir, per tant, el DFA que simuli el NFA pot tenir 2^k estats. Ara necessitem assignar l'estat inicial i el conjunt

d'estats finals del nou DFA, i la seva funció de transició. Per tal, d'explicar-ho, ho farem amb notació formal.

Si tenim un NFA N=(Q,Σ, δ, q_o, F) que reconeix un llenguatge A. Construirem un DFA M= que reconeix A. Abans de fer la construcció, considerem que N no te transicions.

1. Q'=P(Q)

   Per cada estat de M hi ha un conjunt d'estat de N. Podem dir que P(Q) és un conjunt de subconjunts de Q.

2. Per E Q' i a sigma hi ha una $\delta'(R,a) = \{ q \in Q \mid q \in \delta(r,a) \; per \; r \in R\}$.
   Que vol dir això, sigui R un estat de M, aquest és un subconjunt de N. Quan M llegeix un símbol a l'estat R, ... Per que cada estat va a un conjunt d'estats, és la unió de tots els conjunt. Una altre forma de dir-ho és:

$$\delta`(R,a) = \bigcup_{r \in R} \delta(r,a).$$ [1]

3. $q'_o = \{q_o\}$
   M comença en el estat corresponent a la col·lecció que conté el estat inicial N.

4. $F' = \{R \in Q' \mid R \; conté \; un \; estat \; final \; de \; N\}$.
   La màquina M accepta si un dels estats d'algun dels subconjunt de N és un estat final.

En el supòsit anterior, no hem tingut en compte les ε-transicions. Per cada estat R de M definim *E(R)* que és la col·lecció d'estats que des de R podem anar només amb les ε, incloent els bucles . Formalment, $R \subseteq Q$

$$E(R) = \{q \mid q \; que \; podem \; anar \; des \; de \; R \; a \; traves \; de \; 0 \; a \; moltes \; \varepsilon \; transicions\}$$

Dit això haurem de modificar la funció de transició de M afegint els camins cap als estats que tingui ε-transicions. Reemplaçant $\delta(r,a)$ per $E(\delta(r,a))$.

Llavors un llenguatge és regular si i només si almenys un NFA el reconeix. Això ens diu que, un llenguatge és regular si hi ha un NFA que el reconeix, per tant, i tal com dèiem amb anterioritat, tot NFA té un DFA equivalent i podem acabar dient que és condició necessària

[1] La notació $\cup_{r \in R} \delta(r,a)$. significa: la unió del conjunt d'estats $\delta(r,a)$ per cada possible r in R

però no suficient que aquest llenguatge Regular com a mínim hi haurà un DFA que el reconegui.

A la següent figura tenim un NFA i veurem el seu DFA equivalent:

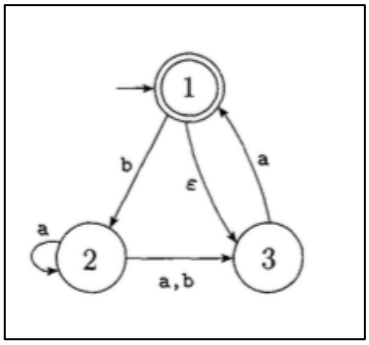


Figura 2: NFA

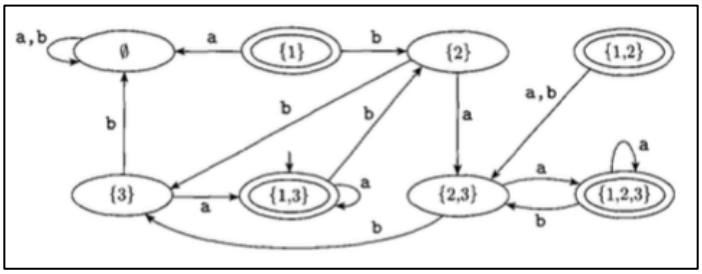


Figura 3 DFA

Tal com hem dit, els canvis són significatius, hem passat de tenir un NFA amb 3 estats a un DFA amb 2^3=8, han desaparegut les ε-transicions i ara cada estat té una destinació per cada símbol del alfabet. També podem veure que cada estat és un conjunt d'estats finit, i que els estats a on està comprés l'estat final del NFA, en el DFA són estats finals.

Tal com veiem segurament podrem reduir el nostre DFA, ja que hi ha molts estats que fan el mateix o similar. La idea serà semblant a quan hem passat de NFA a DFA mantenir la idea principal de la computació.

5.2.2. Minimització

Tal com hem vist en el capítol anterior, obtenir un DFA a partir d'un NFA ens genera un autòmat molt gran, tal com dèiem en els NFA podem tenir dos NFA que reconeixien el mateix llenguatge, doncs podem tenir dos DFA que ens reconeguin el mateix llenguatge.

La solució per tant, serà a partir d'un DFA, obtindrem un DFA equivalent amb el mínim nombre d'estats. Per tal de minimitzar el DFA hi ha 3 algoritmes:

- Hopcroft's algoritme
- Moore's algoritme
- Brzozowski's algoritme

Explicarem el Hopcroft, ja que és l'utilitzat en l'eina.

L'algoritme de Hopcroft és basa en el refinament de particions, o sigui, agrupar els estats del DFA en grups en funció del seu comportament per totes les seqüencies d'entrada. És a dir, per cada dos estats p_1 i p_2 que pertanyen al mateix grup d'equivalència dins de la partició P, i cada paraula d'entrada w, les transicions que determini w sempre han de prendre els Estats p_1 i p_2 a estats iguals, afirma que tant acceptar, o estableix que tots dos rebutgen. No hauria de ser possible per als w a prendre p_1 a un estat d'acceptació i p_2 a un estat rebutjar o viceversa.

L'algorisme comença amb una partició ruda: cada parell d'estats que són equivalents segons la relació Myhill-Nerode pertanyen al mateix conjunt en la partició, però els parells que són no equivalents també podria pertànyer al mateix conjunt. És refina gradualment la partició en un major nombre de conjunts més petits, en cada pas de divisió de conjunts d'estats en parells de subconjunts que necessàriament no són equivalents. La partició inicial és una separació dels estats en dos subconjunts d'estats que clarament no tenen el mateix comportament que l'un a l'altre: els estats d'acceptació i rebuig dels estats. L'algorisme escull llavors repetidament un conjunt A de la partició actual i un símbol d'entrada c, i es divideix cada un dels conjunts de la partició en dos subconjunts (possiblement conjunt buit): el subconjunt d'estats que condueixen a A en símbol d'entrada c, i de la subconjunt d'estats que no condueixen a A. Com A ja se sap que té un comportament diferent al dels altres conjunts de la partició, els subconjunts que condueixen a A també tenen un comportament diferent al dels subconjunts que no condueixen a A. Quan no més divisions d'aquest tipus es poden trobar, l'algoritme acaba.

Lema. Donat un caràcter c fixa i una Y classe d'equivalència que es divideix en classes d'equivalència B i C, només un de B o C és necessària per refinar tota la partició.

Exemple: Suposem que tenim una classe Y equivalència que es divideix en classes d'equivalència B i C. Suposem també tenim classes D, E i F; D i E tenen estats amb

transicions a B en el caràcter c, mentre que F té transicions en C el caràcter c. Pel Lema, podem triar entre B o C, segons els partidor, diguem B. A continuació, els estats de D i E es divideixen per les seves transicions en B. Però F, que no apunta cap a B, simplement no el dividim durant la iteració actual del algoritme; serà refinat per una altra partició.

El propòsit és "outermost" si la condició (si Y és a W) és apedaçar W, el conjunt de distingits. Ens veiem a la declaració anterior en l'algoritme que Y només s'ha dividit. Si Y és en W, que només ha quedat obsoleta com un mitjà per dividir les classes en iteracions futures. Així que Y ha de ser substituït per les dues fractures a causa de l'observació anterior. Si Y no està en W, però, només una de les dues divisions, no tots dos, ha de ser afegit a W causa del Lema anteriorment. L'elecció de la més petita de les dues divisions garanteix que la nova addició a W no és més que la meitat de la mida de I; aquest és el nucli de l'algoritme Hopcroft: el temps és el principal problema.

En el pitjor dels casos, el temps de l'algoritme és *O(ns log n)*, a on *n* és el nombre d'estats i *s* el nombre de símbols del alfabets. Per cada cicle que hi ha partició el temps és *O(log n).* L'estructura de les dades en la partició del refinament permet a cada pas de divisió que el temps sigui proporcional al nombre de transicions que hi intervenen. Això fa que sigui l'algoritme més eficaç conegut per resoldre aquest problema.

5.2.3. Autòmat Finit a Expressió Regular

Una expressió regular i un autòmat finit són equivalents en la seva potent descripció. Aquest fet és sorprenent perquè els autòmats finits i les expressions regulars semblen superficialment diferents. No obstant això, qualsevol expressió regular es pot convertir en un autòmat finit que reconeix un llenguatge que descriu, i viceversa. Recordem que un llenguatge és regular si és reconegut per algun autòmat finit.

Per tal de fer aquest pas, hem d'utilitzar un nou tipus d'autòmats finits, els "generalized nondeterministic finite automaton," GNFA. Primer expliquem com convertir de DFA a GNFA, i després explicarem el pas de GNFA a expressió regular.

Per passar de DFA a GNFA, tenim un DFA D i crearem un GNFA G, el primer que fem és afegir dos estats a G, un estats inicial q_{ini} el qual serà el estat inicial de G, i que enllaçarem amb l'estat inicial de amb una transició ε. Afegirem un altre estat, q_{fi}, el qual serà el nou estat final de G, i enllaçarem tots els estats finals de D amb q_{fi} mitjançant ε-transicions.

I afegirem tots els estats i transicions de D a G, per tant, si D és un DFA amb 3 estats, el nostre GNFA equivalent, ha de tenir 5 estats i totes les transicions de D mès les afegides per enllaçar q_{fi} i q_{init}.

Un cop tenim el nostre GNFA, el pas a expressió regular és ben simple, en la seva idea. Per tal de dur a terme la conversió, anem eliminant un estat cada vegada fins a que només quedin els estats q_{fi} i q_{init} que hem afegit anteriorment.

Per eliminar un estat ho fem de la següent manera:

Seleccionem un estat q_{rip}, l'eliminarem de la màquina i repararem el que queda alterant-ne les expressions regulars de les fletxes per tal que el llenguatge sigui reconegut. Les fletxes alterades compensaran l'absència de q_{rip}. El nou valor de la fletxa que va de qi a qj és una expressió regular que descriu tots els string que porten la màquina de qi a qj directament o a traves de q_{rip}. D'aquest procediment en direm CONVERT(G) al qual rep un G i el converteix en l'equivalent expressió regular. Aquest procediment utilitza la recurssió, cosa que significa que és crida a ell mateix fins que el GNFA té només dos estats.

El procediment serà el següent:

1. Considerem *k* el nombre d'estats de G.
2. Si *k*=2, G té un estat inicial, un acceptable i una única fletxa de transició amb valor R. Retorna R.
3. Si *k*>2, es selecciona qualsevol estat $q_{rip} \in Q$ diferent de q_{start} i q_{accept}. Considera que G' és el GNFA $(Q', \sum, \delta', q_{start}, q_{accept})$ on $Q'=Q-\{ q_{rip} \}$ i per qualsevol $q_i \in Q'-\{q_{accept}\}$ i qualsevol $q_J \in Q'-\{q_{start}\}$ considerem $\delta'(q_i, q_J)=(R1)(R2)^*(R3) \cup (R4)$ per $R1=\delta(q_i, q_{rip})$, $R2= \delta(q_{rip}, q_{rip})$, $R3= \delta(q_{rip}, q_J)$ i $R4= \delta(q_i, q_J)$.
4. Cridar CONVERT(G') i retornar aquest valor.

5.2.4. Expressions Regulars

Per construir expressions que descriuen llenguatges podem utilitzar les operacions regulars anomenades expressions regulars(RE). El valor de les expressions regulars és un llenguatge. Quan volem distingir entre una expressió regular R i el llenguatge que aquesta descriu, escriurem L(R) com el llenguatge de R. Vegem-ne la definició formal.

Direm que R és una expressió regular si:

1. a per qualsevol a és de l'alfabet Σ,
2. ε,
3. ∅,
4. $(R_1 \cup R_2)$, a on R_1 i R_2 són expressions regulars,
5. $(R_1 \circ R_2)$, a on R_1 i R_2 són expressions regulars,
6. (R_1^*), a on R_1 és una expressió regular.

Els items 1 i 2, l'expressió regular a i ε representen els llenguatges que contenen {a},{ ε}, respectivament. I l'item 3, l'expressió regular ∅, representa el llenguatge buit. Els items 4,5 i 6, que les expressions obtingudes representen els llenguatges obtinguts de fer la unió o la concatenació de R_1 i R_2 o d'aplicar l'estrella a R_1, respectivament.

Descripció 1 Expressió Regular

Vist la descripció formal, podem veure que hi ha certa similitud entre els NFA i les ER, ja que tenen les mateixes operacions, i que el resultat de fer alguna d'aquestes operacions, sempre en resultat un membre de les mateixes. Hem explicat com passar de NFA a RE, en el següent apartat expliquem el pas invers.

5.2.5. Expressió Regular a NFA.

Tal com hem dit anteriorment, un llenguatge és regular si hi ha un NFA que el reconegui, i un llenguatge és regular si hi ha una expressió regular que el descriu, per tant, és ben visible la connexió. La idea llavors serà:

Si tenim una expressió regular R que descriu un llenguatge A. Nosaltres podem convertir R en un NFA que reconeixi A. I com hem dit abans, si el NFA reconeix A llavors A és regular.

Com que és un conversió força simple dibuixem les respectives conversions, i la idea llavors serà substituir cada expressió regular per el seu equivalent NFA.

Per fer-ho convertirem l'expressió regular $(ab \cup a)^*$ en el seu NFA seqüencialment.

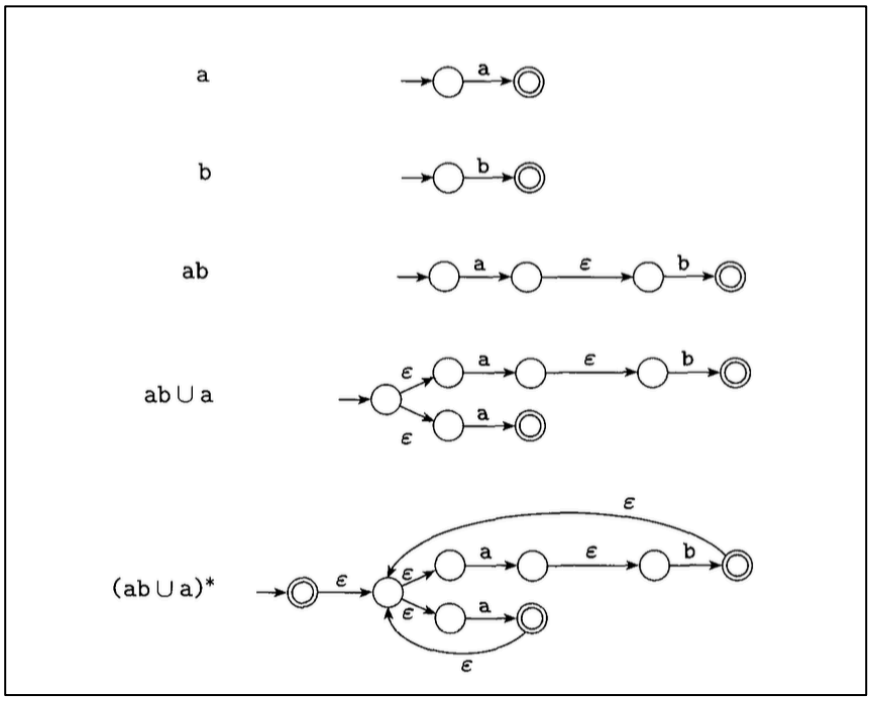


Figura 2 RE a NFA

5.3.Autòmats de Pila

Aquest model es similar al model autòmat Finit no Determinat per amb un component extra anomenat *Stack*. Aquest component allarga la memòria més a allà del límit finit disponible en el control. L'*Stack* permet al autòmat de Pila reconèixer alguns llenguatges poc usuals.

El autòmat de Pila(PDA) és una mena de gramàtica fora de context, aquesta similitud es útil perquè proporciona dos opcions per provar el llenguatge fora de context. Alguns llenguatges son més fàcils de descriure en termes de generadors, i per altra banda n'hi ha que són més fàcils per termes de reconeixedors.

La següent figura es una representació sistemàtica d'un autòmat finit. El control representa l'estat de funció transitòria, la cinta conté les entrades i la fletxa representa l'entrada del "cap" indicant quin és el següent símbol que llegirà la màquina.

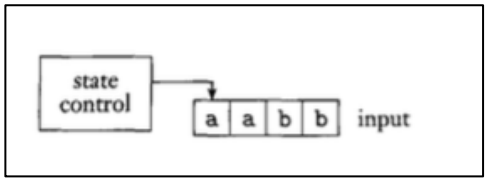


Figura 3 Esquema d'un autòmat de pila

L'autòmat de Pila pot escriure símbols en l'Stack i llegir-los més endavant, escrivint un símbol de "pushes down" davant de tots els altres símbols. D'aquesta manera en qualsevol moment que el símbol estigui al capdamunt de l'Stack, pot ser llegit i eliminat. D'aquesta manera els altres símbols es mouran cap a dalt. Escriure un símbol a la pila es usualment referit com sorgiment del símbol i l'eliminació d'un símbol es coneix com "popping it".

Això ens fa veure que l'autòmat de Pila treballa amb el sistema "Last in first out" que significa que l'últim símbol en entrar serà el primer en sortir, per exemple, si introduïm informació i més tard ampliem la informació afegint-ne més, la primera entrada d'informació no serà llegida fins que no s'hagi eliminat la informació que hem posat al ampliar.

Per exemple en una cafeteria, quan apilem els plats per rentar, cada cop que afegim un plat, el primer plat que havíem posat esta avall de tot i cada cop hi han més plats per rentar per arribar al primer que hem posat, el sistema de l'autòmat de pila és similar a la pila de plats per rentar, cada plat representa cada símbol que escrivim.

L'Stack, és a dir, una pila es valuosa perquè pot tenir il·limitada informació. Cal remarcar que l'autòmat finit és incapaç de reconèixer el llenguatge $\{0^n 1^n \mid n \geq 0\}$ perquè no pot guardar una infinita quantitat de símbols. Una PDA és capaç perquè pot utilitzar la pila de números de 0_s que ha vist anteriorment. A més a més la il·limitada natura de la pila deixa els números de la PDA il·limitats.

Descripció formal de l'autòmat de Pila

La Descripció formal de l'autòmat de Pila és similar a la de l'autòmat finit, amb una única excepció; la Pila. La Pila facilita que hi hagin símbols de diferents alfabets ja que aquesta màquina usa diferents alfabets d'entrada, per aquest motiu podem especificar que l'alfabet d'entrada com $\sum$ i l'Estaco (Pila) com Γ.

