Expérience de programmation générique sur des structures non-séquentielles : les automates

Vincent Le Maout

▶ **To cite this version:**

Vincent Le Maout. Expérience de programmation générique sur des structures non-séquentielles : les automates. Génie logiciel [cs.SE]. Université de Marne la Vallée, 2003. Français. NNT : . tel-00720666

HAL Id: tel-00720666

https://theses.hal.science/tel-00720666

Submitted on 25 Jul 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université de Marne La Vallée

Thèse de doctorat

Spécialité : informatique

présentée par
Vincent Le Maout

sujet

Expérience de programmation générique sur des structures non-séquentielles : les automates

directeur de thèse
Maxime Crochemore

Résumé

Cette thèse constitue une contribution au génie logiciel appliqué à la programmation d'automates. Elle est née d'abord du simple besoin concret d'une bonne librairie de manipulation d'automate ouverte, efficace, extensible et simple d'utilisation. À priori, la programmation générique en C++ semblant la plus adaptée à ces besoins (son originalité par rapport à la programmation purement objets, vient de ce qu'elle impose des contraintes sur les temps de calcul des opérations), le but était d'étendre ces techniques de programmation et leurs domaines d'application aux automates puis aux machines à états en général tout en rendant la librairie plus abordable grâce à la programmation générative et des composants actifs. Ce travail couvre de plus un besoin d'étude sur la programmation par patron en C++ : faisabilité, pouvoir d'expression et efficacité.
Ce thèse espère apporter une solution viable à la programmation générique d'automates avec des concepts novateurs, une implémentation sous forme de librairie C++ ouverte et une expérimentation de programmation générative généralisant les résultats obtenus à l'ensemble des machines dont la structure est basée sur celle d'un graphe.

Abstract

This thesis is a contribution to software engineering applied to automata programming. It started with the simple need of an open automaton manipulation software, efficient, extensible and easy to use. Generic programming in C++ seemed to be the right technique for those needs (the constraints on the time complexity of each operation are an help to achieve the quality goals). The aims are to extend these generic techniques to automata manipulation, then to all state machines and to make the library accessible with generative programming and active components. This work covers the need for an evaluation of the efficiency, expression power and utilisability of template programming in C++.
This thesis hopes to bring a viable solution to template programming of automata with novel concepts, open implementation in C++, and an experiment of generative programming that extends obtained results on automaton to all kinds of state machines.

Table des matières

Table des figures

Chapitre 1

Introduction

Did I mention my belief in the true meaning of « intelligence » ?
« Intelligence is the ability to avoid doing work, yet get the work done ».
Lazy programmers are the best programmers.

Linus Torvalds

Le travail présenté dans ce mémoire constitue une contribution au génie logiciel appliqué à la programmation d'automates. Les automates, depuis leur introduction n'ont eu cesse de prouver leurs intérêts théorique et pratique en particulier dans le domaine du traitement des langages grâce aux travaux notamment de J. Hopcroft et D. Ullman [24], D. Perrin [50] et M. Gross [23] pour le traitement de la langue naturelle. Il a démarré avec plusieurs objectifs concurrents. D'abord un simple besoin concret : l'absence de bonne librairie de manipulation d'automate, le besoin étant d'avoir une librairie *ouverte* capable de gérer de nouveaux types d'automate, *efficace*, les automates utilisés sont très grands (jusqu'à plusieurs millions d'états et de transitions), *extensible*, les opérateurs de manipulation sont variés et évoluent avec les progrès de la linguistique (pour les automates appliqués au traitement de la langue naturelle), et enfin *simple d'utilisation* étant donnée l'aridité des librairies disponibles. À priori, la programmation générique en C++ semblant la plus adaptée à ces besoins, le but était donc d'étendre ces techniques de programmation introduites par D. Musser [44, 45] et leurs domaines d'application aux automates puis aux machines à états en général, étant donné l'abscence de librairie répondant à ces quatres critères fondamentaux. Parallèlement, les résultats plus qu'encourageants obtenus sur les structures de données séquentielles par A. Stepanov [67] et intégrés à la librairie standard du C++ (Standard Template Library [27, 26]) ont fortement incité la communauté des développeurs à généraliser ces paradigmes à l'ensemble du champ d'application de la conception logicielle, c'est-à-dire les structures et algorithmes de base sur les séquences (A. Stepanov et D. Musser [41]), le calcul matriciel (J. Siek et A.

Lumsdaine [60]), les bases de données (M. Gradman et C. Joy [22]), le calcul géométrique (U. Köthe et K. Weihe [33], H. Brönnimann, L. Kettner, S. Schirra et R. Veltkamp [9]), la biochimie ainsi que la biologie moléculaire (D. Moss, A. Bleasby et W. Pitt [38]) et les graphes (M. Forster, A. Pick et M. Raitner [16], J. Siek, L-Q. Lee et A. Lumsdaine [59]). Mais ces derniers n'ont pas profité pleinement des avancées en matière de généricité et ne constituent qu'une ébauche de solution. En 1998, D. Kühl [30] introduit des principes généraux de programmation générique de graphe que deux librairies, GTL [16] et BGL [59], enrichissent sans aller assez loin dans l'abstraction et l'application rigoureuse du paradigme. Leur manque de généricité et leur utilisation malaisée ont incité à développer une librairie plus adaptable grâce à la programmation générique et à la fois plus facile d'utilisation et plus générale grâce à la programmation générative et les librairies actives introduites par K. Czarnecki [11].

Ce travail couvre de plus un besoin d'étude : les techniques de programmation générique par patron en C++ étaient une nouveauté dont les éléments techniques, faisabilité, pouvoir d'expression et efficacité, étaient peu connus et pourtant semblaient prometteurs. Sont-elles simples à mettre en œuvre ? les compilateurs sont-ils à même de gérer toutes les constructions mathématiquement possibles ? Il faut noter qu'au cours de ce travail les compilateurs C++ ont considérablement évolué ce qui a entraîné de nombreuses refontes du code pour s'adapter aux nouveaux standards. À l'heure qu'il est, une standardisation officielle du C++ a très fortement arrondi les angles.
Plus théoriquement, quelles sont les conséquences de la programmation générique sur les méthodes de développement et quelles méthodes de développement doit-on mettre en place pour réaliser du générique ? Quelles sont les approches conceptuelles qui permettent de concevoir du code générique ? Peut-on dégager des schémas et des patrons de conception (*design patterns*) propres à la programmation générique ?
Enfin, il s'agissait de fournir une contribution à l'enrichissement et au développement du logiciel libre avec du code ouvert sous licence LGPL (*Lesser GNU Public License*[1]).

Cette thèse espère apporter une solution viable à la programmation générique d'automates avec des concepts novateurs, une implémentation sous forme de librairie C++ et une expérimentation de programmation générative généralisant les résultats obtenus à l'ensemble des machines dont la structure est basée sur celle d'un graphe.

1.1 Fondements de la programmation générique

En 1976, N. Wirth [81] écrivait :

$$\text{Algorithms} + \text{Data Structures} = \text{Programs}$$

soulignant la dichotomie fondamentale algorithme/structure de données. La programmation générique s'attaque au « + » de cette équation dans un but de réutilisation et d'efficacité : réutilisation pour des raisons évidentes de temps et de coût de développement, efficacité pour ne pas dégrader les performances des programmes et ne pas régresser. Concevoir du code générique signifie se donner les moyens d'être fainéant, c'est-à-dire pouvoir combiner de manière arbitraire des composants et les adapter à de nouveaux contextes d'utilisation sans avoir à

[1] `http ://www.gnu.org`

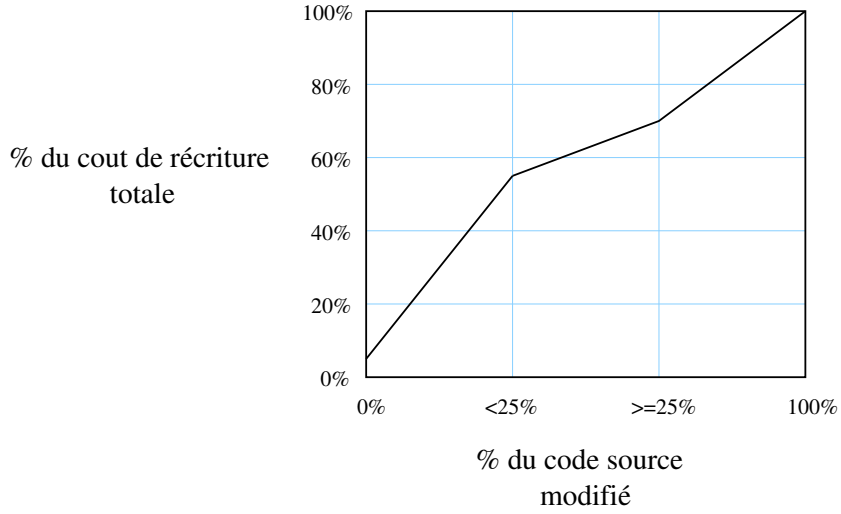

FIG. 1.1 – Coût de la réutilisation logicielle [57, 3]

toucher au code et ce, en assouplissant la manière dont les éléments d'un programme communiquent et interagissent entre eux.

La figure 1.1 tirée de [57] et [3] représente une évaluation empirique du coût de modification d'un composant par rapport au coût de sa récriture complète. Une modification même mineure suppose de comprendre en profondeur le fonctionnement du module et modifier un code mal structuré peut entraîner des changements profonds dans l'implémentation (une modification d'environ 25% des fonctionnalités demande plus de travail que la moitié de ce qui est nécessaire à une refonte complète du composant). Dès lors, la problématique consiste à architecturer un programme de façon à ce qu'il soit adaptable sans effort à un grand nombre de situations. Le résultat d'une réutilisation facilitée et accrue ne se limite pas à une diminution de l'effort et du temps de développement ; il apporte une augmentation substantielle de la qualité et de la fiabilité des applications puisque le temps gagné à l'écriture peut être consacré aux tests et au débogage qui du coup y gagnent en exhaustivité et en rigueur. Par ailleurs, l'utilisation d'un composant donné dans une diversité importante de situations augmente la probabilité d'apparition des bogues et facilite leur éradication car la multiplication des contextes d'utilisation met en lumière rapidement les faiblesses et les défauts du code.

Il n'est évidemment pas matériellement possible d'écrire toutes les versions du code nécessaires pour couvrir tous les besoins à venir, il faut donc concevoir des mécanismes syntaxiques et conceptuels laissant un degré de liberté au réutilisateur. L'orthogonalité (indépendance) entre composants doit être matérialisée dans le code et le couplage entraîné par la communication entre eux doit être minimal. Un algorithme et une structure de données peuvent être développés indépendamment l'un de l'autre et il est possible de ne pas les coupler de manière définitive.

D. Musser définit la programmation générique par « *requirements-oriented* » [42], c'est-à-dire que l'assemblage des composants repose sur les exigences qu'ils s'imposent entre eux.

Un calcul impose des contraintes algorithmiques aux données qu'il manipule, par exemple une recherche dichotomique dans une séquence implique qu'une relation d'ordre soit définie sur les éléments contenus et qu'un accès direct aux données soit possible. Ces contraintes correspondent à un concept donné et l'algorithme peut s'appliquer à tous les types qui s'y conforment. En matérialisant ces contraintes sur l'interface des objets uniquement et en les minimisant, on atteint le niveau maximal d'indépendance entre composants. Ils ne sont plus dépendants de types concrets mais de concepts qui regroupent tous les types partageant la même interface et le même comportement ce qui les rend polymorphes et interchangeables. Au sein d'un même concept, les interfaces sont homogènes mais les implémentations sont hétérogènes.
L'originalité de la programmation générique par rapport à la programmation purement objets, vient de ce qu'elle impose un troisième type de contrainte portant sur les temps de calcul des opérations. Un type ne peut prétendre appartenir à un concept que s'il se conforme à une interface, à un comportement et à des complexités d'exécution précises.

La programmation générique consiste à définir des concepts ou abstractions capturant les caractéristiques d'un ensemble de types partageant des propriétés sur lesquelles reposeront les dépendances interconcepts. Il s'agit de caractériser rigoureusement les contextes dans lesquels un composant peut être utilisé.
La figure 1.2 tirée de [80] montre les trois grandes familles de composants : types, containers[2] et algorithmes. Avec du code non-générique, pour couvrir tous les besoins potentiels il serait nécessaire d'écrire jusqu'à $i \times j \times k$ versions d'algorithmes[3] , un tri sur des tableaux d'entiers, le même tri sur des tableaux de flottants puis sur des listes, etc. En utilisant un mécanisme de paramétrisation des types appelé patron[4], il est possible d'abandonner l'axe i réduisant ainsi la combinatoire à au plus $j \times k$ possibilités. En rendant les algorithmes indépendants des containers sur lesquels ils s'appliquent, seuls $j+k$ composants restent nécessaires (j containers et k algorithmes).

1.2 Mise en œuvre

Les principes de programmation générique peuvent être mis en œuvre dans n'importe quel langage, ils ne font aucune supposition sur la nature de l'implémentation. Des langages ont été spécialement conçus pour supporter pleinement le paradigme, de la conception jusqu'à la production du programme exécutable : SUCHTHAT [56] composé de deux sous-langages indépendants, ALDES pour la spécification d'algorithme et TECTON [28] pour la spécification des containers. Les premières expériences avec des langages non-spécifiques ont été réalisées en Ada [43] et en Scheme [29] puis en C++ pour des raisons d'efficacité. Ce dernier a été conçu pour supporter des mécanismes de généricité les plus efficaces possibles. Nous allons nous intéresser aux principes de mise en œuvre pour les langages objets en général et en C++ en particulier.

[2]Un container est un objet qui en contient d'autres, la définition précise est donnée à la section 1.2.2.4.
[3]Évidemment, certaines combinaisons n'ont pas de sens pour des raisons d'efficacité, de faisabilité ou d'utilité. Il s'agit d'une borne supérieure très large.
[4]« patron » au sens de la coutûre

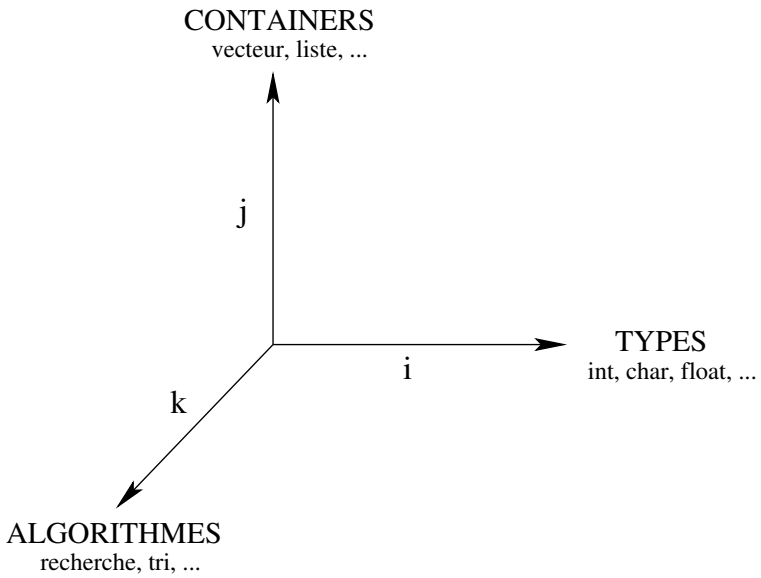

FIG. 1.2 – Décomposition orthogonale de l'espace des composants [80]

1.2.1 Principes

Les principes de mise en œuvre reposent principalement sur deux idées :

1. Programmation par abstractions. On s'intéresse aux interfaces des objets et non à leur type réel. C'est sur les fonctionnalités abstraites que repose l'algorithme, pas sur des représentations concrètes. Peu importe l'implémentation d'un objet pourvu qu'il se conforme au concept approprié à l'algorithme. Plus on minimise ces contraintes, plus on élargit le champ d'application ;

2. Architecture à trois couches (figure 1.3). Les algorithmes ne communiquent pas directement avec les containers sur lesquels ils s'appliquent. Entre les deux, la couche centrale cache la vraie nature de la structure de données. Non seulement l'algorithme n'a pas besoin de connaître le type des objets contenus mais il n'a pas non plus besoin de connaître le type du container. L'orthogonalité algorithme/container est réalisée grâce à des accesseurs dont l'interface est bien définie et qui permettent l'accès aux données sous-jacentes. Ces accesseurs sont généralement appelés itérateurs, curseurs ou énumérateurs.

1.2.2 Généricité en C++

1.2.2.1 Patrons (templates) de classe et de fonction

Les patrons [68] de classe sont des métatypes paramétrés par un ou plusieurs types. L'utilisateur de ces métaclasses, en fournissant explicitement ces types au moment de l'utilisation, instancie le patron et en fait un type concret utilisable. Ce mécanisme permet d'écrire des containers indépendamment du type des objets stockés et de ne prendre une décision concernant le rôle de l'objet qu'au dernier moment. Dans l'exemple suivant, deux patrons standards

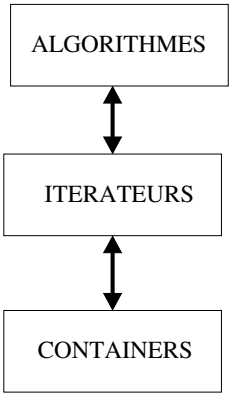

FIG. 1.3 – Architecture générique à trois couches

`list` sont instanciés, le premier objet est configuré pour contenir des entiers, le deuxième pour contenir des listes de flottants :

```
list<int> a;
list<list<float> > b;
```

Les patrons de fonctions acceptent n'importe quel type d'argument pourvu qu'il implémente les opérations nécessaires à l'exécution de la fonction. L'instanciation d'un patron de fonction se fait implicitement, l'utilisateur n'est pas obligé de préciser les types voulus, le compilateur les déduit automatiquement des arguments d'appel. Ces fonctions permettent d'écrire des algorithmes indépendants des types auxquels ils s'appliquent. Par exemple, le patron de fonction `min` suivant renvoie la valeur du plus petit des deux arguments d'appel. Il impose au type `T`, déterminé à l'utilisation, de définir une relation d'ordre matérialisée par l'opérateur `<` :

```
template <class T>
T min(const T &x, const T &y) {
  return x < y ? x : y;
}
```

1.2.2.2 Les itérateurs

Itérateur est le nom officiel de l'accesseur en C++. Tous les containers fournissent un type d'itérateur permettant le parcours des données contenues. Ils permettent d'écrire des algorithmes indépendamment des types des containers et des objets stockés. L'algorithme `copy` copie une séquence source définie par une paire d'itérateurs `[first, last)` appelée intervalle vers une séquence cible accessible à travers l'itérateur `out` :

```
template <class InputIterator, class OutputIterator>
void copy(InputIterator first, InputIterator last, OutputIterator out)
{
  for(; first != last; ++first)
    *out++ = *first;
```

```
}
```

La notation `[first, last)` désigne l'ensemble des positions partant de `first` mais n'incluant pas `last` qui ne sert que de marqueur de fin de séquence[5].
L'algorithme ne fait aucune supposition sur le type des éléments de la séquence ni sur la structure de données. Les seules conditions requises portent sur le type `InputIterator` qui doit implémenter trois opérateurs `++`, `*`, `!=` et sur le type des objets copiés qui doit être assignable (opérateur d'affectation `=`).

1.2.2.3 Expansion en ligne (inlining)

La programmation générique en C++ est viable car la compilation du code génère des exécutables efficaces et ce grâce, notamment, à l'expansion ou insertion en ligne (inlining) qui consiste à remplacer dans le fichier binaire les appels à certaines fonctions par le corps de ces fonctions. Le résultat est similaire à celui obtenu en utilisant des macros en C mais sans en subir les inconvénients puisque ces pseudo-macros sont de vraies fonctions et qu'elles bénéficient donc de tous les garde-fou habituels de la compilation, entre autres les vérifications de type.
Toutes les fonctions ou méthodes ne sont pas expansées en ligne, seules celles que le compilateur juge suffisamment simples le sont ; il n'est par exemple évidemment pas possible d'insérer en ligne une fonction récursive.
Chose peu courante, cette optimisation est non seulement terriblement efficace (il est courant d'observer des facteurs d'accélération de 10 ou 20 entre une compilation avec inlining et sans), mais elle permet une plus grande encapsulation du code et donc une conception plus robuste : elle réduit à néant le coût des appels aux méthodes aussi simples que l'accès à l'attribut d'un objet ou la transmission d'un appel (*forwarding*). Il est donc possible grâce à cette optimisation de se passer d'accéder directement aux données contenues dans un objet ce qui est considéré comme une discipline saine en programmation objet.

1.2.2.4 La Standard Template Library

La STL est une librairie générique de containers et d'algorithmes sur les séquences intégrée au langage C++. Son intérêt repose plus sur les concepts et les spécifications qui la constituent que sur le code en lui-même. On dit d'ailleurs souvent *« STL is not a set of programs, it is a set of requirements »* et on considère souvent comme équivalentes les notions de concept et de spécification qui n'est autre que la matérialisation d'un concept et qui constitue principalement STL. Elle définie une hiérarchie d'abstractions par raffinements successifs sur les propriétés que doivent vérifier les types concrets (les modèles) se conformant aux concepts.

L'espace des composants est découpé en cinq grands concepts : les algorithmes, les itérateurs, les containers, les allocateurs [32] et les adaptateurs qui projettent un concept vers un autre. Ces familles de types abstraits sont constituées de sous-ensembles regroupant les types partageant des caractéristiques précises. Par exemple, le concept *assignable* désigne tous les types concrets dont on peut assigner la valeur à un objet du même type (il définit un opérateur `=`). Ces ensembles incluent également des sous-ensembles obtenus par raffinement du concept. La figure 1.4 montre un extrait de la hiérarchie STL du concept de container séquentiel. Un

[5] C'est un itérateur « *past-the-end* »

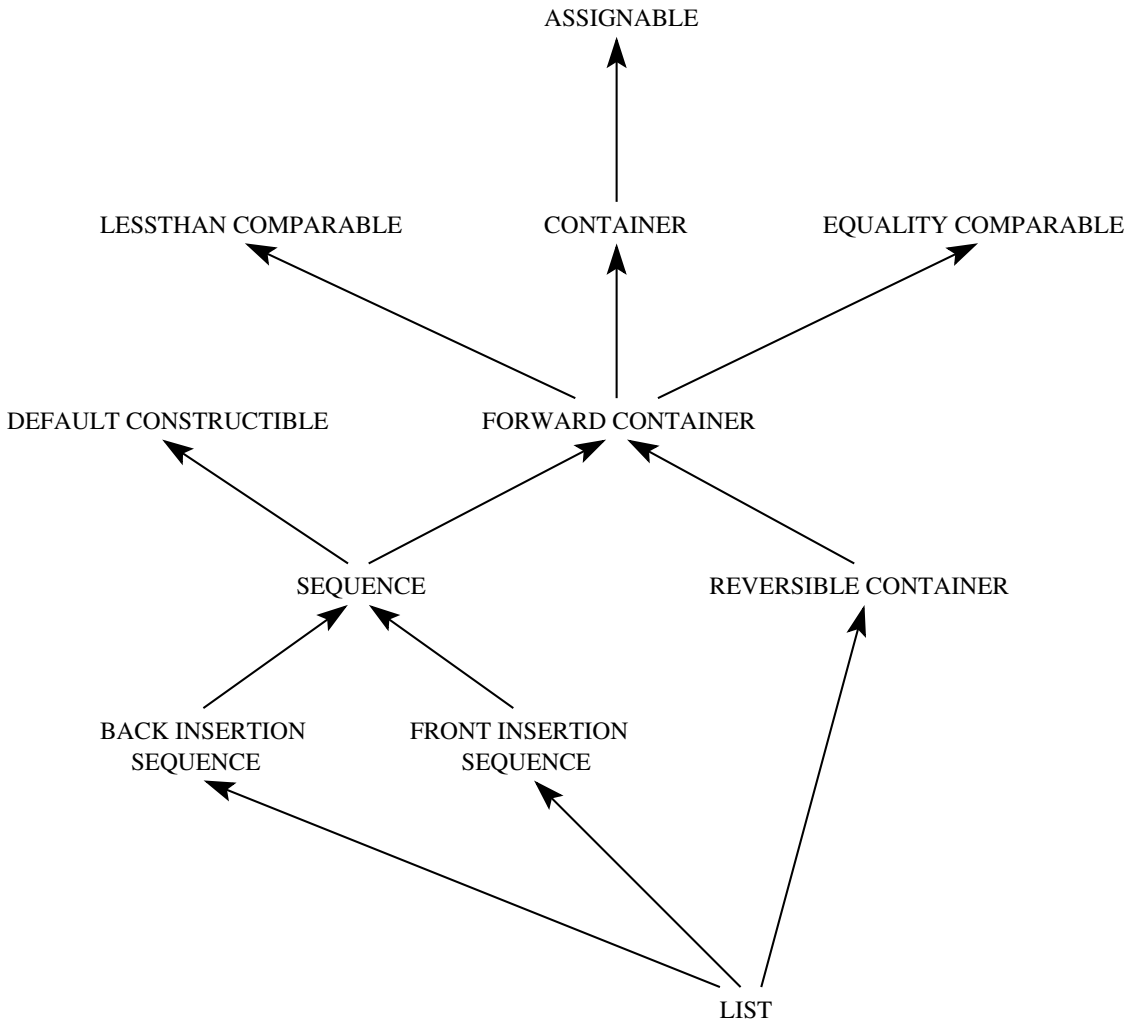

FIG. 1.4 – Extrait de la hiérarchie des containers séquentiels dans STL

container est un objet capable de stocker d'autres objets et fournit des méthodes d'accès auxdits objets ainsi qu'un type d'itérateur. Un *forward container* est un container dont les éléments sont rangés dans un ordre défini ne changeant pas spontanément d'un parcours à l'autre. Il ne permet l'énumération de ses objets que dans un sens alors que le *reversible container* autorise l'itération dans les deux sens. Une *sequence* est un container supportant l'insertion et la suppression d'élément. Enfin, une *front insertion sequence* autorise l'insertion d'éléments en tête et en temps constant alors qu'une *back insertion sequence* ne le permet qu'en queue. En bas de la hiérarchie, le type concret `list` modélise tous ces concepts, c'est-à-dire qu'il vérifie les propriétés de tous les concepts dont il dérive. Il est possible de traverser la liste dans les deux sens car il s'agit d'une liste doublement chaînée et il est possible d'y insérer des éléments partout en temps constant.
STL définit de telles hiérarchies sur les itérateurs, les containers séquentiels et associatifs (arbres, tables de hachage, etc.) et les adaptateurs. Les algorithmes sont codés avec des patrons de fonction prenant en argument des intervalles, c'est-à-dire des paires d'itérateurs marquant le début et la fin de la séquence sur laquelle s'applique le traitement. Ce découpage orthogonal des composants est conçu pour ne pas être figé et se destine à être étendu à tous les domaines de programmation, ce que nous nous proposons de faire pour les automates.

1.3 Automates génériques

Les motivations à développer une librairie générique d'automates reposent sur les constatations suivantes :

- L'automate est un modèle mathématique et informatique « versatile » (au sens anglo-saxon du terme), il trouve son utilité dans un nombre impressionnant de domaines variés où les besoins et contraintes sont multiples, d'où la nécessité d'un code adaptatif.

- La demande pour des applications efficaces reste plus que jamais d'actualité avec l'émergeance des technologies de traitement de la langue naturelle et l'augmentation vertigineuse des quantités de données à traiter. La programmation générique en C++ a l'avantage de fournir des implémentations dont les performances sont comparables à celles du C ou du Fortran [77] et ceci aussi bien en vitesse qu'en mémoire.

- Il existe un lien évident entre tous les modèles mathématiques basés sur des structures de graphe. La base étant la même, il est possible de concevoir des concepts communs à tous ces modèles ainsi que des structures de données qui leur correspondent.

- Il n'existe aucune librairie réellement générique, c'est-à-dire combinant adaptabilité et efficacité. Les recettes qui ont contribué au succès de la programmation générique sur les séquences n'ont pas été appliquées rigoureusement et exhaustivement aux graphes, aux automates et aux machines en général.

1.4 Domaine

Cette section décrit exhaustivement l'ensemble des librairies officiellement disponibles pour l'utilisation et le traitement des graphes et des automates. Elle expose leurs particularités et les faiblesses motivant la conception d'une nouvelle librairie réellement générique :

- La « Library of Efficient Data Types and Algorithms » (LEDA, [36]) est une des plus anciennes librairies de structures de données génériques écrites en C++. Elle couvre un domaine d'application très large et notamment les graphes mais ne supporte pas de mécanisme de généricité assez développé. Antérieure à STL, sa conception et son architecture n'incorpore pas les récentes avancées de la programmation générique en C++.

- L'ex « Generic Graph Components Library » (GGCL, [58]) devenue la « Boost Graph Library » (BGL, [59]). Il s'agit probablement de la librairie la plus sophistiquée et la plus efficace. Écrite en C++, ses principes de conception comprennent notamment l'utilisation du patron de conception visiteur pour les parcours [19] et la possibilité d'associer des paramètres multiples aux nœuds et aux arcs. Elle ne comprend malheureusement que deux représentations différentes de graphes (liste d'adjacence ou container d'arcs) et ne propose pas réellement de nouveaux concepts ni de méthodes neuves de programmation. Il manque en particulier un concept unique d'itérateur généralisé aux graphes.

- La « Graph Template Library » (GTL, [16]) est à mi-chemin entre LEDA et BGL. Le niveau de généricité est plus élevé que pour la première ; il y est fait usage de concepts comme les itérateurs (sur les arcs et les nœuds) permettant une interaction directe avec les algorithmes de STL et donc un niveau de réutilisation raisonnable. Malheureusement la modularité nécessaire au découplage structure/algorithme est loin d'être satisfaisante car elle repose sur un modèle à deux couches qui implique d'incorporer un certains nombres d'algorithmes au container comme par exemple la sauvegarde et le chargement, or le modèle à deux couches montre très vite ses limites et son manque de souplesse, notamment en ce qui concerne les algorithmes de parcours génériques.

- La « Patterns Template Library » (PTL, [65]) est basée sur les structures de données dites intrusives [64]. Elle ne constitue pas vraiment une librairie complète de machine à états mais plutôt un exercice d'application pour un style de conception particulier cher à son auteur. Elle est, de ce fait, extrêmement limitée.

- « Grail+ » [51] est un environnement de calcul sur les automates sous forme de filtres UNIX accompagnés d'une collection de composants écrits en C++ mais non-génériques. Les patrons de classe ne sont paramétrés que par le type d'objet étiquetant les transitions, le concept d'accesseur est inexistant et les algorithmes sont incorporés aux classes.

- « Fire Lite » [78] est écrite en C++ et implémente les algorithmes décrits dans la taxonomie des algorithmes sur les automates et les expressions rationnelles [79]. Probablement la plus complète en ce qui concerne les algorithmes de construction. La librairie est conçue avec soin et l'aspect performance est mis en avant mais la généricité est insuffisante : l'orthogonalité entre structures et algorithmes n'est pas assurée.

– « AUTOMATA » (`http://www-2.cs.cmu.edu/~sutner/automata.html`) est une librairie écrite en Mathematica et dont les parties critiques sont codées en C++. L'architecture n'est pas générique et ne se positionne pas dans une optique de réutilisation.

– « FSA Utilities » (Finite State Automata Utilies) se présente sous forme de filtres UNIX et de librairies en Prolog permettant la manipulation d'automates, d'expressions rationnelles et de transducteurs (`http://odur.let.rug.nl/~vannoord/fsa/fsa.html`).

1.5 Au-delà de la programmation générique

De nouvelles techniques de programmation apparues récemment proposent des palliatifs aux principales faiblesses de la programmation générique et augmentent encore le degré d'adaptabilité des composants logiciels. La programmation générative [12, 8], grâce à la métaprogrammation [72, 35], [11] pp. 251-279, permet d'écrire des librairies actives [76] où un grand nombre d'opérations liées à la conception et l'assemblage des composants est automatisé et relégué au compilateur. Ces librairies génèrent des types extrêmement composites construits à partir de spécifications utilisateur haut-niveau reflétant le contexte d'utilisation. Elles intègrent en outre le paradigme de la programmation dite orientée-aspect [11] pp. 183-242, insistant sur la modularité et la maintenabilité.
Cette expérience de programmation générique a logiquement mené à la conception d'une librairie active de machines à états au sens large du terme capable de générer des types de containers et de curseurs optimisés couvrant la totalité des besoins du domaine.

1.6 Plan

L'idée générale de l'exposé est de partir du modèle des automates, de voir comment on peut les implémenter de manière efficace et générique en imaginant les concepts qui leur sont propres puis de généraliser à toutes les structures de données partageant des similarités. Les différentes étapes sont :

1. Définitions mathématiques formelles, vocabulaire et propriétés des automates (chapitre 2) ;
2. Concepts liés à l'abstraction « machine à états », structures de données et implémentation des containers (chapitre 3). Premier niveau d'adaptabilité de la librairie, une batterie de containers polymorphes aux performances hétérogènes ;
3. L'abstraction accesseur d'automate, le curseur (chapitre 4) ;
4. Deuxième niveau d'adaptabilité, les adaptateurs de curseur (chapitre 5) ;
5. Généralisation de ces concepts à toutes les structures de données basées sur des structures de graphes. Palliatif aux faiblesses de la programmation générique (configuration et assemblage de composant automatiques), programmation générative en C++ grâce à la métaprogrammation statique (chapitre 6).

Le modèle développé dans la suite de cet exposé :

– est réellement adaptatif : les containers d'automate sont paramétrés par le type de l'alphabet et le type de l'information associée aux états. De plus, une variété de containers partageant la même interface mais hétérogènes du point de vue des représentations en mémoire permet d'adapter les complexités des traitements en temps et en espace à une multitude de situations.

– met en œuvre une architecture à trois couches avec les curseurs comme accesseurs. Ces curseurs offrent un degré de liberté supplémentaire car leur comportement est modifiable à souhait par adaptation du comportement de base. Une bonne partie des opérations présentes habituellement dans les algorithmes est reléguée à la couche curseur. Le manque de généricité des librairies existantes vient du fait qu'elles ne proposent pas réellement de modèle à trois couches bien identifiées car elle n'introduisent pas de concept unique d'accesseur sur automate.

– est conçu pour être efficace et utilisable dans un contexte « industriel » de développement ; bon nombre d'optimisations ont été envisagées et l'implémentation repose sur des composants STL solides et éprouvés.

Chapitre 2

Les automates

Les automates et les machines à états finies sont nés quasiment en même temps que l'informatique et ont été intensivement étudiés et utilisés durant les 50 dernières années. D'abord modèles d'étude théoriques riches et puissants, leur succès s'explique aussi par leur efficacité pratique et le nombre impressionnant d'applications pour lesquelles ils se révèlent particulièrement bien adaptés. Les machines à états trouvent leur utilité dans la conception de composants électroniques, la recherche de motifs, le cryptage, la compression de donnée, la linguistique avec les analyses morphologique, syntaxique et sémantique, la correction orthographique, la reconnaissance de la parole, etc. Des graphes aux transducteurs en passant par les automates, les applications des machines à états finies sont beaucoup trop nombreuses pour en dresser une liste exhaustive.

Ce chapitre aborde les définitions formelles des automates. Il constitue la première étape de l'exposé qui nous mènera progressivement du modèle mathématique au modèle informatique abstrait puis à son implémentation concrète. Il s'agit donc ici de définir le plus formellement possible les abstractions mathématiques sur lesquelles reposeront les définitions d'abstractions logicielles.

2.1 Mots et langages

Un automate sert notamment à modéliser et à construire une grammaire. Cette grammaire définit un langage, autrement dit un ensemble de mots autorisés que l'automate est capable de reconnaître.

2.1.1 Définitions

Soit Σ un ensemble appelé *alphabet* dont les éléments sont appelés *lettres*. Un mot w sur l'alphabet Σ est une suite finie de lettres $(\sigma_1, \sigma_2, ..., \sigma_n)$ avec pour longueur $n \geq 0$ notée $|w|$. On désigne la i^{ème} lettre du mot par w_i. Si $n = 0$, w est appelé *mot vide* et noté ϵ. L'ensemble des mots sur l'alphabet Σ est noté Σ^* et $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$. Le *produit* ou *concaténation* de deux mots $u, v \in \Sigma^*$ s'obtient en écrivant séquentiellement les lettres de u puis celles de v :

$$u \cdot v = u_1\, u_2 \,...\, u_n\, v_1\, v_2 \,...\, v_m$$

Par simplification et partout où cela ne prête pas à confusion on omettra le $\cdot$ en écrivant uv plutôt que $u \cdot v$.

Le produit d'un mot u avec lui-même répété k fois s'écrit u^k avec $u^0 = \epsilon$. Si $w = uw'v$ on dit que w' est un *facteur* de w, u est un *préfixe* de w et v est un *suffixe* de w.

Un *langage* sur l'alphabet Σ est un sous-ensemble de Σ^*.

2.1.2 Opérations sur les langages

Soient A et B deux langages. On généralise le produit de mots aux langages comme suit :

$$A \cdot B = \{ab \mid a \in A, b \in B\}$$

On définit les opérations booléennes sur les langages par :

$$A \cap B = \{w \mid w \in A \text{ et } w \in B\}$$

$$A \cup B = \{w \mid w \in A \text{ ou } w \in B\}$$

$$A \setminus B = \{w \mid w \in A \text{ et } w \notin B\}$$

$$A \triangle B = (A \cup B) \setminus (A \cap B) = \{w \mid w \in A \cup B \text{ et } w \notin A \cap B\}$$

$$\overline{A} = \Sigma^* \setminus A = \{w \mid w \in \Sigma^* \text{ et } w \notin A\}$$

2.2 Automates finis

2.2.1 Définition

Pour accroître la généricité et s'adapter aux besoins algorithmiques on ajoute à la définition traditionnelle des automates la possibilité d'associer à chaque état une information quelconque appelée « tag ». L'ensemble τ associe les états de l'automate à leur tag de manière bijective.

Soit $A(\Sigma, Q, I, F, \Delta, T, \tau)$ un 7-uplet d'ensembles finis définit comme suit :

Σ	Un alphabet
Q	Un ensemble d'états
$I \subseteq Q$	Un ensemble d'états initiaux
$F \subseteq Q$	Un ensemble d'états finaux ou terminaux
$\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$	Un ensemble de transitions
T	Un ensemble de tags
$\tau \subset (Q \times T)$	Un ensemble associant les états à leur tags respectifs

L'*étiquette* d'une transition (q, σ, p) est la lettre σ, q est sa *source* et p sa *destination* ou *but*. Lorsque $\sigma = \epsilon$ (le mot vide) la transition est appelée *epsilon-transition*.
On appelle transitions *entrantes* de l'état s, l'ensemble des transitions $(q, \sigma, p) \in \Delta$ telles que $p = s$ et inversement, on appelle transitions *sortantes* l'ensemble des transitions $(q, \sigma, p) \in \Delta$ telles que $q = s$.

On notera $P(X)$ l'ensemble des parties d'un ensemble X et $|X|$ son nombre d'éléments.

On définit deux fonctions de transition δ_1 et δ_2 fréquemment utilisées pour l'accès aux transitions de l'automate :

$$\delta_1 \ : \ Q \times (\Sigma \cup \{\epsilon\}) \rightarrow P(Q)$$
$$\delta_2 \ : \ Q \rightarrow P(\Sigma \times P(Q))$$

La fonction δ_1 associe à un état q et une lettre σ l'ensemble des buts des transitions étiquetées par σ sortant de q. La fonction δ_2 associe à un état q l'ensemble des couples lettre/but de ses transitions sortantes. On étend naturellement la définition de δ_1 aux mots :

$$\begin{array}{rcl} \delta_1^* : P(Q) \times \Sigma^* & \rightarrow & P(Q) \\ (q, \epsilon) & \mapsto & q \\ (q, w \cdot a) & \mapsto & \delta_1(\delta_1^*(q, w), a) \end{array}$$

Le *contexte droit* d'un état q est l'ensemble des lettres étiquetant les transitions sortant de q : $\vec{c}(q) = \{\sigma \in \Sigma \mid \exists p \in Q,\ (q, \sigma, p) \in \Delta\}$.
On appelle *chemin* une suite $c = t_1\ t_2\ ...\ t_n$ de transitions $t_i = (q_i, \sigma_i, p_i)$ telle que $\forall i, t_i \in \Delta$ et pour $0 < i < n$, $q_{i+1} = p_i$. La longueur du chemin n est notée $|c|$ et son étiquette est la concaténation des lettres étiquetant la suite des transitions : $w = \sigma_1\ \sigma_2\ ...\ \sigma_n$.

Le langage reconnu par un automate A est défini par :

$$L(A) = \{w \in \Sigma^* \mid \delta_1^*(I, w) \cap F \neq \emptyset\}$$

soit l'ensemble des étiquettes des chemins menant d'un état initial à au moins un état final.

2.2.2 Exemple

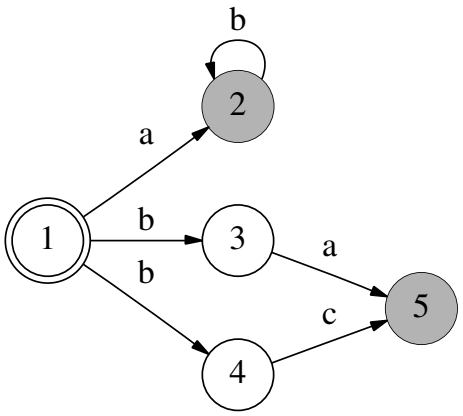

FIG. 2.1 – Un automate A non déterministe

Les conventions graphiques que nous utiliserons sont les suivantes (figure 2.1) : les transitions sont représentées par des flèches reliant deux états et étiquetées par les lettres de l'alphabet ou par ϵ. Les états initiaux sont représentés par des doubles cercles et les états

finaux sont en gris. Sauf absolue nécessité, on omettra la représentation des tags pour des besoins de clarté.

L'automate A a pour alphabet $\{a, b, c\}$, pour ensemble d'états $\{1, 2, 3, 4, 5\}$ avec un état initial 1 et deux états finaux 2 et 5. Il a pour ensemble de transition

$$\Delta = \{(1, a, 2), (2, b, 2), (1, b, 3), (3, a, 5), (1, b, 4), (4, c, 5)\}$$

et il reconnaît les mots ba, bc ainsi que les mots commençant par la lettre a suivie d'un nombre quelconque de b éventuellement nul.

2.2.3 Propriétés

Nous utiliserons l'automate A (figure 2.1) de la section 2.2.2 pour illustrer les définitions.

2.2.3.1 Asynchrone

Un automate est dit *asynchrone* lorsqu'il possède des ϵ-transitions. Le terme d'asynchrone vient du fait qu'il est possible d'avancer sur l'automate en suivant des ϵ-transitions sans pour autant avancer sur le texte. Pour un automate synchrone, avancer sur une transition signifie obligatoirement qu'il faut lire une lettre dans le texte. L'automate A est synchrone.

Remarque : L'asynchronicité entraîne le non déterminisme mais il est toujours possible de supprimer les ϵ-transitions.

2.2.3.2 Déterministe

Un automate est *déterministe* si et seulement si :

1. Il possède un unique état initial i.
2. $(q, \sigma, p), (q, \sigma, p') \in \Delta \Rightarrow p = p'$.
3. Il ne contient pas d'ϵ-transition.

Autrement dit, pour tout état q il existe au plus une transition sortante étiquetée par la lettre σ : $|\delta_1(q, \sigma)| \leq 1$. De plus, on définit l'*état nul*, noté 0, comme résultat de la fonction de transition lors d'un échec et dont la définition déterministe est :

$$\delta_1 \ : \ Q \times \Sigma \rightarrow Q \cup \{0\}$$

$$\delta_1(q, \sigma) = \begin{cases} p & \text{si } (q, \sigma, p) \in \Delta \\ 0 & \text{sinon} \end{cases}$$

Remarque : Pour tout automate fini A il existe un automate fini B déterministe tel que $L(A) = L(B)$. La preuve ce théorème fait intervenir l'automate des parties de Q [6]. L'automate de la figure 2.2 est l'équivalent déterministe de A (les états 3 et 4 ne forment plus qu'un seul état $\{3, 4\}$).

2.2.3.3 Complet

Un automate est *complet* lorsque tout état possède au moins une transition étiquetée par chacune des lettres de l'alphabet :

$$\forall q \in Q, \forall \sigma \in \Sigma, \delta_1(q, \sigma) \neq \emptyset$$

Remarque : On construit facilement pour tout automate A l'automate complet A' reconnaissant le même langage en ajoutant un état « puits »non final vers lequel sont dirigées les transitions sortantes étiquetées par les lettres manquantes (voir figure 2.3 pour le complété de A).

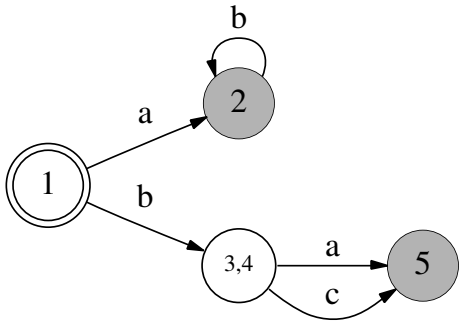

FIG. 2.2 – L'automate A déterminisé et minimisé

2.2.3.4 Acyclique

Un automate est *acyclique* lorsque le graphe orienté sous-jacent ne possède pas de cycle. Pour une définition formelle des cycles sur les graphes voir [6]. A est rendu cyclique par la transition $(2, b, 2)$.
Remarque : les automates acycliques reconnaissent des langages finis.

2.2.3.5 Minimal

Dans l'ensemble des automates finis reconnaissant un langage L, il en existe un plus petit que tous les autres en terme d'états (à un isomorphisme près). De ce théorème, dont la preuve est basée sur l'équivalence de Nérode, découle l'algorithme classique de minimisation d'Hopcroft (basé sur la construction de Moore, cf [6] pour la preuve détaillée et la construction). Pour la minimisation efficace des automates acycliques voir [54]. Pour une description exhaustive des algorithmes de minimisation on se référera à [79].
L'automate de la figure 2.2 est minimal (les états 3 et 4 ont été fusionnés, tous les états sont distincts au sens de Nérode).

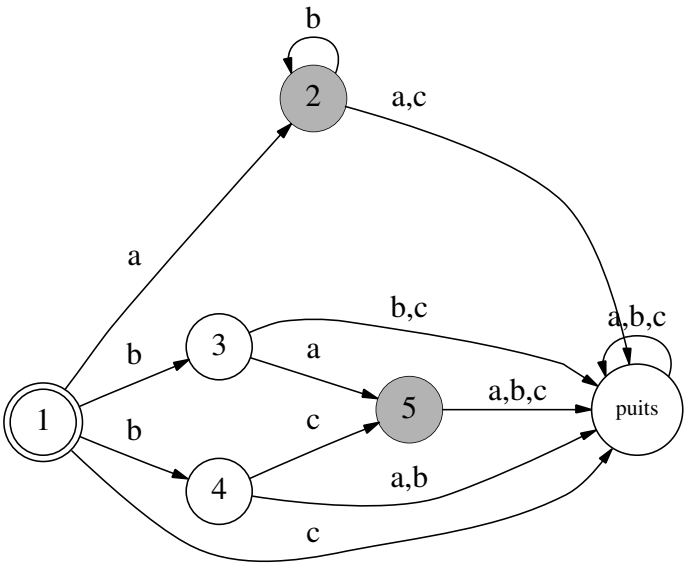

FIG. 2.3 – Le complété de A

2.2.4 Langages et expressions rationnels

2.2.4.1 Définitions

- On appelle opérations rationnelles l'union, le produit et l'étoile.
- Un langage L sur Σ^* est dit *rationnel* lorsqu'il peut-être obtenu de façon finie à partir de parties finies de Σ^* grâce aux opérations rationnelles d'union, de produit et d'étoile.
- La description d'un langage comme une combinaison finie d'opérations rationnelles et de parties finies de Σ^* à un élément est une *expression rationnelle*.

2.2.4.2 Théorème de Kleene

L est rationnel $\Leftrightarrow$ il existe un automate A reconnaissant L

Une preuve de ce théorème fait intervenir pour la partie implication $\Rightarrow$ les propriétés de fermeture pour les opérations rationnelles des langages reconnus par les automates et pour la partie symétrique $\Leftarrow$ l'algorithme de MacNaughton et Yamada permettant de construire une expression rationnelle à partir d'un automate.

Ce théorème est fondamental car il lie directement la définition syntaxique formelle d'un langage (sa spécification rationnelle) à l'automate le reconnaissant par un algorithme.

2.2.5 Automates déterministes étendus et transitions par défaut

Les automates suivants étendent leur définition de manière à augmenter le pouvoir expressif des langages reconnus : en définissant des fonctions de transition déterministes δ_1 plus élaborées ils permettent l'étiquetage des transitions par des opérations ensemblistes sur Σ. Il en découle le plus souvent, en restant dans le cadre des langages rationnels, des algorithmes plus efficaces et plus clairs notamment pour l'algorithme de recherche de motif d'Aho et Corasick [1] et l'implémentation des automates étendus [53].

2.2.5.1 Automate avec état par défaut

On construit un automate $A(\Sigma, Q, i, F, \Delta, s)$ avec un état par défaut en ajoutant à la définition déterministe de l'automate un état $s \in Q$ et en définissant δ_1 par :

$$\delta_1(q,\sigma) = \begin{cases} p & \text{si } (q,\sigma,p) \in \Delta \\ s & \text{sinon} \end{cases}$$

Remarque : Cette définition de δ_1 revient à adjoindre à chaque état q de l'automate une transition dirigée vers s étiquetée par $\Sigma \setminus \vec{c}(q)$, c'est-à-dire l'ensemble des lettres qui n'apparaissent pas sur les transitions sortant de q. Cette fonction de transition rend A complet de façon économique avec s comme état puits et rend donc l'automate cyclique s'il ne l'était déjà.

2.2.5.2 Automate avec transitions par défaut

Un automate déterministe avec transitions par défaut $A(\Sigma, Q, i, F, \Delta, \overline{\Delta})$ où $\overline{\Delta} \subseteq Q \times Q$ associe à tout ou partie des états de Q un état p résultat de la fonction de transition lors d'un échec :

$$\delta_1(q,\sigma) = \begin{cases} p & \text{si } (q,\sigma,p) \in \Delta \text{ sinon} \\ s & \text{si } (q,s) \in \overline{\Delta} \text{ sinon} \\ 0 & \end{cases}$$

De même qu'un automate muni d'un état par défaut, un automate avec transitions par défaut permet d'étiqueter une transition sortante de tout état $q \in Q$ par $\Sigma \setminus \vec{c}(q)$. Mais à la différence d'un état unique par défaut pour tout l'automate, ces transitions permettent de définir un comportement local à un état en cas d'échec.

2.2.5.3 Automate avec substitut

Pour un automate déterministe avec substitut $A(\Sigma, Q, i, F, \Delta, \overline{\Delta})$ où $\overline{\Delta} \subseteq Q \times Q$ définit pour tout ou partie des états de Q un substitut s, on a :

$$\delta_1(q,\sigma) = \begin{cases} p & \text{si } (q,\sigma,p) \in \Delta \text{ sinon} \\ \delta_1(s,\sigma) & \text{si } (q,s) \in \overline{\Delta} \text{ sinon} \\ 0 & \end{cases}$$

Il est à noter qu'une condition est nécessaire pour s'assurer que l'algorithme de δ_1 s'arrête : le sous-graphe orienté $(Q, \overline{\Delta})$ ne doit pas contenir de cycle, ce qui implique qu'au moins un état n'a pas de substitut. Dans l'algorithme d'Aho et Corasick [1], la fonction de transition

s'arrête lorsque les échecs successifs ramènent à l'état initial qui ne possède pas de substitut, ce qui correspond à la réinitialisation complète de la machine.

2.2.6 Opérations sur les automates finis

Les opérations rationnelles (concaténation, union et fermeture de Kleene) ainsi que les opérations ensemblistes (intersection, différence, différence symétrique et complément) définies sur les langages possèdent leurs homologues sur les automates. Elles engendrent des machines reconnaissant le langage correspondant.
Exemple : soient A et A' deux automates déterministes. L'automate déterministe I dont l'ensemble des états est le produit cartésien de Q et Q' - les deux ensembles d'états de A et A' - et dont la fonction de transition δ_1^I est définie par :

$$\delta_1^I((q,q'),\sigma) = (\delta_1^A(q,\sigma), \delta_1^{A'}(q',\sigma))$$

reconnaît le langage obtenu par intersection de $L(A)$ et $L(A')$ si l'ensemble des états terminaux de I est constitué des états (q,q') vérifiant $q \in F$ et $q' \in F'$. Son état initial est (i,i'). Lorsque les états terminaux vérifient la propriété $q \in F$ ou $q' \in F'$, l'automate reconnaît l'union des langages $L(A)$ et $L(A')$ et son état initial (q,q') vérifie $q = i$ ou $q' = i'$.
Le modèle d'automate est donc particulièrement intéressant lorsque ces opérations sur les langages sont nécessaires car leurs définitions sur les machines à états sont constructives et fournissent des algorithmes directement appliquables.

2.3 Automates et parcours

On appelle parcours un algorithme de visite des états ou des transitions d'un automate.

2.3.1 Parcours simple

Un parcours simple s'effectue le long d'une suite finie d'états $(q_1, q_2, ..., q_n)$ vérifiant pour $0 < i < n$, $\exists\sigma$ tel que $(q_i, \sigma, q_{i+1}) \in \Delta$. L'utilisation de δ_1 est suffisante pour pouvoir implémenter un parcours simple, par exemple en se positionnant sur l'état initial i et en avançant par applications successives de la fonction de transition. L'algorithme 2.1 teste l'appartenance d'un mot w au langage reconnu par un automate A en le parcourant simplement.
En extrayant la liste des états sources d'un chemin on obtient un parcours simple. Les parcours suivants sont plus sophistiqués. Ils nécessitent d'itérer sur les ensembles de transitions sortant des états.

2.3.2 Parcours en profondeur

L'algorithme 2.2 est une version récursive générique de l'algorithme classique de parcours en profondeur. Il est paramétré par trois actions aux lignes 1, 7 et 9. Il s'agit de la partie commune aux algorithmes effectuant un parcours qui adaptent son comportement en définissant les trois actions.

Algorithme 2.1 appartient

Entrée : $w \in \Sigma^*$, $A(\Sigma, Q, i, F, \Delta)$ déterministe
Sortie : booléen

$q \leftarrow i$
$k \leftarrow 0$
tant que $k < |w|$ **et** $\delta_1(q, w_{k+1}) \neq 0$ **faire**
    $q \leftarrow \delta_1(q, w_{k+1})$
    $k \leftarrow k + 1$
fin tant que
si $k = |w|$ **et** $q \in F$ **alors**
    retourner vrai
sinon
    retourner faux
fin si

Algorithme 2.2 parcours en profondeur

Entrée : $A(\Sigma, Q, i, F, \Delta)$, $q \in Q$
1: action 1
2: $q \leftarrow$ visité
3: **pour tout** $(a, p) \in \delta_2(q)$ **faire**
4: **si** p non visité **alors**
5: parcours en profondeur(A, p)
6: **fin si**
7: action 2
8: **fin pour**
9: action 3

2.3.3 Parcours en largeur

Ce type de parcours requiert l'utilisation d'une file d'états notée F sur laquelle on définit les deux opérations :

Insertion en fin de file	$F \leftarrow q$ avec $q \in Q$
Extraction de la tête de file	$F \rightarrow q$

Algorithme 2.3 parcours en largeur

Entrée : $A(\Sigma, Q, i, F, \Delta)$
  $F \leftarrow i$
  tant que F n'est pas vide **faire**
    $F \rightarrow q$
    pour tout $(a, p) \in \delta_2(q)$ **faire**
      si p non visité **alors**
        $p \leftarrow$ visité
        $F \leftarrow p$
        action
      fin si
    fin pour
  fin tant que

Chapitre 3

Les containers ASTL

L'Automaton Standard Template Library est un ensemble de composants logiciels génériques codés en C++. Chacun de ces composants entre dans une catégorie bien définie dont l'interface est complètement spécifiée, y compris les complexités de temps de calcul.
Une telle conception nécessite de définir des abstractions logicielles en rapport avec le domaine considéré, c'est-à-dire de définir le comportement des objets, ici des containers d'automate et l'étendue de leur champ d'applications (ce qu'ils doivent savoir faire et ce pour quoi ils ne sont pas faits) en gardant à l'esprit que les deux buts fondamentaux restent la réutilisation et l'efficacité. Grâce à une utilisation intensive de patron de classe, ASTL fournit un ensemble de structures de données polymorphes donc interchangeables avec leurs caractéristiques propres. Chacune des classes d'automate présente des complexités spatiales et temporelles adaptées à des contraintes différentes. Cette variété d'implémentations permet l'adaptation d'une application à des environnements divers sans avoir à modifier le reste de l'architecture car l'interface des automates est standardisée.

