N° d'ordre: 2473

THESE

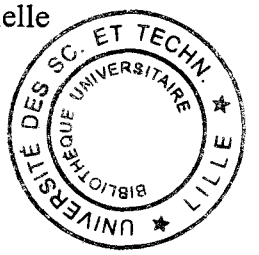
présentée à

L'Université des Sciences et Technologies de LILLE

pour obtenir le grade de

Docteur de l'Université

Spécialité: Productique, Automatique et Informatique Industrielle



par

Laurent ALLAIN
DEA Informatique

Contribution à la Modélisation et à la Spécification: Réseaux Formels et Bases de Données Actives

Soutenue le 18 janvier 1999 devant le jury composé de:

M. Christian CADIVEL	Consultant Aston
M. Jean-Marc GEIB	Professeur à l'USTL Lille
M. Jean-Claude GENTINA	Professeur à l'EC Lille
Mme. Agnès HEBRARD	Maître de conférence à l'EC Lille
M. Jean-Paul PIGNON	Thomson-CSF Optronique
M. Christophe SIBERTIN-BLANC — Rapporteur	Professeur à l'UT1 Toulouse
M. Philippe VANHEEGHE	Professeur à l'ISEN Lille
M. François VERNADAT — Rapporteur	Professeur à la Faculté des Sciences Metz
M. Pascal YIM	Maître de conférence à l'EC Lille

Thèse dirigée par M. P. Yim et M. J.C. Gentina, préparée au Laboratoire d'Automatique et d'Informatique Industrielle de Lille (LAIL, UPRESA CNRS 8021), EC Lille, ISEN.

REMERCIEMENTS

Le travail présenté dans ce mémoire a été réalisé au sein du Laboratoire d'Automatique et d'Informatique Industrielle de Lille (L.A.I.L.), dans l'équipe Production Flexible Manufacturière (P.F.M.) à l'Ecole Centrale de Lille.

Je remercie Monsieur Jean-Claude Gentina, Directeur de l'Ecole Centrale de Lille, de son accueil au sein de l'Ecole Centrale, ainsi que Madame Geneviève Dauphin-Tanguy et Monsieur Marcel Staroswiecki, Directeurs du L.A.I.L., pour leur accueil au sein du L.A.I.L.

Je suis très reconnaissant envers Monsieur Christophe Sibertin-Blanc, Professeur à l'Université de Toulouse, et Monsieur François Vernadat, Professeur à la Faculté des Sciences de l'Université de Metz, d'avoir accepté d'être les rapporteurs de cette thèse et d'avoir pris le temps d'examiner mon travail.

Mes plus vifs remerciements vont à Monsieur Pascal Yim, mon Directeur de thèse, qui m'a encadré durant toute cette thèse et qui m'a fait profiter de toute son expérience, de son esprit de synthèse et de ses critiques.

Je témoigne toute ma gratitude à Monsieur Jean-Marc Geib, Professeur à l'Université des Sciences et Technologies de Lille, d'avoir bien voulu présider le jury de cette thèse.

Je remercie également Monsieur Christian Cadivel, Consultant chez Aston, Madame Agnès Hébrard, Maître de conférence à l'Ecole Centrale de Lille, Monsieur Jean-Paul Pignon, Chef du Groupe Algorithmie des Systèmes Optroniques chez Thomson-CSF Optronique, et Monsieur Philippe Vanheeghe, Responsable du Département Signaux et Systèmes de l'Institut Supérieur d'Electronique du Nord, pour avoir accepté de faire partie du jury.

Que toutes les personnes ayant relu ce mémoire soient associées à mes remerciements, ainsi que toutes les personnes qui m'ont soutenu durant ces trois années, spécialement mes plus proches collègues, dont l'humour a été une véritable soupape de sécurité.

J'ai enfin été touché par la confiance que m'ont accordée Monsieur Jean-Noël Decarpigny, Directeur de l'I.S.E.N., et Monsieur Michel Lannoo, Directeur de la Recherche à l'I.S.E.N., en me proposant d'effectuer ce travail.

TABLE DES MATIERES

INTRODUCTION

La *modélisation des données* dans les systèmes d'information (SI) a fait l'objet de nombreuses recherches et développements, qui ont abouti à proposer des modèles et des outils éprouvés. Ces derniers reposent pour la plupart sur le modèle relationnel [CHE 76][COD 70], et depuis quelques temps sur le modèle orienté objet [ATK 89]. Cependant, la *spécification des traitements* et la modélisation du *comportement dynamique* des systèmes d'information restent difficiles et les solutions proposées sont diverses. La cause en est certainement le manque de précision concernant les spécifications fonctionnelles des problèmes à résoudre, généralement exprimées en pratique à l'aide d'un langage naturel plus ou moins formalisé [MEY 79].

En particulier, le comportement d'un système d'information réagissant à des événements est particulièrement difficile à modéliser. Les bases de données actives utilisent pour la plupart un formalisme basé sur des règles Evénement-Condition-Action. Néanmoins, dans un tel formalisme, les actions sont en général spécifiées de manière non déclarative (code impératif). De plus, le comportement global (séquences d'événements, parallélisme, synchronisations, partage de ressources, ...) n'apparaît pas de manière explicite avec les règles utilisées.

Les réseaux de Petri semblent être un formalisme particulièrement pertinent pour décrire dans un seul modèle les données, les traitements et le comportement d'un système d'information (Cf [CAD 97] pour une synthèse des principales applications des RdP à la modélisation des systèmes d'information: RdP de haut niveau, réseaux CEM, méthode M*, RdP à objets, ...). Néanmoins, plusieurs aspects sont difficiles à traiter avec des RdP de haut niveau usuels:

- la prise en compte de contraintes d'intégrité (contraintes à vérifier pour chaque marquage admissible)
- la consommation/production d'ensembles de jetons dont la cardinalité n'est pas fixée a priori (ensembles définis en compréhension: « *tous les jetons tels que* ... »)
- la spécification déclarative des traitements: on souhaite éviter les notions impératives comme l'affectation, et se limiter à des formules logiques

Les réseaux formels [YIM 95], [CAD 97], [ALL 98] ont été définis dans ce but. Ils sont comparables aux RdP de haut niveau, avec l'introduction de nouveaux types d'arcs et la définition de contraintes sur les places, les marquages, et les transitions. Les annotations d'un réseau formel sont purement déclaratives (contraintes), et ne comportent notamment aucun appel à un code impératif. Nous montrerons à l'aide d'exemples issus de domaines variés la capacité des réseaux formels à modéliser des traitements complexes.

Plus précisément, nous proposons dans cette thèse d'étudier l'utilisation des réseaux formels comme langage de description pour les bases de données actives. Nous décrirons un outil, *NetSPEC*, permettant de compiler une description en termes de réseaux formels vers un moniteur transactionnel (C/SQL). L'outil permet une génération automatique d'une part de la structure de la base (tables), et d'autre part de la partie opératoire, incluant la vérification des contraintes d'intégrité et la prise en charge des événements. Une des principales difficultés d'une telle réalisation consiste à éliminer le non-déterminisme inhérent au comportement d'un réseau formel, en proposant un ordonnancement automatique des tâches, que nous comparerons à l'approche des bases de données actives classiques. En effet, contrairement à un simulateur de réseaux de Petri, nous souhaitons obtenir un code déterministe exécutable sur un SGBD relationnel standard. L'outil sera testé sur plusieurs exemples, représentatifs chacun d'une difficulté spécifique.

Notre but principal est donc une aide à la conception et à la réalisation de bases de données actives opérationnelles. Dans ce sens, notre travail s'inscrit davantage dans le cadre du Génie Logiciel que de l'Informatique Fondamentale. En particulier, nous ne nous intéressons pas ici en détail à la simulation non déterministe d'un réseau formel. Sur ce sujet, on pourra se référer au travail d'Arnaud Lefort [LEF 98], qui a notamment décrit la recherche de toutes les séquences de tirs entre deux marquages partiellement connus, sur un modèle comparable. D'autre part, Khalid Gaber [GAB 99] a défini une sémantique déclarative et opérationnelle basée sur les langages de premier ordre et la théorie des ensembles pour un formalisme simplifié des réseaux formels. Un tel travail nécessite l'usage d'outils théoriques assez lourds, et nous avons choisi de présenter ici une description plus simple du comportement des réseaux formels, utilisant le formalisme du langage Z. Enfin, nous ne développerons pas en détail dans ce mémoire l'analyse des propriétés formelles du réseau qui fait l'objet de la thèse en cours de Philippe Bon (à l'aide des outils de la méthode formelle B).

Le plan que nous adopterons pour la présentation de ce mémoire comporte quatre parties principales:

Le chapitre I est consacré à la présentation de la structure des systèmes d'information et des objectifs poursuivis par les systèmes de gestion de bases de données. Afin de montrer que ces objectifs sont respectés de mieux en mieux au cours de l'évolution des SGBD, nous présentons rapidement les modèles les plus classiques de première et de seconde générations: modèle hiérarchique, modèle réseau, modèle relationnel et modèle orienté objet. Ceci nous permettra de positionner ces modèles sur un référentiel montrant clairement leurs points forts et leurs points faibles.

Le chapitre II concerne le comportement réactif des SGBD. Nous présentons le formalisme généralement utilisé pour modéliser un tel comportement en introduisant le concept des règles-ECA (règles Evénement-Condition-Action), et nous montrons que les SGBD actifs récemment apparus tentent d'apporter une solution. Toutefois, les SGBD actifs étant tous basés sur les modèles relationnel ou orienté objet, nous constatons que la

modélisation d'un système (en particulier l'expression des traitements) est encore très dépendante de la programmation classique.

Le chapitre III présente les réseaux formels comme nouvel outil de modélisation et de spécification du comportement dynamique d'un système. Nous présentons ici les réseaux formels par le biais d'une approche plus « pratique » que théorique, et nous montrons grâce à quelques exemples empruntés à quatre grandes classes d'application (traitement d'informations, planification, contrôle de process, simulation) leur puissance de modélisation.

Le chapitre IV propose d'appliquer le paradigme des réseaux formels à la spécification et à la conception de bases de données actives. Nous montrons ici, par l'étude du compilateur de réseaux formels *NetSPEC*, comment ceux-ci peuvent être utilisés comme langage de description d'un SGBD actif. Les différents problèmes inhérents aux SGBD actifs sont à nouveau abordés et leurs solutions décrites ici. Les exemples étudiés dans le chapitre précédent sont concrètement implémentés à l'aide de *NetSPEC*, et les résultats obtenus sont analysés.