La clau per realitzar una definició formal és la funció de transició de la màquina que ens mostrarà com es comporta i actua. Saben que $\sum_\epsilon = \sum_U \{\epsilon\}$ i Γ_ϵ = U{ϵ}. La funció de transició és Q x $\sum_\epsilon$ x Γ_ϵ per tant, el següent símbol d'entrada i el primer símbol de la pila determinarà el següent moviment de la màquina. Si qualsevol dels dos símbols és ϵ la màquina es mourà sense llegir un símbol d'entrada o un símbol de la pila.

Per el rang de la funció de transició necessitem veure la màquina en una situació particular. La funció δ pot indicar que tornarà a Q juntament amb un membre de Γ, per tant serà un membre de Q x Γ_ϵ . La funció de transició ho incorpora en la usual via, retornant Q x Γ_ϵ com a membre de $P(Q$ x $\Gamma_\epsilon)$. Si ho ajuntem tot quedaria de la forma següent: δ: Q x $\sum_\epsilon$ x Γ_ϵ -> $P(Q$ x $\Gamma_\epsilon)$.

L'autòmat de pila compta amb 6 components $(Q, \sum, \Gamma, \delta, q_0, F)$. Dels quals Q, $\sum$,Γ, i F són components finits.

1. Q és un conjunt finit denominat states.
2. Σ és un conjunt finit denominat alfabet.
3. Γ és l'alfabet de Pila
4. δ: Q x $\sum_\epsilon$ x Γ_ϵ -> $P(Q$ x $\Gamma_\epsilon)$ és la funció de transició
5. $q_0 \epsilon Q$ es l'estat d'entrada
6. $F \epsilon Q$ és l'estat dels components acceptats.

Descripció 2 Autòmat de Pila

Un autòmat de Pila M= $(Q, \sum, \Gamma, \delta, q_0, F)$ computa de la manera següent. Primerament accepta l'entrada w si $w= w_1 w_2 \ldots\ldots w_n$ on cada $w_n \epsilon \sum_\epsilon$ i la seqüència dels estats $r_0, r_1 \ldots\ldots r_m$ ϵ Γ* existeix que satisfà les tres següents condicions:

1. $r_0 = q_0$ i $s_0 = \epsilon$. Aquesta condició significa que M ha començat correctament.
2. Per $i = 0,..., m - 1$ tenim $(r_{i+1}, b) \in \delta(r_1, w_{i+1}, a)$ en el qual $s_i = at$ i $s_{i+1} = bt$ per algun $a,b, \epsilon, \Gamma_\epsilon$ i $t \in \Gamma^*$. Aquesta condició expressa que M es mou correctament en l'estat, la Pila i el següent simbol.
3. $r_m \in F$. Aquesta condició expressa que ha estat acceptat al final de l'entrada.

Vegem un exemple de la representació gràfica d'un PDA:

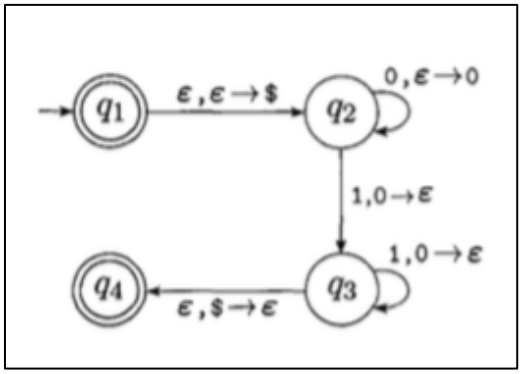


Figura 4 Esquema d'un PDA

5.4.Màquina de Turing

La següent llista resumeix les diferencies entre l'autòmat finit i la Màquina de Turing(MT):

1. Una màquina de Turing pot escriure i llegir el que ha escrit.
2. L'habilitat d'escriure i llegir pot ser cap a la dreta i cap a l'esquerra.
3. El número de caràcters és infinit.
4. Els estats especials per rebutjar i acceptar són immediats.

Si introduïm a la màquina de Turing M_1 per les proves en el llenguatge B = {w#w| w € {0,1}*}. Volem saber si M_1 és un membre de B i rebutjar la resta de caràcters. Per entendre millor M_1 hem d'imaginar-nos que ens trobem en un lloc on hi ha milions de caràcters i les teves habilitats determinaran si ets membre de B. L'entrada de caràcters és molt llarga per marcar-ho de començament però la màquina permet anades i vingudes a l'entrada per marcar els caràcters. La estratègia més obvia és fent zig-zag corresponent els llocs dels dos costats de # i determinar si els caràcters compleixen les ordres. El costat (dret o esquerra) de # marca a quin lloc correspon el caràcter.

Dissenyem M_1 per posar-ho a la pràctica. Això comporta múltiples passades pel sobre del cap de lectura-escriptura. En cada passada marca un dels caràcters a cada costat del símbol #. Pel seguiment de quins símbols han estat revisats M_1 els marca com examinats. Si tots els símbols estan examinats significa que tot ha funcionat perfectament, i M_1 queda acceptada. Si es detecta algun error, M_1 quedarà en l'estat rebuig. En conclusió, M_1 és l'algoritme següent:

M_1 = és cadena d'entrada W.

1. El moviment zig-zag acciona sobre els dos costats del símbol # per revisar quines posicions conten el mateix símbol. Si les posicions no contenen el mateix símbol o si no es troba #, passen a ser rebuig. Els símbols marcats són els que han estat revisats
2. Quan tots els símbols de la esquerra de # han estat revisats, comença a revisar tots els símbols de la dreta de #. Si hi ha algun símbol erroni, passa a ser rebuig, del contrari, passa a ser acceptat.

Passem a veure l'exemple de M_1 mentre la computació de les estades 2 i 3 quan comença l'entrada 011000#011000.

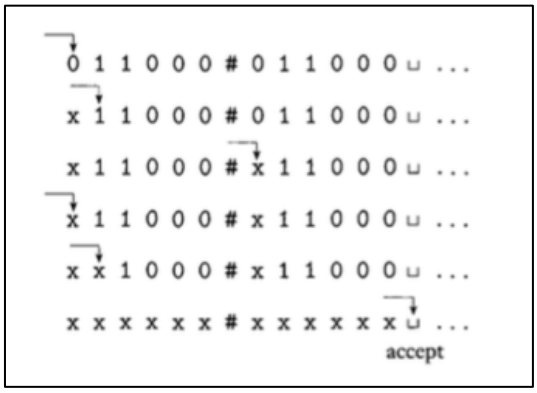


Figura 5 Esquema de la computació

El punt clau de la definició de la màquina de Turing és la funció de transició δ perquè ens indica com la màquina passa d'un pas al següent. Per la màquina de Turing, δ agafa la forma $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\}$. És llavors quan la màquina està en estat q i el seu cap està sobre del símbol a, i si $\delta(q,a) = (r, b, L)$, la màquina escriu el símbol b reemplaçant l' a i passa al estat r. El tercer component L o R qualsevol dels dos, indica si els moviments van cap a la dreta o esquerra. En aquest cap, la L indica que els moviments van cap a l'esquerra. Vegem-ne l'expressió formal:

Una Màquina de Turing(MT) és una 7-tuple$(Q, \Sigma, \Gamma, \delta, q_o, q_{accept}, q_{reject})$,

on Q, Σ, Γ, sòn conjunts finits, i

1. Q és un conjunt finit denominat states,
2. Σ és l'alfabet d'entrada que no conté el símbol en blanc ⊔,
3. Γ és l'alfabet de la pila, a on $\sqcup \in \Gamma$ i $\Gamma \subseteq \Sigma$,
4. $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times (L \times R)$ és la funció de transició,
5. $q_o \in Q$ és l'estat inicial,
6. $q_{accept} \in Q$ és l'estat d'accepta, i
7. $q_{reject} \in Q$ és l'estat de rebutjar, i $q_{accept} \neq q_{reject}$

Descripció 3 Màquina de Turing

Una Màquina de Turing computa de la següent manera: Primerament M rep l'entrada $w = w_1 w_2 \ldots\ldots w_n \epsilon \sum^*$ a les places de l'esquerra de tot indica que hi ha escriptura, la resta de caràcters son Blank symbols. És a dir, són caràcters buits.

La màquina comença a actuar pel primer símbol de l'esquerra, després revisa que ∑ no conté cap símbol blanc. Un cop *M* ha començat, la màquina actua com s'ha descrit a la definició. Si a la formula posa *L* la màquina començarà a examinar per l'esquerra. La computació continua fins que tots els símbols hagin estat classificats a q_{accept} o q_{reject}. Si això no és produeix *M* passarà a ser seguint.

Tal com computa la màquina de Turing, els canvis apareixen en l'estat corrent, la corrent cinta conté i la corrent localització del *cap* de la màquina. Els ajustaments d'aquests tres ítems s'anomenen una *configuració* de la màquina de Turing. Les configuracions usualment indiquen una via especial. Per un estat *q u* i *v* sobre de l'alfabet Γ escrivim *u* i *v* per la configuració on l'estat actual es *q*, la cinta conté *uv*, i el "cap" de la màquina es troba al primer símbol de *v*. La cinta conté només símbols blancs que segueixen el últim símbol de *v*. Per exemple: $1011q_701111$ representa que la configuració quan la cinta es 101101111, l'estat actual es q_7 i el "cap" es troba al segon 0.

La següent imatge indica com actua la màquina de Turing amb una configuració

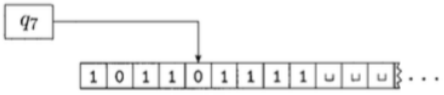


Figura 6 Una Màquina de Turing amb la configuració $10011_{q_7}0111$

Vegem la representació gràfica d'una Màquina de Turing:

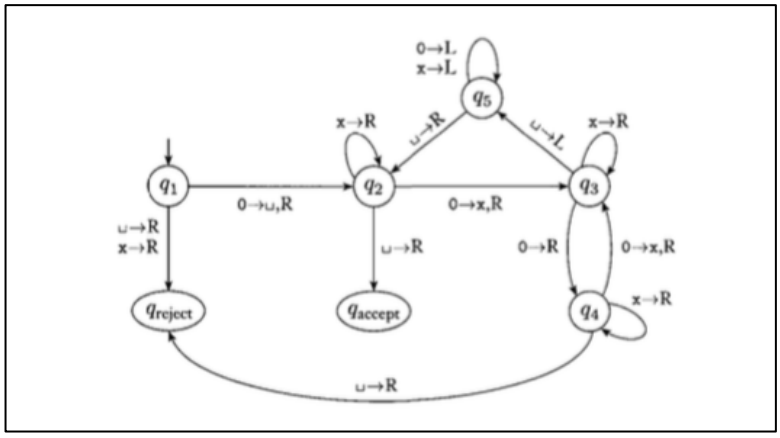


Figura 7 Diagrama d'estats d'una Màquina de Turing M_2

Exemples de la Màquina de Turing

Com hem vist anteriorment amb els autòmats, podem descriure formalment una màquina de Turing especificant cada una de les 7 parts que conté. A més a més no necessitem gaire temps per cada descripció, més que res hi ha un gran nombre de descripcions perquè sigui més fàcil d'entendre. Encara que és important recordar que cada explicació detallada és actualment una ajuda per la contrapart forma. Variants de la màquina de Turing

7.1.1. ETM

Els enumeradors és una variant de la màquina de Turing que va juntament amb una impressora. La màquina de Turing pot utilitzar la impressora com a sortida per imprimir les cintes. Cada cop que la màquina vol afegir informació a la llista s'envia la cinta a la impressora.

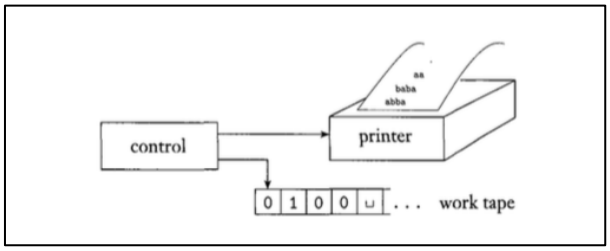


Figura 8 Esquema d'un enumerador

Un enumerador *E* comença amb una entrada blanca, si l'enumerador no es para pot imprimir una llista infinita de cadenes. El llenguatge enumerat per *E* es la col·lecció de totes les cintes que han estat impreses. Per altra banda *E* genera les cadenes del llenguatge en qualsevol ordre, possiblement amb repeticions.

7.1.2. NTM

Una màquina de Turing no determinista és definida de la següent manera: En qualsevol punt en el càlcul de la màquina pot actuar en diferents vies. La funció de transició d'una màquina no determinista de Turing té la següent forma:

$$\delta: Q \times \Gamma \rightarrow P(Q \times \Gamma \times \{L,R\}).$$

La computació d'una màquina no determinista de Turing conté tres branques de possibilitats. Funciona de forma semblant que una autòmat finit nodeterminista.

7.1.3. MTM

La màquina de Turing multicinta és com una màquina de Turing normal però amb diferents cintes, i cada una d'aquestes cintes té el seu propi control de lectura-escriptura. Inicialment l'entrada comença per la cinta 1 i totes les demés estan en blanc. La funció de transició es diferent per permetre la lectura i l'escriptura movent el control per alguna cinta o per totes simultàniament. És a dir:

$$\delta: Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L,R,S\}^k$$

En la qual K és el número de cintes. L'expressió:

$$\delta\ (\ q_1,a_1,\ldots,a_k) = (q_j,b_j,\ \ldots,\ b_k,\ L,\ R,\ldots L)$$

Significa que si la màquina esta en l'estat q_i i el control 1 està llegint els símbols a_i pensant a_k, la màquina va a l'estat q_j escriu els símbols b_1 pensant b_k i directament cada cap de control es mou a l'esquerra o a la dreta o es queda quiet com està especificat.

La màquina multicinta de Turing és més poderosa que l'ordinària màquina de Turing, però podem demostrar que tenen un poder equivalent ja que les dues poden reconèixer el mateix llenguatge.

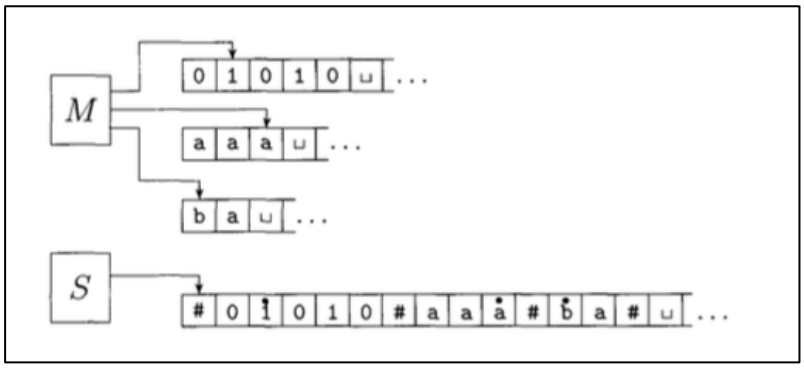


Figura 9 Representació de tres cintes amb una

En la figura anterior podem veure dues màquines equivalents, la primera M conté 3 cintes per tal de fer-hi les operacions corresponents. La segona S, conté una única cinta, però "trossejada" en 3 les quals estan separades pel símbol "#". A la màquina S tenim 3 índex, simbolitzats amb · els quals és mouen entre els símbols "#".

8. Requisits del sistema

La plataforma l'anomenarem Java Universal Turing Machine(d'ara endavant JUTM). Per poder treballar amb el JUTM, l'usuari necessita d'un computador, o bé un smartphone. És necessària la connexió a internet per qualsevol dispositiu. És possible treballar amb el JUTM mitjançant aquests dispositius ja que el sistema s'ha adaptat a pantalles de 800x600 i la majoria dels dispositius mòbils arriben o superen aquesta resolució.

L'únic requisit indispensable és un navegador web. S'ha provat la seva utilització en els següents i el JUTM a funcionat correctament:

- Chromium (linux)
- Google Chrome (OSX, Windows)
- Mozilla Firefox (OSX, Window, Linux)
- Safari (OSX, Win)
- Internet Explorer 10 i superiors

Si qualsevol d'aquest està instal·lat, el JUTM és podrà executar al navegador. No requereix cap llibreria Java addicional o algun altre programari, ja que ve tot integrat en el seu "core".

La direcció web per executar-lo és: http://jutm.elasticbeanstalk.com/

9. Estudis i decisions

9.1.UML.

L'UML o Llenguatge Unificat de Modelat (*Unified Modeling Language,* Llenguatge de Modelat Unificat) és un llenguatge de modelat de sistemes de software, és el més conegut i utilitzat en l'actualitat, està suportat per l'OMG (Object Management Grup). És un llenguatge gràfic per visualitzar, especificar, construir i documentar un sistema. L'UML ofereix un estàndard per descriure un sistema (model), incloent aspectes conceptuals tals com els processos de negoci i funcions del sistema, i aspectes concrets com expressions de llenguatges de programació, esquemes de bases de dades i components reutilitzables.

El Visual Paradigm per UM (VP-UML) és una eina CASE UML que suporta UML 2. M'ofereix una ajuda en el disseny dels casos d'us i el diagrama de classes per tal de fer el disseny abans de començar a programar. Permet la creació de classes i mètodes des d'un diagrama de Classes.

9.2.Java

El Java és un llenguatge de programació dissenyat el 1990 per James Gosling amb altres companys de Sun Microsystems a partir de C++. Des del seu naixement fou pensat com un llenguatge orientat a objectes. Entre el 13 de novembre de 2006 i el maig del 2007 Sun va alliberar parts de Java com a programari lliure de codi obert amb llicència GPL. És un dels llenguatges de programació més utilitzats, i s'utilitza tant per aplicacions web com per aplicacions d'escriptori.

El Java és un llenguatge interpretat i, per tant, pot semblar lent en comparació amb altres llenguatges, però ofereix un índex de reutilització de codi molt elevat, sent possible trobar moltes llibreries lliures de *Java*. És un llenguatge flexible i potent tot i la facilitat amb la qual es programa i dels resultats que ofereix. Un dels trets que el caracteritza i que el fa una eina molt valorada a l'hora de desenvolupar aplicacions distribuïdes, és el fet que és un llenguatge multiplataforma.

Per desenvolupar l'aplicació s'ha utilitzat el IntelliJ Idea, que conté totes les funcionalitats necessàries per crear aplicacions Web amb Java. És un entorn de desenvolupament molt còmode que té totes les eines necessàries integrades, des del Maven per mantenir les llibreries i l'estructura del projecte, fins a l'execució de la Web App sobre qualsevol servidor del tipus JBoss, prèviament configurat.

Per tal de controlar la implementació de l'aplicació s'ha utilitzat un control de versions, el Git. El Git és un sistema de control de versions pensat en l'eficiència i confiabilitat de manteniment de versions d'aplicacions amb una enorme quantitat de fitxers de codi font.

Per tal de configurar el projecte amb un servidor del tipus JBoss, s`ha utilitzat el Glassfish. Les llibreries utilitzades són:

- **JUNG:** una llibreria completa per tal de visualitzar qualsevol tipus de dades que pugui ser representat com un graf o una xarxa.
- **Apache Commons:** és la llibreria utilitzada per tal de penjar els fitxers dels usuaris.