3.1 Structure générale

ASTL se plie aux standards de la programmation générique en C++ définis par la librairie standard :

- Tous les composants sont génériques, ce sont des patrons (*templates*) de classe ou de fonction.

- L'héritage se limite au strict minimum. Il n'y a pas obligatoirement de relation de sous-typage réelle entre objets de concepts imbriqués. Des spécifications précises et rigoureuses suffisent à maintenir un polymorphisme statique. Les composants sont donc interchangeables mais il n'y a pas de méthodes virtuelles pour maximiser l'optimisation à la compilation et pour éviter les problèmes liés à leur interaction avec les composants STL [31].

- Les containers ASTL sont des containers STL ou des combinaisons. On accède aux données grâce à des itérateurs et les séquences sont définies par des intervalles.

- Les algorithmes ne sont pas incorporés aux containers. L'utilisateur « branche »l'algo-

rithme choisi sur la classe d'automate qui satisfait le mieux ses contraintes d'espace et de temps. La communication s'établit à travers les curseurs, accesseurs d'automates équivalents aux itérateurs de séquences. Les algorithmes génériques ne s'appliquent jamais directement à la structure de données.

Dans la section suivante, on définit le type abstrait ou concept d'automate en termes de concepts STL.

3.2 Le concept d'automate

Pour une définition mathématique formelle des automates, de leurs propriétés et des termes s'y rapportant se référer à la section 2.2.1 page 24.
Les définitions suivantes décrivent un ensemble de concepts dont la combinaison constitue le concept général d'automate.

3.2.1 Transitions sortantes

L'ensemble $\delta_2(q) = ((\sigma_1, p_1),\ ...,\ (\sigma_n, p_n))$ des transitions sortant d'un état q est stocké dans un container associatif de paires appelé `Edges` (*pair associative container*) dont les clés sont les lettres $\sigma \in \Sigma$ et les valeurs les couples $(\sigma, p) \in \Sigma \times Q$. Pour les automates non déterministes, ce container associatif est dit à clés multiples, c'est-à-dire que les clés ne sont pas forcément uniques : on peut avoir pour $i \neq j$, $\sigma_i = \sigma_j$. Autrement dit, m valeurs peuvent être associées à une même clé, ces m transitions étant étiquetées par la même lettre. Dans le cas d'un automate déterministe, le container associatif est dit à clés uniques et les transitions d'un état sont toutes étiquetées par des lettres différentes.

3.2.2 États

Un état q est un couple constitué d'un tag noté `Tag` et de n transitions sortantes : $q = (\tau(q), \delta_2(q))$, soit en notation STL `pair<Tag, Edges>`. À chaque état est associé un identifiant unique (*handler*) dont le type sera noté `State`. Dans le cas des automates non déterministes, ce type est un container d'identifiants d'états, modèle de `set` STL constant (voir la section 3.4 pour plus de détails).

3.2.3 États terminaux

L'ensemble des états terminaux F est une séquence de booléens à accès direct (*random access container*), les éléments de la séquence étant indexés par les identifiants d'états.

3.2.4 États initiaux

L'état initial est un modèle de `State`, i.e. dans le cas déterministe, il s'agit d'un simple identifiant et dans le cas contraire, il s'agit d'un ensemble d'identifiants.

3.2.5 Automate

Un automate est un container séquentiel d'états autorisant l'itération grâce à un itérateur monodirectionnel constant (*constant forward iterator*), dans lequel sont agrégées les struc-

tures représentant I et F. La figure 3.1 schématise la structure abstraite d'un automate non déterministe à quatre états dont deux sont initiaux et deux finaux.

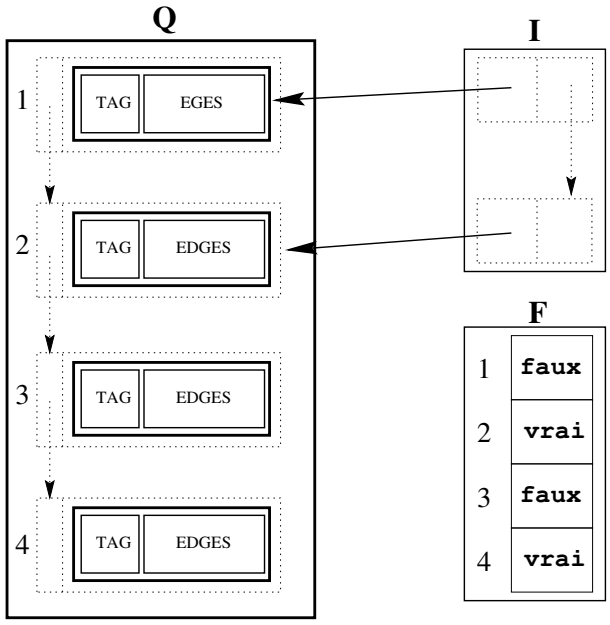

FIG. 3.1 – Un automate conceptuel à 4 états dont 2 initiaux et 2 finaux

La différence de structure entre I et F s'explique par les nuances d'utilisation entre les deux ensembles : s'agissant des états finaux, l'opération la plus courante est de tester l'appartenance d'un état à F alors que pour les états initiaux, il est plus fréquent de vouloir accéder à l'ensemble tout entier plutôt que de tester individuellement l'appartenance d'un état à I.

Le tableau de la figure 3.2 résume les différences structurelles entre automate déterministe et non déterministe.

	DFA	**NFA**
State	Identifiant de type simple (entier, pointeur, ...)	Ensemble d'identifiants. `set` contenant des identifiants simples
Edges	Container associatif à clés uniques de paires (lettre, but)	Container associatif à clés multiples de paires (lettre, but)

FIG. 3.2 – Différences conceptuelles entre automates déterministes et non déterministes

3.3 Concept et modèles d'automate déterministe

Cette section raffine le concept d'automate en présentant l'interface standard des machines déterministes ainsi que les sept modèles s'y conformant.

3.3.1 Interface et comportement

Le concept d'automate déterministe se compose d'exigences (*requirements*) sur les types externes (les paramètres d'instanciation des patrons), les types internes exportés par ces patrons ainsi que sur l'interface.

3.3.1.1 Types externes

Un modèle d'automate déterministe est un patron de classe paramétré par deux types :

Types	Description	Exigences
`Sigma`	L'alphabet	Doit se conformer au concept standard
`Tag`	Le tag associé à chaque état	Imposées par l'algorithme

`Sigma` regroupe une poignée de fonctionnalités utiles à l'automate. Ce concept d'alphabet est un sous-ensemble du trait de caractère standard `Character Traits` dont le modèle par défaut `char_traits` est utilisé par le compilateur pour l'implémentation de la classe `string`. Il doit cependant fournir une constante entière non signée et statique `size` (le nombre d'éléments de l'alphabet) ainsi que deux méthodes supplémentaires, `begin` et `end` renvoyant respectivement un itérateur sur le premier élément de Σ et un itérateur de fin de séquence sur Σ. Une classe héritant du trait standard et définissant ces trois extensions fera donc parfaitement l'affaire sachant que toutes ne sont pas nécessaires car elles dépendent de la représentation en mémoire de l'automate. L'itération sur les éléments de Σ est, elle, requise par certains algorithmes.

Les tableaux suivants fortement inspirés de la documentation SGI [26] énumèrent l'ensemble maximal d'exigences imposées au type d'alphabet :

Types Exportés	Descriptions
`Sigma::char_type`	Le type des caractères décrit par ce trait et le type des lettres étiquetant les transitions de l'automate.
`Sigma::int_type`	Un type entier capable de représenter toutes les valeurs valides d'un objet de type `char_type`
`Sigma::iterator`	Un itérateur monodirectionnel constant sur les éléments de Σ

Notations

`c`, `c1`, `c2` valeurs de type `Sigma::char_type`
`e`, `e1`, `e2` valeurs de type `Sigma::int_type`

Nom	**Expression**	**Type**
Affectation	`Sigma::assign(c1, c2)`	`void`
Égalité	`Sigma::eq(c1, c2)`	`bool`
Comparaison	`Sigma::lt(c1, c2)`[1]	`bool`
Conversion en `char_type`	`Sigma::to_char_type(e)`[2]	`Sigma::char_type`
Conversion en entier	`Sigma::to_int_type(c)`[2,3]	`Sigma::int_type`
Comparaison entière	`Sigma::eq_int_type(e1, e2)`[4]	`bool`
Début d'intervalle	`Sigma::begin()`	`Sigma::iterator`
Fin d'intervalle	`Sigma::end()`	`Sigma::iterator`
Taille	`Sigma::size`[2]	`const size_t`

Nom	**Expression**	**Sémantique**	**Postcondition**
Affectation	`assign(c1, c2)`	Effectue l'affectation `c1 = c2`	`eq(c1, c2) == true`
Égalité	`eq(c1, c2)`	retourne `true` ssi `c1` et `c2` sont égaux	
Comparaison	`lt(c1, c2)`	retourne `true` ssi `c1` est plus petit que `c2`[5]	
Conversion en caractère	`to_char_type(e)`	convertit l'entier `e` de type `int_type` en caractère de type `char_type`	
Conversion en entier	`to_int_type(c)`	convertit le caractère `c` de type `char_type` en entier de type `int_type`	`to_char_type( to_int_type(c))` est l'identité
Comparaison entière	`eq_int_type(e1,e2)`	compare les deux entiers `e1` et `e2`. Si `e1 == to_int_type(c1)` et `e2 == to_int_type(c2)`, `eq_int_type(e1,e2) == eq(c1,c2)`	
Début d'intervalle	`begin()`	renvoie un itérateur monodirectionnel constant sur le premier élément de l'alphabet	
Fin d'intervalle	`end()`	renvoie un itérateur monodirectionnel constant pointant derrière le dernier élément de l'alphabet	
Taille	`size`	Une constante statique[6] égale à $\|\Sigma\|$	`size == end()-begin()`

[1] Requis par la représentation par `map` et par la recherche dichotomique

[2] Requis par les représentations compacte et matricielle

[3] Requis par la représentation par table de hachage

[4] Requis par la représentation compacte

[5] Pour tout `c1`, `c2`, une et une seule des expressions `lt(c1,c2)`, `lt(c2,c1)`, `eq(c1,c2)` doit être vraie.

[6] Ne pas accéder à la taille par une méthode permet au compilateur les optimisations habituelles des expressions grâce aux constantes connues à la compilation

3.3.1.2 Types internes exportés

Un modèle d'automate déterministe se doit d'exporter un certain nombres de types, c'est-à-dire de les définir dans la partie publique de son interface :

Types	Descriptions
`Sigma`	Le premier paramètre d'instanciation.
`Alphabet`	`Sigma::char_type`
`State`	Le type des identifiants d'états. Habituellement un type de base simple comme les entiers ou les pointeurs.
`Tag`	Le deuxième paramètre d'instanciation.
`iterator`	Un itérateur monodirectionnel constant sur la séquence des identifiants d'états.
`Edges`	Un proxy masquant l'objet interne stockant les transitions sortant d'un état. `Edges` possède l'interface et le comportement d'une `map<Alphabet, State>` STL standard contenant les couples $(\sigma, p) \in \delta_2(q)$ pour tout état q

3.3.1.3 Interface

L'interface est volontairement limitée à des opérations basiques : ajout/suppression d'états et de transitions, fonctions de transitions δ_1 et δ_2, accès aux états initial et terminaux et aux tags. L'attribut état nul `null_state` est un état spécial renvoyé par `delta1` lorsqu'une transition n'est pas définie ou par `initial()` lorsqu'il n'y a pas d'état initial. L'utilisation d'un tel état en tant qu'argument de méthode n'est pas valide.

```
template <class sigma, class tag>
class DFA
{
public:
  typedef sigma            Sigma;
  typedef tag              Tag;
  typedef sigma::char_type Alphabet;
  typedef                  State;
  typedef                  Edges;
  State    null_state;

  // Ajouts/suppression
  State      new_state();
  void       set_trans(State from, Alphabet a, State to);
  void       del_trans(State s, Alphabet a);
  State      duplicate_state(State s);
  void       del_state(State s);

  // Modification
  void       initial(State s);
  void       copy_state(State from, State to);
  void       change_trans(State s, Alphabet a, State new_aim);
```

```
  // Accès
  State      initial() const;
  State      delta1(State s, Alphabet a) const;
  Edges      delta2(State s) const;
  const Tag& tag(State s) const;
  Tag&       tag(State s);
  bool&      final(State s);
  bool       final(State s) const;
  u_long     state_count() const;
  u_long     trans_count() const;
  iterator   begin() const;
  iterator   end() const;
}
```

Dans la table qui suit décrivant la sémantique de l'interface, nous avons utilisé comme notations :

`A`	Un objet de type `DFA` et modèle d'automate déterministe
`q`, `from`, `to`, `s`	Des identifiants d'état de type `DFA::State`
`a`	Une lettre de type `DFA::Alphabet`
`0`	L'état nul `A.null_state`

Expression	**Description & Sémantique**
`A.new_state()`*	Alloue un nouvel état `q` et retourne son identifiant. `final(q) == false` et `delta2(q).empty() == true`
`A.set_trans(from, a, to)`*	Ajoute une transition entre les états `from` et `to` étiquetée par la lettre `a`. Précondition : `delta1(from, a) == 0`. Postcondition : `delta1(from, a) == to`
`A.del_trans(s, a)`*	Supprime la transition étiquetée par `a` sortant de `s`. Précondition : `delta1(s, a) != 0`. Postcondition : `delta1(s, a) == 0`
`A.duplicate_state(s)`*	Crée un nouvel état qui est une copie de `s`. Sémantiquement équivalent à `A.copy_state(s, A.new_state());`
`A.del_state(s)`*	Supprime l'état `s`. L'identifiant doit avoir été renvoyé par la méthode `new_state`. Toutes les transitions de `s` sont supprimées. Si avant l'appel `initial() == s`, après l'appel `initial() == 0`
`A.initial(s)`*	Positionne l'état initial sur `s`. Postcondition : `initial() == s`
`A.copy_state(from, to)`*	Écrase l'état `to` avec l'état `from`. Les transitions de `to` sont supprimées avant la copie. Postconditions : `final(from) == final(to)`, `tag(from) == tag(to)`, pour tout `a`, `delta1(from, a) == delta1(to, a)`

`A.change_trans(s, a, to)`*	Redirige la transition étiquetée par `a` sortant de `s` vers `to`. Sémantiquement équivalent à :
	`A.del_trans(s, a);`
	`A.set_trans(s, a, to);`

`A.initial()`	Retourne l'état initial ou `0` s'il n'est pas défini.
`A.delta1(s, a)`	Retourne le but de la transition étiquetée par `a` sortant de `s` : $\delta_1(s, a)$. Retourne l'état nul `A.null_state` si la transition n'est pas définie.
`A.delta2(s)`	Retourne l'ensemble des transitions sortant de l'état `s` stocké dans un objet de type `Edges`.

Expression	Description & Sémantique
`A.tag(s)`	Retourne une référence sur le tag $\tau(s)$ de l'état `s`.
`A.final(s)`	Retourne une référence sur un booléen égale à `true` si `s` $\in F$. `A.final(s) = true` ajoute `s` à F et `A.final(s) = false` supprime `s` de F.
`A.state_count()`	Retourne le nombre d'états $\|Q\|$
`A.trans_count()`	Retourne le nombre de transitions $\|\Delta\|$
`A.begin()`	Retourne un itérateur sur le début de la séquence des identifiants d'état de Q.
`A.end()`	Retourne l'itérateur de fin d'intervalle sur la séquence des identifiants d'état.

Pour des raisons d'efficacité, aucune vérification de la validité des arguments ne devrait être effectuée. L'utilisateur a la responsabilité de passer des arguments valides à ces méthodes, dans le cas contraire le résultat est indéfini. En particulier, ne sont autorisés comme arguments d'appel que les identifiants d'états renvoyés par `new_state` et différents de l'état nul `null_state`.
On pourra utiliser en phase de débuggage un adaptateur implémentant des tests et des évaluations de pré et post-conditions pour chaque opération sachant que l'exécution du programme ne doit pas reposer sur d'éventuels messages d'erreur ou exceptions levées par des méthodes de l'automate.

Remarque Les méthodes marquées d'un astérisque ont un effet de bord sur la structure de l'automate, elles ne sont donc pas utilisables sur des containers constants ou sur des structures à lecture seule comme la représentation compacte. Sur ces objets, seules les opérations constantes signalées par le mot clé `const` sont valides car ils sont construits par copie et figés pendant toute leur durée de vie.

Complexités
Toutes les méthodes sont supposées s'exécuter en temps constant ou temps constant amorti exceptée la méthode `delta1` à qui l'on autorise au pire un temps d'exécution logarithmique en le nombre de transitions sortant de l'état considéré : $\log(|\vec{c}(q)|)$.

Les sections suivantes présentent sept structures de données avec leurs caractéristiques propres. Les classes correspondantes possèdent toutes l'interface standard décrite plus haut. Elle se différencient par les complexités spatiales et temporelles de leurs méthodes. Pour illustrer les représentations internes nous utiliseront l'automate de la figure 3.3.

Notations Concernant les complexités, pour X un ensemble quelconque, nous écrirons X comme raccourci pour $|X|$ et c pour $|\vec{c}(q)|$, le nombre de transitions sortant de q. Les complexités spatiales sont des approximations exprimées en octets et σ représente la quantité de mémoire nécessaire au stockage d'une lettre de l'alphabet. p représente la taille d'un pointeur et t celle d'un tag.

3.3.2 Modèle matriciel

C'est la manière la plus évidente de stocker l'ensemble de transitions : la matrice $Q \times \Sigma$ est constituée d'un vecteur par état de longueur $|\Sigma|$ contenant les buts des transitions sortantes.

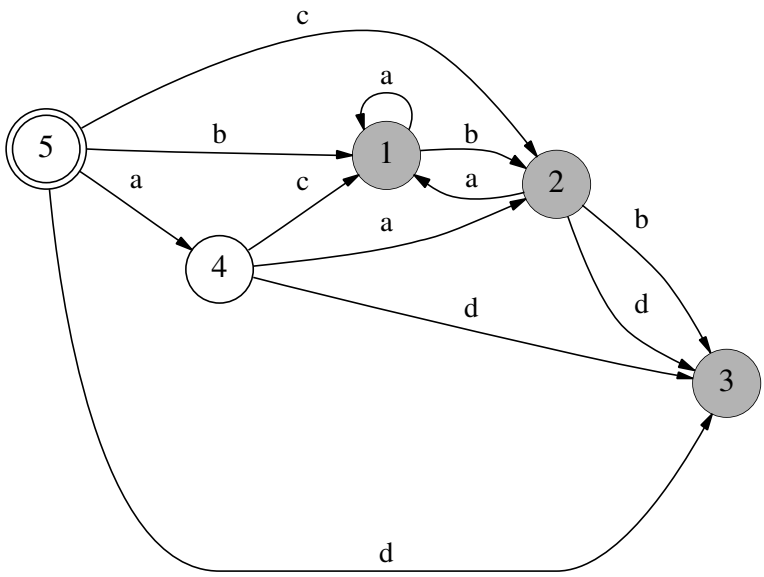

FIG. 3.3 – Automate déterministe de référence

Un tableau de pointeurs vers les lignes permet d'en garder la trace pour la désallocation de l'automate et pour le parcours des états. Les états terminaux sont mémorisés grâce à un vecteur de bits.

Cette structure est la plus rapide pour ce qui est de l'accès aux transitions à travers `delta1` :

Temps	
Ajout d'état	$O(1)$ amorti
Suppression d'état	$O(1)$
Ajout, suppression, accès transition	$O(1)$
Parcours des transitions sortantes	$O(\Sigma)$
Espace	$Q((\Sigma+1)p+t)$

FIG. 3.4 – Complexités pour la représentation matricielle

la méthode `Sigma::to_int_type` associe à chaque lettre de l'alphabet σ un indice i de colonne dans la matrice. La cellule d'indice i de la ligne q contient soit l'identifiant d'état p si $(q, \sigma, p) \in \Delta$, soit l'état nul. La durée de l'accès dépend donc uniquement de la méthode `to_int_type` qui doit s'exécuter en temps constant (cf. section 3.3.1.1). En revanche, l'itération sur la séquence des transitions d'un état suppose le parcours d'une ligne entière de la matrice en un temps proportionel à $|\Sigma|$. L'efficacité moyenne d'une telle itération est conditionnée par le

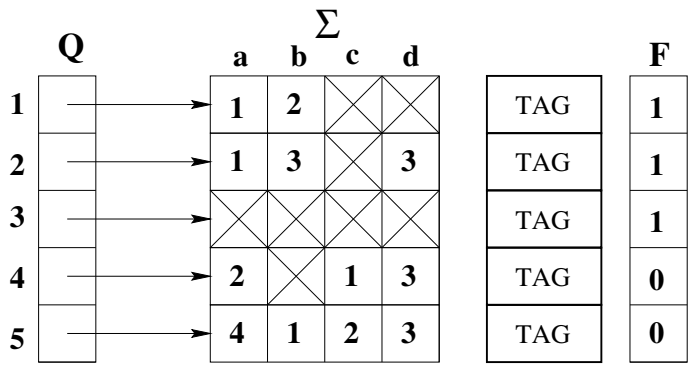

FIG. 3.5 – Représentation matricielle

taux d'occupation de la matrice α. Pour tout état $q \in Q$, on définit :

$$\alpha(q) = \frac{|\vec{c}(q)|}{|\Sigma|}$$

et la valeur moyenne :

$$\overline{\alpha} = \frac{1}{|Q|} \sum_{q \in Q} \alpha(q)$$

Pour $\overline{\alpha} = 1$, la matrice est pleine, pour $\overline{\alpha} = 0$, la matrice est vide mais le temps d'itération reste le même. Plus $\overline{\alpha}$ est petit, plus le nombre d'opérations « inutiles »est proportionnellement important. Cette valeur quantifie aussi la quantité d'espace mémoire allouée inutilement. Lorsque les contraintes sur l'espace sont fortes ou que l'itération sur $\delta_2(q)$ est fréquente et que les langages considérés entraînent des petites valeurs pour $\overline{\alpha}$, l'usage de cette structure dégrade l'efficacité des algorithmes. À l'opposé, les automates complets sur des alphabets petits et denses sont les mieux adaptés à ce type de représentation.

3.3.3 Modèle compact

Cette représentation n'utilise qu'un seul vecteur de paires (σ, p). Les lignes de la matrice décrite à la section précédente sont fusionnées en profitant des trous laissés par les transitions sortantes non définies. Grâce à la bijection `Sigma::to_int_type` notée ici $M_\Sigma : \Sigma \to N$ et projetant les lettres de l'alphabet vers les entiers, on accède à la transition sortant de q étiquetée par σ en vérifiant que la cellule d'indice $q + M_\Sigma(\sigma)$ contient bien un couple (σ, p). Si tel est le cas, p est son but sinon la transition n'est pas définie.
Les tags sont stockés dans un tableau indépendant et chaque état possède un pointeur sur le tag qui lui correspond. Les états finaux sont mémorisés dans un tableau de bits.
Lors de la construction du vecteur de transitions de la figure 3.7, les états subissent une renumérotation qui correspond à leur nouvel indice dans le tableau final : $1 \to 0$, $2 \to 2$, $3 \to 12$, $4 \to 4$, $5 \to 8$. Sur cet exemple, le compactage est particulièrement efficace puisqu'aucun trou n'apparait dans le vecteur. Avec un temps d'accès constant aux transitions, cette représentation est à la fois rapide et économique. Cependant, les automates compacts sont construits par copie et n'autorisent pas d'opération à effet de bord, c'est pourquoi leur interface ne comporte que les méthodes constantes de l'interface standard.

Temps	
Ajout, suppression d'état	non
Ajout, suppression de transition	non
Accès transition	$O(1)$
Parcours des transitions sortantes	$O(\Sigma)$
Espace	$\approx \Delta(\Sigma + p) + Qt$

FIG. 3.6 – Complexités pour la représentation compacte

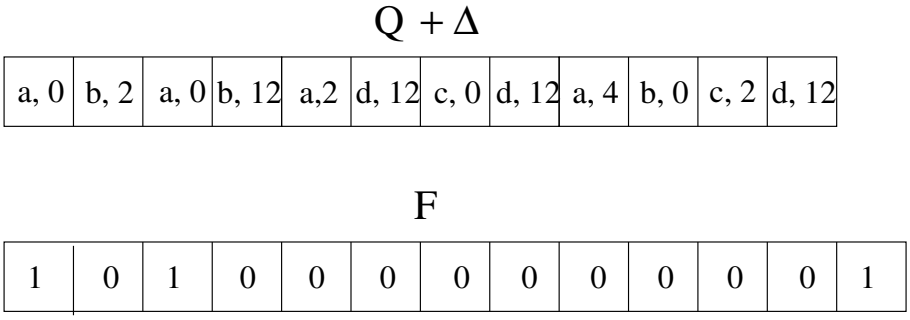

FIG. 3.7 – Représentation compacte

L'espace mémoire utilisé est difficile à quantifier car il dépend de la répartition des transitions dans la matrice, donc de la nature des langages manipulés. Pour des dictionnaires simples de l'anglais ou du français, c'est-à-dire les automates minimaux reconnaissant des ensembles de quelques centaines de milliers de mots, le nombre moyen de transitions par état est de deux, ce qui, comparé à la taille des alphabets (au moins 26 lettres chacun) indique une matrice extrêmement creuse. De plus, le nombre relativement important d'états, quelques dizaines de milliers, augmente les chances d'avoir une répartition adéquate des transitions. Il apparaît que pour un automate du français reconnaissant 40 000 mots sur un alphabet composé de 39 lettres[1], la compaction entraîne une diminution de taille mémoire d'environ 75%.
Pour une description plus detaillée de la représentation par un unique tableau de transitions se référer à [52].

3.3.4 Implémentation par `map`

Dans cette représentation, à chaque état est associée une `map` STL, c'est-à-dire un container associatif de paires à clés uniques `map<Alphabet, State>`. Cette structure assure un accès en temps logarithmique aux transitions grâce à la relation d'ordre partielle définie sur les clés, ici les lettres. Elle constitue un bon compromis espace/vitesse car elle est généralement moins gourmande en mémoire que la représentation matricielle. En outre, elle n'impose à l'alphabet qu'une relation d'ordre, c'est-à-dire un opérateur `<` défini sur le type `Alphabet` alors que la matrice nécessite une bijection de Σ vers les entiers et son inverse. Une `map` repose généralement sur un arbre équilibré du type arbre rouge/noir. La figure 3.9 schématise une structure où les ensembles de transitions sont représentés par des arbres binaires de

[1] 26 lettres plus les caractères accentués

Temps	
Ajout d'état	$O(1)$ amorti
Suppression d'état	$O(1)$
Ajout, suppression, accès transition	$O(\log(c))$
Parcours des transitions sortantes	$O(c)$
Espace	$Q(p+t)+\Delta(\sigma+4p)$

FIG. 3.8 – Complexités pour la représentation par `map`

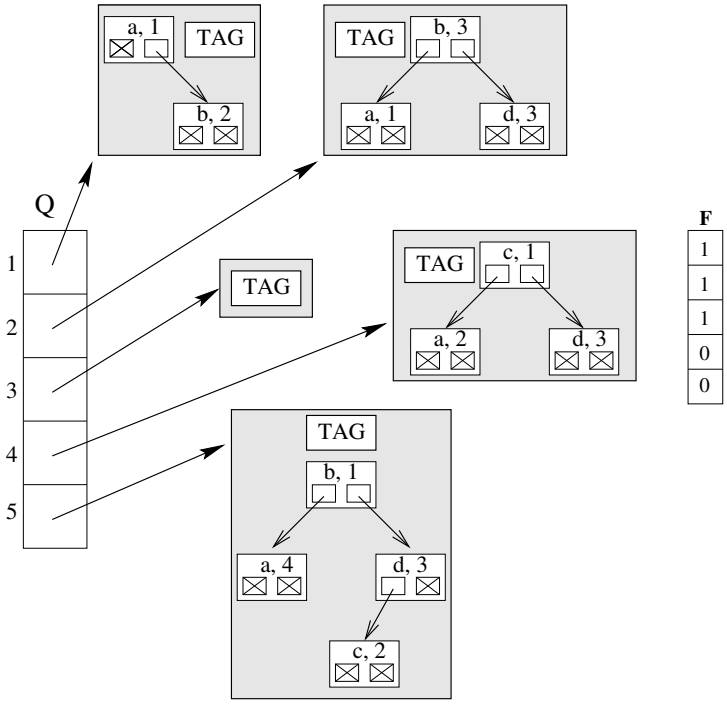

FIG. 3.9 – Représentation par map

paires (lettre, but) dont la clé est la lettre. Les fils gauches de chaque nœud possèdent des clés inférieures lexicographiquement à celle de la racine et les fils droits des clés supérieures.

3.3.5 Implémentation par hachage

Cette représentation présente l'avantage d'une complexité spatiale dépendant principalement du nombre de transitions $|\Delta|$. Une structure de données de type container associatif de hachage unique pour tout l'automate, contient les triplets (q, σ, p) auxquels on accède grâce aux clés (q, σ). Un tel container est paramétré par une fonction de hachage qui projette les clés vers les entiers positifs. On utilise la bijection `Sigma::map`, notée M_Σ, pour définir la fonction de hachage :

$$\begin{array}{rcl} h : Q \times \Sigma & \rightarrow & N \\ (q, \sigma) & \mapsto & q|\Sigma| + M_\Sigma(\sigma) \end{array}$$

de manière à quasiment éliminer les collisions. En effet, cette fonction, jusqu'à une certaine limite, assigne un entier unique à toute paire (q, σ) pour peu que le nombre d'états ou la taille de l'alphabet ne dépassent pas les limites du raisonnable : en considérant des entiers longs de 32 bits et un alphabet de 256 caractères, on pourra avoir 2^{32} valeurs de hachage différentes pour 2^{32} couples (q, σ) différents. Sachant qu'on aura au plus 256 transitions par état, le nombre d'états limite est $\frac{2^{32}}{256} = 2^{24}$ soit plus de 16 millions d'états. Au-delà de cette limite, les collisions provoquées par la fonction risquent de dégrader légèrement les performances d'accès, mais en deçà, l'accès est souvent bien meilleur que la recherche en temps logarithmique.

Pour s'affranchir presque totalement du nombre d'états dans la complexité spatiale de l'au-

Temps	
Ajout d'état	0
Suppression d'état	$O(\Sigma)$
Ajout, suppression, accès transition	$\approx O(1)$
Parcours des transitions sortantes	$O(\Sigma)$
Espace	$\Delta(\sigma + 3p)$

FIG. 3.10 – Complexités pour la représentation par table de hachage

tomate, il n'est pas possible de stocker des tags car l'espace mémoire qu'ils consomment est directement proportionnel à la taille de Q. En revanche, et c'est la seule structure qui croit avec le nombre d'états, le tableau de bits servant à marquer les états terminaux est toujours présent mais reste négligeable face aux quantités de données manipulées par ailleurs. Cette représentation est donc particulièrement intéressante lorsqu'on a beaucoup d'états et peu de transitions.

La suppression d'un état est une opération problématique car il faut rechercher une par une toutes les transitions susceptibles d'être définies, d'où les temps de calcul semblables pour le parcours des transitions et la suppression d'un état, tous deux proportionnels à la taille de l'alphabet.

3.3.6 Implémentation par listes d'adjacence

Les trois représentations suivantes associent une séquence de paires (σ, p) à chaque état de l'automate. L'accès à une transition est effectué soit par recherche dichotomique lorsque les éléments sont ordonnés dans un container à accès direct comme le vecteur, soit linéairement mais de manière optimisé grâce à une des deux heuristiques présentées ci-dessous.

3.3.6.1 Optimisation par recherche dichotomique

Les paires sont maintenues triées dans le tableau grâce à la relation d'ordre défini sur les lettres, garantissant un accès en temps logarithmique par recherche dichotomique. Pour un temps d'accès similaire, cette représentation est plus économique qu'une représentation par `map` car maintenir une structure d'arbre suppose des données supplémentaires. En revanche, l'insertion d'une transition dans un vecteur est une opération en temps linéaire ce qui est bien moins efficace que dans un arbre.

Temps	
Ajout d'état	$O(1)$ amorti
Suppression d'état	$O(1)$
Ajout de transition	$O(c)$
Suppression d'une transition	$O(c)$
Accès à une transition	$O(\log(c))$
Parcours des transitions sortantes	$O(c)$
Espace	$Q(p+t)+\Delta(\sigma+p)$

FIG. 3.11 – Complexités pour le modèle par recherche dichotomique

3.3.6.2 La méthode « move-to-front »

Dans cette représentation, la séquence des transitions sortantes est une liste chaînée. La recherche d'une transition `t` est linéaire mais accélérée lorsque les accès ne sont pas équiprobables : l'heuristique consiste à placer `t` en première position de la liste une fois qu'on l'a trouvée. Cela revient à opérer une rotation d'un élément vers la droite de la sous-séquence `[x,y)` où `x` est la position de début de séquence et `y` la position suivant celle de `t`. La figure 3.12 montre comment après avoir trouvé la transition $(b, 1)$ de l'état 5, les trois premières cellules de la liste subissent une rotation vers la droite qui place la transition voulue en tête, simplement en supprimant la transition de la liste puis en la réinsérant au début.
De cette manière, on mise sur le fait que cette transition soit à nouveau recherchée dans un

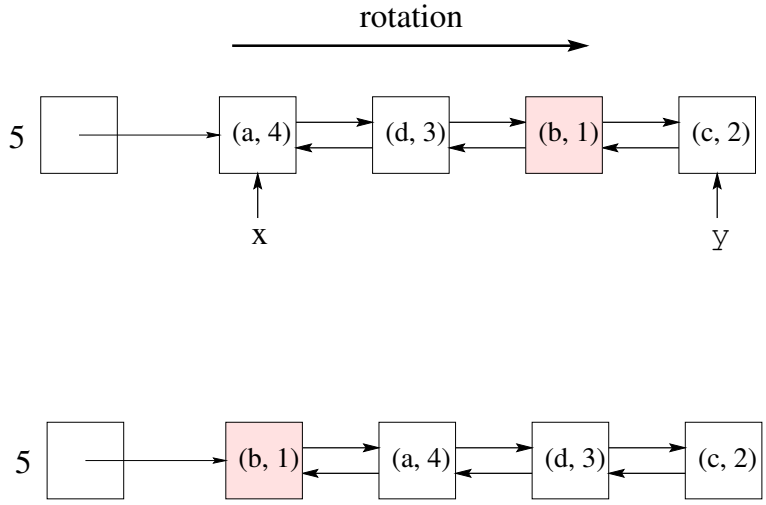

FIG. 3.12 – L'accès à la transition $(b, 1)$

futur proche. Au second accès, le nombre de comparaisons nécessaires à sa localisation se réduira à un. Au fur et à mesure des accès, les transitions les plus « populaires » se retrouveront en début de séquence, réduisant d'autant plus le nombre de tests qu'elles sont fréquemment utilisées. Évidemment, si la probabilité d'accéder à une transition est la même quelle que soit la lettre, l'optimisation est compromise et on retombe en temps véritablement linéaire. Pour des mots issus de la langue naturelle, du moins pour les langues européennes, les probabilités

d'apparition des lettres sont inégales ce dont on profite pleinement ici. L'autre aspect positif est la rapidité d'ajout d'une transition (temps constant) qui autorise la construction en temps raisonnable de gros automates.

Temps	
Ajout d'état	$O(1)$ amorti
Suppression d'état	$O(c)$
Ajout d'une transition	$O(1)$
Suppression d'une transition	$\leq O(c)$
Accès à une transition	$\leq O(c)$
Parcours des transitions sortantes	$O(c)$
Espace	$Q(p+t) + \Delta(\sigma + 3p)$

FIG. 3.13 – Complexités pour la représentation « move-to-front »

3.3.6.3 La méthode « transpose »

L'heuristique « transpose » consiste à échanger à chaque accès la transition trouvée avec sa voisine de gauche immédiate, ce qui a pour effet de trier incrémentalement les transitions par la méthode du tri à bulles et par ordre de probabilité d'accès décroissante, les transitions les plus utilisées se retrouvant en début de séquence. Dans le fond, cette méthode, ainsi que la précédente, revient à déduire une relation d'ordre sur l'alphabet au fur et à mesure du traitement, en fonction de l'utilisation qui est faite de l'automate. La différence pratique avec la méthode « move-to-front », est que le réarrangement des transitions est moins brutal car les éléments remontent progressivement vers la tête. Une transition se trouve placée avant une autre parce que son nombre d'accès est supérieur à celui de sa voisine de droite alors qu'en la déplaçant directement en tête au moindre accès, on est plus enclin à des réarrangements ponctuels ne représentant pas les probabilités moyennes.
Cette représentation fait usage de vecteurs pour stocker les séquences de transitions, ce qui explique les complexités d'ajout et suppression des transitions. Le déplacement des transitions ne s'effectuant pas par suppression et insertion successives comme dans la représentation précédente, mais simplement par échange de valeur, on peut faire l'économie des pointeurs de la liste chaînée sans perdre en efficacité lors de l'accès.

Temps	
Ajout d'état	$O(1)$ amorti
Suppression d'état	$O(1)$
Ajout d'une transition	$O(1)$ amorti
Suppression d'une transition	$O(c)$
Accès à une transition	$\leq O(c)$
Parcours des transitions sortantes	$O(c)$
Espace	$Q(p+t) + \Delta(\sigma + p)$

FIG. 3.14 – Complexités pour la représentation « transpose »

3.4 Concept et modèles d'automate non-déterministe

Le concept d'automate non déterministe ne diffère de son homologue déterministe que par la nature de ses états. En effet, la simulation par une machine du comportement d'un tel automate oblige à considérer un état non déterministe comme un ensemble d'états déterministes. La section suivante détaille les exigences qui sont imposées à ce concept.

3.4.1 Interface, comportement et types externes

L'interface, le comportement, la sémantique et les pré et post-conditions des méthodes sont les mêmes que ceux décrits à la section 3.3 pour les automates déterministes. Le types externes `Tag` et `Sigma`, paramètres d'instanciation, doivent être conformes aux mêmes concepts. Les différences se situent dans les types internes et concernent en particulier les type exportés `State` et `Edges`.

3.4.2 Types internes exportés

Types	Descriptions
`Sigma`	Le premier paramètre d'instanciation.
`Alphabet`	`Sigma::char_type`.
`State`	Le type des identifiants d'états. Possède l'interface d'un `set` constant contenant des identifiants d'états simples.
`Tag`	Le deuxième paramètre d'instanciation.
`iterator`	Un itérateur monodirectionnel constant sur la séquence des identifiants d'états.
`Edges`	Un proxy masquant l'objet interne stockant les transitions sortant d'un état. `Edges` possède l'interface et le comportement d'une `multimap<Alphabet, State>` STL standard contenant les couples $(\sigma, p) \in \delta_2(q)$ pour tout état q.

3.4.3 Implémentation par `multimap`

Une `map` est un container associatif à clés uniques : elle permet d'associer une valeur à une clé. Autrement dit il ne peut exister dans le container qu'un exemplaire de chaque clé. En choisissant comme clés les lettres de l'alphabet et comme valeurs les identifiants d'états, cette structure de données est adaptée pour stocker les transitions sortant d'un état d'un automate déterministe. Une `multimap` est un container associatif à clés multiples, c'est-à-dire qu'il est possible d'y ajouter plusieurs exemplaires de la même clé, chaque exemplaire étant associé à une valeur différente. En fait, cela revient à associer un ensemble de valeurs à une clé, en l'occurrence un ensemble d'états à une lettre. Dans cette implémentation, chaque état q possède sa `multimap` permettant d'accéder à l'ensemble des états atteints en suivant les transitions sortant de q et étiquetées par une lettre précise. En outre, le temps d'accès est proportionnel au logarithme du nombre de clés présentes dans le container, ce qui est requis par les spécifications de l'abstraction automate décrite au début de ce chapitre.

3.4.4 Modèle matriciel

Dans le cas non déterministe, la matrice possède trois dimensions : à chaque état est associé une ligne, à chaque lettre de l'alphabet une colonne et les cellules contiennent des

Temps	
Ajout d'état	$O(1)$
Suppression d'état	$O(c)$
Ajout d'une transition	$O(\log(c))$
Suppression d'une transition	$O(\log(c))$
Accès à une transition	$O(\log(c))$
Parcours des transitions sortantes	$O(c)$
Espace	$Q(4p + t) + \Delta p$

FIG. 3.15 – Complexités pour la représentation par multimap

containers d'identifiants d'états, en l'occurrence des vecteurs. Cette structure de données doit être utilisée avec précaution car si elle est très efficace pour ce qui est de l'accès aux transitions (temps constant), elle est malheureusement grosse consommatrice d'espace mémoire car on alloue $|Q| \times |\Sigma|$ vecteurs. Son utilisation ne devrait être envisagée que pour les petits automates ou les petits alphabets et dans le cas où la vitesse d'accès aux transitions est cruciale.

Temps	
Ajout d'état	$O(\Sigma)$
Suppression d'état	$O(\Sigma)$
Ajout d'une transition	$O(1)$ amorti
Suppression d'une transition	$O(c)$
Accès à une transition	$O(1)$
Parcours des transitions sortantes	$O(\Sigma)$
Espace	$Q(\Sigma p + t) + \Delta p$

FIG. 3.16 – Complexités pour la représentation par matrice 3D

3.5 Conclusion

Les containers d'automate présentés dans ce chapitre présentent toutes les propriétés nécessaires à leur introduction dans une architecture logicielle générique : réutilisabilité, car leur interface et leur comportement sont standardisés (polymorphisme), adaptabilité, de par leurs complexités spatiale et temporelle hétérogènes et enfin efficacité, grâce au typage statique, aux patrons et à l'expansion en ligne. De plus, l'existence de spécifications (les concepts) les plus formelles possibles garantit une extensibilité quasi infinie par ajout de nouvelles classes à la batterie déjà disponible.
Ces containers constituent des fondations solides sur lesquelles reposeront l'ensemble des accesseurs (les curseurs, décrits au chapitre suivant) et au-dessus, les algorithmes.
Pour une documentation de référence exhaustive des concepts et modèles relatifs aux containers d'automate d'ASTL, voir l'annexe A.

Chapitre 4

Les curseurs

Concept central d'ASTL, le curseur centralise les fonctions d'accès aux données et leur conversion au format standard qu'exigent les algorithmes. Il constitue avec l'itérateur la glu qui lie tous les autres composants logiciels entre eux.

Le concept de curseur généralise celui d'itérateur : né de la nécessité de parcourir ou traverser des structures de données, un modèle de curseur est un accesseur, c'est-à-dire un objet représentant une position dans un automate et capable de lire la valeur s'y trouvant. La notion de position regroupe différents concepts selon les besoins algorithmiques : un état q, une transition (q, a, p) ou un chemin. C'est pourquoi la notion de curseur se divise en quatre sous-notions liées par des relations de sous-typage ou d'encapsulation et qui s'imbriquent à la manière des poupées gigognes (voir figure 4.1), l'itérateur classique apparaissant alors comme un cas particulier du curseur. En effet, un itérateur n'est rien d'autre qu'un curseur qui connaît l'enchaînement des transitions à suivre le long de l'automate puisque sur une séquence il n'y a jamais de décision à prendre lors de l'incrémentation. Un curseur muni d'une politique de parcours est un itérateur.
Par raffinements successifs du concept *curseur simple* (cursor) on en déduit les concepts *curseur monodirectionnel* (forward cursor), *curseur pile* ou *file* (stack/queue cursor) et *curseur de parcours* (depth-first cursor dans le cas du parcours en profondeur par exemple). Les curseurs simples et monodirectionnels constituent la base commune à tous les algorithmes de parcours implémentés dans les deux couches les plus externes.
La puissance de ces concepts vient essentiellement de la facilité à modifier leur comportement standard pour étendre leurs fonctionnalités et leurs champs d'application grâce aux adaptateurs.

4.1 Notations

Les sections suivantes décrivent les concepts importants relatifs aux curseurs. Les modèles s'y conformant sont majoritairement des patrons de classes et les algorithmes des patrons de fonctions. Pour chaque concept, une description des paramètres d'instanciation est donnée, cependant, dans le but de clarifier l'exposé et partout où cela ne prêtait pas à confusion, les exemples de codes ont été épurés de toute référence aux patrons, c'est-à-dire au mot clé `template`. Généralement, baptiser judicieusement les types et les noms de variables suffit à rendre le code assez clair pour pouvoir se focaliser sur les abstractions et masquer les détails

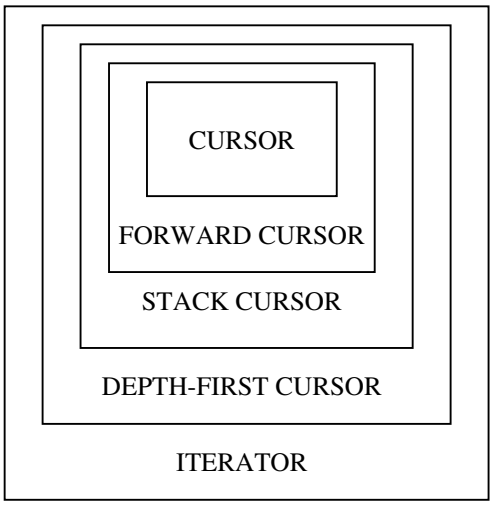

FIG. 4.1 – Les concepts de curseur pour le parcours en profondeur

d'implémentation.
Pour une description technique plus détaillée et des exemples complets d'utilisation se reporter à l'annexe A.3.

4.2 Motivations

À l'origine de l'introduction du concept de curseur il y a :

1. La nécessité de découpler au maximum l'algorithme de la structure de données. Les détails d'implémentation n'intéressent pas l'algorithme qui doit s'affranchir de tout type de représentation en mémoire en faisant un minimum de suppositions sur les objets. La communication s'établit à travers une interface standardisée imposant le moins de choses possible aux deux parties. Le masquage des données est fondamental en programmation générique.
2. Le besoin de factoriser les fonctionnalités communes les plus utilisées et de les placer hors de l'algorithme et de la structure de données afin de minimiser la quantité de code à écrire. Les fonctions de parcours sur les automates présentent des similitudes quels que soient les types d'automates, déterministes ou non, synchrones/asynchrones, etc., et quels que soient les algorithmes, copie, lecture sur flux, etc. Les curseurs « capturent » les propriétés les plus significatives et les plus partagées des traitements appliqués aux automates.
3. La lourdeur des techniques habituelles utilisées pour implémenter les parcours génériques, c'est-à-dire le patron de conception *visiteur* dont la mise en œuvre ne se justifie pas toujours, particulièrement dans les cas les plus simples où son utilisation aurait plutôt tendance à compliquer l'écriture et la compréhension du code. On a souvent besoin d'un parcours classique en profondeur ou en largeur ne nécessitant pas de fonctionnalité particulière ; les curseurs permettent la réutilisation de ces algorithmes avec seulement

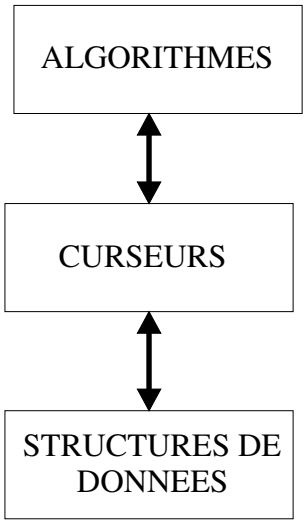

FIG. 4.2 – La structure à trois couches d'ASTL

trois lignes de code.

Les excellents résultats obtenus par STL en utilisant un modèle à trois couches incitent fortement à généraliser cette technique (figure 4.2).

4.3 Le curseur simple

Le curseur constitue le modèle de base dont découle toute la hiérarchie des accesseurs sur automate.

Notation On notera C l'ensemble des curseurs et $c_q \in C$ un curseur pointant sur un état $q \in Q$ de l'automate $A(\Sigma, Q, i, F, \Delta)$.

4.3.1 Définition

Un curseur c_q est un objet pointant sur un état $q \in Q$ de l'automate A. Son rôle consiste à garder la trace de l'état courant tout au long de l'algorithme. Il permet en outre d'implémenter un parcours simple le long d'un chemin.

4.3.2 Propriétés

Un curseur :

1. Possède une valeur singulière lorsqu'il ne pointe sur aucun état ou lorsqu'il pointe sur l'état nul.
2. Est assignable, c'est-à-dire qu'il est possible d'affecter à $c \in C$ la valeur d'un curseur c'. Il est également possible de lui affecter un état $q' \in Q$ après quoi $c = c_{q'}$.

3. Définit une relation d'équivalence :

$$c_q \equiv c'_{q'} \Leftrightarrow q = q'.$$

   Deux curseurs sont équivalents lorsqu'ils pointent sur le même état.
4. Est constructible par défaut. Il possède alors une valeur singulière, c'est-à-dire que sans initialisation préalable, q n'est pas défini donc c_q est inutilisable.
5. Est incrémentable si et seulement si il ne possède pas de valeur singulière.
6. Est déréférençable si et seulement si il ne possède pas de valeur singulière.

4.3.3 Comportement et interface

Le principal intérêt du curseur est sa capacité à suivre un chemin dans l'automate par applications successives de la fonction d'accès δ_1 qu'on étend aux curseurs de manière naturelle :

$$\delta_1 : C \times \Sigma \to C$$

$$\delta_1(c_q, \sigma) = \begin{cases} c_p \text{ si } (q, \sigma, p) \in \Delta \\ c_0 \text{ sinon.} \end{cases}$$

δ_1 permet à un curseur de passer d'état en état en suivant des transitions étiquetées par des lettres précises. Elle apparaît dans l'interface sous le nom de `forward`. Lorsque la transition requise n'est pas définie, le curseur est envoyé sur l'état nul et il possède alors une valeur singulière détectable grâce à la méthode `sink`.

Dans le tableau suivant décrivant l'interface standard d'un curseur, nous utiliserons comme notation :

`X`	pour un type qui est un modèle de curseur
`x, y`	pour désigner deux objets de type `X`
`a`	pour une lettre
$q \in Q$	pour l'état sur lequel pointe `x`
`p`	pour un état quelconque de l'automate

Méthode	**Expression**	**Sémantique**
état source	`x.src();`	retourne q
source final	`x.src_final();`	retourne `vrai` si $q \in F$
avancer par `a`	`x.forward(a);`	avance le long de la transition sortant de q étiquetée par `a`. Retourne `vrai` si $\delta_1(q, a) \neq 0$
existence	`x.exists(a);`	retourne `vrai` si $\delta_1(q, a) \neq 0$
nul	`x.sink();`	retourne `vrai` si `x` pointe sur l'état nul.
comparaison	`(x == y)`	`vrai` si `x` représente le même état que `y`
affectation	`x = y;`	post-condition : `x` est une copie de `y` et pointe sur le même état
affectation d'un état	`x = p;`	post-condition : `x` pointe sur l'état `p`
constructeur par défaut	`X x;`	`x` a une valeur singulière. Seule l'affectation est autorisée
constructeur de copie	`X x = y;`	`x` est une copie de `y`

Lorsqu'un curseur possède une valeur singulière, la seule opération dont la validité est garantie est l'affectation : le curseur se retrouve dans une impasse à la suite d'une opération ayant échoué et la seule opération ayant un sens est de le repositionner sur un état valide de l'automate, c'est-à-dire différent de l'état nul. Un tel évènement survient par exemple lorsque la fonction de transition `forward` échoue mais un curseur a aussi une valeur singulière lorsqu'il n'est pas initialisé. La méthode `sink` permet de tester si le curseur se trouve sur l'état nul. Dans le cas contraire, toutes les opérations standards sont valides et le comportement est bien défini.

4.3.4 Paramètres d'instanciation

Les curseurs standards sont paramétrés par le type de l'automate sur lequel ils vont évoluer. La classe d'automate utilisée pour instancier le patron doit se conformer au concept DFA (automate déterministe) décrit au chapitre 3 :

```
template <class DFA>
class cursor
{
  ...
};
```

Remarque Ce paramétrage ne concerne que les curseurs standards. Bien évidemment, les adaptateurs possédant un comportement différent sont sujets à des variations dans les types et le nombre des paramètres d'instanciation.

4.3.5 Exemple

L'algorithme 2.1 page 31, intensivement utilisé, teste l'appartenance d'un mot w au langage reconnu par un automate A.
De cet algorithme on en déduit l'implémentation de la figure 4.3 en considérant :

1. L'équivalence mot $\equiv$ séquence et par transitivité mot $\equiv$ intervalle. Le mot w est donc défini par deux positions ou itérateurs : `first` $= i_{w_0}$ et `last` $= j_{w_{|w|}}$.
2. Un accès à l'automate A à travers un curseur initialement égal à c_i.
3. Les notations `forward` pour $\delta_1 : C \times \Sigma \to C$ et `src_final` pour le test d'appartenance de q à F.

```
bool appartient(InputIterator first, InputIterator last, Cursor c) {
  while (first != last && c.forward(*first))
    ++first;
  return first == last && c.src_final();
}
```

FIG. 4.3 – L'implémentation `appartient`

Tant que la fonction de transition n'échoue pas, on avance sur les transitions étiquetées par les lettres du mot w. Après avoir lu toutes les lettres, si l'état atteint est final le mot est reconnu.

4.4 Le curseur monodirectionnel

4.4.1 Définitions

4.4.1.1 Transition puits

On appelle *transition puits* tout triplet $(q, \sigma, p) \in Q \times \Sigma \times Q$ pour lequel la fonction de transition δ_1 échoue, i.e. $\delta_1(q, \sigma) = p = 0$. Elles sont donc étiquetées par $\Sigma \setminus \vec{c}(q)$ et pointent vers l'état nul ou puits. Par soucis de simplification, on limitera ces transitions à une par état ce qui revient à intégrer à notre modèle d'automate le mécanisme d'état par défaut décrit à la section 2.2.5.1.

4.4.1.2 Curseur monodirectionnel

Le concept de curseur monodirectionnel raffine celui de curseur simple car c'est un pointeur sur une transition $(q, \sigma, p) \in Q \times \Sigma \times Q$ de l'automate. Il permet non seulement l'accès à cette transition mais aussi le parcours de l'ensemble des transitions sortant de l'état q soit $\delta_2(q)$. On notera c_δ un curseur pointant sur la transition $\delta = (q, \sigma, p)$.

Remarque Le curseur monodirectionnel est déjà un curseur de parcours, mais sa politique d'itération est limitée à des « descentes » dans l'automate dans le sens des transitions définies. Des schémas d'itération plus ambitieux comme le parcours en profondeur qui nécessitent de « remonter » dans l'historique des transitions traversées induisent d'autres concepts présentés dans les sections suivantes.

4.4.2 Propriétés

Un curseur monodirectionnel possède des propriétés équivalentes à celles du curseur définies à la section 4.3.2 mais généralisées aux transitions :

1. Un curseur monodirectionnel a une valeur singulière lorsqu'il ne pointe sur aucune transition ou sur une transition puits.
2. Il est assignable.
3. Il est constructible par défaut. Il possède alors une valeur singulière.
4. Il est incrémentable et déréférençable s'il ne pointe pas sur une transition puits.
5. La relation d'équivalence sur ces curseurs s'applique aux transitions et non plus uniquement aux états sources :
$$c_\delta \equiv c'_{\delta'} \Leftrightarrow \delta = \delta'$$
6. La transition puits $(q, \epsilon, 0)$ matérialise la position de fin de la séquence des transitions sortant d'un état q :
$$\text{pour un curseur } c_{(q,\sigma,p)},\ \delta_2(q) = ((\sigma_1, p_1), ..., (\sigma_n, p_n), (\epsilon, 0))$$

Autrement dit, partant de la première transition (q, σ_1, p_1), un nombre fini d'incrémentations, éventuellement nul, mène c en position fin de séquence $(q, \epsilon, 0)$. L'ordre dans lequel l'ensemble est parcouru n'est pas défini, il n'est cependant pas interdit d'en imposer un à des fins algorithmiques ou d'efficacité.

