# LES GRAPHES DE TRANSITION, L'ANALYSE LEXICALE, L'ETUDE DES CIRCUITS DIGITAUX, LE TEMPS REEL ET TURBO-PROLOG

**Serge MOUTOU**

Le but de cet article est d'introduire des propriétés simples des graphes de transition. Comme le suggère le titre de l'article, cette notion peut être utilisée dans un nombre important de domaines. Nous nous proposons d'en examiner quelques uns ici.

Pour ne pas rester trop abstrait, nous illustrons cet article à l'aide de programmes écrits en Turbo Prolog. La raison de ce choix, est la concision d'écriture que permet ce langage (la recherche d'un chemin dans un graphe de transition se programme en trois lignes !), permettant de ne pas transformer cet article en un "listing".

## I) L'ANALYSE LEXICALE

Nous allons dans un premier temps définir la notion de graphe de transition.

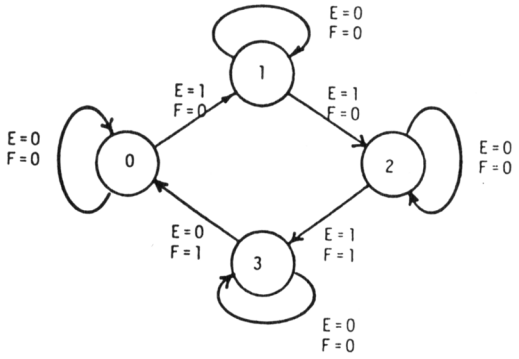
### 1) Définition.

On appellera graphe de transition (ou diagramme de transition) dans la suite du cours un ensemble de données constitué par :

-un ensemble d'états notés souvent par des entiers

-un ensemble d'arêtes ou transitions d'un état vers un autre avec les conditions de franchissement correspondantes appelées étiquettes.

Il existe plusieurs manières de représenter un tel graphe. Nous présentons ci-dessous un schéma simple, ainsi que la liste des transitions, généralement donnée sous forme de tableau appelé table des états.




EXEMPLE DE GRAPHE

et

| E/F | 0/0 | 0/1 | 1/0 | 1/1 |
|---|---|---|---|---|
| 0 | 0 | | 1 | |
| 1 | 1 | | 2 | |
| 2 | 2 | | | 3 |
| 3 | 3 | 0 | | |

TABLEAU DES ETATS CORRESPONDANT
(E et F conditions de transitions)

Si E=0, F=0 on peut passer de l'état 0 à l'état 0 .....

Il existe deux sortes de graphes de transition : déterministes ou non indéterministes. On parle de non déterminisme lorsqu'il est possible d'effectuer plusieurs transitions à partir d'un état, transitions ayant même étiquette, mais conduisant chacune à un état différent. Le graphe donné en exemple est déterministe. Nous n'irons pas plus loin dans la théorie des graphes de transition. Seules nous intéressent maintenant leurs propriétés. Passons donc maintenant aux applications.

**2) Application à l'analyse lexicale.**

Le but de l'analyse lexicale est d'identifier ou de rejeter les mots écrits dans une phrase (d'un programme par exemple)[1]. La relation avec le graphe de transition est alors simple :

Une chaine de caractères est acceptée si elle constitue un chemin possible dans le graphe de transition. Un chemin sera caractérisé par une suite d'étiquettes ou séquence, d'où une déclaration de liste ci-dessous.

Le graphe de transition sera caractérisé par ses arêtes. Tel est le rôle du prédicat **transition(_,_,_)** ci-dessous.

La recherche d'un chemin se fera à l'aide d'un autre prédicat : **chemin(_,_,_)**. La partie déclaration du programme s'impose donc d'elle-même :

```
domains
etat=integer     /* ces deux déclarations peuvent changer */
etiquette=symbol  /* suivant le type d'utilisation du graphe */
sequence=etiquette* /* une sequence est une liste d'étiquettes
                       */
predicates
chemin(etat,etat,sequence)
transition(etat,etat,etiquette)
```

Le graphe sera représenté par une suite de faits, et la recherche d'un chemin reste encore à définir. Nous le ferons de manière récursive :

-condition d'arrêt : lorsqu'il ne reste plus qu'un seul élément dans la liste et que le déplacement demandé peut se réaliser.