9.3.Java Server Faces

Java Server Faces és un marc de treball per aplicacions web basades en Java que simplifica el desenvolupament d'interfícies d'usuari per aplicacions Java EE. **JSF** va utilitzar JavaServer Pages (JSP) com a tecnologia per fer el desplegament de les pàgines (però també podia utilitzar altres tecnologies, com per exemple XUL) però des de l'aparició de la versió 2.0 la tecnologia oficial per representar les pàgines és Facelets tot i que es continua donant suport a les tecnologies anteriors.

Es va començar a desenvolupar el 2001 i la primera versió va sortir el març del 2004. Des de llavors han anat apareixent diferents versions noves que n'han simplificat i millorat el desenvolupament i s'ha acabat convertint en la forma estàndard de crear aplicacions Java EE

JSF està format per tres elements bàsics:

- Un conjunt de components prefabricats per dissenyar entorns d'usuari
- Un model de desenvolupament basat en events. Té tot el codi necessari per gestionar els events que es produeixin en l'aplicació web
- Un model de components que permet desenvolupar components addicionals

En general:

- Proporciona desenvolupament d'interfícies web independents del client basat en components.
- Està pensat per funcionar amb el paradigma MVC (Model-Vista-Controlador)
- Simplifica l'accés i la gestió de les dades des de la web

- Gestiona l'estat de la interfície entre múltiples peticions i clients de forma no intrusiva
- Proporciona un entorn de desenvolupament amigable per molts desenvolupadors amb suport per Ajax.

Per tal d'utilitzar els JSF s'ha utilitzat la llibreria de components Primefaces.

9.4.Primefaces

El Primefaces és una suite de components open source JSF amb varies extensions:

- Conjunt extens de components(HTMLEditor, Dialog, AutoComplete,Grafics I molts altres)
- Construit amb l'estàndard Ajax
- Lleuger, amb un jar, sense configuració prèvia I sense cap dependència.
- Suport a través del "Atmosphere Framework".
- Mobile UI kit per crear aplicacions web per a mòbils.
- Més de 35 temes, suport a l'eina de disseny de nous temes.
- Extensiva documentació.
- Gran, vibrant i activa comunitat d'usuaris
- Desenvolupat amb passió per desenvolupador d'aplicacions per a desenvolupadors d'aplicacions.

9.5.GlassFish

El glassfish és un servidor d'aplicacions open-source per a la plataforma Java EE.

Glassfish és la referència en la implementació de Java EE i que dona suport a "Enterprise JavaBeans", JPA, JSF, JMS, RMI, JSP, servlets, etc. Això permet als desenvolupadors crear "Enterprise aplicacions" que sigui portables i escalables, i la seva integració amb velles tecnologies.

Construit sobre un nucli modular alimentat per OSGI, el glassfish s'executa en la part superior de la implementació de l 'Apache Felix. Pot funcionar amb el Equinox OSGi or Knopflerfish OSGI. HK2 abstreu el sistema de mòduls DOSGI per proporcionar component, que també poden ser vistos com a serveis. Aquest serveis poden ser descoberts i s'injecten en temps de desenvolupament.

9.6.Amazon web Services

A l'hora de decidir on allotjar el servidor es va fer una cerca per trobar quina era la millor opció. L'objectiu era que el cost fos el mínim (a poder ser gratuït) ja que es tracta d'una eina didàctica sense ànim de lucre.

Finalment es va optar per crear una compte de desenvolupador d'AWS (Amazon Web Services), que et dóna la possibilitat d'utilitzar algunes de les seves eines de manera gratuïta durant el primer any. Entre aquestes eines hi ha l'ElasticbeanStack que crea una instància preconfigurada de Linux, amb els servidors d'aplicacions web més utilitzats actualment. La instància preconfigurada és escalable i et permet el balancejant entre múltiples instàncies.

El servei gratuït d'Amazon inclou altres serveis que podrien haver estat interessants per al projecte, com l'RDS per emmagatzemar bases de dades, però el problema d'això és que comportaria lligar el servidor a utilitzar aquest sistema sempre, o haver de fer canvis al codi per adaptar-lo.

10. Anàlisi i disseny del sistema

10.1. Anàlisi del sistema

El disseny del Sistema, és força tancat en si mateix, o sigui, tal com hem dit anteriorment, l'objectiu principal del treball és fer una Web App en la qual l'usuari només hagi de penjar el seu programa, sota una codificació especial, i pugui computar el autòmat corresponent, triant en tot moment quins flags vol tenir activats.

Per l'altre banda hi ha la possibilitat d'efectuar les rutines d'equivalència dels NFA/DFA. Això no obstant només indica que l'usuari ha penjat un NFA o DFA i vol efectuar alguna de les possibles rutines. Dit això, per fer l'anàlisi dels requeriments s'utilitzarà el diagrama de cas d'us, conjuntament amb les seves fitxes corresponents.

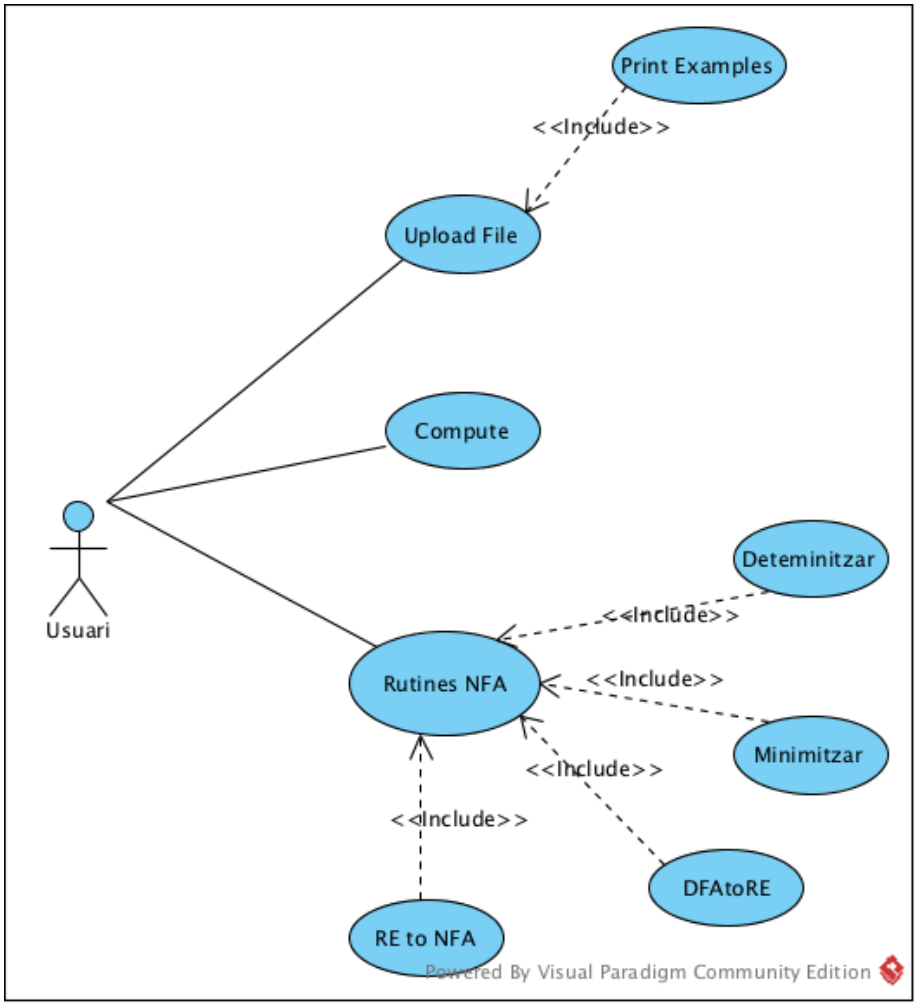


Figura 10 Diagrama de Cas d'us

Cas d'us	**Upload**
Descripció:	L'usuari puja el seu autòmat codificat.
Actor:	Anònim
Precondició:	Cap

Flux Principal:

1. L'usuari visita la web
2. L'usuari codifica el seu autòmat amb l'estàndard preestablert
3. Toca el botó de "upload File".
4. El usuari cerca i penja el seu codi al sistema.
5. El sistema llegeix el fitxer, traduint-lo a string.
6. El sistema construeix el autòmat entrat
7. El sistema dibuixa el autòmat a la pantalla.

Fluxos Alternatius:

6. En cas que no estigui ben codificat retorna error de la línia a on falla.

PostCondició.	L'usuari visualitza el seu autòmat.

El fet de dibuixar el autòmat fa que a la vegada i per obligació del codi, fem un parser del programa entrat per a l'usuari.

Cas d'us	**Dibuixar un Exemple**
Descripció:	L'usuari dibuixa un autòmat dels que hi per exemple
Actor:	Anònim
Precondició:	Cap

Flux Principal:

1. L'usuari visita la web

2. L'usuari cerca un autòmat en els exemples
3. Toca el botó de "print"
4. El Sistema dibuixa l'autòmat seleccionat

Fluxos Alternatius:

PostCondició.	L'usuari visualitza un autòmat del conjunt d'exemples.

Cas d'us	**Computar autòmat**
Descripció:	L'usuari computa l'autòmat que actualment està visualitzat
Actor:	Anònim
Precondició:	Hi ha un autòmat visualitzat

Flux Principal:

1. Selecciona el grup de "flags".
2. Tria sota quins flags vol efectuar la computació.
3. Selecciona el grup de "compute".
4. Entra el seu input en l'espai.
5. Selecciona quina informació vol que hi hagi a l'output.
6. Clica al botó compute.
7. El sistema recull l'input, llegeix tots els flags actius i els inactius.
8. El sistema comprova la consistència dels flags en funció de el autòmat.
9. El sistema computa l'autòmat amb l'input.
10. El sistema obre un finestra i explica els resultats de la computació.

Fluxos Alternatius:

8. En cas que els flags siguin inconsistents amb l'autòmat és mostra un missatge d'error
9. En cas de quedar-se sense memòria o porta molta estona computant mostra un avis i para

PostCondició.	L'usuari visualitza un autòmat del conjunt d'exemples.

Cas d'us	**Determinitzar**
Descripció:	L'usuari aplica l'algoritme de determinització a un NFA
Actor:	Anònim
Precondició:	Hi ha un NFA visualitzat
Flux Principal: 1. Selecciona el grup de "flags". 2. Clica el botó de "determinitzation" 3. El sistema aplica l'algorisme de determinització l'autòmat que està visualitzat. 4. Dibuixa el nou autòmat i el visualitza a la web	
Fluxos Alternatius:	
PostCondició.	L'usuari ha aplicat l'algoritme de determinització al NFA entrat anteriorment.

Cas d'us	**Minimitzar**
Descripció:	L'usuari aplica l'algoritme de minimització a un DFA
Actor:	Anònim
Precondició:	Hi ha un NFA visualitzat
Flux Principal: 1. Selecciona el grup de "flags". 2. Clica el botó de "minimitzation" 3. Si no és determinista a. El sistema aplica l'algoritme de determinització 4. Fi si 5. El sistema aplica l'algorisme de minimització l'autòmat que està visualitzat o que	

	acabem de Determinitzar. 6. Dibuixa el nou autòmat i el visualitza a la web
Fluxos Alternatius:	
PostCondició.	L'usuari ha aplicat l'algoritme de minimització al NFA entrat anteriorment.

Cas d'us	**Generalització d'un DFA**
Descripció:	L'usuari aplica l'algoritme de generalització a un DFA per obtenir l'expressió regular.
Actor:	Anònim
Precondició:	Hi ha un NFA visualitzat
Flux Principal: 1. Selecciona el grup de "flags". 2. Clica el botó de "DFA to RE" 3. Si no és determinista a. El sistema aplica l'algoritme de determinització 4. Fi si 5. Aplica l'algoritme de minimització. 6. El sistema aplica l'algorisme de generalització l'autòmat que està visualitzat o que acabem de Determinitzar. 7. El sistema obre una pantalla i mostra l'expressió regular equivalent al MFA.	
Fluxos Alternatius:	
PostCondició.	L'usuari ha aplicat l'algoritme de generalització per mostrar l'expressió regular.

Cas d'us	**RE to NFA**
Descripció:	L'usuari crea un NFA a partir d'una expressió regular.
Actor:	Anònim
Precondició:	cap

Flux Principal:

1. Selecciona el grup de "flags".
2. Clica el botó de "DFA to RE".
3. El sistema mostra una pantalla per tal d'entrar l'expressió regular.
4. El sistema valida que els operadors siguin correctes.
5. El sistema aplica l'algoritme de Thompson per construir el NFA equivalent.
6. El sistema visualitza l'autòmat generat.

Fluxos Alternatius:

4. En cas que l'expressió regular no contingui els operadors correctes, mostra un error

PostCondició.	L'usuari ha dibuixat el NFA equivalent a un Expressió regular.

9.1.Disseny

Per tal de dur a terme aquets projecte, s'utilitza el projecte en C++ com apunt de partida. Aquest projecte és via línia de comandes. El projecte permet el us de qualsevol dels autòmats Finits, de Pila i MT, poder-los computar segons diferents opcions.

El funcionament és simplista, li entrem qualsevol fitxer que contingui la codificació d'un NFA/DFA o PDA, en fa el parser per saber si esta ben codificat, i un cop passat el parser, el tradueix en la seva equivalent Màquina de Turing per tal de computar amb l'input entrat. En el cas dels NFA els determinitzava i els minimitzava en cas que no haguem activat el "flag" d'indeterminisme.

Per tant, en realitat i perenne l'usuari, el projecte només computava MT o les MT equivalents respecte l'autòmat entrat. Això fa que en realitat, mai computem realment un NFA o PDA.

Permet l'ús d'un pseudodebug, o sigui veure pas a pas que passa a les cintes, i en quin estat estem. També via "flag" ens permet ensenyar la configuració de la màquina en cada moment. I per últim, ens permet obtenir el temps que ha tardat en executar sigui del programa o del MT en si. Vegem el seu diagrama de classes del projecte:

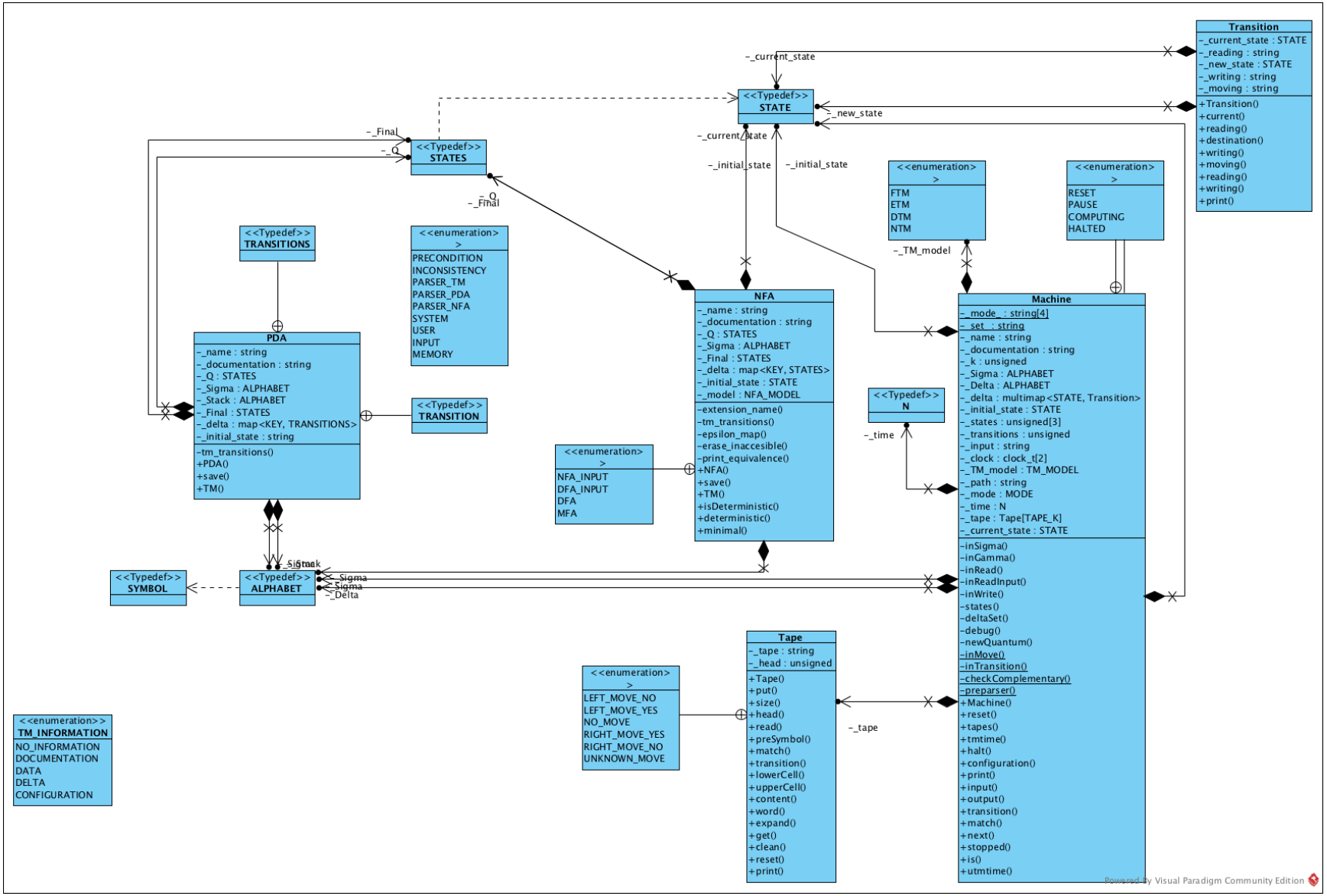


Figura 11 Diagrama de Classes C++

Tal com és pot apreciar en el diagrama de classes, les codificacions de NFA i PDA, no poden ser computats, ja que no tenen cap mètode preparat per fer-ho, i per contra la Machine(MT) hi ha el mètode "compute", i per tant quan volem computar-los veiem que crea la seva MT equivalent.

Al tenir ple accés a aquest projecte, és decideix en traduir-lo en Java i implementar la computació dels NFA/DFA i dels PDA per tal de no haver de traduir les codificacions del usuari i així simular amb més realisme el comportament de els mateixos.

Per tal d'afegir quelcom més en el cor de l'aplicació s'implementa una capa de rutines dels autòmats Finits, a saber:

- Determinització.
- Minimització.
- Generalització d'un DFA.
- Obtenció del NFA equivalent a una Expressió regular.

Aquestes 4 noves funcionalitats, permetran l'usuari poder estudiar les equivalències entre NFA i DFA, la possibilitat d'obtenir a partir de qualsevol NFA de qualsevol mida, el seu homòleg DFA. També permet l'obtenció del DFA mínim, o sigui aquell DFA que és equivalent al entrat però que conté el mínim d'estats possible per tal de reconèixer el mateix llenguatge. La possibilitat d'obtenir l'expressió regular d'un autòmat Finit, o sigui trobar una expressió regular que validi la idea que "un llenguatge és regular si està descrit per un expressió regular.

Abans de començar a traduir el projecte C++ a Java, és dissenya el diagrama de classes amb Java. Tal com és pot apreciar, es prima la idea de codificar cada part d'un autòmat en un objecte, i per tant, augmenta el nombre de classes.