CHAPITRE I

SYSTEMES DE GESTION DE BASES DE DONNEES

Le concept de « *base de données* », élément incontournable pour la modélisation des données d'un système d'information, est apparu dans les années 60 lors d'un premier développement des *systèmes de fichiers*. Ces derniers tendent alors essentiellement à faire apparaître les mémoires secondaires comme idéales, partageables, banalisées, directement adressables, et de capacité infinie [GAR 82]. Alors qu'un système d'information classique s'appuyant sur des fichiers [GAR 79] est conçu autour des *fonctions* (les programmes), un système « base de données » est conçu autour des *données* (les structures).

La première génération des *Systèmes de Gestion de Bases de Données* (SGBD) a été marquée par la séparation de la description des données des programmes d'application et l'avènement des langages d'accès navigationnels [TSI 76] qui permettent de se déplacer dans des structures de type graphe et d'obtenir, un par un, des groupes de données appelés *articles*. Cette première génération, caractérisée par des systèmes obéissants aux premières recommandations du CODASYL [COD 71] d'une part et par IMS d'IBM [IBM 75] d'autre part, est basée sur des modèles d'accès visant essentiellement à optimiser les méthodes de placement des données sur les mémoires secondaires afin de réduire les temps d'accès.

La deuxième génération des SGBD qui a vu le jour 10 ans plus tard à partir du modèle *relationnel* [COD 70], vise à enrichir le SGBD afin de simplifier l'accès aux données. Ainsi, la deuxième génération offre des langages assertionnels basés sur la logique, permettant de spécifier les données que l'on souhaite obtenir sans dire comment y accéder. Avec le modèle relationnel sont apparues de nombreuses méthodes, dont l'objectif était non seulement d'avoir la capacité de définir les données, mais également de spécifier les traitements effectués sur ces dernières. Malheureusement, l'intégration de ces deux pôles n'est pas toujours définie précisément: une méthode comme Merise [TAR 86] par exemple, nécessite trois modèles de description (modèle conceptuel des données ou MCD, modèle conceptuel des traitements ou MCT, et modèle logique des données ou MLD). D'autres types de modèles ont également vu le jour, comme le modèle *orienté objet* [ATK 89] qui propose d'encapsuler la représentation des informations manipulées et les traitements sur ces informations. Les méthodes basées sur les objets, comme BOOCH [BOO 91] ou HOOD [COU 90], permettent d'atteindre un degré de précision très fin dans l'expression des traitements, tout en unifiant encore plus complètement les données à ces derniers.

Dans un autre ordre d'idée, le besoin d'un comportement autre que purement passif des SGBD s'est fait sentir depuis longtemps. Le comportement réactif a d'abord trouvé une solution grâce à la mise en oeuvre des *triggers* [ESW 76] ayant pour rôle de déclencher des transactions en réponse à l'arrivée d'un événement. La plupart des SGBD relationnels supportent aujourd'hui les fonctionnalités de base des triggers, en s'inspirant du standard SQL3. Cependant depuis une dizaine d'années, des recherches plus originales ont été entreprises pour aboutir à une nouvelle génération de SGBD, appelés SGBD « actifs » [CHA 89b][BUC 94].

Si ces nouveaux systèmes permettent à la fois de modéliser les données et de spécifier les traitements, notamment réactifs, leur inconvénient commun est certainement que les concepteurs d'applications ne peuvent pas encore s'affranchir d'une spécification des traitements peu différente de la programmation classique. Tandis que certains systèmes utilisent le modèle relationnel en s'appuyant sur le langage d'interrogation SQL souvent associé à un langage hôte tel que C, COBOL, ou PL1, les autres reposent sur le modèle orienté objet et utilisent principalement des langages tels que Smalltalk ou dérivés de C++.

Ce chapitre a pour objectif de décrire les différents systèmes de gestion de bases de données classiques. A partir de la définition (d'un point de vue informatique) d'un système d'information, les différents objectifs poursuivis par les SGBD sont ensuite déterminés. Une présentation rapide des différents modèles historiques de SGBD et de leurs concepts de base permettra alors de mettre en évidence leurs points forts et leurs points faibles, en montrant comment ces objectifs sont réalisés.

I.1 Structure d'un système d'information automatisé

Le cadre théorique que nous exposons ici s'inspire très largement du rapport rédigé en 1975 par le groupe d'étude du SPARC (Standard Planning and Requirement Committee) [ANS 75][DON 77][TSI 78]. Ce rapport a eu une très large diffusion et un grand impact notamment par ses propositions de vocabulaire qui se sont imposées assez rapidement: « *trop de gens parlaient en termes trop dissemblables de choses comparables* » [TAR 85].

La conception d'un système d'information a pour objectif la réalisation d'une base de données physique dont la description est décrite par le *modèle* — ou schéma — interne. Cette base de données renferme des informations relatives à l'activité de l'organisation, qui considérera cet ensemble de données comme étant le *réel perçu*. La très grande difficulté à décrire directement le réel perçu à l'aide d'un formalisme informatique a entraîné l'ANSI/SPARC à rechercher différents niveaux, étapes ou modèles que devrait comporter le processus de structuration d'une base de données. Un effort particulier a été fait pour tenter de définir des interfaces entre les différents modèles ou acteurs du processus.

Chacun des acteurs de l'organisation utilisent à sa manière le même ensemble de données issues du modèle interne et stockées dans une même base de données. Des raisons d'économie, de cohérence des données, et de synchronisation de l'activité des différents acteurs nécessitent alors cette unicité de mémorisation des données élémentaires. La contrepartie en est que le résultat de cette multitude de perceptions et d'interprétations est un réel perçu diversifié.

Afin d'assurer au maximum la stabilité et le partage des données, et donc leur indépendance, vis-à-vis d'une utilisation ou d'une technique particulière, l'objectif de la structure et des mécanismes proposés est de permettre des changements, soit au niveau de l'utilisation des données (fonctions de l'organisation), soit au niveau du fonctionnement du système informatique (système de gestion de la base de données), en réduisant le plus possible les conséquences néfastes. Pour réaliser cette indépendance, le désaccouplement du réel perçu et du modèle interne nécessite la création d'une charnière ou d'un interface constitué du modèle *conceptuel.*

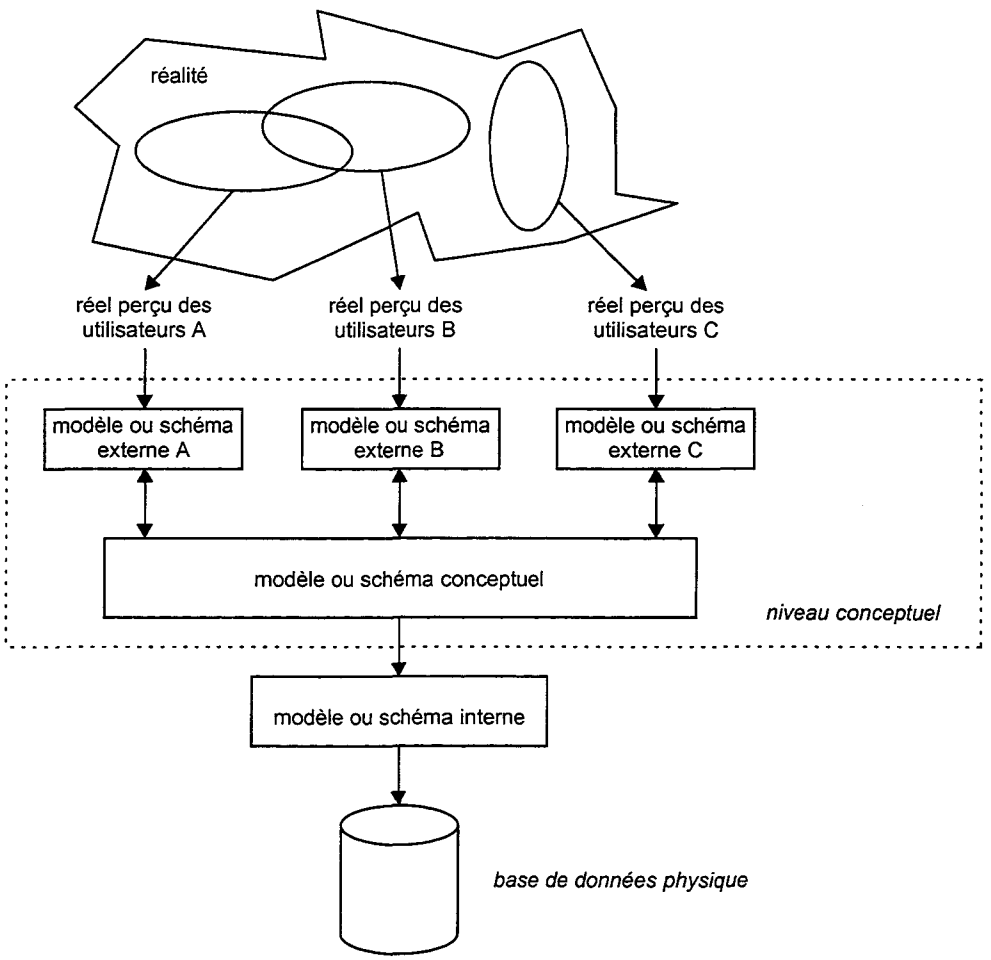


Figure 1: Processus de structuration de la mémoire

Le réel perçu est constitué par la représentation de l'activité de l'organisation que le système de décision se construit pour son usage, descriptible à partir des fonctions du système opérant. Ce réel perçu est exprimé en langage courant de l'organisation: règlements de bordereaux, fiches... qui contiennent des éléments d'informations appelées propriétés (ou caractéristiques, articles, données élémentaires, ...) ainsi que tous les éléments permettant de les relier entre elles.

Le niveau conceptuel constitue l'interface entre le réel perçu et le niveau interne. Il est donc d'une part le domaine de rencontre des gestionnaires (système de décision) et des informaticiens (système opérant), et d'autre part le domaine où s'organise le passage de la diversité des acteurs et de l'évolution des données à la stabilité et à l'unicité de la mémorisation des informations qui est exprimée au niveau interne.