-notre chemin reliera un état "Début" à un état "Fin" s'il existe un état "Intermédiaire" tel que l'on a encore un chemin possible entre "Intermédiaire" et "Fin. Un élément de la liste sera alors enlevé.

Si vous avez compris jusqu'ici, ce que je souhaite, vous pourrez transcrire très facilement ce français en Prolog ? Non ! Alors je vous aide :

```
clauses
chemin(Debut,Fin,[Dernier]):-transition(Debut,Fin,Dernier).
chemin(Debut,Fin,[Tete:Queue]):-
            transition(Debut,Intermediaire,Tete),
        chemin(Intermediaire,Fin,Queue).
transition(1,1,a).
transition(1,2,b).
transition(2,2,b).  /* faits décrivant entièrement le graphe */
transition(2,3,c).
transition(3,3,d).
transition(3,1,a).
```

Je vous accorde que le graphe de transition présenté n'est pas des plus passionnants, mais c'est le principe qui compte. Que faisons-nous ensuite ? Nous compilons et, donnons des buts...

Que peut-on demander comme but ?

Nous voulons savoir si l'on peut se déplacer de l'état 1 vers l'état 3 avec la chaine [a,b,c] alors nous écrivons :

**Goal : chemin(1,3,[a,b,c])** répondra **true**. Essayez...

Nous pouvons aussi demander toutes les chaines de caractères de 3 éléments permettant de relier l'état 1 à l'état 3.

**Goal : chemin(1,3,[A,B,C])** répondra**A=a B=b C=c**
**A=b B=b C=c**
**A=b B=c C=d**
**3 solutions**

Ne nous égarons pas, revenons à l'analyse lexicale, à l'aide d'un exemple. Plusieurs améliorations sont possibles pour utiliser un graphe de transition à des fins d'analyse lexicale. J. LAPORTE [2] propose les modifications suivantes

-il définit un prédicat "fin", qui est la fin effective des transitions,

-il pose qu'une chaine est acceptée si elle satisfait à deux conditions :

1-s'il existe pour la chaine un chemin qui débute à l'état initial et finit à la fin.

2-si les arcs formant le chemin correspondent à la chaine.

De telles modifications définissent un *automate fini* qui généralise la notion de graphe de transition. Elles sont souvent nécessaires dans le cas de l'analyse lexicale, et l'exemple que nous allons traiter le montrera. Nous allons proposer la réalisation d'un automate permettant de décider si un nombre positif est un nombre réel. La table des états correspondante est tirée de [3]. Le nombre réel doit être compris entre deux espaces ; il est donné par une liste de caractères ; la virgule est ici remplacée par la notation américaine : le point. Nous proposons l'automate suivant :

```
domains
liste=char* /* la déclaration a été simplifiée par rapport
        au précédent */
predicates  /*programme*/
chemin(symbol,symbol,liste)
transition(symbol,symbol,char)
chiffre(char)
espace(char)
point(char)   /* Notation américaine de la virgule */
```

```
clauses
transition(e1,e2,X) :- espace(X).
transition(e1,f,X) :- chiffre(X).
transition(e1,f,X) :- point(X).
transition(e2,e2,X) :- espace(X).
transition(e2,e3,X) :- chiffre(X).
transition(e2,e4,X) :- point(X).
transition(e3,f,X) :- espace(X). /* les nombres sont entourés */
transition(e3,e3,X) :- chiffre(X)./* par des espaces */
transition(e3,e4,X) :- point(X).
transition(e4,v,X) :- espace(X).
transition(e4,e5,X) :- chiffre(X).
transition(e4,f,X) :- point(X).
transition(e5,v,X) :- espace(X).
transition(e5,e5,X) :- chiffre(X).
transition(e5,f,X) :- point(X).
chiffre('0'). chiffre('1'). chiffre('2'). chiffre('3'). chiffre('4').
chiffre('5'). chiffre('6'). chiffre('7'). chiffre('8'). chiffre('9').
espace(' ')./* la notation 1.5 E12 ne sera pas acceptée comme */
point('.'). /* toutes les puissances de 10 pour simplifier. */
```

Pour la définition de chemin voir ci-dessus.

