Expressions régulières et automates : Une introduction

Guy Tremblay

Hiver 2010

Table des matières

1 Introduction **1**

2 Expressions régulières Unix **1**

3 Expressions régulières Java : La bibliothèque `java.util.regex` **2**
3.1 Les caractères spéciaux de `java.util.regex` . . . 2
3.2 Exemples d'expressions régulières : Jetons d'un langage de programmation . . . 4
3.3 Exemples d'expressions régulières : Validation de données . . . 4

4 Recherche de motifs et expressions régulières **5**

5 Un programme Java utilisant les expressions régulières : `Grep.java` **7**

6 Un autre programme Java utilisant les expressions régulières : Recherche de toutes les `Occurrences` d'un patron **9**

7 Automates **11**
7.1 Représentation graphique d'un automate et sémantique . . . 11
7.1.1 Un automate pour accepter les entiers (sans signe) . . . 11
7.1.2 Un automate pour accepter les nombres réels à point fixe, avec ou sans signe . . . 12
7.2 Représentation graphique simplifiée d'un automate . . . 13
7.2.1 Une autre version de l'automate pour les entiers (sans signe) . . . 13
7.2.2 Une autre version de l'automate pour les nombres réels point fixe, avec ou sans signe . 13

8 Formalisation des concepts sous-jacents aux automates à l'aide d'UML et de l'outil `USE` **14**

9 Spécifications d'automates à l'aide des concepts du modèle `automates.use` **17**
9.1 Spécification d'un automate pour des entiers sans signe . . . 17
9.2 Spécification d'un automate sur des bits . . . 17
9.3 Spécification correcte d'un automate : Vérification des contraintes . . . 18

10 Comment déterminer si un automate accepte une chaîne **18**

A Versions Ruby et Perl des programmes `Grep` et `Occurrences` **24**
A.1 Versions Ruby . . . 24
A.2 Versions Perl . . . 25

B Un programme Ruby utilisant les expressions régulières : Un interpréteur de pseudocode **26**

C Un exemple de programme Perl : La chanson «*99 Bottles of Beer*» **30**

D Automates non-déterministes **32**
D.1 Des automates pour des entiers (sans signe) . . . 32
D.2 Un automate pour des réels à point fixe, avec ou sans signe . . . 32
D.3 Des automates sur des chaînes de bits . . . 33
D.4 Quelques propriétés des automates . . . 34

E Les diagrammes d'état (*statecharts*) du langage UML **36**
E.1 Un diagramme d'état pour un système (simple) de vente en ligne 37
E.2 Autres éléments de notation des diagrammes d'état UML 38
E.3 Un exemple simple de transducteur 39

1 Introduction

Les expressions régulières sont un élément important de la boîte à outils des développeurs de logiciels professionnels [HT00, Goo06].

De telles expressions, qui sont familières depuis longtemps aux programmeurs travaillant dans un environnement Unix [LO97, ZK05], sont aussi disponibles dans de nombreux langages, soit intégrées de façon directe dans le langage — comme c'est le cas avec Perl [WCS96, Bla04] ou Ruby [TH01] — soit disponibles par l'intermédiaire de bibliothèques — comme nous allons le voir pour le langage Java.

Une *expression régulière* est une forme d'expression qui permet de spécifier des ensembles de mots d'un langage — on dit aussi des ensembles de jetons, ou de lexèmes (*tokens* en anglais).

Plus précisément, une expression régulière permet de spécifier un *patron* (on dit aussi un *motif*, en anglais, *pattern*) qui décrit (qui «*match*») un ensemble de mots possibles — ensemble qui peut être vide, n'avoir qu'un unique élément, ou contenir plusieurs éléments.

Pour spécifier un patron à l'aide d'une expression régulière, on utilise certains caractères spéciaux.

2 Expressions régulières Unix

Dans un environnement Unix, les deux principaux caractères spéciaux (au niveau de la ligne de commande) sont «`?`» et «`*`», le premier dénotant n'importe quel unique caractère, l'autre dénotant n'importe quelle suite de (0, 1 ou plusieurs) caractères. On retrouve aussi les crochets et les accolades : «`[...]`», «`{...}`».

Quelques exemples de commandes Unix, qui illustrent ces deux caractères spéciaux, ainsi que certains autres de ces caractères — `[...]` et `{...}` :

> `$ ls a?.*` : Liste (commande `ls`) les noms de tous les fichiers du répertoire courant dont le nom débute par «`a`», suivi d'un caractère arbitraire, suivi d'un «.», suivi d'une suite arbitraire de (0, 1 ou plusieurs) caractères (le «`*`») représentant une extension de fichier (après le «.»).
>
> `$ rm *` : Parfois le cauchemar : Efface **tous** les fichiers `:(`
>
> `$ ls rep[123]/*.*` : Liste les noms des fichiers avec une extension non nulle (`*.*`) dans tous les sous-répertoires (du répertoire courant) dont les nom débutent par «`rep`» suivi du chiffre `1` ou `2` ou `3`.
>
> `$ ls fich[0-9].*` : Liste les noms de tous les fichiers dont le nom débute par «`fich`», suivi d'un chiffre arbitraire, suivi d'un «.», suivi d'une extension arbitraire.
>
> `$ ls M{in,ax}*` : Liste tous les fichiers qui débutent par `M`, suivi par `in` ou `ax`, suivi par zéro, un ou plusieurs caractères arbitraires.
>
> `$ find . -name "*.bak" -print | xargs -I {} rm {}` : Supprime tous les fichiers du répertoire courant et de ses sous-répertoires (`find` effectue implicitement un parcours récursif) qui ont une extension `.bak` : la commande `find` avec l'argument `print` émet les noms de ces fichiers sur `stdout`, lesquels sont ensuite transmis comme arguments à `rm` par l'intermédiaire de la commande `xargs`.

3 Expressions régulières Java : La bibliothèque `java.util.regex`

Dans la bibliothèque `java.util.regex`, inspirée des expressions régulières plus générales de Perl (voir section C) ou Ruby (voir section B), ces caractères spéciaux (on dit aussi «méta-caractères»), sont ceux décrits dans les paragraphes qui suivent.[1]

3.1 Les caractères spéciaux de `java.util.regex`

Classes de caractères

- «`.`» : Un caractère arbitraire, n'importe lequel — en anglais, on dit parfois un *wildcard*.
- «`[`$c_1 \ldots c_k$`]`» : Un ensemble de caractères. Par exemple, «`[aAeEiIoOuUyY]`» représente un patron qui *matche* une voyelle arbitraire, minuscule ou Majuscule. Par exemple, «`[.]`» *matche* le caractère «.» — l'interprétation *wildcard* est perdue à l'intérieur des crochets.
- «`[^`$c_1 \ldots c_k$`]`» : Le complément d'un ensemble de caractères. Par exemple, «`[^02468]`» représente un patron qui *match* n'importe quel caractère *qui n'est pas* un chiffre pair.
- «`[`c_1`-`c_2`]`» : Un caractère arbitraire compris, inclusivement, entre c_1 et c_2. Par exemple, «`[0-9]`» représente un patron qui *match* un caractère qui est un chiffre décimal — donc équivalent à «`[0123456789]`».
- «`[^`c_1`-`c_2`]`» : Le complément (au sens des ensembles) de «`[`c_1`-`c_2`]`», donc n'importe quel caractère *qui n'est pas compris* entre c_1 et c_2 (inclusivement). Par exemple, «`[^0-9]`» *match* n'importe quel caractère *qui n'est pas un chiffre*.
- «`\d`» : Un chiffre — donc équivalent à «`[0-9]`».
- «`\D`» : Un caractère qui n'est pas un chiffre — équivalent à «`[^0-9]`».
- «`\s`» : Un blanc — équivalent à «`[ \t\n\f\r]`».

  Autres caractères spéciaux utilisés ici (en plus de l'espace ordinaire) :
  - «`\t`» : Caractère de tabulation
  - «`\n`» : Saut de ligne (*newline*)
  - «`\f`» : Saut de page (*form feed*)
  - «`\r`» : retour de chariot (*return*)
- «`\S`» : Un caractère qui n'est pas un blanc — équivalent à «`[^\s]`».
- «`\w`» : Un caractère pour un mot (un identificateur) — équivalent à «`[a-zA-Z_0-9]`».
- «`\W`» : Un caractère qui n'est pas valide pour un identificateur — équivalent à «`[^\w]`».

[1] Dans ce qui suit, nous ne décrirons *qu'un sous-ensemble* des fonctionalités offertes par cette bibliothèque. Pour plus de détails, voir `http://java.sun.com/docs/books/tutorial/essential/regex/`.

Caractères indiquant des *frontières*

Les caractères spéciaux suivants dénotent non pas un (ou plusieurs) caractère(s) mais plutôt *une position entre des* (ou *avant/après* un) caractères.

- «`^`» : Un début de ligne — lorsque présent à l'extérieur de «`[...]`».
- «`$`» : Une fin de ligne.
- «`\b`» : La frontière d'un mot — c'est-à-dire, pas à l'intérieur d'une suite de caractères pouvant former un mot. Par exemple, le motif «`\bclasse\b`» pourra *matcher*, dans le texte «`la classe des classes`», le mot «`classe`» mais pas la suite de caractères «`classe`» provenant du mot «`classes`».
- «`\B`» : La frontière d'un non mot.

Caractères indiquant des répétitions (quantificateurs)

- «+» : Une ou plusieurs occurrences du patron qui précède. Par exemple, «`[0-9]+`» est un patron qui décrit une suite de un ou plusieurs chiffres.
- «*» : Zéro, une ou plusieurs occurrences du patron qui précède. Par exemple, «`[a-zA-Z]*`» est un patron qui décrit une suite de 0, 1 ou plusieurs lettres, minuscules ou majuscules.
- «?» : 0 ou 1 occurrence du patron qui précède.
- «`{n}`» : Exactement `n` occurrences du patron qui précède.
- «`{n,m}`» : Entre `n` et `m` occurrences, inclusivement, du patron qui précède — au moins `n` mais pas plus de `m` occurrences.

Autres caractères

- «`|`» : Un choix entre deux motifs. Par exemple, «`a|[0-9]|def`» est un motif qui *match* soit le caractère `a`, soit un chiffre, soit les trois lettres consécutives `def`.
- «\» : Supprime l'effet spécial du méta-caractère qui suit, ou supprime l'interprétation habituelle d'un caractère ordinaire. Par exemple, le motif «`\.`» dénote l'occurrence du caractère «.» et non pas un caractère arbitraire. Par exemple, le motif «`\t`» dénote un caractère de tabulation et non pas le caractère «`t`».
- «`(...)`» : Un groupement d'éléments. De tels groupements sont utiles pour créer des éléments auxquels on peut ensuite référer via des *références arrières* (*back references*).
- «`\1`», «`\2`», «`\3`», etc. : Une référence arrière vers le premier, deuxième, troisième, ... groupe capturé par le motif courant.

  Exemple : Soit l'expression régulière «`([a-z][a-z])([0-9])\2\1`». Ce motif indique qu'on doit tout d'abord trouver deux lettres minuscules, suivies d'un unique chiffre, suivi du même chiffre (le «`\2`»), suivi des deux lettres initiales (le «`\1`»).

  Le méta-caractère `\1` (ou `\2`, etc.) peut donc être vu comme le nom d'une variable, laquelle contient la chaîne exacte qui a été *matchée* par le motif qui correspond au premier (deuxième, etc.) groupe de parenthèses.

  Exemple : Soit l'expression régulière «`\b(\w+)\b.*\b\1\b`». Ce motif indique qu'on désire trouver un identificateur qui apparaît deux fois comme mot distinct et complet (le `\b` indiquant une frontière de mot).

Un motif complexe s'obtient par juxtaposition (concaténation) de patrons. Ainsi, lorsqu'on met côte à côte des motifs, on indique alors que ces motifs doivent apparaître l'un à la suite de l'autre pour qu'un *match* soit trouvé. Par exemple : «`def`» est en fait une suite de trois motifs juxtaposés (concaténés), chacun formé d'une unique lettre.