4.4.3 Comportement et interface

Un modèle de curseur monodirectionnel doit implémenter l'interface du curseur, c'est-à-dire toutes les fonctionnalités concernant l'état source q de la transition pointée (q, σ, p), plus le comportement relatif à l'accès des transitions sortantes : le choix et l'accès à l'étiquette σ et à l'état but p. La table suivante en résume les fonctionnalités :

Méthode	**Expression**	**Sémantique**
état source	`x.src();`	retourne q
source final	`x.src_final();`	retourne `vrai` si $q \in F$
avancer par `a`	`x.forward(a);`	avance le long de la transition sortant de q étiquetée par `a`. Retourne `vrai` si $\delta_1(q, a) \neq 0$
existence	`x.exists(a);`	retourne `vrai` si $\delta_1(q, a) \neq 0$
nul	`x.sink();`	retourne `vrai` si $q = 0$.
état but	`x.aim();`	retourne p
lettre	`x.letter();`	retourne σ
but final	`x.aim_final();`	retourne `vrai` si $p \in F$
comparaison	`(x == y)`	`vrai` si `x` représente la même transition que `y`
avancer	`x.forward();`	avance le long de la transition pointée par `x`
$1^{ère}$ transition	`x.first_transition();`	positionne `x` sur la première transition sortant de q
transition suivante	`x.next_transition();`	positionne `x` sur la transition suivante de l'ensemble $\delta_2(q)$
trouver	`x.find(a);`	positionne `x` sur la transition étiquetée par `a` si elle existe
constructeur par défaut	`X x;`	`x` a une valeur singulière. Seule l'affectation est autorisée
constructeur de copie	`X x = y;` `X x(y);`	`x` est une copie de `y`
affectation	`x = y;`	post-condition : `x` est une copie de `y`
affectation d'un état	`x = p;`	post-condition : `x` pointe sur l'état `p`

Les méthodes `first_transition` et `next_transition` implémentent l'itération sur les transitions sortant de l'état source. Elles renvoient toutes les deux une valeur booléenne indiquant l'état du curseur après l'opération : `faux` si l'ensemble des transitions est épuisé, ce qui signifie que le curseur pointe sur la transition puits $(q, \epsilon, 0)$, qu'il a une valeur singulière et que jusqu'à ce qu'il soit repositionné sur une transition valide, seules les opérations concernant

l'état source q sont autorisées. De même, la méthode `find` recherche la transition étiquetée par `a`, le positionne dessus et renvoie `vrai` si elle existe. Dans le cas contraire, le curseur se retrouve sur la transition puits.
La méthode `forward()` ne renvoie pas de valeur et ne doit être utilisée que lorsque la transition pointée est bien définie, c'est-à-dire après que l'invocation d'une méthode de positionnement a renvoyé `vrai`.

Remarque Un accès séquentiel déterministe[1] aux transitions sortant d'un état suppose une relation d'ordre sur les couples $(\sigma, p) \in \Sigma \times Q$. Cette relation impose l'ordre dans lequel l'ensemble $\delta_2(q) = ((\sigma_1, p_1), ..., (\sigma_n, p_n), (\epsilon, 0))$ sera parcouru par appels successifs à la méthode `next_transition` :

$$i < j \Leftrightarrow (\sigma_i, p_i) < (\sigma_j, p_j)$$

Beaucoup d'algorithmes se contentent d'un accès non déterministe, c'est-à-dire sans ordre prédéfini même d'une itération sur l'autre, mais il arrive que l'efficacité repose sur l'existence de la relation d'ordre. Par exemple, les opérations ensemblistes effectuent le parcours des transitions sortant d'un état en temps linéaire grâce à la relation d'ordre définie sur Σ :

$$(\sigma, p) < (\sigma', p') \Leftrightarrow \sigma < \sigma'$$

ce qui signifie un accès à la séquence des transitions dans l'ordre alphabétique des lettres les étiquetant[2], dans le cas contraire la complexité est quadratique. La nature de la relation d'ordre dépend de la représentation en mémoire du container d'automate et du comportement du curseur.

Le concept de forward cursor prépare le terrain aux curseur de parcours et aux algorithmes plus sophistiqués. La structure non séquentielle des automates oblige à introduire le concept de trajectoire et à distinguer position dans l'automate et position dans l'algorithme.

4.4.4 Paramètres d'instanciation

Bien que conceptuellement un curseur monodirectionnel contienne un curseur simple et adapte son comportement, pour des raisons de cohérence et de simplification d'utilisation[3], le patron de curseur monodirectionnel standard est paramétré par la classe d'automate. Ce patron hérite de l'interface et de l'implémentation de la classe `cursor` :

```
template <class DFA>
class forward_cursor : public cursor<DFA>
{
  ...
};
```

[1] Ici déterministe désigne un accès aux transitions dans un ordre bien défini et immuable tant que le container n'a pas effectué explicitement d'opérations à effet de bord, à ne pas confondre avec la propriété de déterminisme des automates.

[2] On considère $\sigma < \sigma'$ comme la relation d'ordre alphabétique classique.

[3] L'implémentation du `forward_cursor` repose sur celle du `cursor`. L'utilisation de toute autre classe possédant un comportement même très légèrement différent peut provoquer des incohérences car les suppositions sur les invariants, les pré et post-conditions des méthodes peuvent alors s'avérer érronées. C'est pourquoi on force le `forward_cursor` officiel à utiliser la classe officielle `cursor` et qu'il n'est pas possible d'en changer. En outre, deux paramètres d'instanciation auraient rendu son utilisation malaisée.

4.4.5 Exemple

L'implémentation de la figure 4.4 affiche l'ensemble des transitions sortant de l'état initial d'un automate `A` grâce au curseur `c`.

```
DFA A;
forward_cursor<DFA> c(A, A.initial());
if (c.first_transition()) {
  do {
    cout << c.src() << c.letter() << c.aim() << endl;
  } while (c.next_transition());
}
```

FIG. 4.4 – Affichage des transitions sortant de l'état initial de `A`

4.5 Trajectoire

Cette section introduit la notion de *trajectoire* qui généralise celle de chemin définie à la section 2.2.1. Elle unifie deux concepts distincts selon que l'on parle de parcours en profondeur (PEP) ou en largeur (PEL) et est indispensable au concept d'algorithme de parcours : à une étape donnée d'un parcours, on appelle trajectoire la suite de transitions visitées jusqu'ici. Dans le cas du PEP, on a équivalence entre trajectoire et chemin classique et cette trajectoire est stockée dans une pile. Dans le cas du parcours en largeur la trajectoire est stockée dans une file.

Cette notion de trajectoire généralise la notion de curseur et donc d'itérateur : elle matérialise l'étape du calcul à un instant t et sert à définir des intervalles d'application d'algorithme. En effet, soit `A` un algorithme et `[x, y)` un intervalle composé de deux « positions », une de départ `x` et une d'arrivée `y`. `x` matérialise les conditions de départ de `A` et `y` sa condition d'arrêt. Le calcul s'arrête lorsque, par incrémentations successives de `x`, la condition `(x == y)` est vérifiée. On peut donc construire une bijection entre les positions de l'itérateur courant et les différentes étapes de l'algorithme : à toute étape de `A` on peut associer une unique position dans la séquence, et chaque position dans la séquence suffit pour retrouver l'unique étape de `A` qui lui correspond. On peut donc naturellement imaginer de généraliser cette notion aux algorithmes sur les automates en utilisant des curseurs sur des transitions matérialisant les intervalles. Les avantages de cette technique sont non-négligeables : couplage minimum entre l'algorithme et structure de données, masquage des données, application de l'algorithme sur tout ou partie de la séquence permettant une plus grande liberté de réutilisation.

Malheureusement, de simples positions dans l'automate ne permettent pas de construire la bijection décrite plus haut : une transition seule ne peut suffire à désigner sans ambiguïté une étape d'un parcours comme nous allons le voir.

Imaginons que nous implémentions un algorithme affichant à l'écran l'ensemble des mots reconnus par un automate en prenant garde de ne pas l'appliquer à un automate cyclique auquel cas le langage est infini et notre algorithme ne s'arrêtera pas. En revanche, nous voulons

pouvoir l'appliquer à des automates dont le graphe orienté possède des états convergeants, c'est-à-dire des états dont le nombre de transitions entrantes dépasse 1 comme l'état 6 du DAG (Directed Acyclic Graph) de la figure 4.5. Un DAG est un automate dont le graphe sous-jacent ne possède pas de cycle.

Imaginons maintenant que nous définissions nos intervalles d'application d'algorithme avec

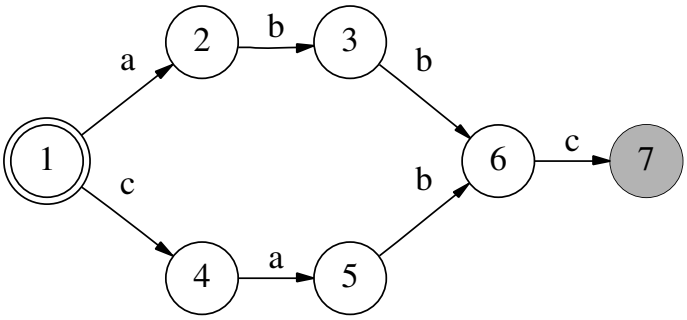

FIG. 4.5 – Un DAG (Directed Acyclic Graph)

des positions dans l'automate : nous pourrions tout simplement utiliser un curseur sur la transition $(1, a, 2)$ comme condition initiale du parcours. Par contre, quelle position choisir pour la condition d'arrêt ? Comment construire avec de simples curseurs l'intervalle représentant l'ensemble de l'automate et l'idée que le parcours doit comprendre toutes ses transitions ? C'est la première difficulté qui nécessite l'introduction du concept de curseur de parcours.

Dans le cas classique, l'intervalle `[x,y)` comprenant l'ensemble d'une séquence a pour `x` un itérateur sur le premier élément et pour `y` un itérateur pointant *derrière* le dernier élément (voir figure 4.6), ce qui a l'avantage de définir une convention standard homogène et cohérente même dans le cas de la séquence vide et de simplifier l'écriture du code exprimant la condition d'arrêt d'une boucle. L'implémentation d'un tel itérateur ne pose généralement pas de

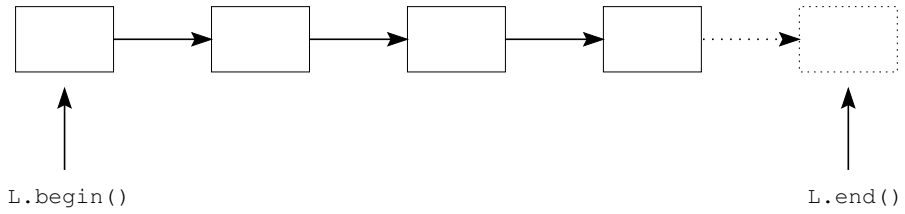

FIG. 4.6 – Un intervalle sur une liste chaînée L à quatre éléments

problème : un pointeur à `NULL` peut-être considéré comme désignant la position suivante du dernier élément d'une liste chaînée, ou bien l'adresse du dernier élément d'un tableau incrémentée de 1. Sur un automate, cette notion ne serait ni intuitive du point de vue conceptuel ni évidente du point de vue de l'implémentation. Quel sens pourrait-on donner à un « curseur pointant derrière la dernière transition de l'automate » ? Quelle que soit la sémantique choisie, elle ne ferait qu'obscurcir le concept et compliquer la mise en œuvre.

La deuxième difficulté énoncée plus haut, vient de ce qu'on ne peut construire de bijection entre les positions simples dans l'automate, c'est-à-dire des transitions, et les étapes de l'algorithme. L'extraction du langage requiert l'utilisation d'un algorithme de parcours en

profondeur décrit à la section 2.3.2. La figure 4.7 décrit l'état de la pile des transitions à chaque étape du parcours lors de l'extraction du langage du DAG de la figure 4.5. Chaque représentation de pile est sous-titrée par le numéro de l'étape et les numéros des actions entreprises à cette étape.

On voit qu'on peut associer à la position $(6, c, 7)$ quatre étapes de l'algorithme : les étapes

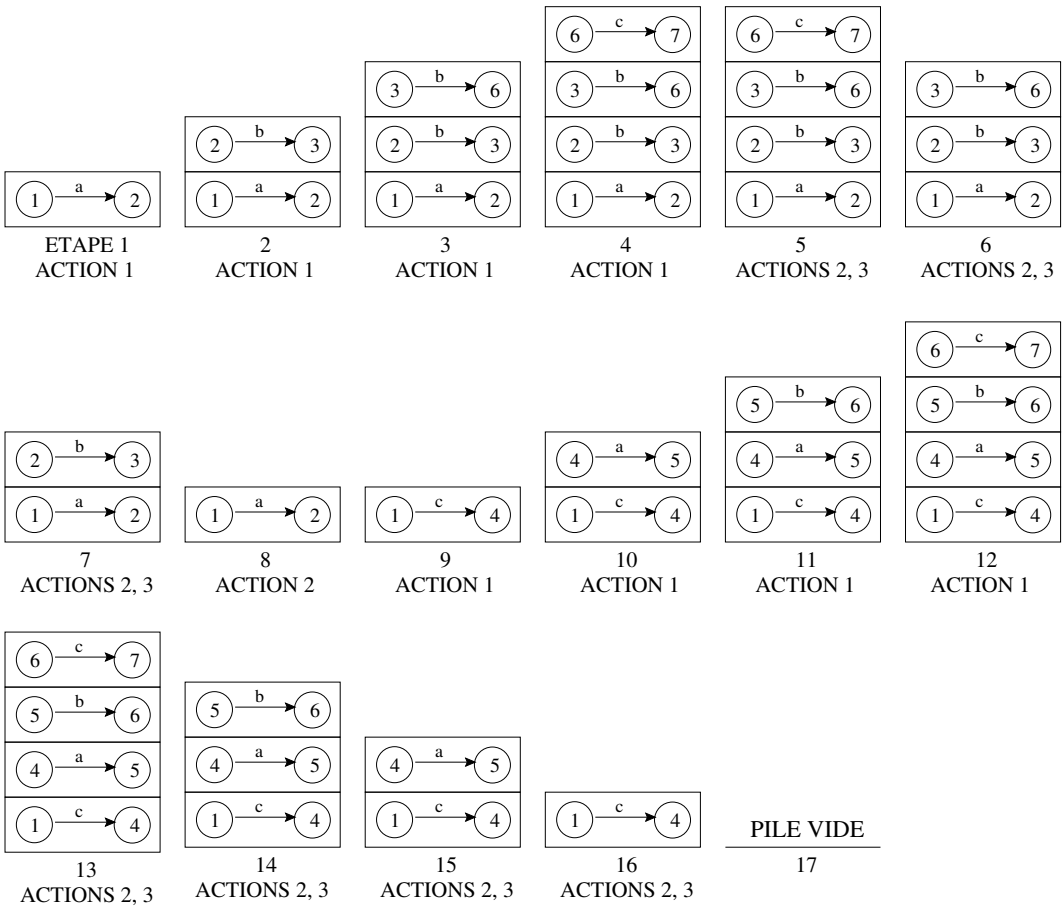

FIG. 4.7 – L'état de la pile à chaque étape du parcours en profondeur du DAG

4, 5, 12 et 13 où le sommet de la pile contient cette transition car c'est à ce moment là que le traitement de la transition s'effectue. Il est normal de retrouver chaque transition au moins deux fois en sommet de pile : une fois au cours de la phase descendante (étapes 4 et 12 avec action 1) et une fois au cours de la phase ascendante (étapes 5 et 13 avec actions 2 et 3). Si nous prenons comme convention qu'une transition désigne l'étape où elle apparaît lors de la descente car elle est censée matérialiser les conditions de départ du parcours, $(6, c, 7)$ désigne encore deux étapes : 4 et 12. Il est impossible pour l'algorithme de déterminer à laquelle de ces deux positions le parcours démarrera. On ne peut donc associer sans ambiguïté une position à une étape de l'algorithme.

Ceci amène à faire la distinction entre position dans l'automate et position dans l'algorithme : une position dans l'automate désigne une transition alors qu'une position dans l'algorithme désigne *un état de la pile*. Dans le cas particulier des itérateurs et des algorithmes séquentiels,

les deux notions de position sont confondues mais sur les automates ces deux concepts distincts sont matérialisés par deux modèles différents. La figure 4.7 « aplanit » la structure de l'automate et ramène le problème au cas classique de la séquence. La définition d'un intervalle sur une séquence implique deux itérateurs représentant à la fois deux positions dans cette séquence et deux étapes précises de l'algorithme car deux positions suffisent pour matérialiser les conditions de départ et d'arrêt du calcul. Par exemple, l'intervalle de la figure 4.8 s'étend sur la première moitié de la liste chaînée. Définir un intervalle dans la séquence des étapes de

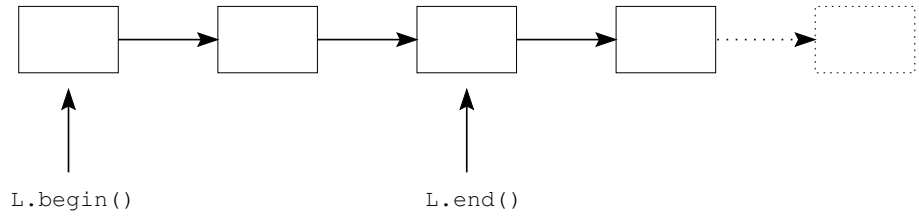

FIG. 4.8 – L'intervalle désignant la première moitié de la liste L

calcul, ici entre 1 et 17 signifie fournir deux piles, une pour les préconditions et une pour la condition d'arrêt. Muni de ces conventions, il est aisé d'appliquer un parcours soit :

- à l'ensemble des transitions de l'automate grâce à l'intervalle défini par les étapes 1 et 17.
- à un sous-automate, par exemple celui de la figure 4.9 grâce à l'intervalle défini par les étapes 1 et 9.

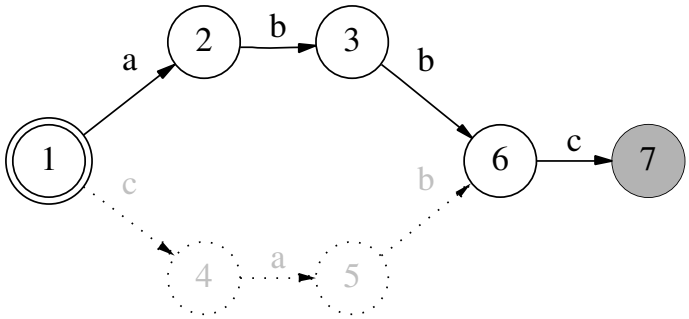

FIG. 4.9 – Le sous-automate défini par les piles de transitions 1 et 9

Le concept d'intervalle d'application d'un algorithme nécessite donc l'introduction de deux concepts : la pile et la file de curseurs sur lesquelles reposeront les curseurs de parcours en profondeur et en largeur, aussi définis au début de cette section comme des trajectoires (voir figure 4.10).

4.6 Le curseur pile

La gestion de la trajectoire du parcours en profondeur est la raison d'être du curseur pile. Ses fonctionnalités fournissent une base au curseur de parcours en matérialisant le chemin

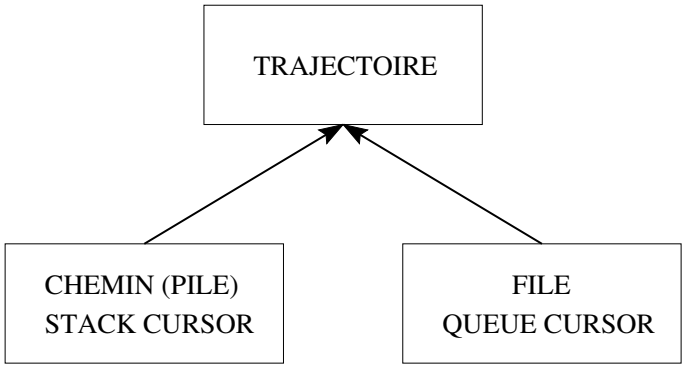

FIG. 4.10 – Le concept de trajectoire généralise la notion de chemin

associé à chaque étape de l'algorithme.

4.6.1 Définition

Un curseur pile est une pile de curseurs monodirectionnels. Il garde la trace des transitions traversées et permet le retour en arrière par dépilement. Son comportement est exactement le même que celui du monodirectionnel : les opérations réalisées s'appliquent au curseur de sommet de pile et une méthode supplémentaire autorise la « remontée » dans le chemin. Par contre, le concept sous-jacent est le chemin et non la simple transition.

4.6.2 Propriétés

Voir les propriétés du curseur monodirectionnel à la section 4.4.2 avec les nuances suivantes :

1. Un curseur pile est constructible par défaut. Il représente alors la pile vide et a une valeur singulière. La seule opération garantie et utile est donc la comparaison avec un autre curseur.
2. La possibilité d'affecter un curseur pile à un autre curseur pile paraît naturelle et cohérente avec les autres concepts de curseur. Cependant, on peut douter de l'efficacité et de l'utilité d'une telle opération, c'est pourquoi il n'est pas imposé à un curseur pile de vérifier les propriétés d'affectation énoncées à la section 4.3.2. De plus, affecter un état arbitraire à ce type de curseur pour le repositionner dessus casse la cohérence des données puisque le curseur peut alors ne plus représenter un chemin (q_i, σ_i, p_i) car on prend le risque, pour un certain i de ne plus vérifier $q_{i+1} = p_i$.
3. Un curseur pile est incrémentable lorsqu'il est en position d'avancer sur une transition définie (non puits) ou de reculer (dépiler).
4. Il est déréférençable si sa pile n'est pas vide et si la transition pointée n'est pas une transition puits.
5. Il définit une relation d'équivalence :

$$c \equiv c' \Leftrightarrow \text{ la pile de } c = \text{ la pile de } c'$$

Les deux piles sont équivalentes si elles ont la même taille (elle contiennent le même nombre d'éléments) et que chaque élément en position i d'une des deux piles est équivalent à son homologue en position i dans l'autre pile.

4.6.3 Comportement et interface

Un modèle de curseur pile fournit les mêmes fonctionnalités que le monodirectionnel. Les appels de méthode s'appliquent au sommet de la pile mais les deux méthodes `forward` empilent le curseur résultant de l'avancée sur une transition. La méthode `backward` dépile le curseur courant pour se repositionner sur l'étape précédente du parcours.
La table suivante résume les différences de fonctionnalités entre curseurs pile et monodirectionnels :

Méthode	**Expression**	**Sémantique**
avancer	`x.forward();`	Avance sur la transition courante et empile le curseur résultant
avancer par `a`	`x.forward(a);`	Avance sur la transition sortante étiquetée par `a` si elle existe, empile le curseur résultant et retourne `vrai`
reculer	`x.backward();`	Dépile le curseur courant. Retourne `faux` si la pile résultante est vide
comparaison	`(x == y)`	Compare les piles de `x` et `y`

4.6.4 Paramètres d'instanciation

Le patron de classe `stack_cursor` est paramétré par le type des curseurs monodirectionnels qu'il contient. Les fonctionnalités de gestion du container sont héritées de l'adaptateur STL `stack` :

```
template <class ForwardCursor>
class stack_cursor : protected stack<ForwardCursor>
{
  ...
};
```

4.7 Le curseur file

Le curseur file est le pendant pour les parcours en largeur du curseur pile.

4.7.1 Définition

Un curseur file est une file de curseurs monodirectionnels. Il stocke les transitions visitées dans une file et représente la trajectoire associée à chaque étape du parcours en largeur. Les opérations standards du curseur monodirectionnel s'effectue sur le dernier élément de la file.

4.7.2 Propriétés

Les propriétés du curseur file sont plus restrictives que celles des autres concepts de curseur :

1. Un curseur file est constructible par défaut. Il représente alors la file vide et a une valeur singulière. La seule opération garantie est la comparaison.
2. Un curseur file est incrémentable mais uniquement dans le cadre du parcours en largeur. Il n'est pas possible de forcer le curseur à suivre une transition étiquetée par une lettre précise.
3. Les propriétés d'affectation risque d'engendrer des incohérences dans les données et des baisses d'efficacité. C'est pourquoi il n'est pas possible d'affecter un autre curseur ou un état à un curseur file (voir la remarque sur le curseur pile à la section 4.6.2).

4.7.3 Comportement et interface

Un curseur file ne doit pas implémenter la méthode d'avancée avec lettre `forward(a)` car cela rendrait le parcours incohérent. De même l'utilisation de `find` doit être interdite. Par contre, sa méthode `forward()` renvoie une valeur booléenne

Méthode	**Expression**	**Sémantique**
transition suivante	`x.next_transition();`	Passe à la transition sortante suivante et enfile le curseur résultant
avancer	`x.forward();`	Extrait la tête de file, avance sur cette transition et enfile le curseur résultant. Retourne `faux` si l'opération est impossible (file vide)
comparaison	`(x == y)`	Compare les files de `x` et `y`

4.7.4 Paramètres d'instanciation

La structure interne du curseur file est quasi similaire à celle du curseur pile. Seule la nature du container change. Ici, on fait usage de la classe STL `queue` :

```
template <class ForwardCursor>
class queue_cursor : protected queue<ForwardCursor>
{
  ...
};
```

4.8 Le curseur de parcours en profondeur générique

4.8.1 Problématique

Le concept de curseur de parcours en profondeur (PEP) vient de la nécessité de rendre générique les algorithmes de parcours. En effet la mécanique du PEP est certainement l'une des composantes algorithmiques les plus utilisées, il était donc naturel de vouloir l'implémenter une fois pour toutes et de pouvoir la réutiliser facilement. Pour situer la problématique et les solutions utilisées jusqu'ici, nous présentons dans l'algorithme 4.1 une version itérative de l'algorithme 2.2 de la page 31. Cette version fait appel à la notion de pile, à une version plus algorithmique de la fonction δ_2 et au type abstrait liste sur lequel on définira trois opérations

d'accès.

Soit P une pile de transitions sur laquelle on définit trois opérations :

Empilement d'une transition	$P \downarrow (q, \sigma, p)$
Dépilement	$P \uparrow$
Accès à la transition du sommet	$P \rightarrow (q, \sigma, p)$

On considère δ_2 comme une fonction associant à un état la liste des triplets (q, σ, p) de ses transitions :

$$\begin{array}{rcl} \delta_2 : Q & \rightarrow & L \text{ l'ensemble des listes} \\ q & \mapsto & L((q, \sigma_1, p_1), ..., (q, \sigma_n, p_n)) \text{ avec } (q, \sigma_i, p_i) \in \Delta,\ 1 \leq i \leq n \end{array}$$

Soit e un élément de la liste L de taille n. La fonction *tête* extrait la tête de la liste, *dernier* le dernier élément et $suivant(e, L)$ renvoie l'élément suivant e dans la liste L :

$$tête(L) = e_1$$

$$dernier(L) = e_n$$

$$suivant(e_i, L) = e_{i+1}$$

On retrouve dans cette version itérative les trois actions du parcours récursif. L'action 1 concerne le traitement des transitions pendant la phase descendante, l'action 2 le traitement d'une transition lorsque tout le sous-automate de celle-ci a été traité et l'action 3 effectuée lorsque toute les transitions sortant de l'état source ont été visitées et que l'algorithme dépile. Notez qu'à la différence du parcours en profondeur récursif sur les états, cette version itérative s'applique sur les transitions.

L'unique méthode viable utilisée jusqu'ici pour rendre générique ce genre de parcours consiste à utiliser un patron de conception (*design pattern*) appelé *visiteur* qui permet de paramétrer des parcours sur des structures de graphes en regroupant dans une classe tout ou partie des méthodes implémentant les actions nécessaires au PEP à des étapes bien précises du calcul. C'est la version objet des fonctions à trous utilisées en programmation procédurale auxquelles on fournit un ensemble de pointeurs sur des fonctions. Ce patron largement répandu est notamment mis en œuvre dans la Generic Graph Component Library (GGCL), la Boost Graph Library (BGL) et la Graph Template Library (GTL). Par ailleurs des langages expérimentaux ont été mis au point pour la spécification des itérations sur des structures de graphe [49] [48] mais nous ne nous intéresserons qu'aux implémentations ne recourant pas à des outils externes au C++.

GTL met à la disposition du programmeur un objet algorithme DFS pour Depth-First Search paramétré par huit fonctions virtuelles [16]. Par héritage, l'utilisateur définit ces fonctions dans sa sous-classe. La méthode `run` suivante implémente le parcours générique :

```
int dfs::run (graph& G)
{
  node curr;
```

Algorithme 4.1 parcours en profondeur itératif des transitions

Entrée : $A(\Sigma, Q, i, F, \Delta)$
  $P \downarrow tête(\delta_2(i))$
  tant que P n'est pas vide **faire**
    $P \rightarrow (q, \sigma, p)$
    tant que p non visité **et** $\delta_2(p) \neq \emptyset$ **faire**
      $P \downarrow tête(\delta_2(p))$
      $P \rightarrow (q, \sigma, p)$
      $q \leftarrow$ visité
      action 1
    fin tant que
    $P \uparrow$
    si $(q, \sigma, p) \neq dernier(\delta_2(q))$ **alors**
      action 2
      $P \downarrow suivant((q, \sigma, p), \delta_2(q))$
    sinon
      action 3
    fin si
  fin tant que

```
  node dummy;
  dfs_number.init (G, 0);
  if (comp_number) comp_number->init (G);
  if (preds) preds->init (G, node());
  if (back_edges) used = new edge_map<int> (G, 0);
  init_handler (G);
  if (G.number_of_nodes() == 0) return GTL_OK;
  if (start == node()) start = G.choose_node();
  new_start_handler (G, start);
  dfs_sub (G, start, dummy);
  if (whole_graph && reached_nodes < G.number_of_nodes()) {
      forall_nodes (curr, G) {
          if (dfs_number[curr] == 0) {
              new_start_handler (G, curr);
              dfs_sub (G, curr, dummy);
          }
      }
  }
  if (back_edges) {
      delete used;
      used = 0;
  }
  end_handler(G);
  return GTL_OK;
}
```

Trois des huit méthodes de paramétrage apparaissent dans le code : `init_handler`, `end_handler` et `new_start_handler` sont les actions à effectuer au démarrage et à l'arrêt de l'algorithme. Les cinq autres :

- `entry_handler`
- `before_recursive_call_handler` (action 1)
- `after_recursive_call_handler` (action 2)
- `leave_handler` (action 3)
- `old_adj_node_handler`

sont appelées pendant l'exécution de `dfs_sub`, ce qui rend le code difficile à maintenir : le traitement est fractionné et réparti dans plusieurs fichiers et la compréhension de l'algorithme dans son ensemble n'en est que plus compliquée idem pour le débogage. La généricité à outrance obscurcit encore le programme avec des opérations inutiles dans un très grand nombre de cas sans parler des pertes d'efficacité. L'écriture de ces nombreux handlers nécessite le plus souvent de consulter la documentation et même le code source pour comprendre où et comment ils seront utilisés.
À vouloir concevoir un algorithme suffisamment général pour gérer tous les cas de figure, on se heurte au problème de la « surgénéricité » : un composant inefficace qui n'est adapté à aucun problème en particulier, lourd à utiliser, à comprendre et donc à maintenir. Ces fonctions à trous rendent malaisée la définition d'un nouveau parcours. Avec les curseurs il existe deux choix :

1. Utiliser un adaptateur qui change le comportement du curseur sous-jacent sans avoir à toucher à l'algorithme utilisateur ;
2. modifier la politique de parcours de l'algorithme sans changer le curseur.

Le premier choix permet une réutilisation par les curseurs, le deuxième permet de conserver une approche haut niveau, sans entrer dans les entrailles de la librairie et donc plus générique.

La solution consiste donc à developper une base minimale sur laquelle viendront se greffer les applications nouvelles plutôt que d'écrire un squelette de fonction censé prévoir tous les cas et tous les besoins. C'est pourquoi nous proposons de faire le contraire : à l'opposé d'un algorithme à trous où l'on insère ses blocs de code, un curseur permet d'insérer dans le code de l'utilisateur des fonctionnalités de parcours en profondeur.

4.8.2 Définition

Un curseur de PEP est un itérateur sur la séquence des transitions $(q_i, \sigma_i, p_i) \in Q \times \Sigma \times Q$ du PEP.

4.8.3 Propriétés

1. Un curseur de PEP itère sur les transitions dans l'ordre « profondeur d'abord ».
2. Il mémorise le chemin parcouru dans un curseur pile.
3. Il ne traverse pas de transition puits. Lors d'une arrivée sur une transition puits, le sommet est dépilé.
4. Il est constructible par défaut, auquel cas il représente la pile vide.

5. Il est déréférençable si sa pile n'est pas vide.
6. Il est incrémentable si sa pile n'est pas vide.

4.8.4 Comportement et interface

Un curseur de PEP est un itérateur : il « projette » la structure complexe de l'automate sur une séquence de transitions et nous ramène en terrain connu. Appliqué au DAG de la figure 4.5, le parcours engendre la séquence des sommets de pile de la figure 4.7. La méthode `forward` permet d'incrémenter le curseur et de passer à la transition suivante. Seulement, nous avons besoin d'une information supplémentaire qui distingue le curseur de l'itérateur. La nature séquentielle de l'accès aux données à travers un itérateur gomme totalement l'aspect récursif de l'algorithme 2.2 de la page 31. Dès lors, il nous faut un formalisme permettant de décider où placer le code des actions 1, 2 et 3 des lignes 1, 7 et 9. L'action 1 est effectuée avant l'appel récursif donc il concerne la phase descendante du calcul alors que les actions 2 et 3 sont situées après l'appel, ce qui les rattache à la phase ascendante. Le curseur, en plus de nous octroyer l'accès aux transitions doit nous renseigner sur la phase courante de l'algorithme, d'où le comportement de la méthode d'incrémentation : `forward` renvoie `faux` si le curseur « recule » (action 3 : dépilement). Dans les deux autres cas (action 1 : avancée sur une transition ou action 2 : positionnement sur la transition suivante avec `next_transition`) elle renvoie `vrai`.

La table suivante décrit l'interface complète d'un curseur de PEP. Les notations sont les mêmes que précédemment : `x` et `y` sont des curseurs de PEP pointant sur une transition (q, σ, p) et `s` est un curseur pile.

Méthode	**Expression**	**Sémantique**
état source	`x.src();`	retourne q
source final	`x.src_final();`	retourne `vrai` si $q \in F$
lettre	`x.letter();`	retourne σ
état but	`x.aim();`	retourne p
but final	`x.aim_final();`	retourne `vrai` si $p \in F$
avancer	`forward();`	passe à la transition suivante dans l'ordre du parcours en profondeur. Retourne `faux` si `x` dépile la transition (phase ascendante)
comparaison	`(x == y)`	Compare les piles de `x` et `y`
constructeur par défaut	`X x;`	`x` est initialisé avec la pile vide
constructeur	`X x(s);`	`x` est initialisé à la position désignée par la pile de `s`
constructeur de copie	`X x(y);`	Post-condition : `x` est une copie de `y`

Remarque : Le curseur de parcours en profondeur de possède pas de méthode `sink` car il ne traverse ni n'accède à aucune transition puits.

Pour illustrer ce comportement, nous implémentons à la figure 4.11 la version itérative du parcours de l'algorithme 4.1. Elle est paramétrée par les trois actions de l'algorithme

correspondant. Le problème des automates cycliques et du marquage des états visités est abordé à la section 4.9.

```
void parcours_en_profondeur(DepthFirstCursor first, DepthFirstCursor last)
{
  while (first != last) {
    do
      action_1();
    while (first.forward());    // push
    do
      action_3();
    while (! first.forward());  // pop
    action_2();
  }
}
```

FIG. 4.11 – L'implémentation du parcours en profondeur itératif

4.8.5 Paramètres d'instanciation

Un curseur de PEP est paramétré par le type de l'objet mémorisant la trajectoire et par le type de marqueur d'état. Par défaut, le marquage n'est pas effectué car il est alors fait usage d'un marqueur de type `always_false` qui, comme le nom l'indique, trompe le curseur de parcours en ne lui indiquant jamais qu'il pourrait éventuellement être déjà passé par le chemin emprunté et ce, sans surcoût à l'exécution :

```
template <class StackCursor, class Marker = always_false>
class dfirst_cursor
{
  ...
};
```

Pour parcourir un DAG ou un automate cyclique, l'utilisateur peut fournir un type de marqueur différent. Ce marqueur doit être un objet fonction conforme aux spécifications de la section 4.9.

4.8.6 Exemple

L'implémentation `langage` de la figure 4.12 extrait le langage reconnu d'un automate entre les deux « positions » `first` et `last`.

4.9 Automates cycliques et marquage des états

Les automates cycliques et les DAG (graphe orienté acyclique) imposent dans une moindre mesure de pouvoir se rendre compte efficacement au cours d'un parcours que les états atteints

```
void langage(DepthFirstCursor first, DepthFirstCursor last)
{
  vector<char> w;
  while (first != last) {              // jusqu'à la condition d'arrêt
    w.push_back(first.letter());       // empiler la lettre courante
    if (first.aim_final()) affiche(w); // si le but est final afficher mot
    while (! first.forward())          // tant qu'on remonte
      w.pop_back();                    // dépiler la dernière lettre du mot
  }
}
```

FIG. 4.12 – L'implémentation `langage`

ont déjà été visités et de pouvoir accéder aux résultats des calculs les concernant. L'algorithme doit s'arrêter et donner le résultat optimal en temps optimal : on ne doit stocker et calculer que le strict minimum dépendant de la structure de l'automate, mais maintenir efficacement en temps et en espace un ensemble de valeurs de type quelconque n'est pas une tâche triviale.

On peut distinguer trois niveaux de structure imposant des contraintes croissantes :

- La structure d'arbre. C'est la structure minimale en nombre de transitions pour lier les états de l'automate entre eux. Elle ne contient pas de cycle et tout état a au plus une transition entrante. Elle n'impose donc aucun traitement particulier et supporte des exécutions d'algorithmes en temps optimal. C'est le comportement par défaut des curseurs de parcours.
- La structure de graphe orienté acyclique. Dans un DAG il n'y a pas de cycle mais un état peut avoir plus d'une transition entrante comme l'état 6 de la figure 4.5. Dans ce cas, la nécessité de marquer les états dépend de la nature de l'algorithme.
- Les automates avec cycles. Ils contraignent l'algorithme à marquer les états d'une manière ou d'une autre et à s'arrêter lorsqu'il détecte un cycle.

La structure de l'automate impose donc une certaine discipline, mais elle n'est pas la seule. La nature des algorithmes et les contraintes d'optimalité et d'efficacité également. Pour les algorithmes acycliques, le marquage des états visités précédemment permet la détection de cycles et l'algorithme est rendu valide par un dépilement du curseur en situation de cycle. Pour les algorithmes cycliques, la politique de marquage peut être modifiée et étendue en fonction des besoins puisque c'est un des deux paramètres de fonctionnement des curseurs de parcours.

Prenons le cas du DAG et appliquons l'algorithme `langage` de la figure 4.12 à l'automate de la figure 4.5. Cet algorithme ne doit pas rebrousser chemin lorsque à l'étape 11 (figure 4.7) il s'engage sur la transition $(6, c, 7)$ déjà visitée à l'étape 4. L'automate reconnaît bien deux mots abbc, cabc et `langage` doit les mettre en évidence. En revanche, lors d'une copie de l'automate par exemple[4], l'algorithme doit être capable non seulement de détecter les états déjà visités mais d'accéder aux copies des états concernés. Dans le cas contraire, le résultat n'est pas la copie exacte et il n'est pas optimal. L'automate de la figure 4.13 est la copie du

[4]Voir algorithme `ccopy` à la section 5.6.4 pour plus de détails sur l'algorithme de copie d'automate.

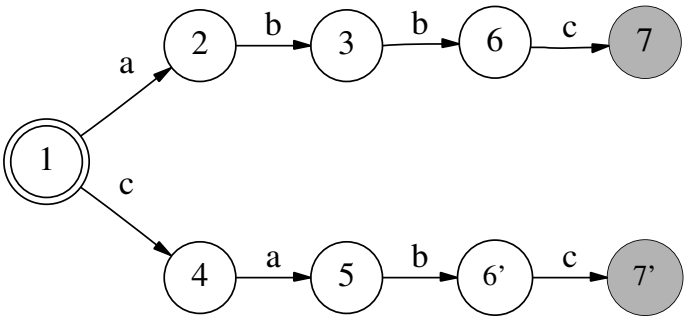

FIG. 4.13 – Copie inexacte du DAG

DAG sans marquage des états. Il reconnaît bien le même langage que l'original mais possède plus d'états et de transitions car la factorisation due à l'état convergeant 6 a disparu et les états $6'$ et $7'$ ne sont que des copies redondantes des états 6 et 7.
Dans le cas de la structure d'arbre, le marquage des états ne devrait pas être fait même s'il n'a aucune influence sur le résultat. C'est une opération superflue et pour des raisons d'efficacité elle ne doit pas être imposée.
Pour les automates cycliques, hormis le cas des algorithmes n'impliquant que des parcours simples comme l'implémentation `appartient` de la figure 4.3, le marquage est obligatoire pour que le calcul s'arrête. C'est pourquoi il doit être systématiquement utilisé.

Il ressort de cette analyse que l'utilisation ou non d'un mécanisme de marquage des états dépend de deux choses à la fois : la structure de l'automate et la nature de l'algorithme. La première est conditionnée par l'enchaînement des transformations et effets de bords des opérations qui lui sont appliquées et la deuxième par les contraintes inhérentes des algorithmes. L'une est donc interne à l'automate et l'autre interne à l'algorithme ce qui impliquerait d'écrire plusieurs versions du code, une pour chaque structure d'automate, mais cela contredit la philosophie de la programmation générique. La solution se trouve dans la troisième partie de notre modèle, les curseurs. En effet, la décision d'utiliser les marqueurs ne peut être prise que de manière externe à l'algorithme et à l'automate. Elle sera donc prise par l'utilisateur car la responsabilité de sa mise en œuvre ne doit pas être donnée à l'algorithme pour éviter la multiplication des versions ni à l'automate pour les mêmes raisons. Les curseurs de parcours sont chargés d'implémenter le marquage des états et le comportement adéquat. Ils sont donc paramétrés par un mécanisme appelé *marqueur d'états* qu'on définit comme un objet fonction unaire $Q \to \{vrai, faux\}$ à état dont l'opérateur `()` à effet de bords sert à renseigner le curseur sur le statut de l'état passé en paramètre (visité / non encore visité). Après l'appel, l'état est considéré comme visité.

Pour un objet `x` modèle de marqueur d'états et `q` un objet représentant un état de l'automate, on a :

Expression	**Sémantique**	**Post-condition**	**Complexité**
`x(q);`	retourne `faux` s'il s'agit du premier appel de toute la durée de vie de `x` avec `q` pour argument	`q` est visité	au pire $\log(\|Q\|)$

On définit cinq modèles se conformant à ce concept avec leur caractéristiques propres (figure 4.14).

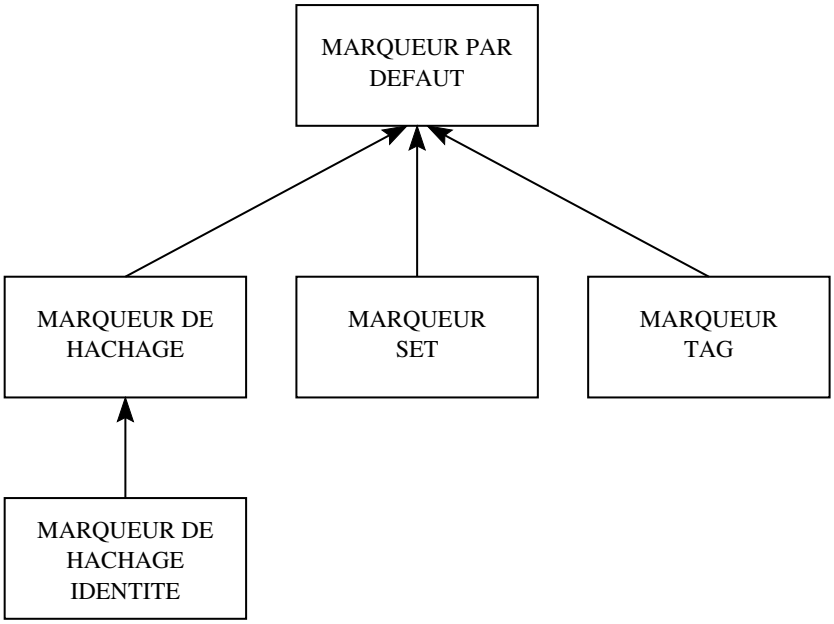

FIG. 4.14 – Hiérarchie des marqueurs d'états

Marqueur par défaut

C'est le modèle de base. Son comportement consiste à toujours renvoyer `faux` quelque soit l'état demandé. C'est le comportement minimaliste par défaut pour rester cohérent avec les autres systèmes de marquage et n'écrire qu'une version du curseur de parcours.

Marqueur de hachage

Utilise une table de hachage au sens de la programmation générique, c'est-à-dire un container associatif simple paramétré par une fonction de hachage. Cette fonction projette les éléments de Q (les clefs) vers les entiers positifs. La qualité du hachage dépend de la bonne répartition des valeurs entières (minimisation du nombre de collisions). Cette représentation est très intéressante lorsque la fonction de hachage est simple à écrire. Quand les types des états sont plus sophistiqués, la mise au point de la fonction peut s'avérer problématique. Par exemple sur un AFN, quelle fonction choisir pour associer un ensemble d'états à une valeur entière ? Lorsqu'il n'y a pas de bonne solution on peut se rabattre sur la définition d'une relation d'ordre sur les états et utiliser le marqueur set.

Marqueur de hachage identité

C'est le cas d'utilisation le plus simple et le plus direct du marqueur de hachage : lorsque les états de l'automate sont désignés par des entiers, la fonction identité $Q \to Q$ fournit le candidat idéal pour la fonction de hachage avec un temps de calcul nul et une répartition des valeurs parfaite.

Marqueur set

Un `set` est un container associatif simple représenté par un arbre rouge/noir ([47] pour un exposé détaillé sur la mis en œuvre des `sets`). L'accès logarithmique aux données repose sur une relation d'ordre définie sur les clefs, ici les états de l'automate. Si sa vitesse est souvent moins bonne que celle d'une table de hachage, il se révèle être un choix judicieux lorsque la définition d'une relation d'ordre est plus aisée qu'une fonction de hachage.

Marqueur tag

L'utilisation des tags pour marquer les états possède un avantage non négligeable : de tous les marqueurs, c'est la représentation la plus efficace en temps. On intègre à la structure de données de chaque état un drapeau booléen par exemple. Malheureusement, les inconvénients sont majeurs et il faut en faire une utilisation prudente et rigoureuse car l'information est interne à l'automate et il n'est pas possible de s'en débarrasser après une opération ponctuelle. À l'opposé, lorsque plusieurs passes sont nécessaires pour appliquer un traitement, la réinitialisation des drapeaux après un premier passage est problématique dans le cadre des algorithmes génériques développés jusqu'ici. Un calcul pouvant s'appliquer indifféremment à l'automate entier comme à un de ses sous-graphes, l'intégrité des données est difficilement garantie au lancement du deuxième algorithme, une partie des états ayant été visitée et pas l'autre. Pour des questions d'efficacité, il n'est pas raisonnable d'effectuer, préalablement à chaque algorithme, une passe sur les états en temps linéaire pour rétablir la valeur correcte des drapeaux. L'utilisation d'un tel système de marquage doit donc se faire dans un cadre précis, bien délimité et forcément peu contraignant.
La figure 4.15 résume les caractéristiques de chaque marqueur d'états.

Marqueur	**Temps d'accès**	**Contraintes**
Hachage	$\approx O(1)$	Fonction de hachage $Q \to N$
Hachage identité	$\approx O(1)$	États désignés par des entiers
Set	$\log(\|Q\|)$	Relation d'ordre partielle sur les états
Tag	$O(1)$	Interne à l'automate. Intégrité des données difficile à garantir.

FIG. 4.15 – Caractéristiques des différents modèles de marqueurs d'états

4.10 Conclusion

Les concepts présentés dans ce chapitre constituent une nouvelle optique pour les méthodes de conception de code manipulant des automates. En particulier, ils valident le principe fondamental d'une architecture à trois couches et préparent le terrain vers les techniques d'extension des fonctionnalités par adaptation du comportement de base : une bonne partie du traitement « glisse » de la couche algorithme vers la couche curseur ce qui permet de dépouiller les algorithmes de tous détails ou spécificités ne les concernant pas. Il en résulte une plus grande factorisation du code car ces algorithmes ne forment que le cœur du traitement, ce qui les rends adaptables de manière externe en leur fournissant des curseurs dont le comportement diffère de celui de base.

Le chapitre suivant décrit l'implémentation des adaptateurs de curseur et des algorithmes et la manière dont ils interagissent entre eux.

Chapitre 5

Les adaptateurs

La puissance des curseurs tient à la facilité avec laquelle il est possible de modifier et d'étendre leur comportement de manière efficace. Les adaptateurs, concept de programmation objets fondamental, centralisent avec élégance les mécanismes d'encapsulation, de combinaison et d'empilement des fonctionnalités.

5.1 Définitions

Comme leur nom l'indique, les adaptateurs adaptent l'interface et/ou le comportement des curseurs à des opérations plus complexes sans surcoût à l'exécution. Ce sont des clients du ou des curseurs sous-jacents, c'est-à-dire qu'ils les réutilisent, et qu'ils implémentent les fonctionnalités supplémentaires requises par une variante de l'algorithme de base.
On définit l'*arité* d'un adaptateur comme le nombre de curseurs qu'il contient. Un adaptateur unaire encapsule un curseur, un adaptateur binaire, deux curseurs, etc.

La figure 5.1 représente les interactions entre l'algorithme, le curseur adaptant et le(s) curseur(s) adapté(s). Ces adaptateurs appartiennent aussi au concept de curseur permettant

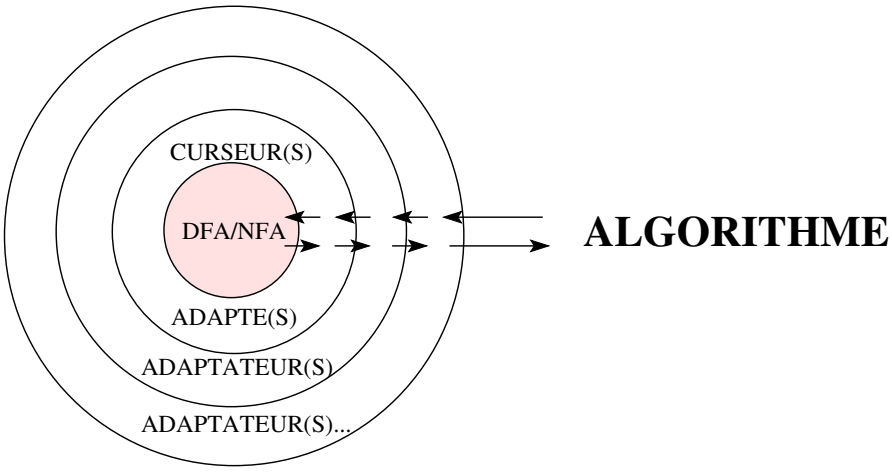

Fig. 5.1 – Interactions entre algorithme et adaptateur de curseur

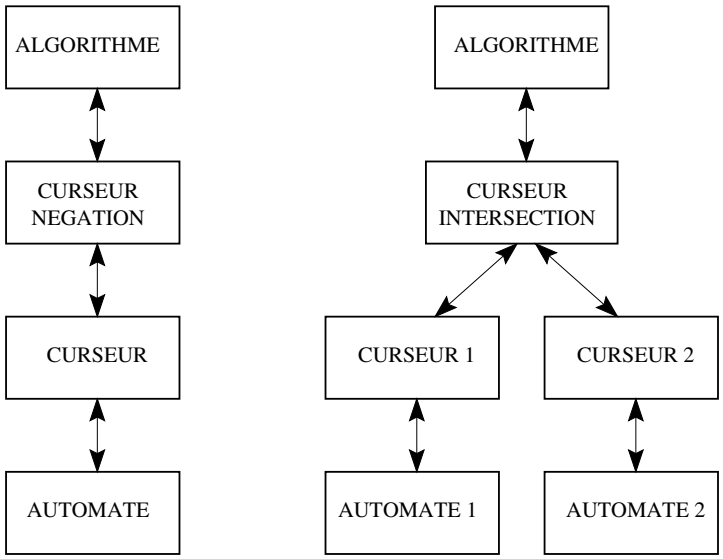

FIG. 5.2 – Les adaptateurs ensemblistes *complémentarité* et *intersection*

un polymorphisme statique et toutes les opérations additionnelles sont effectuées à la volée. Leur écriture impose principalement au programmeur de se poser trois questions : quelles sont les opérations à effectuer lors des appels des méthodes `first_transition`, `next_transition` et `forward` ? Ceci a pour effet de segmenter et modulariser agréablement l'implémentation d'un algorithme.

Les sections suivantes présentent un ensemble de cas où l'implémentation par adaptation du comportement de base est particulièrement puissante et efficace cependant le principe laisse libre cours à l'imagination du développeur et ouvre la porte à d'innombrables types de curseur.

5.2 Les opérations ensemblistes

Les opérations ensemblistes sur les automates définies à la section 2.2.6 page 30 sont mises en œuvre grâce à des adaptateurs de curseur monodirectionnel unaires (le complémentaire dans Σ^*) ou binaires (intersection, union, différence, différence symétrique, concaténation). La figure 5.2 décrit les interactions de l'adaptateur unaire du complémentaire `not_cursor` et de l'adaptateur binaire d'intersection `intersection_cursor` avec les autres composants. Nous allons nous intéresser à l'intersection pour illustrer la démarche de création d'un tel curseur.