Le graphe de transition (on peut parler ici encore d'automate) pourra prendre 7 états : e1, e2, e3, e4, e5, v, f. On part toujours de l'état 1. La chaine est acceptée si l'état final est v (pour vrai), elle sera rejetée si l'état final n'est pas v ou si le prédicat est faux.

Exemple d'utilisation :

```
Goal: chemin(e1,X,[' ','.','2',' '])
X=v
1 solution
```

nous apprend que ' .2 ' est un réel positif.

## CONCLUSION :

Nous avons pu nous rendre compte de l'intérêt qu'apportaient les techniques basées sur les graphes de transition dans le cas de l'analyse lexicale. Le champ d'applications de celles-ci peut être étendu à une certaine partie de l'analyse syntaxique [4]. Les grammaires traitées sont dites régulières. Lorsqu'elles ne le sont pas on fait appel à un automate un peu particulier : l'automate à pile [5,6].

## II) ELECTRONIQUE DES CIRCUITS DIGITAUX.

L'utilisation des graphes de transition en électronique digitale n'est pas nouvelle. Il suffit d'ouvrir les références [7,8,9] pour y découvrir ces notations. Nous allons maintenant décrire les problèmes que nous voulons traiter.

### 1) Machines séquentielles.

La notion de machine séquentielle est une notion abstraite donc capable de représenter un certain nombre de problèmes : électronique, automatisme... Pour rester concrets nous allons en donner une définition simple et nous nous contenterons de l'application à l'étude des circuits digitaux.

Nous appellerons alphabet tout ensemble non vide dont nous donnons les éléments :

X={1,0} est un alphabet.

L'ensemble des séquences sur un alphabet X sera noté $X^+$, 01 est une séquence de longueur 2 sur l'alphabet X ....

Une machine sera une correspondance qui associe à toute séquence d'entrée une séquence de sortie, on note M: $X^+$ --> $Y^+$,

X est l'alphabet d'entrée et Y celui de sortie.

Le graphe de transition et ses propriétés nous serviront à l'étude des machines séquentielles. Nous renvoyons le lecteur à une bibliographie plus spécialisée [9] pour la synthèse du graphe à partir du schéma du circuit.

**2) Applications des graphes de transition.**

Prenons le graphe de transition présenté lors de l'analyse lexicale. Quelques changements sont nécessaires pour représenter le nouveau problème à traiter. En effet, un circuit digital est en général composé d'entrées et de sorties, il faut donc prévoir ces dernières. Sans entrer dans les détails (voir le début de l'analyse lexicale) nous donnons le programme suivant :

```
domains
liste=symbol
predicates
chemin(integer,integer,liste,liste)
transition(integer,integer,symbol,symbol)
clauses
chemin(Debut,Fin,[Dernier],[Sortie]):-
   transition(Debut,Fin,Dernier,Sortie).
chemin(Debut,Fin,[Tete:Queue],[Tetes:Queues]):-
   transition(debut,Intermediaire,Tete,Tetes),
   chemin(Intermediaire,Fin,Queue,Queues).
transition(1,1,e0,s1).
transition(1,3,e1,s1).
transition(2,3,e0,s0).
transition(2,1,e1,s1).
transition(3,4,e0,s0).
transition(4,1,e0,s1).
transition(4,2,e1,s1).
transition(5,2,e0,s0).
transition(5,2,e1,s0).
```

Le graphe de transition donné en exemple est tiré de [8]. Les spécialistes auront reconnu les formats d'entrée (un "e" suivi de 1 ou 0) et de sortie (un "s"suivi de 1 ou 0). Ce format permet naturellement de définir des machines à plusieurs entrées ou plusieurs sorties : il suffira de noter e11001 une des cinq entrées possibles, par exemple....