3.2 Exemples d'expressions régulières : Jetons d'un langage de programmation

Dans les langages de programmation, les expressions régulières sont utilisées pour décrire les mots (jetons) du langage. Par exemple, voici quelques expressions régulières décrivant des jetons du langage Java :

- «`\b(catch|do|if|while)\b`» : L'un des mots-clés `catch`, `do`, `if`, `while`.
- «`[a-zA-Z_][a-zA-Z0-9_]*`» : Les identificateurs.
- «`[A-Z][a-zA-Z0-9_]*`» : Les identificateurs dénotant des noms de classe.
- «`[A-Z_][A-Z0-9_]*`» : Les identificateurs dénotant des noms de constantes (convention où une constante est écrite uniquement en MAJUSCULES).
- «`[0-9]+`» : Les nombres entiers, sans signe.
- «`[+\-]?[0-9]+[.][0-9]+`» : Les nombres réels à point fixe, avec ou sans signe.

  Note : À l'intérieur des crochets, le «.» perd son interprétation spéciale.
- «`//.*$`» : Un commentaire débutant n'importe où sur la ligne puis incluant ensuite tous les caractères subséquents jusqu'à la fin de la ligne.

3.3 Exemples d'expressions régulières : Validation de données

Les expressions régulières peuvent être utilisées pour effectuer certaines formes de validation de données.

Par exemple, supposons qu'un-e étudiant-e doive remplir un formulaire Web et doive fournir son code permanent. Avant de vérifier dans la base de données (coûteux!) si ce code permanent existe vraiment, il est préférable de vérifier au préalable que la *forme* du code permanent fourni *semble correcte*.

Un code permanent (UQAM) est formé comme suit :

- Les trois premières lettres du nom de famille.
- Le première lettre du prénom.
- Le jour de naissance.
- Le mois de naissance, modifié pour tenir compte du sexe de l'étudiant-e : on ajoute `50` lorsqu'il s'agit d'une étudiante.
- Les deux derniers chiffres de l'année de naissance.
- Deux chiffres additionnels, pour assurer l'unicité du code permanent.

Ainsi, une étudiante nommée Nellie Durocher née le 27 janvier 2004 aurait, par exemple, un code permanent ayant l'allure suivante : `DURN27510489`.

En utilisant une expression régulière appropriée, donc sans faire une analyse détaillée des valeurs numériques spécifiées par le jour, mois et année, il est donc possible d'effectuer une «*assez bonne*» (mais pas parfaite) validation d'un code permanent : si le code permanent

fourni par l'étudiant-e ne satisfait pas (ne *match* pas) l'expression régulière, alors on peut être certain que le code permanent est invalide — bien que l'inverse que ne soit pas certain à 100 %.

Question : Quelle expression régulière exprimée dans la notation de `java.util.regexp` permet de représenter le mieux possible un code permanent?

4 Recherche de motifs et expressions régulières

Les expressions régulières sont souvent utilisées pour faire de la *recherche de motifs* — recherche de patrons, en anglais, pour faire du *pattern-matching*.

En gros, étant donné un motif et un texte, il s'agit de déterminer si le motif apparaît dans le texte — souvent en indiquant *tous les endroits* du texte où le motif apparaît.

Les deux programmes Java (`Grep.java` et `Occurrences.java`) présentés dans les sections qui suivent sont des programmes qui effectuent de telles recherches de motifs.

Une approche «*naïve*» pour la recherche de motifs procéderait, en gros, tel qu'indiqué dans le pseudocode 1 — on dit aussi une approche de «*force brute*».

```
ALGORTIHME trouverMotif( motif, texte )
DEBUT
  posMotif ← 0
  posTexte ← 0
  dernierDepart ← 0
  TANTQUE posMotif < longueur(motif) ET posTexte < longueur(texte)
    SI motif[posMotif] = texte[posTexte] ALORS
      # Jusqu'ici ça va!
      posMotif ← posMotif + 1
      posTexte ← posTexte + 1
    SINON
      # Ça ne va plus : on recommence au début du motif.
      posMotif ← 0
      dernierDepart ← dernierDepart + 1
      posTexte ← dernierDepart
    FIN
  FIN

  SI posMotif = longueur(motif) ALORS
    RETOURNER "Motif trouvé à la position " + dernierDepart + " :)"
  SINON
    RETOURNER "Motif pas trouvé :("
  FIN
FIN
```

Pseudocode 1: Algorithme naïf (force brute) pour la recherche d'un motif dans un texte.

Par exemple, soit le motif «`route`» (un mot simple, sans caractère spécial) qu'on désire rechercher dans le texte «`La roue qui roule sur la route.`». L'algorithme naïf procédera alors tel qu'indiqué dans l'exemple d'exécution 1 — les caractères soulignés indiquent les positions du motif et du texte qui, jusqu'au point indiqué, *matchent* les uns avec les autres.

La même idée d'analyse du texte pour trouver le motif peut aussi s'appliquer lorsque le motif est une expression régulière plutôt qu'un mot simple. Le processus de *matchage* est alors évidemment plus complexe — on verra, à la section 7, comment les *automates* peuvent être utilisés pour reconnaître si un mot (texte) correspond à un motif.

```
La roue qui roule sur la route.
route
```

```
La roue qui roule sur la route.
 route
```

```
La roue qui roule sur la route.
  route
```

```
La roue qui roule sur la route.
   route
```

```
La roue qui roule sur la route.
   route
```

```
La roue qui roule sur la route.
   route
```

```
La roue qui roule sur la route.
    route
```

```
La roue qui roule sur la route.
     route
```

```
La roue qui roule sur la route.
      route
```

```
...
```

```
La roue qui roule sur la route.
           route
```

```
La roue qui roule sur la route.
           route
```

```
La roue qui roule sur la route.
           route
```

```
La roue qui roule sur la route.
            route
```

```
...
```

```
La roue qui roule sur la route.
                       route
```

```
La roue qui roule sur la route.
                       route
```

Exemple d'exécution 1: Exemple d'éxécution de l'algorithme naïf de *pattern-matching*.

Soulignons toutefois que cet algorithme naïf, en pratique, n'est jamais utilisé, ni pour des motifs simples, ni pour des expressions régulières, car son efficacité est très (!) mauvaise — $O(n \times m)$, où m est la longueur du motif et n est la longueur du texte. Des algorithmes plus performants sont plutôt utilisés, qui effectuent divers pré-traitements (e.g., construction de structures de données auxiliaires) de façon à effectuer le *pattern-matching* de façon efficace — par exemple, $O(n + m)$ pour la recherche d'un motif simple.

5 Un programme Java utilisant les expressions régulières : `Grep.java`

Programme Java 1 Un programme Java qui réalise une version simplifiée de la commande `grep` du système Unix.

```
import java.io.*;
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class Grep {

   public static void main( String[] args ) throws IOException {
      BufferedReader input = new BufferedReader( new InputStreamReader(System.in) );

      Pattern patron = Pattern.compile( args[0] );

      String ligne;
      while ( ( ligne = input.readLine()) != null ) {
         Matcher matcher = patron.matcher( ligne );
         if ( matcher.find() ) {
            // La ligne contient le patron: on l'imprime.
            System.out.println( ligne );
         }
      }
   }
}
```

Le Programme Java 1 illustre l'utilisation de la bibliothèque `java.util.regex` pour réaliser, en Java, une version simplifiée de la commande `grep` du système Unix — dont la description (`man grep`) est présentée dans l'Exemple d'exécution 2.

```
$ man grep
NAME
     grep - search a file for a pattern

SYNOPSIS
     /usr/bin/grep  [ -bchilnsvw ] limited-regular-expression   [ filename ... ]

DESCRIPTION
     The grep utility searches files for a pattern and prints all lines that contain
     that pattern.
```

Exemple d'exécution 2: Résultat (partiel) affiché par `man grep` sur la machine `arabica.info.uqam.ca`.

Les figures 3 et 4 présentent quelques exemples d'appels de ce programme, dans le premier cas exécutés via les entrées/sorties standards, dans le deuxième cas en utilisant comme données d'entrée le programme `Grep.java` lui-même (via une redirection de l'entrée standard).

```
$ java Grep "^abc$"
abcdef
 abc
123abc
abc
abc
abc
abc
$ java Grep "abc$"
defabc 123
def abc
def abc
abc def
ab c
abc
abc
$ java Grep ".abc.$"
abc
 abc d
abcde
 abcd
 abcd
1234abcd
1234abcd
```

Exemple d'exécution 3: Première série d'exemples d'utilisation du programme `Grep.java`.

```
$ java Grep "ma.*" < Grep.java
    public static void main( String[] args ) throws IOException {
            Matcher matcher = patron.matcher( ligne );
            if ( matcher.find() ) {
$ java Grep "ma" < Grep.java
    public static void main( String[] args ) throws IOException {
            Matcher matcher = patron.matcher( ligne );
            if ( matcher.find() ) {
$ java Grep "[0-9]+" < Grep.java
        Pattern patron = Pattern.compile( args[0] );
$ java Grep "^[^\s]" < Grep.java
import java.io.*;
import java.util.regex.Pattern;
import java.util.regex.Matcher;
public class Grep {
}
$ java Grep "^[\S]" < Grep.java
import java.io.*;
import java.util.regex.Pattern;
import java.util.regex.Matcher;
public class Grep {
}
$ java Grep "[^\w;}]$" <Grep.java
public class Grep {
    public static void main( String[] args ) throws IOException {
        while ( ( ligne = input.readLine()) != null ) {
            if ( matcher.find() ) {
                // La ligne contient le patron: on l'imprime.
$ java Grep "read" < Grep.java
        while ( ( ligne = input.readLine()) != null ) {
$ java Grep "\wread\w" < Grep.java
$ java Grep "\bread\b" < Grep.java
$ java Grep "\bread" < Grep.java
        while ( ( ligne = input.readLine()) != null ) {
```

Exemple d'exécution 4: Deuxième série d'exemples d'utilisation du programme `Grep.java`.

Dans le premier cas, on remarque que chaque ligne qui *match* le motif indiqué est immédiatement réémise sur la sortie standard (lignes avec des caractères soulignés), d'où le dédoublement de certaines lignes — la fin des entrées est indiquée, au clavier, par «`^D`» et n'apparaît dans la figure.

Au niveau du code Java pour le programme `Grep.java`, on constate que la chaîne reçue en argument pour représenter le motif doit être «compilée» (méthode `Pattern.compile`). Le but de cette étape de compilation est de produire un objet de type `Matcher`. À quoi correspond un tel objet? Dans le domaine de l'informatique appelé la «théorie des automates», on sait depuis longtemps qu'à chaque expression régulière correspond un *automate* — on dit aussi *automate fini*, *automate à état fini*, en anglais *finite state automata* : ce sujet sera traité à la section 7.