Soient $A(\Sigma, Q, i, F, \Delta)$ et $A'(\Sigma, Q', i', F', \Delta')$ deux automates dont on veut calculer l'intersection $B(\Sigma, Q'', i'', F'', \Delta'')$ définie par :

$$B = (\Sigma, Q \times Q', (i, i'), F \times F', \Delta'')$$

B est un automate dont les états sont des couples pris dans l'ensemble du produit cartésien

$Q \times Q'$ et dont l'ensemble des transitions Δ'' est défini par la fonction de transition :

$$\delta''_1((q, q'), \sigma) = (\delta_1(q, \sigma), \delta'_1(q', \sigma)).$$

Les états terminaux sont les couples dont les états sont tous deux terminaux dans les automates de départ.
Soit `x` un curseur d'intersection évoluant sur l'automate B et encapsulant deux curseurs monodirectionnels `c1` et `c2` respectivement sur A et A'. D'après les définitions précédentes, `x` doit avoir le comportement suivant :

– L'état pointé par `x` est constitué d'une paire d'états des automates sous-jacents :

$$q'' = (q, q').$$

```
State src() const {
  return make_pair(c1.src(), c2.src());
}
```

– q'' est final si q et q' sont finaux :

```
bool src_final() const {
  return c1.src_final() && c2.src_final();
}
```

– `forward` implémente la fonction de transition δ''_1. Une transition étiquetée par a sortant de l'état q'' est définie si et seulement si il en existe une étiquetée par la même lettre sortant des états q et q'. Autrement dit, `x` peut avancer sur une transition si `c1` et `c2` le peuvent :

```
bool forward(int a) {
  return c1.forward(a) && c2.forward(a);
}
```

– Une transition sortant de q'' étiquetée par a est définie dans B si elle est définie pour q et q' :

```
bool exists(int a) const {
  return c1.exists(a) && c2.exists(a);
}
```

– L'état q'' est un état puits si au moins un des deux états q et q' est un état puits :

```
bool sink() const {
  return c1.sink() || c2.sink();
}
```

Cette interface concerne un modèle de curseur d'intersection simple. Elle est suffisante pour tester si un mot appartient à B. Le niveau de fonctionnalités supérieur qu'offre le curseur d'intersection monodirectionnel doit permettre de parcourir les transitions d'un état, $\delta_2((q, q'))$,

grâce aux méthodes `first_transition` et `next_transition`.
D'après la sixième propriété des curseurs de la section 4.4.2 que nous généralisons à l'intersection, les transitions sortant d'un état sont rangées en séquence dont la transition puits $((q, q'), \epsilon, 0)$ matérialise la position de fin :

$$\delta_2((q, q')) = ((\sigma_1, (p_1, p'_1)), ..., (\sigma_n, (p_n, p'_n)), (\epsilon, 0))$$

Ici 0 représente l'état puits de l'automate intersection, c'est-à-dire un couple d'états dont au moins une des deux composantes est nulle. 0 peut donc prendre les valeurs $(q, 0)$, $(0, q')$ ou $(0, 0)$. On construit cette séquence en choisissant les transitions communes aux deux automates :

$$(\sigma_i, (p_i, p'_i)) \in \delta_2((q, q')) \Leftrightarrow (\sigma_i, p_i) \in \delta_2(q) \text{ et } (\sigma_i, p'_i) \in \delta_2(q')$$

Comme tout adaptateur, le curseur réalisera l'intersection des deux séquences à la volée et de manière incrémentale en recherchant l'élément commun suivant l'élément courant lors de l'appel à `next_transition`.
Pour des raisons d'efficacité, nous allons imposer que ces transitions soient triées selon l'ordre croissant des lettres les étiquetant :

$$\text{pour } 1 \leq i, j \leq n,\ i < j \Leftrightarrow \sigma_i < \sigma_j$$

Cette contrainte supplémentaire nous permet d'écrire des méthodes `first_transition` et `next_transition` de complexité linéaire. Plus exactement, au cours d'une itération complète de $\delta_2((q, q'))$ le nombre de transitions comparées est borné par la somme des cardinaux des contextes droits des deux états sources : $|\vec{c}(q)| + |\vec{c}(q')|$. Sans cette propriété, le temps de parcours de la séquence intersection est quadratique.

– La méthode privée ci-dessous factorise les parties communes de `first_transition` et `next_transition`. Son rôle consiste à trouver la transition suivante $(\sigma_{i+1}, (p_{i+1}, p'_{i+1}))$ commune aux deux curseurs `c1` et `c2` positionnés sur (q, σ_i, p_i) et (q', σ_i, p'_i). Elle renvoie faux si $(\sigma_{i+1}, (p_{i+1}, p'_{i+1})) = (\epsilon, 0)$:

```
bool find_next()
{
  while(1) {
    if (c1.letter() < c2.letter()) {
      if (!c1.next_transition()) return false;
    }
    else
      if (c2.letter() < c1.letter()) {
        if (!c2.next_transition()) return false;
      }
      else    //  c1.letter() == c2.letter()
        return true;
  }
  return false;
}
```

– La méthode `first_transition` positionne le curseur `x` sur les deux premières transitions communes des curseurs `c1` et `c2` :

```
bool first_transition() {
  return c1.first_transition() && c2.first_transition() && find_next();
}
```

– Partant de l'élément courant $(\sigma_i, (p_i, p'_i))$, la méthode `next_transition` itère sur les deux séquences à la fois jusqu'à trouver la transition commune suivante $(\sigma_{i+1}, (p_{i+1}, p'_{i+1}))$:

```
bool next_transition() {
  return c1.next_transition() && c2.next_transition() && find_next();
}
```

– Enfin, les méthodes `forward` et `find` complètent l'interface :

```
void forward()
{
  c1.forward();
  c2.forward();
}

bool find(int a) {
  return c1.find(a) && c2.find(a);
}
```

Remarque Nous avons imposé le même alphabet Σ aux trois automates A, A', B et ce dans un but de simplification de l'exposé. En fait, cette limitation n'en est pas vraiment une car il est possible de modifier les propriétés des alphabets de manière externe, soit en utilisant des adaptateurs de curseur filtrant les caractères en entrée ou en sortie de l'interface (voir la section 5.5.3 sur les automates isomorphes), soit en redéfinissant les relations d'ordre et d'équivalence sur les éléments de Σ. Évidemment, ces deux possibilités ne sont pas mutuellement exclusives. La seconde consiste à fournir à l'adaptateur de curseur deux nouveaux opérateurs de comparaisons au sein de ce qu'on appelle un *trait* [46]. Un trait est une classe centralisant les méthodes implémentant les opérations standards propres à un type particulier. Le trait standard `char_traits` fournit entre autres une méthode de comparaison `eq` (equal) renvoyant vrai si les deux caractères passés en argument peuvent être considérés comme égaux. La méthode `lt` (lower than) renvoie vrai si le premier argument est inférieur au deuxième. Le comportement par défaut consiste à utiliser les opérateurs `==` et `<` sur les caractères mais la possibilité est laissée à l'utilisateur de l'adapter selon ses besoins. Par exemple, le trait suivant rend la comparaison des caractères insensible à la casse (« case-insensitive ») :

```
struct insensitive_traits
{
  static bool eq(int x, int y) {
    return tolower(x) == tolower(y);
```

```
  }

  static bool lt(int x, int y) {
    return tolower(x) < tolower(y);
  }
};
```

La fonction C standard `tolower` convertit un caractère en son équivalent en minuscule si nécessaire induisant une équivalence entre `a` et `A`, `b` et `B`, etc. En munissant l'adaptateur d'intersection de ces opérateurs :

```
intersection_cursor<fcursor1, fcursor2, insensitive_traits> c;
```

on rend le calcul plus « lâche » à condition bien sûr que les comparaisons de lettres étiquetant les transitions passe par le trait. Voici le code réel de la méthode `find_next` introduite plus haut :

```
bool find_next()
{
  while(1) {
    if (traits::lt(c1.letter(), c2.letter())) {
      if (!c1.next_transition()) return false;
    }
    else
      if (traits::lt(c2.letter(), c1.letter())) {
        if (!c2.next_transition()) return false;
      }
      else    //  c1.letter() == c2.letter()
        return true;
  }
  return false;
}
```

En lieu et place de l'opérateur `<` des appels à la méthode statique `lt` sont effectués. Comme d'habitude, une utilisation usuelle d'un composant doit se traduire par du code simple à écrire et lisible, c'est pourquoi par défaut le type du trait utilisera le comportement standard de la classe `char_traits<int>` ce qui, dans ce cas précis, nous ramène à la précédente version de la fonction.

Le polymorphisme permet n'importe quelle combinaison de manière immédiate comme décrit à la figure 5.3 où l'adaptateur implémente la différence symétrique de deux automates A_1 et A_2 définie par $(A_1 \cup A_2) \setminus (A_1 \cap A_2)$. L'utilisation d'un tel objet en particulier et des adaptateurs en général est décrite à la section 5.6 sur les algorithmes.

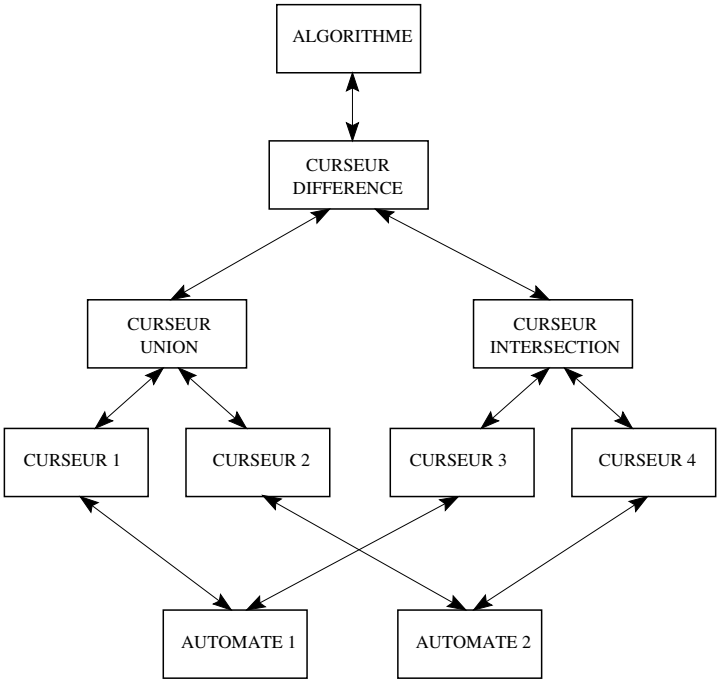

FIG. 5.3 – La différence symétrique de deux automates A_1 et A_2

5.3 Les transitions par défaut

Les mécanismes de transitions et d'états par défaut décrits à la section 2.2.5 s'implémentent très simplement avec des adaptateurs. Par exemple, un curseur `x` adaptant un curseur `y` sur un automate possédant un état par défaut `s` aura pour méthode `forward` :

```
bool forward(int a) {
  if (! y.forward(a))
    y = s;
  return true;
}
```

Cette fonction utilise la propriété affirmant qu'un état est assignable à un curseur : l'instruction `y = s` positionne le curseur encapsulé sur l'état puits `s` en cas d'échec, rendant l'automate sous-jacent virtuellement complet. C'est la seule différence de comportement avec le curseur standard.
En ce qui concerne l'automate avec transitions par défaut, il faut, lors de l'échec d'une avancée sur une transition non définie, que le curseur essaie de suivre la transition étiquetée par la lettre `default`, cette lettre étant fixée par l'utilisateur à la construction de l'adaptateur :

```
bool forward(int a) {
  if (y.find(a)) {
```

```
    y.forward();
    return true;
  }
  return y.forward(default);
}
```

Si le curseur est capable de se positionner sur la transition étiquetée par `a` il avance dessus sinon une avancée sur la transition par défaut est tentée.
Enfin, dans le cas des automates avec substituts, lorsque la transition n'est pas définie, le processus itère jusqu'à ce qu'un des substituts stockés dans les tags des états convienne. Ce genre de curseur est utile à l'algorithme de recherche de mots d'Aho et Corasick [1] :

```
bool forward(int a) {
  if (! y.find(a)) {
    y = src_tag();
    return !y.sink() && forward(a);
  }
  y.forward();
  return true;
}
```

Par convention un état ne possèdant pas de substitut a l'état nul pour substitut. Lorsque la transition demandée n'est pas définie, le curseur est placé sur le substitut contenu dans le tag de l'état source `src_tag()` et s'il est différent de l'état nul, l'opération est relancée. L'acyclisme du sous-graphe des substituts doit garantir l'arrêt de l'algorithme.

5.4 Construction paresseuse (Lazy Implementation)

Les adaptateurs de curseur vus jusqu'ici mettent en œuvre un calcul à la demande ou à la volée. À chaque appel à une méthode, une partie précise de l'algorithme est exécutée mais des opérations récurrentes et répétitives pénalisent la vitesse d'exécution puisqu'il est possible qu'on réapplique un algorithme à une portion des données déjà traitée. La construction paresseuse permet de mémoriser les résultats des calculs précédents afin d'accélérer le traitement (il s'agit tout simplement d'un système de cache). Elle a l'avantage par rapport à la construction globale du résultat de ne traiter que les parties de l'automate requises par l'algorithme au moment où il en a besoin ce qui permet entre autre de s'affranchir des problèmes de consommation de mémoire excessive.

L'efficacité de la construction paresseuse repose sur les propriétés suivantes de l'algorithme :

1. Les accès aux données résultant du calcul ne sont pas équiprobables. Plus les probabilités sont déséquilibrées plus le gain de temps est important ;
2. Le traitement final ne nécessite pas l'entière construction des résultats. En effet, lorsque la partie intéressante ne représente qu'un sous-ensemble réduit des données, l'effort de construction est disproportionné par rapport à l'utilisation qu'on va en faire. Il peut

même être impossible de construire l'automate résultant d'une opération pour des raisons de taille en mémoire ou de temps de calcul ;

3. Pour qu'une implémentation paresseuse soit possible, il faut que les règles d'application de l'algorithme soit exprimables de façon locale, c'est-à-dire que le calcul ne doit impliquer que l'état source de départ et les différents paramètres de l'algorithme (voir [37] pour une description plus détaillée sur des algorithmes « à la demande » sur les transducteurs).

La plupart des opérations de base sur les automates vérifient la troisième propriété. Les deux premières dépendent de l'utilisation qui est faite du résultat. Par exemple, la minimisation ne vérifie pas la troisième propriété car elle repose sur la relation d'équivalence de Nérode qui n'est pas locale à un état mais dépend des successeurs de cet état et du langage qu'il reconnaît. En revanche, les opérations rationnelles et ensemblistes gagnent à être appliquées de cette manière lorsque l'automate résultant n'est pas requis dans son intégralité. Nous allons prendre pour illustrer l'efficacité de cette technique l'algorithme de construction et de recherche de motifs décrit dans [2].

Partant d'une expression rationnelle le but est de construire le plus efficacement possible un automate déterministe reconnaissant le langage décrit par l'expression. La construction classique dite de Thompson passe par un automate non-déterministe asynchrone (avec epsilon-transitions) déterminisé par parcours en utilisant l'epsilon-clôture. L'inconvénient de cette technique est que l'automate non-déterministe croît relativement vite en terme de nombre d'états lorsque la complexité de l'expression augmente ce qui entraîne évidemment un accroissement du temps de calcul. De plus, il arrive que la déterminisation engendre un automate démesuré : dans le pire des cas, l'automate résultant possède 2^n états où n représente le nombre d'états de l'automate d'origine (tous les états de l'automate des parties de Q sont construits). Pour $n > 15$, la construction peut devenir problématique en terme d'espace mémoire et de temps de calcul. En outre, la réactivité de l'application est sérieusement mise à mal.
L'algorithme idéal consiste donc à ne pas construire d'automate non-déterministe mais à le simuler. En construisant l'arbre de l'expression, on ajoute aux nœuds une information désignant les nœuds suivants susceptibles d'être atteint après avoir lu une lettre particulière. En associant à chaque feuille l'ensemble de ses positions possibles dans le texte on est capable séquentiellement, en lisant le texte, de savoir pour une position p quelles sont les lettres autorisées à la position $p+1$. De cette manière, on se ramène à un temps de précalcul (construction de l'arbre) linéaire en le nombre de nœuds/feuilles de l'arbre. Cette structure de données est directement utilisable pour la recherche du motif mais revient à gérer un AFN ce qui est très inefficace.
La deuxième phase proposée par les auteurs consiste à partir de cet arbre « décoré » à construire l'automate déterministe. L'alternative paresseuse que nous proposons consiste à encapsuler l'arbre dans un curseur appelé `regexp_cursor`, c'est-à-dire à lui donner une interface standard masquant l'implémentation, puis à utiliser un adaptateur appelé `lazy_cursor` encapsulant ce curseur et un AFD. Lorsque l'ordre lui est donné d'avancer sur une transition étiquetée par une lettre du texte, il vérifie que la transition est bien définie dans l'automate. Si c'est le cas, il n'y a rien de spécial à faire et sa position est mise à jour. Dans le cas contraire, il utilise le curseur d'expression rationnelle pour s'informer de l'existence de la transition. Si elle existe, il l'ajoute à l'automate et met à jour sa position. Lors des accès ultérieurs, cette

transition sera définie et ne donnera donc lieu à aucun calcul spécifique. On ne construit donc que les transitions intéressantes, c'est-à-dire étiquetées par les lettres présentes dans le texte. De plus, les occurrences des lettres n'étant pas équiprobables, les parties de l'automate les plus fréquemment visitées sont très rapidement construites.
L'intérêt indéniable d'utiliser un curseur pour implémenter la construction paresseuse est qu'il n'est aucunement nécessaire d'écrire du code supplémentaire : muni de n'importe quel curseur vérifiant le concept standard, le `lazy_cursor` est capable de « cacher » les résultats des opérations effectuées pour une utilisation ultérieure. Son utilisation est des plus simples :

```
// Construction de l'arbre à partir de l'expression:
regexp_cursor e("(a|b)*cd");

// Déclaration de l'automate ayant pour tag des états
// de l'automate adapté :
DFA_matrix<e::State> cache;

// Recherche du motif sur l'entrée standard:
istream_iterator<char> first(cin), last;
if (appartient(first, last, lazy_c(e, cache)) == true)
  cout << "trouvé";
```

Remarquez que le DFA stocke dans ses tags, les états de l'automate de départ car il est nécessaire de maintenir la bijection entre les états du cache et les positions du curseur adapté : lorsqu'une transition n'est pas définie dans `cache` il faut repositionner `e` sur l'état correspondant et s'en servir pour vérifier l'existence de la transition. De plus, le curseur fainéant maintient en interne la fonction inverse grâce à une `map` des états de `e` vers ceux du cache.
Pour ce qui est de l'efficacité, les mesures ont montré un facteur d'accélération de 50 entre le temps de recherche du `regexp_cursor` seul et le temps de recherche avec le cache.

5.5 Automates virtuels

La grande force des curseurs tient à l'encapsulation des structures de données sous-jacentes. Un curseur cache à l'algorithme qui l'utilise la vraie nature des données et rend son application plus indépendante et donc plus générique. L'utilisation de curseurs simulant la structure d'un automate permet de surmonter nombres de problèmes (notamment d'explosion combinatoire) et de simplifier énormément certaines opérations. Suivent quelques exemples particulièrement bien adaptés à l'utilisation d'automates virtuels.

5.5.1 Le curseur Σ^*

La construction d'un automate reconnaissant le langage Σ^*, opération simple en soi à condition de considérer un alphabet fini de taille raisonnable, est rendue inutile par l'existence du curseur `sigma_star`. Ce curseur, pas contrariant, renvoie toujours `vrai` lors des opérations de positionnement et d'avancée sur une transition. Il ne contient quasiment aucune donnée et est donc optimal en espace et en temps.
L'avancée sur une transition est toujours possible puisque l'automate est complet. De plus, toutes les transitions ramènent à l'état initial, il n'est donc pas nécessaire de mémoriser l'état courant :

```
bool forward(int a) {
  return true;
}
```

L'automate ne possède qu'un état, il est initial et a pour identifiant `1` :

```
State src() const {
  return 1;
}
```

Le curseur ne peut être positionné que sur l'état initial qui est aussi terminal :

```
bool src_final() const {
  return true;
}
```

L'automate étant complet, $\delta_1(q)$ est toujours défini, quelque soit q. Il est donc impossible que le curseur se retrouve sur l'état puits :

```
bool sink() const {
  return false;
}
```

Le typage statique des patrons de classes et de fonctions autorise une optimisation complète du code à la compilation : les appels aux fonctions simples sont supprimés grâce à l'inlining ainsi que celles ne renvoyant que des valeurs constantes connues au moment de l'instanciation du code. Ceci a pour effet de réduire à néant le coût d'utilisation en temps et en espace d'un tel curseur.

5.5.2 L'automate des permutations

Prenons par exemple l'automate reconnaissant toutes les permutations du mot 012. La figure 5.4 montre l'arbre et l'automate minimal reconnaissant le langage {012, 021, 102, 120, 201, 210}.
La table de la figure 5.5 indique les tailles de l'arbre et de l'automate minimal reconnaissant les permutations d'un mot de longueur n pour $5 \leq n \leq 10$.
Le nombre d'états et de transitions de l'arbre est rédhibitoire. Au-delà de $n = 8$ l'espace mémoire consommé même par les représentations d'automates les plus économiques devient déraisonnable car le nombre d'états et de transitions croit factoriellement. Cependant, l'extrême redondance du langage reconnu entraine une diminution très appréciable à la minimisation,

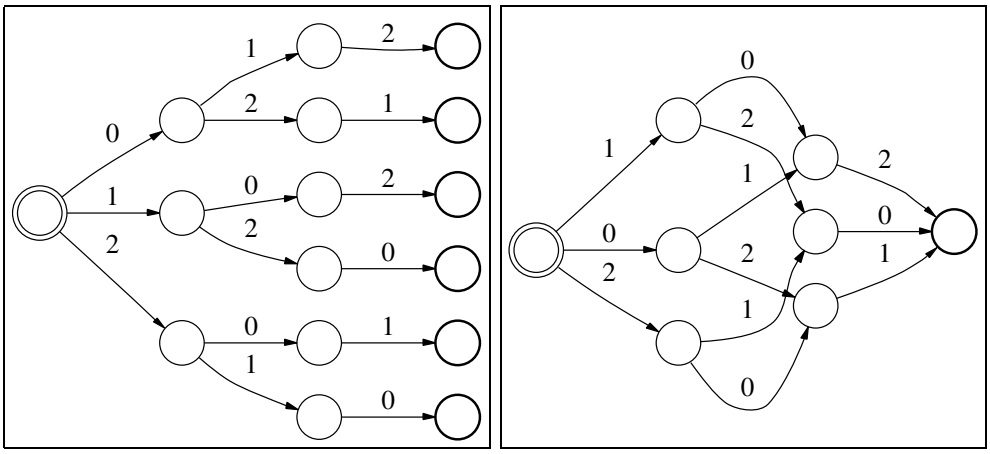

FIG. 5.4 – L'arbre et l'automate minimal reconnaissant les permutations du mot 012

	Arbre		**Automate minimal**	
Longueur	**États**	**Transitions**	**États**	**Transitions**
5	326	325	32	80
6	1957	1956	64	192
7	13700	13699	128	448
8	109601	109600	256	1024
9	986410	986409	512	2304
10	9864101	9864100	1024	5120

FIG. 5.5 – Tailles des automates reconnaissant les permutations d'un mot de longueur n

mais ce n'est que reculer pour mieux sauter. Clairement les nombres d'états et de transitions de l'automate minimal suivent une loi exponentielle en fonction de la longueur du mot n :

$$|Q| = 2^n$$

$$|\Delta| = \frac{n|Q|}{2} = \frac{n2^n}{2}$$

Cette complexité spatiale nettement meilleure que celle de l'arbre reste exponentielle ce qui n'est généralement pas viable d'un point de vue pratique. La solution intermédaire consiste à construire directement l'automate minimal sans passer par la minimisation de l'arbre.
Considérons l'automate A ayant pour alphabet Σ et pour ensemble d'états des parties de Σ : $A = (\Sigma, P(\Sigma), \emptyset, \{\Sigma\}, \Delta)$. L'état initial est l'ensemble vide et seul l'état Σ est final. L'ensemble des transitions Δ est défini par :

$$(q, \sigma, p) \in \Delta \Leftrightarrow \sigma \in \Sigma \setminus q \ \text{ et } \ p = q \cup \{\sigma\}$$

L'automate des permutations de $\{0, 1, 2\}$ apparait à la figure 5.6. Les états y sont représentés par les partie de Σ correspondantes. Cet automate reconnaît l'ensemble des permutations de Σ et est minimal. On en déduit l'algorithme de construction 5.1 en $O(2^{|\Sigma|})$.

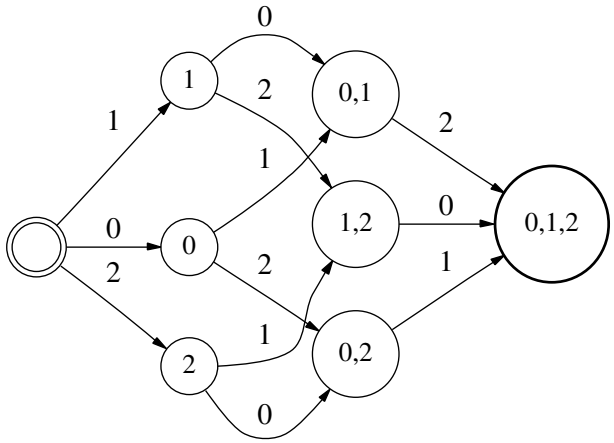

FIG. 5.6 – L'automate des permutations $A(\{0,1,2\}, P(\{0,1,2\}), \emptyset, \{\{0,1,2\}\}, \Delta)$

Algorithme 5.1 Permutations

Entrée : $A(\Sigma, Q, i, F, \Delta), q \in P(\Sigma)$
  $Q \leftarrow Q \cup q$ {Création d'un état}
  si $q = \emptyset$ **alors**
    $i \leftarrow q$ {Ajout de l'état initial}
  fin si
  si $q = \Sigma$ **alors**
    $F \leftarrow \{q\}$ {Ajout de l'état final}
  fin si
  pour tout $\sigma \in \Sigma \setminus q$ **faire**
    Permutations(A, $q \cup \{\sigma\}$)
    $\Delta \leftarrow \Delta \cup (q, \sigma, (q \cup \{\sigma\}))$ {Ajout d'une transition}
  fin pour

Avec un temps de calcul exponentiel, cet algorithme ne constitue qu'une solution bancale au problème. L'idéal reste d'utiliser un curseur qui nous permettra tout simplement de nous passer de l'automate en incorporant une variante incrémentale de l'algorithme de construction. Considéront l'implémentation des sous-ensembles suivante :

- Un sous-ensemble E_i d'un ensemble E est représenté par un vecteur de bits `e` de longueur $|E|$. Par soucis de simplification, on ne considèrera que des sous-ensembles de taille inférieure à 33 représentables par des entiers long de 32 bits.
- L'opérateur `<<` décale les bits d'un entier `e` de `n` positions vers la gauche et `>>` les décale vers la droite, i.e. `(e << 1) == (2*e)` et `(e >> 1) == (e/2)`. L'opérateur binaire `|` effectue un OU bit à bit des deux opérandes et `&` un ET bit à bit.
- On assigne un entier positif unique j à chaque élément $x \in E$ et $x \in E_i$ si et seulement si le $j^{ème}$ bit de l'entier le représentant est à 1. Il en découle pour un élément x d'indice `x` et deux sous-ensembles a et b que :

$$\begin{aligned} \{x\} &= \texttt{1 << x} \\ \emptyset &= \texttt{0} \\ a \cup b &= \texttt{a | b} \\ a \cap b &= \texttt{a \& b} \\ x \in a \Leftrightarrow (\{x\} \cap a) \neq \emptyset &= \texttt{(1 << x) \& a != 0} \end{aligned}$$

Soit `c` un curseur pointant sur un état `q` de l'automate des permutations d'un alphabet à 32 lettres. Les états sont des sous-ensembles de Σ et sont donc représentés par des entiers long non signés et l'opérateur préfixe `~` inverse les bits d'un entier. De plus, `c` maintient une variable booléenne `puits` vrai si `q` est l'état nul. Le code suivant implémente le comportement de `c`.
L'état courant est tout simplement `q`, le sous-ensemble de Σ :

```
State src() const {
  return q;
}
```

Un état est terminal si l'ensemble qui lui correspond contient toutes les lettres de l'alphabet donc s'il ne contient que des bits à 1 :

```
bool src_final() const {
  return q == ~0;
}
```

On ne peut avancer sur une transition étiquetée par a que si le sous-ensemble courant ne contient pas cette lettre. Si $\{a\} \cap q = \emptyset$ la transition est définie et on ajoute a à q par union, sinon on positionne à *vrai* la variable booléenne `puits` :

```
bool forward(int a) {
  if (1 << a & q == 0)
    q = q | 1 << a;
  else
    puits = true;
  return !puits;
}
```

Enfin, on interroge la variable `puits` pour savoir si `c` pointe sur l'état nul :

```
bool sink() const {
  return puits;
}
```

Chaque élément du sous-ensemble `q` de Σ est représenté par un bit à 1. L'état initial de l'automate étant l'ensemble vide il est représenté par `0` et l'état final est l'entier dont tous les bits sont à 1, soit `~0`.

```
permutation_cursor c;
c = 0;  // positionnement sur l'état initial
c = ~0; // positionnement sur l'état final
```

5.5.3 Automates isomorphes

Deux automates $A(\Sigma, Q, i, F, \Delta)$ et $A'(\Sigma, Q', i', F', \Delta')$ sont dits *isomorphes* s'il existe une bijection β entre les états de A et de A' :

$$\beta \ : \ Q \rightarrow Q'$$

vérifiant $\beta(i) = i'$, $\beta(F) = F'$ et $\beta(\delta_1(q, \sigma)) = \delta_1(\beta(q), \sigma))$. Autrement dit, l'automate A' est équivalent à A à une renumérotation près des états. D'un point de vue informatique, cela signifie donner une valeur différente à chaque état de A en changeant éventuellement le type des données représentant un état. L'adaptateur de curseur `state_filter_cursor` paramétré par les deux fonctions β et β^{-1} notées `f` et `g` modifie les comportements des méthodes d'accès à l'état source et de positionnement du curseur (opérateur d'affectation). Le curseur encapsulé est noté `x` :

```
State src() const {
  return f(x.src());
}

self& operator= (State q) {
  x = g(q);
  return *this;
}
```

Une variante de ce mécanisme consiste à définir une bijection entre les alphabets Σ et Σ' de deux automates :

$$f \ : \ \Sigma \rightarrow \Sigma'$$

L'adaptateur de curseur `filter_out_cursor` implémente l'injection $f_1 \ : \ \Sigma \rightarrow \Sigma'$ en filtrant les caractères lors de la lecture des étiquettes à travers le curseur adapté `x` :

```
int letter() const {
  return f1(x.letter());
}
```

Le `filter_in_cursor` implémente la fonction inverse $f_2 \ : \ \Sigma' \rightarrow \Sigma$ en projetant les caractères avant leur utilisation par le curseur adapté :

```
bool forward(int a) {
  return x.forward(f2(a));
}

bool exists(int a) const {
  return x.exists(f2(a));
}
```

Considérons la fonction g_w suivante associant à un entier i la $i^{ème}$ lettre d'un mot $w \in \Sigma^*$:

$$\begin{array}{rccc} g_w & : & N & \rightarrow & \Sigma \\ & & i & \mapsto & w_i \end{array}$$

Par exemple, à partir d'un alphabet $\{0, 1, 2, 3\}$, la fonction g_{abcd} définit un nouvel alphabet $\{a, b, c, d\}$ en envoyant 0 vers a, 1 vers b, 2 vers c et 3 vers d. Appliquée à l'automate des permutations sur un alphabet $\{0, 1, 2, 3\}$, elle engendre l'automate des permutations du mot *abcd* de la figure 5.7.

Les possibilités offertent par ces filtres sont multiples car ils permettent la définition d'al-

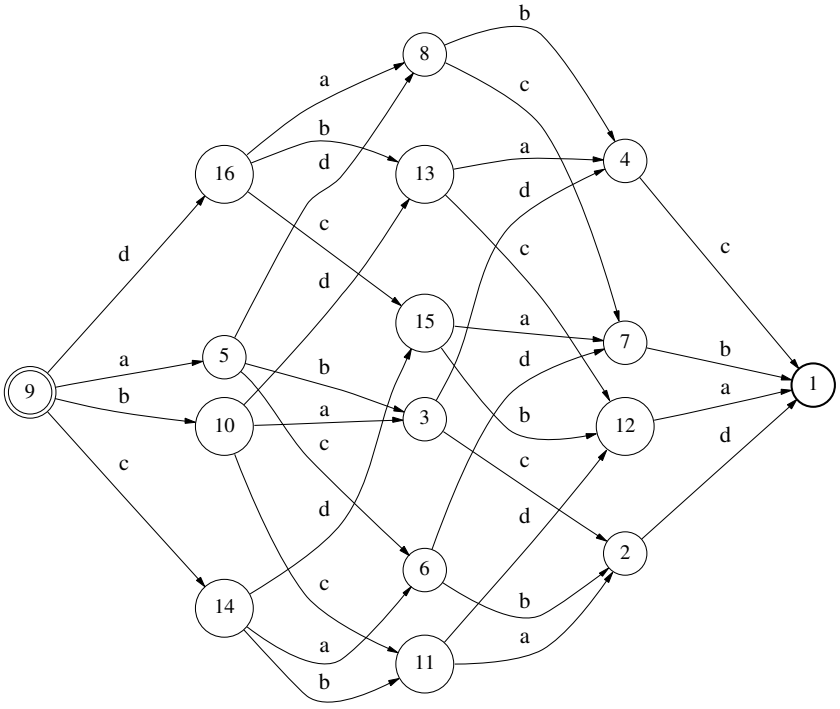

FIG. 5.7 – L'automate des permutations du mot abcd

phabets équivalents et de classes de caractères. Par exemple, une fonction f_2 projetant les caractères majuscules vers les minuscules permet de s'affranchir des problèmes de casse car du point de vue de l'automate, n'apparaîtront dans le texte que des lettres minuscules.

5.6 Les algorithmes et leur utilisation

ASTL ne contient que quatre algorithmes à proprement parler. Ils sont décrits individuellement dans les sections suivantes. Cette liste est déjà largement suffisante pour appliquer un grand nombre d'algorithmes de base car une partie importante du traitement est réalisée en amont et à la demande par les curseurs et autres adaptateurs. Les fonctions d'aide exposées ci-après servent à masquer la complexité des objets manipulés.

5.6.1 Les fonctions d'aide (helper functions)

L'empilement des fonctionnalités et des responsabilités ajouté à la structure composite des curseurs font qu'il est souvent fastidieux d'avoir à construire explicitement le type des objets que l'on désire manipuler. Un curseur de PEP contient un curseur pile contenant des curseurs monodirectionnels eux-mêmes constitués de curseurs. Le type de chacun de ces objets n'est pas imposé à l'utilisateur. Le comportement par défaut est défini, mais une liberté totale est laissée au programmeur pour adapter et modifier ces objets pourvu qu'ils soient toujours des modèles de curseur, c'est-à-dire qu'ils se conforment à l'interface et au comportement standards. L'existence de ces fonctions d'aide se justifie donc pour deux raisons, la première étant de donner un comportement par défaut à l'initialisation qui satisfasse les besoins habituels (les curseurs ne sont jamais initialisés à la construction) et la deuxième venant de la nécessité de soulager l'utilisateur lors de la déclaration de ses objets.

Le listing suivant déclare un curseur `c` sur un automate matriciel `A` sans tag et ayant pour alphabet des caractères sur 8 bits (ce sont les paramètres d'instanciation par défaut). Il teste l'appartenance du mot `ASTL` au langage reconnu par `A` :

```
const char mot[] = "ASTL";
cursor<DFA_matrix< > > c(A);

if (appartient(mot, mot + 4, c))  // erreur
  cout << "mot reconnu" << endl;
```

Le piège rencontré ici est l'oubli de la prise en compte du comportement par défaut des objets : un curseur ne possède pas par défaut de valeur valide, comme tout pointeur en C. Il doit perdre sa valeur singulière et être initialisé avant toute utilisation sinon le comportement est indéfini. `c` est bien construit sur l'automate `A` mais l'état sur lequel il pointe n'a pas été précisé. L'appel de la fonction a toutes les chances de générer une erreur d'accès à la mémoire et l'arrêt du processus. Pourquoi alors ne pas définir un comportement par défaut positionnant par exemple le curseur sur l'état initial de l'automate ? En fait, la multiplication des valeurs et comportements par défaut est une source non négligeable de bogues et autres désagréments : on peut s'attendre à un certain comportement qui se révèle être différent de ce que l'on escomptait, mal l'interpréter ou tout simplement oublier de l'implémenter. L'absence d'initialisation lorsqu'elle n'est pas précisée par le programmeur a le bon goût de simplifier les choses ; le curseur ne fait que ce qu'on lui demande et rien de plus. Pour ne pas surroger à cette règle quasi fondamentale du C et du C++ nous allons déléguer la gestion du comportement par défaut à un certain nombre de fonctions dites d'aide, mais avant d'en exposer

l'utilisation pratique, intéressons-nous au deuxième point qui milite pour leur introduction dans une librairie.

Voici par exemple le listing minimal nécessaire à la déclaration complète d'un curseur de PEP standard `x` sur un automate matriciel `A` :

```
depth_first_cursor<stack_cursor<forward_cursor<DFA_matrix<> > > > x(A);
```

Un deuxième construit par défaut servira de borne de fin d'intervalle pour notre algorithme :

```
depth_first_cursor<stack_cursor<forward_cursor<DFA_matrix<> > > > y;
```

Enfin, nous pouvons appliquer l'algorithme `langage` sur l'intervalle `[x,y)` qui affichera l'ensemble des mots reconnus sur la sortie standard `cout` :

```
langage(cout, x, y);
```

La lourdeur des déclarations de `x` et `y` est rédhibitoire pour le programmeur et la relecture complètement indigeste. Une des règles d'or de la programmation générique veut que dans le cadre d'une utilisation classique et fréquente d'un composant logiciel la procédure soit simple et lisible. On perd tout le bénéfice d'un composant très adaptable si son utilisation requiert autant d'efforts et de temps que sa récriture pour des besoins ponctuels. C'est pourquoi il faut se munir de fonctions d'aide ou *helper functions* chargées de construire les types et les objets. En outre, l'erreur commise ici est la même que pour le premier exemple car sauf précision, le curseur n'est positionné sur aucune transition en particulier et l'exécution de la fonction `langage` provoquera une erreur. L'absence d'initialisation par défaut impose d'initialiser `x` avec un curseur pile contenant un curseur monodirectionnel positionné sur la première transition sortant de l'état initial de `A` :

```
forward_cursor<DFA_matrix<> > fx(A, A.initial());
fx.first_transition();
stack_cursor<forward_cursor<DFA_matrix<> > > sx(fx);
depth_first_cursor<stack_cursor<forward_cursor<DFA_matrix<> > > > x(sx), y;
langage(cout, x, y);
```

Malheureusement, ce n'est pas suffisant. `A` pourrait être vide, dans ce cas l'état initial serait l'état nul donc l'appel à `first_transition` ne serait pas valide et provoquerait une erreur. Voici la version correcte du code :

```
forward_cursor<DFA_matrix<> > fx(A, A.initial());
if (! fx.sink()) {
```

```
 fx.first_transition();
 stack_cursor<forward_cursor<DFA_matrix<> > > sx(fx);
 depth_first_cursor<stack_cursor<forward_cursor<DFA_matrix<> > > > x(sx), y;
 langage(cout, x, y);
}
```

La quantité de code est disproportionnée par rapport à l'importance du résultat : appliquer un algorithme aussi basique que `langage` devrait être une opération basique.
La fonction `forwardc` se charge de construire à partir d'un automate un curseur monodirectionnel positionné sur l'état initial :

```
forward_cursor<DFA> forwardc(const DFA &A)
{
  return forward_cursor<DFA>(A, A.initial());
}
```

La fonction `dfirstc`, combinée avec la précédante permet de se passer complètement de la déclaration de `x` :

```
depth_first_cursor<stack_cursor<ForwardCursor> > dfirstc(ForwardCursor c)
{
  if (c.sink() || !c.first_transition())
    return depth_first_cursor<stack_cursor<ForwardCursor> >();
  else
    return depth_first_cursor<stack_cursor<ForwardCursor> >(c);
}
```

Un appel à `dfirstc(forwardc(A))` renverra un curseur de PEP initialisé à vide si l'automate ne contient pas d'état ou si l'état initial ne possède pas de transition sortante. Dans le cas contraire il sera positionné sur la première transition sortant de l'état initial. Reste le cas de la borne de fin d'intervalle `y`. Il est raisonnable de supposer que le cas le plus fréquent est l'application d'un algorithme à l'ensemble d'un automate et que les traitements limités à un sous-automate constituent un besoin plus ponctuel. C'est pourquoi les algorithmes utilisant les curseurs de parcours ont par défaut pour condition d'arrêt la pile vide :

```
void langage(ostream &out, DFirstCursor a, DFirstCursor b = DFirstCursor());
```

Ce qui réduit le code de départ à :

```
langage(cout, dfirstc(forwardc(A)));
```

On peut même réduire la formulation des appels au strict nécessaire grâce aux métafonctions :

```
langage(cout, dfirstc(A));
```

Ces deux fonctions d'aide `dfirstc` et `forwardc` font partie de l'ensemble de ces fonctions qui rend l'utilisation des algorithmes génériques simple et lisible donc viable. Il en existe au moins une pour chaque type concret de curseur et d'adaptateur. Leur mise au point ne devrait pas être négligée car elles se révèlent indispensables ; en déduisant les types à construire à partir des paramètres d'appel, elles font travailler le compilateur plutôt que le programmeur ce qui est un avantage indéniable.
Le code de la fonction `dfirstc` présenté ici est en fait une simplification de l'implémentation réelle qui fait appel à des techniques de métaprogrammation décrites au chapitre 6. Pour une description plus complète voir la section 6.2.3.

5.6.2 `appartient`

Cet algorithme introduit à la section 4.3.5 page 55 teste l'appartenance d'un mot au langage reconnu par un automate. Il fait appel au concept de curseur le plus faible et le polymorphisme des itérateurs définissant la séquence de caractères du mot élargit son champ d'application à toutes sortes de données séquentielles. Par exemple, l'implémentation suivante teste la présence d'un motif dans un fichier en utilisant les itérateurs de lecture sur flux `istream_iterator` :

```
DFA_matrix<> motif;
// ... construction de l'automate ...
ifstream fichier;
fichier.open("nom_de_fichier");
istream_iterator<char> first(fichier), last;
if (appartient(first, last, cursor(motif)))
  cout << "trouvé";
fichier.close();
```

Notez que l'application d'un tel algorithme à des combinaisons d'adaptateurs est immédiate grâce à l'utilisation des fonctions d'aide. Le code suivant recherche sur l'entrée standard `cin` une occurrence d'un mot appartenant à la différence symétrique de la figure 5.3 :

```
DFA_matrix<> A[2];
istream_iterator<char> first(cin), last;
if (appartient(first, last,
               differencec(unionc(forwardc(A[0]), forwardc(A[1])),
                           intersectionc(forwardc(A[0]), forwardc(A[1])))))
  cout << "trouvé";
```

5.6.3 langage

L'algorithme `langage` de la figure 4.12 page 71 extrait l'ensemble des mots reconnus par un automate par parcours en profondeur. Son utilisation a été décrite à la section 5.6.1. Sa version incrémentale, à savoir le `language_iterator` met en évidence le lien entre curseur et itérateur.

curseur + parcours en profondeur = itérateur

Un itérateur de langage est un itérateur sur la séquence des mots tels qu'ils sont extraits par parcours en profondeur : l'objet encapsule un curseur de PEP et à chaque incrémentation le parcours est effectué jusqu'à tomber sur un état terminal. Le mot courant est alors rendu disponible à l'utilisateur par déréférencement et sous forme d'objet de type `string`.
Le code suivant utilisant l'itérateur de langage et l'algorithme standard `copy` donne un résultat équivalent à l'appel de la fonction `langage` :

```
DFA A;
language_iterator<forward_cursor<DFA> > first(forwardc(A)), last;
copy(first, last, ostream_iterator<string>(cout));
```

L'itérateur `first` est construit à partir d'un curseur monodirectionnel renvoyé par la fonction d'aide `forwardc` et positionné implicitement sur la première transition sortant de l'état initial de `A`. Ce curseur servira à initialiser le curseur de PEP sous-jacent. La borne de fin d'itération `last` n'est pas initialisée, elle représente donc la pile vide. `first` et `last` possèdent des interfaces d'itérateur de lecture (input iterator), i.e. ils définissent un opérateur d'incrémentation `++` et un opérateur de déréférencement `*` renvoyant un mot de type `string`. Ils sont donc conformes aux exigences de l'algorithme `copy` qui, en recopiant la séquence des mots `[first, last)` sur la sortie standard `cout`, réalise l'affichage du langage reconnu par `A`.

5.6.4 ccopy et clone

`clone` copie un automate au cours de la phase descendante du parcours en profondeur effectuant une copie conforme et `ccopy`, pour cursor copy, copie un automate au cours de la phase ascendante du parcours en profondeur. De même que pour `clone`, l'automate à copier est défini par un intervalle constitué d'une paire de curseurs de PEP. Par défaut, la valeur du curseur de fin d'intervalle est la pile vide et en premier argument d'appel l'utilisateur spécifie l'automate destination. À la différence de `clone`, `ccopy` ne copie pas les chemins menant à des états puits non finaux. Par exemple, le code suivant convertit un automate matriciel `A` en un automate `B` de type `DFA_map` en éliminant les éventuels chemins inutiles :

```
DFA_matrix<> A;
// ... construction de A ...
DFA_map<> B;
DFA_map<>::State i = ccopy(B, dfirstc(A));
B.initial(i);
```

`ccopy` renvoie l'état initial du parcours, en l'occurrence l'identifiant de la copie de l'état initial de `A`.

Ces deux algorithmes de copie sont fondamentaux car hautement réutilisés, comme le montrent les exemples de la figure 5.8 où les flèches entre les composants dénotent des relations de réutilisation.
La relation entre copie et adaptateurs représente l'action de construire le résultat d'une opé-

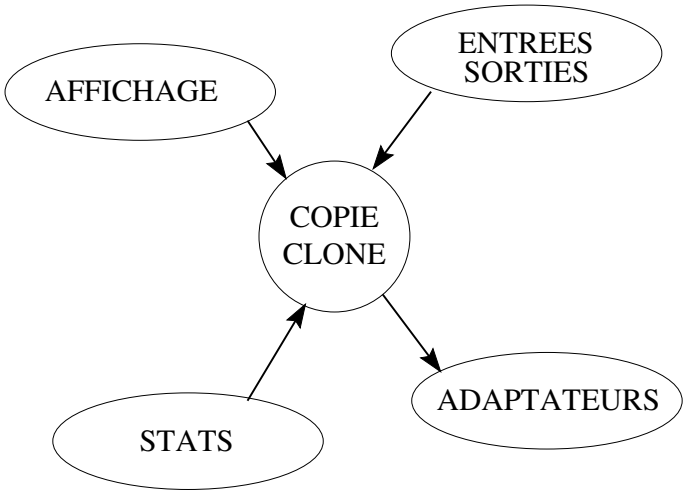

FIG. 5.8 – Les relations de réutilisation entre algorithmes

ration appliquée à un automate à travers un adaptateur de curseur comme on pourrait le faire pour obtenir `R` l'automate intersection de `A1` et `A2` :

```
DFA_matrix<> A1, A2, R;
// ... construction des automates A1 et A2 ...
ccopy(R, dfirstc(intersectionc(forwardc(A1), forwardc(A2))));
```

La fonction construit l'automate résultant `R` par appels à `R.new_state()` (création d'un nouvel état) et `R.set_trans(q,a,p)` (ajout d'une transition). En outre, elle maintient une bijection entre les états de l'automate d'origine et leur copie dans `R` sans laquelle la copie d'un DAG ou d'un automate cyclique deviendrait problématique (cf section 4.5). En écrivant la version « à la demande » d'un algorithme et en la combinant avec la fonction de copie on en obtient la version constructive.
Les interfaces des classes d'automates étant standardisées, on peut jouer sur le polymorphisme du premier argument pour implémenter tout un tas de fonctionnalités notamment les entrées/sorties graphiques ou sur flux et les fonctions de statistiques comme le calcul du nombre d'états ou de transitions.

5.6.4.1 Représentation graphique

Les schémas d'automates présents dans cet ouvrage ont été réalisé grâce à la commande `dot` du package GRAPHVIZ du laboratoire AT&T [20]. `dot` traite en entrée une description d'automate selon une grammaire fixée et produit en sortie un fichier postscript. Les attributs des éléments graphiques sont spécifiés par l'utilisateur (fontes, forme des états, étiquettes et

styles des transitions, etc.) et un certain nombre d'heuristiques paramétrables sont utilisées pour essayer de disposer le graphe de la façon la plus claire.

Le code suivant émet sur la sortie standard `cout` la représentation de l'automate des permutations de la figure 5.4 :

```
DFA_dot output(cout);
ccopy(output, dfirstc(permutationc(3)));
```

La classe `DFA_dot` utilise un type entier pour désigner les états qui seront numérotés dans l'ordre de leur création. Ceci a pour effet de définir un isomorphisme entre l'automate en mémoire et sa représentation graphique. Les états sont renumérotés dans l'ordre du parcours en profondeur ce qui leur donne une représentation graphique bien définie similaire aux entiers. L'interface d'un automate de ce type ne comprend que les trois méthodes imposées par `ccopy`. Il stocke en interne un compteur d'états `q` et un ensemble d'états terminaux `F` :

```
State new_state() {
  ++q;
  // déclaration d'un état:
  cout << q << " [label=" << q << "];";
  return q;
}

template <class Alphabet>
void set_trans(State q, const Alphabet& a, State p) {
  // déclaration d'une transition. Les états terminaux sont
  // négatifs, les autres positifs :
  cout << (F.find(q) ? -q : q) " -> " << (F.find(p) ? -p : p)
       << " [label=" << a << "];";
}

proxy final(State q) {
  return proxy(q, this);
}

class proxy
{
  DFA_stream *dfa;
  State q;

public:
  proxy& operator= (bool b) {
    if (b == true) dfa->F.insert(q);
    return *this;
  }
};
```

La méthode `final` renvoie un objet de type `proxy` possédant le comportement d'une référence sur un booléen. Lorsqu'on lui assigne la valeur `true`, l'état concerné est ajouté à l'ensemble des états terminaux `F` de l'automate grâce à la méthode `insert`.

5.6.4.2 Lecture et écriture sur flux

L'écriture sur flux utilise un type d'automate similaire à celui utilisé pour la représentation graphique :

```
DFA_stream output(cout);
ccopy(output, dfirstc(permutationc(3)));
```

Un objet de type `DFA_stream` se comporte quasiment comme un automate `DFA_dot`. À la création d'un état ou d'une transition, sa représentation ASCII est envoyée sur le flux de sortie, ici la sortie standard. En quelque sorte, on ne sauvegarde pas l'automate mais l'enchaînement des transitions le long du parcours ce qui se révèle être un outil très puissant puisqu'il permet à la relecture soit, évidemment, de reconstruire l'automate et de l'utiliser comme bon nous semble, soit d'appliquer directement un algorithme utilisant un parcours similaire sans avoir à le stocker en mémoire. Le code suivant affiche le langage reconnu par l'automate des permutations de l'exemple précédent sans le reconstruire :

```
langage(cout, istream_cursor(cin));
```

L'objet de type `istream_cursor` simule un curseur de parcours en profondeur en lisant la séquence des transitions sur l'entrée standard. L'algorithme `langage` requiert un parcours du type `clone`, c'est-à-dire la liste des transitions de la phase descendante. Il est donc nécessaire de s'assurer qu'on réutilise bien une séquence de type clone avec l'algorithme `clone`, de même pour `ccopy`.

5.6.4.3 Statistiques

Dans l'exemple suivant, on cherche à connaître le nombre d'états et de transitions de l'automate `A` réduit aux voisins du mot de référence `tomate` situés à une distance d'édition d'au plus 2 : le `neighbor_cursor` ne considère que la partie de `A` reconnaissant les mots transformables en `tomate` en moins de 3 insertions, suppressions ou substitutions :

```
DFA_matrix<> A;
DFA_stats S;
ccopy(S, dfirstc(neighborc(forwardc(A), "tomate", 2)));
cout << " |Q|     = " << S.state_count()
     << " |Delta| = " << S.trans_count();
```

La fonction `neighborc` construit un adaptateur encapsulant `forwardc(A)`, le mot de référence et la distance maximale. Le curseur « voisins » est un adaptateur de curseur pile, pour parcourir l'automate il reste donc à construire le curseur de PEP grâce à `dfirstc`. Quant à lui, l'objet `S` enregistre les nombres d'appels à `new_state` et `set_trans` et les rend accessibles à l'issue du parcours.

5.6.4.4 Word Count

Pour finir d'illustrer les algorithmes, voici le code de la fonction `wc` comptant le nombre de mots reconnu par un automate acyclique, c'est-à-dire le nombre d'états terminaux rencontrés durant la phase descendante du parcours en profondeur :

```
unsigned int wc(DepthFirstCursor first,
                DepthFirstCursor last = DepthFirstCursor())
{
  unsigned int i = 0;
  while (first != last) {
    do if (first.src_final()) ++i; // Phase descendante
    while (first.forward());

    while (!first.forward());      // Phase montante
  }
  return i;
}
```

5.7 Conclusion

Les abstractions présentées dans ce chapitre ont mené à des implémentations d'objets simples, efficaces et facilement combinables. Le nombre très restreint d'algorithmes (seulement deux sont indispensables : `ccopy` et `clone`) souligne l'intérêt majeur de l'effort d'unification et d'homogénéisation qui avait déjà pleinement profité aux traitements sur les séquences de STL. Notons au passage que pour terminer complètement la modélisation et atteindre le niveau de réutilisabilité maximale il reste à découpler totalement les algorithmes des automates pour ne les faire dialoguer qu'avec les curseurs car les esprits chagrins n'auront pas manqué de remarquer que les deux fonctions centrales de copie dépendent toujours d'un objet `DFA` modèle d'automate déterministe alors qu'une architecture à trois couches clairement identifiées impose une syntaxe d'appel à `ccopy` ressemblant plus à

```
DFA_matrix<> A, B;
ccopy(output_cursor(A), dfirstc(B));
```

où l'objet `output_cursor` implémente les fonctionnalités de construction d'un automate. Le découplage entre fonctions et structure de données est alors total ce qui permet plus de liberté, mais il n'existe toujours pas de concept générique d'accesseur modifiant une structure de données au cours d'une itération car les problèmes d'implémentation sont nombreux [18]. Pour l'instant, seules des solutions ad-hoc existent.

Si les avantages de ces nouveaux paradigmes de programmation sont maintenant mis en évidence, leurs faiblesses demeurent, particulièrement en ce qui concerne l'interface entre l'utilisateur et la librairie. En effet, si l'effort d'apprentissage des concepts mathématiques ou informatiques inhérents à un domaine spécifique, ici les automates, est indispensable à celui qui désire en tirer pleinement parti, l'information liée à des problèmes relevant purement de l'implémentation et de la machinerie de la librairie ne devrait en aucune manière concerner l'utilisateur même à un niveau abstrait. Il est par exemple regrettable d'avoir à assimiler de nouvelle structure de données lorsqu'on ne désire rien de plus qu'un objet automate vérifiant certaines propriétés algorithmiques basiques. L'utilisation d'un automate à accès rapide aux transitions en temps constant ne devrait même pas impliquer de savoir ce qu'est la représentation matricielle ou un vecteur STL. C'est dans cette optique que sont apparus ces dernières années de nouveaux concepts comme la programmation générative, les librairies actives et

la programmation intentionnelle qui prône l'utilisation d'abstractions logicielles basées sur des concepts beaucoup plus haut niveau que la programmation générique et qui présente un certain nombres d'aspects intéressants lorsqu'elle est couplée avec les notions précédemment citées.
C'est pourquoi, fort de cette expérience d'analyse et de conception sur les automates, tant sur le plan des containers que sur celui des accesseurs, nous nous proposons dans le chapitre suivant d'une part de décrire les résultats de la généralisation de nos travaux à la quasi totalité des machines à états finies, et d'autre part d'exposer les résultats d'expérimentations des méthodes de métaprogrammation en C++ appliquées au domaine des machines à états.

Chapitre 6

Métaprogrammation statique

Depuis l'invention du Fortran et des compilateurs, la réutilisation est au centre des préoccupations des programmeurs ; c'est un problème d'autant plus crucial qu'il recouvre les notions de portabilité et d'adaptabilité qui sont fondamentales. Dans un premier temps, les langages procéduraux ont réussi à définir un modèle abstrait de machine capable de gérer des types de données composites et de renforcer l'indépendance du code vis-à-vis de la machine. Puis le modèle objet a ajouté de la modularité avant que la programmation générique n'augmente encore le niveau d'abstraction en s'attaquant cette fois au problème de l'efficacité d'un code adaptable. Cependant, durant tout le processus qui consiste à produire un exécutable à partir d'un fichier source, il existe encore nombre d'opérations automatisables donc reléguables à la machine et dont dépend crucialement la qualité de l'application finale, par exemple le choix de la représentation en mémoire d'un objet en fonction de la situation ou des optimisations possibles. Il est aussi possible de simplifier l'écriture et la maintenance du code en fournissant une interface plus abstraite que celle des composants logiciels et en séparant les aspects comme l'allocation mémoire, la synchronisation des threads ou les contraintes temps réel. La programmation générative et les librairies actives s'attaquent à tous ces domaines où des progrès importants et prometteurs ont été effectués et la métaprogrammation est l'outil principal du C++ permettant de les mettre en œuvre.