Tal com hem dit ja amb anterioritat, el fet que un PDA no deixa de ser un NFA amb una Pila i que un PDA és una MT amb una cinta amb dos moviments, això genera la idea que podem generalitzar el comportament d'un autòmat, fent que hi hagi una classe Abstracta, de nom autòmat, que implementi tot el codi que sigui equivalent en el comportament dels autòmats.

Això serveix per no duplicar codi entre objectes de comportament igual. Però a la vegada obliga que una transició d'un NFA, d'un PDA o d'un TM, la seva codificació està al annex 1.

Dit això aquest és el disseny de classes pensat en el seu inici, per tal de no carregar-lo en accés i fer impossible el seu enteniment, presentem el pensat només per els NFA.

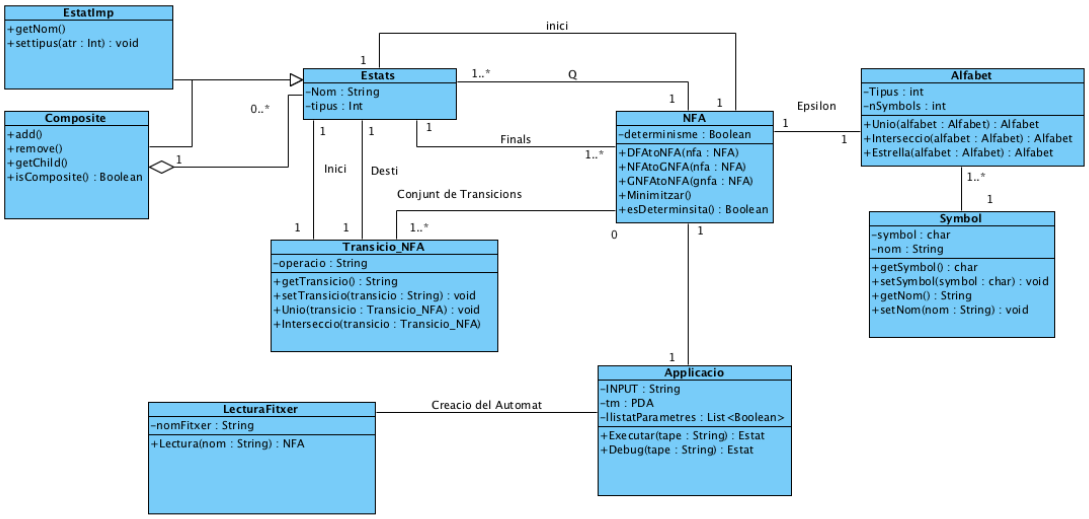


Figura 12 Primer Diagrama de Classes NFA

Podem nota l'existència d'un patró de disseny com el "composite", el qual ens havia de servir per tal d'agrupar tots aquells estats els quals poden anar amb els mateixos paràmetres.

És necessari dir que el diagrama d'un PDA és igual, només difereix en el fet del canvi de les transicions, i un atribut més que seria la pila. I per les Maquines de Turing hi ha més canvis, com l'existència de la classe Tape, la qual implementa el comportament d'una cinta, i s'enllaça amb la classe autòmat, i la diferència en la implementació de la transició.

En el seu inici no és contemplava la posició de les rutines de treball dels NFA/DFA, a recordar, determinització, minimització, generalització, etc. Cal dir que hi ha l'existència d'una classe especifica per tal d'efectuar la lectura del fitxer, ja que un dels requisits dels projecte és la possibilitat de computar el nostre propi autòmat, i per tant, s'ha de tenir en compte i s'ha de validar, fer el parsing, del codi. Així descarreguem la classe autòmat de la construcció del mateix. Veiem també que hi ha una classe Aplicació, la qual porta el pes de la computació i les diferents validacions per tal de simular el comportament de qualsevol autòmat.

En la mesura que anaven passant els "sprints" corresponents s'han pres diferents decisions. Són les següents:

- **Sprint 4:** *Canvi en la codificació d'un estat*: és treu el patró de disseny compositiu i és fa una única classe "State", fent que en la classe autòmat, l'atribut final_state passi a ser un "Set<State>" això millora en el fet que no hem de vigilar en tot moment que fem una petició al estat si és compositiu o no.
- **Sprint 5:** *Afegim la Classe Parameters:* és descarrega la classe Aplicació del control dels paràmetres d'entrada i és crea una classe apart. La classe Parameters, contindrà tots els flags de tots els tipus d'autòmats, com el seu nom i el fitxer.
- **Sprint 6:** *Canvi codificació de la funció de transició:* La funció de transició compresa com a una simple llista de transicions és canvia a una estructura més robusta i de més fàcil accés com un MAP<State, List<Transició>>. Això és fa per què en les MT hi ha un augment considerable de consultes a la llista de transicions d'un estat. Així reduïm el nombre de cerques per tal de trobar totes les transicions d'un estat. Tal com diem, la clau del MAP, és l'estat origen i el valor una llista amb les transicions de l'estat.
- **Sprint 7:** *Creació del Package BasicOperations:* és decideix crear un paquet amb les classes necessàries per tal d'implementar de forma perenne a la classe NFA tot el seguit de rutines.
- **Sprint 9:** *Creació d'un Paquet per les RE:* és decideix crear un paquet amb totes les classes necessàries per tal de crear una expressió regular des d'un NFA. Això fa que és creí una classe per generar l'arbre de pírcing d'una expressió regular, la transformació d'una RE en post ordre.

Després de fer els canvis que he esmentat aquest n'és el nou diagrama de classes:

Al ser molt gran, hem tret tots els mètodes privats:

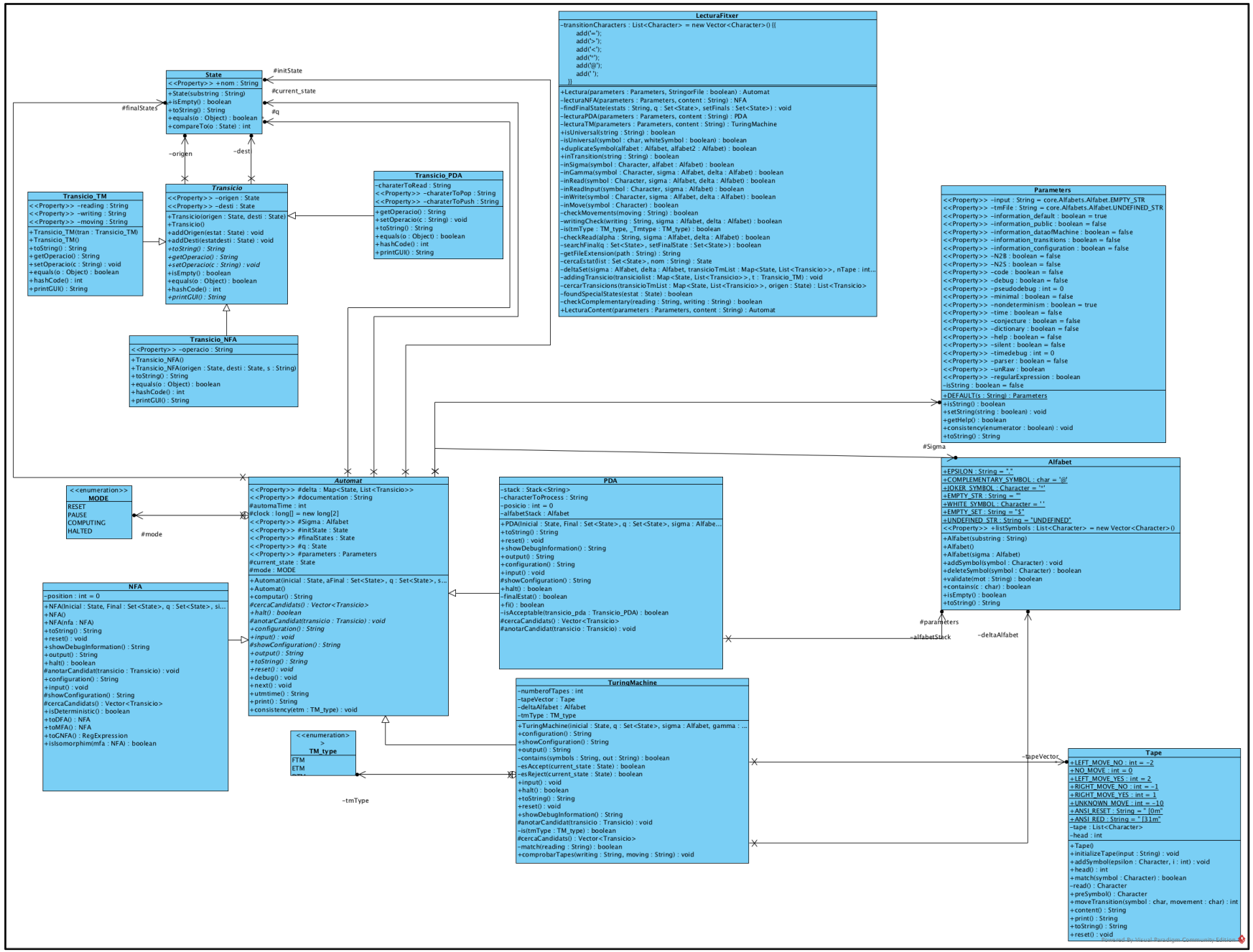


Figura 13 Diagrama de Classe Java

Un cop decidit que el llenguatge del nucli seria en Java decideixo utilitzar per la part web els Java Server Faces, ja que a ser la capa web de Java, ens deslliure de la preocupació de la relació entre el servidor i el client. El Java Server Faces que es basa en el patró Model-Vista-Controlador. Dins dels Java Server Faces escullo la suite de component PrimeFaces, decideixo utilitzar-la ja que té un component de nom diagrama, que permet el dibuix d'una màquina d'estats. O sigui, la suite conté un component per dibuixar correctament un autòmat. També permet la possibilitat de modificar al nostre gust qualsevol dels seus components, ja que és un projecte Open Source.

Per tal d'entendre el funcionament dels Java Server Faces i del Primefaces explicarem una mica per d'amunt el funcionament del patró MVC.

El patró MVC.

Els Java Server Faces usa el patró M-V-C(Model-Vista-Controlador) per a crear les aplicacions web, per això s'ha trobat coherent fer-ne una petita introducció abans d'explicar amb més detall el Play 2.0.

Definicó del patró.

Aquest patró normalment se'l coneix com a patró de patrons degut que és un patró que que n'engloba 3 en un mateix disseny. Aquests són l'Strategy (implementat pel controlador), l'Observer (implementat pel model) i finalment el Composite (implementat per la vista).

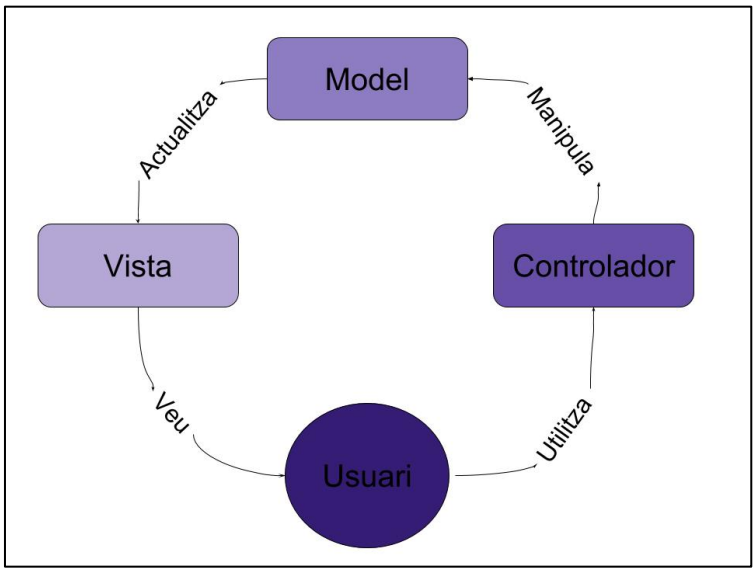


Figura 14 Diagrama MVC

Les parts d'aquest patró queden clarament diferenciades ja que tenim una part que interactua amb l'usuari (Vista), una part no visible que controla (Controlador) possibles canvis que l'usuari pugui fer sobre la vista i finalment una tercera tampoc visible anomenada model que és amb la qual l'usuari interactua a través de les vistes. El controlador és l'encarregat d'actualitzar el model en funció dels canvis que faci l'usuari i retornar a les vistes els canvis sobre el model que s'hagin pogut produir.

Tot i que s'explicarà amb detall durant el següent apartat de la memòria quina és cada part del projecte en el patró, es presenta aquí un esquema molt senzill del patró per entendre'l gràficament:

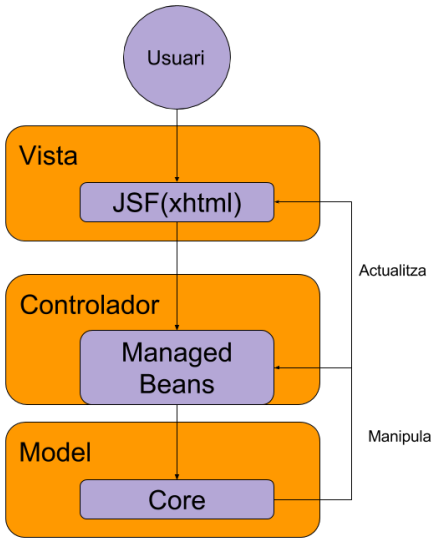


Figura 15 Representació patró MVC

Vist l'esquema anterior, a molt alt nivell veiem que sota els noms de les classes, demostrant doncs que l'aplicació manté l'estructura del patró. Ho podem veure clarament, ja que la part de Vista són tots els arxius amb extensió .xhtml. Els controladors són els que estan al Package bens, i el model són totes les classes que estant al Package Core.

Com ja he mencionat anteriorment, s'ha utilitzat la suite de Components de Java Server Faces, PrimeFaces. El Primefaces és una llibreria de components para Java Server Faces (JSF) de codi obert que compta amb un conjunt de components enriquits que faciliten la creació d'aplicacions web.

El Primefaces conté un conjunt de components com Editors de Html, autocompletar, panells, gràfics, diagrames, diàlegs, etc. Té suport ajax amb desplegament parcial, el qual permet controlar quins components volem actualitzar i quins no. I el més important, el disseny Responsive de tots els components, fent-ho adaptable a qualsevol dispositiu.

Graph Layout

El Primefaces proveeix del component “diagram”, el qual contindrà tot el necessari per dibuixar un autòmat, però no conté cap algoritme per tal de dibuixar de forma planar un autòmat.

Llavors, per tal de dibuixar l’autòmat hem d’utilitzar un algoritme de dibuix de grafs, ja que no podem saber si el nostre autòmat el podem dibuixar sense encreuar qualsevol de les transicions. Per tant, és decideix utilitzar l’algoritme de “Kamada-Kawai” ja que és l’algoritme que obté el major nombre de creuament de transicions, i és el que en el mon dels grafs unidireccionals convergeix amb més rapidesa per sobre dels “Spring Embedding”, el “FR”, etc.

Per tal d’utilitzar aquest algoritme és busca l’existència d’alguna llibreria que l’implementi i ens permeti dibuixar-lo en qualsevol component. Després de la cerca, és troba el paquet “JUNG”.

El JUNG (Java Universal Network/Graph Framework) és una llibreria software en Java que permet modelar, analitzar, visualitzar qualsevol tipus de dades que poden ser representades amb un graf. Per tant, ens proveeix d’una API la qual conté la implementació d’uns quants algoritmes de dibuix de grafs, entre ell el “Kamada-Kawai”

10. Implementació i proves

Per tal d'implementar l'aplicació i tal com s'ha dit ja amb anterioritat, s'ha utilitzat el llenguatge de programació Java, i el Java Server Faces. Al JSF a l'utilitzar el patró MVC, descrit anteriorment, ara explico la implementació seguint aquest patró.

Tot seguit és passarà a descriure cada una de les parts, a un alt nivell, és a dir, el seu esquema i una petita explicació i llavors els elements que hi intervenen i perquè.

El Model.

El model al ser el cor de l'aplicació l'explicarem a través dels Java packages a on estan encapsulades les classes.

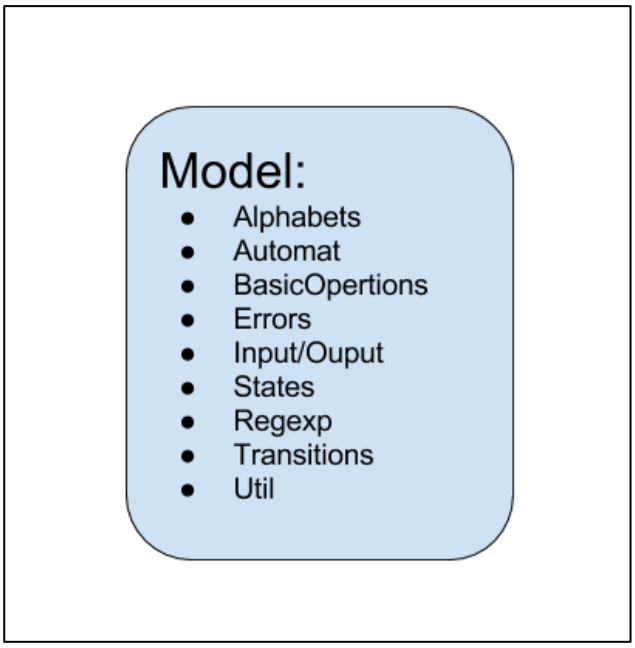


Figura 16 Diagrama de Packages del Model

Tal com dèiem a l'apartat de Conceptes Previs, hem hagut de codificar les codificacions Matemàtiques dels autòmats Finits Deterministes, indeterministes, els autòmats de Pila i la Màquina de Turing. El Package autòmat el deixem per el final.

En el Package “Alphabet” hi ha la classe Alphabet que conté tots els símbols d’un alfabet. Cal recordar que un NFA/DFA tenen només un alfabet, però que els autòmats de pila i la Màquina de Turing en tenen mínim 2.

També conté la classe Tape, que és la que s’encarrega de simular el comportament d’una Cinta, aquesta només és utilitzada per les Maquines de Turing. Està relacionat amb la classe Màquina de Turing en una relació de 1 a n, o sigui que una MT tindrà tantes Tapes com ella necessiti.

Una de les millores d’aqueta eina respecte a la seu progenitor(Projecte UTM C++) és la implementació de les rutines de determinització, minimització, generalització i creació d’NFA a partir d’una RE. Totes aquestes rutines són implementades al Package de “BasicOperations”.

En el Package “error” hi han dos Excepcions que he implementat per tal de tenir un ple control dels errors en el comportament del Core.

Un usuari pot introduir el seu propi programa i executar-lo per tant, en el Package “Input” hi ha la classe que implementa l’entrada via fitxer i els múltiples flags de computabilitat de l’eina.

El Package State conté la classe State. La qual conté el nom, i les posicions dels estats.

El Package Regexp que conté totes les classes principals i secundaries per tal d’implementar una Expressió regular.

El Package Transition conté la implementació de les diferents tipus de transició necessàries de cada autòmat. Aquest Package conté la classe Abstracte Transition, la qual només conté dos objectes, els estats d’inici i de destí de la transició. Vegem l’esquema.