6 Un autre programme Java utilisant les expressions régulières : Recherche de toutes les `Occurrences` d'un patron

Programme Java 2 Programme Java trouvant toutes les occurrences d'un patron, y compris les occurrences multiples sur une même ligne.

```
import java.io.*;
import java.util.regex.Pattern;
import java.util.regex.Matcher;

// Ce programme trouve (sur stdin) et indique (sur stdout) toutes les
// occurrences d'un motif (specifie sur la ligne de commande). Il
// indique aussi les occurrences multiples sur une meme ligne.

public class Occurrences {


  public static void main( String[] args ) throws IOException {

     BufferedReader input = new BufferedReader( new InputStreamReader(System.in) );

     Pattern patron = Pattern.compile( args[0] );

     int numLigne = 0;
     String ligne;
     while ( ( ligne = input.readLine()) != null ) {
         numLigne += 1;
         Matcher matcher = patron.matcher( ligne );
         while ( matcher.find() ) {
            // La ligne contient le patron: on trouve toutes les occurrences.
            String motifTrouve = matcher.group();
            int debut = matcher.start();
            int fin = matcher.end();
            System.out.println( "Motif trouve a la ligne " + numLigne +
                                " position " + debut + " a " + fin +
                                " : \"" + motifTrouve + "\"" );
         }
     }
  }
}
```

Le Programme Java 2 utilise la bibliothèque `java.util.regex` pour trouver, sur l'entrée standard (`stdin`), toutes les occurrences d'un motif, y compris les *occurrences multiples sur une même ligne.* Chaque fois que le motif est trouvé, on émet sur la sortie standard (`stdout`)

la valeur exacte du motif trouvé, ainsi que sa position dans le texte (numéro de ligne et position sur la ligne).

```
$ java Occurrences "attern" < Occurrences.java
Motif trouve a la ligne 2 position 24 a 30 : "attern"
Motif trouve a la ligne 16 position 2 a 8 : "attern"
Motif trouve a la ligne 16 position 19 a 25 : "attern"

$ java Occurrences "^\s+\b[A-Z]\w+" < Occurrences.java
Motif trouve a la ligne 14 position 0 a 15 : "  BufferedReader"
Motif trouve a la ligne 16 position 0 a 8 : "   Pattern"
Motif trouve a la ligne 19 position 0 a 7 : "   String"
Motif trouve a la ligne 22 position 0 a 12 : "      Matcher"
Motif trouve a la ligne 25 position 0 a 8 : "           String"
Motif trouve a la ligne 28 position 0 a 8 : "           System"

$ java Occurrences ".+(in).+\+*\1" < Occurrences.java
Motif trouve a la ligne 5 position 0 a 40 : "// Ce programme trouve (sur stdin) et in"
Motif trouve a la ligne 12 position 0 a 34 : "    public static void main( Strin"
Motif trouve a la ligne 14 position 0 a 75 : "  BufferedReader input = new\
     BufferedReader( new InputStreamReader(System.in"
Motif trouve a la ligne 20 position 0 a 32 : "  while ( ( ligne = input.readLin"
Motif trouve a la ligne 27 position 0 a 9 : "           int fin"
```

Exemple d'exécution 5: Exemples d'utilisation du programme `Occurrences.java`.

l'exemple d'exécution 5 présente quelques exemples d'appels de ce programme, exécutés en utilisant comme données le programme `Occurrences.java` lui-même

7 Automates

Dans ce qui suit, nous allons nous intéresser aux «automates acceptants» («*recognizers*»), c'est-à-dire à des automates qui reçoivent en entrée une suite caractères et qui déterminent si les caractères ainsi reçus forment un mot valide — on dit alors que la suite de caractères est «acceptée» par l'automate. Il existe diverses autres sortes d'automates, qui peuvent émettre des informations au moment des transitions ou effectuer des actions lors de l'entrée dans un état, mais nous ne les aborderons pas ici.

Plus précisément, pour simplifier la présentation, nous allons nous restreindre à des automates qui acceptent des expressions régulières simples, obtenues principalement par les caractères spéciaux suivants : «`.`», «`[c1-c2]`», «`[^c1-c2]`», «`*`», «`+`», «`?`», «`|`».

7.1 Représentation graphique d'un automate et sémantique

7.1.1 Un automate pour accepter les entiers (sans signe)

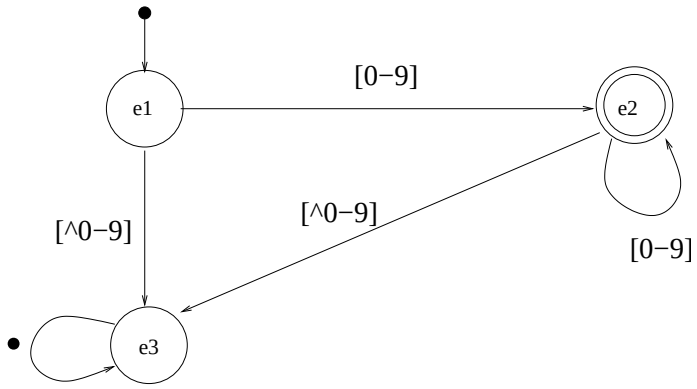

Figure 1: Première version d'un automate acceptant les nombres entiers sans signe.

La figure 1 présente une version graphique d'un automate acceptant (uniquement) les jetons représentant des entiers sans signe, tel que spécifié précédemment par une expression régulière. Des noms sont indiqués sur les états pour pouvoir plus facilement les identifier et, donc, les distinguer les uns des autres ; autrement, pour de tels automates, le nom n'est pas nécessairement très significatif .

L'interprétation d'un tel automate — sa sémantique — est la suivante :

- Un cercle représente un *état*.
- L'état *initial* est celui indiqué par la flèche avec la queue arrondie (petite boule) noire. Dans cet exemple, l'état `e1` est l'état initial — cet état, dans notre modèle d'automate, est toujours unique.
- Les flèches (sans la queue noire arrondie) représentent des *transitions*, indiquant le passage d'un état source (queue de la flèche) à un état cible (tête de la flèche). Notre exemple comporte cinq (5) transitions, dont certaines sont dites «cycliques» — la source et la cible sont identiques, i.e., la transition part d'un état pour aller vers lui-même.
- Lorsqu'un caractère est «lu» (on dit aussi «reçu»), alors on doit «effectuer une transition» qui dépend du caractère reçu. En d'autres mots, on doit suivre la flèche annotée avec un patron simple (décrivant un caractère possible permis) qui *match* le caractère reçu.

Par exemple, lorsque l'automate de la figure 1 est dans l'état `e1` et qu'un chiffre est lu/reçu — caractère qui *match* le patron «`[0-9]`» — alors l'automate effectue une transition (un changement d'état) vers l'état `e2`. Par contre, toujours dans l'état `e1`, si un caractère autre qu'un chiffre est reçu, alors une transition s'effectue vers l'état `e3`, puisque ce caractère *match* alors le motif «`[^0-9]`».

- Un état *acceptant* est indiqué par un double cercle. Si après avoir reçu tous les caractères d'un jeton, l'automate se trouve dans un état acceptant, alors le jeton est *accepté* ; dans le cas contraire, il est *rejeté*.

On appelle *alphabet* l'ensemble des caractères pouvant être reçus et traités par un automate donné — on parle donc «d'un automate sur un alphabet», des automates distincts pouvant traiter des alphabets différents.

Pour que le comportement de l'automate soit bien défini, il faut que pour chaque caractère possible — donc pour chaque caractère de l'alphabet — il y ait une transition qui puisse être effectuée. Comme l'illustre l'exemple de la figure 1, ceci signifie qu'il existe généralement un état *cul-de-sac*, associé aux mots invalides, avec de nombreuses transitions conduisant vers cet état : à partir du moment où le jeton est assurément invalide à cause d'un caractère erroné, il le reste indéfiniment.

7.1.2 Un automate pour accepter les nombres réels à point fixe, avec ou sans signe

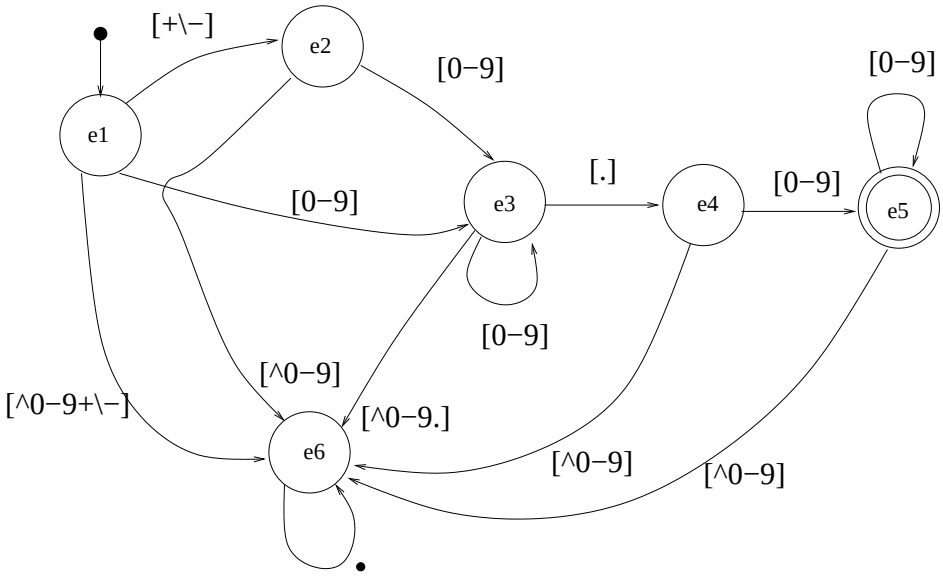

Figure 2: Première version d'un automate acceptant les nombres réels à point fixe, avec ou sans signe.

La figure 2 présente une version graphique d'un automate acceptant les jetons représentant des nombres réels à point fixe, avec ou sans signe (expression régulière présentée précédemment). On constate que lorsque le nombre d'états augmente, le fait que lorsqu'un caractère non acceptable est reçu conduit à l'état d'erreur (cul-de-sac) augmente grandement la complexité de l'automate

7.2 Représentation graphique simplifiée d'un automate

Dans ce qui suit, nous allons utiliser une convention pour permettre de simplifier la représentation graphique des automates, convention consistant à ne pas indiquer explicitement l'état d'erreur (cul-de-sac).

7.2.1 Une autre version de l'automate pour les entiers (sans signe)

La présence de nombreuses transitions conduisant à l'état cul-de-sac d'erreur alourdit énormément la représentation graphique des automates. Pour simplifier cette représentation, on va utiliser la convention suivante :

- On suppose qu'il existe toujours un état d'erreur, qui n'a pas besoin d'être indiqué explicitement.
- L'état d'erreur possède toujours une transition vers lui-même annotée du patron «.» — donc lorsqu'on entre dans cet état, il est impossible d'en ressortir!
- Seules les transitions ne conduisant pas à l'état d'erreur sont indiquées sur le graphique représentant l'automate.
- Si, dans un état donné, le caractère reçu n'apparait dans aucune des transitions indiquées explicitement, alors on effectue une transition, qui n'est pas indiquée de façon *explicite*, vers l'état d'erreur.

La figure 3 illustre l'utilisation de cette convention pour représenter des automates qui acceptent les entiers sans signe.

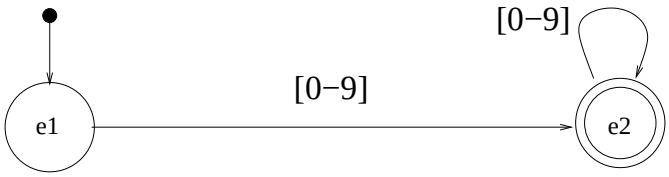

Figure 3: Deuxième version d'un automate acceptant les nombres entiers sans signe, avec état et transition d'erreur implicites.

7.2.2 Une autre version de l'automate pour les nombres réels point fixe, avec ou sans signe

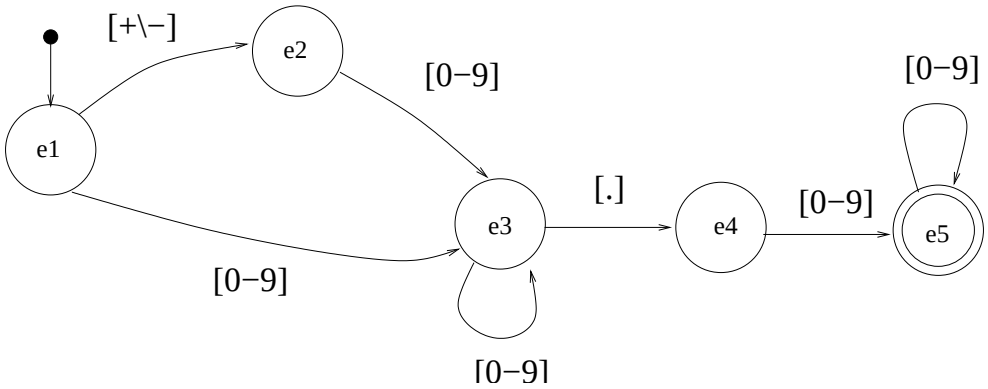

Figure 4: Un automate acceptant les nombres réels à point fixe avec ou sans signe.