Si l'on veut ainsi savoir ce que va répondre le circuit pour un passage de l'état 1 vers l'état 2 avec comme entrée 1,1,0 on lui demande :

```
Goal : chemin(1,2,[e1,e1,e0],[X,Y,Z]) et il répond :
X=s1, Y=s0, Z=s0
1 Solution
```

Pour bien comprendre l'intérêt de ce qui va suivre, il faut être conscient des problèmes suivants. Lorsqu'un circuit digital représenté par un graphe de transition est mis sous tension, il est rarement possible de prévoir dans quel état il va se trouver : la connaissance de la topographie du circuit ne le permet pas. Pour essayer de trouver cet état nous ne pouvons qu'expérimenter et noter les diverses réponses aux divers stimulis.

Nous allons alors nous intéresser à la recherche des séquences particulières, c'est-à-dire que l'on va examiner la suite de réponses engendrées par une série d'entrées.

a) Séquence de positionnement (homing sequence).

Nous voulons trouver ici l'état final à partir d'une séquence d'entrée et de l'observation de la sortie. Montrons que la séquence 0,0,0 est de positionnement, pour la machine décrite par le graphe de transition donné en exemple. Pour cela lançons le but :

**Goal : chemin(X,Y,[e0,e0,e0],[Z,T,U])**
**X=1, Y=1, Z=s1, T=s1, U=s1**
**X=2, Y=1, Z=s0, T=s0, U=s1**
**X=3, Y=1, Z=s0, T=s1, U=s1**
**X=4, Y=1, Z=s1, T=S1, U=s1**
**X=5, Y=4, Z=s0, T=s0, U=s0**
**5 Solutions**

Cette séquence permet de distinguer les états finaux 1 et 4. Essayez la séquence 0,0,1...

b) Recherche de la séquence de synchronisation
(synchronizing sequence)

La séquence de synchronisation permet d'amener le circuit dans un état connu quel que soit son état initial. Une telle séquence n'existe naturellement pas toujours. Montrons que la séquence 0,0,0,0 en est une.

**Goal : chemin(X,Y,[e0,e0,e0,e0],[Z,T,U,V])**
**X=1, Y=1, Z=s1, T=s1, U=s1, V=s1**
**X=2, Y=1, Z=s0, T=s0, U=s1, V=s1**
**X=3, Y=1, Z=s0, T=s1, U=s1, V=s1**
**X=4, Y=1, Z=s1, T=s1, U=s1, V=s1**
**X=5, Y=1, Z=s0, T=s0, U=s0, V=s1**
**5 Solutions**

montre que cette séquence amène toujours la machine dans l'état 1. Il existe d'autres séquences de synchronisation et nous invitons le lecteur à les chercher. Une application de cette dernière séquence est le reset d'un microprocesseur qui doit amener celui-ci dans un état déterminé pour n'importe quel état dans lequel il se trouve.

**3) Conclusion.**

Comme nous venons de le voir, les propriétés d'un circuit sont liées à son graphe de transition. L'informatique est un outil important pour l'analyse d'un certain nombre d'entre elles. Tel est un des objectifs de la conception assistée par ordinateur (CAO et IAO).

Pour certaines phases de la conception d'un circuit (tests et simulations), il est important d'en avoir une description sous une forme évoluée comme celle présentée. Il faut donc savoir partir d'une description schématique, "compiler" et déduire des propriétés. Mais un éditeur graphique de CAO, pédagogique comme j'ai eu l'occasion de concevoir [10], ou professionnel, ne donne en sortie, après toutes les manipulations graphiques nécessaires, que deux listes :

-la liste des portes utilisées,

-la liste des liaisons entre portes,

et ces données sont souvent insuffisantes.

Nous allons pouvoir découvrir maintenant combien ces derniers problèmes sont proches de ceux posés par le temps réel.

## III) LE TEMPS RÉEL.

### 1)Définitions

La programmation classique de tâches séquentielles gérées par un exécutif temps réel est une programmation asynchrone. La meilleure preuve est toutes les techniques dites de *synchronisation* mises à disposition du programmeur :

gestion des sémaphores, moniteurs...[11]. Ces techniques se révèlent intéressantes pour résoudre les problèmes de contrôles automatiques par calculateurs [12]. Cependant, certains systèmes, les systèmes réactifs [13], ne sont pas facilement programmés par ces techniques.