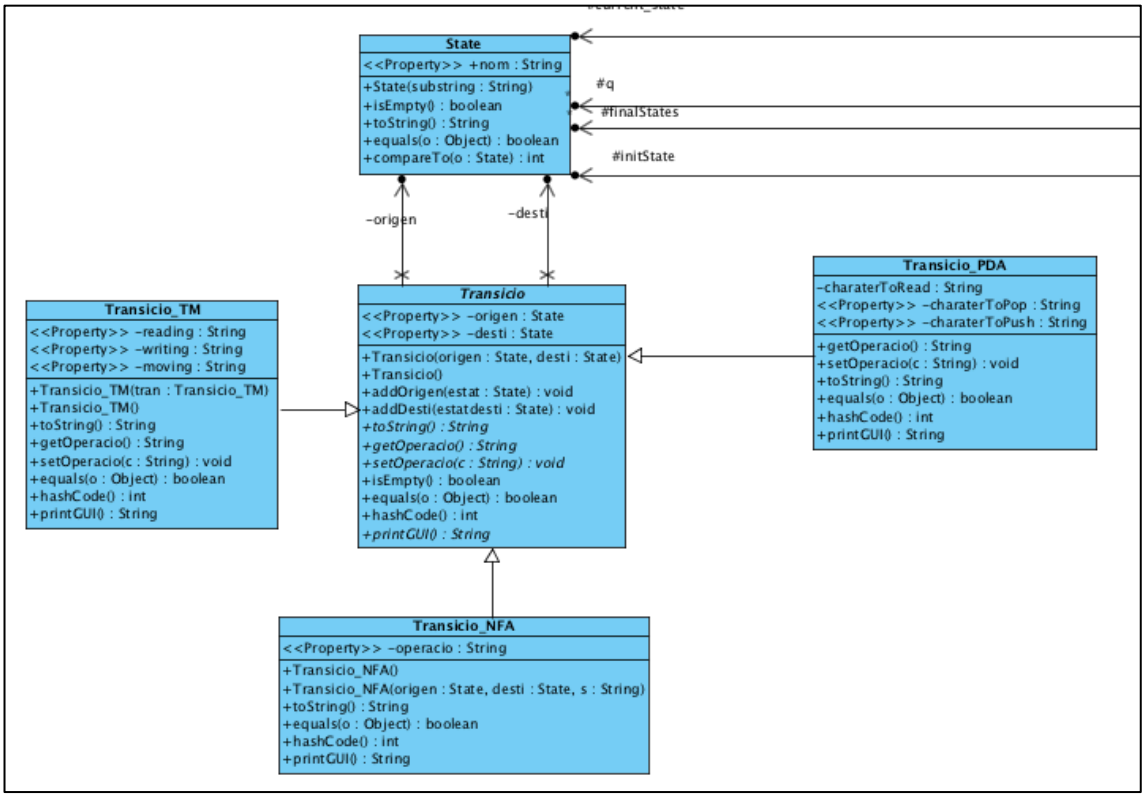


Figura 17 Diagrama del Package transició

El Package “autòmat” aquest Package l’explicarem amb més deteniment ja que és la classe que conté tot el funcionament de qualsevol dels 3 tipus d’autòmats. Tal com s’ha dit anteriorment, una MT no és més que un PDA en el qual l’hi hem afegit tantes Cintes com un vulgui i a l’hora un PDA no és més que un NFA amb la possibilitat d’emmagatzemar en una Pila. Dit això, l’herència dels 3 tipus és clara. Vegem el seu diagrama de Classes

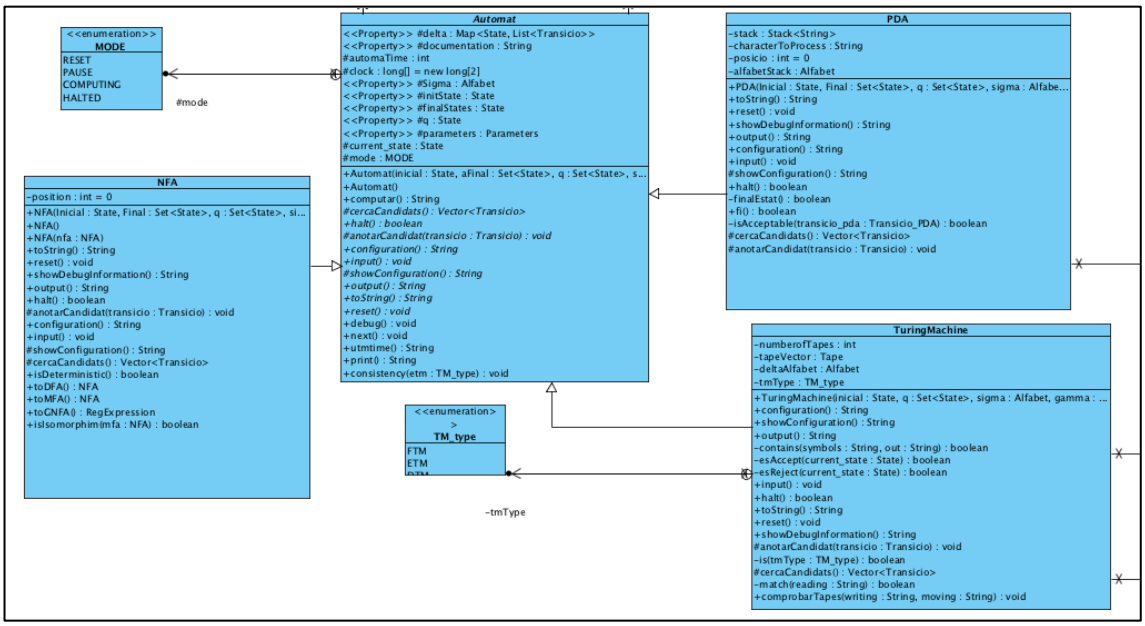


Figura 18 Package autòmat

Per tant, la classe autòmat tindrà els següents atributs, a on diferenciarem entre els necessaris per poder computar, els pertanyents a la codificació de el autòmat i altres d'auxiliars.

```
//Automat Attributes
protected Alfabet Sigma;
protected Map<State, List<Transicio>> delta;
protected State initState;
protected Set<State> finalStates;
protected Set<State> q;
protected Parameters parameters;
protected String documentation;

//Compute Attributes
protected State current_state;
protected MODE mode;
protected int automaTime;
protected long[] clock = new long[2];
```

Figura 19 Atributs autòmat

El primer bloc d'atributs hi han tots els necessaris per codificar un autòmat segon la notació matemàtica, extreien dels 3 aquells atributs que són iguals, i fem especial esment en els dos últims. L'atribut "Paràmetres" és un objecte que conté tots els flags que l'usuari hagi activat o els per defecte de cada autòmat, també conté el nom de el autòmat i l'input. El atribut "documentation" és un String que conté informació sobre quin nom té l'autòmat, de quin tipus és, quina expressió regular implementa el llenguatge regular que reconeix, un conjunt finit d'exemples i el contacte amb el propietari del programa.

Per tal d'explicar l'últim bloc, explicarem una miqueta el funcionament del mètode "compute". El mètode "compute" s'encarrega de computar el programa entrat per l'usuari amb l'input que ell hagi decidit. O sigui, un cop carregat el programa, configurats tots els flags possibles i amb un input o no "computem" el nostre autòmat per tal d'intentar arribar a una solució. Cal dir que en el cas del indeterminisme, podem caure en una de les infinites branques i per tal de solucionar aquest problema l'eina llançarà una excepció per tal de no caure en bucle infinits.

```
public String computar() throws Exception {
    current_state = this.initState;
    String printMethod = this.print() + "\n";
    this.input();
    String configurationString = this.configuration() + '\n';
    while (!this.halt()) {
        this.next();
        configurationString += this.configuration() + '\n';
        this.debug();
    }
    printMethod = parameters.isInformation_configuration() ? configurationString + '\n' : printMethod;
    String utmtime = this.utmtime();
    printMethod += utmtime + '\n';
    String output = this.output();
    printMethod += "Computing Result: " + output + '\n';
    return printMethod;
}
```

Figura 20 Mètode Compute

Tal com veiem el mètode computar ens retorna un "string" amb tot el que hagi pogut succeir en el mètode. Els flags de sortida estan presents en tot moment i per tant, veiem l'existència del mètode "this.configuration()" que en funció dels flags d'entrada retornarà, a saber: les transicions, la documentació pública, la configuració de el autòmat o res. El primer que fem és establir l'estat inicial com a l'estat actual. En cas que haguem seleccionat qualsevol dels flags de sortida el mètode print ens el retornarà i el guardem en el string de sortida.

En el mètode input és abstracte, ja que en el cas particular de tenir una ETM, aquestes no tenen entrada. Dins d'aquest mètode validem que l'input estigui format per qualsevol dels símbols del alfabet Sigma de el autòmat, en cas que hi hagi un símbol erroni, el mètode llança una excepció. El mètode input a més de validar l'alfabet correcte de l'input, afegeix si l'usuari ho demana un input preestablert, activant el flag corresponent. Tal com he dit anteriorment el mètode "configuration" serveix per implementar els flags de sortida de cada tipus d'opció. Cada autòmat té una configuració diferent ja que podem tenir o no tenir Cintes.

El mètode "halt" serveix per dictaminar si hem acabat la computació del input o per altre banda si no podem seguir computant i hem d'avortar.

En el cas dels NFA el mètode "halt" té en compte que el "mode" de computació no sigui "HALTED" o que haguem llegit tot l'input. En el cas dels PDA també comprovem que ni estigui en mode "HALTED" o que haguem arribat a llegir tot el input, si ens trobem en un estat final i en funció de l'estat de la pila. I per últim els MT mirem que el mode sigui "HALTED".

El mètode "next" és el més important dins el mètode "compute" ja que s'encarrega de moure l'autòmat. Que volem dir amb això, té la responsabilitat de trobar el destí en funció del símbol d'entrada, en cas que no hi hagi cap possibilitat de moviment posarem la màquina en mode "HALTED" . Tal com he dit anteriorment, a l'implementar el Core amb programació defensiva, comprovem a cada pas que el flag d'indeterminisme estigui actiu i en cas que tinguem múltiples possibilitats llança una excepció. Altrament, escull un dels candidats a destí de forma aleatòria. Un cop haguem escollit el destí, escollim la transició corresponent i fem tots els canvis en la configuració necessaris per moure la màquina d'estat. Els canvis en la configuració els implementa cada autòmat en la seva classe, ja que un NFA només ha d'avançar l'índex de l'input, però en el cas dels PDA, ha de tenir en compte si el caràcter de l'input és ε, si el que hi ha al cim de la pila és un ε o si el que hem d'empilar és ε. Hem de dir que quan hi ha una ε en qualsevol de les 3 opcions, fa que efectuem una acció diferent, podem obtenir fins a 2^3 combinacions diferents d'entrada, pop i push. Les MT haurem de d'actualitzar totes les cintes que hi hagin i llegir el contingut de la posició.

Un cop computat, comprovem el flag de "documentation" per tal d'afegir-la o no al inici del resultat.

Al tenir la possibilitat de saber el temps que ha tardat l'autòmat a computar, hi ha el mètode "utmtime" que comprova el flag i gràcies a l'Array de Long sabem el temps que ha transcorregut entre l'inici de la computació i el final, ja que en iniciar la computació ens guardem l'hora inicial del sistema i un cop acabem i entrem en el mètode fem la diferència de l'hora d'inici amb l'actual i ho afegim al string de retorn.

I per acabar el mètode output en el cas dels NFA dóna com a resultat {accept, reject} o {true, false} en funció del booleanFlag. En el cas dels PDA, igual que amb els NFA, el resultat el mostrarem com {accept, reject } o {true, false}. I les MT com indica la codificació, l'ultima cinta conté el resultat de la computació, per tant, llegim tota la cinta i la mostrem per pantalla.

Per tal d'explicar amb més deteniment la codificació dels autòmats, explicarem cada autòmat per separat, ja que hem explicat la part que els uneix.

Autòmat Finit Determinista i Indeterminista.

Per tal d'explicar la codificació dels NFA/DFA vegem-ne la descripció matemàtica:

Un autòmat finit és una 5-tupla $(Q,\Sigma, \delta,q_o, F)$, on:

1. Q és un conjunt finit denominat states.
2. Σ és un conjunt finit denominat alfabet.
3. $\delta: Q \times \Sigma \rightarrow Q$ és la funció de transició. $\in$
4. $q_o : Q$ és l'estat inicial.
5. $F : Q$ és un conjunt d'estats d'accept.

Descripció 4 Autòmat Finit Determinista

Tal com veiem en la figura anterior s'ha decidit implementar els autòmats seguin la descripció matemàtica anterior. Seguin la definició un autòmat tindrà per tant, mínim 5 atributs, un Set amb el conjunt dels estats, l'alfabet Sigma que contindrà els símbols del nostre futur programa, un Map amb la clau estat i per valor una llista de transicions, un estat marcat com a inicial i per últim i no menys important un set d'estats que contindrà el conjunt d'estats que forment els estats finals.

Tal com hem explicat anteriorment, podem dir que la classe autòmat engloba tot el necessari per tal de codificar un NFA/DFA. L'únic atribut que té apart de tenir els que té la classe superior és un enter per saber en quina posició de l'input que estem llegint.

Autòmat de Pila.

Per tal d'explicar la codificació dels PDA vegem-ne la descripció matemàtica:

Un autòmat de Pila (PDA) és una 6-tuple$(Q, \Sigma, \Gamma, \delta, q_o, F)$,

on Q, Σ, Γ, F sòn conjunts finits, i

1. Q és un conjunt finit denominat states.
2. Σ és l'alfabet d'entrada
3. Γ és l'alfabet de la pila.
4. $\delta: Q \times \Sigma_\epsilon \rightarrow \mathcal{P}(Q \times \Gamma_\epsilon)$ és la funció de transició
5. $qo \in Q$ és l'estat inicial.
6. $F \subseteq Q$ és el conjunt d'estats finals.

Descripció 5 PDA

Si ens hi fixem, veurem que el PDA conté un altre alfabet Γ, el alfabet de la Pila, hi ha un atribut "stack" que conté la pila de símbols d'entrada, la qual en cada moment de la seqüència d'entrada efectuem un pop i un push. I per acabar, també guardem el caràcter que estem processant, o sigui el caràcter d'entrada per tal de saber cap a on ens dirigirem en el següent pas.

Tal com hem dit en anterioritat, els PDA poden ser indeterministes, per tant, els flags que afecten només als PDA són molt reduïts, per no dir que cap flag és només específic dels PDA.

Dels 3 autòmats diferents que hi han, és l'autòmat que menys línies de codi té, ja que en el cas dels NFA hi ha tota la part de les rutines de minimització, etc. I en el cas de les Maquines de Turing hi ha tota la lògica de consistència dels paràmetres i la sempre fàcil, però tediosa feina de cada pas anar actualitzant les infinites cintes.

Màquines de Turing.

La classe que codifica les Màquines de Turing extent la classe autòmat, i per tant, implementa els mètodes abstractes i incorpora tots els atributs i mètodes de la classe superior. A partir d'aquí hem de fer èmfasi en els atributs corresponents a la seva particular codificació.

Per tal de veure-ho més clar mostrem la descripció matemàtica de les Màquines de Turing:

Un autòmat de Pila (PDA) és una 7-tuple$(Q, \Sigma, \Gamma, \delta, q_o, q_{accept}, q_{reject})$,

on Q, Σ, Γ, sòn conjunts finits, i

1. Q és un conjunt finit denominat states,
2. Σ és l'alfabet d'entrada que no conté el **símbol en blanc** ⊔,
3. Γ és l'alfabet de la pila, a on $\sqcup \in \Gamma$ i $\Gamma \subseteq \Sigma$,
4. $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times (L \times R)$ és la funció de transició,
5. $q_o \in Q$ és l'estat inicial,
6. $q_{accept} \in Q$ és l'estat d'accept, i
7. $q_{reject} \in Q$ és l'estat de rebutjar, i $q_{accept} \neq q_{reject}$

Descripció 6 Formal de MT

Tal com veiem en el figura, podem tenir una infinitat de cintes, per tal de codificar-les dins les MT he decidit implementar com un Vector de "Tape", per tant, una MT tindrà les cintes encapsulades en un vector.

També tenim igual que en els PDA un alfabet Γ de les cintes, per tal de saber si els símbols que anem posant els podrem reconèixer, ja que sinó, com els compararem, o no.

En el cas de les TM veiem que tenim només un estat d'accept i un de reject, per codificació prèvia de l'eina, s'ha decidit que no cal guardar aquests dos estats, sinó, que "obliguem" a l'usuari a que la seva MT contingui dos estats de nom accept i reject. Fet, que en cas que l'usuari no ho hagi posat en la seva codificació del programa, salti una excepció de la falta d'algun dels dos estats

La Vista.

L'usuari només interactua amb la mateixa vista, la qual està fragmentada en 5 arxius diferents. Això ocorre per l'ús de plantilles que ens permet el JSF. Per tal de veure la seva disposició, veiem l'arbre d'arxius de la vista.

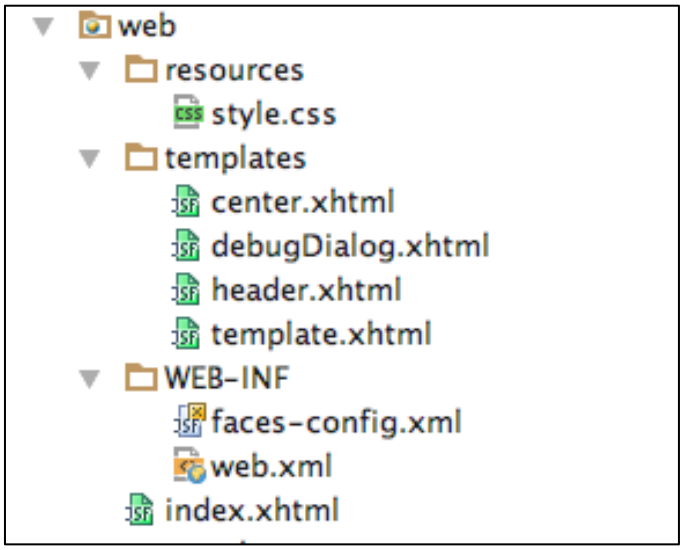


Figura 21 Arbre d'arxius de la vista

Podem veure que tenim tres carpetes en la vista. La carpeta "resources", que conté els estils necessaris per visualitzar la web, la carpeta "template", que conté totes les vistes que són comunes entre les diferents pagines i la "web-inf", que conté la informació necessària perquè el glassfish executi l'aplicació. Ara explicarem amb més detall la vista índex.

Figura 22 vista del índex

El índex és la vista que l'usuari té de la pàgina. Les parts de l'índex tal com he dit abans estan separades, ja que al ser elements dinàmics, les diferents parts estan considerades

com a objectes. Per tal d'incloure totes les parts dins la mateixa vista, el nexe d'unió és la vista "template.xhtml", la qual inclou: el "header.xhtml", el "center.xhtml" i el "diagram.xhtml".