Au niveau conceptuel, le premier aspect concerne la traduction du réel perçu par le modèle conceptuel, modèle unique pour l'organisation ou le domaine concerné, qui contient la description (propriétés, objets, et relations entre les objets) des informations à mémoriser. Le second aspect concerne la description par le biais du modèle externe des informations mémorisées: il faut et il suffit que la structure des informations du modèle externe soit compatible avec celle du modèle conceptuel. Autrement dit, non seulement les propriétés du modèle externe constituent un sous-ensemble de celles du modèle conceptuel, mais il doit exister un processus de passage du modèle externe au modèle conceptuel (mapping).

Signalons finalement qu'afin de favoriser la compréhension entre les gestionnaires et les informaticiens, les modèles conceptuel et externe devront être exprimés à l'aide du même langage, suffisamment naturel pour satisfaire les uns, tout en étant rigoureux pour les autres [TAR 85][TAR 86].

A un moment donné, il n'existe qu'un seul modèle interne qui définit dans le système informatique la structure de données exprimée par le modèle conceptuel. Contrairement au modèle conceptuel qui ne peut pas se concevoir petit à petit sans remettre en cause tout le système d'information (problème de cohérence des données), le modèle interne est assujetti d'une part au système de gestion des données (système de fichiers ou système de gestion de bases de données), et d'autre part à la stratégie d'organisation cherchant à optimiser l'exploitation du système informatique: un changement du système de gestion ou de la stratégie d'organisation ne peut pas se répercuter sur le modèle conceptuel, bien qu'il soit nécessaire de modifier l'interface modèle interne-modèle conceptuel.

I.2 Objectifs des SGBD

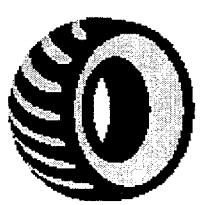
Un *système de gestion de bases de données* (Database Management System ou DBMS) est constitué d'un ensemble de programmes et de matériels dédiés à la création et à la gestion de bases de données de manière fiable et efficace, dans le but de modéliser une partie du monde réel [GAR 82].

Dans l'approche « classique » [JOU 77][WID 77][GAR 79], les fichiers (structure, organisation, support, etc.) sont définis en fonction d'une application, voire même d'un programme. Les données contenues dans ces fichiers sont directement associées aux programmes qui les traitent par des descriptions contenues dans les programmes eux-mêmes. Le contenu des fichiers n'étant connu du système d'information que par les descriptions qu'en donnent les programmes, il n'y a donc aucune indépendance possible entre les données et les programmes.

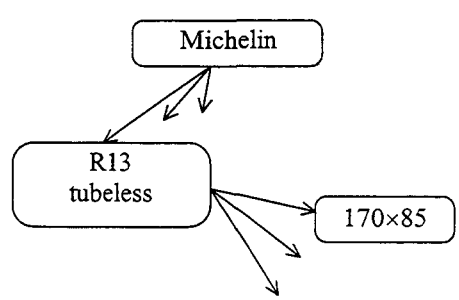
Les DBMS se distinguent clairement des systèmes de fichiers par le fait qu'ils permettent la description des données (définition des noms, formats et caractéristiques) de manière séparée de leur utilisation (mise à jour et recherche), c'est-à-dire des traitements.

Les objectifs principaux que cherchent à atteindre les DBMS sont d'assurer, au niveau des données, l'indépendance physique (1), l'indépendance logique (2), la manipulation (3), l'efficacité des accès (4), l'absence de redondance (5), la cohérence (6), le partage (7), et la sécurité (8) [GAR 82]. Il n'est pas illusoire de penser que peu d'entre-eux satisfont pleinement tous les objectifs, mais tous y tendent plus ou moins bien. La classification des objectifs poursuivis par les DBMS n'est pas sans fondement: un objectif quelconque ne peut prétendre être réalisé que si les objectifs précédents le sont aussi.

L'un des objectifs essentiels est de permettre de réaliser l'*indépendance physique* des données, c'est-à-dire l'indépendance des structures de stockage et des structures de données du monde réel [STO 74]. Il s'agit donc, d'un point de vue informatique, d'assembler les données élémentaires entre elles en ne tenant compte que des critères de performances et de flexibilité. Si l'obligation d'assurer l'indépendance physique n'existait pas, nous devrions trouver dans le monde réel des notions identiques à celles qui apparaissent dans les programmes d'application: méthodes d'accès, modes de placements, critères de tris, chaînages, codages des données, ce qui bien sûr n'est pas réaliste.



Monde réel: pneumatique Michelin 170×85 tubeless type R13



Programme d'application: chaînages des données contenues dans les fichiers

Exemple 1: Indépendance physique des données

Cependant, chacun doit pouvoir assembler les données à sa guise, selon son centre d'intérêt, en ne s'intéressant qu'à une partie précise de la sémantique des données. Permettre à chacun de voir les données comme il le souhaite et permettre l'évolution de cette vue sans remettre en question la vue des autres — du moins dans une certaine mesure — sont les avantages apportés par l'*indépendance logique* [DAT 71].

Considérons des données suivantes:
VEHICULE(n° véhicule, marque, type, couleur)
PERSONNE(n° SS, nom, prénom)
PROPRIETAIRE(n° SS, n° véhicule, date d'achat)

le gérant d'un parking ne s'intéressera qu'à un assemblage de données tel que:
PERSONNE(n° SS, nom, prénom, n° véhicule)

tandis que le gérant d'un centre de contrôle technique ne s'intéressera qu'à cet assemblage:
VEHICULE(n° véhicule, marque, type, date d'achat)

Exemple 2: Indépendance logique des données

Puisque tout un chacun voit les données, il doit aussi pouvoir les *manipuler*, c'est-à-dire les interroger et les mettre à jour. Il est important de voir les données indépendamment de leur implémentation en machine, ce qui est réalisé grâce aux objectifs 1 et 2. Par conséquent, les données que l'on souhaitera retrouver doivent être décrites au moyen d'un langage non procédural — DML ou Data Manipulation Language — sans indiquer la manière de les retrouver, ce qui est propre à la machine. Non procédural signifie ici que l'utilisateur n'a pas à spécifier un quelconque algorithme pour le problème à résoudre, mais qu'il doit seulement définir le problème formellement [RIS 92].

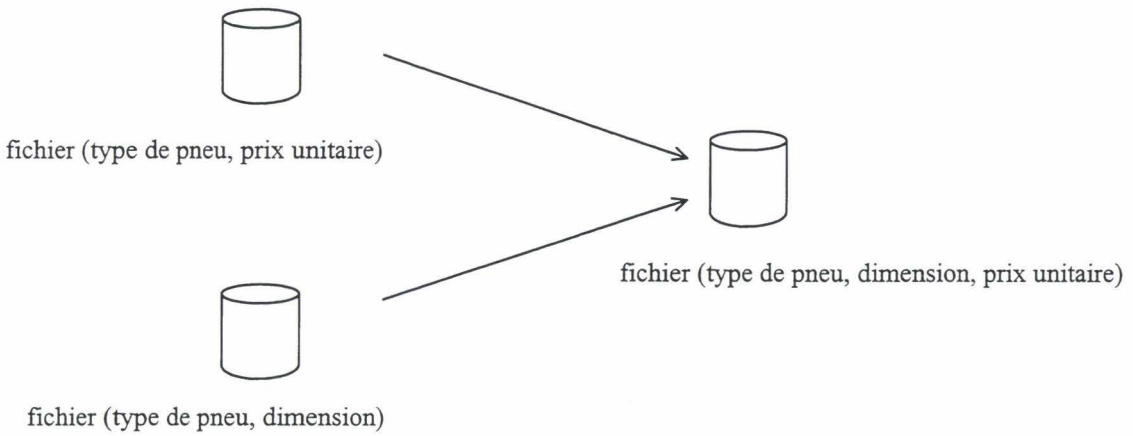
Les langages non procéduraux cités dans l'objectif 3 se doivent d'être *efficaces*, notamment au niveau des accès aux mémoires secondaires (disques) qui restent le goulot d'étranglement majeur des systèmes gourmands en entrées-sorties, problème amplifié par la méconnaissance de la structure interne de stockage des données de la part de l'utilisateur non informaticien. D'un autre côté, pour les concepteurs d'application, des DML très efficaces permettant d'accéder rapidement aux données en utilisant les chemins d'accès définis dans la structure de stockage doivent être disponibles. De plus, ces DML doivent avoir la particularité de pouvoir s'intégrer dans des programmes classiques écrits à l'aide de langages hôtes (COBOL, PL1, C, etc.).

```
type = R13
dimension = 170×85
DBMSCALL(« Michelin fabrique-t-il ce genre de pneumatique? »)
si réponse = 'oui'
  alors
    ...
```

Exemple 3: DML non procédural intégrable dans un langage hôte

La *redondance d'information* existe abondamment dans les systèmes classiques à fichiers, puisque chaque application possède ses données propres. Outre la perte d'espace en mémoire secondaire, la mise à jour des données communes entraîne un gâchis important en moyens humains. Dans une première approche, les DBMS tentent de résoudre le

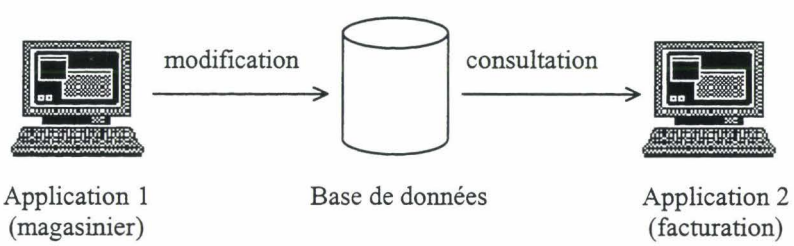
problème en intégrant dans un seul fichier partagé l'ensemble des fichiers comportant plus ou moins de redondance. Cependant, la non redondance d'information ne sera véritablement résolue que par l'apport de la normalisation des associations, comme nous le verrons lors de la présentation des DBMS relationnels.



Exemple 4: Elimination des données redondantes par fusion de fichiers

L'élimination de la redondance entre données n'implique pas la cohérence de ces dernières. En effet, certaines données sont soumises à des règles — intervalle de valeurs par exemple — soit au niveau de la donnée élémentaire, soit au niveau de l'ensemble de données s'il existe des dépendances. Un DBMS doit donc veiller à ce que les applications respectent ces règles lors des modifications des données et ainsi assurer la *cohérence* des données.

Relativement à l'objectif 3, et dans l'optique du *partage* des données, une application doit pouvoir accéder aux données comme si elle était seule à les utiliser, sans savoir qu'une autre application peut les modifier concurremment (problème de l'exclusion mutuelle des ressources partageables [CRO 75]).