La figure 4 illustre l'utilisation de cette convention pour représenter des automates qui acceptent les nombres à point fixe, avec ou sans signe.

Signalons qu'il est possible pour un automate d'avoir plusieurs états acceptants. Par exemple, si l'on voulait définir un automate qui accepte les nombres à point fixe avec ou sans signe, et avec ou sans partie décimale, avec ou sans chiffres après le point — donc soit des entiers, soit des réels, avec ou sans signe dans les deux cas — alors il suffirait de modifier l'automate de la figure 4 pour faire en sorte que l'état `e3` soit lui aussi un état acceptant, en plus de l'état `e5`.

8 Formalisation des concepts sous-jacents aux automates à l'aide d'UML et de l'outil `USE`

Il est possible, en utilisant la notation UML, de formaliser les diverses notions associées aux automates que nous venons de voir. Ainsi, la figure 5 présente un modèle UML décrivant la forme générale de tels automates finis.

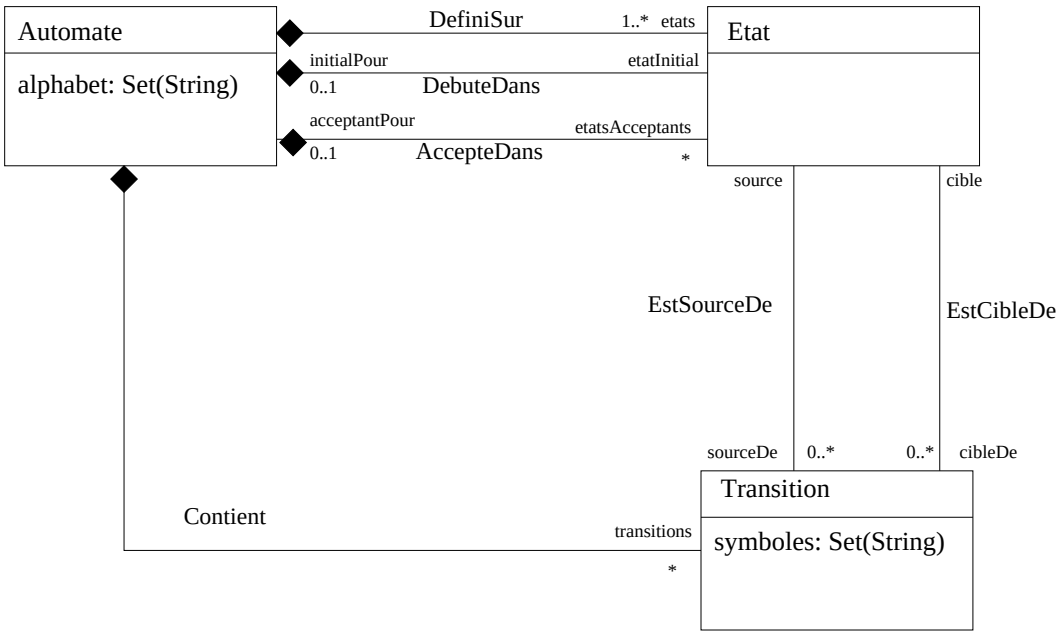

Figure 5: Modèle UML pour des automates finis.

Ensuite, le Modèle `USE` 1 (p. 15) présente ce même modèle défini dans la notation de l'outil `USE`. Plus précisément, les entités et relations définissant les automates finis sont spécifiées ; des contraintes appropriées, exprimées en OCL, seront présentées plus loin.

Les principaux éléments de ce modèle sont les suivants :

- Un automate est `DefiniSur` un ensemble d'états (rôle `etats`) et `Contient` un ensemble des transitions (rôle `transitions`). Un automate est aussi caractérisé (attribut) par l'`alphabet` sur lequel il est défini.
- Un automate possède un unique état initial (rôle `etatInitial`) et un ensemble d'états acceptants (rôle `etatsAcceptants`).

Modèle USE 1 Fichier `automates.use` ($1^{ère}$ partie) : Spécification des entités et relations de la figure 5 (automates finis).

```
model Automates

class Automate
attributes
  nom: String
  alphabet: Set(String)
end

class Etat
end

class Transition
attributes
  symboles: Set(String)
end

association EstSourceDe between
  Etat[1] role source
  Transition[0..*] role sourceDe
end

association EstCibleDe between
  Etat[1] role cible
  Transition[0..*]  role cibleDe
end

association DefiniSur between
  Automate[1]
  Etat[1..*]  role etats
end

association Contient between
  Automate[1]
  Transition[*] role transitions
end

association DebuteDans between
  Automate[0..1] role initialPour
  Etat[1]     role etatInitial
end

association AccepteDans between
  Automate[0..1] role acceptantPour
  Etat[1..*]  role etatsAcceptants
end
```

- Une transition est caractérisée (attribut) par l'ensemble des `symboles` possibles permettant à la transition de s'effectuer. Une transition est aussi caractérisée par une source (rôle `source`) et une cible (rôle `cible`).

  Remarque : Les petits losanges noirs, associés aux relations de l'entité `Automate`, dénotent qu'il s'agit de *relations de composition (forte)* : une entité `Etat`, ou `Transition`, ne peut donc exister qu'en étant associé à un (unique) `Automate`. Bien que cette forme de composition puisse aussi être indiquée avec l'outil `USE`, elle ne l'a pas été, et ce pour simplifier la présentation : nous ne reviendrons pas sur les notions de composition et d'aggrégation dans le cadre de ce cours.

Modèle `USE` 2 Fichier `automates.use` ($2^{ième}$ partie) : Spécification des contraintes sur les entités et relations de la figure 5 et du Modèle `USE` 1.

```
constraints

context Automate
  inv SymbolesSontDesChars:
      alphabet->forAll( s | s.size() = 1 )

context Transition
  inv SymbolesSontDesChars:
      symboles->forAll( s | s.size() = 1 )


context Automate
  inv EtatInitialValide:
      etats->includes( etatInitial )

context Automate
  inv EtatsAcceptantsValides:
      etats->includesAll( etatsAcceptants )


context Transition
  inv EtatsValides:
      automate.etats->includes(cible)
      and
      automate.etats->includes(source)


context Transition
  inv SymbolesSontValides:
      automate.alphabet->includesAll( symboles )
```

Bien que le diagramme UML et les descriptions de classes et de relations qui précèdent nous permettent de modéliser les aspects clés d'un automate, certaines propriétés importantes pour assurer que l'automate soit correctement défini ne sont pas spécifiées. Entre autres :

- Le type `String` a été utilisé pour dénoter un symbole sur lequel les transitions sont effectuées, et ce uniquement parce que le type `Char` *n'existe pas en OCL*. En fait, un symbole est simplement une chaîne *de longueur 1*.
- L'état initial et les états acceptants d'un automate doivent nécessairement être des états de cet automate.

- Les transitions associées à un automate doivent nécessairement relier des états appropriés de cet automate.
- Les divers symboles sur lesquels s'effectuent des transitions doivent nécessairement faire partie de l'alphabet sur lequel est défini l'automate.

Ces contraintes sont formulées, en OCL, dans le Modèle `USE` 2 (p. 16).

9 Spécifications d'automates à l'aide des concepts du modèle `automates.use`

À l'aide des concepts (entités et relations) présentés dans le Modèle `USE` 1, il est possible de définir des automates acceptant des ensembles spécifiques de jetons.

9.1 Spécification d'un automate pour des entiers sans signe

Système concret `USE` 1 Un automate acceptant des entiers.

```
!create ae: Automate
!set ae.alphabet := Set{'0','1','2','3','4','5','6','7','8','9','?'}

!create e1, e2: Etat
!insert (ae, e1) into DefiniSur
!insert (ae, e2) into DefiniSur
!insert (ae, e1) into DebuteDans
!insert (ae, e2) into AccepteDans

!create t1, t2: Transition
!insert (ae, t1) into Contient
!insert (ae, t2) into Contient


!set t1.symboles := Set{'0','1','2','3','4','5','6','7','8','9'}
!insert (e1, t1) into EstSourceDe
!insert (e2, t1) into EstCibleDe

!set t2.symboles := Set{'0','1','2','3','4','5','6','7','8','9'}
!insert (e2, t2) into EstSourceDe
!insert (e2, t2) into EstCibleDe
```

Le Système concret `USE` 1 décrit, à l'aide de commandes `USE` appropriées, l'automate présenté graphiquement à la figure 1 (p. 11) et qui correspond à l'expression régulière «`[0-9]+`».

Remarque : Tous les caractères possibles pour l'alphabet n'ont pas été spécifiés (attribut `ae.alphabet`). Le caractère `'?'` doit donc être vu comme un caractère arbitraire représentant n'importe quel autre caractère qu'un chiffre.

9.2 Spécification d'un automate sur des bits

La figure 6 présente un automate qui accepte des chaînes de bits (donc uniquement des 0 et des 1) qui débutent nécessairement par «11» et qui contiennent un nombre pair de 0 (sans contrainte sur les 1 autres que les deux premiers).

De telles chaînes peuvent être décrites par l'expression régulière suivante :

```
11(1*01*01*)*
```

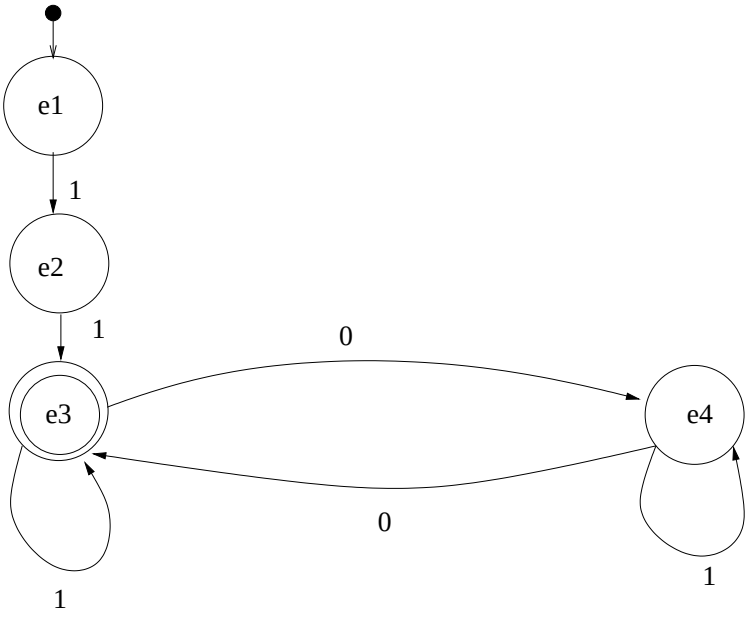

Figure 6: Un automate acceptant des chaînes de bits débutant par «11» et avec un nombre pair de «0».

Le Système concret `USE` 2 (p. 19) décrit, à l'aide de commandes `USE` appropriées, cet automate sur des bits.

On peut voir que la correspondance entre une expression régulière et un automate qui accepte le même langage n'est pas toujours évidente et directe. En fait, il existe un algorithme (que nous n'examinerons pas) qui permet, à partir d'une expression régulière, de générer un automate équivalent, puis de le minimiser, c'est-à-dire de le simplifier le plus possible.

9.3 Spécification correcte d'un automate : Vérification des contraintes

Spécification formelle ⇒ possibilité de **vérification**

C'est lors de la spécification d'un automate concret à l'aide des concepts et contraintes du Modèle `USE` 1 que les contraintes associées aux divers concepts prennent tout leur sens.

Par exemple, supposons qu'on ait une spécification contenant la description concrète d'un automate telle que celle présentée dans le Système concret `USE` 3 (p. 20). Cette spécification (partielle) contient de nombreuses erreurs, lesquelles peuvent effectivement être détectées par l'outil `USE`, tel que cela est illustré dans l'exemple d'exécution 6.

10 Comment déterminer si un automate accepte une chaîne

Spécification formelle ⇒ possibilité de **simulation**

L'intérêt d'avoir une notation précise et formelle permettant de spécifier des automates est, entre autres, qu'il est ensuite possible de «simuler» leur comportement lorsqu'ils reçoivent diverses chaînes de caractères en entrée.