Nous allons voir comment on peut mettre en place une architecture logicielle ou le compilateur joue un rôle actif dans la fabrication d'une version adaptée du code, augmentant ainsi l'efficacité du programmeur et de l'application sans adjonction de préprocesseurs ou d'outils externes. Nous appliquerons cette méthode à l'élaboration d'une métaclasse de machine à états plus générique que celle d'ASTL car capable de gérer de façon optimale tout type de machine dont la structure est basée sur celle d'un graphe.

6.1 Librairies actives et programmation générative

Avec la programmation générative et les librairies actives une nouvelle étape est franchie au-delà du modèle objet et de la programmation générique. Il s'agit de repenser le rôle que tient le compilateur dans le processus de développement allant de la conception de l'architecture logicielle à la création du programme exécutable par la machine [76]. La programmation générative remet en cause les interactions classiques entre librairie, compilateur et application. La compilation de ce genre de code génère des types et des algorithmes qui ne sont pas

définitivement précablés dans le code source, c'est l'utilisation qui induit les structures de données et les calculs nécessaires.

Une librairie active n'est pas une simple collection passive d'objets et d'algorithmes, elle contient du code sous forme de métaprogrammes et participe à la construction des composants logiciels par assemblage de sous-composants à partir des exigences de l'utilisateur. Pour mettre en œuvre ce type de librairie de manière rigoureuse et solide, il est nécessaire de se plier à une certaine discipline concernant quelques aspects de la programmation et de la conception qui sont décrits dans la section suivante.

6.1.1 Paradigmes de programmation émergeants

Même s'ils sont loin d'être tous nouveaux, ces paradigmes de programmation, principes ou méthodes commencent à être mieux connus et mieux implémentés. La programmation générative unifie tous ces domaines :

- *Domaine spécialisé.* Une librairie active est un module dédié à un domaine spécifique dont l'écriture est préférablement laissée à des spécialistes connaissant le domaine, les techniques de programmation et d'optimisation. Ces domaines sont généralement étroits et ont été étudiés exhaustivement : il ne s'agit pas de s'attaquer à des notions d'un genre nouveau ou expérimentales mais d'implémenter, une fois pour toutes, toutes les opérations concernant un domaine bien connu et maitrisé la plupart du temps depuis des années comme par exemple les algorithmes d'ordonnancement.

- *Séparation de l'espace des problèmes et des solutions.* L'interface abstraite de la librairie n'emprunte que très peu à l'informatique. Les concepts que manipule l'utilisateur se limitent presque uniquement à ceux du domaine considéré. Le lien entre les deux espaces est le *Configuration Knowledge* qui projette les spécifications vers des spécifications logicielles internes de manière automatique. Exemple : l'adaptation au processeur dans un système.

- *Programmation générique.* Le but est de construire des familles de composants logiciels orthogonales les unes aux autres donc développables séparément et qu'on assemble arbitrairement. De plus, l'efficacité étant un critère important, ces composants doivent comporter des optimisations spécifiques au domaine et à la situation (machine cible, système d'exploitation, types des données, algorithme). Ils sont généralement au cœur des librairies actives et ne sont manipulés que par la librairie elle-même ce qui pallie la principale faiblesse de la programmation générique qui ne possède pas de mécanisme de configuration et d'assemblage de composants automatiques. Cela épargne à l'utilisateur le travail d'assimilation de la documentation et des informations propres à la mise en œuvre de la librairie (structures de données, techniques d'optimisations) et lui permet de se concentrer sur son domaine.
  Le C++ met à disposition plusieurs outils pour y parvenir : la métaprogrammation, les patrons de classe et leur spécialisation, l'évaluation des constantes de compilation [75] et les patrons d'expression [71]. Le choix du C++ plutôt qu'un autre langage objet tient à la liberté totale qu'il offre à l'utilisateur d'architecturer son code comme bon lui semble et d'en contrôler tous les aspects aussi bas niveau soient-ils. En outre, l'efficacité des exécutables produits défie toute concurrence.

– *Séparation des préoccupations (« separation of concerns ») et des aspects* [25]. La programmation dite orientée aspects entend obtenir la même orthogonalité entre aspects que celle des composants logiciels génériques données - containers - algorithmes entre eux.
On appelle aspect les différentes propriétés d'une application. L'aspect de base est l'algorithme appliqué aux données et son résultat. Les aspects spécifiques concernent des domaines précis du traitement que l'on peut analyser et concevoir indépendamment les uns des autres comme la gestion des erreurs, l'allocation mémoire ou la localisation des objets dans un système distribué. Dans un programme classique, les opérations nécessaires à la gestion de ces aspects sont disséminées un peu partout dans le code et mêlées aux autres opérations (« *tangling code* »). Avec la programmation orientée aspects, les problèmes sont non seulement isolés à la conception mais ils le sont aussi dans l'implémentation et le code. Des modules centralisent les fonctionnalités propres à chaque aspect, il en résulte donc une maintenabilité accrue et une meilleure lisibilité.
Ces mécanismes sont généralement mis en place grâce à des métaclasses, des filtres incorporés par héritage, par encapsulation (technique dite du décorateur) ou par application du patron de conception *stratégie* aussi appelé *politique*[1]. Le C++ n'est pas le langage le plus adapté à ce genre de conception, toutefois, le paramétrage des patrons de classe par des types d'objets gérant un aspect précis de l'exécution a donné de bons résultats en programmation générique, notamment avec les allocateurs, et il s'avère souvent suffisant.

– *Métaprogrammation.* La métaprogrammation est l'outil de base servant à l'implémentation d'une librairie active en C++. Les patrons de classe permettent d'écrire des métafonctions exécutables par le compilateur et servant à l'aiguiller vers la meilleure combinaison de composants en fonction du contexte.

6.1.2 Principes généraux

DEMRAL (*Domain Engineering Method for Reusable Algorithmic Libraries* [11] pp. 273-291) définit une méthodologie complète et détaillée sur le processus de conception et d'écriture d'une librairie active. En voici une synthèse résumant ses objectifs, sa méthode et son architecture.

6.1.2.1 Objectifs

– Fournir à l'utilisateur une interface « intentionnelle » abstraite de très haut niveau. Son code spécifie les problèmes en termes de concepts propres au domaine et à un niveau de détail approprié. Les problèmes liés à l'implémentation des composants ne doivent pas interférer avec le code utilisateur. Il s'agit toujours de décomposer les problèmes en abstractions pertinentes ;

– Générer du code efficace en terme de vitesse et de consommation d'espace mémoire. La généricité ne doit pas s'accompagner d'une dégradation des performances, toutes les optimisations possibles doivent être envisagées et mises en œuvre et toute fonctionnalité

[1] En version originale, *strategy* et *policy*

inutile doit être écartée de l'exécutable résultant ;

– Permettre une adaptabilité et une extensibilité maximale. Éviter la duplication de code et l'enchevêtrement des aspects.

6.1.2.2 Méthodologie

On décompose l'élaboration de la librairie en trois phases, l'analyse du domaine, la conception puis la mise en œuvre :

1. L'analyse du domaine définit l'objet et l'étendue du champ d'application de la librairie. Elle identifie les caractéristiques (*features*) et contraintes propres à sa spécialité ainsi que les relations aux domaines connexes. Il s'agit entre autre d'extraire les abstractions, les concepts clés et les relations qu'ils entretiennent entre eux (similitudes, variations, dépendences). Cette première phase a pour but d'exprimer clairement les spécifications abstraites de la librairie sans aborder la conception logicielle.

2. La deuxième étape concerne la conception de l'architecture générale et du langage approprié nécessaire à l'expression des besoins de l'utilisateur (DSL de configuration, *configuration domain-specific language*). La grammaire de ce langage découle directement de l'analyse abstraite précédente et des décisions architecturales de la présente phase.

3. Enfin, la troisième partie consiste à mettre en œuvre le DSL, son parser et l'ensemble des composants logiciels utilisés. Cette collection de composants possède une interface appelée ICCL (*Implementation Components Configuration Language*). Le parser traduit l'expression du DSL en une expression de l'ICCL et l'assembleur de composant construit les classes et les algorithmes demandés.

6.1.2.3 Architecture

La figure 6.1 montre comment une configuration utilisateur abstraite exprimée grâce au DSL est parsée et traduite en configuration concrète exprimée dans le langage interne qu'est l'ICCL. À partir de cette expression l'assembleur de composants construit les types et éventuellement les algorithmes requis. L'ensemble parser - assembleur est appelé générateur.

6.1.3 Mise en œuvre

Cette section décrit les techniques de mise en œuvre du DSL, de l'ICCL et du générateur de manière générale et en C++ en particulier.

6.1.3.1 Domain-specific language (DSL)

Les *Domain-Specific Languages* sont des langages de programmation dédiés à un domaine d'application particulier, permettant à l'utilisateur de travailler directement avec les concepts du domaine concerné grâce à des abstractions de plus haut niveau que les composants logiciels, même génériques. En utilisant de tels langages, le programmeur, qui maîtrise le vocabulaire

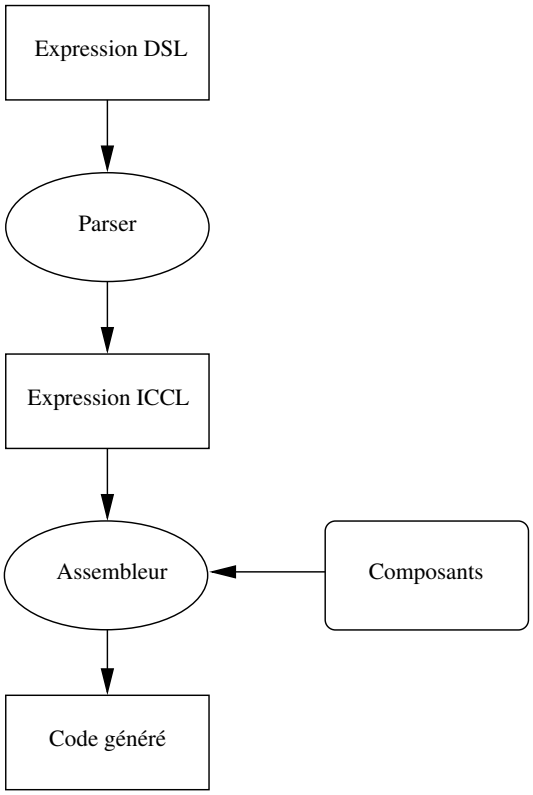

FIG. 6.1 – Structure d'une librairie active

et les spécificités du domaine, peut s'épargner l'apprentissage des techniques purement informatique, d'autant plus que la plupart du temps des mécanismes d'optimisation très pointus sont incorporés. Le spectre des domaines est déjà vaste : le calcul symbolique, numérique ou matriciel (Maple), les finances, l'électronique, la téléphonie et bien d'autres.
Les DSL se présentent sous forme d'applications indépendantes ou sont implémentés dans un langage de programmation déjà existant. L'utilisation du C++ permet par exemple de profiter des techniques de programmation générique qui leur font parfois défaut. La Generative Matrix Computation Library (GMCL, [11] pp. 293-427) définit un tel langage dans le domaine de la programmation matricielle et sur le C++. Elle va plus loin que la Matrix Template Library (MTL, [60]) qui ne fait qu'étendre STL au domaine des matrices. Notre travail sur les machines à états s'inspire fortement de GMCL et de MTL en étendant les concepts de programmation générative aux machines à états et en définissant en C++ le DSL correspondant.

Les DSL se caractérisent par un vocabulaire propre au domaine auquel ils s'appliquent et évidemment par une grammaire. Peu importe sa forme et sa mise en œuvre, le langage doit s'attacher à :

- Être le plus exhaustif et le plus pointu possible.
- Définir des abstractions proche du domaine concerné et le plus éloigné possible des considérations purement informatiques.

– Produire du code efficace en tenant compte des éventuelles optimisations propres au domaine et à la machine cible.

Le DSL spécifique aux matrices regroupent des concepts comme les « formes » de matrice (triangulaire, diagonale, symétrique), le type des éléments (entiers, nombres à virgule flottante), les matrices lignes ou colonnes, etc. Dans le domaine des machines à états on rencontre les notions d'orientation des transitions, d'information portée par les états ou les transitions, de temps d'accès aux transitions et de déterminisme. Le DSL permet aussi d'exprimer des directives générales sur le type d'optimisation désiré (en temps, en espace) et le degré de vérification des données manipulées à l'exécution (*bound checking*). En outre, il est chargé de fournir des valeurs par défaut pour chacun des paramètres non spécifiés par le programmeur.

En C++, le DSL ne constitue que la façade de la librairie. Les expressions sont construites grâce à des patrons de classe imbriqués représentant un arbre puis des métafonctions permettent sa compilation en une expression ICCL.

6.1.3.2 Implementation Component Configuration Language (ICCL)

Le DSL est l'interface abstraite entre la librairie et le monde extérieur. L'ICCL est en fait l'interface proprement dite ; elle permet la configuration et le paramétrage des composants logiciels concrets qui seront utilisés lors de la compilation. L'utilisateur n'a pas besoin d'en avoir connaissance car le générateur se fait l'intermédiaire entre ses spécifications et le code disponible dans la librairie. En C++, l'ICCL prend la forme d'une collection de patrons de classes et de fonctions que le compilateur combinera et instanciera en fonction des besoins.

6.1.3.3 Générateur

Le générateur au sens large comprend la partie parser ou générateur de configuration et l'assembleur de composants. Le générateur de configuration se charge d'interpréter les spécifications venant du DSL pour construire le composant. L'assembleur se sert des sous-composants pour construire les types à partir d'expressions formulées en ICCL par le générateur de configuration.
En C++, c'est le compilateur qui génère, à partir d'un type sous forme de patrons de classe imbriqués, un type concret de composant grâce à un métaprogramme. Il gère en outre les valeurs par défaut des spécifications qui n'auraient pas été précisées. Le générateur n'est donc rien d'autre qu'une grosse métafonction des types abstraits définissables par le DSL vers les types concrets disponibles dans l'ICCL.

La section suivante introduit les techniques de métaprogrammation en C++ qui sont à la base de l'élaboration des librairies actives.

6.2 Métaprogrammation statique en C++

Un métaprogramme est un programme qui en manipule un autre. Le meilleur exemple est celui des compilateurs : ils traduisent du code source d'un langage dans un autre. L'idée de la métaprogrammation est de mettre toujours plus à contribution le compilateur soit par le biais de compilateurs ouverts, paramétrables et extensibles, soit en utilisant les seules possibilités

qu'offre le langage lorsque le pouvoir d'expression est suffisamment important, c'est-à-dire lorsque les directives pouvant être adressées au compilateur sont de nature assez sophistiquée pour pouvoir diriger le processus de compilation vers telle ou telle implémentation. Bien qu'involontairement, le C++, grâce à la puissance des patrons de classe, permet d'écrire du code destiné au compilateur uniquement, ne générant donc pas de code exécutable à proprement parler. L'exercice consiste donc à utiliser au maximum les informations connues dès la compilation pour aiguiller le compilateur vers l'implémentation cible. Ces informations comprennent les constantes numériques[2] ainsi que les types et leurs propriétés statiques. Elles sont exploitées lors de l'instanciation des patrons de classes et des fonctions du métaprogramme qui en est exclusivement constitué.

6.2.1 Un exemple introductif

En guise d'introduction, voici le métaprogramme capable de calculer le n^{ème} terme de la suite de Fibonacci F définie par $F_0 = 0$, $F_1 = 1$ et par la formule récursive $F_n = F_{n-1} + F_{n-2}$ pour $n \geq 2$. Il y est fait usage de trois techniques fondamentales : le paramétrage des patrons par des constantes numériques, l'évaluation des constantes statiques par le compilateur et la spécialisation de patron.

```
template <int n>
struct Fibonacci
{
  static const int F = Fibonacci<n-1>::F + Fibonacci<n-2>::F;
};

template < >
struct Fibonacci<0>
{
  static const int F = 0;
};

template < >
struct Fibonacci<1>
{
  static const int F = 1;
};

void main() {
  int i = Fibonacci<27>::F;
}
```

Le premier patron défini une métafonction des entiers vers les entiers. À `n` il associe `F`, la valeur de la suite de Fibonnaci au rang n, ce qui illustre une utilisation intéressante des patrons de classe : le paramétrage par des constantes numériques. Il est en effet possible de paramétrer ces patrons soit par des types, soit par constantes qui vont intervenir dans le processus de compilation. Ici, l'instanciation du patron pour $n = 27$ requiert l'instanciation des deux

[2]Dans la pratique, les booléens entrent aussi dans cette catégorie.

autres patrons `Fibonacci<26>` et `Fibonacci<25>`. Le processus se poursuit jusqu'à tomber sur les cas d'arrêt de la récursion `Fibonacci<1>` et `Fibonacci<0>` qui sont deux spécialisations du patron `Fibonacci`. Dépourvus de paramètres puisque fournis explicitement, ils assurent l'arrêts de l'instanciation récursive de `Fibonacci<27>`. L'utilisation de constantes statiques autorise leur initialisation à la compilation plutôt qu'à la construction de l'objet au moment de l'exécution. De plus, le C++ autorise l'initialisation à partir d'expressions arithmétiques ou booléennes dont l'évaluation est faite par le compilateur. Bien évidemment, ces expressions ne peuvent impliquer que des calculs sur des constantes.

Finalement, l'appel à la métafonction lors de l'initialisation de `i` se traduira en fait par une simple affectation :

```
int i = 196418;
```

épargnant à l'application un calcul inutile. Notez que malgré la définition récursive de la métafonction, le calcul est réalisé par le compilateur en temps linéaire puisque l'instanciation pour une valeur de `n` fixée du patron garantit son existence pour toute l'unité de compilation. Ils ne sont donc instanciés qu'une fois quelque soit `n`, ce métaprogramme est donc doublement efficace.
Il est aussi à noter que le patron `Fibonacci` et ses 28 instanciations n'ont aucune existence propre dans le code exécutable généré ce qui est caractéristique d'un métaprogramme.

Les sections suivantes introduisent une poignée de ces métafonctions qui sont à la base des librairies actives écrites en C++.

6.2.2 Les métafonctions

Les trois métafonctions suivantes permettent de mettre en œuvre des métastructures de contrôle de flot d'exécution. À la différence de l'exemple précédent `Fibonacci`, elles servent à manipuler des types de données, c'est-à-dire qu'elle prennent en argument et/ou renvoie des types d'objet et non pas des entiers.
Par convention, le résultat `R` d'une fonction `F` est exporté par la structure `F` et accessible grâce à l'opérateur de portée `F::R`.

6.2.2.1 Méta IF

La métafonction suivante associe à une valeur booléenne le type `Then` ou le type `Else` selon qu'elle est vraie ou fausse :

```
template <bool condition, class Then, class Else>
struct IF
{
  typedef Then R;
};

template <class Then, class Else>
struct IF<false, Then, Else>
{
  typedef Else R;
```

```
};
```

La première structure est instanciée lorsque le booléen `condition` est vrai puisque la deuxième est une spécialisation pour le cas `condition == false`.

6.2.2.2 Méta EQUAL

La métafonction suivante teste l'égalité des deux types d'instanciation du patron :

```
template <class A, class B>
struct EQUAL
{
  static const bool R = false;
};

template <class A>
struct EQUAL<A, A>
{
  static const bool R = true;
};
```

Le deuxième patron est une spécialisation partielle du premier signifiant au compilateur qu'il devra préférer la deuxième structure à la première lorsque les deux types `A` et `B` sont égaux. Combiné au méta IF vu précédemment, il est possible de contrôler le travail d'instanciation du compilateur. Par exemple le patron de classe suivant choisit d'utiliser un container de type `string` standard lorsqu'il est instancié avec des caractères. Dans le cas contraire, un vecteur est utilisé :

```
template <class T>
struct which_container
{
  IF<EQUAL<T, char>::R, string, vector<T> >::R container;
};
```

Il est même possible de l'utiliser dans une expression booléenne classique :

```
template <class T>
void f(T x)
{
  if (EQUAL<T, int>::R == true) cout << "Paramètre entier";
  else cout << "Autre type de paramètre" << endl;
}
```

Durant la phase d'optimisation, le compilateur pourra supprimer le test et la partie du code qui ne sera jamais exécutée.

Remarque La syntaxe particulière liée à l'utilisation de patrons de classe impose malheureusement de considérer cette métafonction comme un opérateur préfixe dont le nom est un nom de patron et les deux opérandes les paramètres d'instanciation dudit patron ce qui, reconnaissons-le ne va pas dans le sens d'une clarification du code. Il est clair que, du point de vue du « sucre syntaxique » , le C++ n'est pas des plus adaptés à ce genre de gymnastique car il n'a jamais été prévu pour ce type de manipulations.

6.2.2.3 Méta SWITCH

La structure suivante, plus complexe que les précédentes, implémente l'équivalent d'un `switch` du C s'appliquant sur des types d'objet. Un switch prend un argument `A` de type entier en C alors que pour le méta switch `A` est un type. Ce type est comparé à une liste de types, chacun de ses éléments ayant un type associé `R`. Le premier type de cette liste égal à `A` fournit le résultat `R` et si aucun type ne correspond on utilise le résultat par défaut.
D'un point de vue purement syntaxique, un `switch` n'est que la récriture plus élégante d'une liste de `if`, `then`, `else`. Ici, l'aiguillage s'effectue de manière récursive en considérant les `case` subséquents comme des éléments d'une liste chaînée contenant trois types :

1. `MATCH` : le type auquel l'argument `A` du switch doit correspondre.
2. `R` : le type renvoyé par le switch si son argument correspond à `MATCH`.
3. `NEXT` : le type a utilisé dans le cas où `MATCH` ne correspond pas à l'argument du switch. `NEXT` est du type `CASE` ou `DEFAULT`.

La liste des différents cas possibles est donc construite de façon récursive avec la structure `DEFAULT` comme cas d'arrêt, qui ne contient donc pas de champ `MATCH` et `NEXT` :

```
template <class A, class B, class C>
struct CASE
{
  typedef A MATCH;
  typedef B R;
  typedef C NEXT;
};

template <class A>
struct DEFAULT
{
  typedef A R;
};
```

On exprimera donc l'ensemble des cas possibles par un type récursif que le compilateur « déroulera » jusqu'à rencontrer un type correspondant à son argument `A`. Voici par exemple l'expression du méta CASE associant les types entiers signés du C à leurs pendants non signés :

```
CASE<char, unsigned char,
CASE<short, unsigned short,
CASE<int, unsigned int,
DEFAULT<unsigned long> > > >
```

Reste à mettre en œuvre la structure de switch proprement dite. Elle s'instancie avec deux types ; `A` l'argument principal et une liste de `CASE` notée `FirstCase` car on manipule la liste de manière récursive en agissant uniquement sur la tête de liste.
Pour comparer `A` avec chacun des types de la liste, il est fait usage du méta-comparateur `EQUAL`. Si `A` est égal au type `MATCH` de la tête de liste alors on renvoie son type `R` sinon on renvoie le résultat du switch appliqué à l'élément suivant `NEXT`. La récursion s'arrête lorsque le dernier élément `DEFAULT` est atteint, il est donc fourni une spécialisation partielle du patron `SWITCH` assurant l'arrêt des appels :

```
template <class A, class FirstCase>
struct SWITCH
{
  typedef IF< EQUAL<A, typename FirstCase::MATCH>::R, typename FirstCase::R,
  typename SWITCH<A, typename FirstCase::NEXT>::R>::R R;
};

template <class A, class B>
struct SWITCH<A, DEFAULT<B> >
{
  typedef B R;
};
```

L'exemple suivant illustre l'utilisation d'une telle structure dans une librairie active. Il s'agit du concept de générateur de structure de données qui, partant de spécifications utilisateur (ses exigences), génère le type concret du container qui sera effectivement utilisé dans l'implémentation. Pour simplifier, ici les spécifications ne portent que sur deux points :

1. La complexité en temps d'un accès à un élément du container.
2. Le type des objets stockés dans le container.

Pour pouvoir exprimer ses besoins, l'utilisateur a à sa disposition trois types de structures selon le temps de calcul désiré :

```
struct TEMPS_LINEAIRE {};
struct TEMPS_LOGARITHMIQUE {};
struct TEMPS_CONSTANT {};
```

La méta-fonction `container_generator` s'instancie avec l'une des trois constantes précédentes et le type d'élément désiré. Le type réel est fourni sous la forme habituelle de type exporté `R` :

```
template <class Complexite, class T>
struct container_generator
{
  typedef SWITCH<Complexite,
            CASE<TEMPS_LINEAIRE, list<T>,
            CASE<TEMPS_LOGARITHMIQUE, set<T>,
          DEFAULT<vector<T> > > > >::R R;
};
```

Son utilisation est directe et très simple. Par exemple, le code suivant :

```
container_generator<TEMPS_LINEAIRE, int>::R c;
```

est strictement équivalent à :

```
list<int> c;
```

Cet exemple met le doigt sur l'aspect le plus important des librairies actives : il n'est pas nécessaire à l'utilisateur de connaître les composants réels, leur nom ou leurs paramètres d'instanciation pour pouvoir les utiliser. Les spécifications atteignent un niveau d'abstraction maximal sans perte d'efficacité à l'exécution puisqu'il s'agit d'un métaprogramme destiné au compilateur uniquement.

6.2.3 La fonction d'aide `dfirstc`

L'implémentation de la fonction d'aide `dfirstc` décrite à la section 5.6.1 du chapitre sur les adaptateurs de curseur illustre le processus d'automatisation qui mène à une utilisation plus aisée des composants d'une librairie. Son rôle se réduit à construire des curseurs de parcours en profondeur à partir d'un objet passé en argument, cet objet pouvant être indifféremment un modèle d'automate, de curseur monodirectionnel ou pile. Son utilisation épargne à l'utilisateur la déclaration des types, la construction du curseur et son initialisation tout en améliorant la clarté du code.

La fonction `dfirstc` doit pouvoir associer à chacun des concepts connus le type de curseur de PEP qui lui correspond. Soit `A` le type de l'objet passé en argument, `dfirstc` renvoie un objet appartenant à un des trois types suivants selon son concept :

`A` est un modèle d'automate → `dfirst_cursor<stack_cursor<forward_cursor<A> > >`
`A` est un modèle de curseur monodirectionnel → `dfirst_cursor<stack_cursor<A> >`
`A` est un modèle de curseur pile → `dfirst_cursor<A>`

Pour typer la valeur de retour de `dfirstc`, nous avons donc besoin d'une métafonction capable d'associer à un type concret `A` et son concept `C` un type de curseur de PEP `R`. Comme d'habitude, elle prendra la forme d'un patron de classe, nommé `DFIRST_TYPE`, paramétré par deux types `A` et `C`, exportant le type `R`. En définissant les trois classes suivantes on donne une existence syntaxique aux concepts :

```
struct DFA_concept { };
struct forward_cursor_concept { };
struct stack_cursor_concept { };
```

et on peut implémenter la métafonction en définissant un patron général que l'on spécialisera pour chaque concept :

```
template <class A, class C>
struct DFIRST_TYPE
{ };

template <class A>
struct DFIRST_TYPE<A, DFA_concept>
{
  typedef dfirst_cursor<stack_cursor<forward_cursor<A> > > R;
};

template <class A>
struct DFIRST_TYPE<A, forward_cursor_concept>
{
  typedef dfirst_cursor<stack_cursor<A> > R;
};

template <class A>
struct DFIRST_TYPE<A, stack_cursor_concept>
{
```

```
  typedef dfirst_cursor<A> R;
};
```

La première version n'a pas d'autre utilité que conduire à une erreur de compilation lors de son instanciation car elle n'exporte pas de type `R`, mais son instanciation n'aura lieu que s'il n'existe aucune spécialisation correspondant aux types `A` et `C` utilisés, c'est-à-dire lorsque l'objet considéré n'appartient à aucun des trois concepts, ce qui est bien une erreur d'écriture du code.
Après avoir codé `DFIRST_TYPE`, la difficulté est d'arriver à déduire le concept à partir du type concret. On peut par exemple imposer à l'objet passé en argument de `dfirstc` d'exporter son concept ce qui nous permettrait de typer sa valeur de retour. La fonction aurait alors pour signature :

```
template <class A>
DFIRST_TYPE<A, A::concept>::R dfirstc(const A &x);
```

Pour faciliter l'écriture d'une nouvelle classe, on enrichit les trois structures de concept vues plus haut d'une exportation de type dont l'utilisateur va pouvoir profiter par héritage :

```
struct DFA_concept
{
  typedef DFA_concept concept;
};

struct forward_cursor_concept
{
  typedef forward_cursor_concept concept;
};

struct stack_cursor_concept : public forward_cursor_concept
{
  typedef stack_cursor_concept concept;
};
```

Il suffit de faire hériter toute nouvelle classe du concept auquel elle se conforme :

```
class mon_forward_cursor : public forward_cursor_concept
{
  ...
};
```

En héritant de `forward_cursor_concept`, la classe `mon_forward_cursor` hérite aussi du type exporté `concept`.
Une fois le problème du typage de la valeur de retour de `dfirstc` résolu, on met en place un aiguillage guidant le compilateur vers le code à instancier selon le concept considéré par spécialisation d'un deuxième patron de classe :

```
template <class A>
DFIRST_TYPE<A, A::concept>::R dfirstc(const A &x) {
  return dfirstc_(A, A);
}
```

L'appel à `dfirstc_` met en jeu le polymorphisme par héritage. Il ne reste plus qu'à écrire les trois fonctions :

```
template <class A>
dfirst_cursor<stack_cursor<forward_cursor<A> > >
dfirstc_(const A &x, DFA_concept);

template <class A>
dfirst_cursor<stack_cursor<A> >
dfirstc_(const A &x, forward_cursor_concept);

template <class A>
dfirst_cursor<A>
dfirstc_(const A &x, stack_cursor_concept);
```

Ceci illustre le principal problème des patrons de classe et de fonction : l'impossibilité de contraindre le type utilisé pour l'instanciation. Le code du patron impose des exigences implicites au type mais il n'existe pas, au niveau du langage, de mécanisme permettant de les garantir, de les vérifier rigoureusement, ni même de les spécifier. On ne peut donc forcer l'instanciation d'un patron avec un type se conformant à un certain concept, d'où la nécessité de cette architecture relativement complexe mettant en jeu l'héritage, le polymorphisme, les patrons et leur spécialisation. Dans le cas contraire, il aurait suffit d'écrire trois fonctions `dfirstc`, une pour chaque concept et le choix de la bonne fonction au moment de l'appel aurait été entièrement laissé au compilateur.

6.3 Vers une librairie active de machines à états

Cette section décrit l'application de la méthode DEMRAL à l'élaboration d'une librairie active de machines à état capable de construire des types de containers et de curseurs à partir de spécifications abstraites.
Dans la suite de l'exposé, les termes machine et machines à états désigneront tous les modèles mathématiques basés sur les graphes, y compris ces derniers.

Le rôle du programmeur utilisant une librairie active consiste à construire ses spécifications selon la grammaire définie. Le rôle de la librairie consiste à :

- Parser les spécifications ;
- Assigner les valeurs par défaut des paramètres non spécifiés ;
- Assembler les composants suivant les spécifications ;
- Réaliser l'ensemble des optimisations possibles ;
- Mettre à disposition le composant final en encapsulant les éléments dans une interface appropriée.

Les spécifications de l'utilisateur expriment ses besoins quant à la nature de la machine à états désirée. Elles portent sur tous les aspects du domaine et doivent permettre la construction de tous les types de machines à états au sens large : graphes, arbres, automates, transducteurs [55] ainsi que leurs généralisations et leurs extensions comme les automates et les transducteurs

pondérés [37], le but étant d'étendre les concepts d'automate d'ASTL et de la faire apparaître comme un cas particulier d'une librairie plus générale.

6.3.1 Domaine

Avant d'aborder la grammaire du DSL, il est nécessaire de définir précisément l'étendue du domaine et d'introduire quelques concepts et notations.

Le but de notre travail est de développer une librairie de *machine à états* contenant des types de données abstraits et des algorithmes destinés à être intégrés dans des applications plus larges. Les principales applications d'une telle librairie sont :

- recherche de motifs
- reconnaissance de la parole
- indexation
- modélisation de grammaire
- compression de données
- analyses morphologigue, syntaxique et sémantique
- désambiguïsation de phrases
- dictionnaires
- correcteurs orthographiques
- modélisation du comportement
- réaccentuation de texte
- flexions, conjugaisons en langue naturelle
- traduction automatique
- apprentissage inductif de grammaire
- phonétisation et plus généralement l'implémentation de règle de reécriture.

Le modèle mathématique de machines représentables par des nœuds reliés entre eux par des arcs se décompose en trois principaux sous-modèles : graphe, automate, transducteur. On considèrera aussi comme machine à états tout ce qui est représentable par des états et des transitions, c'est-à-dire les machines de Moore et de Mealy, les automates et transducteurs pondérés, les réseaux de Pétri et neuronaux.

6.3.2 Concepts

Les concepts clés sont au nombre de quatre :

1. *État* ou *nœud*.
2. Un état peut être pourvu d'une information appelée *tag*.
3. *Transition* ou *arc*. Un arc est orienté si la distinction est faite entre le nœud source et le nœud cible. Dans le cas contraire, l'arc est dit non-orienté.
4. Une transition porte éventuellement une information appelée *étiquette*.

Nous allons maintenant aborder les définitions de nos principaux modèles, les graphes, les automates et les transducteurs. On construira ensuite à partir de ces trois modèles le concept général de machine à états.

6.3.3 Définitions

6.3.3.1 Graphe

Couple constitué de deux ensembles de nœuds et d'arcs (ou arêtes) $G = (Q, \Delta)$ avec $\Delta \subset Q \times Q$ pour le cas le plus simple.
Un graphe est *valué* lorsque les arcs portent une information et la valuation est une fonction associant les éléments de Δ à des éléments d'un ensemble quelconque I. Il est souvent aussi nécessaire de considérer des graphes où les nœuds portent une information accessible grâce à $t : Q \to T$ où T représente l'ensemble des valeurs associées aux nœuds.
Lorsque le sens des arcs est pertinent, on parle de graphe *orienté*.

6.3.3.2 Automate

Définition formelle au chapitre 2.

6.3.3.3 Transducteur

Un transducteur peut être considéré comme un automate dont l'alphabet est constitué de couples de lettres. La première permet de reconnaître des mots en entrée et la seconde d'en émettre en sortie ; un transducteur définit donc une fonction de l'ensemble des mots d'entrée vers l'ensemble des mots émis. C'est un sextuplet $(\Sigma_1, \Sigma_2, Q, i, F, \Delta)$ où :

Σ_1	L'alphabet d'entrée
Σ_2	L'alphabet de sortie
Q	Un ensemble d'états
$i \in Q$	L'état initial
$F \subseteq Q$	Un ensemble d'états terminaux
$\Delta \subseteq (Q \times (\Sigma_1 \cup \epsilon) \times (\Sigma_2 \cup \epsilon) \times Q)$	Un ensemble de transitions

On peut naturellement y adjoindre la notion de tags utilisée pour les automates.

6.3.3.4 Accès nommés aux transitions, clés, labels et étiquettes

On appelle accès nommé aux transitions sortant d'un état $q \in Q$ l'action de s'informer sur l'existence, la destination ou l'information portée par une transition de q à partir d'une clé $k \in K$. Plus formellement, un accès nommé est une fonction $f : Q \times K \to \Delta$, avec Δ le sous ensemble des triplets (q, d, p) définis par la machine où d représente le type d'information porté par les transitions appelée étiquette (on notera D l'ensemble des étiquettes). Par exemple dans le cas des automates, l'étiquette $d \in \Sigma$ est une lettre de l'alphabet et $k \in \Sigma$ pour permettre des accès nommés aux transitions lors de la reconnaissance d'un mot. Pour les transducteurs, $d \in \Sigma_1 \times \Sigma_2$ est un couple d'éléments de l'alphabet d'entrée et de l'alphabet de sortie mais $k \in \Sigma_1$. Ici, k et d ne sont pas de même nature. Généralement pour un graphe on n'a pas besoin de clé, les algorithmes ne nécessitant pas d'accès nommés mais cela ne signifie pas forcément que les arcs ne portent pas d'information. Par exemple, le problème du voyageur de commerce où les villes sont modélisées par les nœuds et les arcs portent la distance entre deux villes, ne requiert pas d'accès aux arcs à l'aide d'une clé mais ils portent tout de même

une information. C'est pourquoi il faut faire une distinction claire entre la donnée stockée sur une transition et la clé permettant d'y accéder. Dans la grammaire du DSL, l'étiquette portée par une transition est constituée d'une clé et d'un label, le label n'intervenant pas dans l'accès nommé aux transitions. L'existence d'une clé est uniquement conditionnée par l'algorithme destiné à être appliqué à la machine.
Muni des concepts de clé, de label et d'étiquette nous pouvons donner la définition de ce que nous appellerons une machine à états.

6.3.3.5 Machine à états (FSM, Finite State Machine)

Une machine à états A est un nonuplet $(Q, K, L, D, I, F, \Delta, T, \tau)$ d'ensembles finis où :

Q	est un ensemble de nœuds ou états
K	est un ensemble de clés
L	est un ensemble de labels
$D \subseteq K \times L$	est un ensemble d'étiquettes, c'est-à-dire de couples clé/label (k, l)
$I \subseteq Q$	est un ensemble d'états initiaux
$F \subseteq Q$	est un ensemble d'états finaux ou terminaux
$\Delta \subseteq Q \times D \times Q$	est un ensemble d'arcs ou transitions
T	est un ensemble de tags
$\tau \subseteq Q \times T$	l'ensemble des couples (q, t) associant un tag à chaque état

Plus fonctionnellement, on considère l'ensemble Δ sous forme d'une fonction de transition δ_1 associant un état et une clé à un ensemble de couples étiquette/but :

$$\delta_1 \ : \ Q \times K \rightarrow D \times Q$$

$$\delta_1(q, k) = \{(d, p) \in D \times Q \mid (q, d, p) \in \Delta \ et \ d = (k, l)\}$$

Le déterminisme limite à un le nombre d'éléments de cet ensemble.
On définit aussi la fonction δ_2 associant un état à l'ensemble de ses transitions sortantes :

$$\delta_2 \ : \ Q \rightarrow D \times Q$$

$$\delta_2(q) = \{(d, p) \in D \times Q \mid (q, d, p) \in \Delta\}$$

De plus, les arcs de la machine sont toujours considérés comme orientés. On verra un arc (q, p) non orienté comme une paire d'arcs orientés (p, q) et (q, p).

Cette définition recouvre les notions de graphe, arbre, automate déterministe ou pas, transducteur, machine de Moore et machine de Mealy, automate et transducteur pondérés, réseau de Pétri, bimachine. Tous les concepts définis ici ne sont pas forcément nécessaires dans une situation donnée et certains ensembles peuvent être vides, c'est pourquoi la librairie doit pouvoir donner à l'utilisateur la possibilité de choisir ceux qui sont appropriés et d'ignorer ceux qui ne sont pas pertinents.

6.4 Le DSL « machine à état »

La grammaire permettant à l'utilisateur de construire ses spécifications et de décrire le genre de machine qui l'intéresse est notée plus bas selon un formalisme GenVoca (un modèle de construction de générateur de code [4]) se rapprochant des déclarations de patrons en C++ ; cette écriture facilitera le passage à l'écriture du code.

6.4.1 Grammaire

Les symboles non terminaux sont paramétrés par une liste de symboles entre crochets. Les symboles terminaux sont écrits en gras, **T** représente un type quelconque :

fsm :	fsm[transition_container_type, tag_type, optimization_flag, edges_orientation, final_states, initial_states, bounds_checking]
transition_container_type :	transition_container_type[container_multiplicity, key, label_type, optimization_flag, access_complexity]
container_multiplicity :	**unique** | **multiple**
key :	key[key_type, key_multiplicity] | **none**
key_multiplicity :	**unique** | **multiple**
key_type :	**T** | **state**
label_type :	**T** | **none**
acces_complexity :	constant_time[mapping, reverse_mapping, `size_t`] | logarithmic[order_relation] | linear[equiv_relation] | hashing[hash_function]
mapping :	**unary_function : key_type** → **unsigned long**
reverse_mapping :	**unary_function :** `unsigned long` → **key_type**
order_relation :	**binary_function : key_type** × **key_type** → `bool`
equiv_relation :	**binary_function : key_type** × **key_type** → `bool`
hash_function :	**unary_function : key_type** → `unsigned long`
optimization_flag :	**space** | **speed** | **tradeoff**
tag_type :	**T** | **none**
edges_orientation :	**directed** | **nondirected**
final_states :	**with_final_states** | **no_final_states**
initial_states :	**one_initial_state** | **several_initial_states** | **no_initial_states**
bounds_checking :	**check_bounds** | **no_bounds_checking**

L'axiome, le fsm, est complètement défini par sept variables : le type du container de transition, le type du tag associé à chaque état, une directive d'optimisation, l'existence d'états terminaux et initiaux, deux variables binaires concernant la présence ou l'abscence d'orientation des transitions et la nécessité ou pas de vérifier à l'exécution la validité des opérations demandées par l'utilisateur.
Avec une telle grammaire, on est capable de générer approximativement 3000 structures de données valides.

6.4.2 Le container de transitions

Le container de transition possède cinq paramètres :

1. container_multiplicity : de ce paramètre dépend la morphologie globale de l'objet ; il donne le choix entre :
   – une représentation où à chaque nœud q est associée une structure de données contenant l'ensemble de ses arcs sortants $\delta_2(q) = \{(d, p) \in D \times Q \mid (q, d, p) \in \Delta\}$ telle que la représentation matricielle des automates dans ASTL (valeur **multiple**).
   – une représentation où un unique container stocke l'ensemble des triplets $(q, d, p) \in Q \times D \times Q$ de Δ. L'avantage d'une telle représentation est que l'espace mémoire occupé par la machine ne dépend que du nombre d'arcs et pas du tout du nombre d'états (valeur **unique**).
2. key : le type de la clé servant aux accès nommés aux transitions. La valeur **none** indique qu'aucune clé n'est nécessaire et que la fonction δ_1 est inutile. Dans le cas contraire il faut fournir deux choses :
   – key_type : le type concret de la clé. Il doit être constructible par défaut et assignable. De plus, le temps d'accès aux transitions lui impose des contraintes supplémentaires comme une relation d'ordre ou une relation d'équivalence.
   – key_multiplicity : de ce paramètre dépend le déterminisme de la machine. Avec des clés uniques (valeur **unique**), les containers associatifs utilisés pour stocker les transitions ne pourront contenir qu'un exemplaire de chaque clé. Autrement dit, il n'y aura pas deux arcs sortant d'un état étiquetés par la même clé, ce qui rejoint la notion d'automate déterministe lorsque $K = \Sigma$. Avec la valeur **multiple**, un nombre quelconque d'exemplaires est permis.
3. label_type : le type des labels étiquetant les transitions. On lui impose simplement d'être constructible par défaut et assignable. Il peut prendre la valeur **none** lorsque les labels ne sont pas nécessaires.
4. optimization_flag : une indication sur le type d'optimisation à appliquer au container de transition. Elle concerne l'ajout et la suppression des transitions. Elle peut prendre trois valeur :
   – **space** : optimisation en espace. Le générateur de configuration choisira parmi les composants se conformant aux spécifications et les combinaisons possibles, ceux et celles qui consomment le moins de mémoire.
   – **speed** : les composants les plus rapides seront utilisés au détriment de l'espace mémoire.
   – **tradeoff** : un compromis est trouvé entre les deux options précédentes.
5. access_complexity : complexité en temps pour les accès nommés aux transitions par δ_1. Ce paramètre n'a de sens que si un type de clé a été défini. Le temps d'accès peut prendre une des quatre valeurs suivantes :
   – constant_time : L'accès en temps constant induit une représentation matricielle de la machine. Il nécessite une bijection m entre les éléments de K et les entiers positifs. La fonction **mapping** doit implémenter m et **reverse_mapping** m^{-1}. En outre, il est nécessaire de connaître la taille de K qui doit être constante (troisième paramètre).
   – logarithmic : Le temps d'accès est proportionnel au logarithme du nombre de transitions présentes dans le container soit $\log(|\delta_2|)$. Ce type d'accès nécessite de définir une

relation d'ordre sur les éléments de K. La fonction **order_relation** doit implémenter l'opérateur < des objets clé.
- linear : La transition sera recherchée linéairement dans un container séquentiel. Cela suppose une relation d'équivalence sur les éléments de K. La fonction **equiv_relation** doit implémenter l'opérateur == sur les clés.
- hashing : Les transitions sont stockées dans une table de hachage qui requiert une fonction de hachage **hash_function** associant un entier positif à toute clé de K de manière déterministe. Il n'est pas imposé comme pour l'accès en temps constant que la fonction soit une bijection mais une bonne répartition des valeurs de hachage facilite la recherche et accélère l'accès.

6.4.3 Le type de tag

Le tag peut-être de n'importe quel type selon l'algorithme envisagé. On impose simplement qu'il soit constructible par défaut et assignable. La valeur **none** indique que les tags ne sont pas nécessaires.

6.4.4 L'optimisation

Les trois valeurs possibles pour la directive d'optimisation sont **space**, **speed** ou **tradeoff**. Elles ont la même sémantique que pour le container de transition mais elles s'appliquent ici à l'ensemble de la structure.

6.4.5 L'orientation des arcs

L'option edges_orientation peut prendre deux valeurs : **directed** ou **non_directed**. Dans le premier cas, chaque arc possède un nœud source et un nœud but, dans le deuxième on ne fait pas de distinction entre les deux états.

6.4.6 Les états terminaux

La valeur **with_final_states** impose à la machine de maintenir un ensemble de valeur booléennes indiquant l'appartenance de chaque état à l'ensemble des états terminaux.

6.4.7 Les états initiaux

L'état initial de la machine peut être unique, auquel cas la valeur **one_initial_state** est appropriée (automates et transducteurs déterministes). Avec **several_initial_states**, la machine stocke un ensemble d'états initiaux (automates et transducteurs non-déterministes) et la valeur **no_initial_states** indique l'inutilité d'une telle fonctionnalité et la notion d'état initial ne sera pas implémentée.

6.4.8 Vérification des opérations

L'option bounds_checking positionnée à la valeur **check_bounds** entraîne l'adjonction d'un garde-fou lors des appels de méthode : la validité des arguments est vérifiée et les pré-conditions sont testées. Dans le cas contraire (valeur **no_bounds_checking**) la validité des opérations est supposée être assurée par l'utilisateur et aucun test n'est mis en place.

6.4.9 Exemple d'utilisation

Pour rendre les choses plus claires et avant d'aborder l'ICCL et la manière dont les composants internes sont structurés, nous allons étudier un exemple concret d'utilisation du générateur de container et du DSL.
Le générateur est une méta-fonction à cinq paramètres, nous allons les définir de manière à obtenir l'équivalent de l'automate matriciel déterministe d'ASTL. Tout d'abord le type de la clé :

```
typedef key<unsigned char, unique> dfa_key;
```

Les clés sont des caractères non signés et chaque clé est unique car l'automate est déterministe. Ensuite il faut définir le temps d'accès :

```
typedef access_complexity<
            constant_time<identity<unsigned char>,
                          identity<unsigned char>,
                          256> dfa_access;
```

Le temps d'accès est constant et la bijection m associant les clés aux entiers positifs est simplement l'identité, il en va donc de même pour m^{-1}. Enfin, le troisième paramètre est la taille de l'alphabet. Nous pouvons donc maintenant spécifier le premier paramètre de la méta-fonction globale, c'est-à-dire le type du container de transitions :

```
typedef transition_container_type<multiple,
                                  dfa_key,
                                  none,
                                  speed,
                                  dfa_access> transition_container;
```

On choisit tout d'abord un container de transitions par état, puis on spécifie le type de clé, on précise qu'on ne veut pas de label, que l'ajout et la suppression de transition doivent être les plus rapides possible et que l'accès est en temps constant. On peut maintenant complètement spécifier le type abstrait du container :

```
typedef fsm<transition_container,
            none,
            speed,
            directed,
            one_initial_state,
            with_final_states,
            no_bounds_checking> abstract_fsm_type;
```

À l'aide de la métafonction `fsm_generator` on déclenche l'exécution du métaprogramme par le compilateur dont le résultat sera le type concret de la machine :

```
typedef fsm_generator<abstract_fsm_type>::R fsm_type;
```

Dans cet exemple, tous les paramètres ont été renseignés mais cela n'est pas obligatoire car le rôle du générateur est aussi de positionner à des valeurs par défaut les paramètres absents des spécifications incomplètes. Par exemple, les arcs sont orientés par défaut, il existe un

état initial et un ensemble d'états terminaux, la validité des opérations n'est pas vérifiée et l'optimisation en temps est préférée, ce qui rend équivalent à la précédente la déclaration suivante :

```
typedef fsm<transition_container, none> abstract_fsm_type;
```

Il existe une valeur par défaut pour tous les paramètres.

6.4.10 Valeurs par défaut des paramètres du DSL

Les valeurs par défaut favorisent le compromis entre vitesse et consommation d'espace mémoire de façon à s'adapter à un maximum de situation, évidemment pas de manière optimale mais raisonnablement efficace. Par exemple, le type obtenu en ne spécifiant aucune valeur dans l'expression correspond à un automate ASTL déterministe `DFA_map` sans tag et étiqueté par des caractères.

Le choix de ces valeurs par défaut doit être méticuleux car de bonnes valeurs, c'est-à-dire couvrant les besoins les plus courants, permettent à l'utilisateur de choisir la granularité de détail des spécifications qui lui convient et donc de ne pas s'embarrasser de concepts et de mécanismes dont il n'a pas besoin.

Paramètre	**Valeur**
container_multiplicity	multiple
key_type	`unsigned char`
key_multiplicity	unique
label_type	none
optimization_flag	speed
access_complexity	logarithmic
order_relation	`std::char_traits<key_type>::lt(key_type&, key_type&)`
tag_type	none
edges_orientation	directed
final_states	with_final_states
initial_states	one_initial_states
bounds_checking	no_bounds_checking

6.5 ICCL « machine à états »

L'assembleur construit l'objet demandé en assemblant des composants dont la granularité est beaucoup plus fine que celle rencontrée jusqu'ici en programmation générique. L'interface finale du FSM est en effet décomposée en plusieurs parties, chacune d'elles regroupant des méthodes et des attributs propres à des fonctionnalités particulières : ajout/suppression de transitions, accès nommé, itération sur les transitions d'un état, etc. Chaque morceau d'interface ainsi formé est implémenté par une classe différente et on construit l'objet final en empilant les morceaux par héritage paramétré ou par composition de patrons. Cette technique par synthétisation d'objets est appelée programmation par mixin (*mixin-based programming*).

Les décisions de conception quant à la segmentation des fonctionnalités dépendent des possibilités offertes à l'utilisateur par le DSL, de considérations architecturales internes et de l'application du paradigme de séparation des aspects.

6.5.1 Grammaire

Voici un extrait de la grammaire de l'ICCL concernant les machines déterministes. La syntaxe utilisée est empruntée à GenVoca : les symboles non terminaux ainsi que ceux qui sont paramétrés par des types désignent des patrons de classe qui seront instanciés par l'assembleur de composants. De même que pour le DSL, les symboles terminaux sont notés en gras.

FSM_type :	**FSM**[opt_bounds_checked_FSM]
opt_bounds_checked_FSM :	state_manager | **bounds_checked**[state_manager]
state_manager :	state_container[**label_manager**, tag_manager, transition_manager, initial_state_manager, final_state_manager]
tag_manager :	**tag_aggreger**[**tag_type**] | **empty_tag_manager**
transition_manager :	**transition_management** [transition_container, **edges_orientation**]
transition_container :	**hash_unique_container**[**key**, **label**, **hash_function**] | **map_unique_container**[**key**, **label**, **order_relation**] | **hash_multiple_container**[**key**, **label**, **hash_function**] | **map_multiple_container**[**key**, **label**, **order_relation**] | **vector_multiple_container**[**key**, **label**, **mapping**] | **list_multiple_container**[**key**, **label**, **equiv_relation**] | **vmap_multiple_container**[**key**, **label**, **order_relation**] | **dmap_multiple_container**[**key**, **label**, **order_relation**]
initial_state_manager :	**unique_initial_state_manager** | **multiple_initial_states_manager**
final_state_manager :	**bit_set_manager** | **char_set_manager** | **state_set_manager**

6.5.2 Principes d'implémentation

6.5.2.1 Instanciation fainéante des patrons

Les méthodes des patrons de classe sont considérées comme des patrons de fonction et gérées comme telles. Elles ne sont instanciées réellement que lorsqu'elles sont appelées, ce qui peut apparaître souvent comme problématique notamment lors des tests. Cependant, cette spécificité du C++ peut être utilisée à notre avantage. Un exemple simple tiré de STL est le *reverse iterator*. Un reverse iterator est un adaptateur inversant la sémantique des opérateurs d'incrémentation et de décrémentation : au lieu d'avancer lors d'un appel à l'opérateur `++`, l'itérateur recule et inversement ce qui permet d'adapter les algorithmes de manière externe. Par exemple, il est possible de trier des éléments en ordre décroissant plutôt qu'en ordre croissant simplement en adaptant avec cet objet les itérateurs passés à l'algorithme `sort`. Voici un extrait du code de l'adaptateur :

```
template <class Iterator>
```

```
class reverse_iterator
{
protected:
  Iterator current;
public:
  typedef iterator_traits<Iterator>::iterator_category iterator_category;

  self& operator++() {
    --current;
    return *this;
  }

  self& operator--() {
    ++current;
    return *this;
  }

  self operator+(difference_type n) const {
    return self(current - n);
  }

  reference operator[](difference_type n) const {
    return *(*this + n);
  }
  // ...
};
```

Le patron `reverse_iterator` requiert évidemment du type d'instanciation `Iterator` qu'il se conforme au moins au concept d'itérateur bidirectionnel. Mais il implémente aussi l'interface et le comportement de l'itérateur à accès direct avec ses opérateurs `+`, `+=`, `-`, `-=` et `[ ]`. Lors d'une instanciation de l'adaptateur avec un itérateur bidirectionnel, les opérateurs `+` et `-` ne seront pas définis par l'itérateur adapté `current` mais puisqu'aucun appel aux fonctionnalités d'accès direct ne sera effectué (l'adaptateur appartient à la même catégorie que l'adapté) la compilation et l'exécution se passeront bien.

Il est donc possible, dans un cadre strictement contrôlé, de se passer d'implémenter certaines méthodes et contraintes, ce qui s'avère d'une grande utilité pour la programmation par empilement de couches fonctionnelles et optionnelles (programmation par mixin) : les méthodes et les implémentations non requises par les spécifications de l'utilisateur peuvent dans certaines conditions être omises et purement et simplement supprimées des types manipulés ce qui entraîne un couplage plus lâche entre couches et une plus grande souplesse dans les combinaisons de composants. Cette technique est par exemple utilisée pour la mise en œuvre de la couche de vérification des paramètres des méthodes (bounds checking). La classe chargée de vérifier les préconditions d'appel comprend toutes les méthodes théoriquement utilisables même si l'objet encapsulé ne les implémente pas.

6.5.2.2 Programmation par mixin

La programmation par mixin ([7], [62], [63]) consiste en une conception par couches des composants logiciels. En C++, le terme mixin désigne un arrangement particulier de la hiérarchie d'héritage où une classe implémentant une couche étend les fonctionnalités de sa super-classe sans avoir connaissance du type qu'elle enrichit : la classe dérivée est définie indépendamment et la classe de base est un paramètre d'utilisation de la première déterminé à l'utilisation et non à la conception.
En C++, ce type d'architecture peut être implémenté par un héritage paramétré où la classe dérivée adapte l'interface et le comportement de la classe de base définie au moment de l'instanciation du code :

```
template <class Base>
class X : public Base
{ };
```

Exemple (inspiré de [63]) : un automate ASTL comptant le nombre d'appels à la fonction de transition :

```
template <class DFA>
class compteur : public DFA
{
  int appels;
public:
  compteur()
    : appels(0), DFA()
  { }

  State delta1(State q, const Alphabet &a) const {
    ++appels;
    return DFA::delta1(q, a);
  }
};
```

Cette structuration par adaptation et enrichissement donne de la souplesse à l'utilisation car tous les types partageant la même interface sont utilisables directement et la redondance est minimale :

```
compteur<DFA_matrix<> > A1;
compteur<DFA_map<> >    A2;
```

Un mixin n'est rien d'autre qu'un adaptateur raffinant le concept auquel appartient sa super-classe. Les contraintes et problèmes posés par cette technique sont majoritairement dûs aux dépendances entre couches, c'est-à-dire entre les classes mixin et leurs super-classes. Ces dépendances doivent être minimales mais il apparaît rapidement qu'un certain nombre de difficultés doivent être contournées :

– un patron de classe n'est pas un type, les vérifications habituelles ne s'appliquent que lorsque le patron est instancié. De plus, seules les méthodes appelées sont instanciées car elle sont considérés comme des patrons de fonctions, ce qui signifie que toutes les erreurs présentes dans le code ne seront détectées que si la couverture est totale, ce qui est rarement le cas. Cependant, cette instanciation paresseuse du code peut être utilisée à notre avantage.