Exemple 5: Partage des données

Bien que sortant du cadre de notre étude, signalons finalement que les données doivent faire l'objet d'une *protection* contre les accès non autorisés ou mal intentionnés.

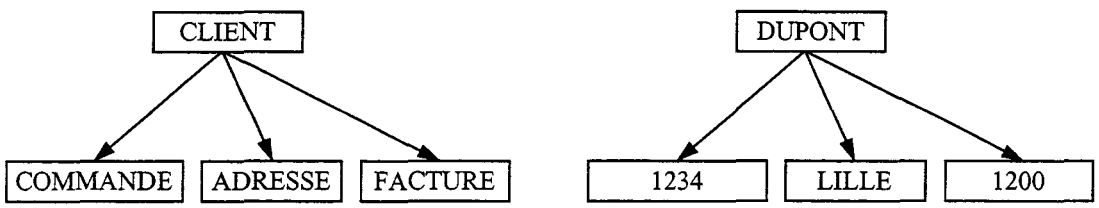
I.3 SGBD de première génération

Les différents modèles de SGBD les plus représentatifs sont ici décrits en expliquant comment ces derniers sont structurés. Les principales notions les caractérisant seront alors dégagées afin de présenter un résumé comparatif.

I.3.1 Le modèle hiérarchique

Le modèle hiérarchique est le plus ancien des modèles importants de bases de données, mais est encore largement utilisé dans l'industrie [MRI 72][IBM 75]. Sa forte présence ne doit pas nous étonner: le monde réel nous apparaît souvent au travers d'une hiérarchie, et les bases de données modélisent les informations de ce monde réel. Cependant, à l'époque de sa création, la véritable impulsion qui lui a donné naissance a été la nécessité d'optimiser les méthodes de placement des données sur les mémoires secondaires afin de réduire les temps d'accès [GAR 82].

Le modèle hiérarchique repose sur l'implémentation d'une structure de données en forme d'arbres, où chaque noeud, appelé segment, est une collection de champs rangés consécutivement dans la base. Les segments sont reliés par des associations [1:n] qui à un segment père (noeud parent ou segment propriétaire) font correspondre n segments fils (noeud enfant ou segment dépendant). Ainsi, en dehors du segment racine (sommet de la hiérarchie), tout segment n'existe que par son segment propriétaire et n'est accessible que par celui-ci. Un segment propriétaire et ses segments dépendants constituent un groupe, simple si aucun segment dépendant n'a de descendant, composé dans le cas contraire, et répétitif si un segment dépendant a plusieurs descendants. Finalement, une base de données hiérarchique peut être considérée comme un ensemble d'arbres, appelé forêt, dont les noeuds sont des segments. Cette définition s'applique aussi bien au niveau des types qu'au niveau des occurrences [TSI 76]:



Groupe CLIENT - niveau des types *Groupe CLIENT - niveau des occurrences*

Figure 2: Exemples de hiérarchies

De la définition donnée précédemment va découler naturellement une représentation à l'aide d'un graphe: les noeuds du graphe sont bien sûr les segments, et les arcs orientés représentent les associations père-fils. A titre d'exemple, la Figure 3 illustre le modèle hiérarchique de l'application *GENCOD*, qui nous servira tout au long de ce chapitre, et dont le cahier des charges est décrit au Paragraphe III.4.3.

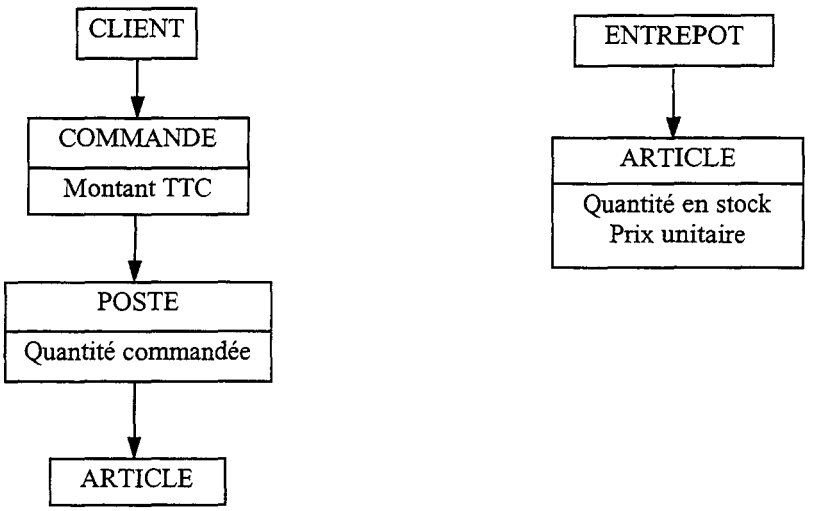


Figure 3: *GENCOD* — Modèle hiérarchique

Cette représentation présente intrinsèquement des défauts majeurs, aussi bien au niveau de l'implémentation des données, qu'au niveau des traitements:

- La redondance d'information: puisqu'un segment fils ne peut avoir plusieurs segments propriétaires, la duplication est alors une contrainte imposée qui engendre à son tour la difficulté de mise à jour d'une entité. La mise à jour nécessite l'exploration de toutes les hiérarchies depuis leurs racines respectives, rendant ainsi difficile le maintien de la cohérence des données pendant l'opération, et dans une moindre mesure l'encombrement des supports physiques de stockage.
- La difficulté d'associer des données appartenant à des hiérarchies différentes: dans l'exemple ci-dessus, il est impossible de répondre à la question « *Rechercher les entrepôts de stockage des articles de la commande 1234* ».

Si nous voulons respecter les objectifs de cohérence des données et d'indépendance données-programmes, il devient nécessaire d'autoriser les liens entre arbres afin de permettre la modélisation des associations [1..n] sans avoir à dupliquer les segments. On aboutit alors à des modèles hiérarchiques étendus dont les possibilités se rapprochent de celles des modèles réseau [IBM 75][BOU 78].

L'un des plus anciens DBMS de type hiérarchique est IMS/VS (Integrated Management System/Virtual Storage) d'IBM [LYO 90]. Le concept fondamental de fonctionnement d'IMS est relativement simple: la structure d'une base de données est décrite selon deux niveaux:

- Le premier niveau correspond à une structure d'enregistrement des données assimilables par le système, qui ne distingue pas le schéma conceptuel du schéma interne, et qui repose sur les associations physiques (Figure 3).
- Le second niveau correspond à une structure logique des données telles qu'elles sont vues par un programme d'application, et qui repose cette fois sur les associations logiques (représentées ici en pointillés):

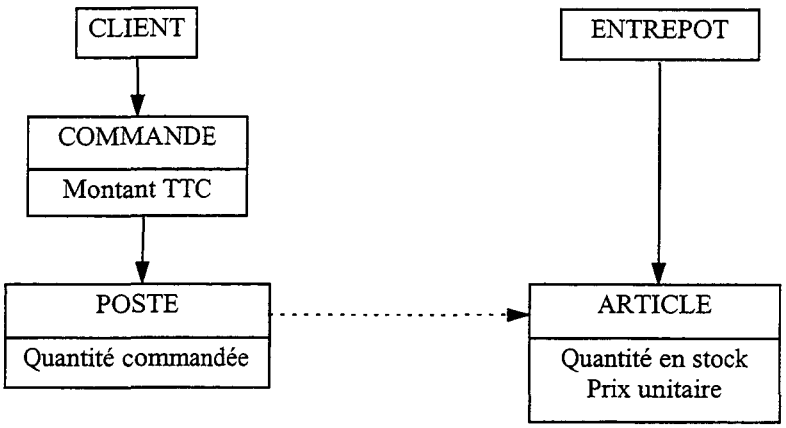


Figure 4: *GENCOD* — Modèle hiérarchique étendu IMS/VS

Même basé sur le modèle hiérarchique étendu, un système comme IMS/VS continue à souffrir totalement ou partiellement des inconvénients déjà énoncés:

- Supposons que le prix d'un article soit en baisse, il faut modifier l'ensemble des occurrences de cet article. Autrement dit, si la redondance d'information n'existe plus au niveau des types, elle continue à être présente au niveau des occurrences.
- S'il est facile d'obtenir la liste des articles commandés par un client, il est plus difficile d'obtenir la liste des clients ayant commandé un article particulier.
- Dans un autre ordre d'idée, la description d'une base de données hiérarchique se fait segment par segment en respectant l'ordre défini sur l'arborescence, ce qui donne peu de liberté au concepteur confronté à un DDL — Data Definition Language — complexe et lourd à utiliser:

```
(1)   DBD       NAME = PDB-CLIENT, ACCESS = HISAM
(2)   DATASET   DD1 = BDCLI, DEVICE = 3390
(3)   SEGM      NAME = CLIENT, BYTES = 42
(4)   FIELD     NAME = (NOM, SEQ), BYTES = 42, START = 1
      ...
      SEGM      NAME = COMMANDE, PARENT = CLIENT, BYTES = 18
(n)   FIELD     NAME = NUMCOM, BYTES = 4, START = 1
      ...
```

Exemple 6: *GENCOD* — Description de la base hiérarchique au moyen d'un DDL

I.3.2 Le modèle réseau

Le modèle réseau [TAY 76] a été initialement proposé à partir des recommandations du CODASYL/DBTG (Committee Of Data System Language/Data Base Task Group) [COD 71], puis a été l'objet de quelques améliorations [COD 78]. Par rapport au modèle hiérarchique, les apports du modèle réseau sont l'apparition des associations [n:m], la plus grande cohérence des données, et la redondance d'information moindre.

Si le modèle hiérarchique repose sur la structure d'arbre (cas particulier d'un graphe ne présentant aucun cycle), le modèle réseau exploite totalement la structure de graphe. Autrement dit, si le noeud d'un arbre ne peut être en relation qu'avec un seul autre noeud

par le biais d'une relation parent-enfant, le noeud d'un graphe réseau peut être en relation avec plusieurs autres noeuds, sans offrir nécessairement de position hiérarchique, ce qui implique par conséquent la suppression de la notion de noeud racine.