Système concret `USE 2` Un automate acceptant des chaines de bits débutant par «11» et comptant un nombre pair de «0».

```
!create ab: Automate
!set ab.alphabet := Set{'0','1'}

!create e1, e2, e3, e4: Etat
!insert (ab, e1) into DefiniSur
!insert (ab, e2) into DefiniSur
!insert (ab, e3) into DefiniSur
!insert (ab, e4) into DefiniSur
!insert (ab, e1) into DebuteDans
!insert (ab, e3) into AccepteDans

!create t1, t2, t3, t4, t5, t6: Transition
!insert (ab, t1) into Contient
!insert (ab, t2) into Contient
!insert (ab, t3) into Contient
!insert (ab, t4) into Contient
!insert (ab, t5) into Contient
!insert (ab, t6) into Contient


!set t1.symboles := Set{'1'}
!insert (e1, t1) into EstSourceDe
!insert (e2, t1) into EstCibleDe

!set t2.symboles := Set{'1'}
!insert (e2, t2) into EstSourceDe
!insert (e3, t2) into EstCibleDe

!set t3.symboles := Set{'0'}
!insert (e3, t3) into EstSourceDe
!insert (e4, t3) into EstCibleDe

!set t4.symboles := Set{'1'}
!insert (e4, t4) into EstSourceDe
!insert (e4, t4) into EstCibleDe

!set t5.symboles := Set{'0'}
!insert (e4, t5) into EstSourceDe
!insert (e3, t5) into EstCibleDe

!set t6.symboles := Set{'1'}
!insert (e3, t6) into EstSourceDe
!insert (e3, t6) into EstCibleDe
```

Système concret USE 3 Un automate incorrectement spécifié.

```
!create ai: Automate
!set ai.alphabet := Set{'0','1'}

!create e1, e2, e3, e4: Etat
!insert (ai, e1) into DefiniSur
!insert (ai, e2) into DefiniSur
!insert (ai, e4) into DebuteDans
!insert (ai, e3) into AccepteDans

!create t1, t2, t3: Transition
!insert (ai, t1) into Contient
!insert (ai, t2) into Contient

!set t1.symboles := Set{'0','1'}
!insert (e3, t1) into EstSourceDe
!insert (e2, t1) into EstCibleDe

!set t2.symboles := Set{'00'}
!insert (e2, t2) into EstSourceDe
!insert (e3, t2) into EstCibleDe

!set t3.symboles := Set{'3'}
!insert (e1, t3) into EstSourceDe
!insert (e1, t3) into EstCibleDe
```

Ainsi, dans le cas des automates définis dans la notation USE à l'aide du style illustré plus haut (Système concret USE 1 et 2), il est possible de vérifier que diverses chaînes sont acceptées, ou non, par l'automate en définissant une opération auxiliaire appropriée, de type requête, telle que spécifiée dans le Modèle USE 3. L'exemple d'exécution 7 présente quelques exemples d'utilisation de cette requête.
Remarque : Une partie des éléments d'informations affichés lors du chargement des fichiers `automate-*.cmd` a été remplacée par «...» pour alléger la présentation.

```
$ use -nogui automates.use automate-incorrect.cmd
use version 2.4.0, Copyright (C) 1999-2008 University of Bremen
automate-incorrect.cmd> !create ai: Automate
automate-incorrect.cmd> !set ai.alphabet := Set{'0','1'}
automate-incorrect.cmd>
automate-incorrect.cmd> !create e1, e2, e3, e4: Etat
automate-incorrect.cmd> !insert (ai, e1) into DefiniSur
automate-incorrect.cmd> !insert (ai, e2) into DefiniSur
automate-incorrect.cmd> !insert (ai, e4) into DebuteDans
automate-incorrect.cmd> !insert (ai, e3) into AccepteDans
automate-incorrect.cmd>
automate-incorrect.cmd> !create t1, t2, t3: Transition
automate-incorrect.cmd> !insert (ai, t1) into Contient
automate-incorrect.cmd> !insert (ai, t2) into Contient
automate-incorrect.cmd>
automate-incorrect.cmd> !set t1.symboles := Set{'0','1'}
automate-incorrect.cmd> !insert (e3, t1) into EstSourceDe
automate-incorrect.cmd> !insert (e2, t1) into EstCibleDe
automate-incorrect.cmd>
automate-incorrect.cmd> !set t2.symboles := Set{'00'}
automate-incorrect.cmd> !insert (e2, t2) into EstSourceDe
automate-incorrect.cmd> !insert (e3, t2) into EstCibleDe
automate-incorrect.cmd>
automate-incorrect.cmd> !set t3.symboles := Set{'3'}
automate-incorrect.cmd> !insert (e1, t3) into EstSourceDe
automate-incorrect.cmd> !insert (e1, t3) into EstCibleDe
automate-incorrect.cmd>
automate-incorrect.cmd>
use> check
checking structure...
Multiplicity constraint violation in association `Contient':
  Object `t3' of class `Transition' is connected to 0 objects of class `Automate'
  but the multiplicity is specified as `1'.
Multiplicity constraint violation in association `DefiniSur':
  Object `e4' of class `Etat' is connected to 0 objects of class `Automate'
  but the multiplicity is specified as `1'.
Multiplicity constraint violation in association `DefiniSur':
  Object `e3' of class `Etat' is connected to 0 objects of class `Automate'
  but the multiplicity is specified as `1'.
checking invariants...
checking invariant (1) `Automate::EtatInitialValide': FAILED.
  -> false : Boolean
checking invariant (2) `Automate::EtatsAcceptantsValides': FAILED.
  -> false : Boolean
checking invariant (3) `Automate::SymbolesSontDesChars': OK.
checking invariant (4) `Transition::EtatsValides': FAILED.
  -> false : Boolean
checking invariant (5) `Transition::SymbolesSontDesChars': FAILED.
  -> false : Boolean
checking invariant (6) `Transition::SymbolesSontValides': FAILED.
  -> false : Boolean
checked 6 invariants in 0.007s, 5 failures.

.
```

Exemple d'exécution 6: Vérifications des contraintes sur un système concret avec la commande `check`.

Modèle USE 3 Fichier `automates.use` ($3^{ième}$ partie) : Spécification de la requête `accepte`.

```
class Automate

attributes
  alphabet: Set(String)

operations

  accepte( s: String ): Boolean =
    accepteDansEtat( s, etatInitial )

  accepteDansEtat( s: String, e: Etat ): Boolean =
    if s.size() = 0
       then etatsAcceptants->includes(e)
       else
         transitionsAvecSymbole(e, s.substring(1,1))
            ->exists( t | accepteDansEtat( s.substring(2,s.size()), t.cible ) )
    endif

   transitionsAvecSymbole( e: Etat, s: String ): Set(Transition) =
     transitions->select( t | t.source = e and t.symboles->includes(s) )
end
```

```
use -nogui automates.use automate-entiers.cmd
use version 2.4.0, Copyright (C) 1999-2008 University of Bremen
automate-entiers.cmd> !create ae: Automate
automate-entiers.cmd> !set ae.alphabet := Set{'0','1','2','3','4','5','6','7','8','9','?'}
automate-entiers.cmd>
automate-entiers.cmd> !create e1, e2: Etat
...
automate-entiers.cmd> !create t1, t2: Transition
...
automate-entiers.cmd>
use> ?ae.accepte('1001')
-> true : Boolean
use> ?ae.accepte('123.2')
-> false : Boolean
use> ?ae.accepte('abc')
-> false : Boolean
use> ?ae.accepte('12345')
-> true : Boolean
use>
$
$ use -nogui automates.use automate-bits.cmd
use version 2.4.0, Copyright (C) 1999-2008 University of Bremen
automate-bits.cmd> !create ab: Automate
automate-bits.cmd> !set ab.alphabet := Set{'0','1'}
automate-bits.cmd>
automate-bits.cmd> !create e1, e2, e3, e4: Etat
...
automate-bits.cmd>
automate-bits.cmd> !create t1, t2, t3, t4, t5, t6: Transition
...
automate-bits.cmd>
use> ?ab.accepte('11')
-> true : Boolean
use> ?ab.accepte('')
-> false : Boolean
use> ?ab.accepte('1100111')
-> true : Boolean
use> ?ab.accepte('10000')
-> false : Boolean
use> ?ab.accepte('1100000111010')
-> false : Boolean
use> ?ab.accepte('110000011101')
-> true : Boolean
use>
$
```

Exemple d'exécution 7: Exemples illustrant l'utilisation de la requête `accepte` pour détermine si un automate donné accepte, ou non, certaines chaînes.

A Versions Ruby et Perl des programmes `Grep` et `Occurrences`

A.1 Versions Ruby

Script Ruby 1 Un script Ruby qui réalise une version simplifiée de la commande **grep** du système Unix.

```
#!/usr/local/bin/ruby

patron = ARGV[0]

while ligne = STDIN.gets
  print ligne if ligne =~ /#{patron}/
end
```

Script Ruby 2 Un script Ruby pour trouver toutes les occurrences d'un patron, y compris les occurrences multiples sur une même ligne.

```
#!/usr/local/bin/ruby

patron = /#{ARGV[0]}/

numLigne = 0;

while ligne = STDIN.gets

  numLigne += 1

  while m = patron.match( ligne )
    debut = m.begin(0)
    fin = m.end(0)
    motifTrouve = m[0]
    puts "Motif trouve a la ligne #{numLigne} position #{debut} a #{fin} : \"#{motifTrouve}\""
    ligne = m.post_match
  end

end
```

Les script Ruby 1 et 2 présentent les versions Ruby de scripts ayant un comportement semblable aux programmes Java 1 (`Grep.java`) et 2 (`Occurrences.java`).

A.2 Versions Perl

Script Perl 1 Un script Perl qui réalise une version simplifiée de la commande `grep` du système Unix.

```
#!/usr/bin/perl

while (<STDIN>) {
    print if /$ARGV[0]/
}
```

Script Perl 2 Un script Perl pour trouver toutes les occurrences d'un patron, y compris les occurrences multiples sur une même ligne.

```
#!/usr/bin/perl

while (<STDIN>) {
    $numLigne += 1;

    while ( /$ARGV[0]/g ) {
print
    "Motif trouve a la ligne ", $numLigne,
    " position ", length($`), " a ", length($`)+length($&),
    " : \"", $&, "\"", "\n";
    }
}
```

Les script Perl 1 et 2 présentent les versions Perl de scripts ayant un comportement semblable aux programmes Java 1 (`Grep.java`) et 2 (`Occurrences.java`).

B Un programme Ruby utilisant les expressions régulières : Un interpréteur de pseudocode

Le langage Ruby, tout comme `Perl`, est un langage dynamique — on dit aussi «langage de scripts» — qui incorpore directement dans le langage la spécification et la manipulation d'expressions régulières.

Comme l'illustre l'exemple qui suit, les expressions régulières ne servent pas uniquement à faire des recherches de motifs, mais peuvent aussi servir à faire des manipulations et transformations complexes de texte.

Les expressions régulières peuvent aussi servir à faire des manipulations complexes de texte

```
CONST N = 100

FONCTION trouverPos( a, x ) RETOURNE i
DEBUT
  trouve := FAUX
  i := N
  REPETER
    SI a[i] == x ALORS
      trouve := VRAI
    SINON
      i := i - 1
    FIN
  JUSQUE trouve OU i == 0
FIN trouverPos

TABLEAU a[1..N]

i := 0
REPETER
  i := i + 1
  a[i] := 100 - i + 1
JUSQUE i == N


i := trouverPos( a, 23 )

SI i > 0 ALORS
  ECRIRELN "Trouve a la position ", i
SINON
  ECRIRELN "Pas trouve"
FIN
```

Pseudocode 2: Un petit programme, avec une fonction `trouverPos`, écrit en pseudocode (fichier `test7.pc`).