La tecnologia ens permet inserta i designar la seva posició d'una forma molt fàcil.

```
<h:head>

    <p:ajaxStatus onstart="statusDialog.show();" onsuccess="statusDialog.hide();"/>

    <p:layout fullPage="true" resizeTitle="resize" style="background-color:#FFFFFF;">

        <p:layoutUnit position="north" size="68" id="north">
            <ui:include src="header.xhtml"/>
        </p:layoutUnit>

        <p:layoutUnit style="width: 30%" styleClass="layoutUnitCenter" position="center">
            <ui:include src="center.xhtml"/>
        </p:layoutUnit>

    </p:layout>

    <p:dialog modal="true" widgetVar="statusDialog" showHeader="false"
              draggable="false" closable="false" resizable="false"
              visible="false" position="center" maximizable="false" minimizable="false">
        <p:graphicImage value="#{resource['images/ajax-loader.gif']}"/>
    </p:dialog>
</h:head>
```

Figura 23 Insert Vistes

En la figura anterior veiem com tenim el component <p:layout> que especifica quina és la disposició dels diferents components que formen la vista. Podem veure què només hi ha 2 components, el "header" i el "center".

Podem veure que està formada per un header, un menú amb totes les possibilitats just a sota. El menú està organitzat per àrees i etapes per tal de computar un autòmat.

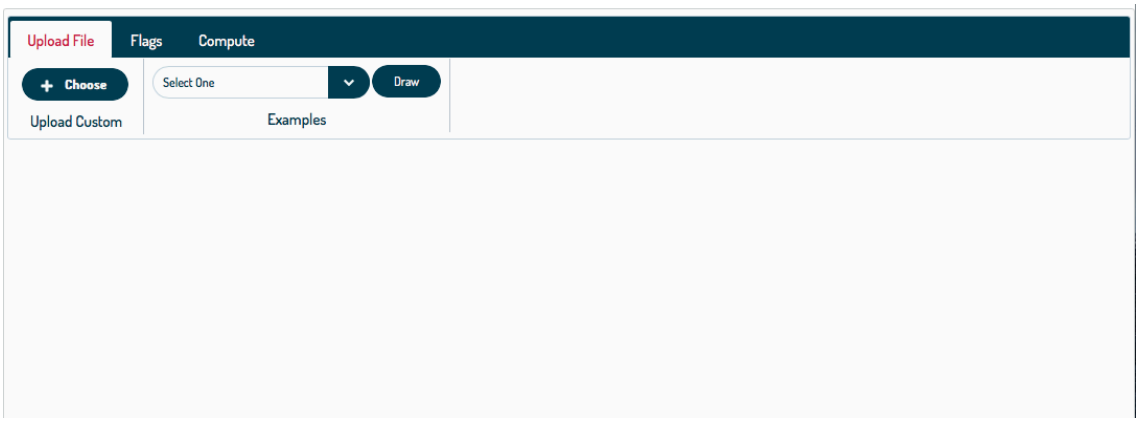


Figura 24 Vista de la Plana Web

En aquesta vista podem penjar la codificació del nostre autòmat, i també podem seleccionar qualsevol dels autòmats que hi ha com a exemple.

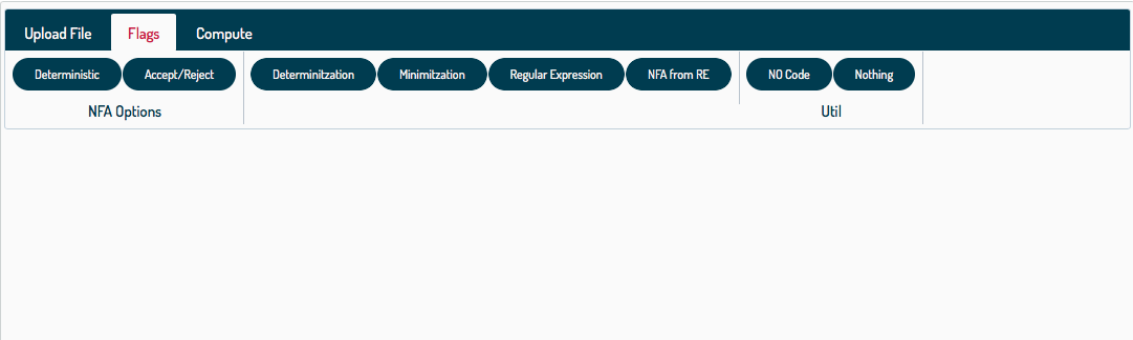


Figura 25 Vista dels Flags

En el apartat de flags, és a on podem dictaminar com volem que sigui la computació del nostre autòmat, a l'inici hi ha els flags de computació determinista, boolean result, les rutines sobre un NFA i els dos últims que serveixen per computar en binari o base 10 i per el mode conjecture. EL mode conjectura consisteix entrar ε al input i que anem cap allà a on vulgui el propi autòmat.

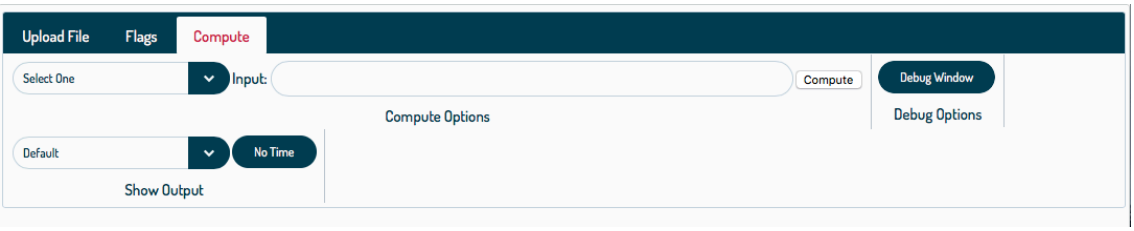


Figura 26 Vista Compute

Per últim el menú de "compute", a on podem escollir entre un seguit d'inputs preestablert, podem escriure el nostre propi input, obrir la finestra de debug i per últim, com volem que sigui el output i el temps.

Tot això només és la vista de l'aplicació, a on cada component pot ser controlat o no per un "managed bean". Tot seguit explicarem els controladors.

Els controladors.

Com hem explicat en l'apartat anterior, el controlador és l'objecte encarregat d'actualitzar les vistes si el model ha canviat a petició de l'usuari. En el nostre cas és, que per exemple, que l'usuari hagi penjat el seu propi programa, llavors el controlador haurà de dibuixar-lo a la vista.

El primer problema plantejat va ser que el JSF redirigeix cada "canvi" en una nova pàgina, i no és el que volíem, sinó que donada una pàgina part del contingut variés de forma dinàmica sense haver-ho de fer la totalitat de la pàgina.

El segon problema va ser com organitzar el controlador de la pagina, ja que al ser tot una mateixa vista, podria anar tot al mateix fitxer, però per tal de fer-ho més comprensible, vaig decidir crear un controlador per cada fitxer xhtml, sempre hi quan fos necessari la seva implementació. Vegem la distribució dels controladors i les seves responsabilitats:

Figura 27 Arbre de fitxers.

El controlador "FileUploadControllerView" controla tot el referent a la pujada de la codificació d'un autòmat, la seva responsabilitat és llegir el fitxer que pengi l'usuari i passar-lo a un string. Això ho fem per repartir la responsabilitat de fer aquesta acció. El component que s'encarrega de penjar el fitxer esta implementat per Primefaces.

```
<p:fileUpload id="fileupload" fileUploadListener="#{fileUploadControllerView.handleFileUpload}"
              mode="advanced"
              auto="true" sizeLimit="100000" update="msgs"
              allowTypes="/(\.|\/)(nfa|pda|tm)$/"/>
```

Figura 28 File Upload

En aquesta figura podem veure que el component per penjar el fitxer, necessita d'un event per tal de cridar al controlador i així fer la lectura del fitxer. També tal com és veu podem dictaminar quin tipus d'extensions han de tenir els fitxer perquè siguin reconeguts per l'aplicació, en cas que no siguin correctes, ensenya un missatge d'error.

El controlador "StateMachineControllerView", tal com diu el seu nom, controla tot el referent al dibuix dels autòmats. Tal com hem dit amb anterioritat, el controlador "FileUploadControllerView" rep el fitxer, el converteix a Java i llavors crida el controlador "StateMachineController" que s'encarrega de crear la instància de l'autòmat que haguem entrat i dibuixar-lo en el component diagram.

L' StateMachineControllerView és el controlador amb més pes dels tres ja que un cop instanciat el autòmat ha de crear un objecte "graph" de la llibreria JUNG, i després aplicar l'algoritme del Layout, el "Kamada-Kawai". Un cop aplicat l'algoritme, tenim l'autòmat amb els estats ubicats de la millor forma dins un panell. Per tant, tornem a traduir el "graph" en

el model de dades que necessita el component "Diagram" de Primefaces. És aquí on apliquem totes la simbologia d'un autòmat. El model de dades és força rudimentari, i això ocasiona que els elements(estats) només puguin ser lligats en els seus punts cardinals. Que volem dir amb això? que les arestes que simbolitzen les transicions només poden sortir o entrar des d'un dels 4 punts cardinals, fent que a vegades una aresta es sobreposi a un estat, el qual en sigui destinatari o el originari.

```
<p:diagram id="diagram" value="#{diagramStateMachineView.model}"
           style="height: 100%" styleClass="ui-widget-content">
```

Figura 29 Diagram component

Aquí veiem com la classe "StateMachine" de nom a la vista "diagramStateMachineView" conté el atribut que després el Primefaces dibuixa com el nostre autòmat.

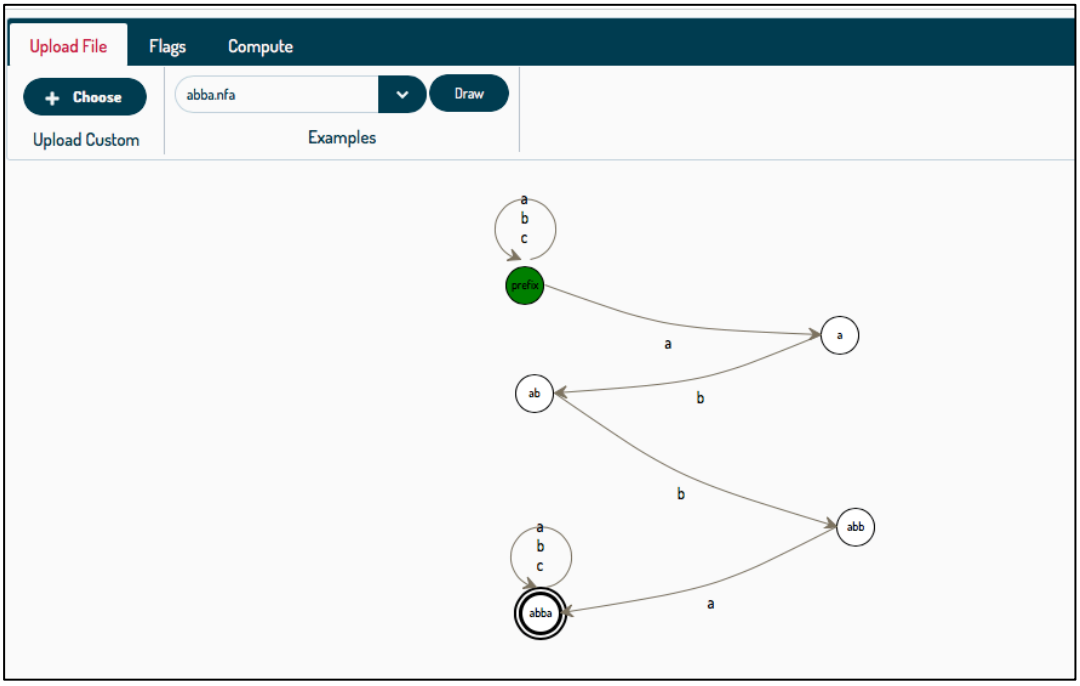


Figura 30 NFA ABBA

Podem veure com és visualitza un NFA. Veiem que les arestes tenen curvatura negativa per definició si l'estat al qual és dirigeixen està en una posició menor que l'origen. I és a la inversa, quan un estat destí està en una posició més elevada que el origen, llavors la curvatura és positiva.

L'explicació del per que de la curvatura, és quan tenim dos transicions que sòn inverses, o sigui que, la transició A té origen l'estat 1 i destí l'estat 2, i la transició B té com a estat origen 2 i com a destí 1, això fa que es sobreposin les dos arestes i les seves etiquetes, fent que sigui incomprensible les dues. Fent aquest canvi permetem que dos transicions no és pugui sobreposar.

Fins ara hem explicat com penjar un fitxer i com l'aplicació el dibuixa, el següent controlador "paramsControllerView" és el que conté tots els components dels apartats de flags, exemple inputs, i l'apartat de compute.

Aquest controlador, podríem dir que dirigeix els altres dos, ja que és l'encarregat de capturar tots els flags, que estant codificats com a botons, conté el textfield per l'input, crida al mètode computar d'un autòmat, crea la finestra de debug i en treu un output via una finestra. També s'encarrega de portar a terme les rutines dels NFA, i amb l'ajuda del "StateMachineViewController" dibuixa els autòmats després d'haver aplicat les rutines que digui l'usuari.

10. Implantació i resultats

En aquest apartat anirem explicant quines proves s'han anat fent durant el transcurs de tot el projecte. Primer mostrarem les proves fetes al Core, a la majoria de funcions implementades. Algunes d'aquestes no són requisits funcionals de l'aplicació però sí funcions útils per a testejar l'aplicació. Altres, en canvi, sí que són requerides ja que sense elles l'apartat gràfic no funcionaria.

Per fer tot el conjunt de proves del Core, s'ha creat un Main, igual que en el projecte de partida, així podíem provar que totes les funcions i el que és més important, tots els resultats sortissin idèntics als del projecte de partida.

Per tal que l'usuari pugui interactuar amb un conjunt d'exemples, he decidit incorporar-los en el projecte. Inicialment aquests exemples no estaven en el projecte de partida.

```
Usage: utm [-{hrv} |(-{01234bBcdDmnptuxy} )^* function [input]]
  function is the path of a tm-file
  input is the word to compute (between " for to avoid shell interpreter)
Options:
  -0 Output of information: none information (default)
  -1 Output of information: public documentation
  -2 Output of information: quantitative data
  -3 Output of information: transitions
  -4 Output of information: configurations
  -b Map binary outputs to false/true
  -B Map 0/1 output to reject/accept (it has priority over -b)
  -c Code/decode decimal numbers to/from binary
  -d Debug
  -D Pseudo-debug (computation delayed)
  -h Help
  -n Non-deterministic design
  -p Parser file (disable all)
  -r Read-me file (documentation)
  -t Computation time
  -v Version number and license
  -x Conjecture mode
  -y Translate UTM-words (input and output)
This is a reminder. For details, see documentation (-r).

Entrada:
abba
CONFIGURATION=[
abba
MODE: PAUSE
TIME: 0
Current State= 1
] END CONFIGURATION


CONFIGURATION=[
abba
MODE: COMPUTING
TIME: 1
Current State= 3
] END CONFIGURATION


CONFIGURATION=[
abba
MODE: COMPUTING
TIME: 2
Current State= 1
] END CONFIGURATION
```

Figura 31 Resultat Computació NFA ABBA

Podem veurem com és l'output del Main, a on tenim l'entrada per teclat. Per definició, és mostra la ajuda, i després és pot veure com és demana per l'input. Acte seguit podem

veure que tenim activat el flag de configuració. Vist que es per línia de comandes, hem utilitzat els colors del bash per tal de reflexar en quina posició estem.

Proves de funcionament del JUTM gràfic.

Per a veure que l'aplicació es comporta com s'esperava, el que es fa és anar descrivint el funcionament dels elements del menú d'opcions i mostrar, mitjançant captures de pantalla, que funcionen com s'esperava.

El primer que farem serà entrar la nostra codificació d'un NFA que validi els mots els quals la llargada de la seqüència d'entrada sigui múltiple de 3.

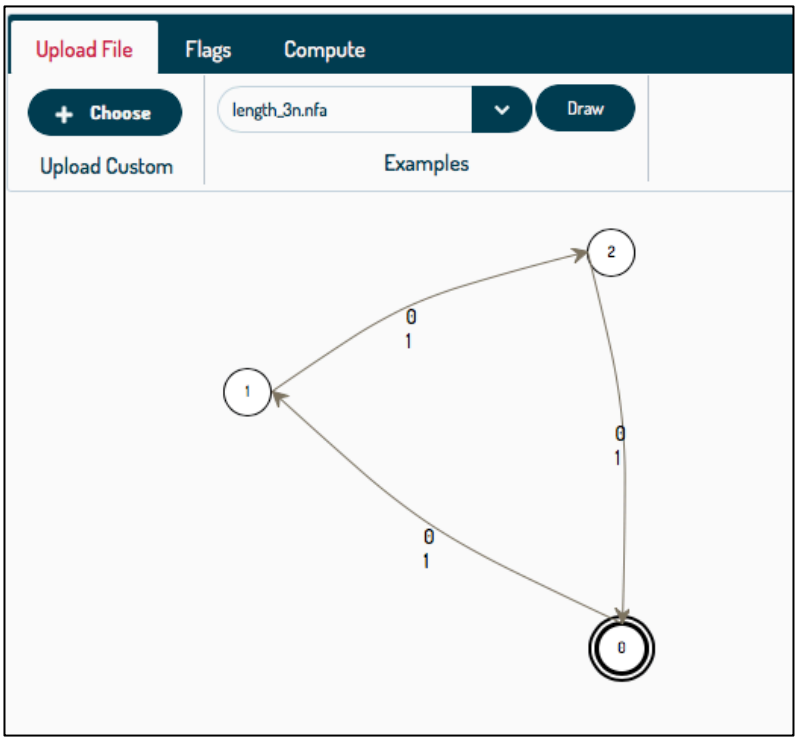


Figura 32 NFA de múltiple de 3

En el cas que quan dibuixem no és vegi la simbologia de l'estat inicial, això indica que l'estat final, és a la vegada estat inicial.