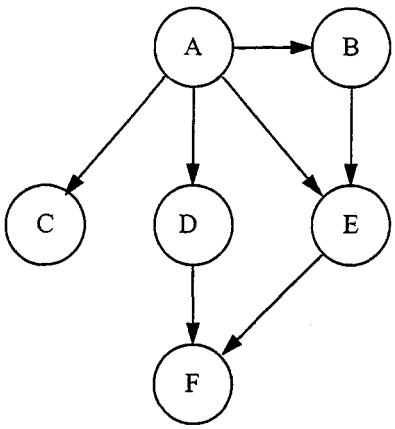


Figure 5: Exemple de graphe réseau

En CODASYL, il est seulement possible de définir des associations entre un article (*record*) appelé propriétaire et n articles membres. Ces associations, qui sont donc à un niveau purement hiérarchique, peuvent permettre de former aussi bien des arbres, des cycles, ou des réseaux, et sont appelées ensembles (*set*). L'ensemble est une relation binaire non porteuse de données, ayant les cardinalités [0:1] ou [1:1] de l'article membre vers l'article propriétaire, et les cardinalités [0:n] ou [1:n] de l'article propriétaire vers l'article membre [TAY 76]. Ce schéma correspond aux notions de schéma conceptuel et interne du modèle:

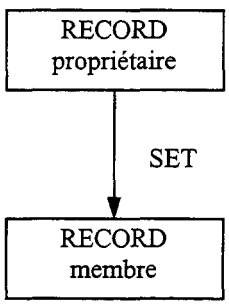


Figure 6: Association propriétaire-membre

En utilisant la notation de Bachman [BAC 69], une base de données réseau se représente au niveau des types à l'aide d'un graphe des types dont les sommets sont les noms des types d'articles et dont les arcs orientés du type propriétaire vers le type membre représentent les types d'ensembles.

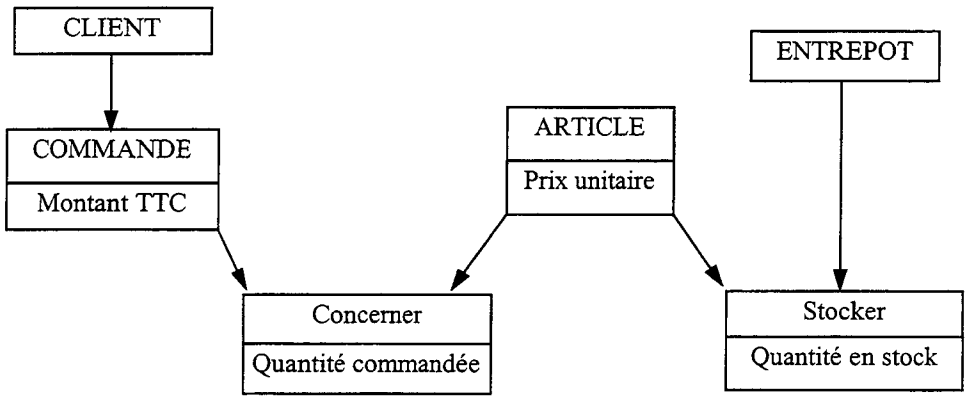


Figure 7: GENCOD — Modèle réseau

I.3.3 Conclusion

Bien qu'ils puissent rester très compétitifs au niveau interne, les modèles hiérarchique et réseau ne restent finalement que des modèles d'accès, directement issus de l'organisation des enregistrements dans les fichiers. Ils souffrent de la complexité des langages de manipulation des données historiquement associés à ces modèles (langages navigationnels), d'autant plus que tous les traitements ne sont pas réalisables, à moins de dupliquer les informations au dépend de la robustesse de la base (cohérence des données).

Le référentiel basé sur les cinq plus importants objectifs des SGBD nous permettra de mieux situer les domaines de prédilection de ces deux modèles. Nous faisons aussi apparaître dans ce référentiel un axe intégration des traitements, bien qu'il ne fasse pas partie des objectifs de base.

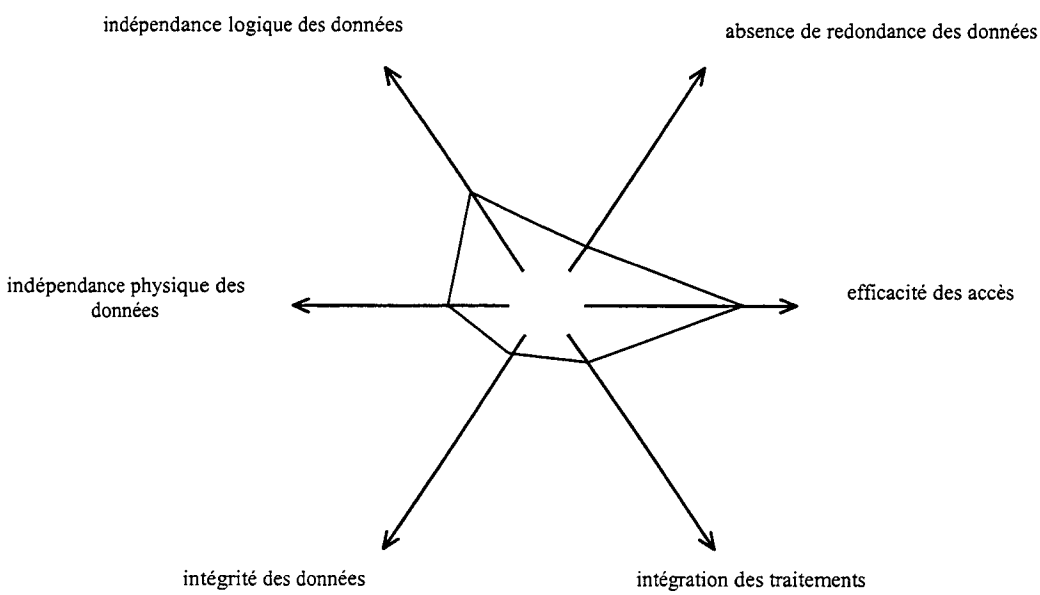


Figure 8: Référentiel des objectifs — SGBD de première génération

I.4 SGBD de seconde génération

Les deux modèles hiérarchique et réseau peuvent être qualifiés de modèles d'accès, fonctionnant sur un mode procédural, c'est-à-dire nécessitant de décrire la marche à suivre pour atteindre l'information. A l'inverse, le *modèle relationnel* apparu en 1970 [COD 70], fonctionne sur un mode non procédural et repose sur des bases mathématiques: la théorie des relations appliquées aux problèmes particuliers des bases de données, qui a pour but d'éliminer les comportements anormaux des relations lors des mises à jour, et les redondances entre les données. En première approche, le *modèle orienté objet* poursuit les mêmes objectifs que le modèle relationnel, en adaptant le concept de programmation orientée objet aux bases de données.

I.4.1 Le modèle relationnel

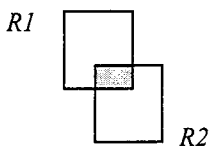
En termes mathématiques: étant donnés les ensembles E_1, E_2, ..., E_n, une relation $\mathcal{R}$ est définie comme un sous-ensemble du produit cartésien d'un certain nombre de ces ensembles, non nécessairement distincts, et est notée $\mathcal{R}(e_1, e_2, ..., e_n)$ avec $e_i \in E_i$.

- $E_1, E_2, ..., E_n$ sont les *domaines* de la relation
- n s'appelle le *degré* de la relation
- $\mathcal{R}$ est donc un ensemble de *tuples* distincts
- $e_1, e_2, ..., e_n$ s'appellent les *constituants* de la relation ou les *attributs* de la relation, et prennent leurs valeurs dans les domaines $E_1, E_2, ... E_n$
- il peut y avoir des relations différentes ayant les mêmes constituants qui prennent leurs valeurs dans les mêmes domaines, mais qui décrivent des phénomènes différents
- il peut y avoir dans une même relation des constituants prenant leurs valeurs dans les mêmes domaines

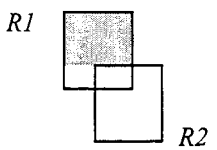
En pratique, une relation peut être représentée par une table, dont chaque ligne représente un *tuple*, pour laquelle l'ordre des lignes n'a pas de signification, et dans laquelle il n'est pas possible d'avoir deux lignes identiques. Une base de données relationnelle est donc perçue par l'utilisateur comme une collection de tables. La vue tabulaire des données reste indépendante de la manière dont l'information est stockée.

L'algèbre relationnelle est un système formel qui manipule des relations — les opérandes — par l'intermédiaire d'opérateurs ensemblistes usuels et d'opérateurs spéciaux définis pour les relations: *sélection*, *projection*, et *jointure*:

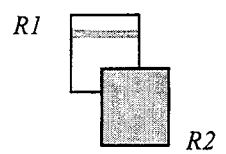
- $R_1 \cup R_2$ (union) est une relation contenant tous les tuples qui appartiennent à R_1, à R_2, ou aux deux.



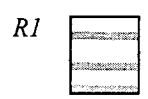
- $R_1 \cap R_2$ (intersection) est une relation contenant tous les tuples qui appartiennent à R_1 et à R_2 (sur des domaines identiques).



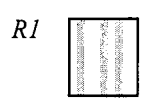
- $R_1 - R_2$ (différence ensembliste) est une relation contenant tous les tuples de R_1 qui n'appartiennent pas à R_2.



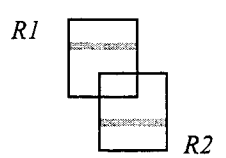
- $R_1 \times R_2$ (produit) est l'ensemble des couples constitués d'un élément de R_1 et d'un élément de R_2.



- sélection(R, P) est une relation contenant les tuples de R qui satisfont le prédicat P.



- projection(R, D_1, ..., D_n) est une relation qui ne contient que les domaines D_i de R.



- jointure(R_1, D_1, R_2, D_2) est une relation qui contient toutes les paires de tuples de R_1 et de R_2 qui ont les mêmes valeurs dans les domaines D_1 et D_2.

I.4.1.1 Dépendance fonctionnelle et clé de relation

La notion de *dépendance fonctionnelle* sert à décrire les interactions entre les différents constituants des relations, afin de caractériser les relations qui peuvent être décomposées (à l'aide de projections et de jointures) sans perte d'information [COD 70].

Soit la relation $\mathcal{R}(x, y, z, t)$ dont les constituants x, y, z, t prennent leurs valeurs dans les domaines D_x, D_y, D_z, D_t, non nécessairement distincts, alors le constituant y est dit fonctionnellement dépendant du constituant x, si et seulement si à toute valeur x de D_x correspond une seule et unique valeur y de D_y. Autrement dit, la valeur de x détermine celle de y, ou encore lorsque x est connu, y l'est aussi. La dépendance fonctionnelle de y par rapport à x est notée $x \rightarrow y$.