Le pseudocode 2 présente un petit programme impératif simple (avec une fonction auxiliaire `trouverPos`) écrit *dans un pseudo-code*. On aimerait pouvoir évaluer — on dit aussi «interpréter» — ce programme. En choisissant correctement les constructions syntaxiques permises dans ce pseudocode, on peut réussir à l'évaluer *en le transformant en code Ruby exécutable*, et ce à l'aide de diverses transformations fondées sur la reconnaissance et la substitution de motifs.

Script Ruby 3 Un script Ruby pour interpréter un programme écrit en pseudocode (fichier `eval-pc.rb`).

```
#!/usr/bin/env ruby

fich = ARGV[0]
lignes = IO.readlines( fich )

fonctions = Hash.new

mapping_simples = {
  "ALORS" => "then",   "CONST" => "",     "DEBUT" => "",
  "ET" => "and",       "FAIRE" => "do",   "FAUX" => "false",
  "FIN" => "end",      "OU" => "or",      "REPETER" => "while true",
  "RETOURNER" => "",   "SINON" => "else", "SI" => "if",
  "TANTQUE" => "while","VRAI" => "true",
}

lignes.each { |l|
  if l =~ /FONCTION\s+(.+)\((.+)\)\s+RETOURNE\s+(.+)/ then
    fonctions[$1] = $3
    l.sub!( /FONCTION\s+(.+)\((.+)\)\s+RETOURNE\s+(.+)/, 'def \1(\2)' )
  end
  if l =~ /FIN\s+([\w]+)/ then
    l.sub!( /FIN\s+([\w]+)/, fonctions[$1] + "; end" )
  end

  l.sub!( /POUR\s+(.+)\s+:=\s+(.+)\s+A\s+(.+)\s+FAIRE/, 'for \1 in \2..\3 do' )

  l.sub!( /JUSQUE\s+(.+)/, 'break if \1;end' )

  mapping_simples.each { |mot_pseudo, mot_ruby|
    l.sub!( /#{mot_pseudo}([\W]+)/, mot_ruby + '\1' )
  }

  l.sub!( ":=", "=" )

  l.sub!( /ECRIRELN$/, 'print "\n"' )
  l.sub!( /ECRIRELN(.*)/, 'print \1, "\n"' )
  l.sub!( /ECRIRE(.*)/, 'print \1' )

  if l =~ /LIRE(.*),(.*)/
    l.sub!( /LIRE\s*(.*)/, '\1 = STDIN.gets.scan(/[0-9]+/).map{|x| x.to_i}' )
  else
    l.sub!( /LIRE\s*(.*)/,
            '\1[__bornes_inf["\1"]..__bornes_sup["\1"]] = STDIN.gets.scan(/[0-9]+/).map{|x| x.to_i}
                                if __tableaux["\1"];
             \1 = STDIN.gets.to_i unless __tableaux["\1"]' )
  end

  l.sub!( /TABLEAU\s*(.+)\[(.+)..(.+)\]/,
          '\1 = Array.new; (\2..\3).each{|i| \1[i] = 0};
           __tableaux["\1"] = true; __bornes_inf["\1"] = \2; __bornes_sup["\1"] = \3' )
}

defs_aux = ["class Array\n", "def to_s\n", "join( \" \" )\n", "end\n", "end\n"]
defs_aux << ["__tableaux = Hash.new\n", "__bornes_inf = Hash.new\n", "__bornes_sup = Hash.new\n"]

lignes = defs_aux + lignes
lignes = lignes.join
#puts lignes
eval lignes, binding
```

```
class Array
def to_s
join( " " )
end
end
__tableaux = Hash.new
__bornes_inf = Hash.new
__bornes_sup = Hash.new
   N = 100

  def trouverPos( a, x )

    trouve = false
    i = N
    while true
      if a[i] == x then
        trouve = true
      else
        i = i - 1
      end
    break if trouve or i == 0;end
  i; end

  a = Array.new; (1..N).each{|i| a[i] = 0};
  __tableaux["a"] = true;
  __bornes_inf["a"] = 1;
  __bornes_sup["a"] = N

  i = 0
  while true
    i = i + 1
    a[i] = 100 - i + 1
  break if i == N;end


  i = trouverPos( a, 23 )

  if i > 0 then
    print  "Trouve a la position ", i, "\n"
  else
    print  "Pas trouve", "\n"
  end

Trouve a la position 78
```

Exemple d'exécution 8: Code Ruby généré (et évalué) par l'interpréteur de pseudocode pour le programme du fichier `test7.pc` (pseudocode 2).

Le script Ruby 3 présente un tel interpréteur. Un exemple d'utilisation de cet interpréteur avec le fichier `test7.pc` présenté plus haut (pseudocode 2) produit alors le résultat suivant :

```
$  eval-pc.rb test7.pc
Trouve a la position 78
```

L'exemple d'exécution 8 présente le résultat de l'exécution de cet appel si on supprime le caractère «`#`» (commentaire) au début de l'avant-dernière ligne du fichier `eval-pc.rb`, donc si on imprime (`puts`) le contenu du code généré (tableau `lignes`) : le code généré est du code Ruby normal (bien que pas nécessairement très lisible `;)`.

En termes d'expressions régulières, les éléments intéressants à souligner sont les suivants :

- Une expression régulière est indiquée entre des barres obliques — «`/.../`»
- L'opérateur «`=~`» est l'opérateur de *pattern-matching*.

  Plus précisément, l'expression «`v =~ /`*<exprReg>*`/`» retourne `true` si la chaîne `v` (qui peut être une variable) *matche* l'expression régulière indiquée à droite.
- L'instruction «`l.sub!( /`*<exprReg>*`/,` *<chaineDeSubstitution>* `)`» modifie en place (`sub!`) la chaîne `l` en effectuant la substitution indiquée.

  Par exemple, l'instruction Ruby suivante remplace une instruction `POUR` (pseudocode) par une instruction `for` (Ruby) :

```
l.sub!( /POUR\s+(.+)\s+:=\s+(.+)\s+A\s+(.+)\s+FAIRE/,
        'for \1 in \2..\3 do' )
```

C Un exemple de programme Perl : La chanson «*99 Bottles of Beer*»

Le langage Perl [WCS96] est un langage dynamique de script qui supporte, de façon directe et puissante, les expressions régulières — Perl était initialement un acronyme de «*Practical Extraction Report Language*».

Un défaut souvent reproché à Perl est que les programmes résultants peuvent parfois être *très difficiles à lire.* Le script Perl 3 en est une illustration éloquente. Ce programme est un script Perl, tout à fait légal, qui utilise diverses expressions régulières et instructions de *pattern-matching* pour générer le texte de la chanson «*99 Bottles of Beer*» : voir l'exemple d'exécution 9 (produit sur la machine `arabica`)..

Script Perl 3 Un programme pour générer la chanson «*99 Bottles of Beer*».

```
    ''=~(       '(?{'       .('`'       |'%')       .('['       ~'-')
    .('`'       |'!')       .('`'       |',')       .'"'.       '\\$'
    .'=='       .('['       ~'+')       .('`'       |'/')       .('['
    ~'+')       .'||'       .(';'       &'=')       .(';'       &'=')
    .';-'       .'-'.       '\\$'       .'=;'       .('['       ~'(')
    .('['       ~'.')       .('`'       |'"')       .('!'       ~'+')
  .'_\\{'      .'(\\$'      .';=('.      '\\$=|'      ."\|".(      ''`~'.'
 ).((''`')|    '/').').'    .'\\"'.+(    '{'~'[').    (''`|'"')    .(''`|'/'
).('['~'/')  .('['~'/').  (''`|',').(  ''`|('%')).  '\\".\\"'.(  '['~('(')).
'\\"'.('['~  '#').'!!--'  .'\\$=.\\"'  .('{'~'['').  (''`|'/').(  ''`|"\&").(
'{'~"\[").(  ''`|"\"").(  ''`|"\%").(  ''`|"\%").(  '['~(')')).  '\\").\\"'.
('{'~'['').(  ''`|"\/").(  ''`|"\.").(  '{'~"\[").(  '['~"\/").(  ''`|"\(").(
''`|"\%").(  '{'~"\[").(  '['~"\,").(  ''`|"\!").(  ''`|"\,").(  ''`|(',')).
'\\"\\}'.+(  '['~"\+").(  '['~"\)").(  ''`|"\)").(  ''`|"\.").(  '['~('/')).
'+_,\\",'.(  '{'~('[')).  ('\\$;!').(  '!'~"\+").(  '{'~"\/").(  ''`|"\!").(
''`|"\+").(  ''`|"\%").(  '{'~"\[").(  ''`|"\/").(  ''`|"\.").(  ''`|"\%").(
'{'~"\[").(  ''`|"\$").(  ''`|"\/").(  '['~"\,").(  ''`|('.')).  ',.(('{')~
'[').("\["~  '+').("\`"|  '!').("\["~  '(').("\["~  '(').("\{"~  '[').("\`"|
')').("\["~  '/').("\{"~  '[').("\`"|  '!').("\["~  ')').("\`"|  '/').("\["~
'.').("\`"|  '.').("\`"|  '$').("\,".(  '!'~('+')).  '\\",_,\\"'  .'!'.("\!"~
'+').("\!"~  '+').'\\"'.  ('['~',').(  ''`|"\(").(  ''`|"\)").(  ''`|"\,").(
''`|('%')).  '++\\$="})'  );$:=('.')~  '~';$~='@'|  '(';$~=')'~  '[';$/=''';
```

Remarque : Le site Web `http://99-bottles-of-beer.net` est consacré à des programmes qui génèrent cette chanson :

> *This Website holds a collection of the Song 99 Bottles of Beer programmed in different programming languages. Actually the song is represented in 1249 different programming languages and variations.*

Le script Perl 3 est, assurément, l'un des plus étranges![2]

[2] Il faut savoir que ce programme n'a pas été écrit directement à la main, mais a plutôt été généré par un autre programme Perl : voir `http://search.cpan.org/dist/Acme-EyeDrops/lib/Acme/EyeDrops.pm`.

```
99 bottles of beer on the wall, 99 bottles of beer!
Take one down, pass it around,
98 bottles of beer on the wall!

98 bottles of beer on the wall, 98 bottles of beer!
Take one down, pass it around,
97 bottles of beer on the wall!

97 bottles of beer on the wall, 97 bottles of beer!
Take one down, pass it around,
96 bottles of beer on the wall!

.
.
.

48 bottles of beer on the wall, 48 bottles of beer!
Take one down, pass it around,
47 bottles of beer on the wall!

47 bottles of beer on the wall, 47 bottles of beer!
Take one down, pass it around,
46 bottles of beer on the wall!

46 bottles of beer on the wall, 46 bottles of beer!
Take one down, pass it around,
45 bottles of beer on the wall!

.
.
.

3 bottles of beer on the wall, 3 bottles of beer!
Take one down, pass it around,
2 bottles of beer on the wall!

2 bottles of beer on the wall, 2 bottles of beer!
Take one down, pass it around,
1 bottle of beer on the wall!

1 bottle of beer on the wall, 1 bottle of beer!
Take one down, pass it around,
No bottles of beer on the wall!
```

Exemple d'exécution 9: Résultat (partiel) de l'exécution du script Perl 3.

D Automates non-déterministes

Un automate est dit *déterministe* si pour n'importe quel état de l'automate et pour n'importe quel symbole de l'alphabet, il n'existe *qu'une unique transition possible pour ce symbole*. Un automate qui n'est pas déterministe est dit *non-déterministe*.

Pour qu'un mot (une suite de symboles) soit accepté par un automate non-déterministe, il suffit *qu'il soit possible* de parcourir (il suffit qu'il existe) une série de transitions qui permet de se rendre de l'état initial jusqu'à l'état final.

Une forme d'automate non-déterministe souvent rencontrée est celle qui permet d'avoir des *transitions silencieuses* — on dit aussi «transitions invisibles» ou «transitions spontanées». Une telle transition peut s'effectuer *sans consommer aucun symbole de la chaîne d'entrée*. La présence de telles transitions conduit donc naturellement à des automates non-déterministes, puisqu'on peut soit choisir une transition avec un symbole (et consommer ce symbole), soit choisir une transition silencieuse sans consommer aucun symbole.