– nécessaire standardisation des interfaces et des échanges entre couches. Il est évident que la communication verticale entre objets de couches différentes nécessite une conception rigoureuse des interfaces et des comportements. De plus, ces liens peuvent s'étendre sur plusieurs niveaux, par conséquent les couches ne sont pas dépendantes uniquement de leur super-couche directe.

– problème des constructeurs. Une classe n'hérite jamais des constructeurs de sa super-classe. Au moment de l'écriture de la classe dérivée, la classe de base n'est pas connue mais son constructeur doit être appelé explicitement ce qui suppose de connaître les types des arguments. L'interchangeabilité de la classe de base est donc mise à mal. Plusieurs solutions existent : les classes de construction ou la métaprogrammation [15].

– le sous-typage engendré par l'utilisation de mixin et de patrons ne permet pas toujours le polymorphisme entre classe de base et classe dérivée. Exemple :

```
template <class T, class Alloc>
inline
bool operator==(const list<T, Alloc> &x, const list<T, Alloc> &y);
```

Conformément au standard du C++, cet opérateur doit être surchargé pour toutes les classes dérivant de `list`.

La programmation par mixin va beaucoup plus loin que le simple héritage paramétré. Notamment, la technique de conception par collaboration et couches mixin (*mixin layers* [62], [70]) permet un meilleur découpage et une réelle séparation des aspects mais l'héritage paramétré simple suffit à la mise en œuvre de notre métaclasse de machine à états expérimentale.

6.5.2.3 Architecture par couches

La figure 6.2 montre l'empilement des couches constituant le type final de la classe machine à états. Les boîtes arrondies dénotent les composants optionnels pouvant être purement et simplement absents du type sans que les dépendances entre couches n'aient à en pâtir : il s'agit des fonctionnalités indépendantes du reste de l'architecture ; elles ne seront pas intégrées si les spécifications ne les requièrent pas. Par ailleurs, ces boîtes optionnelles peuvent n'être que des coquilles vides implémentant uniquement l'interface qui leur correspond mais pas le code. Ces « façades » sont utiles lorsque les fonctionnalités qui leur incombent ne sont pas nécessaires mais que l'algorithme les manipulant exige de la classe une interface complète.

6.5.3 Interface du concept FSM

Cette section décrit les spécifications du concept « machine à états » dans un formalisme proche de STL. Chacune des sous-parties correspond à une couche de l'architecture décrite précédemment et est implémentée par une classe différente.
L'interface est exposée dans son intégralité mais l'existence ou non des méthodes et des fonctionnalités dépend des spécifications de la machine : toutes les opérations ne sont pas intégrées au type généré ; seules celles qui sont requises par l'utilisateur font partie du type de la machine.

Notations

`X`	Un type modèle de FSM
`x`	Un objet de type `X`
`y`	Un objet de type `const X`
`q`, `p`	Des identifiants d'état de type `X::state_type`
`k`	Un objet de type `X::key_type`
`l`	Un objet de type `X::label_type`
`r`	Une référence sur un objet de type `X::label_type`
`b`	Une variable de type `bool`

Types associés

Expression	Description
`X::state_type`	Type des identifiants d'état
`X::key_type`	Type des clés (éventuellement `none`)
`X::label_type`	Type des labels (éventuellement `none`)
`X::tag_type`	Type des tags (éventuellement `none`)
`X::edges_type`	Type du proxy permettant l'itération sur les arcs sortant d'un état
`X::const_iterator`	Type des itérateurs sur la séquence des identifiants d'état

Valeurs statiques associées

Expression	Description	Type
`X::directed`	Un booléen indiquant si les arcs sont orientés ou non	`bool`

Gestion des états

Expression	Description	Type
`x.new_state();`	Alloue un nouvel état	`state_type`
`x.remove_state(q);`	Supprime l'état `q` et son tag	`void`
`x.clear();`	Supprime tous les états, toutes les transitions et tous les tags de la machine	`void`
`x.copy_state(q, p);`	Copie `q` vers `p`	`void`
`x.duplicate_state(q);`	Duplique l'état `q`	`X::state_type`
`y.begin();`	Renvoie un itérateur sur le début de la séquence des identifiants d'état	`X::const_iterator`
`y.end();`	Renvoie un itérateur sur la fin de la séquence des identifiants d'état	`X::const_iterator`

Gestion des tags

Expression	Description	Type
`x.tag(q);`	Accès au tag de l'état `q`	`X::tag_type&`
`y.tag(q);`	Accès au tag de l'état `q`	`const X::tag_type&`

Gestion des transitions

Expression	Description	Type
`x.new_transition(q, k, p);`	Création d'une transition entre les états `q` et `p` de clé `k`	`void`
`x.remove_transition(q, k);`	Suppression de la transition de clé `k` sortant de `q`	`size_t`
`x.redirect_transition(q, k, p);`	Redirection de la transition de clé `k` sortant de `q` vers `p`	`void`
`x.edges(q);`	Retourne le proxy permettant l'accès à l'ensemble des transitions de l'état `q`	`edges_type`

Accès nommés

Expression	Description	Type
`y.target(q, k);`	Retourne le but de la transition de clé `k` sortant de l'état `q`	`X::state_type`
`y.target(q, k, r);`	Retourne le but de la transition de clé `k` sortant de `q` et affecte à `r` son label	`X::state_type`

Gestion des labels

Expression	Description	Type
`x.new_transition(q, k, l, p);`	Création d'une transition de clé `k` et de label `l` entre les états `q` et `p`	`void`
`x.label(q, k);`	Accès au label de la transition de clé `k` sortant de `q`	`X::label_type&`
`y.label(q, k);`	Accès au label de la transition de clé `k` sortant de `q` sur un container constant	`const X::label_type&`

Gestion des états terminaux

Expression	Description	Type
`x.final(q, b);`	Ajoute l'état `q` à l'ensemble des états terminaux si `b` est vrai, sinon supprime `q` de l'ensemble	`void`
`y.final(q);`	Teste l'appartenance de `q` à l'ensemble des états terminaux	`bool`

Gestion des états initiaux

Expression	Description	Type
`x.initial();`	Accès à l'ensemble des états initiaux (cas non déterministe)	`set<state_type>&`
`x.initial();`	Accès à l'état initial (cas déterministe)	`state_type&`
`y.initial();`	Accès à l'ensemble des états initiaux (cas non déterministe)	`const set<state_type>&`
`y.initial();`	Accès à l'état initial (cas déterministe)	`state_type`

6.6 Projection du DSL vers l'ICCL

La projection est l'application stricte du paradigme de séparation de l'espace des problèmes et de l'espace des solutions. À une expression utilisateur formulée grâce au DSL, le parser va associer une expression formulée sous forme de combinaison de patrons de classes constituant le type final de la machine à états.

6.6.1 Mécanique

Dans un premier temps (figure 6.3) le parser vérifie la validité du patron d'expression passé à la métafonction du générateur (syntaxe et types), puis une sous-fonction affecte les valeurs par défaut aux paramètres non renseignés dans l'expression. Dans un troisième temps, le parser s'assure que la combinaison des paramètres utilisateur a un sens ; en effet, toute expression n'est pas valide, certaines configurations peuvent ne pas avoir de sens et le compilateur doit les détecter et générer une erreur. Par exemple, on ne peut pas avoir de container unique de transitions à accès en temps constant.
La phase suivante consiste à construire la configuration à plat correspondant à l'expression,

c'est-à-dire encapsuler dans une classe les définitions des types des composants que l'assembleur réutilisera dans la dernière phase pour constituer le type final en combinant les patrons et les classes couche par couche. On retrouve la configuration à la base de la hiérarchie (figure 6.2) car toutes les couches doivent avoir accès à l'information.

6.6.2 Exemple

Le tableau suivant montre comment les spécifications du DSL concernant la complexité d'accès aux transitions sont projetées vers les composants internes décrits dans la grammaire de l'ICCL. Il s'agit du cas où les valeurs des clés sont uniques (déterminisme) et où les containers de transitions sont multiples (un par état). Cette projection est paramétrée par une autre variable, la directive d'optimisation qui détermine le type de container à utiliser.

Complexité ↓ / Optimisation →	**space**	**speed**	**tradeoff**
constant_time	`vector`	`vector`	`vector`
logarithmic	vmap	`map`	dmap
linear	`vector`	`list`	`deque`
hashing	`hash_map`	`hash_map`	`hash_map`

Les containers utilisés sont des objets STL standards hormis **vmap** et **dmap** qui implémentent des `map` (même interface et même comportement) avec respectivement un `vector` et une `deque`. Ce ne sont rien d'autre que des tableaux maintenus triés et auxquels l'accès se fait par dichotomie. Un vecteur consomme moins d'espace qu'une map mais l'insertion y est moins rapide, c'est pourquoi il est préféré lorsque la directive d'optimisation porte sur l'espace mémoire. Lorsqu'il s'agit d'aller vite, la `map` est utilisée. Le compromis entre les deux est d'utiliser une `deque` pour laquelle les temps d'insertion seront en moyenne plus rapide que dans le vecteur et l'espace consommé moins important que pour la `map`.

Toutes les projections qui ont lieu lors du parsing sont représentables par des matrices telles que celle décrite plus haut. Ces matrices peuvent avoir plus de deux dimensions lorsque la projection dépend de plusieurs paramètres, par exemple le type du container de transitions dépend en fait de cinq paramètres. Certains n'ont pas d'influence sur le résultat, comme dans le cas de la table de hachage vu précédemment, et certaines cellules des matrices sont vides lorsque la combinaison des paramètres n'a pas de sens. À chaque matrice correspond une métafonction associant les n types dont dépend la projection au type de composant concret. Elles déterminent le comportement de l'assembleur de composants qui prend le relais du parser et empile les couches requises par l'expression.

6.6.3 L'interface externe « FSM Wrapper »

La dernière couche de la hiérarchie (figure 6.2), le « FSM Wrapper » finalise l'interface de classe générée. Il est chargé d'implémenter toute méthode ou définition de type relative à l'abstraction à laquelle appartient la classe. C'est ici qu'on a la possibilité d'envelopper la machine dans l'interface qui correspond le mieux à des besoins purement « cosmétiques » , principalement des renommages de types et de méthodes selon le vocabulaire propre au sous domaine abordé. Par exemple, l'interface d'un graphe ne comporte pas les notions d'états ou

de transitions mais de nœuds et d'arcs (node et vertex). De plus, l'interface d'un graphe non valué ne nécessite pas de références à des types de clé ou de label ; dans ce cas le type de clé est tout simplement l'état :

```
template <class FSM>
class graph : public FSM
{
public:
  typedef state_type node_type;

  node_type new_node() {
    return new_state();
  }

  void new_vertex(node_type q, node_type p) {
    new_transition(q, p, p);
  }

  // ...
};
```

Cette dernière couche permet toutes les projections de l'interface fondamentale vers d'autres abstractions plus appropriées à la situation.
Voici un extrait de l'interface possible pour un arbre binaire. Les clés sont de type entier et peuvent prendre deux valeurs, 0 ou 1 selon que l'on désigne le fils gauche ou le fils droit d'un nœud :

```
template <class FSM>
class binary_tree : public FSM
{
public:
  typedef state_type node_type;

  node_type left_son(node_type q) const {
    return target(q, 1);
  }

  node_type right_son(node_type q) const {
    return target(q, 2);
  }

  // ...
};
```

Évidemment, cet habillage est optionnel et par défaut l'interface se limite à celle décrite à la section 6.5.3. Une métafonction permet de projeter l'interface de base vers une des abstractions suivantes : tree, graph, automaton (l'interface d'ASTL), transducer. Par exemple pour un arbre binaire :

```
typedef interface_mapper<fsm_type, tree<2> >::R binary_tree_type;
```

le paramètre d'instanciation du patron `tree` désigne le nombre de fils que peut avoir un nœud. Ce patron est spécialisé pour la valeur 2 et implémente les méthodes d'accès aux fils gauche et droit vues plus haut.

6.7 Le générateur de curseurs

Le générateur de curseurs profite pleinement du DSL « machine à états » car la structure et l'interface d'un curseur dépendent uniquement du type de container sur lequel il évolue. Il n'est donc nullement besoin d'écrire une nouvelle grammaire, ni même de récrire un nouveau parser en intégralité, seuls l'ICCL et le comportement de l'assembleur diffèrent. Ils sont beaucoup plus simples et ne nécessitent que très peu de code. L'utilisation du générateur de curseurs est immédiat pourvu qu'on ait sous la main les spécifications abstraites du container vues à la section 6.4.9 :

```
typedef cursor_generator<abstract_fsm_type>::R cursor_type;
```

Ce type de réutilisation est typique d'une conception où s'applique le paradigme de séparation des problèmes et des solutions : la double interface DSL et ICCL donne un degré de liberté supplémentaire en permettant à l'un des deux modules d'évoluer et de changer sans perturber le reste de l'architecture et en permettant une réutilisation maximale.

6.8 Conclusion

Il ressort de cette expérience de programmation générative de machine à états en C++ un certain nombre de points positifs et négatifs, ces derniers ayant principalement trait aux difficultés dûes au langage d'implémentation :

- L'utilisation plus qu'intensive de patrons de classe et de fonction présente plusieurs inconvénients. Ils ne sont pas insurmontables mais mettent en lumière le fait que le langage n'a pas été conçu pour ce genre de programmation et qu'il n'est pas des plus adapté notamment en ce qui concerne le sucre syntaxique. Les métaprogrammes ne sont pas toujours faciles à lire même si la décomposition des grosses métafonctions en sous-fonctions permet d'augmenter la lisibilité.

- Le déboguage des patrons et des métaprogrammes n'est pas des plus aisés. Le C++ ne possède pas de structure ni de mécanique permettant de contraindre les instanciations de manière à constituer des gardes-fou. De plus, la complexité des types construits à la compilation par imbrication de patrons dépassent souvent les limites de ce qu'un être

humain peut lire et appréhender.

- Il n'existe pas de moyen d'émettre des messages d'erreur propres et explicites lors d'un problème à la compilation d'un métaprogramme. L'utilisation de nom de classe pour signaler une erreur ne constitue qu'une demie solution car tous les compilateurs ne renseignent pas le programmeur sur l'endroit du code où le problème réel se situe.

- Enfin, les deux derniers aspects concernent plus les compilateurs que le langage lui-même. Les techniques utilisées en métaprogrammation sont souvent aux limites de ce que peuvent supporter les compilateurs actuels. Les temps de compilation dépendent de la complexité du programme et du métaprogramme bien que ce dernier ne produise pas de code à proprement parler ce qui conduit à des temps de calcul importants même pour un exécutable réduit ne réalisant pas d'énormes opérations.

- Il est rare de pouvoir compiler un métaprogramme un tant soit peu complexe mais conforme aux spécifications du C++ sans buter sur un bogue. Dans ce domaine, le compilateur GNU g++ 3.0 [66] se démarque nettement des autres de par sa solidité, sa conformité au standard et l'efficacité du code exécutable produit.

Cette expérience a permis en outre de mettre en valeur et de valider un ensemble de démarches et de techniques de conception :

- Un haut niveau d'abstraction pour les spécifications simplifie énormément l'utilisation et la compréhension de la librairie. Il n'est pas nécessaire d'avoir une connaissance profonde de l'implémentation des machines à états pour l'utiliser et l'utilisateur choisit le degré de détail de ses spécifications.

- La grammaire du DSL permet de générer plus de 3000 structures de données différentes. Cette adaptabilité est un des points forts des librairies actives.

- Les types sont construits à partir de petits composants modulaires et facilement combinables ce qui implique une redondance minimale et une factorisation des fonctionnalités maximale.

- La séparation de l'espace des problèmes et de l'espace des solutions découple presque totalement l'interface externe abstraite de l'interface interne des composants concrets. Ce degré de liberté supplémentaire permet l'évolution indépendante des interfaces et leur réutilisation : le générateur de curseur utilise le même DSL que le générateur de container.

- L'efficacité des implémentations spécifiques écrites « à la main » est conservée. L'utilisation de patrons, de typage statique et d'inlining garantit un niveau élevé d'optimisation du code.

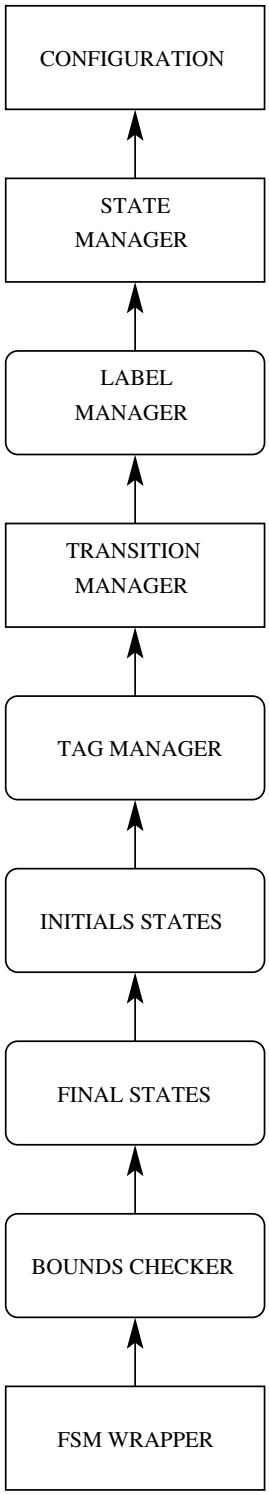

FIG. 6.2 – Architecture par couches du métatype « machine à états »

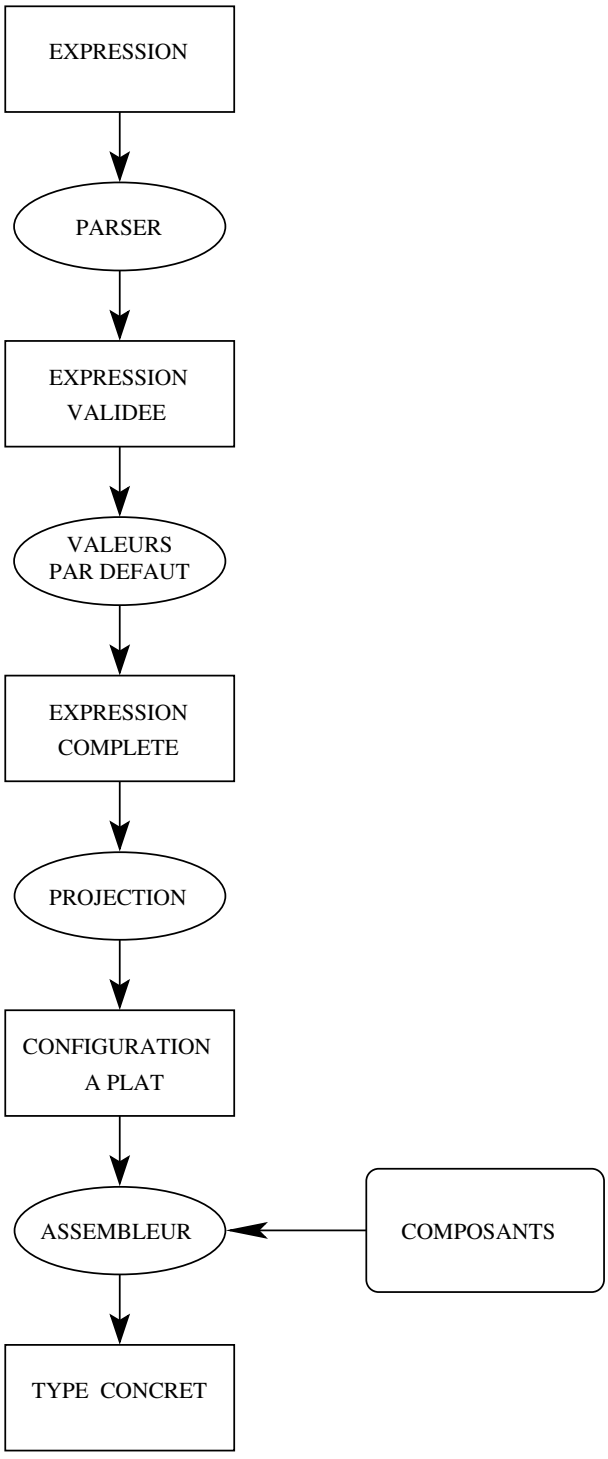

FIG. 6.3 – Génération d'un type concret à partir d'une spécification abstraite

Chapitre 7

Conclusion

L'écriture de la librairie ASTL a comblé un vide en ce qui concerne le génie logiciel sur les automates grâce à une analyse profonde des besoins et la définition rigoureuse de concepts et de composants concrets. Elle a permis de valider le concept d'architecture à trois niveaux de la STL, algorithmes - itérateurs - containers. Le modèle à deux niveaux algorithmes - containers ayant rapidement trouvé ses limites car il ne permet pas dans de trop nombreuses situations de factoriser correctement le développement. Il est nécessaire pour certains algorithmes et certains containers de réaliser des développements spécifiques pour obtenir la version optimale.
Cette architecture à trois niveaux utilisée dans cette expérience n'est pas une simple application de l'approche de la STL. Le concept de curseur qui constitue la glu entre les algorithmes et les automates apporte une autre dimension : intégrer des notions étendues de parcours dans les itérateurs. Les itérateurs classiques ne permettent que des parcours séquentiels ; pour un objet reposant sur une structure de graphe comme les automates, la notion de parcours est primordiale. Les curseurs permettent de définir le parcours indépendamment de l'algorithme, ce qui permet une grande souplesse de développement, rend l'écriture d'algorithmes plus simple sans perdre en efficacité. Les différents parcours proposés en standard (simple, local, en profondeur, en largeur, marqués ou non) ne sont pas limitatifs, l'utilisateur pouvant par exemple aisément piloter un parcours récursif en utilisant un `stack_cursor`.

Cette expérience a aussi montré que la puissance d'une architecture générique pouvait s'exprimer horizontalement avec la multiplication naturelle des couples algorithmes-containers. Et aussi verticalement, ce qui est original, en appliquant le principe des adaptateurs qui permet de manipuler les curseurs comme une algèbre munie d'opérateurs de composition, intersection, union, etc. Un exemple de puissance de cette approche est donné par le logiciel `word-grep` écrit comme un assemblage de curseurs, qui réalise une recherche multiple d'expressions régulières sur des mots en utilisant une combinaison relativement complexe d'une dizaine de curseurs et d'adaptateurs et ceci avec un code très proche de l'optimalité. Le logiciel écrit en peu de temps soutient la comparaison avec `gnu-grep` qui a lui demandé des années pour atteindre la qualité actuelle.
La librairie développée a déjà prouvé son efficacité dans plusieurs domaines, le développement d'un outils de manipulation de transducteurs statistiques, `word-grep`, une version par automates de `locate`, un outil d'analyse de logs et une implémentation de cache dans un analyseur morphologique.

La programmation générique d'outils demande un apprentissage de techniques de développement spécifiques : elle requiert une connaissance approfondie des mécanismes de patron qu'offre le C++ et elle est difficile car elle ne peut s'appuyer sur les facteurs externe du développement (ergonomie, adaptation aux besoins, etc.) du fait que les outils développés doivent s'appliquer dans un maximum de cas. C'est une programmation qui s'impose un très haut niveau d'abstraction quant aux cibles et aux attentes que l'on a des fonctionnalités. Plus l'approche est abstraite et haut niveau plus la librairie sera puissante et utile. Parallèlement, elle exige une approche pragmatique et très proche des problématiques d'implémentation tant dans les facteurs de l'optimalité des algorithmes utilisés que dans une judicieuse utilisation des ressources mémoire. Tous ces aspects contribuent à rendre plus complexe non seulement la conception mais aussi l'utilisation. L'expérience de programmation générative nous a permis de confirmer que les librairies actives pallient agréablement la principale faiblesse de la programmation générique : sa relative complexité d'utilisation. En cachant complètement les concepts informatiques et en proposant une interface toujours plus abstraite, ce type de librairie constitue une réelle avancée au-delà de la programmation générique pure. En outre, elles permettent une factorisation des fonctionnalités défiant toute concurrence et une meilleure séparation des aspects.
Cette programmation par patron s'avère d'une puissance redoutable mais constitue une arme à double tranchant à plusieurs titres : peu de compilateurs sont assez robustes pour gérer toutes les constructions théoriquement possibles et beaucoup de rigueur est nécessaire lors du développement pour éviter d'aller trop loin dans la complexité du code. De manière générale, en C++, toute la difficulté est d'arriver à n'utiliser dans toutes les possibilités qu'offre le langage, que ce qui est strictement nécessaire.

En ce qui concerne les patrons de conception, l'approche générique n'apporte rien de réellement nouveau à l'ensemble des patrons classiques, on en retrouve d'ailleurs beaucoup dans ce genre de code et c'est l'un des aspects positifs de ce type de programmation car il ne désoriente pas le programmeur habitué aux pratiques courantes. Au contraire, elle insiste sur la nécessité de construire le code de manière claire et en cela, valide bon nombre de motifs architecturaux déjà connus car l'élément le plus prépondérant reste comme dans tout développement, le soucis constant que l'on doit porter à l'architecture logicielle globale car c'est elle qui permet d'obtenir à moindre coût et moindre risque un logiciel pérenne et fiable.

Cette expérience nous indique plusieurs directions de recherche et développement : du côté tactique, la consolidation de l'extension de la librairie à d'autres types de structure de données mal couverts par les librairies actuelles. En particulier, approfondir les notions d'algorithmes génériques et d'adaptateurs de curseur sur les graphes, les transducteurs, les bi-machines, etc. Du côté stratégique, simplifier l'écriture de telles librairies, en développant un DSL pour la STL, en migrant vers java qui est un langage plus simple qui commence aujourd'hui à offrir une syntaxe de patron permettant une approche générique, en appliquant des principes de programmation aspectuelle plus poussés qui permettent la dissociation de l'approche centrée sur les besoins et de l'approche centré sur l'algorithmique et l'efficacité.

Annexe A

ASTL 1.2
Documentation de référence

Cette annexe décrit les concepts et modèles d'ASTL version 1.2. Le formalisme utilisé s'inspire de la documentation STL de SGI ([26]). Il a été pris soin de documenter les composants le plus rigoureusement possible dans le but de ne laisser aucune place à l'interprétation : tous les concepts utilisés sont définis préalablement ici ou dans la documentation de STL.
La première partie du chapitre présente les concepts fondamentaux alphabet, tag, état, edges et automate, puis viennent les concepts de curseurs et enfin les modèles de containers, de curseurs standards et d'adaptateurs. Enfin, la dernière partie concerne les algorithmes.

A.1 Concepts

La description d'un concept implique les informations suivantes :

1. **Catégorie** : le sous-ensemble de l'espace des composants abstraits contenant le concept décrit (curseur, container, outils).
2. Le **type de composant**, ici concept.
3. **Description** du composant.
4. **Raffinement de :** relation de sous-typage avec d'autres concepts. Le concept hérite des propriétés et des contraintes de ses super-types.
5. **Les types associés**, c'est-à-dire les types qu'un objet se conformant au concept doit exporter/publier (définir dans son interface publique).
6. Les **notations** utilisées dans la section *sémantique des expressions*.
7. **Définitions :** introduction du vocabulaire propre au concept.
8. **Expressions valides** : signatures des méthodes imposées.
9. **Sémantique des expressions** : description exhaustive des pré/post-conditions et sémantique des méthodes.
10. **Garanties de complexité** : le temps de calcul imposé aux différentes méthodes.
11. **Invariants :** propriétés vérifiées à tout instant par un objet se conformant au concept.
12. **Modèles :** types concrets se conformant au concept.
13. **Notes :** remarques diverses.
14. **Voir aussi :** pointeur vers les concepts liés.

Alphabet

Catégorie : Outils **Type de composant : Concept**

Description

Un alphabet regroupe les informations et les opérations relatives à un certain type.

Raffinement de

Le concept d'alphabet est un raffinement du concept STL Character Traits.

Types associés

L'ensemble des types et méthodes imposés est constitué d'une partie de celui du Character Traits et de fonctionnalités d'itération :

Type de caractère	`X::char_type`	Le type de caractère décrit par ce trait
Type entier	`X::int_type`	Un type entier capable de représenter toute valeur valide de type `char_type`
Type d'itérateur	`X::iterator`	Un type d'itérateur capable de parcourir tout l'alphabet

Notations

`X`	un type modèle d'alphabet
`c`, `c1`, `c2`	valeurs de type `X::char_type`
`e`, `e1`, `e2`	valeurs de type `X::int_type`

Expressions valides

Nom	Expression	Requiert	Type
Affectation	`X::assign(c1, c2)`	`c1` est une lvalue	`void`
Egalité	`X::eq(c1, c2)`		`bool`
Comparaison	`X::lt(c1, c2)`		`bool`
Conversion en `char_type`	`X::to_char_type(e)`		`X::char_type`
Conversion en entier	`X::to_int_type(c)`		`X::int_type`
Comparaison entière	`X::eq_int_type(e1, e2)`		`bool`
Début d'intervalle	`X::begin()`		`X::iterator`
Fin d'intervalle	`X::end()`		`X::iterator`
Taille	`X::size`		`size_t`

Sémantiques des expressions

Nom	Expression	Sémantique	Postcondition
Affectation	`assign(c1, c2)`	Effectue l'affectation `c1 = c2`	`eq(c1, c2) == true`
Egalité	`eq(c1, c2)`	retourne `true` ssi `c1` et `c2` sont égaux	
Comparaison	`lt(c1, c2)`	retourne `true` ssi `c1` est plus petit que `c2`	
Conversion en caractère	`to_char_type(e)`	convertit l'entier `e` de type `int_type` en caractère de type `char_type`	
Conversion en entier	`to_int_type(c)`	convertit le caractère `c` de type `char_type` en entier de type `int_type`	`to_char_type( to_int_type(c))` est l'identité
Comparaison entière	`eq_int_type(e1,e2)`	compare les deux entiers `e1` et `e2`. Si `e1 == to_int_type(c1)` et `e2 == to_int_type(c2)`, `eq_int_type(e1,e2) == eq(c1,c2)`	
Début d'intervalle	`begin()`	renvoie un forward iterator constant sur le premier élément de l'alphabet	
Fin d'intervalle	`end()`	renvoie un forward iterator constant pointant derrière le dernier élément de l'alphabet	
Taille	`size`	Une constante statique égale à la taille de l'alphabet	`size ==end()-begin()`

Garanties de complexité

Toutes les opérations s'exécutent en temps constant.

Invariants

Pour tout `c1`, `c2`, une et une seule des expressions `X::lt(c1, c2)`, `X::lt(c2, c1)`, `X::eq(c1, c2)` doit être vraie.

Modèles

`char_subset`, `int_subset`, `plain`, `french`, `digits`.

Notes

Toutes les méthodes présentées ne sont pas forcément nécessaires, c'est la représentation en mémoire et la méthode d'accès aux transitions qui détermine les besoins :

- Un accès direct impose une taille finie pour l'alphabet et une bijection entre ses éléments et les entiers positifs (attribut `size`, méthodes `to_char_type` et `to_int_type`).
- Un accès en temps logarithmique impose une relation d'ordre sur les éléments de l'alphabet (méthode `lt`).
- Un accès en temps linéaire, une relation d'équivalence (méthodes `eq` et `eq_int_type`).
- Un accès par hachage, une fonction de hachage (méthode `to_int_type`), c'est-à-dire une fonction déterministe projetant les éléments de l'alphabet vers les entiers positifs.

Voir aussi

Le concept `Character Traits` de STL.

Tag

Catégorie : Outils **Type de composant : Concept**

Description
Un tag contient l'information associée à un état d'un automate.

Raffinement de
Le concept de tag est un raffinements des concepts STL assignable, default constructible, copy constructible, equality comparable.

Modèles
`empty_tag`.

Notes
Le type `empty_tag` s'utilise lorsqu'aucune information particulière n'est nécessaire.

État Déterministe

Catégorie : Outils **Type de composant : Concept**

Description

Un état déterministe désigne une position dans un automate, c'est-à-dire un objet contenant un tag et l'ensemble des transitions sortant de cet état. Il ne contient pas directement l'information mais sa valeur permet au container d'y accéder. On appelle communément ces objets des *handlers*.

Raffinement de

assignable, default constructible, copy constructible, equality comparable, lessthan comparable.

Modèles

Le type `State` exporté par les DFA.

Notes

- La plupart du temps ces handlers sont représentés par des types simples comme les entiers ou les pointeurs.
- Une valeur spéciale, l'état nul ou puits doit être définie. Les états déterministes étant des types généralement exportés par des modèles de DFA, cette valeur est accessible à travers leur interface.
- Le concept d'état déterministe sert de base à celui d'état non-déterministe défini comme un ensemble d'états déterministes. Les complexités imposées aux méthodes d'un tel état, pour des raisons d'efficacité, le force à maintenir l'ensemble trié, c'est pourquoi il est requis que les modèles d'état déterministe définissent une relation d'ordre (concept lessthan comparable).

Voir aussi

Le concept de DFA.

État Non Déterministe

Catégorie : Outils **Type de composant : Concept**

Description

Un état non déterministe est un container associatif simple à clés uniques représentant un ensemble de positions dans un automate non déterministe.

Raffinement de

assignable, default constructible, copy constructible, equality comparable, lessthan comparable, simple associative container, unique associative container, état déterministe.

Notations

`X` Un type modèle d'état non déterministe

Types associés

Type des états	`X::value_type`	Le type des objets stockés dans l'ensemble

Modèles

Le type `State` exporté par les NFA.

Notes

Une implémentation par `set` STL est généralement utilisée.

Voir aussi

Le concept de NFA et d'état déterministe.

Edges Déterministe

Catégorie : Outil **Type de composant : Concept**

Description

Un edges est un proxy permettant l'accès aux transitions d'un état et l'itération sur la séquence des transitions. Un edges déterministe possède l'interface d'une map STL constante dont le type des clés est la lettre et le type des données les états (paires (lettre, état)). Il ne permet pas l'ajout ou la suppression de transition mais seulement la consultation.
Ce concept est destiné à l'implémentation des automates déterministes.

Raffinement de

container de paires associatif à clés uniques (pair associative container, unique associative container).

Notations

`X` Un type modèle d'edges déterministe

Types associés

clé	`X::key_type`	le type des lettres étiquetant les transitions
donnée	`X::data_type`	le type des identifiants d'état
valeur	`X::value_type`	le type des transitions, `pair<key_type, data_type>`

Expressions valides

Les expressions valides sont celles qui s'appliquent aux containers associatifs constants de paires à clés uniques. Les méthodes non constantes ne sont pas autorisées.

Sémantiques des expressions

La sémantique des expressions est la même que celle des containers associatifs constants de paires à clés uniques.

Garanties de complexité

Le temps d'accès direct à une transition dépend de l'implémentation de l'automate sous-jacent.

Voir aussi

DFA, Edges non-déterministe.

Edges Non-Déterministe

Catégorie : Outil **Type de composant : Concept**

Description

Un edges est un proxy permettant l'accès aux transitions d'un état et l'itération sur la séquence des transitions. Un edges non-déterministe possède l'interface d'une multimap STL constante dont le type des clés est la lettre et le type des données les états (paires (lettre, état)). Il ne permet pas l'ajout ou la suppression de transition mais seulement la consultation. Ce concept est destiné à l'implémentation des automates non-déterministes.

Raffinement de

container de paires associatif à clés multiples (pair associative container, multiple associative container).

Notations

`X` Un type modèle d'edges non déterministe

Types associés

clé	`X::key_type`	le type des lettres étiquetant les transitions
donnée	`X::data_type`	le type des identifiants d'état
valeur	`X::value_type`	le type des transitions, `pair<key_type, data_type>`

Expressions valides

Les expressions valides sont celles qui s'appliquent aux containers associatifs constants de paires à clés multiples. Les méthodes non constantes ne sont pas autorisées.

Sémantiques des expressions

La sémantique des expressions est la même que celle des containers associatifs constants de paires à clés multiples.

Garanties de complexité

Le temps d'accès direct à une transition dépend de l'implémentation de l'automate sous-jacent.

Voir aussi

NFA, Edges déterministe.

DFA

Catégorie : Container **Type de composant : Concept**

Description

Un DFA est un container d'automate déterministe. Il alloue les états, les tags qui leurs sont associés et les transitions. Il stocke l'ensemble des états terminaux et l'état initial. La couche des curseurs repose sur ce concept.

Types associés

alphabet	`Sigma`	un type de trait modèle d'Alphabet
lettres	`Alphabet`	le type des objets étiquetant les transitions de l'automate : `Sigma::char_type`
états	`State`	le type des identifiants d'état
tags	`Tag`	le type des objets associés aux états
transitions d'un état	`Edges`	proxy de transitions modèle d'edges déterministe
itérateur	`const_iterator`	un type d'itérateur sur la séquence des identifiants d'état de l'automate

Notations

- `X` Un type modèle d'automate déterministe
- `x` Un objet de type `X`
- `q`, `p` Des identifiants d'état de type `X::State`
- `a` Une lettre de type `X::Alphabet`
- `0` Un identifiant spécial représentant l'état puits (ou nul).

Définitions

Un identifiant d'état est dit *valide* s'il a été renvoyé par la méthode d'allocation d'état `new_state` et n'a jamais été passé en argument de `del_state`. Il est invalidé lorsque l'automate est détruit.

Expressions valides

Toutes les expressions contenant des identifiants d'état requièrent qu'ils soient valides.

Nom	Expression	Requiert	Type de retour
état nul	`x.null_state`		`X::State`
création d'état	`x.new_state()`	`x` est mutable	`X::State`
création de transition	`x.set_trans(q, a, p)`	`x` est mutable	`void`
suppression de transition	`x.del_trans(q, a)`	`x` est mutable	`void`
duplication d'état	`x.duplicate_state(q)`	`x` est mutable	`X::State`
suppression d'état	`x.del_state(q)`	`x` est mutable	`void`
positionnement de l'état initial	`x.initial(q)`	`x` est mutable	`void`
copie d'état	`x.copy_state(q, p)`	`x` est mutable	`void`
redirection	`x.change_trans(q, a, p)`	`x` est mutable	`void`
état initial	`x.initial()`		`X::State`
état but	`x.delta1(q, a)`		`X::State`
transitions d'un état	`x.delta2(q)`	La valeur de retour est un modèle d'edges déterministe	`X::Edges`
tag	`x.tag(q)`		`X::Tag&`
final	`x.final(q)`		`bool&`
nombre d'état	`x.state_count()`		`unsigned long`
nombre de transitions	`x.trans_count()`		`unsigned long`
début de séquence d'états	`x.begin()`		`X::const_iterator`
fin de séquence d'états	`x.end()`		`X::const_iterator`

Sémantiques des expressions

Nom	Expression	Précondition	Sémantique	Postcondition
état nul	`x.null_state`			
création d'état	`q=x.new_state()`			`final(q) == false` et `delta2(q)` est vide
création de transition	`x.set_trans(q,a,p)`	`delta1(q,a) == 0`		`delta1(q,a) == p`
suppression de transition	`x.del_trans(q,a)`	`delta1(q,a) != 0`		`delta1(q,a) == 0`
duplication d'état	`p = x.duplicate_state(q)`		`copy_state(q, new_state())`	`final(q) == final(p)`, `delta2(q) == delta2(p)`, `tag(q) == tag(p)`

Nom	Expression	Précondition	Sémantique	Postcondition
suppression d'état	`x.del_state(q)`		Toutes les transitions de `q` sont supprimées ainsi que son tag	`q` n'est plus un identifiant valide
affectation de l'état initial	`x.initial(q)`			`initial() == q`
copie d'état	`x.copy_state(q, p)`		`del_state(p)` `p = duplicate_state(q)`	`final(q) == final(p)`, `delta2(q) == delta2(p)`, `tag(q) == tag(p)`
redirection	`x.change_trans(q,a,p)`	`delta1(q,a) != 0`	`del_trans(q,a)` `set_trans(q,a,p)`	`delta1(q,a) == p`
état initial	`x.initial()`			
état but	`x.delta1(q,a)`			
transitions d'un état	`x.delta2(q)`			
tag	`x.tag(q)`			
final	`x.final(q)`			
nombre d'état	`x.state_count()`			
nombre de transitions	`x.trans_count()`			
début de séquence d'états	`x.begin()`			
fin de séquence d'états	`x.end()`			

Garanties de complexité

Toutes les méthodes sauf `delta1` s'exécutent en temps constant amorti.

Modèles

`DFA_matrix`, `DFA_map`, `DFA_bin`, `DFA_hash`, `DFA_mtf`, `DFA_tr`, `DFA_min`, `DFA_compact`.

Voir aussi

Alphabet, Tag, État Déterministe, NFA.

NFA

Catégorie : Container **Type de composant : Concept**

Description

Un NFA est container d'automate non-déterministe. Les états sont des ensembles d'identifiants d'états déterministes. Son interface est semblable à celle du DFA.

Raffinement de

DFA.

Types associés

Identiques à ceux du DFA.

Expressions valides

Identiques à celles du DFA.

Sémantiques des expressions

Identiques à celles du DFA.

Garanties de complexité

Identiques à celles du DFA.

Modèles

`NFA_matrix`, `NFA_mmap`, `NFA_epsilon`.

Voir aussi

État non-déterministe, DFA.

Curseur Simple

Catégorie : Curseurs **Type de composant : Concept**

Description

Un curseur simple est un pointeur sur un état d'un DFA ou d'un NFA. Il cache la structure de données sous-jacente et permet les déplacements le long des transitions définies.

Raffinement de

assignable, default constructible, copy constructible, equality comparable.

Types associés

Type d'état	`X::State`	Le type des identifiants d'états de l'automate parcouru
Type d'alphabet	`X::Alphabet`	Le type des lettres étiquetant les transitions de l'automate
Type de tag	`X::Tag`	Le type des tags associés aux états de l'automate

Notations

- `X` Un type modèle de curseur simple
- `x`, `y` Des objets de type `X`
- `q` Un identifiant d'état de type `X::State`
- `a` Une lettre de type `X::Alphabet`

Définitions

Un curseur simple est *singulier* s'il n'a pas été initialisé, il est *déréférençable* s'il pointe sur un état de l'automate autre que l'état puits (`null_state`) et *incrémentable* s'il est déréférençable.

Expressions valides

Nom	Expression	Requiert	Type de retour
constructeur par défaut	`X x;`		
constructeur de copie	`X x(y);` `X x = y;`		
affectation	`x = y;`	`x` est une l-value	`X&`
positionnement	`x = q;`	`x` est une l-value	`X&`
comparaison	`x == y;`		convertible en `bool`
état pointé	`x.src();`		`X::State`
état puits	`x.sink();`		convertible en `bool`
incrémentation	`x.forward(a);`		convertible en `bool`
existance d'une transition	`x.exists(a);`		convertible en `bool`
état final	`x.src_final();`		convertible en `bool`
tag	`x.src_tag();`		`X::Tag`

Sémantiques des expressions

Nom	Expression	Précondition	Sémantique	Postcondition
constructeur par défaut	`X x;`			`x` est singulier

constructeur de copie	`X x(y);`
	
	`X x = y;`
				`x == y`

affectation	`x = y;`			`x == y`
positionnement	`x = q;`		`x` pointe sur `q`	`x.src() == q`
comparaison	`x == y;`		`x.src() == y.src()`	
état pointé	`x.src();`	`x` est déréférençable		
état puits	`x.sink();`		`true` si `x` pointe sur l'état puits	
incrémentation	`x.forward(a);`	`x` est déréférençable	avance sur la transition étiquetée par `a` sortant de `x.src()`	`x.sink() == true` si la transition n'est pas définie
existance d'une transition	`x.exists(a);`	`x` est déréférençable	`true` s'il existe une transition étiquetée par `a` sortant de `x.src()`	
état final	`x.src_final();`	`x` est déréférençable	`true` si `x.src()` est final	
tag	`x.src_tag();`	`x` est déréférençable		

Garanties de complexité

Les méthodes autres que `forward`, `exists` et `final` s'exécutent en temps constant amorti.

Modèles

`cursor`.

Notes

Le déterminisme de l'automate sous-jacent n'est pas garanti mais quelque soit son type, le comportement du curseur simple ne varie pas. Autrement dit, sur un automate non-déterministe le curseur doit assurer la déterminisation à la volée.

Voir aussi

curseur monodirectionnel.

Curseur Monodirectionnel

Catégorie : Curseurs **Type de composant : Concept**

Description

Un curseur monodirectionnel est un pointeur sur une transition d'un automate, c'est-à-dire un triplet (état source, lettre, état but). Il permet l'itération sur l'ensemble des transitions sortant de l'état source.

Raffinement de

Curseur simple.

Types associés

Identiques à ceux du curseur simple.

Notations

`X`	Un type modèle de curseur monodirectionnel
`x`, `y`	Des objets de type `X`
`q`	Un identifiant d'état de type `X::State`
`a`	Une lettre de type `X::Alphabet`

Définitions

Un curseur monodirectionnel est *singulier* s'il n'est pas initialisé. Il est *déréférençable* lorsqu'il pointe sur une transition dont l'état source et l'état but sont définis et tous deux différents de l'état puits. Il est *incrémentable* s'il est déréférençable.

Expressions valides

En plus de celles du curseur simple, les expressions suivantes doivent être valides :

Nom	Expression	Requiert	Type de retour
lettre	`x.letter();`		`X::Alphabet`
première transition	`x.first_transition();`		convertible en `bool`
transition suivante	`x.next_transition();`		convertible en `bool`
incrémentation	`x.forward();`		`void`
positionnement	`x.find(a);`		convertible en `bool`
état but	`x.aim();`		`X::State`
état but final	`x.aim_final();`		convertible en `bool`
tag de l'état but	`x.aim_tag();`		`X::Tag`

Sémantiques des expressions

Nom	Expression	Précond.	Sémantique	Postcond.
comparaison	`x == y;`		`x.src() == y.src()` `&&` `x.aim() == y.aim()` `&& x.letter() == y.letter()`	
lettre	`x.letter();`	`x` est déréférençable	la lettre de la transition pointée	
première transition	`x.first_transition();`	`x.sink()` `== false`	positionne `x` sur la première transition sortant de `x.src()`	`x` est déréférençable s'il existe des transitions
transition suivante	`x.next_transition();`	`x` est déréférençable		`x` est déréférençable s'il restait des transitions
incrémentation	`x.forward();`	`x` est déréférençable	avance le long de la transition de source `x.src()` et étiquetée par `x.letter()`	`x.src()` est égal à `x.aim()` avant incrémentation
positionnement	`x.find(a);`	`x.sink()` `== false`	positionne `x` sur la transition étiquetée par `a`	`x` est déréférençable si la transition est définie
état but	`x.aim();`	`x` est déréférençable		
état but final	`x.aim_final();`	`x` est déréférençable		
tag de l'état but	`x.aim_tag();`	`x` est déréférençable		

Garanties de complexité

Les méthodes autres que `first_transition`, `next_transition` et `find` s'exécutent en temps constant amorti.

Modèles

`forward_cursor`.

Voir aussi

curseur simple, curseur pile, curseur file.

Curseur Pile

Catégorie : Curseurs **Type de composant : Concept**

Description

Un curseur pile est une pile de curseurs monodirectionnels. Il vérifie les mêmes propriétés, les méthodes s'appliquant au sommet de la pile. A chaque incrémentation le curseur résultant est empilé et une méthode spéciale permet le dépilement. Ce type de curseur entre en jeu dans l'implémentation du parcours en profondeur. Il permet en outre de matérialiser les conditions de départ et d'arrêt des parcours.

Raffinement de

curseur monodirectionnel.

Types associés

Identiques à ceux du curseur monodirectionnel.

Notations

`X`	Un type modèle de curseur pile
`x`, `y`	Des objets de type `X`

Définitions

Un curseur pile est *singulier* lorsqu'il n'a pas été initialisé ou lorsque sa pile est vide. Il est *déréférençable* lorsque sa pile n'est pas vide et que le curseur de sommet de pile est déréférençable. Il est *incrémentable* s'il est déréférençable.

Expressions valides

En plus de celles du curseur monodirectionnel, l'expression suivante doit être valide :

Nom	Expression	Requiert	Type de retour
dépilement	`x.backward();`		convertible en `bool`

Sémantiques des expressions

Toutes les opérations relatives au curseur monodirectionnel s'appliquent au curseur de sommet de pile.

Nom	Expression	Précondition	Sémantique	Postcondition
constructeur par défaut	`X x;`		construit un curseur dont la pile est vide	`x` est singulier
comparaison	`x == y`		compare les piles de `x` et `y`	
dépilement	`x.backward();`	la pile de `x` n'est pas vide	dépile le curseur de sommet de pile et renvoie `false` si la pile résultante est vide	
incrémentation	`x.forward();`	`x` est déréférençable	avance le long de la transition courante et empile le curseur résultant	

Garanties de complexité

La méthode `backward` s'exécute en temps constant, les complexités des autres méthodes sont identiques à celles du curseur monodirectionnel.

Modèles

`stack_cursor`.

Voir aussi

curseur monodirectionnel, curseur file, curseur de parcours en profondeur.

Curseur File

Catégorie : Curseurs **Type de composant : Concept**

Description

Un curseur file est une file de curseurs monodirectionnels. Il vérifie les mêmes propriétés mais les méthodes d'itération sur les transitions d'un état enfile les positions successives. Une méthode spéciale permet de les défiler et d'implémenter le parcours en largeur.

Raffinement de

curseur monodirectionnel.

Types associés

Identiques à ceux du curseur monodirectionnel.

Notations

`X`	Un type modèle de curseur file
`x`, `y`	Des objets de type `X`

Définitions

Un curseur file est *singulier* s'il n'a pas été initialisé ou lorsque sa file est vide. Il est *déréférençable* si sa file n'est pas vide et que le curseur courant est déréférençable. Il est *incrémentable* s'il est déréférençable.

Expressions valides

En plus de celles du curseur monodirectionnel, l'expression suivante doit être valide :

Nom	Expression	Requiert	Type de retour
défiler	`x.dequeue();`		convertible en `bool`

Sémantiques des expressions

Nom	Expression	Précondition	Sémantique	Postcond.
constructeur par défaut	`X x;`		construit un curseur dont la file est vide	`x` est singulier
comparaison	`x == y`		compare les files de `x` et `y`	
défiler	`x.dequeue();`	la file n'est pas vide	défile un curseur monodirectionnel qui devient la position courante	si la file est vide `x` est singulier
première transition	`x.first_transition();`	`x` n'est pas singulier et `x.sink() == false`	le curseur courant est positionné sur la première transition de l'état source puis enfilé	si la transition est définie `x` est déréférençable
transition suivante	`x.next_transition();`	`x` est déréférençable	le curseur courant est positionné sur la transition suivante puis enfilé	s'il restait des transitions `x` est déréférençable

Garanties de complexité

La méthode `dequeue` s'exécute en temps constant, les complexités des autres méthodes sont identiques à celles du curseur monodirectionnel.

Modèles

`queue_cursor`.

Voir aussi

curseur monodirectionnel, curseur pile, curseur de parcours en largeur.

Curseurs de Parcours en Profondeur

Catégorie : Curseurs **Type de composant : Concept**

Description

Un curseur de parcours en profondeur (PEP) permet l'itération sur la séquence des transitions d'un automate dans l'ordre « profondeur d'abord ». Il repose sur un curseur pile qui ne contient que des curseurs monodirectionnels déréférençables et sert à définir des intervalles pour les algorithmes basés sur un PEP.

Raffinement de : default constructible, copy constructible, equality comparable.

Types associés : Identiques à ceux du curseur simple.

Notations

`X`	Un type modèle de curseur de PEP
`x`, `y`	Des objets de type `X`
`q`	Un identifiant d'état de type `X::State`
`a`	Une lettre de type `X::Alphabet`
`p`	Un objet dont le type est un modèle de curseur pile

Définitions

Un curseur de PEP est *singulier* s'il n'est pas initialisé ou lorsque sa pile est vide. Il est *déréférençable* et *incrémentable* s'il n'est pas singulier. On note `last - first` le nombre de transitions traversées lors d'un parcours défini par l'intervalle `[first, last)` où `first` et `last` sont deux curseurs de PEP.

Expressions valides

Nom	Expression	Requiert	Type de retour
constructeur	`X x(p);`	`p` est un modèle de curseur pile	
constructeur par défaut	`X x;`		
constructeur de copie	`X x(y);` `X x = y;`		
incrémentation	`x.forward();`		convertible en `bool`
comparaison	`x == y`		convertible en `bool`
lettre	`x.letter();`		`X::Alphabet`
état source	`x.src();`		`X::State`
état but	`x.aim();`		`X::State`
état source final	`x.src_final();`		convertible en `bool`
état but final	`x.aim_final();`		convertible en `bool`
tag de l'état source	`x.src_tag();`		`X::Tag`
tag de l'état but	`x.aim_tag();`		`X::Tag`

Sémantiques des expressions

Nom	Expression	Précondition	Sémantique	Postcondition
constructeur	`X x(p);`	`p` est déréférençable	construit un curseur dont la pile contient uniquement `p`	`x` est déréférençable et incrémentable
constructeur par défaut	`X x;`		construit un curseur dont la pile est vide	`x` est singulier
incrémentation	`x.forward();`	`x` n'est pas singulier	passe à la transition suivante du parcours en profondeur. Renvoie `true` s'il y a eu empilement, `false` en cas de dépilement	
comparaison	`x == y`		compare les piles de `x` et `y`	
lettre	`x.letter();`	`x` n'est pas singulier		
état source	`x.src();`	`x` n'est pas singulier		
état but	`x.aim();`	`x` n'est pas singulier		
état source final	`x.src_final();`	`x` n'est pas singulier		
état but final	`x.aim_final();`	`x` n'est pas singulier		
tag de l'état source	`x.src_tag();`	`x` n'est pas singulier		
tag de l'état but	`x.aim_tag();`	`x` n'est pas singulier		

Garanties de complexité

Toutes les méthodes s'exécutent en temps constant.

Modèles

`dfirst_cursor`, `dfirst_cursor_mark`

Notes

- La méthode `forward` renvoie `true` si l'incrémentation fait partie de la phase descendante du parcours (empilements) et `false` lors de la phase ascendante (dépilements). Le parcours s'arrête lorsque la comparaison entre le curseur et la condition d'arrêt est vraie. La condition d'arrêt est un curseur de PEP initialisé avec une pile généralement vide. Il n'est cependant pas interdit d'utiliser une pile non-vide, cela a pour effet de limiter le parcours à un sous-ensemble de l'automate.
- Le curseur de PEP ne gère pas les problèmes de cycle dans l'automate. Pour avoir un curseur garantissant l'arrêt du parcours il faut utiliser un curseur de PEP avec marquage.

Voir aussi

curseur pile, curseur de parcours en largeur, curseur de PEP avec marquage.

Curseur de Parcours en Largeur

Catégorie : Curseurs **Type de composant : Concept**

Description

Un curseur de parcours en largeur (PEL) permet l'itération sur la séquence des transitions d'un automate dans l'ordre « largeur d'abord ». Il repose sur un curseur file qui ne contient que des curseurs monodirectionnels déréférençables.

Raffinement de : default constructible, copy constructible, equality comparable.

Types associés

Identiques à ceux du curseur simple.

Notations

`X`	Un type modèle de curseur de PEL
`x`, `y`	Des objets de type `X`
`q`	Un identifiant d'état de type `X::State`
`a`	Une lettre de type `X::Alphabet`
`f`	Un objet dont le type est un modèle de curseur file

Définitions

Un curseur de PEL est *singulier* s'il n'est pas initialisé ou lorsque sa file est vide. Il est *déréférençable* et *incrémentable* s'il n'est pas singulier. On note `last - first` le nombre de transitions traversées lors d'un parcours défini par l'intervalle `[first, last)` où `first` et `last` sont deux curseurs de PEL.