A titre d'exemple, pour la relation *ARTICLE* (Figure 9), on a les dépendances fonctionnelles:

Code article → *Prix unitaire*
Code article → *Libellé article*
Libellé article → *Prix unitaire*

Les dépendances fonctionnelles obéissent à plusieurs règles d'inférences triviales:

- la réflexivité: tout ensemble d'attributs détermine lui-même ou une partie de lui-même: $Y \subseteq X \Rightarrow X \rightarrow Y$
  Libellé article ⊆ (*Code article*, *Libellé article*) ⇒
  (*Code article*, *Libellé article*) → *Libellé article*
- l'augmentabilité: si X détermine Y, les deux ensembles d'attributs peuvent être enrichis par un même troisième: $X \rightarrow Y$ et $\Rightarrow XZ \rightarrow YZ$
  Code article → *Prix unitaire* ⇒
  (*Code article*, *Libellé article*) → (*Libellé article*, *Prix unitaire*)
- la transitivité: $X \rightarrow Y$ et $Y \rightarrow Z \Rightarrow X \rightarrow Z$
  Code article → *Libellé article* et *Libellé article* → *Prix unitaire* ⇒
  Code article → *Prix unitaire*

Il est possible à partir de la définition de dépendance fonctionnelle, de donner une définition de la notion intuitive de *clé*, c'est-à-dire l'ensemble minimum d'attributs qui déterminent tous les autres. Soit un sous-ensemble X des attributs d'une relation $\mathcal{R}(A_1..A_n)$, alors X est une clé si et seulement si:

$$1.\ X \rightarrow A_1 .. A_n \qquad 2.\ \neg\exists\ Y \subset X \mid Y \rightarrow A_1 .. A_n$$

I.4.1.2 Formes normales

Afin de permettre la décomposition des relations sans perte d'information pour aboutir à un schéma conceptuel représentant les entités et associations canoniques du monde réel, E.F. Codd a introduit [COD 72] à partir de la notion de dépendance fonctionnelle les trois premières formes normales. Les quatrième et cinquième formes normales ont été introduites plus tard [COD 74], mais sont peu ou pas utilisées et ne seront pas décrites ici.

Les dépendances fonctionnelles sont des règles indépendantes du temps que doivent vérifier les valeurs des attributs: il est donc nécessaire qu'une décomposition préserve ces règles. La troisième forme normale est importante car toute relation a au moins une décomposition dans cette forme, telle que cette décomposition préserve les dépendances fonctionnelles et soit sans perte d'information. Le processus de normalisation permet donc de structurer l'information en identifiant de façon précise les associations qui existent entre les informations et les contraintes qu'elles doivent satisfaire.

La *première forme normale* ne peut être justifiée que par un souci de simplicité, et permet d'obtenir des tables rectangulaires — l'intersection d'une ligne et d'une colonne ne contient qu'une seule valeur d'attribut —, et consiste donc à éviter les domaines composés de plusieurs valeurs. Autrement dit, il faut que chaque attribut qui ne fait pas partie de l'identifiant clé soit fonctionnellement dépendant de l'identifiant clé.

Pour une commande contenant plusieurs postes, la relation
COMMANDE(numéro de commande, poste$_1$<article$_1$, quantité$_1$>, ..., poste$_n$<article$_n$, quantité$_n$>)
n'est pas en première forme normale car les informations concernant les postes se répètent autant de fois qu'il y a de lignes de commande. Il faut donc éclater la relation en deux parties:
COMMANDE(numéro de commande)
POSTE(numéro de commande, article, quantité)

Exemple 7: Transformation d'une relation en première forme normale

Une relation est une *deuxième forme normale* si et seulement si elle est une première forme normale et si tous les attributs qui ne font pas partie de l'identifiant clé sont fonctionnellement dépendants de la totalité de celui-ci et non d'un sous-ensemble des constituants de l'identifiant clé. Ceci n'a aucun intérêt si l'identifiant clé est composé d'un seul constituant.

Pour un poste de la commande, la relation
POSTE(numéro de commande, nom de l'article, quantité,prix unitaire)
n'est pas en deuxième forme normale car le prix d'un article ne dépend que de l'article. Il faut donc éclater la relation en deux parties:
POSTE(numéro de commande, nom de l'article, quantité)
ARTICLE(nom de l'article, prix unitaire)

Exemple 8: Transformation d'une relation en deuxième forme normale

La *troisième forme normale* permet d'assurer l'élimination des redondances dues aux dépendances transitives: une relation est une troisième forme normale si et seulement si elle est une deuxième forme normale et si tout attribut n'appartenant pas à une clé ne dépend pas d'un attribut non clé. Si une relation ne possède qu'un seul identifiant clé, on peut dire que tout attribut n'appartenant pas à la clé ne dépend pas transitivement de la clé.

Pour un article, la relation
ARTICLE(nom de l'article, fabricant, référence constructeur)
n'est pas en troisième forme normale car l'attribut non clé *référence constructeur* détermine *fabricant*. Il faut donc l'éclater en:
ARTICLE(nom de l'article, référence constructeur)
REFERENCE(référence constructeur, fabricant)

Dans un autre ordre d'idée, pour une commande, la relation
FACTURE(numéro de facture, montant TTC)
n'est pas en troisième forme normale car le montant de la facture ne dépend pas directement du numéro de facture mais résulte d'un calcul. Il faut donc associer à la relation une formule (procédure) de calcul:
FACTURE(numéro de facture)
formule de calcul(montant TTC)
Mais dans ce cas de figure, recalculer le montant à chaque fois que l'on en a besoin peut être pénalisant (coût des transmissions dans le cas d'une base éclatée sur plusieurs sites par exemple), voire faux (si le prix des articles est mis à jour, une facture déjà éditée ne doit pas voir son montant modifié). Il est alors d'usage de ne pas normaliser la relation.

Exemple 9: Transformation d'une relation en troisième forme normale

La Figure 9 illustre le modèle conceptuel des données de l'application *GENCOD*. Outre le nom des différentes relations et leurs attributs, nous remarquons qu'il est décoré de *cardinalités* indiquant le nombre de fois qu'une entité est en relation avec une autre (par exemple un client peut passer plusieurs commandes, mais une commande n'est passée que par un et un seul client).

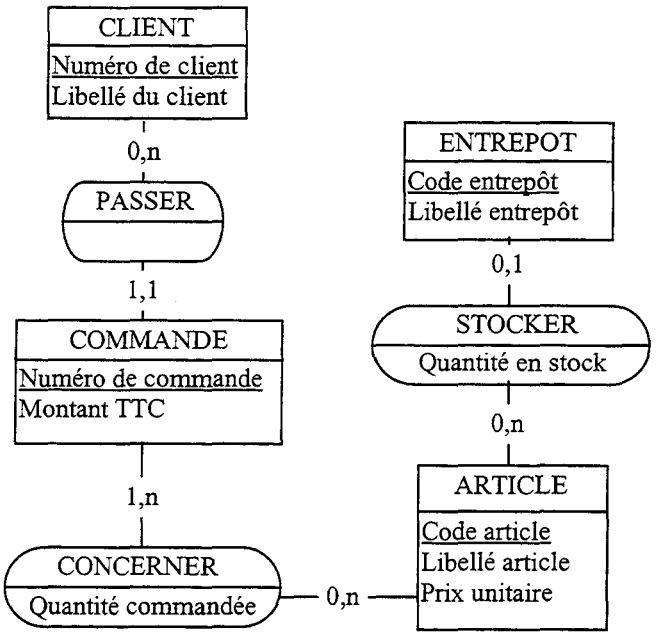


Figure 9: *GENCOD* — modèle relationnel

I.4.1.3 Expression des traitements

Même si l'apport du modèle relationnel lors de la définition des données est réel par rapport aux modèles hiérarchiques, ce modèle souffre encore du manque de la définition des traitements (manipulation des données) dans le schéma conceptuel. La méthode Merise [TAR 86] par exemple, reporte la définition des traitements (opérations à effectuer et leur synchronisation) dans le MCT (Modèle Conceptuel des Traitements). Cependant l'un des grands mérites du modèle relationnel a été d'avoir motivé le développement de multiples langages d'interrogation assertionnels basés sur l'algèbre relationnelle de Codd, qui permettent d'appliquer des séquences d'opérateurs spéciaux aux relations: le langage d'IBM *SQL* [IBM 81], dérivé de *SEQUEL* [CHA 76] semble en représenter l'ultime aboutissement:

```
CREATE TABLE Article(
    Code_Article INTEGER NOT NULL,Libellé_Article VARCHAR(20) NOT NULL,
    Prix_Unitaire DECIMAL(5,2) NOT NULL,Numéro_Entrepôt INTEGER NOT NULL,
        PRIMARY_KEY(Code_Article),
        FOREIGN KEY Entrepôt(Numéro_Entrepôt)
            REFERENCES Entrepôt ON DELETE RESTRICT)
```

Exemple 10: *GENCOD* — Requête SQL de la création de la relation *Article*

```
UPDATE Commande
    SET Montant_TTC = SELECT SUM(A.Prix_Unitaire×B.Quantite_Commandée)
    FROM Article A,Concerner B,Commande C
        WHERE C.Numéro_Commande=B.Numéro_Commande
            AND B.Code_Article=A.Code_Article
            AND NOT EXISTS(
                    SELECT * FROM Concerner D,Article E
                        WHERE D.Quantité_Commandée>E.Quantité_En_Stock)
```

Exemple 11: *GENCOD* — Requête SQL de la fonction de mise à jour du montant de la commande

I.4.2 Le modèle orienté objet

Le modèle orienté objet est une adaptation du paradigme de la programmation orientée objet appliqué aux systèmes de bases de données. Ce modèle est donc un modèle sémantique qui fournit des mécanismes pour modéliser les caractéristiques non seulement structurelles, mais également comportementales, d'une application. Le modèle est basé sur le concept d'*encapsulation* des données et du code opérant sur ces données dans un *objet*. Une collection d'objets ayant les mêmes structures et opérations est appelée une *classe*.