D.1 Des automates pour des entiers (sans signe)

L'automate de la figure 7 est déterministe, alors que ceux des figures 8 et 9 sont non-déterministes. Dans tous les cas, l'automate accepte les nombres entiers simples (sans signe),

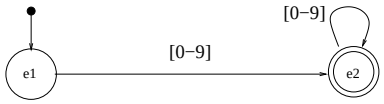

Figure 7: Un automate déterministe acceptant les nombres entiers (sans signe).

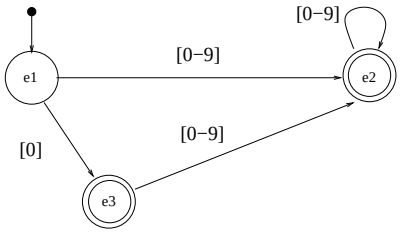

Figure 8: Un automate non-déterministe acceptant les nombres entiers (sans signe).

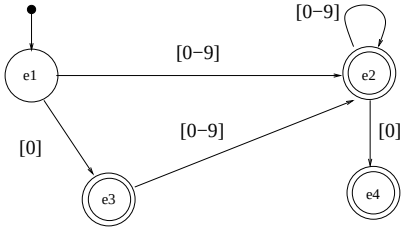

Figure 9: Un autre automate non-déterministe acceptant les nombres entiers (sans signe).

D.2 Un automate pour des réels à point fixe, avec ou sans signe

Il arrive fréquemment que, pour un langage donné, un automate non-déterministe soit plus simple qu'un automate déterministe équivalent.

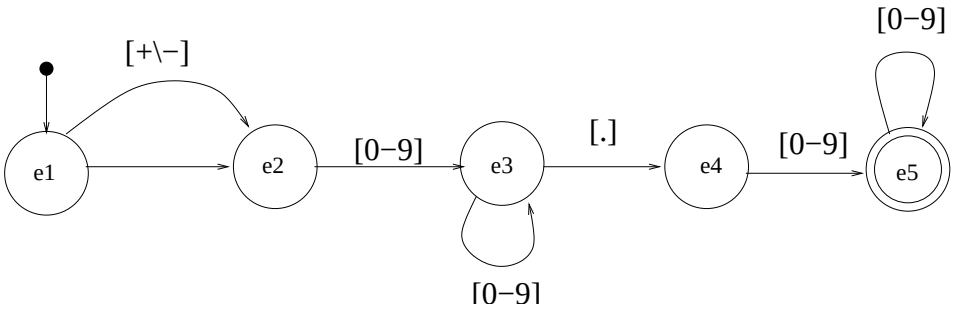

Figure 10: Un automate non-déterministe (avec transition silencieuse) acceptant les nombres réels à point fixe avec ou sans signe.

Ainsi, la figure 10 illustre un automate pour des réels à point fixe, avec sans signe, comportant une transition de moins que l'automate de la figure 4. Comme on le verra dans d'autres exemples, dans certains cas, on peut réduire (parfois grandement) tant le nombre d'états que le nombre de transitions.

D.3 Des automates sur des chaînes de bits

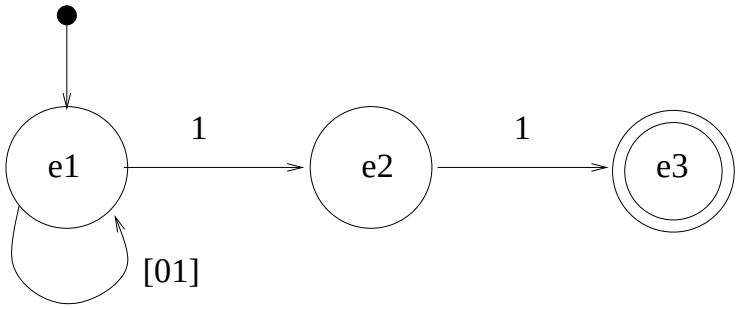

Figure 11: Un automate non-déterministe pour des chaînes de bits se terminant par deux bits «1» consécutifs.

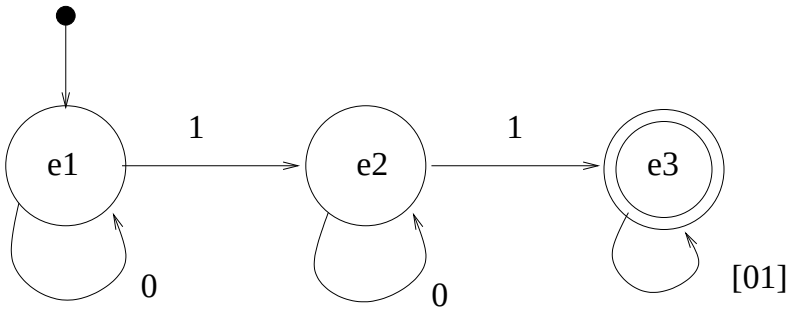

Figure 12: Un automate non-déterministe pour des chaînes de bits qui comptent au moins deux bits «1».

La figure 11 présente un automate non-déterministe (sans transition silencieuse) sur l'alphabet `{0,1}` et qui permet d'accepter les chaînes de bits qui se terminent par deux bits «1» consécutifs. La figure 12, quant à elle, présente un automate non-déterministe (sans transition

silencieuse) sur l'alphabet `{0,1}` et qui permet d'accepter les chaînes de bits qui comptent au moins deux bits «1», n'importe où dans la chaîne.

Exercice : Pour les deux automates qui précèdent, dessinez des automates déterministes qui acceptent exactement les mêmes chaînes de bits.

D.4 Quelques propriétés des automates

Les propriétés suivantes sont connues depuis longtemps [HU79] et sont relativement faciles à démontrer, même si nous ne le ferons pas ici — c'est de niveau baccalauréat en informatique, mais pour un programme avec une orientation plus théorique que celui de l'UQAM :

- Les automates déterministes et les automates non-déterministes ont exactement le même *pouvoir expressif*.

  En d'autres mots, pour n'importe quel automate non-déterministe, il existe un automate déterministe qui accepte exactement le même langage.

  En fait, il existe un algorithme qui permet de produire un automate déterministe équivalent à partir d'un automate non-déterministe.

- Les expressions régulières et les automates permettent d'exprimer les mêmes langages.

  En d'autres mots, pour n'importe quelle expression régulière, il existe un automate qui accepte toutes les chaînes, et uniquement les chaînes, représentées par l'expression régulière.

  En fait, il existe un algorithme qui permet de produire un automate non-déterministe (avec transition silencieuse) à partir d'une expression régulière.

  La figure 13 présente, de façon graphique et simplifiée, les grandes lignes d'un tel algorithme pour transformer une expression régulière arbitraire en un automate non-déterministe (avec transitions silencieuses).

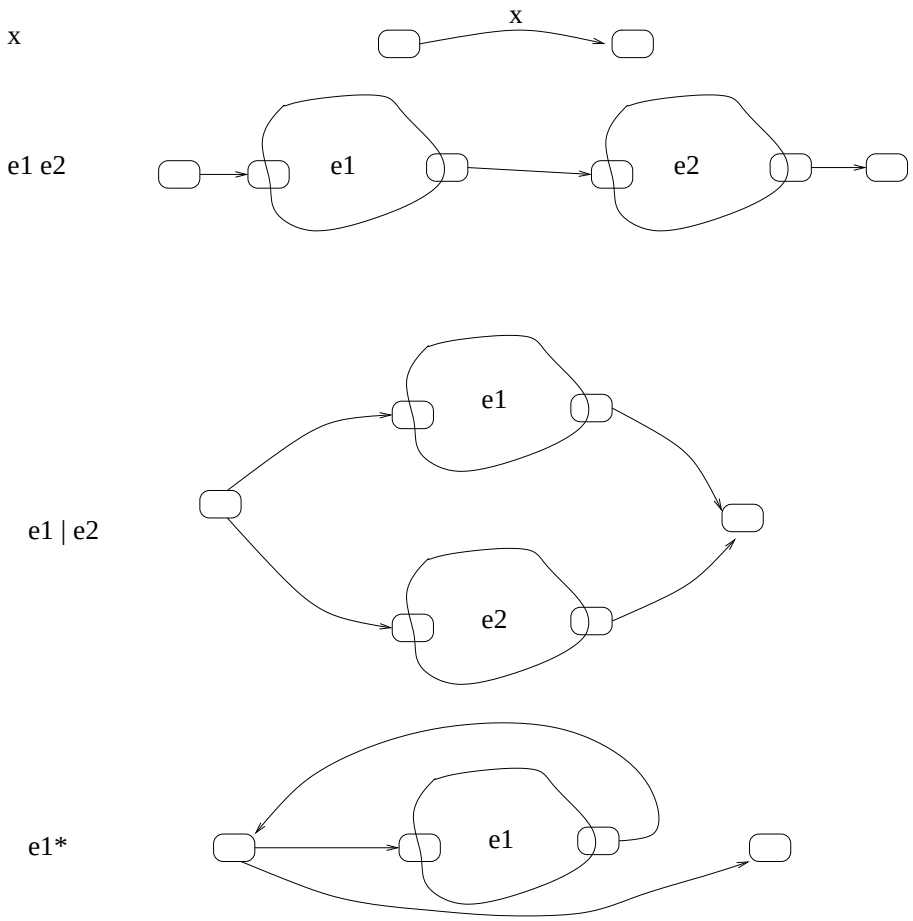

Figure 13: Représentation graphique de la transformation d'une expression régulière en un automate non-déterministe (avec transitions silencieuses).

E Les diagrammes d'état (*statecharts*) du langage UML

Le langage UML comprend un grand nombre de notations différentes, la plupart graphiques — par exemple, diagrammes de classes, diagrammes d'objets — mais aussi certaines notations textuelles — par exemple, OCL. Dans ce qui suit, nous allons jeter un coup d'oeil (rapide) aux diagrammes d'états, aussi appelés *statecharts* [Har88, HG97], une notation graphique d'UML qui permet de représenter des machines à états finis (*finite state machine*), i.e., des automates.

En UML, le rôle d'un *statechart* est de modéliser l'évolution de l'état d'un objet au cours de son cycle de vie, au fur et à mesure que des événements surviennent, événements qui peuvent conduire à des changements d'état. Ces événements, dans le cas d'objets, vont généralement correspondre à des envois de messages (i.e., des appels de méthodes). Alors qu'un diagramme de classes modélise des aspects de la structure *statique* d'un système, un diagramme d'état modélise plutôt certains aspects de son comportement *dynamique*. De plus, un diagramme d'état met principalement l'accent sur l'ordonnancement des événements — dans quel ordre les événements se produisent — plutôt que sur le contenu des activités réalisées par un objet — dans ce dernier cas, on utiliserait plutôt des diagrammes d'activité.

Plus spécifiquement, voici la définition donnée par Booch, Rumbaugh et Jacobson d'une machine à états [BRJ99, p. 290] :

> *A* state machine *is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events. A* state *is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. An* event *is the specification of a significant occurrence that has a location in time and space. In the context of a state machine, an event is an occurrence of a stimulus that can trigger a state transition. A* transition *is a relationship between two states indicating that an objetct in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.*

E.1 Un diagramme d'état pour un système (simple) de vente en ligne

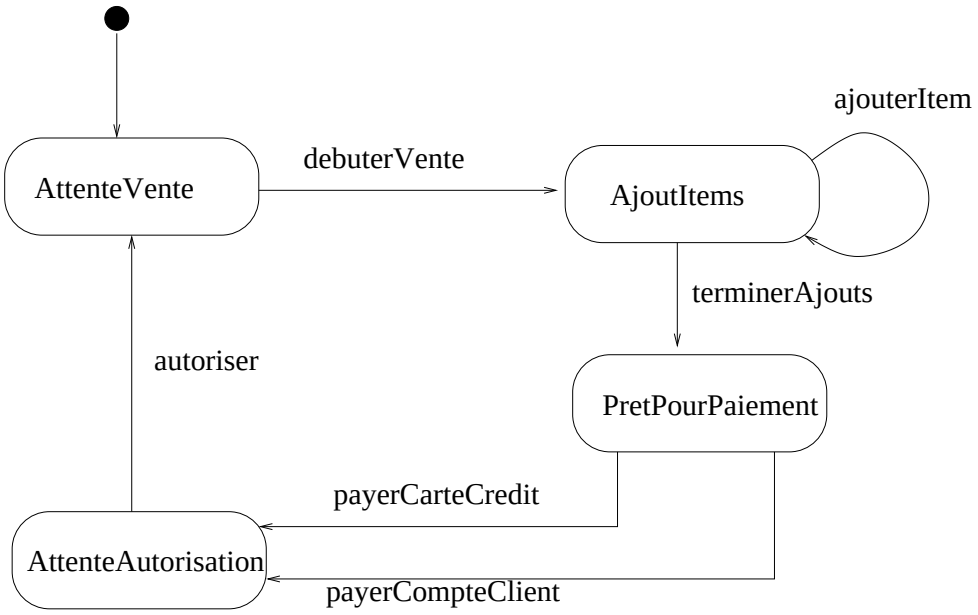

Figure 14: Un diagramme d'état pour un système de vente en ligne.