Expressions valides

Nom	Expression	Requiert	Type de retour
constructeur	`X x(f);`	`f` est un modèle de curseur file	
constructeur par défaut	`X x;`		
constructeur de copie	`X x(y);` `X x = y;`		
incrémentation	`x.next_transition();`		convertible en `bool`
comparaison	`x == y`		convertible en `bool`
lettre	`x.letter();`		`X::Alphabet`
état source	`x.src();`		`X::State`
état but	`x.aim();`		`X::State`
état source final	`x.src_final()`		convertible en `bool`
état but final	`x.aim_final()`		convertible en `bool`
tag de l'état source	`x.src_tag()`		`X::Tag`
tag de l'état but	`x.aim_tag()`		`X::Tag`

Sémantiques des expressions

Nom	Expression	Précondition	Sémantique	Postcond.
constructeur	`X x(f);`	`f` est déréférençable	construit un curseur dont la file contient uniquement `f`	`x` est déréférençable et incrémentable
constructeur par défaut	`X x;`		construit un curseur dont la file est vide	`x` est singulier
incrémentation	`x.next_transition();`	`x` n'est pas singulier	passe à la transition suivante du parcours en largeur. Renvoie `true` s'il y a eu enfilement, `false` en cas de défilement	
comparaison	`x == y`		compare les files de `x` et `y`	
lettre	`x.letter();`	`x` n'est pas singulier		
état source	`x.src();`	`x` n'est pas singulier		
état but	`x.aim();`	`x` n'est pas singulier		
état source final	`x.src_final()`	`x` n'est pas singulier		
état but final	`x.aim_final()`	`x` n'est pas singulier		
tag de l'état source	`x.src_tag()`	`x` n'est pas singulier		
tag de l'état but	`x.aim_tag()`	`x` n'est pas singulier		

Garanties de complexité

Toutes les méthodes s'exécutent en temps constant.

Modèles

`bfirst_cursor`, `bfirst_cursor_mark`.

Voir aussi

curseur file, curseur de parcours en profondeur, curseur de PEL avec marquage.

Fonction de Marquage

Catégorie : Foncteur **Type de composant : Concept**

Description

Une fonction de marquage est un objet fonction (foncteur) chargé de garder la trace de l'ensemble des états déjà visités lors d'un parcours d'automate. Elle prend en argument un identifiant d'état `q` et renvoie `false` s'il n'a jamais été utilisé comme argument auparavant. Pendant toute la durée de vie restante du foncteur, un appel à la fonction avec pour paramètre `q` renverra `true`.

Raffinement de

fonction unaire (adaptable unary function).

Types associés

Type des états	`argument_type`	le type des états dont on veut garder la trace
Type de retour	`result_type`	booléen

Notations

- `X` Un type modèle de foncteur de marquage
- `x` Un objet de type `X`

Expression valide

Nom	Expression	Requiert	Type de retour
test	`x(q);`	`q` est lessthan-comparable ou convertible en `int`	`bool`

Sémantiques des expressions

Nom	Expression	Précondition	Sémantique	Postcondition
test	`x(q);`		test si un appel avec `q` a déjà été effectué depuis la construction de `x`	`x(q) == true`

Garanties de complexité

L'appel à la fonction de test s'exécute au pire en un temps proportionnel au logarithme du nombre des états déjà visités.

Modèles

`set_marker`, `hash_marker`, `tag_marker`.

Notes

Il est requis que le type des états soit convertible en entier pour le cas où le marquage s'effectue par mémorisation dans une table de hachage. Il faut qu'il définisse une relation d'ordre (lessthan-comparable) lorsque le stockage est basé sur un arbre (`set` STL par exemple). Ces deux propriétés sont vérifiées par le type des états déterministes des automates ASTL mais seule la deuxième est vraie pour les états non-déterministes.

Curseurs de PEP avec Marquage

Catégorie : Curseurs **Type de composant : Concept**

Description

Un curseur de PEP avec marquage garde la trace des états visités lors du parcours en profondeur grâce à un foncteur de marquage garantissant l'arrêt de l'itération sur les automates cycliques. Son interface est exactement la même que celle d'un curseur de PEP.

Raffinement de

curseur de PEP.

Types associés

Identiques à ceux du curseur de PEP.

Expressions valides

Identiques à celles du curseur de PEP.

Sémantiques des expressions

Identiques à celles du curseur de PEP.

Garanties de complexité

À chaque incrémentation un appel à la fonction de marquage est effectué donc `forward` s'exécute en un temps proportionnel à celui de l'appel.

Invariants

Un curseur de PEP avec marquage passe exactement deux fois sur les transitions du parcours, une première fois pendant la phase descendante et une seconde fois pendant la phase ascendante.

Modèles

`dfirst_cursor_mark`.

Voir aussi

curseur de PEP, curseur de PEL avec marquage.

Curseurs de PEL avec Marquage

Catégorie : Curseurs **Type de composant : Concept**

Description
Un curseur de PEL avec marquage garde la trace des états visités lors du parcours en largeur grâce à un foncteur de marquage garantissant l'arrêt de l'itération sur les automates cycliques. Son interface est exactement la même que celle d'un curseur de PEL.

Raffinement de
curseur de PEL.

Types associés
Identiques à ceux du curseur de PEL.

Expressions valides
Identiques à celles du curseur de PEL.

Sémantiques des expressions
Identiques à celles du curseur de PEP.

Garanties de complexité
À chaque incrémentation un appel à la fonction de marquage est effectué donc `forward` s'exécute en un temps proportionnel à celui de l'appel.

Modèles
`bfirst_cursor_mark`.

Voir aussi
curseur de PEL, curseur de PEP avec marquage.

A.2 Modèles

Les modèles sont des types concrets appartenant à une plusieurs catégories de concepts. La description d'un modèle implique les informations suivantes :

1. **Catégorie** : le sous-ensemble de l'espace des concepts abstraits auquel le modèle se conforme.
2. Le **type de composant**, ici tout simplement type.
3. La **description** du composant.
4. **Définition :** le fichier source contenant le code du composant.
5. Les **paramètres d'instanciation** à fournir pour utiliser le patron de classe.
6. Le ou les **modèles** auxquels le composant se conforme.
7. Les **contraintes de type** : les propriétés que doivent vérifier les types utilisés pour l'instanciation du composant.
8. Les **classes de base publiques** dont hérite éventuellement le composant.
9. Les **membres** de la classe, c'est-à-dire les types exportés, les attributs et méthodes publiques. Un tableau contient leurs noms, le concept qui les impose à la classe et leurs descriptions.
10. Les **nouveaux membres** qui sont propres à la classe.
11. Les **fonctions d'aide** permettant de construire le composant à la volée lors de l'utilisation d'un algorithme.
12. **Notes :** remarques diverses.
13. **Voir aussi :** pointeur vers les concepts et les modèles connexes.

Remarque Tous les modèles de curseur possède au moins une fonction d'aide associée dont le nom est construit en remplaçant le suffixe `_cursor` du nom de classe par `c`. Par exemple, la fonction associée au type `forward_cursor` s'appelle `forwardc`.

range<T, T x, T y>

Catégorie : Outils **Type de composant : Type**

Description

`range` est un patron de classe définissant un alphabet dont les éléments de type numérique `T` appartiennent à l'intervalle [`x`, `y`]. Il définit toutes les méthodes et tous les attributs du concept Alphabet.

Définition

`alphabet.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`T`	Le type des lettres	
`x`	La borne inférieure de l'alphabet	
`y`	La borne supérieure de l'alphabet	

Modèle de

Alphabet, Character Traits

Contraintes de type

- `T` est un type de base.
- `x` est de type `T`.
- `y` est de type `T`.
- `y - x` > 0.

Classes de base publiques

`std::char_traits`

Membres

Membre	Imposé par	Description
`char_type`	Alphabet	Le type des lettres `T`
`int_type`	Alphabet	Un type entier capable de représenter toute valeurs valides de type `char_type`
`iterator`	Alphabet	Un type d'itérateur monodirectionnel permettant le parcours de l'alphabet
`bool assign(char_type c1, char_type c2)`	Alphabet	Effectue l'affectation `c1 = c2`

Membre	Imposé par	Description
`bool eq(char_type c1, char_type c2)`	Alphabet	Retourne vrai si `c1 == c2`
`bool lt(char_type c1, char_type c2)`	Alphabet	Retourne vrai si `c1 < c2`
`int_type to_int_type(char_type c)`	Alphabet	Projette bijectivement le caractère `c` vers une valeure entière de l'intervalle [0, `y - x`]
`char_type to_char_type(int_type e)`	Alphabet	Fonction inverse de `to_int_type`
`bool eq_int_type(int_type e1, int_type e2)`	Alphabet	Retourne vrai si `e1 == e2`
`iterator begin()`	Alphabet	Retourne un itérateur sur le début de l'alphabet
`iterator end()`	Alphabet	Retourne un itérateur sur la fin de l'alphabet
`size_t size`	Alphabet	La taille de l'alphabet `y - x`

Voir aussi

Alphabet, Character Traits.

DFA_matrix<Sigma, Tag>

Catégorie : Containers **Type de composant : Type**

Description

Un `dfa_matrix` est un container d'automate déterministe stockant les transitions des états dans une matrice permettant un accès en temps constant. Son utilisation requiert un alphabet de taille fixe et une bijection entre les éléments de l'alphabet et les entiers positifs. Cette bijection est définie par les deux méthodes statiques `Sigma::to_int_type` (projection de l'alphabet vers les entiers) et `Sigma::to_char_type` (fonction inverse). La taille est définie par `Sigma::size`.

Définition

`dfa_matrix.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`Sigma`	Le trait de l'alphabet	`plain`
`Tag`	Le type des tags	`empty_tag`

Modèle de

DFA.

Contraintes de type

- `Sigma` est un modèle d'alphabet définissant `size`, `to_int_type()`, `to_char_type()`.
- `Tag` est constructible par défaut et assignable.

Classes de base publiques

`DFA_concept`.

Membres

Membre	**Imposé par**	**Description**
`Sigma`	DFA	Le trait de l'alphabet.
`Alphabet`	DFA	Le type des lettres étiquetant les transitions, `Sigma::char_type`.
`State`	DFA	Le type des identifiants d'états modèle d'état déterministe.
`Tag`	DFA	Le type des tags associés aux états.
`Edges`	DFA	Le type du proxy permettant l'itération sur les transitions sortant d'un état. `Edges` doit être un modèle d'edges déterministe.
`const_iterator`	DFA	Type de l'itérateur utilisé pour parcourir l'ensemble des identifiants d'état de l'automate.
`null_state`	DFA	La valeur de l'état nul, de type `State`.
`DFA_matrix()`	DFA	Crée un automate vide.
`~DFA_matrix()`	DFA	Destructeur.
`State new_state()`	DFA	Crée un nouvel état.
`void set_trans(State q, const Alphabet &a, State p)`	DFA	Crée une transition étiquetée par `a` sortant de `q` et dirigée vers `p`.
`void del_trans(State q, const Alphabet &a)`	DFA	Supprime la transition de `q` étiquetée par `a`.
`State duplicate_state(State q)`	DFA	Renvoie l'identifiant d'un nouvel état qui est une copie conforme de `q`.
`void del_state(State q)`	DFA	Supprime l'état q.
`void initial(State q)`	DFA	Positionne l'état initial sur `q`.
`void copy_state(State q, State p)`	DFA	Remplace l'état `p` par une copie conforme de l'état `q`.
`void change_trans(State q, const Alphabet &a, State p)`	DFA	Redirige la transition étiquetée par `q` et sortant de `q` vers `p`.
`State initial() const`	DFA	Renvoie l'identifiant de l'état initial, `null_state` s'il n'est pas défini.
`State delta1(State q, const Alphabet &a) const`	DFA	Renvoie le but de la transition étiquetée par `a` et sortant de `q`, `null_state` si elle n'est pas définie.
`Edges delta2(State q) const`	DFA	Renvoie un proxy permettant l'accès à l'ensemble des transitions sortant de l'état `q`.

Membre	Imposé par	Description
`Tag& tag(State q)`	DFA	Renvoie une référence sur le tag associé à l'état `q`.
`const Tag& tag(State q) const`	DFA	Renvoie une référence constante sur le tag associé à l'état `q`.
`bool& final(State q)`	DFA	Renvoie une référence sur un booléen permettant de choisir l'appartenance de l'état `q` à l'ensemble des états terminaux.
`bool final(State q) const`	DFA	Renvoie un booléen indiquant si l'état `q` appartient à l'ensemble des états terminaux.
`unsigned long state_count() const`	DFA	Renvoie le nombre d'états de l'automate.
`unsigned long trans_count() const`	DFA	Renvoie le nombre de transitions de l'automate.
`const_iterator begin() const`	DFA	Renvoie un itérateur sur le début de la séquence des états de l'automate.
`const_iterator end() const`	DFA	Renvoie un itérateur sur la fin de la séquence des états de l'automate.

Nouveaux membres

Ces membres ne sont pas imposés par le concept de DFA mais sont spécifiques au `DFA_matrix` :

Membre	Description
`DFA_matrix(unsigned long n)`	Constructeur réservant une partie de l'espace mémoire nécessaire à la création de `n` états : les `n` prochains appels à `new_state` seront plus efficaces.

Voir aussi

DFA, `DFA_map`, `DFA_bin`, `DFA_mtf`, `DFA_tr`, `DFA_hash`, `DFA_compact`, `DFA_min`.

DFA_map<Sigma, Tag>

Catégorie : Containers **Type de composant : Type**

Description

Un `dfa_map` est un container d'automate déterministe stockant les transitions des états dans des `map` STL permettant un accès en temps logarithmique. Cette représentation requiert une relation d'ordre sur les éléments de l'alphabet définie par la méthode `Sigma::lt`.

Définition

`dfa_map.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`Sigma`	Le trait de l'alphabet	`plain`
`Tag`	Le type des tags	`empty_tag`

Modèle de

DFA.

Contraintes de type

- `Sigma` est un modèle d'alphabet définissant `lt()`.
- `Tag` est constructible par défaut et assignable.

Classes de base publiques

`dfa_concept`.

Membres

Identiques à ceux du `DFA_matrix`.

Nouveaux membres

Identiques à ceux du `DFA_matrix`.

Voir aussi

DFA, `DFA_matrix`, `DFA_bin`, `DFA_mtf`, `DFA_tr`, `DFA_hash`, `DFA_compact`, `DFA_min`.

DFA_bin<Sigma, Tag>

Catégorie : Containers **Type de composant : Type**

Description

Un `DFA_bin` est un container d'automate déterministe stockant les transitions des états dans des vecteurs triés permettant un accès en temps logarithmique. L'insertion et la suppression d'une transition s'exécutent en temps linéaire, ce qui est moins efficace que pour le `DFA_map` mais cette représentation consomme moins d'espace mémoire. Son utilisation nécessite une relation d'ordre sur les éléments de l'alphabet définie par la méthode `Sigma::lt`.

Définition

`dfa_bin.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`Sigma`	Le trait de l'alphabet	`plain`
`Tag`	Le type des tags	`empty_tag`

Modèle de

DFA.

Contraintes de type

- `Sigma` est un modèle d'alphabet définissant `lt()`.
- `Tag` est constructible par défaut et assignable.

Classes de base publiques

`dfa_concept`.

Membres

Identiques à ceux du `DFA_matrix`.

Nouveaux membres

Identiques à ceux du `DFA_matrix`.

Voir aussi

DFA, `DFA_matrix`, `DFA_map`, `DFA_mtf`, `DFA_tr`, `DFA_hash`, `DFA_compact`, `DFA_min`.

DFA_mtf<Sigma, Tag>

Catégorie : Containers **Type de composant : Type**

Description

Un `DFA_mtf` est un container d'automate déterministe stockant les transitions des états dans des vecteurs. L'accès s'exécute en temps linéaire mais l'heuristique « move-to-front » accélère la recherche pour les transitions les plus utilisées. Cette représentation nécessite une relation d'équivalence sur les éléments de l'alphabet définie par la méthode `Sigma::eq`.

Définition

`dfa_mtf.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`Sigma`	Le trait de l'alphabet	`plain`
`Tag`	Le type des tags	`empty_tag`

Modèle de

DFA.

Contraintes de type

- `Sigma` est un modèle d'alphabet définissant `eq()`.
- `Tag` est constructible par défaut et assignable.

Classes de base publiques

`dfa_concept`.

Membres

Identiques à ceux du `DFA_matrix`.

Nouveaux membres

Identiques à ceux du `DFA_matrix`.

Voir aussi

DFA, `DFA_matrix`, `DFA_map`, `DFA_bin`, `DFA_tr`, `DFA_hash`, `DFA_compact`, `DFA_min`.

DFA_tr<Sigma, Tag>

Catégorie : Containers **Type de composant : Type**

Description

Un `DFA_tr` est un container d'automate déterministe stockant les transitions des états dans des vecteurs. L'accès s'exécute en temps linéaire mais l'heuristique « transpose » accélère la recherche pour les transitions les plus utilisées. Cette représentation nécessite une relation d'équivalence sur les éléments de l'alphabet définie par la méthode `Sigma::eq`.

Définition

`dfa_tr.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`Sigma`	Le trait de l'alphabet	`plain`
`Tag`	Le type des tags	`empty_tag`

Modèle de

DFA.

Contraintes de type

- `Sigma` est un modèle d'alphabet définissant `eq()`.
- `Tag` est constructible par défaut et assignable.

Classes de base publiques

`dfa_concept`.

Membres

Identiques à ceux du `DFA_matrix`.

Nouveaux membres

Identiques à ceux du `DFA_matrix`.

Voir aussi

DFA, `DFA_matrix`, `DFA_map`, `DFA_bin`, `DFA_mtf`, `DFA_hash`, `DFA_compact`, `DFA_min`.

DFA_hash<Sigma>

Catégorie : Containers **Type de composant : Type**

Description

Un `DFA_hash` est un container d'automate déterministe stockant les transitions des états dans une unique table de hachage. La complexité en espace ne dépend pas du nombre d'états mais du nombre de transitions et il n'est pas possible d'y stocker des tags. Cette représentation nécessite une fonction de hachage définie par `Sigma::to_int_type` (fonction déterministe associant un entier positif à tout élément de l'alphabet).

Définition

`dfa_hash.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`Sigma`	Le trait de l'alphabet	`plain`

Modèle de

DFA.

Contraintes de type

- `Sigma` est un modèle d'alphabet définissant `to_int_type`.

Classes de base publiques

`dfa_concept`.

Membres

Identiques à ceux du `DFA_matrix` hormis les méthodes d'accès aux tags.

Voir aussi

DFA, `DFA_matrix`, `DFA_map`, `DFA_bin`, `DFA_mtf`, `DFA_tr`, `DFA_compact`, `DFA_min`.

DFA_compact<DFA, Filter>

Catégorie : Containers **Type de composant : Type**

Description

Un `DFA_compact` est un container d'automate déterministe utilisant un unique tableau de transitions permettant un substantiel gain de place par rapport à la représentation matricielle classique. Il est construit par copie et ne supporte aucune opération à effet de bord car sa structure compressée est figée.

Définition

`dfa_compact.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`DFA`	Le type d'automate à compresser	
`Filter`	Le type d'un foncteur permettant la projection des tags de l'automate de départ vers un type arbitraire	`identity<DFA::Tag>`

Modèle de

DFA.

Contraintes de type

- `DFA` est un modèle de DFA avec tags.
- `Filter` est un modèle de fonction unaire adaptable (adaptable unary function) avec pour type d'argument `DFA::Tag`.

Classes de base publiques

`dfa_concept`.

Membres

Tous les membres constants du `DFA_matrix`.

Nouveaux membres

Ces membres ne sont pas imposés par le concept de DFA mais sont spécifiques au `DFA_compact` :

Membre	Description
`DFA_compact(const DFA &a)`	Construction d'un automate compact par copie de l'automate `a`.
`map(const string &p)`	Projection en mémoire de l'automate compact stocké sur disque dans le fichier `p`.
`unmap()`	Libération d'un automate précédemment projeté en mémoire.
`read(const string &p)`	Chargement de l'automate contenu dans le fichier `p`
`write(const string &p)`	Sauvegarde binaire dans le fichier `p`.

Notes

- Les fonctionnalités de projection en mémoire et de sauvegarde/chargement binaire impose au type de tag d'être un POD (Plain Old Data Type).
- La structure compacte non seulement les transitions mais aussi les tags associés aux états.

Voir aussi

DFA, `DFA_matrix`, `DFA_map`, `DFA_bin`, `DFA_mtf`, `DFA_tr`, `DFA_hash`, `DFA_min`.

DFA_min

Catégorie : Containers **Type de composant : Type**

Description

Un `DFA_min` est un container d'automate déterministe maintenant une structure minimale. Il n'est pas possible de créer ou de détruire des états et des transitions directement mais uniquement d'ajouter ou de supprimer des mots au langage reconnu. L'interface est limitée aux méthodes n'ayant pas d'effet de bord et les éléments de l'alphabet sont des caractères sur 8 bits.

Définition

`dfa_min.h`

Paramètres d'instanciation

Aucun.

Modèle de

DFA.

Classes de base publiques

`dfa_concept`.

Membres

Tous les membres constants du `DFA_matrix` hormis ceux concernant l'accès aux tags.

Nouveaux membres

Ces membres ne sont pas imposés par le concept de DFA mais sont spécifiques au `DFA_min` :

Membre	Description
`template <class InputIterator> insert(InputIterator i, InputIterator j)`	Ajout au langage reconnu du mot défini par l'intervalle `[i,j)`.
`insert(const char *s)`	Spécialisation de la méthode d'ajout pour les chaînes C.
`insert(const string &s)`	Spécialisation de la méthode d'ajout pour les chaînes C++ contenant des caractères sur 8 bits.

Voir aussi

DFA, `DFA_matrix`, `DFA_map`, `DFA_bin`, `DFA_mtf`, `DFA_tr`, `DFA_hash`, `DFA_compact`.

NFA_map<Sigma, Tag>

Catégorie : Containers **Type de composant : Type**

Description

Un `NFA_map` est un container d'automate non-déterministe stockant les transitions des états dans des `multimap` STL. Cette représentation nécessite une relation d'ordre sur les éléments de l'alphabet définie par la méthode `Sigma::lt`.
L'interface est la même que pour les automates déterministes, c'est le type des états qui les différencie.

Définition

`nfa_mmap.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`Sigma`	Le trait de l'alphabet	`plain`
`Tag`	Le type des tags	`empty_tag`

Modèle de

NFA.

Contraintes de type

- `Sigma` est un modèle d'alphabet définissant `lt()`.
- `Tag` est constructible par défaut et assignable.

Classes de base publiques

`nfa_concept`.

Membres

Identiques à ceux du `DFA_matrix`.

Nouveaux membres

Identiques à ceux du `DFA_matrix`.

Voir aussi

NFA, `NFA_matrix`, `NFA_epsilon`.

NFA_matrix<Sigma, Tag>

Catégorie : Containers **Type de composant : Type**

Description

Un `NFA_matrix` est un container d'automate non-déterministe stockant les transitions des états dans une matrice à trois dimensions. Il impose les mêmes contraintes à l'alphabet que le `DFA_matrix` (bijection alphabet ↔ entiers).

Définition

`nfa_matrix.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`Sigma`	Le trait de l'alphabet	`plain`
`Tag`	Le type des tags	`empty_tag`

Modèle de

NFA.

Contraintes de type

- `Sigma` est un modèle d'alphabet définissant `size`, `to_int_type`, `to_char_type`.
- `Tag` est constructible par défaut et assignable.

Classes de base publiques

`nfa_concept`.

Membres

Identiques à ceux du `DFA_matrix`.

Nouveaux membres

Identiques à ceux du `DFA_matrix`.

Voir aussi

NFA, `NFA_mmap`, `NFA_epsilon`.

NFA_epsilon<Sigma, Tag>

Catégorie : Containers **Type de composant : Type**

Description

Un `NFA_epsilon` est un container d'automate non-déterministe avec ϵ-transition (asynchrone). Son cadre d'utilisation est strictement réservé à la construction à partir d'une expression rationnelle par la méthode de Thompson : il ne peut sortir d'un état qu'au maximum deux ϵ-transitions et une transition normale.

Définition

`nfa_epsilon.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`Sigma`	Le trait de l'alphabet	`plain`
`Tag`	Le type des tags	`empty_tag`

Modèle de

NFA.

Contraintes de type

- `Sigma` est un modèle d'alphabet.
- `Tag` est constructible par défaut et assignable.

Classes de base publiques

`nfa_concept`.

Membres

Identiques à ceux du `DFA_matrix`.

Nouveaux membres

Ces membres ne sont pas imposés par le concept de NFA mais sont spécifiques au `NFA_epsilon` :

Membre	Description
`set_trans(State q, State p)`	Création d'une ϵ-transition entre les états `q` et `p`.
`set_trans(State q, State p1, State p2)`	Création de deux ϵ-transitions sortant de `q` et dirigées vers `p1` et `p2`.

Voir aussi

NFA, `NFA_mmap`, `NFA_matrix`.

cursor<DFA>

Catégorie : Curseur **Type de composant : Type**

Description

Un `cursor` est un accesseur d'automate : il permet aux algorithmes d'accéder à la structure du container sous-jacent en masquant le type réel de l'objet. Il s'agit en fait d'un pointeur sur l'état d'un automate dont les fonctionnalités se limitent aux parcours simples.

Définition

`cursor.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`DFA`	Le type de l'automate	

Modèle de

Curseur simple.

Contraintes de type

– `DFA` est un modèle de DFA.

Classes de base publiques

`cursor_concept`.

Membres

Membre	Imposé par	Description
`State`	curseur simple	Le type des identifiants d'états de l'automate sous-jacent, `DFA::State`.
`Alphabet`	curseur simple	Le type des lettres étiquetant les transitions de l'automate, `DFA::Alphabet`.
`Tag`	curseur simple	Le type des tags associés aux états, `DFA::Tag`.
`cursor()`	curseur simple	Construit un curseur singulier.
`cursor(const cursor &c)`	curseur simple	Constructeur de copie. Le curseur pointe sur le même automate et le même état que `c`.

Membre	Imposé par	Description
`State src() const`	curseur simple	Renvoie l'état sur lequel pointe le curseur.
`cursor& operator=(State q)`	curseur simple	Positionne le curseur sur l'état `q`.
`cursor& operator=( const cursor &c)`	curseur simple	Positionne le curseur sur l'état pointé par `c`.
`bool operator==( const cursor &c) const`	curseur simple	Renvoie vrai si `c` pointe sur le même état que le curseur.
`bool sink() const`	curseur simple	Renvoie vrai si le curseur pointe sur l'état nul de l'automate `null_state`.
`bool forward( const Alphabet &a)`	curseur simple	Avance sur la transition étiquetée par `a` sortant de l'état courant. Si la transition n'est pas définie, le curseur est envoyé sur l'état nul et prend une valeur singulière.
`bool exists( const Alphabet &a) const`	curseur simple	Renvoie vrai s'il existe une transition étiquetée par `a` sortant de l'état courant.
`bool src_final() const`	curseur simple	Renvoie vrai si l'état courant appartient à l'ensemble des états terminaux de l'automate.
`Tag tag() const`	curseur simple	Renvoie le tag de l'état courant.

Nouveaux membres

Ces membres ne sont pas imposés par le concept de curseur simple mais sont spécifiques au `cursor` :

Membre	Description
`cursor(const DFA &a)`	Construit un curseur singulier pointant sur l'automate `a`.
`cursor(const DFA &A, State q)`	Construit un curseur pointant sur l'état `q` de l'automate `A`.

Fonctions d'aide

```
template <class DFA>
cursor<DFA> plainc(const DFA &a);

template <class DFA>
cursor<DFA> plainc(const DFA &a, typename DFA::State q);
```

Notes

Un curseur pointant sur l'état nul, c'est-à-dire pour lequel `sink() == true` possède une valeur singulière : la seule opération valide est le repositionnement sur un état non nul de l'automate.

Voir aussi

curseur simple, `forward_cursor`, `stack_cusor`, `queue_cursor`.

forward_cursor<DFA>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `forward_cursor` est un pointeur sur une transition d'automate, c'est-à-dire un triplet (état source, lettre, état but). Il implémente toutes les fonctionnalités du curseur simple et permet en outre le parcours itératif des transitions sortant de l'état source.

Définition

`cursor.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`DFA`	Le type de l'automate	

Modèle de

curseur monodirectionnel.

Contraintes de type

– `DFA` est un modèle de DFA.

Classes de base publiques

`forward_cursor_concept`.

Membres

Membre	Imposé par	Description
`State`	curseur simple	Le type des identifiants d'états de l'automate sous-jacent, `DFA::State`.
`Alphabet`	curseur simple	Le type des lettres étiquetant les transitions de l'automate, `DFA::Alphabet`.
`Tag`	curseur simple	Le type des tags associés aux états, `DFA::Tag`.
`forward_cursor()`	curseur simple	Construit un curseur singulier.

Membre	**Imposé par**	**Description**
`forward_cursor( const forward_cursor &c)`	curseur simple	Constructeur de copie. Le curseur pointe sur le même automate et la même transition que `c`.
`State src() const`	curseur simple	Renvoie l'identifiant de l'état source sur lequel pointe le curseur.
`cursor& operator=(State q)`	curseur simple	Positionne l'état source du curseur sur `q`.
`cursor& operator=( const forward_cursor &c)`	curseur simple	Positionne le curseur sur la transition pointée par `c`.
`bool operator==( const forward_cursor &c) const`	curseur simple	Renvoie vrai si `c` pointe sur la même transition que le curseur.
`bool sink() const`	curseur simple	Renvoie vrai si l'état source pointé par le curseur est l'état nul `null_state`.
`bool forward(const Alphabet &a)`	curseur simple	Avance sur la transition étiquetée par `a` sortant de l'état source. Si la transition n'est pas définie, le curseur est envoyé sur l'état nul et prend une valeur singulière.
`bool exists( const Alphabet &a) const`	curseur simple	Renvoie vrai s'il existe une transition étiquetée par `a` sortant de l'état source.
`bool src_final() const`	curseur simple	Renvoie vrai si l'état source appartient à l'ensemble des états terminaux de l'automate.
`Tag tag() const`	curseur simple	Renvoie le tag de l'état source.
`State aim() const`	curseur monodirectionnel	Renvoie l'identifiant de l'état but. Le curseur doit avoir été positionné correctement au préalable.
`Alphabet letter() const`	curseur monodirectionnel	Renvoie la lettre de la transition courante. Le curseur doit avoir été positionné correctement au préalable.
`bool first_transition()`	curseur monodirectionnel	Positionne le curseur sur la première transition sortant de l'état source. Renvoie vrai si elle existe.
`bool next_transition()`	curseur monodirectionnel	Positionne le curseur sur la transition sortante suivante. Renvoie vrai si elle existe.

Membre	Imposé par	Description
`bool find(const Alphabet &a)`	curseur monodirectionnel	Positionne le curseur sur la transition étiquetée par `a` sortant de l'état source. Renvoie vrai si elle existe.
`void forward()`	curseur monodirectionnel	Avance sur la transition courante. Le curseur doit avoir été positionné correctement au préalable grâce à une des méthodes `first_transition`, `next_transition()` ou `find`.
`bool aim_final() const`	curseur monodirectionnel	Renvoie vrai si l'état but appartient à l'ensemble des états terminaux de l'automate. Le curseur doit avoir été positionné correctement au préalable.
`Tag aim_tag() const`	curseur monodirectionnel	Renvoie le tag associé à l'état but de la transition courante. Le curseur doit avoir été positionné correctement au préalable.

Nouveaux membres

Ces membres ne sont pas imposés par le concept de curseur monodirectionnel mais sont spécifiques au `forward_cursor` :

Membre	Description
`forward_cursor(const DFA &a)`	Construit un curseur singulier pointant sur l'automate `a`.
`forward_cursor(const DFA &A, State q)`	Construit un curseur pointant sur l'état `q` de l'automate `A`.
`forward_cursor(const DFA &A, State q, const Alphabet &a)`	Construit un curseur pointant sur la transition de source `q` et de lettre `a`. Le comportement n'est pas défini si la transition n'existe pas.

Fonctions d'aide

```
template <class DFA>
forward_cursor<DFA> forwardc(const DFA &a, typename DFA::State q);

template <class DFA>
forward_cursor<DFA> forwardc(const DFA &a);
```

Notes

Les appels aux méthodes s'appliquant à la lettre ou l'état but de la transition courante ne sont valides que si le curseur a été correctement positionné préalablement, c'est-à-dire lorsque l'une des méthodes `first_transition`, `next_transition` ou `find` a renvoyé vrai. Les appels

concernant l'état source ne sont valides que si le curseur a été construit ou positionné sur un état valide de l'automate (`sink() == false`).

Voir aussi

curseur simple, curseur monodirectionnel, `cursor`, `stack_cursor`, `queue_cursor`.

stack_cursor<ForwardCursor, Container>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `stack_cursor` est un curseur monodirectionnel stockant le chemin parcouru dans une pile de curseurs. Chaque avancée sur une transition empile le curseur courant et une méthode supplémentaire permet de le dépiler. Le curseur de parcours en profondeur `dfirst_cursor` repose sur le `stack_cursor`.

Définition

`cursor.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`ForwardCursor`	Le type de curseur à stocker dans la pile	
`Container`	Le container séquentiel implémentant la pile	`vector<ForwardCursor>`

Modèle de

curseur simple, curseur monodirectionnel, curseur pile.

Contraintes de type

- `ForwardCursor` est un modèle de curseur monodirectionnel.
- `Container` est un modèle de séquence à insertion en queue (back insertion sequence). Le type des objets stockés, `Container::value_type`, doit être `ForwardCursor`.

Classes de base publiques

`cursor_concept`, `forward_cursor_concept`, `stack_cursor_concept`.

Membres

Membre	Imposé par	Description
`State`	curseur simple	Le type des identifiants d'états du curseur sous-jacent, `ForwardCursor::State`.
`Alphabet`	curseur simple	Le type des lettres étiquetant les transitions , `ForwardCursor::Alphabet`.
`Tag`	curseur simple	Le type des tags associés aux états, `ForwardCursor::Tag`.
`stack_cursor()`	curseur simple	Construit un curseur singulier dont la pile est vide.
`stack_cursor( const stack_cursor &c)`	curseur simple	Constructeur de copie. Le curseur pointe sur le même automate et la même transition que `c`. La pile est recopiée.
`State src() const`	curseur simple	Renvoie l'identifiant de l'état source sur lequel pointe le curseur.
`cursor& operator=(State q)`	curseur simple	Positionne l'état source du curseur sur `q`.
`bool operator==( const stack_cursor &c) const`	curseur simple	Renvoie vrai si la pile de curseurs courante est égale à celle de `c`.
`bool sink() const`	curseur simple	Renvoie vrai si l'état source pointé par le curseur est l'état nul `null_state`.
`bool forward(const Alphabet &a)`	curseur simple	Avance sur la transition étiquetée par `a` sortant de l'état source et empile le curseur résultant. Si la transition n'est pas définie, le curseur est envoyé sur l'état nul et prend une valeur singulière.
`bool exists( const Alphabet &a) const`	curseur simple	Renvoie vrai s'il existe une transition étiquetée par `a` sortant de l'état source.
`bool src_final() const`	curseur simple	Renvoie vrai si l'état source appartient à l'ensemble des états terminaux de l'automate.
`Tag tag() const`	curseur simple	Renvoie le tag de l'état source.
`State aim() const`	curseur monodirectionnel	Renvoie l'identifiant de l'état but. Le curseur doit avoir été positionné correctement au préalable.

Membre	Imposé par	Description
`Alphabet letter() const`	curseur monodirectionnel	Renvoie la lettre de la transition courante. Le curseur doit avoir été positionné correctement au préalable.
`bool first_transition()`	curseur monodirectionnel	Positionne le curseur sur la première transition sortant de l'état source. Renvoie vrai si elle existe.
`bool next_transition()`	curseur monodirectionnel	Positionne le curseur sur la transition sortante suivante. Renvoie vrai si elle existe.
`bool find(const Alphabet &a)`	curseur monodirectionnel	Positionne le curseur sur la transition étiquetée par `a` sortant de l'état source. Renvoie vrai si elle existe.
`void forward()`	curseur monodirectionnel	Avance sur la transition courante et empile le curseur résultant. Le curseur doit avoir été positionné correctement au préalable grâce à une des méthodes `first_transition`, `next_transition()` ou `find`.
`bool aim_final() const`	curseur monodirectionnel	Renvoie vrai si l'état but appartient à l'ensemble des états terminaux de l'automate. Le curseur doit avoir été positionné correctement au préalable.
`Tag aim_tag() const`	curseur monodirectionnel	Renvoie le tag associé à l'état but de la transition courante. Le curseur doit avoir été positionné correctement au préalable.
`bool backward()`	curseur pile	Dépile le curseur du sommet et renvoie faux si la pile résultante est vide.

Nouveaux membres

Membre	Description
`stack_cursor(const ForwardCursor &c)`	Constructeur initialisant la pile du curseur avec `c`.

Fonctions d'aide

```
template <class ForwardCursor>
stack_cursor<ForwardCursor> stackc(const ForwardCursor &x);
```

Notes

Les appels aux méthodes du `stack_cursor` ne sont valides que lorsque la pile est non vide.

Voir aussi

curseur monodirectionnel, curseur pile, `forward_cursor`, `queue_cursor`.

queue_cursor<ForwardCusor, Container>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `queue_cursor` est un curseur monodirectionnel stockant les transitions traversées dans une file de curseurs. Une méthode supplémentaire permet de les défiler et d'implémenter le parcours en largeur.

Définition

`cursor.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`ForwardCursor`	Le type de curseur à stocker dans la file	
`Container`	Le container séquentiel implémentant la file	`deque<ForwardCursor>`

Modèle de

curseur simple, curseur monodirectionnel, curseur file.

Contraintes de type

- `ForwardCursor` est un modèle de curseur monodirectionnel.
- `Container` est un modèle de séquence à insertion en tête (front insertion sequence). Le type des objets stockés, `Container::value_type`, doit être `ForwardCursor`.

Classes de base publiques

`cursor_concept`, `forward_cursor_concept`, `queue_cursor_concept`.

Membres

Membre	Imposé par	Description
`State`	curseur simple	Le type des identifiants d'états de l'automate sous-jacent, `DFA::State`.
`Alphabet`	curseur simple	Le type des lettres étiquetant les transitions de l'automate, `DFA::Alphabet`.
`Tag`	curseur simple	Le type des tags associés aux états, `DFA::Tag`.
`queue_cursor()`	curseur simple	Construit un curseur singulier dont la file est vide.
`queue_cursor( const queue_cursor &c)`	curseur simple	Constructeur de copie. Le curseur pointe sur le même automate et la même transition que `c`. La file est recopiée.
`State src() const`	curseur simple	Renvoie l'identifiant de l'état source sur lequel pointe le curseur.
`cursor& operator=(State q)`	curseur simple	Positionne l'état source du curseur sur `q`.
`bool operator==( const queue_cursor &c) const`	curseur simple	Renvoie vrai si `c` pointe sur la même transition que le curseur.
`bool sink() const`	curseur simple	Renvoie vrai si l'état source pointé par le curseur est l'état nul `null_state`.
`bool forward(const Alphabet &a)`	curseur simple	Avance sur la transition étiquetée par `a` sortant de l'état source. Si la transition n'est pas définie, le curseur est envoyé sur l'état nul et prend une valeur singulière.
`bool exists( const Alphabet &a) const`	curseur simple	Renvoie vrai s'il existe une transition étiquetée par `a` sortant de l'état source.
`bool src_final() const`	curseur simple	Renvoie vrai si l'état source appartient à l'ensemble des états terminaux de l'automate.
`Tag tag() const`	curseur simple	Renvoie le tag de l'état source.
`State aim() const`	curseur monodirectionnel	Renvoie l'identifiant de l'état but. Le curseur doit avoir été positionné correctement au préalable.

Membre	Imposé par	Description
`Alphabet letter() const`	curseur monodirectionnel	Renvoie la lettre de la transition courante. Le curseur doit avoir été positionné correctement au préalable.
`bool first_transition()`	curseur monodirectionnel	Positionne le curseur sur la première transition sortant de l'état source et enfile le curseur. Renvoie vrai si elle existe.
`bool next_transition()`	curseur monodirectionnel	Positionne le curseur sur la transition sortante suivante et enfile le curseur. Renvoie vrai si elle existe.
`bool find(const Alphabet &a)`	curseur monodirectionnel	Positionne le curseur sur la transition étiquetée par `a` sortant de l'état source. Renvoie vrai si elle existe.
`void forward()`	curseur monodirectionnel	Avance sur la transition courante. Le curseur doit avoir été positionné correctement au préalable grâce à une des méthodes `first_transition`, `next_transition()` ou `find`.
`bool aim_final() const`	curseur monodirectionnel	Renvoie vrai si l'état but appartient à l'ensemble des états terminaux de l'automate. Le curseur doit avoir été positionné correctement au préalable.
`Tag aim_tag() const`	curseur monodirectionnel	Renvoie le tag associé à l'état but de la transition courante. Le curseur doit avoir été positionné correctement au préalable.
`bool dequeue()`	curseur file	Défile le curseur de tête qui devient le curseur courant.

Nouveaux membres

Membre	Description
`queue_cursor(const ForwardCursor &c)`	Constructeur initialisant la file du curseur avec `c`.

Fonctions d'aide

```
template <class ForwardCusor>
queue_cursor<ForwardCusor> queuec(const ForwardCusor &x);
```

Notes

Les appels aux méthodes du `queue_cursor` ne sont valides que lorsque la file est non vide.

Voir aussi

curseur monodirectionnel, curseur file, `forward_cursor`, `stack_cursor`.

dfirst_cursor<StackCursor>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `dfirst_cursor` implémente le parcours en profondeur. Une méthode permet l'incrémentation, c'est-à-dire le passage à la transition suivante sur la séquence des transitions dans l'ordre « profondeur d'abord ».

Définition

`cursor.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`StackCursor`	Le type du curseur pile sous-jacent	

Modèle de

curseur de parcours en profondeur (PEP).

Contraintes de type

– `StackCursor` est un modèle de curseur pile.

Classes de base publiques

`dfirst_cursor_concept`.

Membres

Membre	Imposé par	Description
`State`	curseur PEP	Le type des identifiants d'états de l'automate sous-jacent, `DFA::State`.
`Alphabet`	curseur PEP	Le type des lettres étiquetant les transitions de l'automate, `DFA::Alphabet`.
`Tag`	curseur PEP	Le type des tags associés aux états, `DFA::Tag`.
`dfirst_cursor()`	curseur PEP	Construit un curseur dont la pile est vide.
`dfirst_cursor( const dfirst_cursor &c)`	curseur PEP	Construit un curseur par copie.
`State src() const`	curseur PEP	Renvoie l'identifiant de l'état source sur lequel pointe le curseur.
`bool src_final() const`	curseur PEP	Renvoie vrai si l'état source appartient à l'ensemble des états terminaux de l'automate.
`Alphabet letter() const`	curseur PEP	Renvoie la lettre étiquetant la transition courante.
`bool aim() const`	curseur PEP	Renvoie l'identifiant de l'état but sur lequel pointe le curseur.
`bool aim_final() const`	curseur PEP	Renvoie vrai si l'état but appartient à l'ensemble des états terminaux de l'automate.
`bool operator==( const dfirst_cursor &c) const`	curseur PEP	Compare les piles des curseurs sous-jacent et renvoie vrai si elle sont égales.
`bool forward()`	curseur PEP	Incrémente le curseur d'une position sur la séquence des transitions. Renvoie vrai si le curseur a empilé une nouvelle transition ou faux s'il y a eu dépilement.
`Tag src_tag() const`	curseur PEP	Renvoie le tag associé à l'état source de la transition courante.
`Tag aim_tag() const`	curseur PEP	Renvoie le tag associé à l'état but de la transition courante.

Nouveaux membres

Membre	Description
`dfirst_cursor(const StackCursor &c)`	Construction d'un curseur ayant pour pile `c`.

Fonctions d'aide

La fonction `dfirstc` renvoie un `dfirst_cursor` construit à partir de l'objet passé en argument. Cet objet peut être un modèle de DFA, de curseur monodirectionnel ou de curseur pile.

Notes

Les appels aux méthodes autres que `operator==` ne sont valides que lorsque la pile du curseur est non vide.

Voir aussi

curseur de PEP, curseur de PEL, `dfirst_mark_cursor`, `bfirst_cursor`.

dfirst_mark_cursor<StackCursor, Marker>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `dfirst_mark_cursor` implémente le parcours en profondeur sur des automates cycliques. Une méthode permet l'incrémentation, c'est-à-dire le passage à la transition suivante sur la séquence des transitions dans l'ordre « profondeur d'abord » et une fonction de marquage garantit que le parcours s'arrête et que chaque transition est traversée exactement deux fois.

Définition

`cursor.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`StackCursor`	Le type du curseur pile sous-jacent	
`Marker`	Le type du foncteur de marquage d'état	`set_marker`

Modèle de

curseur de parcours en profondeur (PEP).

Contraintes de type

- `StackCursor` est un modèle de curseur pile.
- `Marker` est un modèle de foncteur de marquage d'état.

Classes de base publiques

`dfirst_cursor_concept`.

Membres

Identiques à ceux du `dfirst_cursor`.

Nouveaux membres

Membre	Description
`dfirst_cursor(const StackCursor &c)`	Construction d'un curseur ayant pour pile `c`.

Fonctions d'aide

La fonction `dfirstmarkc` renvoie un `dfirst_mark_cursor` construit à partir de l'objet passé en argument. Cet objet peut être un modèle de DFA, de curseur monodirectionnel ou de curseur pile.

Notes

Les appels aux méthodes autres que `operator==` ne sont valides que lorsque la pile du curseur est non vide.

Voir aussi

curseur de PEP, curseur de PEL, `dfirst_cursor`, `bfirst_cursor`.

bfirst_cursor<QueueCursor>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `bfirst_cursor` implémente le parcours en largeur. Une méthode permet l'incrémentation, c'est-à-dire le passage à la transition suivante sur la séquence des transitions dans l'ordre « largeur d'abord ».

Définition

`cursor.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`QueueCursor`	Le type du curseur file sous-jacent	

Modèle de

curseur de parcours en largeur (PEL).

Contraintes de type

– `QueueCursor` est un modèle de curseur file.

Classes de base publiques

`bfirst_cursor_concept`.

Membres

Membre	Imposé par	Description
`State`	curseur PEL	Le type des identifiants d'états de l'automate sous-jacent, `DFA::State`.
`Alphabet`	curseur PEL	Le type des lettres étiquetant les transitions de l'automate, `DFA::Alphabet`.
`Tag`	curseur PEL	Le type des tags associés aux états, `DFA::Tag`.
`bfirst_cursor()`	curseur PEL	Construit un curseur dont la file est vide.
`bfirst_cursor( const bfirst_cursor &c)`	curseur PEL	Construit un curseur par copie.
`State src() const`	curseur PEL	Renvoie l'identifiant de l'état source sur lequel pointe le curseur.
`bool src_final() const`	curseur PEL	Renvoie vrai si l'état source appartient à l'ensemble des états terminaux de l'automate.
`Alphabet letter() const`	curseur PEL	Renvoie la lettre étiquetant la transition courante.
`bool aim() const`	curseur PEL	Renvoie l'identifiant de l'état but sur lequel pointe le curseur.
`bool aim_final() const`	curseur PEL	Renvoie vrai si l'état but appartient à l'ensemble des états terminaux de l'automate.
`bool operator==( const bfirst_cursor &c) const`	curseur PEL	Compare les files des curseurs sous-jacent et renvoie vrai si elle sont égales.
`bool next_transition()`	curseur PEL	Incrémente le curseur d'une position sur la séquence des transitions. Renvoie vrai s'il reste des transitions.
`Tag src_tag() const`	curseur PEL	Renvoie le tag associé à l'état source de la transition courante.
`Tag aim_tag() const`	curseur PEL	Renvoie le tag associé à l'état but de la transition courante.

Nouveaux membres

Membre	Description
`bfirst_cursor(const QueueCursor &c)`	Construction d'un curseur dont la file contient `c`.

Fonctions d'aide

La fonction `bfirstc` renvoie un `bfirst_cursor` construit à partir de l'objet passé en ar-

gument. Cet objet peut être un modèle de DFA, de curseur monodirectionnel ou de curseur file.

Notes

Les appels aux méthodes autres que `operator==` ne sont valides que lorsque la file du curseur est non vide.

Voir aussi

curseur de PEL, curseur de PEP, `dfirst_cursor`.

intersection_cursor<ForwardCursor1, ForwardCursor2, LowerThan>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `intersection_cursor` est un adaptateur binaire de curseurs monodirectionnels. Il encapsule deux autres curseurs dont il calcule l'intersection à la volée.

Définition

`set_operation.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`ForwardCursor1` `ForwardCursor2`	Les types des curseurs dont l'intersection est calculée	
`LowerThan`	Un objet fonction implémentant un opérateur `<` servant à comparer les lettres des transitions des deux automates	`bool operator<( ForwardCursor1::Alphabet, ForwardCursor2::Alphabet)`

Modèle de

curseur monodirectionnel.

Contraintes de type

- `ForwardCursor1` et `ForwardCursor2` sont des modèles de curseur monodirectionnel. De plus les transitions de ces curseurs doivent être triées en ordre croissant selon la relation d'ordre définie sur les lettres.

Classes de base publiques

`forward_cursor_concept`.

Membres

Les membres sont identiques à ceux du `forward_cursor` exceptés :

Membre	Imposé par	Description
`State`	curseur simple	Le type des identifiants d'états de l'automate intersection, `pair<ForwardCursor1::State, ForwardCursor2::State>`.
`Alphabet`	curseur simple	Le type des lettres étiquetant les transitions de l'automate intersection, `ForwardCursor1::Alphabet`.
`Tag`	curseur simple	Le type des tags associés aux états, `pair<ForwardCursor1::Tag, ForwardCursor2::Tag>`.

Nouveaux membres

Membre	Description
`intersection_cursor( const ForwardCursor1 &x, const ForwardCursor2 &y)`	Construction d'un curseur intersection des deux curseurs `x` et `y`.

Fonctions d'aide

```
template <class ForwardCursor1, class ForwardCursor2>
intersection_cursor<ForwardCursor1, ForwardCursor2>
intersectionc(const ForwardCursor1 &x1, const ForwardCursor2 &x2);
```

Notes

La contrainte imposant des transitions triées permet le parcours des transitions sortant d'un état en temps linéaire.

Voir aussi

curseur monodirectionnel, `union_cursor`, `diff_cursor`, `sym_diff_cursor`, `not_cursor`, `concatenation_cursor`.

union_cursor<ForwardCursor1, ForwardCursor2, LessThan>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `union_cursor` est un adaptateur binaire de curseurs monodirectionnels. Il encapsule deux autres curseurs dont il calcule l'union à la volée.

Définition

`set_operation.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`ForwardCursor1` `ForwardCursor2`	Les types des curseurs dont l'union est calculée	
`LowerThan`	Un objet fonction implémentant un opérateur `<` servant à comparer les lettres des transitions des deux automates	`bool operator<( ForwardCursor1::Alphabet, ForwardCursor2::Alphabet)`

Modèle de

curseur monodirectionnel.

Contraintes de type

- `ForwardCursor1` et `ForwardCursor2` sont des modèles de curseur monodirectionnel. De plus les transitions de ces curseurs doivent être triées en ordre croissant selon la relation d'ordre définie sur les lettres.

Classes de base publiques

`forward_cursor_concept`.

Membres

Les membres sont identiques à ceux du `forward_cursor` exceptés :

Membre	Imposé par	Description
`State`	curseur simple	Le type des identifiants d'états de l'automate union, `pair<ForwardCursor1::State, ForwardCursor2::State>`.
`Alphabet`	curseur simple	Le type des lettres étiquetant les transitions de l'automate union, `ForwardCursor1::Alphabet`.
`Tag`	curseur simple	Le type des tags associés aux états, `pair<ForwardCursor1::Tag, ForwardCursor2::Tag>`.

Nouveaux membres

Membre	Description

`union_cursor(`
`  const ForwardCursor1 &x,`
`  const ForwardCursor2 &y)`
	Construction d'un curseur union des deux curseurs `x` et `y`.

Fonctions d'aide

```
template <class ForwardCursor1, class ForwardCursor2>
union_cursor<ForwardCursor1, ForwardCursor2>
unionc(const ForwardCursor1 &x1, const ForwardCursor2 &x2);
```

Notes

La contrainte imposant des transitions triées permet le parcours des transitions sortant d'un état en temps linéaire.

Voir aussi

curseur monodirectionnel, `intersection_cursor`, `diff_cursor`, `sym_diff_cursor`, `not_cursor`, `concatenation_cursor`.

diff_cursor<ForwardCursor1, ForwardCursor2>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `diff_cursor` est un adaptateur binaire de curseurs monodirectionnels. Il encapsule deux autres curseurs sur deux automates A et B dont il calcule la différence A B à la volée.

Définition

`set_operation.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`ForwardCursor1` `ForwardCursor2`	Les types des curseurs dont la différence est calculée	

Modèle de

curseur monodirectionnel.

Contraintes de type

– `ForwardCursor1` et `ForwardCursor2` sont des modèles de curseur monodirectionnel.

Classes de base publiques

`forward_cursor_concept`.

Membres

Les membres sont identiques à ceux du `forward_cursor` exceptés :

Membre	Imposé par	Description
`State`	curseur simple	Le type des identifiants d'états de l'automate différence, `pair<ForwardCursor1::State, ForwardCursor2::State>`.
`Alphabet`	curseur simple	Le type des lettres étiquetant les transitions de l'automate différence, `ForwardCursor1::Alphabet`.
`Tag`	curseur simple	Le type des tags associés aux états, `ForwardCursor1::Tag`.

Nouveaux membres

Membre	Description

`diff_cursor(`
`  const ForwardCursor1 &x,`
`  const ForwardCursor2 &y)`
	Construction d'un curseur différence des deux curseurs `x` et `y`.

Fonctions d'aide

```
template <class ForwardCursor1, class ForwardCursor2>
diff_cursor<ForwardCursor1, ForwardCursor2>
diffc(const ForwardCursor1 &x1, const ForwardCursor2 &x2);
```

Voir aussi

curseur monodirectionnel, `intersection_cursor`, `union_cursor`, `sym_diff_cursor`, `not_cursor`, `concatenation_cursor`.

sym_diff_cursor<ForwardCursor1, ForwardCursor2, LessThan>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `sym_diff_cursor` est un adaptateur binaire de curseurs monodirectionnels. Il encapsule deux autres curseurs sur deux automates A et B dont il calcule la différence symétrique $(A \cup B) \setminus (A \cap B)$ à la volée.

Définition

`set_operation.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`ForwardCursor1` `ForwardCursor2`	Les types des curseurs dont la différence symétrique est calculée	

`LowerThan`	Un objet fonction implémentant un opérateur `<` servant à comparer les lettres des transitions des deux automates	`bool operator<(`
		`ForwardCursor1::Alphabet,`
		`ForwardCursor2::Alphabet)`

Modèle de

curseur monodirectionnel.

Contraintes de type

- `ForwardCursor1` et `ForwardCursor2` sont des modèles de curseur monodirectionnel. De plus les transitions de ces curseurs doivent être triées en ordre croissant selon la relation d'ordre définie sur les lettres.

Classes de base publiques

`forward_cursor_concept`.

Membres

Les membres sont identiques à ceux du `forward_cursor` exceptés :

Membre	Imposé par	Description
`State`	curseur simple	Le type des identifiants d'états de l'automate différence symétrique, `pair<ForwardCursor1::State, ForwardCursor2::State>`.
`Alphabet`	curseur simple	Le type des lettres étiquetant les transitions de l'automate différence symétrique, `ForwardCursor1::Alphabet`.
`Tag`	curseur simple	Le type des tags associés aux états, `pair<ForwardCursor1::Tag, ForwardCursor2::Tag>`.