Un objet est un concept regroupant à la fois des propriétés déclaratives et des propriétés actives représentant une abstraction d'éléments du monde réel. Les propriétés déclaratives d'un objet, ou *attributs*, sont des valeurs caractérisant l'état de l'objet, tandis que les propriétés actives d'un objet, appelées *méthodes*, définissent son comportement, qui est constitué d'un ensemble d'actions que l'objet peut entreprendre au cours de son existence en fonction des événements qui surviennent [GAR 90]. Globalement, un objet est caractérisé par les actions qu'il subit et qu'il demande aux autres objets [BOO 91]. Une méthode associée à un objet ne peut être invoquée et exécutée qu'en réponse à la réception d'un message émis par un autre objet, seul mécanisme d'accès autorisé par les objets [HEW 77]. Finalement, l'*héritage* permet de définir à partir d'un objet existant, un nouvel objet comportant des propriétés similaires, en explicitant uniquement les différences du nouvel objet.

Un modèle orienté objet doit donc supporter les notions de classe, d'objet avec attributs et méthodes, et d'héritage. De plus, comme les DBMS traditionnels, il a besoin d'un langage d'interrogation. Nous citerons le cas d'*ODMG* [CAT 95], qui fait appel au langage orienté objet C++, et à un dérivé de SQL — *OQL* — qui possède par rapport à ce dernier la capacité de manipuler des objets complexes en invoquant leurs méthodes:

```
class Commande {
// Attributs encapsulés, ou privés, i.e. non accessibles de l'extérieur
        int Numéro_Commande ;
        float Montant_TTC ;
// intégrités référentielles
        Ref<Client> est_passée_par inverse passe_commande_de ;
        // nous devons trouver dans l'objet Client la même intégrité référentielle
        List<Ref<Concerner>> contient_le_poste inverse est_une_ligne_de_commande_de ;
// méthodes
        Calcul_Montant() ;
        ...
}
```

Exemple 12: *GENCOD* — Définition de la classe *Commande* en ODMG

```
select sum(a.Prix_Unitaire×b.contient_le_poste.Quantité_Commandée)
    from Article a,Commande b
        where a.Code_Article=b.contient_le_poste.Code_Article
            and not exists(
                select * from Article c,Commande d
                    where c.Quantité_En_Stock<d.contient_le_poste.Quantité_Commandée)
```

Exemple 13: *GENCOD* — Requête OQL du calcul du montant TTC de la commande

I.5 Conclusion

Par rapport aux modèles hiérarchique et réseau, le modèle relationnel tend à rendre transparente l'organisation physique des données. L'apport d'une théorie mathématique mais surtout de la normalisation des relations rend alors possible la définition de la plupart des traitements, la diminution de la redondance des données, et le respect de leur intégrité. D'autre part, l'apport d'une méthode comme Merise par exemple facilite grandement le développement des applications.

Le mérite du modèle orienté objet est quant à lui, de permettre la manipulation d'objets de grande complexité, car ces derniers sont stockés tels quels dans la base. Mais bien qu'il existe un consensus sur ce type de modèle, aucun standard n'a encore véritablement émergé.

Le référentiel basé sur les six plus importants objectifs des SGBD nous permettra de mieux situer les domaines de prédilection de ce deux modèles.

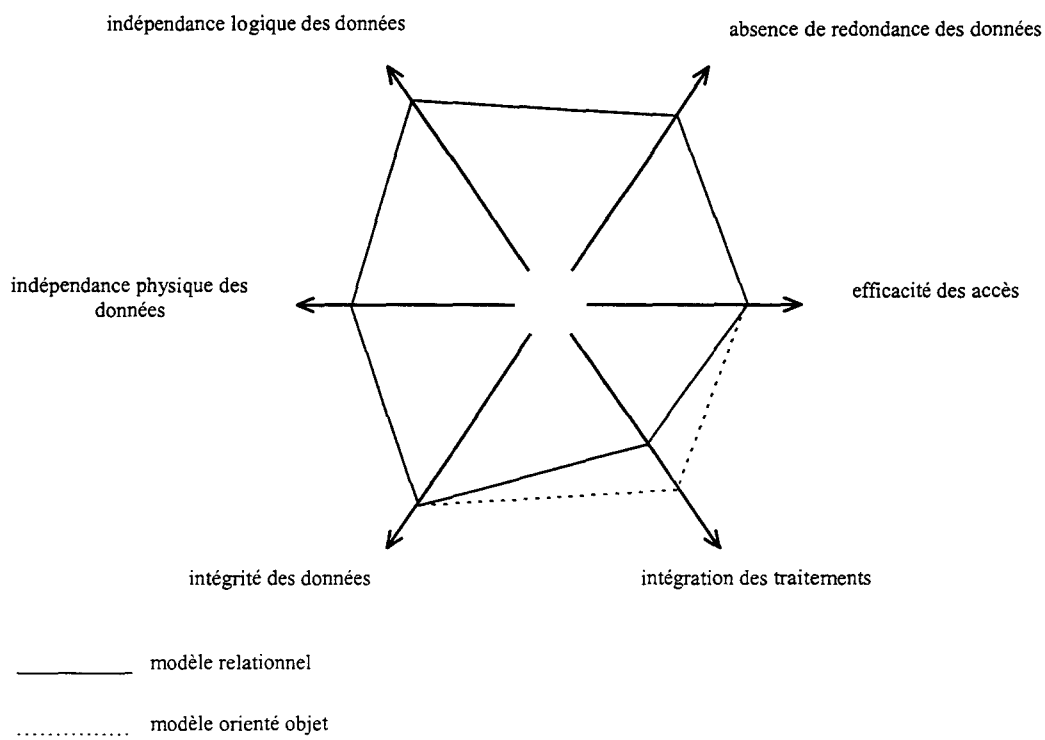


Figure 10: Référentiel des objectifs — SGBD de deuxième génération

CHAPITRE II

BASES DE DONNEES ACTIVES

Les systèmes de gestion de bases de données actives (Active DBMS — ADBMS) ont récemment fait l'objet de nombreuses recherches. Plusieurs systèmes et modèles ont été proposés, qui revendiquent le qualificatif « actif ». Parmi ceux-ci, nous pourrons citer par ordre alphabétique *Ariel* [HAN 91], *HiPAC* [CHA 89b], *Ode* [AGR 89], *POSTGRES* [STO 90b], *SAMOS* [GAT 94b], et *Starburst* [LAP 90].

Très succinctement, nous pouvons dire qu'un ADBMS est une extension d'un DBMS classique — ou « passif » par opposition — donnant la possibilité de spécifier un comportement réactif — ou dynamique — et par conséquence étant capable de réagir automatiquement à certaines situations apparaissant dans la base de données elle-même, ou provenant de son environnement extérieur. L'outil de spécification du comportement réactif est généralement basé sur l'emploi de règles appelées *règles-ECA*.

Ce chapitre présente les concepts relatifs à la spécification et à l'implémentation des fonctionnalités supplémentaires à apporter à un DBMS passif [CHA 93a] [DIT 95][HAN 92b], en vue d'obtenir un ADBMS.

II.1 Règles-ECA

Les *règles-ECA* (Event-Condition-Action) consistent en des *événements*, des *conditions*, et des *actions*. Le sens d'une telle règle est:

> *« Lorsqu'un événement arrive, tester une condition et si cette condition est satisfaite, exécuter une action ».*

L'origine des règles-ECA remonte aux travaux effectués en intelligence artificielle (système expert) [BRO 85], pour laquelle, et de manière très générale, une *règle de production* prend la forme:

condition (ou prémisse) $\rightarrow$ *conclusion*

La communauté travaillant dans le domaine des bases de données actives s'est inspirée de cette forme de règle et l'a transformée (bien que le terme de « règle de production » ait été conservé) en [DIT 95]:

prédicat → *action*

Une règle de ce type, qualifiée de *pattern-based*[1], est *tirée* lorsque le *prédicat* — ou pattern — est reconnu, et par conséquence, l'*action* est exécutée. Il faut noter que le prédicat n'est évalué que si une modification de données est intervenue antérieurement. Dans ce sens, une règle *pattern-based* est implicitement tirée lorsqu'une opération de type *insertion*, *destruction*, ou *modification* est effectuée dans la base de données.

La différence majeure entre une règle de production et une règle-ECA est introduite par la spécification explicite de l'événement qui a engendré le tir. Une règle-ECA prend alors la forme:

on *event*
if *condition*
then *action*

Dans ce cas, la règle est tirée lorsque l'événement survient. Une fois la règle tirée, la condition est évaluée (cette condition porte sur les données ou l'état de la base), et si cette condition est satisfaite, l'action est exécutée. Une telle règle est qualifiée d'*event-based*.

La plupart des ADBMS supportent les règles *event-based*, bien que certains ne supportent que les règles *pattern-based*, et d'autres les deux. Certains mécanismes peuvent exister pour *lier* — binding — les données aux événements et/ou aux conditions, ce qui permettra plus tard de paramétrer les conditions et/ou les actions. Finalement, d'autres mécanismes permettent de déterminer un ordre d'exécution des règles lorsque plusieurs d'entre-elles sont simultanément éligibles. Nous examinerons chacun de ces aspects dans les paragraphes suivants.

II.1.1 Spécification des événements

II.1.1.1 Définition

Une instance d'événement peut être considérée comme un couple (*<type événement>*,*<instant occurrence>*) dans lequel *<type événement>* dénote la description des occasions pour lesquelles une réaction devrait avoir lieu, et *<instant occurrence>* représente le moment où de telles occasions arrivent effectivement.