On appelle système réactif un système déterministe, répondant donc toujours de la même manière à son environnement pour des mêmes séquences d'entrées en provenance de celui-ci. Comme exemples [13], nous donnerons : distributeurs de billets de banque, interfaces claviers-souris, jeux vidéo... Ces systèmes ont donné lieu à une série d'implémentations de langages dits synchrones. Cette hypothèse de synchronisme consiste à dire que les sorties sont fournies de manière synchrone aux entrées, autrement dit que leur calcul ne prend pas de temps. Il est naturellement impossible de réaliser ceci, mais cette hypothèse peut se trouver justifiée pour certaines applications, c'est au concepteur d'en décider.

**2) Les langages synchrones.**

Je ne serais naturellement pas à ma place si je me permettais de présenter un langage dont je n'ai aucune pratique et peu de connaissance. Le point sur lequel je tiens à insister cependant, est qu'un langage synchrone comme ESTERELLE [13] compile un programme en un automate fini, (graphe de transition un peu particulier comme nous l'avons vu), par exploration exhaustive de l'espace des états du programme. Quand on sait qu'il existe des outils informatiques de vérification de propriétés sur les automates, on en voit directement l'intérêt.

LUSTRE fait la même chose, mais la formalisation des problèmes est différente : un programme y est un système d'équations aux suites [14], suites représentant des évènements temporels.

## IV CONCLUSION.

J'espère avoir montré à travers cet article la simplicité pour manipuler des graphes de transition avec un langage comme Turbo Prolog. Ceci étant acquis, il reste à construire des "compilateurs" générants ces automates, soit à partir de descriptions de grammaires ou de circuits (spécifications), soit à partir de mini-langages synchrones. Peut-être aurons-nous le plaisir de lire un jour, un compte rendu de projet de BTS 2I allant dans ce sens ?

## REMERCIEMENTS

Je tiens à remercier ici Monsieur Paul Caspi du Laboratoire Circuits et Systèmes de Grenoble qui a lu avec une attention toute particulière ce manuscrit permettant la correction de certaines erreurs.

Serge MOUTOU
29, rue Lucien Leblanc
10120 Saint-André les Vergers

## BIBLIOGRAPHIE

[1] S. Moutou INTELLIGENCE ARTIFICIELLE ET TURBO PROLOG (3eme partie) A paraître dans INFO 2I (bulletin de liaison de l'AFII).

[2] J. Laporte et D. Delport "TURBO PROLOG. CONSTRUISEZ DES APPLICATIONS" (1987) Eyrolles.

[3] J. Dondoux, "INTRODUCTION À L'INFORMATIQUE",(1980), Armand Collin.

[4] Une analyse syntaxique par automate est proposée par : Th. Papiernik et M. Boukhobza, MICRO SYSTËMES N°79, p181, (1987).

[5] R. Bornat "UNDERSTANDING AND WRITING COMPILERS",Reédition (1986), Macmillan

[6] A. Faure "PERCEPTION ET RECONNAISSANCE DES FORMES",(1985) Editests.

[7] J.M. Bernard "CONCEPTION STRUCTURÉE DES SYSTËMES LOGIQUES"(1987) Eyrolles

[8] A. Miczo "DIGITAL LOGIC TESTING AND SIMULATION", (1986) Harper and Row

[9] D. Mange "ANALYSE ET SYNTHËSE DES SYSTËMES LOGIQUES",(1981) Dunod.

[10] M. Mahieu et S. Moutou, Soumis à publication dans le BULLETIN DE L'UNION DES PHYSICIENS

[11] S. Krakowiak "PRINCIPES DES SYSTËMES D'EXPLOITATION DES ORDINATEURS", (1985),Dunod

[12] Y. Sevely "SYSTËMES ET ASSERVISSEMENTS LINÉAIRES ÉCHANTILLONÉS"(1973) Dunod.

[13] G. Berry, P. Couronne et G. Gonthier, TECHNIQUE ET SCIENCE INFORMATIQUES N° 4,p305,(1987). Toute cette partie sur le temps réel est inspirée de cet article dont je conseille naturellement la lecture à tout le monde.

[14] P. Caspi, N. Halbwachs, TECHNIQUE ET SCIENCE INFORMATIQUES N° 2,p 75, (1986)