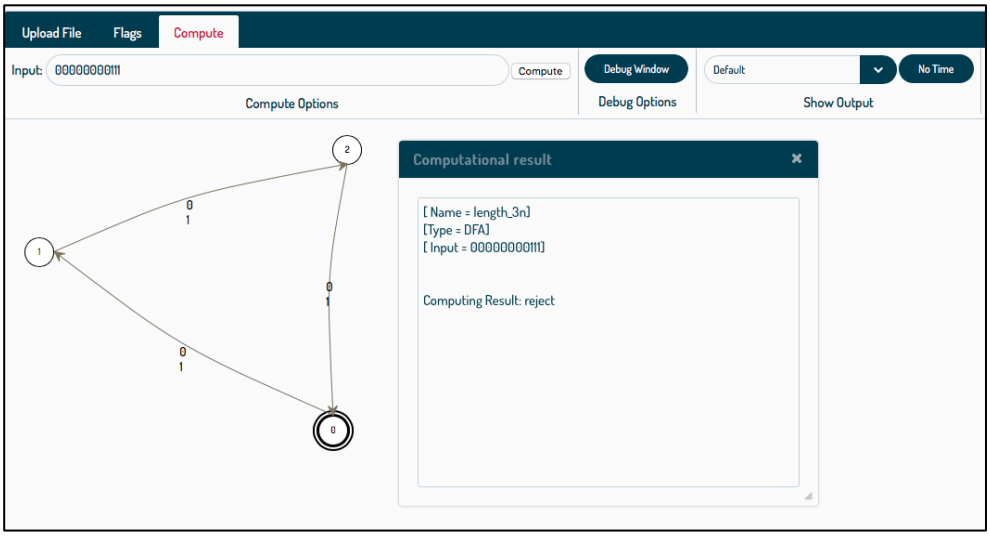


Figura 33 Computació NFA

Després de comprovar que dibuixem un NFA qualsevol, provarem la seva computació. Li entrem la seqüència "00000000111", que la llargada no és múltiple de 3, i podem veure com s'obre una finestra amb els resultats de la computació. Podem veure, que ens diu de quin tipus és l'autòmat, quin nom té i quin ha sigut l'input d'entrada. I el resultat que ha sigut "reject". Si activem el flag del resultat booleà veiem el següent:

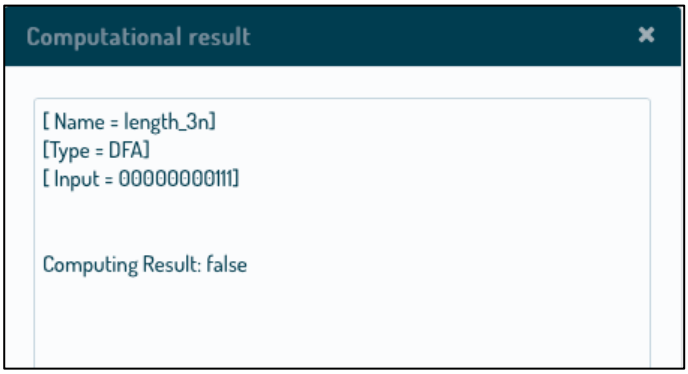


Figura 34 Flag booleà activat

Ara per tal de provar que funciona el dibuix dels PDA, n'entrarem un que validi que la seqüència entrada té el mateix de nombre de 0 i de 1.

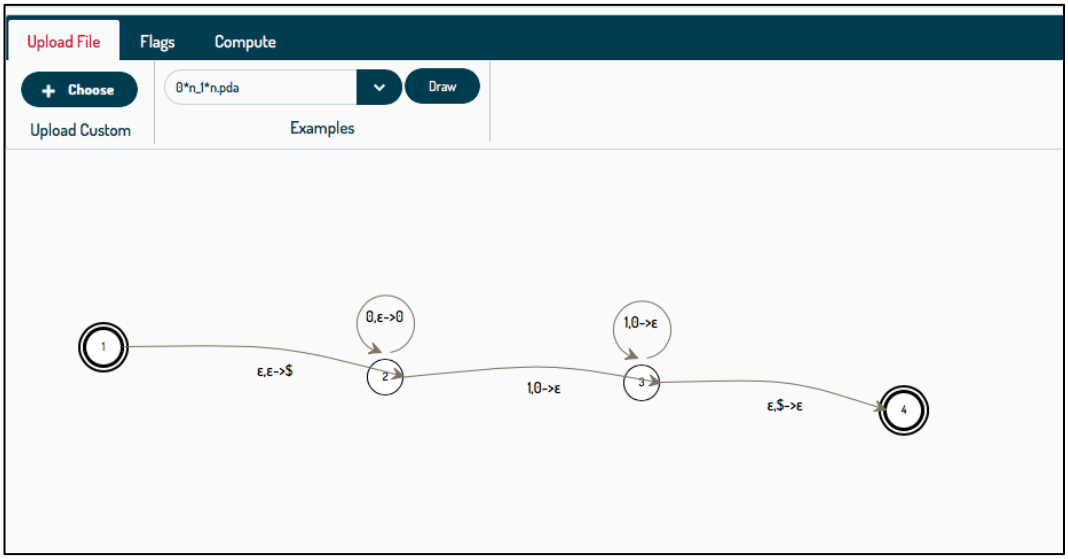


Figura 35 PDA A_1

Tal com és pot apreciar, l'etiqueta de cada fletxa conté la funció especifica d'un PDA. Vegem que les etiquetes sempre estan a sota de l'aresta i per tant, fan que sigui a vegades difícil la seva comprensió, tal com és dirà en el treball futur és una cosa a solucionar.

Ara per tal de veure la seva computació, hi entrarem la seqüència "0101010101" i activarem el flag de configurations.

[Name = 0^n_1^n] [Type = PDA] [Input = 0011] CONFIGURATION=[0011 Stack: [.] MODE: PAUSE TIME: 0 Current State= 1] END CONFIGURATION CONFIGURATION=[0011 Stack: [$] MODE: COMPUTING TIME: 1 Current State= 2] END CONFIGURATION CONFIGURATION=[0011 Stack: [$, 0] MODE: COMPUTING TIME: 2 Current State= 2] END CONFIGURATION	CONFIGURATION=[0011 Stack: [$, 0, 0] MODE: COMPUTING TIME: 3 Current State= 2] END CONFIGURATION CONFIGURATION=[0011 Stack: [$, 0] MODE: COMPUTING TIME: 4 Current State= 3] END CONFIGURATION CONFIGURATION=[0011 Stack: [$] MODE: COMPUTING TIME: 5 Current State= 3] END CONFIGURATION	CONFIGURATION=[0011 Stack: [] MODE: COMPUTING TIME: 6 Current State= 4] END CONFIGURATION Computation Done! Computing Result: accept

Descripció 7 Resultat Configuration

Aquest és el resultat de la computació. Podem veure com hem utilitzat els colors per tal de dictaminar en quina posició de l'input estem.

I per últim comprovem que la màquina dibuixa qualsevol MT, i la seva representació. Aprofitarem també per obtenir el resultat de saber si la llargada de la seqüència segueix l'expressió regular següent: $a^i\_b^j\_c^{ij}$, o sigui que el mot tindrà i vegades a, j vegades b i per últim, i*j vegades c. El input serà "aabbcccc".

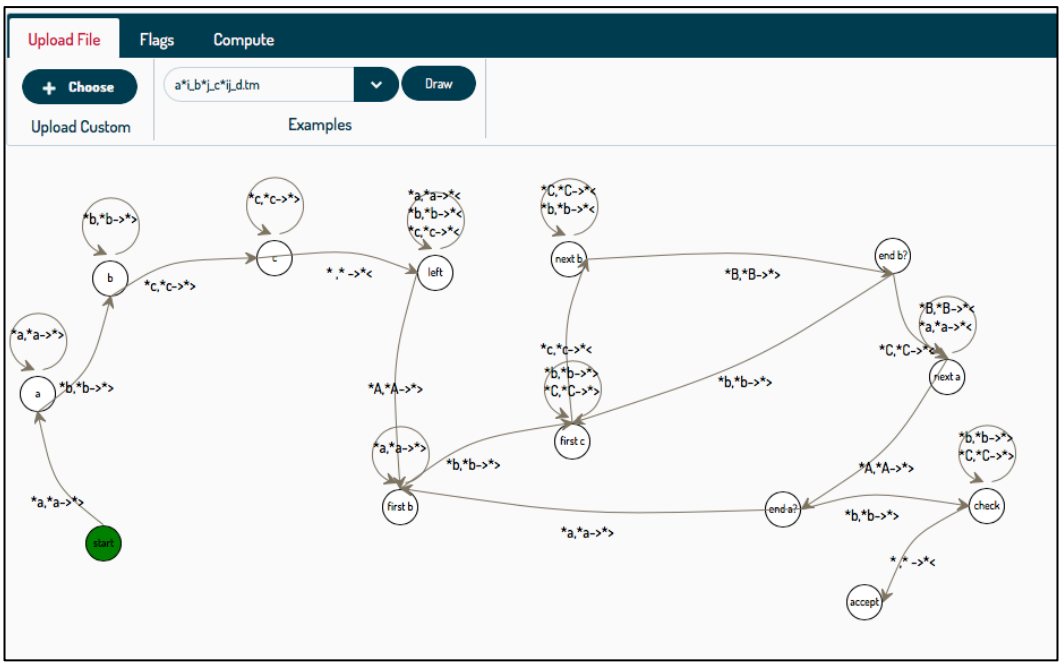


Figura 36 TM $a^i\_b^j\_c^{ij}$

Podem veure el dibuix del TM i aquest és el resultat:

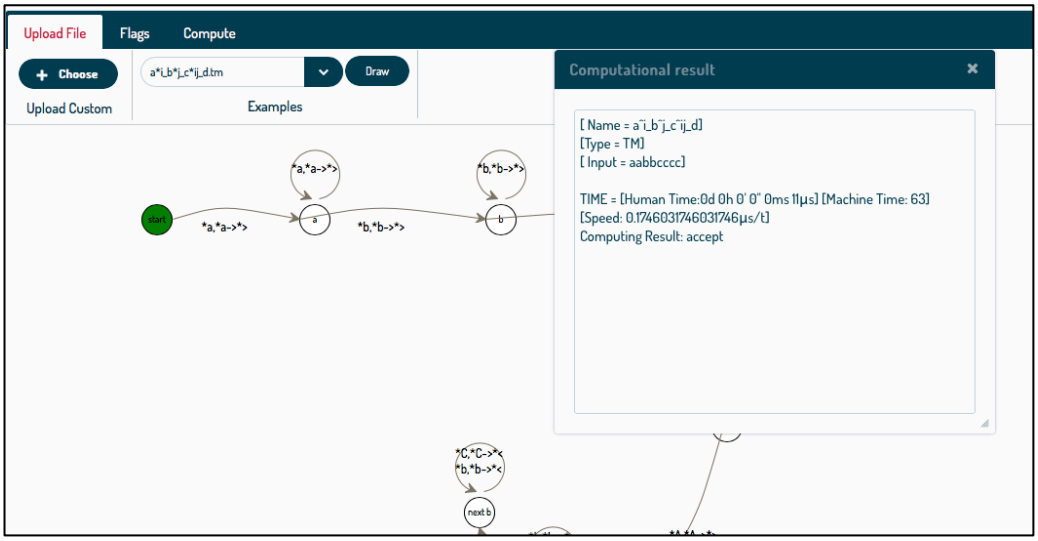


Figura 37 Resultat de la computació.

Fins ara hem provat els diferents autòmats, el dibuix, i la seva computació, ara provarem les diferents rutines dels NFA.

La primera serà sobre el NFA A, que valida que l'input conté "abba", i aplicarem l'algorisme de determinització. Vegem el NFA "abba":

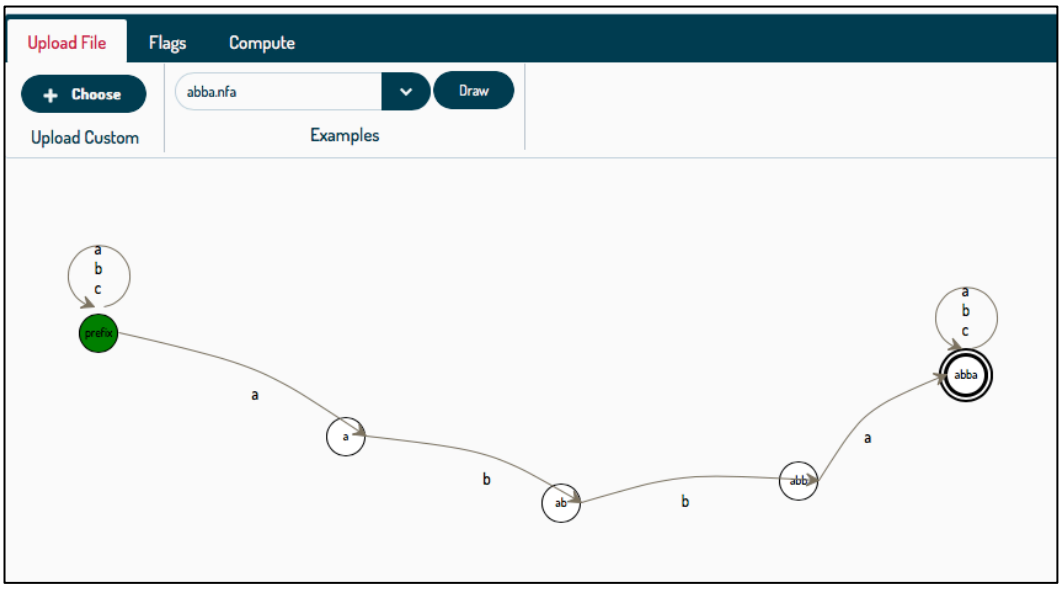


Figura 38 NFA N abba

Per tal d'efectuar la determinització del nostre autòmat, clicarem a l'apartat de "Flags", aquí en el botó de determinitzation, i veurem com canvia el nostre autòmat. A simple vista, podem veure que tots els estats tenen transicions amb tots els signes del alfabet, que ha

augmentat el nombre d'estats de 5 a 8, el qual és potencia de 2. Ha augmentat el nombre d'estats finals, això indica que el conjunt d'estats equivalents han de contenir el estat final del NFA. També i és molt especial, només podrem passar a un estat final si hem llegit una 'a'.

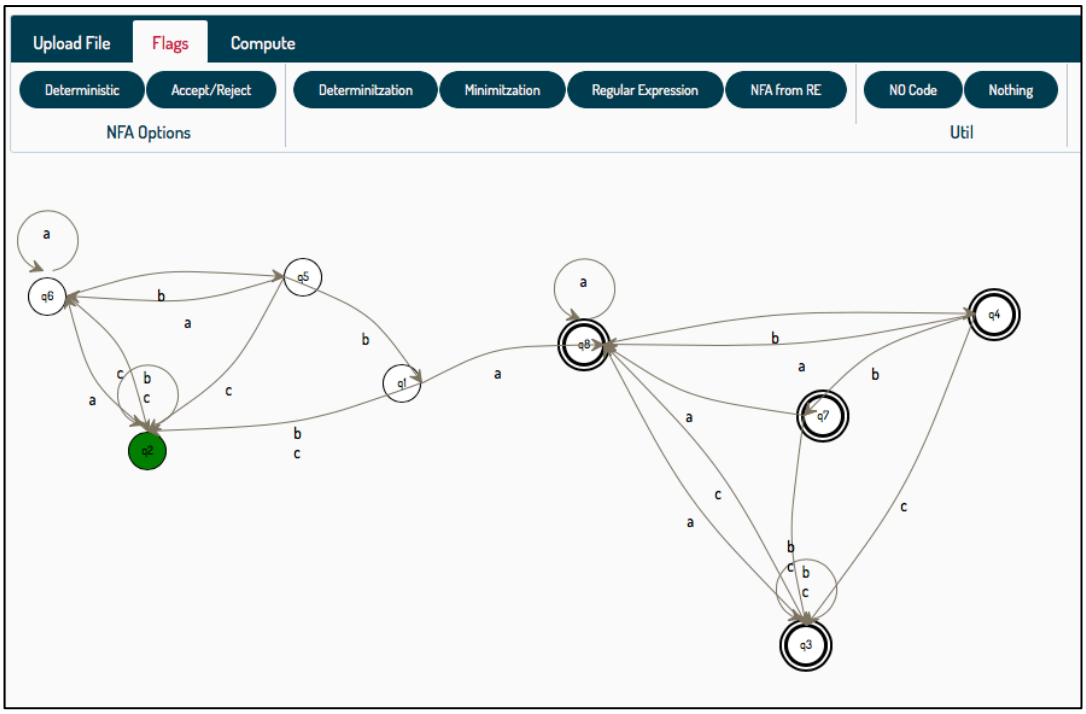


Figura 39 DFA D equivalent

Ara un cop hem determinitzat, podem veure clarament, que el grup d'estats finals és minimitzable, ja que un cop llegida la seqüència "abba" el que vingui després ens es indiferent. Per tant, ara aplicarem l'algoritme de minimització, clicant el seu botó corresponent. Per tal d'efectuar l'algorisme, abans de minimitzar-lo, comprovem que sigui determinista, i si no ho és apliquem l'algoritme de determinització.

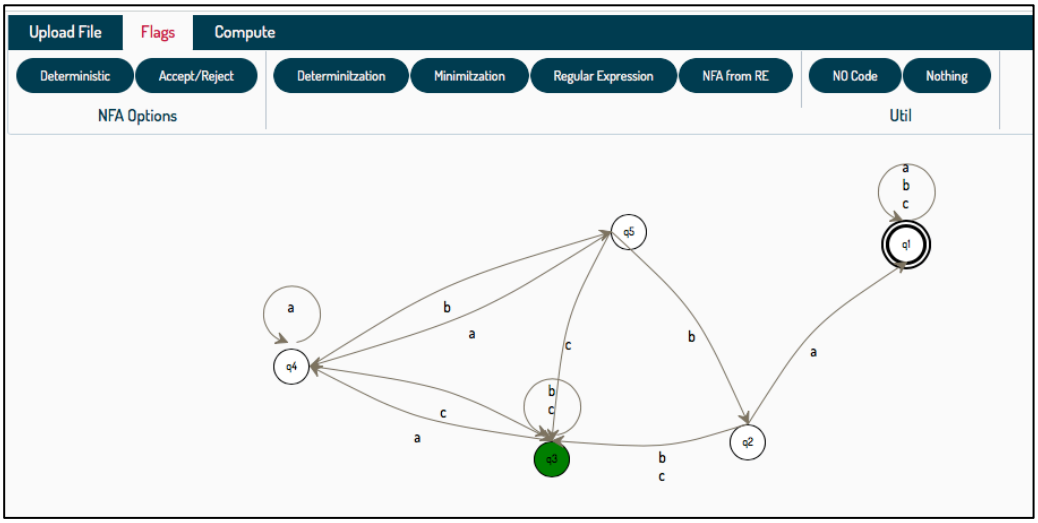


Figura 40 MFA equivalent al DFA

Tal com dèiem abans, el conjunt dels estats finals, s'ha reduït fins a un, fent que entrant qualsevol dels símbols, no ens movem d'ell mateix. Un cop feta la minimització, tal com diu la teoria, un llenguatge és regular si hi ha una expressió regular que el determina i un DFA el reconeix, per tant, obtenim la seva expressió regular. Per fer-ho, hem de clicar en l'agrupació de flags en el botó de Regular expression. Vegem-ne el resultat:

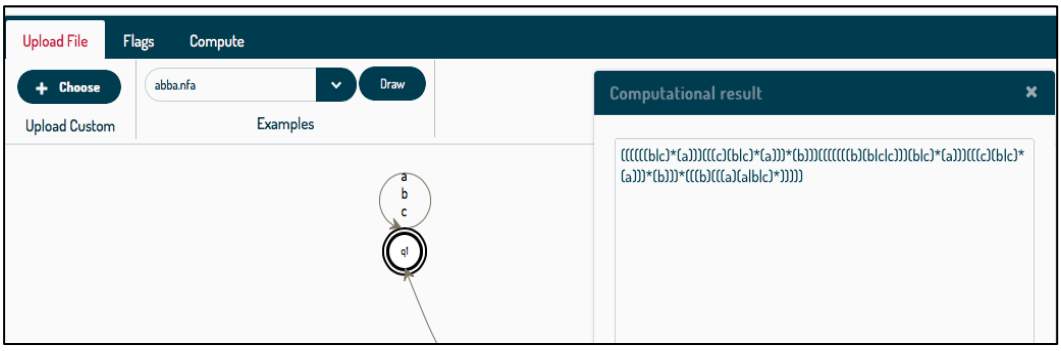


Figura 41 Expressió regular del MFA

Per tal de fer-ne comprensible la lectura, és deixa dibuixat l'autòmat i s'obre una finestra amb l'expressió regular del mateix.