La figure 14 présente un diagramme d'état pour un système (simplifié) de vente en ligne. Ce diagramme représente l'état d'un objet qui permet gérer de telles ventes. Initialement, l'objet est en attente qu'une transaction de vente soit amorcée. Lorsque l'objet reçoit un message pour débuter une vente, il entre dans l'état permettant d'ajouter des items, ce qu'on peut alors faire plusieurs fois. Lorsque tous les items ont été ajoutés, on doit alors effectuer le paiement, lequel requiert la réception d'une autorisation. Une fois ce cycle terminé, on peut alors amorcer le traitement d'une autre vente.

Comme pour un «automate acceptant», les états et transitions permettent de modéliser ce qui est permis, donc les séquences légales d'événements, qui correspondent à des réceptions de messages. Dans l'exemple ci-haut, ceci signifie, par exemple, qu'il n'est pas possible d'effectuer un paiement tant que la fin des ajouts n'a pas été explicitement signalé.

E.2 Autres éléments de notation des diagrammes d'état UML

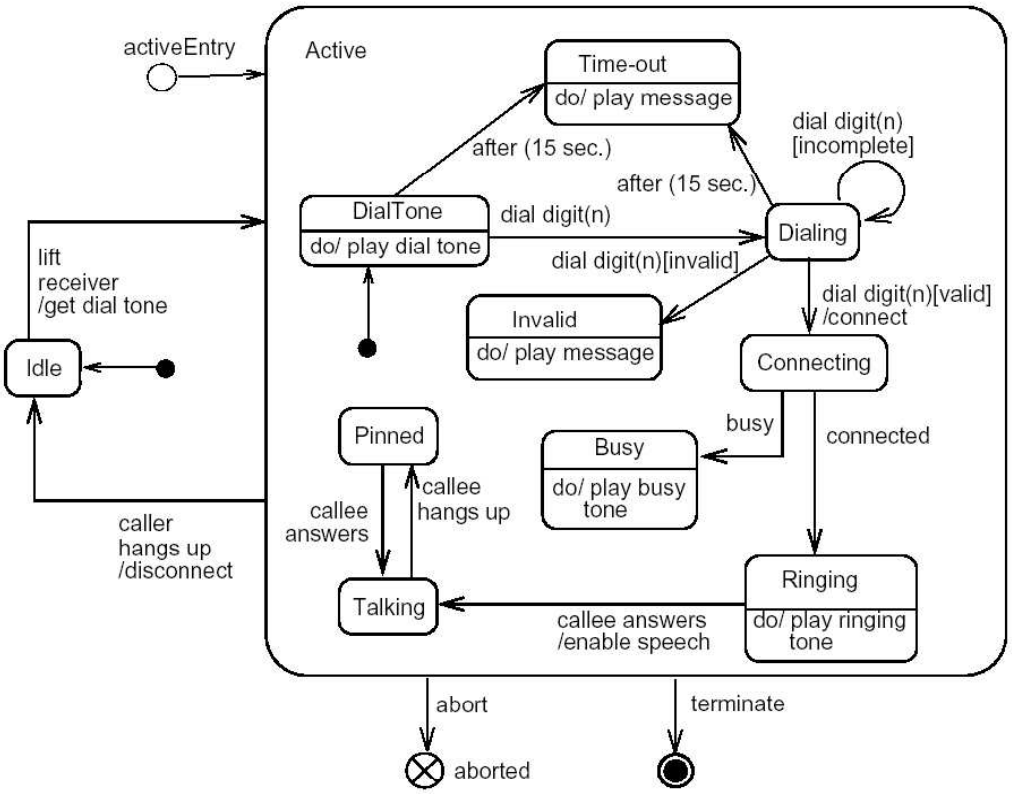

Figure 15.41 - State machine diagram representing a state machine

Figure 15: Un diagramme d'état pour un téléphone.

La figure 15 présente un exemple plus complet et complexe, tiré de «*OMG Unified Modeling Language (OMG UML), Superstructure (version 2.2)*» (p. 569)[3] On voit sur ce diagramme divers éléments nouveaux de notation pour les machines à états finis (mais que nous n'examinerons pas en détail) :

- Lorsqu'une transition s'effectue parce qu'un événement survient, il est possible de générer un événement au moment où la transition s'effectue (symbole après la barre oblique). Un tel événement générer peut, par exemple, correspondre à un envoi de message (un appel de méthode) effectué par l'objet en réaction à l'événement.
- Une garde (condition booléenne sur une transition, entre crochets) peut être présente sur une transition, assurant ainsi que la transition ne s'effectuera que si la condition est vérifiée.
- Les états peuvent être organisés de façon hiérarchique, i.e., un état peut contenir des sous-états.

[3] `http://www.uml.org`

- Lorsqu'on entre dans un état, on peut indiquer une activité à exécuter tout au long de la présence dans l'état (instruction `do`).
- Le diagramme d'état peut terminer sa série de transitions de différentes façons (i.e., succès ou exécution avortée).
- On peut associer des délais d'attente à des transitions silencieuses.

De nombreux autres éléments de notation sont aussi possibles, entre autres, la présence d'états *concurrents* (pas présents dans l'exemple).

E.3 Un exemple simple de transducteur

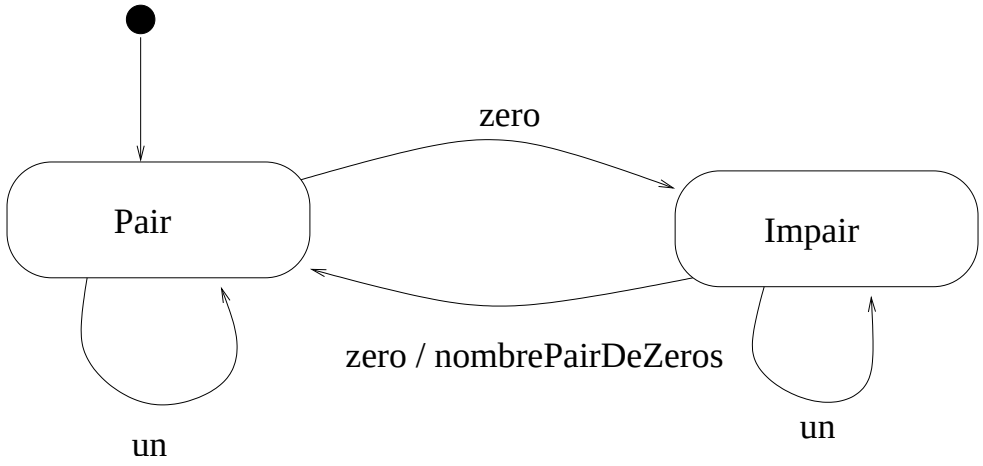

Figure 16: Un diagramme d'état pour un transducteur simple (style machine de Mealy).

La figure 16 présente un exemple simple, dans la notation UML, d'un *transducteur*. La machine reçoit des signaux dénotant la réception d'un bit 0 (`zero`) ou d'un bit 1 (`un`). Lorsqu'un 0 est reçu est que le nombre total de 0 reçus est pair (mais non nul), alors un signal approprié (message `nombrePairDeZeros`) est émis.

Contrairement aux automates acceptants tels que ceux vus précédemment, il n'y a pas ici de notion d'état «acceptant» : c'est au moment où la transition s'effectue qu'un événement approprié est généré. Un automate où les actions se font au cours des transitions est dite «machine de Mealy».

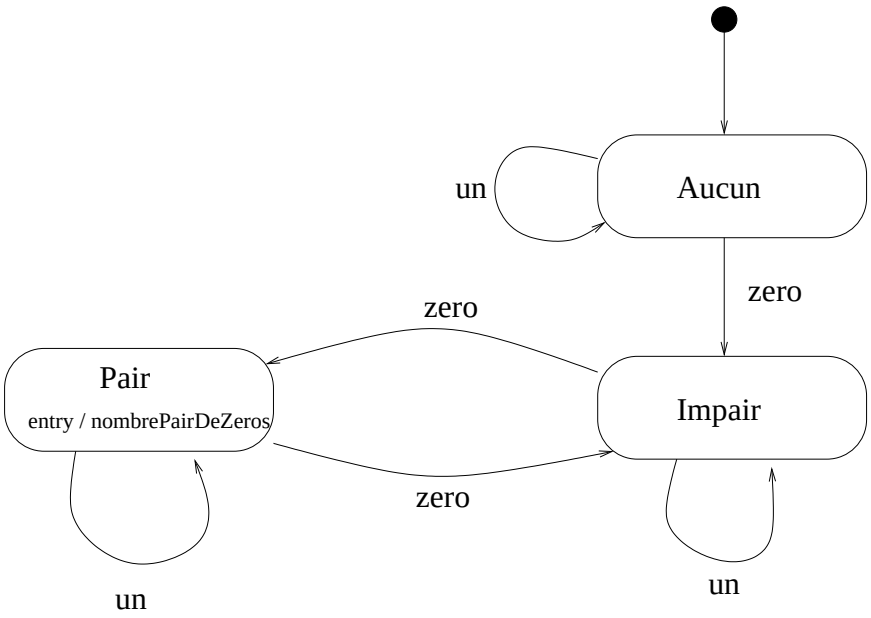

Figure 17: Un diagramme d'état pour un transducteur simple (style machine de Moore).

Il existe aussi des automates où les actions sont plutôt associées aux états, auquel cas on parle alors de «machine de Moore». Ce style de machine peut aussi être représenté dans la notation UML, comme l'illustre la figure 17.

Références

[Bla04] C. Blaess. *Scripts sous Linux—Shell Bash, Sed, Awk, Perl, Tcl, Tk, Python, Ruby...*. Eyrolles, 2004. [QA76.76O63B4899].

[BRJ99] G. Booch, J. Rumbauch, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999. [QA76.76D47.B665].

[Goo06] P. Goodliffe. *Code Craft—The Practice of Writing Excellent Code*. No Starch Press, 2006.

[Har88] D. Harel. On visual formalisms. *Comm. of the ACM*, 31(5):514–530, May 1988.

[HG97] D. Harel and E. Gery. Executable object modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[HT00] A. Hunt and D. Thomas. *The Pragmatic Programmer—From journeyman to master*. Addison-Wesley, Reading, MA, 2000. [QA76.6H858].

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979. [QA267H56].

[LO97] M. Loukides and A. Oram. *Programming with GNU Software*. O'Reilly, 1997.

[TH01] D. Thomas and A. Hunt. *Programming Ruby: The Pragmatic Programmer's Guide*. Addison-Wesley, Reading, MA, 2001.

[WCS96] L. Wall, T. Christiansen, and R.L. Schwartz. *Programming Perl (2nd Edition)*. O'Reilly, 1996.

[ZK05] A. Zeller and J. Krinke. *Essential Open Source Toolset*. John Wiley & Sons, Ltd, Chichester, UK, 2005.