Nouveaux membres

Membre	Description

`sym_diff_cursor(`
`  const ForwardCursor1 &x,`
`  const ForwardCursor2 &y)`
	Construction d'un curseur différence symétrique des deux curseurs `x` et `y`.

Fonctions d'aide

```
template <class ForwardCursor1, class ForwardCursor2>
sym_diff_cursor<ForwardCursor1, ForwardCursor2>
sym_diffc(const ForwardCursor1 &x1, const ForwardCursor2 &x2);
```

Notes

La contrainte imposant des transitions triées permet le parcours des transitions sortant d'un état en temps linéaire.

Voir aussi

curseur monodirectionnel, `intersection_cursor`, `union_cursor`, `diff_cursor`, `not_cursor`, `concatenation_cursor`.

concatenation_cursor<ForwardCursor1, ForwardCursor2>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `concatenation_cursor` est un adaptateur binaire de curseurs monodirectionnels. Il encapsule deux autres curseurs sur deux automates A et B dont il calcule la la concaténation $A \cdot B$ à la volée.

Définition

`set_operation.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`ForwardCursor1` `ForwardCursor2`	Les types des curseurs dont la concaténation est calculée	

Modèle de

curseur monodirectionnel.

Contraintes de type

– `ForwardCursor1` et `ForwardCursor2` sont des modèles de curseur monodirectionnel.

Classes de base publiques

`forward_cursor_concept`.

Membres

Les membres sont identiques à ceux du `forward_cursor` exceptés :

Membre	Imposé par	Description
`State`	curseur simple	Le type des identifiants d'états de l'automate concaténation, `pair<ForwardCursor1::State, vector<ForwardCursor2::State> >`.
`Alphabet`	curseur simple	Le type des lettres étiquetant les transitions de l'automate concaténation, `ForwardCursor1::Alphabet`.
`Tag`	curseur simple	Le type des tags associés aux états, `ForwardCursor1::Tag`.

Nouveaux membres

Membre	Description

`concatenation_cursor(`
`  const ForwardCursor1 &x,`
`  const ForwardCursor2 &y)`
	Construction d'un curseur concaténation des deux curseurs `x` et `y`.

Fonctions d'aide

```
template <class ForwardCursor1, class ForwardCursor2>
concatenation_cursor<ForwardCursor1, ForwardCursor2>
concatenationc(const ForwardCursor1 &x1, const ForwardCursor2 &x2);
```

Voir aussi

curseur monodirectionnel, `intersection_cursor`, `union_cursor`, `diff_cursor`, `not_cursor`, `sym_diff_cursor`.

not_cursor<ForwardCursor, Sigma>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `not_cursor` est un adaptateur de curseur monodirectionnel unaire. Il encapsule un autre curseur sur un automate A et calcule son complémentaire $\Sigma^* \setminus A$.

Définition

`set_operation.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`ForwardCursor`	Le type du curseur dont le complémentaire est calculé	
`Sigma`	Le trait de l'alphabet de l'automate adapté	

Modèle de

curseur monodirectionnel.

Contraintes de type

- `ForwardCursor` est un modèle de curseur monodirectionnel.
- `Sigma` est un modèle d'alphabet.

Classes de base publiques

`forward_cursor_concept`.

Membres

Les membres sont identiques à ceux du `forward_cursor` exceptés :

Membre	Imposé par	Description
`State`	curseur simple	Le type des identifiants d'états de l'automate complémentaire, `ForwardCursor::State`.
`Alphabet`	curseur simple	Le type des lettres étiquetant les transitions de l'automate complémentaire, `ForwardCursor::Alphabet`.
`Tag`	curseur simple	Le type des tags associés aux états, `ForwardCursor::Tag`.

Nouveaux membres

Membre	Description
`not_cursor(const ForwardCursor &x)`	Construction d'un curseur complémentaire de `x`.

Fonctions d'aide

```
template <class ForwardCursor>
not_cursor<ForwardCursor> notc(const ForwardCursor &x);
```

Voir aussi

curseur monodirectionnel, `intersection_cursor`, `union_cursor`, `diff_cursor`, `sym_diff_cursor`, `concatenation_cursor`.

sigma_star_cursor<Sigma>

Catégorie : Curseurs **Type de composant : Type**

Description
Un `sigma_star_cursor` simule un curseur monodirectionnel sur l'automate reconnaissant le langage Σ^*. Cet automate virtuel ne comporte qu'un seul état initial et final.

Définition
`sigma_star.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`Sigma`	Le trait de l'alphabet	`plain`

Modèle de
curseur monodirectionnel.

Contraintes de type

- `Sigma` est un modèle d'alphabet.

Classes de base publiques
`forward_cursor_concept`.

Membres
Identiques à ceux du `forward_cursor`.

Nouveaux membres
Aucun.

Fonctions d'aide

```
template <class Sigma>
sigma_star_cursor<Sigma> sigma_starc();
```

Notes
La fonction d'aide doit être instanciée explicitement.

Voir aussi
curseur monodirectionnel.

permutation_cursor

Catégorie : Curseurs **Type de composant : Type**

Description
Un `permutation_cursor` simule l'automate reconnaissant toutes les permutations du mot 123...n.

Définition
`combinatory.h`

Paramètres d'instanciation
Aucun.

Modèle de
curseur monodirectionnel.

Classes de base publiques
`forward_cursor_concept`.

Membres
Identiques à ceux du `forward_cursor`.

Nouveaux membres

Membre	Description
`permutation_cursor(unsigned int n)`	Construction d'un curseur sur un automate virtuel reconnaissant les permutations du mot 123...n.

Fonctions d'aide

```
permutation_cursor permutationc(unsigned int n);
```

Notes
L'automate simulé est minimal.

Voir aussi
curseur monodirectionnel, `combination_cursor`.

debug_forward_cursor<DFA>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `debug_forward_cursor` est un curseur monodirectionnel classique implémentant des tests sur les invariants, les préconditions et les postconditions des appels de méthode. Lorsque les conditions ne sont pas remplies, un message est émis sur la sortie erreur standard.

Définition

`debug.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`DFA`	Le type de l'automate sous-jacent	

Modèle de

curseur monodirectionnel.

Contraintes de type

- `DFA` est un modèle de DFA.

Classes de base publiques

`forward_cursor_concept`.

Membres

Identiques à ceux du `forward_cursor`.

Nouveaux membres

Membre	Description
`debug_forward_cursor(const DFA &a)`	Construction d'un curseur sur l'automate `a`.
`debug_forward_cursor(const DFA &a, State q)`	Construction d'un curseur sur l'automate `a` pointant sur l'état `q`.

Fonctions d'aide

```
template <class DFA>
debug_forward_cursor<DFA> debugc(const DFA &a);
```

Notes

Idéalement, le curseur de déboguage devrait être un adaptateur, ce qui donnerait plus de souplesse à son utilisation. Malheureusement, l'encapsulation de la structure de donnée par le curseur adapté rend impossible les tests sur la validité des opérations.

Voir aussi

curseur monodirectionnel, `trace_forward_cursor`.

trace_forward_cursor<ForwardCursor>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `trace_forward_cursor` est un adaptateur de curseur monodirectionnel émettant sur la sortie standard un compte rendu de tous les appels de méthode le concernant, l'appel est ensuite transmis au curseur encapsulé.

Définition

`debug.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`ForwardCursor`	Le type du curseur adapté	

Modèle de

curseur monodirectionnel.

Contraintes de type

– `ForwardCursor` est un modèle de curseur monodirectionnel.

Classes de base publiques

`forward_cursor_concept`.

Membres

Identiques à ceux du `forward_cursor`.

Nouveaux membres

Membre	Description
`trace_forward_cursor(const ForwardCursor &x)`	Construction d'un curseur encapsulant `x`.

Fonctions d'aide

```
template <class ForwardCursor>
trace_forward_cursor<ForwardCursor> tracec(const ForwardCursor &x);
```

Voir aussi

curseur monodirectionnel, `debug_forward_cursor`.

def_trans_cursor<Cursor>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `def_trans_cursor` est un adaptateur de curseur simple dont la méthode d'incrémentation utilise une transition par défaut lorsque la transition requise n'est pas définie. La transition par défaut est étiquetée par une lettre spéciale (lettre par défaut) fournie à la construction de l'objet.

Définition

`default.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`Cursor`	Le type du curseur adapté	

Modèle de

curseur simple.

Contraintes de type

- `Cursor` est un modèle de curseur simple.

Classes de base publiques

`cursor_concept`.

Membres

Identiques à ceux du `cursor`.

Nouveaux membres

Membre	Description
`def_trans_cursor(const Cursor &x, const Cursor::Alphabet &a)`	Construction d'un curseur encapsulant `x` dont la lettre par défaut est `a`.

Fonctions d'aide

```
template <class Cursor>
def_trans_cursor<Cursor> def_transc(const Cursor &x, const Alphabet &a);
```

Voir aussi

curseur simple, `cursor`, `def_state_cursor`, `substitute_cursor`.

def_state_cursor<ForwardCursor>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `def_state_cursor` est un adaptateur de curseur monodirectionnel substituant à l'état nul du curseur adapté un état spécial (état par défaut) fourni à la construction de l'objet.

Définition

`default.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`ForwardCursor`	Le type du curseur adapté	

Modèle de

curseur monodirectionnel.

Contraintes de type

– `ForwardCursor` est un modèle de curseur monodirectionnel.

Classes de base publiques

`forward_cursor_concept`.

Membres

Identiques à ceux du `forward_cursor`.

Nouveaux membres

Membre	Description
`def_state_cursor(const ForwardCursor &x, const State &q)`	Construction d'un curseur encapsulant `x` ayant pour état par défaut `q`.

Fonctions d'aide

```
template <class ForwardCursor>
def_state_cursor<ForwardCursor>
def_statec(const ForwardCursor &x, const ForwardCursor::State &p);
```

Voir aussi

curseur monodirectionnel, `forward_cursor`, `def_trans_cursor`, `substitute_cursor`.

substitute_cursor<ForwardCursor>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `substitute_cursor` est un adaptateur de curseur monodirectionnel imposant aux tags de l'automate sous-jacent de contenir un état appelé substitut. Lorsqu'une transition requise n'est pas définie, le curseur se positionne sur le substitut de l'état courant et réitère l'opération.

Définition

`default.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`ForwardCursor`	Le type du curseur adapté	

Modèle de

curseur monodirectionnel.

Contraintes de type

- `ForwardCursor` est un modèle de curseur monodirectionnel.

Classes de base publiques

`forward_cursor_concept`.

Membres

Identiques à ceux du `forward_cursor`.

Nouveaux membres

Membre	Description
`substitute_cursor(const ForwardCursor &x)`	Construction d'un curseur encapsulant `x`.

Fonctions d'aide

```
template <class ForwardCursor>
substitute_cursor<ForwardCursor> substitutec(const ForwardCursor &x);
```

Notes

- Seule l'acyclicité du graphe des substituts garantit l'arrêt de la méthode `forward`.
- Ce genre de curseur est typiquement utilisé dans l'implémentation de l'algorithme de recherche de mots d'Aho et Corasick.

Voir aussi

curseur monodirectionnel, `forward_cursor`, `def_trans_cursor`, `def_state_cursor`.

filter_in_cursor<ForwardCursor, Filter>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `filter_in_cursor` est un adaptateur de curseur monodirectionnel appliquant une fonction arbitraire `Filter` aux arguments des méthodes de type `Alphabet` avant de transmettre l'appel au curseur encapsulé. Cette fonctionnalité permet la réalisation de projection d'alphabets.

Définition

`filter.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`ForwardCursor`	Le type du curseur adapté	
`Filter`	Le type d'un foncteur servant de filtre avant les appels aux méthodes	

Modèle de

curseur monodirectionnel.

Contraintes de type

- `ForwardCursor` est un modèle de curseur monodirectionnel.
- `Filter` est un modèle de fonction unaire adaptable (adaptable unary function) dont le type de retour `Filter::result_type` est `ForwardCursor::Alphabet`.

Classes de base publiques

`forward_cursor_concept`.

Membres

Identiques à ceux du `forward_cursor`.

Nouveaux membres

Membre	Description
`filter_in_cursor(const ForwardCursor &x, const Filter &f = Filter())`	Construction d'un curseur encapsulant `x` ayant pour fonction de filtre `f`.

Fonctions d'aide

```
template <class ForwardCursor, class Filter>
filter_in_cursor<ForwardCursor, Filter>
filter_inc(const ForwardCursor &x, const Filter &f = Filter());
```

Voir aussi

curseur monodirectionnel, `forward_cursor`, `filter_out_cursor`.

filter_out_cursor<ForwardCursor, Filter>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `filter_out_cursor` est un adaptateur de curseur monodirectionnel appliquant une fonction arbitraire `Filter` aux lettres renvoyées par la méthode `letter`. Ce type de curseur permet la réalisation de projection d'alphabets.

Définition

`filter.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`ForwardCursor`	Le type du curseur adapté	
`Filter`	Le type d'un foncteur servant de filtrant la valeur de retour de la méthode `letter`	

Modèle de

curseur monodirectionnel.

Contraintes de type

- `ForwardCursor` est un modèle de curseur monodirectionnel.
- `Filter` est un modèle de fonction unaire adaptable (adaptable unary function) dont le type d'argument `Filter::argument_type` est `ForwardCursor::Alphabet`.

Classes de base publiques

`forward_cursor_concept`.

Membres

Identiques à ceux du `forward_cursor`.

Nouveaux membres

Membre	Description
`filter_out_cursor(const ForwardCursor &x, const Filter &f = Filter())`	Construction d'un curseur encapsulant `x` ayant pour fonction de filtre `f`.

Fonctions d'aide

```
template <class ForwardCursor, class Filter>
filter_out_cursor<ForwardCursor, Filter>
filter_outc(const ForwardCursor &x, const Filter &f = Filter());
```

Voir aussi

curseur monodirectionnel, `forward_cursor`, `filter_in_cursor`.

lazy_cursor<Cursor, DFA>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `lazy_cursor` est un adaptateur de curseur simple construisant un automate au fur et à mesure de son avancée sur les transitions du curseur adapté. Si une transition requise existe déjà dans l'automate cache, le curseur la suit, sinon elle est créée et les appels subséquents à la méthode `forward` sur cette transition ne transmettront plus l'appel au curseur encapsulé, économisant ainsi du temps de calcul. De cette manière, l'automate est construit incrémentalement et seules les transitions nécessaires sont créées, économisant ainsi de l'espace mémoire.

Définition

`lazy.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`Cursor`	Le type du curseur adapté	
`DFA`	Le type de l'automate servant de cache	

Modèle de

curseur simple.

Contraintes de type

- `Cursor` est un modèle de curseur simple.
- `DFA` est un modèle de DFA avec `DFA::Tag` de type `Cursor::State` et `DFA::Alphabet` de type `Cursor::Alphabet`.

Classes de base publiques

`cursor_concept`.

Membres

Identiques à ceux du `cursor`.

Nouveaux membres

Membre	Description
`lazy_cursor(const Cursor &c, DFA &A)`	Construction d'un curseur encapsulant `c` et ayant pour automate cache `A`.

Fonctions d'aide

```
template <class Cursor, class DFA>
lazy_cursor<Cursor, DFA> lazyc(const Cursor &x, DFA &a);
```

Notes

La raison pour laquelle le `lazy_cursor` est un modèle de curseur simple et non monodirectionnel est qu'un tel mécanisme n'a pas d'intérêt pour le parcours des transitions sortantes car dans ce cas (parcours en profondeur ou en largeur par exemple) l'algorithme `ccopy` est plus efficace.

Voir aussi

curseur simple, `cursor`, `ccopy`, `clone`.

neighbor_cursor<ForwardCursor>

Catégorie : Catégorie **Type de composant : Type**

Description

Un `neighbor_cursor` est un curseur pile masquant les parties de l'automate sous-jacent qui s'éloigne trop d'une distance donnée du mot de référence. La distance est appelée distance d'édition et est définie par un nombre d'insertions, de suppressions et de substitutions.

Définition

`neighbor.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`ForwardCursor`	Le type du curseur stocké dans la pile	

Modèle de

curseur pile.

Contraintes de type

- `ForwardCursor` est un modèle de curseur monodirectionnel.

Classes de base publiques

`stack_cursor_concept`.

Membres

Identiques à ceux du `stack_cursor`.

Nouveaux membres

Membre	Description

`template <class InputIterator>`
`neighbor_cursor(const ForwardCursor &x,`
`InputIterator first`
`InputIterator last,`
`int d)`
	Construction d'un curseur dont la pile contient `x`, ayant pour mot de référence l'intervalle `[first, last)` et pour distance d'édition limite `d`

Fonctions d'aide

```
template <class ForwardCursor, class InputIterator>
neighbor_cursor<ForwardCursor>
neighborc(const ForwardCursor &x,
          InputIterator first, InputIterator last, int d);
```

```
template <class ForwardCursor>
neighbor_cursor<ForwardCursor>
neighborc(const ForwardCursor &x, const char *w, int d);
```

Voir aussi

curseur pile, `stack_cursor`, `forward_cursor`.

regexp_cursor

Catégorie : Curseurs **Type de composant : Type**

Description

Un `regexp_cursor` est un curseur simple simulant l'automate reconnaissant une expression rationnelle. Cette expression est définie sur des caractères 8 bits.

Définition

`regexp.h`

Paramètres d'instanciation

Aucun.

Modèle de

curseur simple.

Classes de base publiques

`cursor_concept`.

Membres

Identiques à ceux du `cursor`.

Nouveaux membres

Membre	Description
`regexp_cursor(const char *e)`	Construction d'un curseur à partir de l'expression rationnelle `e`.

Fonctions d'aide

```
regexp_cursor regexpc(const char *e);
```

Notes

L'utilisation directe du `regexp_cursor` est déconseillée car les complexités de calcul lors de la recherche sont du même ordre que celles associées à la gestion d'un automate non-déterministe. L'utilisation d'un curseur de construction paresseuse `lazy_cursor` permet de pallier ce manque d'efficacité.

Voir aussi

curseur simple, `lazy_cursor`.

string_cursor<ForwardIterator>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `string_cursor` est un curseur monodirectionnel simulant l'automate plat reconnaissant un mot défini par un intervalle d'itérateurs. Le patron est instanciable avec le type `char*` car il existe une version spécialisée du curseur pour les chaînes de caractères C.

Définition

`string_cursor.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`ForwardIterator`	Le type d'itérateur matérialisant les bornes de début et de fin du mot. Cet itérateur peut être un pointeur sur une chaîne C, c'est-à-dire de type `char*` car il existe une version spécialisée du curseur.	

Modèle de

curseur monodirectionnel.

Contraintes de type

– `ForwardIterator` est un modèle d'itérateur monodirectionnel (forward iterator).

Classes de base publiques

`forward_cursor_concept`.

Membres

Identiques à ceux du `forward_cursor`.

Nouveaux membres

Membre	Description
`string_cursor(ForwardIterator first, ForwardIterator last)`	Construction un curseur sur le mot défini par la séquence `[first, last)`.
`string_cursor(const char *s)`	Construction un curseur sur le mot défini par la chaîne C `s`. Ce constructeur n'est disponible que pour la version spécialisée du patron `string_cursor<char*>`.

Fonctions d'aide

```
template <class ForwardIterator>
string_cursor<ForwardIterator>
stringc(ForwardIterator first, ForwardIterator last);

string_cursor<char*> stringc(const char *s);
```

Voir aussi

curseur monodirectionnel.

hash_cursor<DFA>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `hash_cursor` est un curseur monodirectionnel calculant à la volée la valeur de hachage du mot parcouru. L'automate doit avoir été préparé au préalable avec l'algorithme `hash`.

Définition

`hash.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`DFA`	Le type d'automate	

Modèle de

curseur monodirectionnel.

Contraintes de type

- `DFA` est un modèle de DFA.
- `DFA::Tag` doit définir une méthode `int hash() const`.

Classes de base publiques

`forward_cursor<DFA>`.

Membres

Identiques à ceux du `forward_cursor`.

Nouveaux membres

Membre	Description
`hash_cursor(const DFA &A)`	Construction d'un curseur sur l'automate `A`.
`hash_cursor(const DFA &A, State q)`	Construction d'un curseur sur l'automate `A` pointant sur l'état `q`.
`int hash() const`	Renvoie la valeur de hachage courante.

Fonctions d'aide

```
template <class DFA>
hash_cursor<DFA> hashc(const DFA &A);
```

Notes

La méthode `hash()` renvoie une valeur de hachage calculée par l'algorithme `hash`. Les tags des états de l'automate contiennent une valeur entière représentant le nombre de mots reconnus par l'état. Cette valeur est calculée par l'algorithme `hash`.

Voir aussi

curseur monodirectionnel, `forward_cursor`, `hash`.

closure_cursor<ForwardCursor>

Catégorie : Curseurs **Type de composant : Type**

Description

Un `closure_cursor` est un adaptateur de curseur monodirectionnel masquant toutes les transitions qui ne sont pas étiquetées par la lettre spécifiée à la construction de l'objet.

Définition

`closure.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`ForwardCursor`	Le type du curseur adapté	

Modèle de

curseur monodirectionnel.

Contraintes de type

– `ForwardCursor` est un modèle de curseur monodirectionnel.

Classes de base publiques

`forward_cursor_concept`.

Membres

Identiques à ceux du `forward_cursor`.

Nouveaux membres

Membre	Description
`closure_cursor(const ForwardCursor &c, const Alphabet &a)`	Construction d'un curseur encapsulant `c` dont la lettre spéciale est `a`.

Fonctions d'aide

```
template <class ForwardCursor>
closure_cursor<ForwardCursor>
closurec(const ForwardCursor &c, const ForwardCursor::Alphabet &a);
```

Notes

Ce curseur permet entre autre de calculer l'epsilon-clôture d'un état en utilisant un curseur de parcours en profondeur.

Voir aussi

curseur monodirectionnel, `forward_cursor`.

istream_cursor

Catégorie : Curseurs **Type de composant : Type**

Description

Un `istream_cursor` est un curseur de parcours en profondeur lisant la séquence de transitions sur un flux. Il permet de charger un automate sauvegardé en utilisant l'algorithme `clone` ou d'appliquer un algorithme à la volée sans avoir à reconstruire l'automate.

Définition

`stream.h`

Paramètres d'instanciation

Aucun.

Modèle de

curseur de PEP.

Classes de base publiques

`dfirst_cursor_concept`.

Membres

Identiques à ceux du `dfirst_cursor`.

Nouveaux membres

Membre	Description
`istream_cursor(istream &in)`	Construction d'un curseur sur le flux d'entrée `in`.

Fonctions d'aide

```
istream_cursor istreamc(istream &in);
```

Notes

Ce curseur de lecture sur flux a des capacités limitées : il ne permet pas pour l'instant (début 2002) de lire un automate sauvegardé grâce à l'algorithme `ccopy` ni de recharger les tags des états.

Voir aussi

curseur de PEP, `dfirst_cursor`, `clone`.

DFA_dot

Catégorie : Outils **Type de composant : Type**

Description

Un `DFA_dot` implémente un sous-ensemble des méthodes du DFA et permet l'écriture sur flux d'un automate au format graphviz grâce aux algorithmes `ccopy` et `clone` auquel il est passé en premier argument.

Définition

`dot.h`

Paramètres d'instanciation

Aucun.

Modèle de

copy constructible, assignable.

Classes de base publiques

Aucune.

Membres

Membre	Imposé par	Description
`State`	DFA	Le type des identifiants d'états, `unsigned long`.
`null_state`	DFA	La valeur de l'état nul de type `State`.
`State new_state()`	DFA	Crée un nouvel état
`bool_ref`	DFA	Un type de référence sur un booléen servant à la méthode `final`.

`template <class Alphabet>`
`void set_trans(State q,`
`Alphabet a, State p)`
	DFA	Envoie sur le flux une représentation ASCII de la transition.

`bool_ref final(State q)`	DFA	Renvoie une référence sur un booléen permettant de positionner l'appartenance de `q` à l'ensemble des états terminaux.

Nouveaux membres

Les méthodes spécifiques au `DFA_dot` servent majoritairement à configurer l'aspect et la présentation du graphique produit :

Membre	Description
`DFA_dot(ostream &out)`	Construction d'un automate sur le flux de sortie `out`.
`DFA_dot& ratio(float x)`	Positionne le rapport hauteur/largeur du graphique.
`DFA_dot& states_label(const string &x)`	Par défaut, les états sont numérotés dans l'ordre du parcours ; cette méthode permet de fixer la chaîne de caractères affichées à l'intérieur des états.
`DFA_dot& state_fontsize(unsigned long x)`	Fixe la taille de fonte des labels d'état.
`DFA_dot& edge_fontsize(unsigned long x)`	Fixe la taille de fonte des étiquettes des transitions.
`DFA_dot& rankdir(const string &s)`	Fixe l'orientation du graphique.
`DFA_dot& ranksep(float f)`	Fixe l'espacement entre les hauteurs d'état.

Notes

Les propriétés que vérifie un tel objet représentent l'ensemble minimal de méthodes et de types nécessaire à l'application des algorithmes `ccopy` et `clone`.

Voir aussi

`dot`, `ccopy`, `clone`, `DFA_stream`, `DFA_stats`.

DFA_stream

Catégorie : Outils **Type de composant : Type**

Description

Un `DFA_stream` implémente un sous-ensemble des méthodes du DFA et permet l'écriture sur flux d'un automate sous forme ASCII grâce aux algorithmes `ccopy` et `clone` auquel il est passé en premier argument. Le curseur de parcours en profondeur `istream_cursor` permet de le relire.

Définition

`stream.h`

Paramètres d'instanciation

Aucun.

Modèle de

copy constructible, assignable.

Membres

Membre	Imposé par	Description
`State`	DFA	Le type des identifiants d'états, `unsigned long`.
`null_state`	DFA	La valeur de l'état nul de type `State`.
`State new_state()`	DFA	Crée un nouvel état
`bool_ref`	DFA	Un type de référence sur un booléen servant à la méthode `final`.

`template <class Alphabet>`
`void set_trans(State q,`
`Alphabet a, State p)`
	DFA	Envoie sur le flux une représentation ASCII de la transition.

`bool_ref final(State q)`	DFA	Renvoie une référence sur un booléen permettant de positionner l'appartenance de `q` à l'ensemble des états terminaux.

Nouveaux membres

Membre	Description
`DFA_stream(ostream &out)`	Construction d'un automate sur le flux de sortie `out`.

Voir aussi

`DFA_dot`, `DFA_stats`, `ccopy`, `clone`, `istream_cursor`.

DFA_stats

Catégorie : Outils **Type de composant : Type**

Description

Un `DFA_stats` implémente un sous-ensemble des méthodes du DFA permettant le calcul des nombres d'états et de transitions d'un automate grâce aux algorithmes `ccopy` et `clone` auquel il est passé en premier argument.

Définition

`stats.h`

Paramètres d'instanciation

Aucun.

Modèle de

copy constructible, assignable.

Membres

Membre	Imposé par	Description
`State`	DFA	Le type des identifiants d'états, `unsigned long`.
`null_state`	DFA	La valeur de l'état nul de type `State`.
`DFA_stats()`	DFA	Constructeur par défaut.
`State new_state()`	DFA	Incrémente le compteur d'états.
`bool_ref`	DFA	Un type de référence sur un booléen servant à la méthode `final`.

`template <class Alphabet>`
`void set_trans(State q,`
`Alphabet a, State p)`
	DFA	Incrémente le compteur de transitions.

`bool_ref final(State q)`	DFA	Incrémente le compteur d'états terminaux.
`unsigned long state_count() const`	DFA	Renvoie le nombre d'états de l'automate.
`unsigned long trans_count() const`	DFA	Renvoie le nombre de transitions de l'automate.

Nouveaux membres

Membre	Description
`unsigned long final_count() const`	Renvoie le nombre d'états terminaux de l'automate.

Voir aussi

`DFA_dot`, `DFA_stream`, `ccopy`, `clone`.

language_iterator<ForwardCursor>

Catégorie : Itérateurs **Type de composant : Type**

Description

Un `language_iterator` est un itérateur d'entrée (input iterator) et un adaptateur de curseur monodirectionnel. Il encapsule un curseur de PEP et permet l'extraction incrémentale du langage reconnu par un DFA acyclique. A chaque incrémentation, le mot suivant est extrait dans l'ordre du parcours en profondeur et est rendu disponible sous forme de `string` à travers l'opérateur de déréférencement `*`.

Définition

`language.h`

Paramètres d'instanciation

Paramètre	Description	Valeur par défaut
`ForwardCursor`	Le type du curseur adapté	

Modèle de

input iterator.

Contraintes de type

- `ForwardCursor` est un modèle de curseur monodirectionnel.
- `ForwardCursor::Alphabet` est de type `char`.

Classes de base publiques

`iterator<input_iterator_tag, string>`.

Membres

Membre	Imposé par	Description
`value_type`	trivial iterator	Le type des objets de la séquence, ici `string`.
`distance_type`	input iterator	Un type entier signé servant à représenter la distance entre deux itérateurs.
`language_iterator()`	trivial iterator	Constructeur par défaut.
`const string& operator*() const`	trivial iterator	Déréférencement.
`language_iterator& operator++()`	input iterator	Incrémentation préfixe. Passe au mot reconnu suivant dans l'ordre du parcours en profondeur.
`language_iterator operator++(int)`	input iterator	Incrémentation postfixe.

Nouveaux membres

Membre	Description
`language_iterator(const ForwardCursor &c)`	Construction d'itérateur encapsulant le curseur `c`.

Voir aussi

curseur monodirectionnel, `forward_cursor`, input iterator.

A.3 Algorithmes

La description des algorithmes comprend les rubriques suivantes :

1. **Catégorie** : algorithme.
2. **Type de composant** : fonction.
3. Le **prototype** de la fonction.
4. Sa **description**.
5. **Définition** : le fichier source contenant le code de l'algorithme.
6. Les **contraintes de type** s'appliquant aux paramètres de la fonction.
7. Les **préconditions** que doivent vérifier les arguments d'appel.
8. La **complexité** de calcul.
9. Un **exemple** de code.
10. **Notes :** remarques diverses.
11. **Voir aussi :** pointeurs vers les algorithmes liés.

ccopy

Catégorie : Algorithme **Type de composant : Fonction**

Prototype

```
template <class DFirstCursor, class DFA>
DFA::State ccopy(DFA &out, DFirstCursor first,
                           DFirstCursor last = DFirstCursor());
```

Description

`ccopy` (cursor copy) copie l'automate défini par l'intervalle de curseurs de PEP `[first,last)` vers le container `out`. Les transitions et les états sont copiés pendant la phase ascendante du parcours et les chemins ne menant pas à des états terminaux sont ignorés. Par défaut, la condition d'arrêt de l'algorithme `last` est la pile vide.
`ccopy` renvoie l'identifiant de la copie de l'état initial du parcours. Aucune opération concernant l'état initial de l'automate n'est effectuée car la généricité de l'algorithme permet d'appliquer la copie à un sous-automate sans état initial.

Définition

`ccopy.h`

Contraintes de type

- `DFirstCursor` est un modèle de curseur de PEP.
- `DFA` est un modèle de DFA.
- `DFirstCursor::Alphabet` doit être convertible en `DFA::Alphabet`.

Préconditions

- `[first, last)` est un intervalle valide.

Complexité

Soit $n =$ `last - first`. La complexité en temps est $n \log(n)$.

Exemple

```
DFA_matrix<> src;
DFA_map<>    dest;
DFA_map<>::State i = ccopy(dest, dfirstc(src));
dest.initial(i);
```

Notes

- En fait, cet algorithme n'impose au type `DFA` que de définir les méthodes `final`, `set_trans`, `new_state` et la valeur `null_state`.
- L'algorithme ne détecte pas les cycles : pour copier un automate cyclique il faut utiliser un type de curseur de PEP avec marquage des états.

Voir aussi

`clone`, DFA, curseur de PEP, curseur de PEP avec marquage.

clone

Catégorie : Algorithme **Type de composant : Fonction**

Prototype

```
template <class DFirstCursor, class DFA>
DFA::State clone(DFA &out, DFirstCursor first,
                           DFirstCursor last = DFirstCursor());
```

Description

`clone` copie l'automate défini par l'intervalle de curseurs de PEP `[first, last)` vers le container `out`. Les transitions et les états sont copiés pendant la phase descendante du parcours et une copie exacte de l'automate de départ est effectuée. Par défaut, la condition d'arrêt de l'algorithme `last` est la pile vide.
`clone` renvoie l'identifiant de la copie de l'état initial du parcours. Aucune opération concernant l'état initial de l'automate n'est effectuée car la généricité de l'algorithme permet d'appliquer la copie à un sous-automate sans état initial.

Définition

`ccopy.h`

Contraintes de type

- `DFirstCursor` est un modèle de curseur de PEP.
- `DFA` est un modèle de DFA.
- `DFirstCursor::Alphabet` doit être convertible en `DFA::Alphabet`.

Préconditions

- `[first, last)` est un intervalle valide.

Complexité

Soit $n =$ `last - first`. La complexité en temps est $n \log(n)$.

Exemple

```
\begin{verbatim}
DFA_matrix<> src;
DFA_map<>    dest;
DFA_map<>::State i = clone(dest, dfirstc(src));
dest.initial(i);
```

Notes

- En fait, cet algorithme n'impose au type `DFA` que de définir les méthodes `final`, `set_trans`, `new_state` et la valeur `null_state`.
- L'algorithme ne détecte pas les cycles, pour copier un automate cyclique il faut utiliser un type de curseur de PEP avec marquage des états.

Voir aussi

`ccopy`, DFA, curseur de PEP, curseur de PEP avec marquage.

is_in

Catégorie : Algorithme **Type de composant : Fonction**

Prototype

```
template <class InputIterator, class Cursor>
bool is_in(InputIterator first, InputIterator last, Cursor c);
```

Description

L'algorithme `is_in` teste l'appartenance d'un mot défini par l'intervalle d'itérateurs `[first, last)` au langage reconnu par l'automate pointé par le curseur `c`.

Définition

`language.h`

Contraintes de type

- `InputIterator` est un modèle d'itérateur d'entrée (input iterator).
- `Cursor` est un modèle de curseur simple.
- `InputIterator::value_type` doit être convertible en `Cursor::Alphabet`.

Préconditions

- `[first, last)` est un intervalle valide.
- `c` est positionné sur un état valide de l'automate ou sur l'état puits.

Complexité

Linéaire : au plus `last - first` appels à la méthode `Cursor::forward`.

Exemple

```
DFA_bin<> dfa;
istream_iterator<char> first(cin), last;
if (is_in(first, last, plainc(dfa)))
  cout << "reconnu";
```

Notes

L'algorithme ne tient pas compte du mot vide, i.e. même si l'intervalle est nul et que l'état initial de l'automate est terminal, `is_in` ne le reconnaîtra pas.

Voir aussi

itérateur d'entrée (input iterator), curseur simple.

language

Catégorie : Algorithme **Type de composant : Fonction**

Prototype

```
template <class DepthFirstCursor>
void language(ostream &out, DepthFirstCursor first,
                            DepthFirstCursor last = DepthFirstCursor());
```

Description

L'algorithme `language` émet sur le flux de sortie `out` l'ensemble des mots reconnus par l'automate acyclique défini par l'intervalle de curseur `[first, last)`. Par défaut, la condition d'arrêt du parcours `last` est la pile vide.

Définition

`language.h`

Contraintes de type

- `DepthFirstCursor` est un modèle de curseur de PEP.
- `DepthFirstCursor::Alphabet` défini un opérateur d'écriture sur flux `<<`.

Préconditions

- `[first, last)` est un intervalle valide.

Complexité

Linéaire : `last - first`.

Exemple

```
DFA_mtf<> dfa;
language(cout, dfirstc(dfa));
```

Notes

L'utilisation de cet algorithme n'a de sens que sur les automates acycliques.

Voir aussi

curseur de PEP, `language_iterator`.

first_match

Catégorie : Algorithme **Type de composant : Fonction**

Prototype

```
template <class ForwardIterator, class Cursor>
ForwardIterator
first_match(ForwardIterator first, ForwardIterator last, Cursor c);
```

Description

L'algorithme `first_match` recherche le plus petit préfixe de la séquence `[first, last)` reconnu par l'automate pointé par le curseur `c`. Il renvoie un itérateur pointant derrière le dernier caractère du préfixe reconnu. S'il ne trouve rien, `first_match` retourne `first` ce qui signifie qu'une transition n'était pas définie ou qu'aucun état terminal n'a été atteint pendant l'itération.

Définition

`matcher.h`

Contraintes de type

- `ForwardIterator` est un modèle d'itérateur monodirectionnel (forward iterator).
- `Cursor` est un modèle de curseur simple.
- `ForwardIterator::value_type` doit être convertible en `Cursor::Alphabet`.

Préconditions

- `[first, last)` est un intervalle valide.
- `c` n'a pas une valeur singulière.

Complexité

Linéaire : au plus `last - first` appels à la méthode `Cursor::forward`.

Exemple

```
const char *w = "ASTL";
DFA_tr<> dfa;
const char *m = first_match(w, w + 4, plainc(dfa));
if (m == w) cout << "échec";
else copy(w, m, ostream_iterator<char>(cout));
```

Notes

Cet algorithme ignore le mot vide.

Voir aussi

`longest_match`, `match_count`.

longest_match

Catégorie : Algorithme **Type de composant : Fonction**

Prototype

```
template <class ForwardIterator, class Cursor>
ForwardIterator
longest_match(ForwardIterator first, ForwardIterator last, Cursor c)
```

Description

L'algorithme `longest_match` recherche le plus long préfixe de la séquence `[first, last)` reconnu par l'automate pointé par le curseur `c`. Il renvoie un itérateur pointant derrière le dernier caractère du préfixe reconnu. S'il ne trouve rien, `longest_match` retourne `first` ce qui signifie qu'une transition n'était pas définie ou qu'aucun état terminal n'a été atteint pendant l'itération.

Définition

`matcher.h`

Contraintes de type

- `ForwardIterator` est un modèle d'itérateur monodirectionnel (forward iterator).
- `Cursor` est un modèle de curseur simple.
- `ForwardIterator::value_type` doit être convertible en `Cursor::Alphabet`.

Préconditions

- `[first, last)` est un intervalle valide.
- `c` n'a pas une valeur singulière.

Complexité

Linéaire : au plus `last - first` appels à la méthode `Cursor::forward`.

Exemple

```
const char *w = "ASTL";
DFA_tr<> dfa;
const char *m = longest_match(w, w + 4, plainc(dfa));
if (m == w) cout << "échec";
else copy(w, m, ostream_iterator<char>(cout));
```

Notes

Cet algorithme ignore le mot vide.

Voir aussi

`first_match`, `match_count`.

match_count

Catégorie : Algorithme **Type de composant : Fonction**

Prototype

```
template <class InputIterator, class Cursor>
unsigned int
match_count(InputIterator first, InputIterator last, Cursor c)
```

Description

L'algorithme `match_count` compte le nombre d'états terminaux atteints lors du parcours de l'automate sur le mot défini par l'intervalle `[first, last)`.

Définition

`matcher.h`

Contraintes de type

- `ForwardIterator` est un modèle d'itérateur monodirectionnel (forward iterator).
- `Cursor` est un modèle de curseur simple.
- `ForwardIterator::value_type` doit être convertible en `Cursor::Alphabet`.

Préconditions

- `[first, last)` est un intervalle valide.
- `c` n'a pas une valeur singulière.

Complexité

Linéaire : au plus `last - first` appels à la méthode `Cursor::forward`.

Exemple

```
const char *w = "ASTL";
DFA_hash<> dfa;
cout << match_count(w, w + 4, plainc(dfa)) << \" occurrences\" << endl;
```

Notes

Cet algorithme ignore le mot vide.

Voir aussi

`first_match`, `longest_match`.

acyclic_minimization

Catégorie : Algorithme **Type de composant : Fonction**

Prototype

```
template <class DFA>
void acyclic_minimization(DFA &A);
```

Description

L'algorithme `acyclic_minimization` minimise l'automate acyclique `A`. Cet algorithme est beaucoup plus efficace que l'algorithme général sur les automates cycliques.

Définition

`minimize.h`

Contraintes de type

- `DFA` est un modèle de DFA dont les transitions sont triées (`DFA_matrix`, `DFA_map`, `DFA_bin`).
- `DFA::Tag` doit définir les méthodes suivantes :
  1. `unsigned char height() const.`
  2. `void height(unsigned char x).`
  3. `unsigned long degree_in().`
  4. `void degree_in(unsigned long x).`
  5. `unsigned long equivalent_to() const.`
  6. `void equivalent_to(unsigned long x).`

Préconditions

- `A` doit être acyclique.

Complexité

$n \log(n)$ où n représente le nombre de transitions de l'automate.

Exemple

```
DFA_matrix<plain, minimization_tag> dfa;
acyclic_minimization(dfa);
```

Notes

Le fichier `minimize.h` contient la définition d'un tag `minimization_tag` implémentant les méthodes requises par l'algorithme.

Voir aussi

`brzozowski`.

brzozowski

Catégorie : Algorithme **Type de composant : Fonction**

Prototype

```
template <class DFA1, class DFA2>
void brzozowski(const DFA1& source, DFA2 &dest);
```

Description
L'algorithme `brzozowski` copie l'automate `source` vers `dest` en le minimisant.

Définition
`minimize.h`

Complexité
Identique à celle de la déterminisation.

Exemple

```
DFA_matrix<> source;
DFA_map<> dest;
brzozowski(source, dest);
```

Notes
Cet algorithme ne copie pas les tags.

Voir aussi
`acyclic_minimization`.

tree_build

Catégorie : Algorithme **Type de composant : Fonction**

Prototype

```
template <class DFA, class InputIterator>
DFA::State
tree_build(DFA &A, InputIterator first, InputIterator last,
           const DFA::Tag &t);

template <class DFA, class InputIterator>
void
tree_build(DFA &A, InputIterator first, InputIterator last);
```

Description

La première version de l'algorithme `tree_build` ajoute le mot `[first, last)` à l'automate `A` dont la structure est un arbre, affecte à l'état terminal créé le tag `t` et renvoie son identifiant. La deuxième version ajoute un ensemble de mots défini par l'intervalle `[first, last)` dont les itérateurs pointent sur des containers séquentiels contenant les mots.

Définition

`astl_tree.h`

Contraintes de type

- `DFA` est un modèle de DFA.
- `InputIterator` est un modèle d'itérateur d'entrée (input iterator).
- Première version : `InputIterator::value_type` doit être convertible en `DFA::Alphabet`.
- Deuxième version : `InputIterator::value_type` doit être un modèle de container dont le type d'élément est convertible en `DFA::Alphabet`.

Préconditions

- `[first, last)` est un intervalle valide.
- `A` doit être vide ou avoir une structure d'arbre.

Complexité

Première version : linéaire ; au plus `last - first` appels à `DFA::new_state` et `DFA::set_trans`. Deuxième version : équivalent à `last - first` appels à la première version de l'algorithme.

Exemple

```
DFA_bin<> dfa;
tree_build(dfa, istream_iterator<string>(cin), istream_iterator<string>());
```

Bibliographie

[1] A. Aho and M. Corasick. Efficient string matching. *Communications of the ACM*, 18(6) :333–340, 1975.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] D. Batory. Intelligent components and software generators. Technical Report CS-TR-97-06, Université du Texas d'Austin, janvier 1997.

[4] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The genvoca model of software-system generators. *IEEE Software*, 11(5) :89–94, septembre 1994.

[5] D. S. Batory. Subjectivity and software system generators. Technical Report CS-TR-95-32, Université du Texas, Austin, janvier 1995.

[6] D. Beauquier, J. Berstel, and Ph. Chrétienne. *Elements d'Algorithmique*. Masson, 1992.

[7] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of the Conference on Object-Oriented Programming : Systems, Languages and Applications*, pages 303–311. ACM, 1990.

[8] U. Breymann, K. Czarnecki, and U. Eisenecker. Generative components : One step beyond generic programming. `http ://home.t-online.de/home/Ulrich.Eisenecker/dag.htm`, 1998.

[9] H. Brönnimann, L. Kettner, S. Schirra, and R. Veltkamp. Applications of the generic programming paradigm in the design of CGAL. In M. Jazayeri, R. G. K. Loos, and D. R. Musser, editors, *Generic Programming '98*, volume 1766 of *Lecture Notes in Computer Science*, pages 206–217. Springer, 2000.

[10] T. Cargill. *C++ Programming Style*. Addison-Wesley, 1992.

[11] K. Czarnecki. *Generative Programming : Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. Thèse de doctorat, Technische Universität Ilmenau, Germany, 1998.

[12] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen. Generative programming and active libraries. In M. Jazayeri, R. Loos, and E. Barke, editors, *Generic Programming '98*, volume 1766 of *Lecture Notes in Computer Science*, pages 25–39, Berlin Heidelberg, 2000. Springer-Verlag.

[13] J. C. Dehnert and A. Stepanov. Fundamentals of generic programming. In M. Jazayeri, R. G. K. Loos, and D. R. Musser, editors, *Generic Programming '98*, volume 1766 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2000.

[14] H. Eichelberger and J. W. V. Gudenberg. UML description of the STL. In *First Workshop on C++ Template Programming, Erfurt, Germany*, octobre 2000.

[15] U. W. Eisenecker, F. Blinn, and K. Czarnecki. A solution to the constructor-problem of mixin-based programming in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.

[16] M. Forster, A. Pick, and M. Raitner. The graph template libray 0.3.1 (gtl). Documentation, Université de Passau, `http ://www.fmi.uni-passau.de/Graphlet/GTL/`, 1999.

[17] W. B. Frakes and R. Baeza-Yates, editors. *Information Retrieval*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.

[18] E. Gamess, D. R. Musser, and A. Sánchez-Ruíz. Complete traversals and their implementation using the standard template library. In R. M. Anwandter, editor, *Proceedings of the XXIII Latinamerican Conference on Informatics*, volume 1, pages 221–230, Valaparaiso, Chili, novembre 1997. CLEI.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[20] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software–Practice & Experience*, 30(11) :1203–1233, septembre 2000.

[21] G. Glass and B. Schuchert. *The STL <primer>*. Prentice Hall, 1996.

[22] M. Gradman and C. Joy. Database template library. `http ://www.geocities.com/corwinjoy/dtl/`, 2002.

[23] M. Gross. The use of finite automata in the lexical representation of natural language. In D. Perrin M. Gross, editor, *Electronic Dictionaries and Automata in Computational Linguistics*, volume 377, pages 35–50, LITP Spring School on Theoretical Computer Science, 1989. Springer-Verlag.

[24] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Adison-Wesley Publishing Company, Reading, Massachusets, USA, 1979.

[25] W. L. Hürsch and C. V. Lopes. Separation of concerns, février 1995. College of Computer Science, Northeastern University, Boston.

[26] Silicon Graphics Inc. Standard template library programmer's guide. Documentation en ligne, `http ://www.sgi.com/Technology/STL/`, 1999.

[27] *ISO/IEC Final Draft International Standard 14882 : Programming Language C++*. 1998.

[28] D. Kapur and D. R. Musser. Tecton : a framework for specifying and verifying generic system components. Technical Report 92-20, Rensselaer Polytechnic Institute, Troy, 1992.

[29] A. Kershenbaum, D. R. Musser, and A. Stepanov. Higher order imperative programming. Technical Report 88-10, Rensselaer Polytechnic Institute, avril 1988.

[30] D. Kühl. Generic graph algorithm. In M. Jazayeri, R. G. K. Loos, and D. R. Musser, editors, *Generic Programming '98*, volume 1766 of *Lecture Notes in Computer Science*, pages 249–255. Springer, 2000.

[31] D. Kühl. STL and OO don't easily mix, septembre 2000. `http ://www.oonumerics.org/tmpw00/kuehl.html`.

[32] K. Kreft and A. Langer. Allocator types. *C++ Report*, 10(6) :54–61, juin 1998.

[33] U. Köthe and K. Weihe. The STL model in the geometric domain. In M. Jazayeri, R. G. K. Loos, and D. R. Musser, editors, *Generic Programming '98*, volume 1766 of *Lecture Notes in Computer Science*, pages 232–248. Springer, 2000.

[34] K. Lieberherr and B. Patt-Shamir. The refinement relation of graph-based generic program. In M. Jazayeri, R. Loos, and E. Barke, editors, *Generic Programming '98*, volume 1766 of *Lecture Notes in Computer Science*, pages 40–52, Berlin Heidelberg, 2000. Springer-Verlag.

[35] A. Ludwig and D. Heuzeroth. Metaprogramming in the large. In G. Butler and S. Jarzabek, editors, *Generative and Component-Based Software Engineering, Second International Symposium, GCSE 2000, Erfurt, Germany, October 9-12, 2000, Revised Papers*, volume 2177 of *Lecture Notes in Computer Science*, pages 178–187. Springer, 2001.

[36] K. Mehlhorn and S. Näher. Leda : A library of efficent data types and algorithms. In A. Kreczmar and G. Mirkowska, editors, *Mathematical Foundations of Computer Science 1989 (MFCS'89) : Proc.*, pages 88–106. Springer, Berlin, Heidelberg, 1989.

[37] M. Mohri, F. Pereira, and M. Riley. A rational design for a weighted finite-state transducer library. *Lecture Notes in Computer Science*, 1436 :144–158, 1997.

[38] D. Moss, A. Bleasby, and W. Pitt. The bioinformatics template library (btl). Présentation à la conférence Object's in Bioinformatics '97, European Bioinformatics Institute, Cambridge, UK., juin 1997.

[39] P-A. Muller. *Modélisation objet avec UML.* Eyrolles, 1997.

[40] D. Musser. Measuring computing times and operation counts of generic algorithms, 2000. `http ://www.cs.rpi.edu/~ musser/gp/timing.html`.

[41] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide.* Addison-Wesley, 1996.

[42] D. R. Musser, S. Schupp, and R. Loos. Requirement oriented programming. In M. Jazayeri, R. G. K. Loos, and D. R. Musser, editors, *Generic Programming '98*, volume 1766 of *Lecture Notes in Computer Science*, pages 12–24. Springer, 2000.

[43] D R. Musser and A. Stepanov. A library of generic algorithms in ada. In *Using Ada (1987 International Ada Conference)*, pages 216–225, New York, NY, 1987. ACM SIGAda.

[44] D. R. Musser and A. Stepanov. Generic programming. In *Springer LNCS 358*, pages 13–25. ISSAC88 et AAEC-6, 1989.

[45] D. R. Musser and A. Stepanov. Algorithm-oriented generic libraries. *Software–Practice & Experience*, 24(7) :623–642, juillet 1994.

[46] N. C. Myers. Traits : a new and useful template technique. *C++ Report*, 7(5) :32–35, juin 1995. `http ://www.cantrip.org/traits.html`.

[47] M. Nelson. *C++ Programmer's Guide to the Standard Template Library.* IDG Books Worldwide, 1995.

[48] J. Ovlinger and M. Wand. A language for specifying recursive traversals of object structures. In *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 70–81, 1999.

[49] J. Ovlinger and M. Wand. A language for specifying traversals of object structures, 1999. College of computer science, Northeastern University, Boston.

[50] D. Perrin. *Automates et algorithmes sur les mots.* Annales des Télécommunications. CNET, Paris, 1989.

[51] D. Raymond and D. Wood. Grail : A C++ library for automata and expressions. *Journal of Symbolic Computation*, 17 :341–350, 1995.

[52] D. Revuz. *Dictionnaires et Lexiques, Méthodes et Algorithmes*. Thèse de doctorat, Université Paris VII, février 1991.

[53] D. Revuz. Operations on extended automata. *Lecture Notes in Computer Science*, 1436 :171–175, 1997.

[54] D. Revuz. Minimal acyclic automaton. *Lecture Notes in Computer Science*, 1436 :171–175, 2000.

[55] E. Roche and Y. Schabes. *Finite State Language Processing*. The MIT Press, Cambridge, Massachusetts, 1997.

[56] S. Schupp and R. Loos. Suchthat - generic programming works. In M. Jazayeri, R. G. K. Loos, and D. R. Musser, editors, *Generic Programming '98*, volume 1766 of *Lecture Notes in Computer Science*, pages 133–145. Springer, 2000.

[57] R. W. Selby. Empirically analyzing software reuse in a production environment. In W. Tracz, editor, *Software Reuse : Emerging Technology*, pages 176–189. IEEE Computer Society Press, 1988.

[58] J. G. Siek, L-Q. Lee, and A. Lumsdaine. The generic graph component library. In *Proceedings of the Conference on Object-Oriented Programming : Systems, Languages and Applications*, pages 399–414. ACM, 1999.

[59] J. G. Siek, L-Q. Lee, and A. Lumsdaine. *Boost Graph Library, The User Guide and Reference Manual*. Addison-Wesley, 2002.

[60] J. G. Siek and A. Lumsdaine. The matrix template library : A unifying framework for numerical linear algebra. In *Proceedings of the ECOOP'98 Workshop on Parallel Object-Oriented Computing*. POOSC'98, 1998.

[61] Y. Smaragdakis and D. Batory. Implementing reusable object-oriented components. In *5th International Conference on Software Reuse*, pages 36–45, Victoria, Canada, 1998.

[62] Y. Smaragdakis and D. S. Batory. Implementing layer designs with mixin layers. In Eric Jul, editor, *ECCOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, Proceedings*, volume 1445 of *Lecture Notes in Computer Science*, pages 550–570. Springer, 1998.

[63] Y. Smaragdakis and D. S. Batory. Mixin-based programming in C++. In G. Butler and S. Jarzabek, editors, *Generative and Component-Based Software Engineering, Second International Symposium, GCSE 2000, Erfurt, Germany, October 9-12, 2000, Revised Papers*, volume 2177 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2001.

[64] J. Soukup. Intrusive data structures. *C++ Report*, 1998. publié en trois partie entre mai et octobre, disponible en version électronique à `http ://www.codefarms.com/publications/intrusiv/intr.htm`.

[65] J. Soukup. The pattern template library 1.1 (ptl). Documentation, Code Farms Inc., `http ://www.codefarms.com/ptl/`, 1998.

[66] R. M. Stallman. Using and porting the gnu compiler collection for gcc-3.0. Documentation, Free Software Foundation, Inc., `http ://www.gnu.org/`, 2001.

[67] A. Stepanov and M. Lee. The standard template library. Technical report, Hewlett-Packard Laboratories, 1995.

[68] B. Stroustrup. *Le Langage C++*. Addison-Wesley, seconde edition, 1992.

[69] S. Teale. *C++ IOStreams Handbook*. Addison-Wesley, 1994.

[70] M. VanHilst and D. Notkin. Using c++ templates to implement role-based designs. In K. Futatsugi and S. Matsuoka, editors, *ISOSTAS '96, Second JSSST International Symposium on Object Technologies for Advanced Software*, volume 1049 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 1996.

[71] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5) :26–31, juin 1995.

[72] T. L. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4) :36–43, mai 1995.

[73] T. L. Veldhuizen. Scientific computing : C++ versus fortran. *Dr. Dobb's Journal*, pages 34–41, novembre 1997.

[74] T. L. Veldhuizen. Arrays in Blitz++. In D. Caromel, R. R. Oldehoeft, and M. Tholburn, editors, *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, volume 1505, pages 223–230, Berlin, Heidelberg, New York, Tokyo, 1998. Springer-Verlag.

[75] T. L. Veldhuizen. C++ templates as partial evaluation. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 13–18. PEPM'99, ACM, 1999.

[76] T. L. Veldhuizen and D. Gannon. Active libraries : Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*, Philadelphia, PA, USA, 1998. SIAM.

[77] T. L. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran ? In Y. Ishikawa, editor, *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, volume 1343 of *Lecture Notes in Computer Science*, pages 49–56. Springer-Verlag, 1997.

[78] B. W. Watson. The design and implementation of the fire engine : A c++ toolkit for finite automata and regular expressions. Technical report, Eindhoven University of Technology, 1994.

[79] B. W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. Thèse de doctorat, Eindhoven University of Technology, Pays-Bas, 1995.

[80] J. Weidl. The standard template library tutorial. Université de Vienne, institut des systèmes d'information, département des systèmes distribués, `http ://www.infosys.tuwien.ac.at/Research/Component/tutorial/`, avril 1996.

[81] N. Wirth. *Algorithms + Data Structures = Programs*. PrenticeHall, 1976.