I per últim si volem obtenir el NFA a partir d'una expressió regular cliquem a l'apartat de flags, i després al botó de RE to NFA, se'ns apareixerà una finestra a on podrem escriure la nostre expressió regular i acte seguit podrem veure'n el resultat.

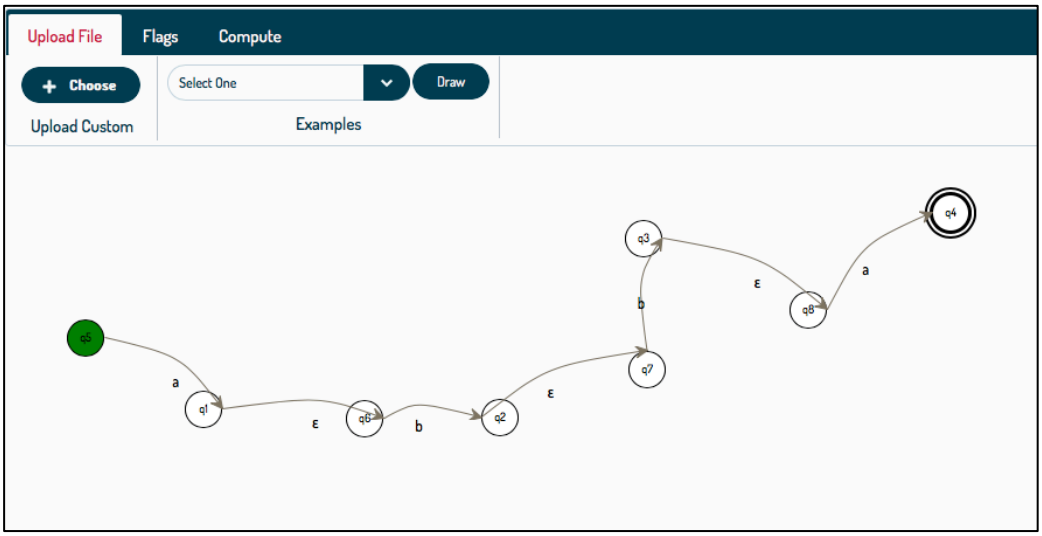


Figura 42 NFA equivalent a la RE

11. Conclusions

Si tenim en compte l'objectiu planejat a l'inici del treball i el punt assolit a la data de finalització, podem dir que s'ha complert les expectatives inicials ja que s'ha pogut crear la plataforma d'ajuda a la docència per tal que: l'usuari o alumne de computació, pugui dissenyar i manipular els autòmats d'una manera visual i gràfica més agradable que des del punt de partida, via línia de comandes o la manera tradicional, mitjançant paper i que com sabem, el bon disseny precisa d'un anàlisi precís, prèviament entès i en paper la possibilitat d'errors i canvis en el disseny genera moltes vegades tornar a començar per tal d'entendre'n bé el disseny. Això fa que la plataforma sigui molt útil per que permet a l'usuari fer i desfer tant com vulgui i no haver de tornar a començar de nou.

Si bé és cert que hi ha hagut problemes i imprevistos durant el desenvolupament del projecte, la implementació de la rutina de pas d'un DFA a RE, per mi va ser el més complex de fer, cosa que vaig haver de dedicar-hi més temps del previst en la planificació explicada anteriorment. Per altre banda, la implementació de l'algoritme de donat un RE a NFA, em va portar menys temps del previst, fet que ha ocasionat que les dues tasques és compensin entre elles, des del punt de vista del temps.

Un cop tancat el projecte, i des d'una perspectiva molt més pràctica i amb més coneixement de causa que en l'inici del treball, he anat prenent consciencia de les moltes possibilitats que ofereix la plataforma tant per la part del usuari com el programador. Òbviament doncs, s'haurien pogut implementar més elements i mòduls, comentats a la secció de de Treball Futur, però degut a la limitació del temps assignat i la impossibilitat d'abastar-ho de forma realista com a TFG, s'han considerat exclusivament les opcions més rellevants pel context de la seva implementació en l'assignatura de "Fonaments de Computació" de la Universitat de Girona.

Per la realització del projecte és imprescindible haver cursat l'assignatura de "Fonaments de Computació", on he pogut accedir a l'estudi i el disseny d'autòmats finits. Sense la base teòrica apresa i estudiada a l'assignatura, aquest projecte no s'hagués dut a terme.

Tanmateix, en base els coneixements del Grau en Enginyeria Informàtica, la part del nucli s'ha pogut portar a bon terme entre d'altres, gracies a l'assignatura de "Metodologia i Tecnologia de la Programació", cursada en el pla vell, en la qual vaig aprendre a programar; a l'assignatura de "Estructures de Dades i Algorítmica" on vaig aprendre tècniques avançades de programació com serien les cerques intel·ligents, buscar la millor estructura de dades per a codificar el nostre objecte, l'ús eficient d'arbres i recursivitat,

etc; la ”Enginyeria del Software I” i ”Enginyeria del Software II”, a on vaig poder entendre i utilitzar els diferents patrons de disseny, dels quals en el projecte n'utilitzo uns quants; I altres assignatures del pla que d'una forma més directa o indirecta m'han ajudat a adquirir el coneixement necessari per poder tirar endavant aquest projecte.

Fruit d'aquesta programació i coneixement, he pogut aprofundir de forma autodidàctica en l'aprenentatge de tot el relacionat amb el Front-End. Aquest apartat m'ha ajudat en la recerca sobre el funcionament, la comprensió de diferents components, la decisió del servidor a utilitzar, etc. Els llenguatge de “Front-End” com el Java Server Faces, el JavaScript, el Phyton, el Ruby, etc. són llenguatges que s'utilitzen amb molta freqüència en el món empresarial i per això m'ha motivat involucrar el JSF en aquest treball. He hagut d'utilitzar molts de tutorials i les ajudes que proveeix el JSF per tal de desenvolupar la part gràfica de la plataforma.

Com a conclusió, puc afirmar sense cap dubte que l'après durant el Grau m'ha servit per desenvolupar la totalitat del projecte i que sense els conceptes adquirits durant el pas per les diferents assignatures això no hagués estat possible. L'aprenentatge d'aquests conceptes i la recerca del nou ha propiciat una millora en la meva inserció en el món laboral, a causa d'haver pogut aprendre i utilitzar diferents eines i llenguatges que ara mateix s'utilitzen en gran mesura.

12. Treball futur

En aquest apartat el que plantejo és fer un llistat de possibles millores de l'eina, o ampliacions de la mateixa, ja sigui de funcionalitat que el client pugui requerir, a millores observades pel propi desenvolupador o bé suggeriments fets pels estudiants que puguin testejar l'aplicació previst pel proper curs.

En una aplicació d'aquestes característiques, i amb un nivell de flexibilitat tan elevat, les possibles millores són moltes. I molt especialment en el camp de la docència que és cap a on està enfocada l'eina. Les propostes actuals immediates podrien ser:

Expressió regular

A l'aplicar l'algorisme de "Generalització" d'un DFA, s'obté una expressió regular, que no necessàriament ha de ser la solució sintàcticament més curta. La millora doncs, consisteix en implementar un mòdul nou per aplicar algoritmes de simplificació a fi de trobar l'expressió regular més curta i fer-la més comprensible obviant haver de reconvertir l'expressió a l'autòmat mínim(fet que si permet fer la plataforma).

Representació d'un autòmat.

La plataforma permet la codificació d'un autòmat seguint un seguit de regles per tal de poder-lo "upload" a la plataforma. El fet que l'usuari hagi primer de codificar l'autòmat i després pujar-lo, pot presentar certes molèsties. Per tant, un dels nous treballs futurs seria crear una nova interfície, per tal que l'usuari pugui directament crear l'autòmat, indicant nodes i arestes online. Això aportaria més comoditats per autòmats petits i no tant còmode per autòmats mitjans o grans. Aquest mòdul ha de permetre doncs que l'usuari dibuixi a la plataforma com si ho fes sobre el paper.

Debug

La plataforma permet executar els autòmats en forma de debug, o sigui, poder executar pas a pas i anar mostrant la configuració del autòmat. Per tal de facilitar-ne l'ús, una millora dels del punt de vista pedagògic, seria trobar una forma més eficient per tal mostrar les configuracions i la comprensió del resultat encarat cap a l'usuari final, ja que ara mateix està encarat als estudiants de l'assignatura de "Fonaments de Computació".

Ajuda

Per tal d'explicar el funcionament de la plataforma, hi ha l'existència d'un arxiu a on s'explica les regles que ha de complir la codificació del autòmat per tal de ser "upload" del autòmat. Aquest arxiu, enfocat hores d'ara cap un conjunt d'usuaris predeterminat com

serien els alumnes de l'assignatura de Fonaments de Computació, potser emprat també per qualsevol altre tipologia d'usuari, no obstant seria molt interesant extendre i ampliar el contingut de l'ajuda per facilitar-en el seu us. Per tant, la millora hauria de ser crear una ajuda més genèrica per tal que tot el conjunt d'usuaris pugui aprendre el funcionament de la plataforma.

13. Bibliografia

[1].The New New Product Development Game. (n.d.). Retrieved January 18, 2016, from https://hbr.org/1986/01/the-new-new-product-development-game

[2] Trello. (n.d.). *Trello*. From Trello: www.trello.com

[3] Sipser, M. (2006). *Introduction to the theory of computation*. Boston : Thomson Course Technology. Retrieved from http://cataleg.udg.edu/record=b1293059~S10*cat

Dfa, A., & State, D. (2005). Reducing a DFA to a Minimal DFA Reducing a DFA to a Minimal DFA Reducing a DFA to a Minimal DFA.

Nfa, a, & Dfa, a. (2005). Converting an NFA to a DFA NFA to DFA Algorithm : Convert NFA to DFA Example. *Analysis*.

Rigau, J. (2014). Fundamentals of Computing Universal Turing Machine, 1–29.

Johanna, H., & Larsson, L. (n.d.). DFA minimisation using the Myhill-Nerode theorem, 1–10.

Informàtica, D. D., & Aplicada, I. M. (n.d.). ALGORÍTMICA :

Seshia, S. a, & Berkeley, U. C. (n.d.). Minimization of DFAs What is Minimization ? Why is Minimization Important ?, 1–15.

JUNG - Java Universal Network/Graph Framework. (n.d.). Retrieved January 17, 2016, from http://jung.sourceforge.net/applet/index.html

PrimeFaces hello world example. (n.d.). Retrieved January 17, 2016, from http://www.mkyong.com/jsf2/primefaces/primefaces-hello-world-example/

PrimeFaces ShowCase. (n.d.). Retrieved January 17, 2016, from http://www.primefaces.org/showcase/ui/data/diagram/editable.xhtml

PrimeFaces Tutorial (Prime Faces for JSF 2) with Eclipse. (n.d.). Retrieved January 17, 2016, from http://www.coreservlets.com/JSF-Tutorial/primefaces/

14. Annex

14.1. Codificació d'un Autòmat.

Per codificar qualsevol autòmat el arxiu ha d'estar codificat en l'estàndard d'ASCII, hi ha de contenir les alfabets (Σ i Γ) i les transicions(δ). El conjunt de transicions estaran escrites sense ordre, només amb l'excepció de la primera transició, la qual ha de tenir d'estat origen el estat inicial de l'autòmat.

Per les TM la codificació ha de complir les següents regles:

- La primera línia hi haurà d'haver l'alfabet Σ: Sigma=[s_1...s_n], la següent aparició de Sigma serà ignorada.
- La segona línia defineix el nou alfabet

$$\Delta = \Gamma - \Sigma - \{\sqcup\}$$

- Codificat com a Delta=[s_1...s_n]. Nota que Σ mai pot ser buit però Δ ho pot ser.
- Tota la informació següent pot ser classificada en tres grups si el primer caràcter de la línia és:
    - | Informació publica.
      La mínima informació sobre MT és: description(descripció informal), funcional definition(definició formal), input (format del input, amb un exemple), output(format del output, amb un exemple), data del disseny, numero de versió, versió del JUTM, nom dels autors i forma de contacte(pex. Email) Per tota aquesta informació s'han d'utilitzar els següents tokens: *Description*, *Function*, *Input*, *Output*, *Date*, *Version*, *JUTM-version*, *Author*, and *Contact* dins d'aquestes línies.
    - [Transició.
      $\delta(q_i, \alpha) = (q_j, \beta, \omega)$ a on $\{q_i, q_j\} \subseteq Q, \alpha \in \Gamma^k, \beta \in \Gamma^{k-1}$, i $\omega \in \Omega^k$ és codifica com $[q_i|\alpha] = [q_j|\beta|\omega]$-Nota per les ETMs $\beta \in \Gamma^k$ i k=1 per les DTMs. Qualsevol altre informació despres d'aquesta no serà llegida.
    - ALPHABET –{|,|}
      Informació privada, qualsevol informació que el desenvolupador consideri.

Les regles per els DFA:

- Primera línia: ídem que les MT
- Segona línia: ídem que les MT.

- Tercera línia: el conjunt d'estats finals. Han d'estar codificats així: $F = [q_0|..|q_n]$.
- I l'únic que canvia és la codificació de transició. La codificació és la següent: $[q_i|\alpha] = [q_j]$, a on q_i i q_j són estats i α és el input a llegir.

Per els PDA les regles són les següents:

- Primera línia: ídem que les MT
- Segona línia: hi haurà l'alfabet de la Pila, i estarà codificat així: Stack=[s_1...s_n]
- Tercera línia: hi haurà el conjunt d'estats finals. Codificats igual que en els NFA.
- l'únic que canvia és la codificació de transició. La codificació és la següent: $[q_i|\alpha\beta] = [q_j|\omega]$, a on q_i i q_j són estats i α és el input a llegir, β és el símbol al cim de la pila i ω és el símbol que empilem.

15. Manual d'usuari i/o instal·lació

15.1. Manual d'instal·lació

Per tal de fer la instal·lació del software necessitem un servidor Glassfish, i només cal afegir el jutm.war que hi ha adjunt en la carpeta "glassfish4/glassfish/domains/"DomainName"/applications/__internal" i engegar el glassfish amb la comanda "asadmin start-domain DomainName" , a on DomainName és el nom del domini que vulguem utilitzar.

I llavors amb un navegador web compatible: http://localhost:8080/jutm/index.xhtml

15.2. Manual d'usuari

El manual d'usuari és basa en la possibilitat de poder penjar els nostres propis dissenys, i poder fer-ne la corresponent computació. Per tal de fer la codificació del nostre disseny, ja està explicat en l'annex 1, el qual explica com han de ser els fitxers i quin tipus de codificació han de seguir els usuari per tal de no tenir problemes de parser.

15.2.1. "Upload" Autòmat.

Per tal de fer el "upload" de la codificació del nostre autòmat, hem clicar en el apartat de "upload", després clicar en el botó "Choose" i buscar en el nostre sistema l'arxiu del autòmat codificat i clicar "Draw". Aquest serà el resultat:

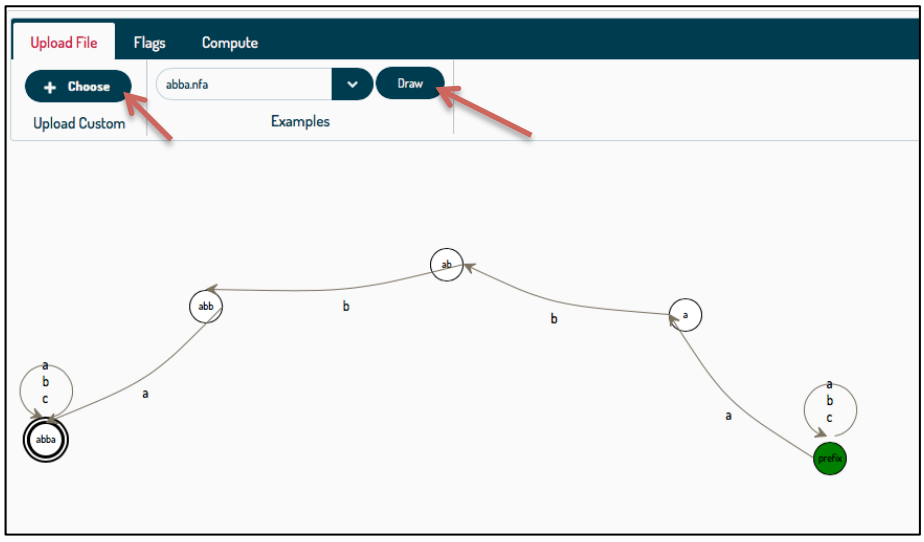


Figura 43 Upload Automat

15.2.2. Computar un Autòmat

Per tal de computar un autòmat, hem d'haver dibuixar qualsevol autòmat diferent, per tant, un cop ho haguem, fet anem a la pestanya "Compute", en el requadre que posa input, hi escrivim el nostre mot o seqüencia d'entrada, en aquesta pestanya podem triar el tipus d'output que volem, i cliquem "compute". Vegem-ne el resultat.

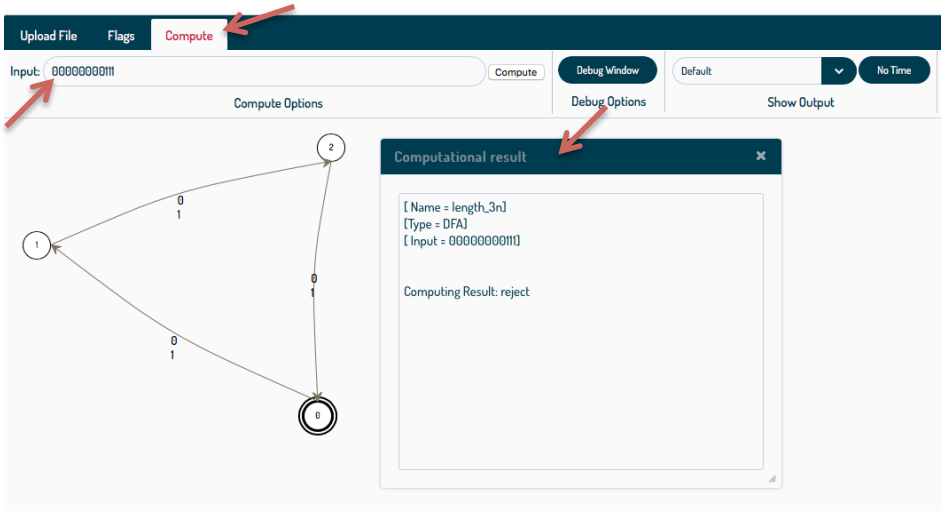


Figura 44 "Compute" Autòmat

15.2.3. Rutines NFA

Per tal, d'aplicar qualsevol de les rutines de determinització, minimització, etc.

Hem d'haver dibuixat un NFA/DFA amb anterioritat. Si ho hem fet, només hem d'anar a l'apartat de Flags, i clicar sobre qualsevol dels quatre botons que hi ha.