Un événement est une fonction du domaine temporel sur l'ensemble des valeurs booléennes {True et False}:

[1] Nous avons préféré garder dans le reste du document la terminologie anglo-saxonne afin de ne pas faire intervenir une traduction approximative.

$$E : T \rightarrow \{\text{True, False}\}$$

$$E(t) = \begin{cases} \textit{True} & \text{si un événement de type } E \text{ survient à l'instant } t \\ \textit{False} & \text{sinon} \end{cases}$$

Les événements peuvent être *primaires* ou *composés*. Les événements primaires, prédéfinis dans le système, sont directement caractérisés par l'instant où *« quelque chose »* se passe dans la base de données ou dans son environnement. Ils peuvent être classés en plusieurs catégories: les événements propres à la base de données, les événements temporels, et les événements explicites. Les événements propres à la base de données sont relatifs à des opérations telles que les *transactions*, les opérations *insertion*, *destruction*, *modification*, ou *sélection* (modèle relationnel), l'application de *méthodes* (modèle orienté objet). Les événements temporels sont relatifs au temps et sont de trois types: absolu (e.g. à 16 heures), relatif (e.g. 5 secondes après l'apparition d'un événement précis), ou périodique (e.g. toutes les 10 minutes). Les événements explicites sont détectés et signalés par les programmes d'application eux-mêmes (e.g. capteurs, messages). Les événements composés sont la résultante de la combinaison d'événements primaires et/ou d'événements composés, à l'aide d'opérateurs de disjonction (OU), de conjonction (ET), de séquence, ou de répétition. Ils sont caractérisés par l'instant où leur dernier composant arrive, ce qui implique que le seul paramètre réellement nécessaire à un événement est *<instant occurrence>*, mais il peut y en avoir d'autres (par exemple le composant à l'origine de l'événement). La Figure 11 illustre cette classification:

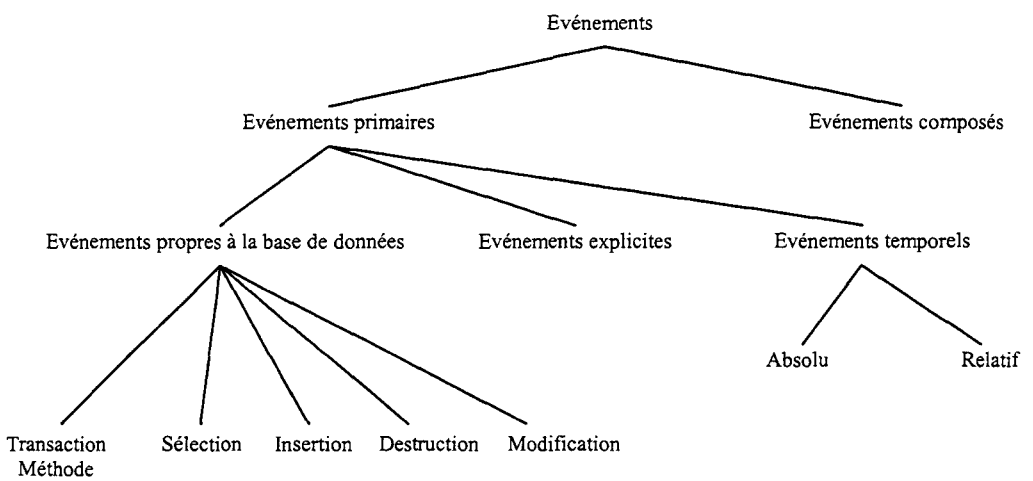


Figure 11: Classification des événements

Le fait qu'un événement soit assimilé à une occurrence « instantanée » peut poser quelques problèmes si le composant responsable de son origine est une opération (modification de données dans la base pour le modèle relationnel, application d'une méthode pour le modèle orienté objet) possédant intrinsèquement une durée d'exécution non nulle: pendant cet intervalle de temps, l'environnement n'a plus de sens car il n'est pas stable (les contraintes d'intégrité ne sont pas respectées). Il est alors d'usage de résoudre ce problème en transformant un intervalle de temps en deux événements logiques, repérant le début et la fin de l'intervalle:

after *insert*	// juste après la création d'une donnée
before *delete*	// immédiatement avant la destruction d'une donnée
begin-of *transaction*	// au tout début d'une transaction
end-of *transaction*	// à la fin d'une transaction

II.1.1.2 Composition des événements - Sémantique des opérateurs

Les événements primaires constituent les briques de base utilisées dans des spécifications d'événements plus expressives et souvent plus utiles [CHA 93c]. La sémantique des événements composés formés à partir des opérateurs de disjonction, de conjonction, et de négation[2] est la suivante:

- **OR** (∇): la disjonction de deux événements E_1 et E_2, notée $E_1 \nabla E_2$ survient lorsque E_1 survient ou lorsque E_2 survient.

$$(E_1 \nabla E_2)(t) = E_1(t) \vee E_2(t)$$

- **AND** (Δ): la conjonction de deux événements E_1 et E_2, notée $E_1 \Delta E_2$ survient lorsqu'à la fois E_1 et E_2 surviennent, indépendamment de l'ordre de leur arrivée.

$$(E_1 \Delta E_2)(t) = (E_1(t^1) \wedge E_2(t)) \vee (E_1(t) \wedge E_2(t^1)) \text{ avec } t^1 \leq t$$

  Notons par ailleurs que les opérateurs OR et AND sont commutatifs et associatifs:

$$(E_1 \nabla E_2)(t) = (E_2 \nabla E_1)(t)$$
$$(E_1 \Delta (E_2 \Delta E_3))(t) = ((E_1 \Delta E_2) \Delta E_3)(t) = (E_1 \Delta E_2 \Delta E_3)(t)$$

- **NOT** ($\neg$): la négation d'un événement, notée $\neg(E_2)[E_1, E_3]$ détecte l'absence d'occurrence de l'événement E_2 dans l'intervalle fermé $[E_1, E_3]$.

$$\neg(E_2)[E_1, E_3](t) = E_1(t^1) \wedge \sim E_2(t^2) \wedge E_3(t) \text{ avec } t^1 \leq t^2 \leq t$$

- **ANY**: la conjonction de m événements parmi n événements, notée Any(m, E_1, E_2, ..., E_n) où $m \leq n$, survient lorsque m événements parmi n événements *distincts* surviennent, indépendamment de leur ordre d'arrivée.

$$\text{ANY}(m, E_1, E_2, ..., E_n)(t) = E_i(t^1) \wedge E_j(t^2) \wedge ... \wedge E_k(t^m)$$
avec $t^1 \leq t^2 \leq ... \leq t^m = t$
pour $i, j, ..., k$ distincts et $\leq n$

Par exemple,

$$\text{ANY}(3, E_1, E_2, ..., E_n)(t) = E_i(t^1) \wedge E_j(t^2) \wedge ... \wedge E_k(t^3)$$
avec $(t^1 \leq t)$ et $(t^2 \leq t)$

[2] Nous noterons la *disjonction*, la *conjonction*, et la *négation* des événements avec les opérateurs ∇, Δ, et $\neg$ respectivement. Les symboles $\vee$, $\wedge$, et $\sim$ seront réservés pour les opérateurs booléens.

et ($t^3 = t$) et ($i \neq j \neq k$)
pour ($i \leq n$) et ($j \leq n$) et ($k \leq n$)

Pour spécifier m occurrences distinctes d'un événement E_1:

$ANY(m, E_1^*)(t) =$ $E_1(t^1) \wedge E_1(t^2) \wedge ... \wedge E_1(t^m)$
avec $t^1 \leq t^2 \leq ... \leq t^m = t$

- **SEQ** (;): la séquence de deux événements E_1 et E_2, notée $E_1;E_2$ survient lorsque E_2 survient et que E_1 est déjà survenu.

$$(E_1 ; E_2)(t) = E_2(t) \wedge E_1(t^1) \text{ avec } t^1 < t$$

- **Périodicité** (P, P*): un événement périodique est un événement E qui se répète à intervalle régulier (période constante et finie), et est noté $P(E_1, [t], E_2)$ où E_1 et E_2 sont des événements et t la constante positive de périodicité. P survient à chaque t dans $]E_1, E_2]$.

$P(E_1, [TI], E_2)(t) =$ $E_1(t^1) \wedge \sim E_2(t^2)$
avec $t^1 < t^2 \leq t$
et $t^1 + i \times TI = t$ pour $0 < i < t$

L'opérateur cumulatif P* accumule les instants d'occurrences de l'événement périodique:

$P^*(E_1, [TI], E_2)(t) =$ $E_1(t^1) \wedge \sim E_2(t)$
avec $t^1 + TI \leq t$

Dans *Snoop* [CHA 93b], S. Chakravarthy a dérivé du contexte général de détection des événements composés quatre contextes utilisables dans quatre grandes classes d'applications (la Figure 12 donne un exemple de la détection d'un événement composé X tel que $X = (E_1 \Delta E_2) ; E_3 ; (E_2 \Delta E_4)$):

- Contexte « récent »: utilisable dans les applications pour lesquelles les événements arrivent à cadence élevée et dont les multiples occurrences d'un même type d'événement remplacent les valeurs précédentes des données. Autrement dit, l'effet des occurrences de plusieurs événements du même type est substitué par l'occurrence la plus récente. Application typique: capteurs (e.g. monitoring). L'exemple 1 de la Figure 12 nous montre que e_1^1 (e_i^j est la $j^{ème}$ occurrence de l'événement E_i) ne peut pas être l'initiateur de X, puisqu'il est substitué par e_1^2 (occurrence plus récente que e_1^1). Dans le premier cas, e_2^1 est le terminateur de $E_1 \Delta E_2$ et e_2^2 est l'initiateur de $E_2 \Delta E_4$, qui est lui-même terminateur de X. Dans l'exemple 2, e_2^1 est substitué par e_2^2 (occurrence plus récente que e_2^1): e_3^1 ne peut donc pas participer à la détection de X puisqu'il arrive avant e_2^2 et c'est e_3^2 qui est pris en compte. e_2^2 fait également office d'initiateur de $E_2 \Delta E_4$, ce qui oblige e_4^1 à être substitué par e_4^2 pour respecter la chronologie des occurrences. On remarquera que si le début de la détection d'un événement composé peut être différé, la fin de la détection a lieu dès que possible (cette remarque reste valable pour les trois autres contextes).
- Contexte « chronique »: utilisable dans les applications pour lesquelles il existe une correspondance qui doit être maintenue entre les différents types d'événements et leurs occurrences. Application typique: dépendances causales

(e.g. dysfonctionnement d'un logiciel et sa mise à jour). Dans l'exemple 3, tous les initiateurs (e_1^1 pour $E_1\Delta E_2$ et e_2^1 pour $E_2\Delta E_4$) sont pris en compte dès que possible, alors qu'ils le sont le plus tardivement possible dans l'exemple 4.

- Contexte « continu »: utilisable dans les applications pour lesquelles la détection d'un événement composé dans une fenêtre de temps glissante doit être supportée. Application typique: finances (e.g. modification de la valeur du change d'une monnaie). Toutes les combinaisons des prises en compte (au plus tôt ou au plus tard) des initiateurs sont permises: ainsi dans les exemples 5 et 6, e_1^1 est l'initiateur de X, alors que c'est e_2^1 dans les exemples 7 et 8. Pour chaque groupe de deux exemples, l'initiateur de $E_2\Delta E_4$ peut être e_2^1 ou e_2^2.
- Contexte « cumulatif »: utilisable dans les applications pour lesquelles les occurrences multiples d'un événement ont besoin d'être regroupées et utilisées d'une manière adéquate lorsque l'événement arrive. Dans l'exemple 9 par exemple, on évite qu'une occurrence d'un événement soit à la fois initiateur et terminateur.

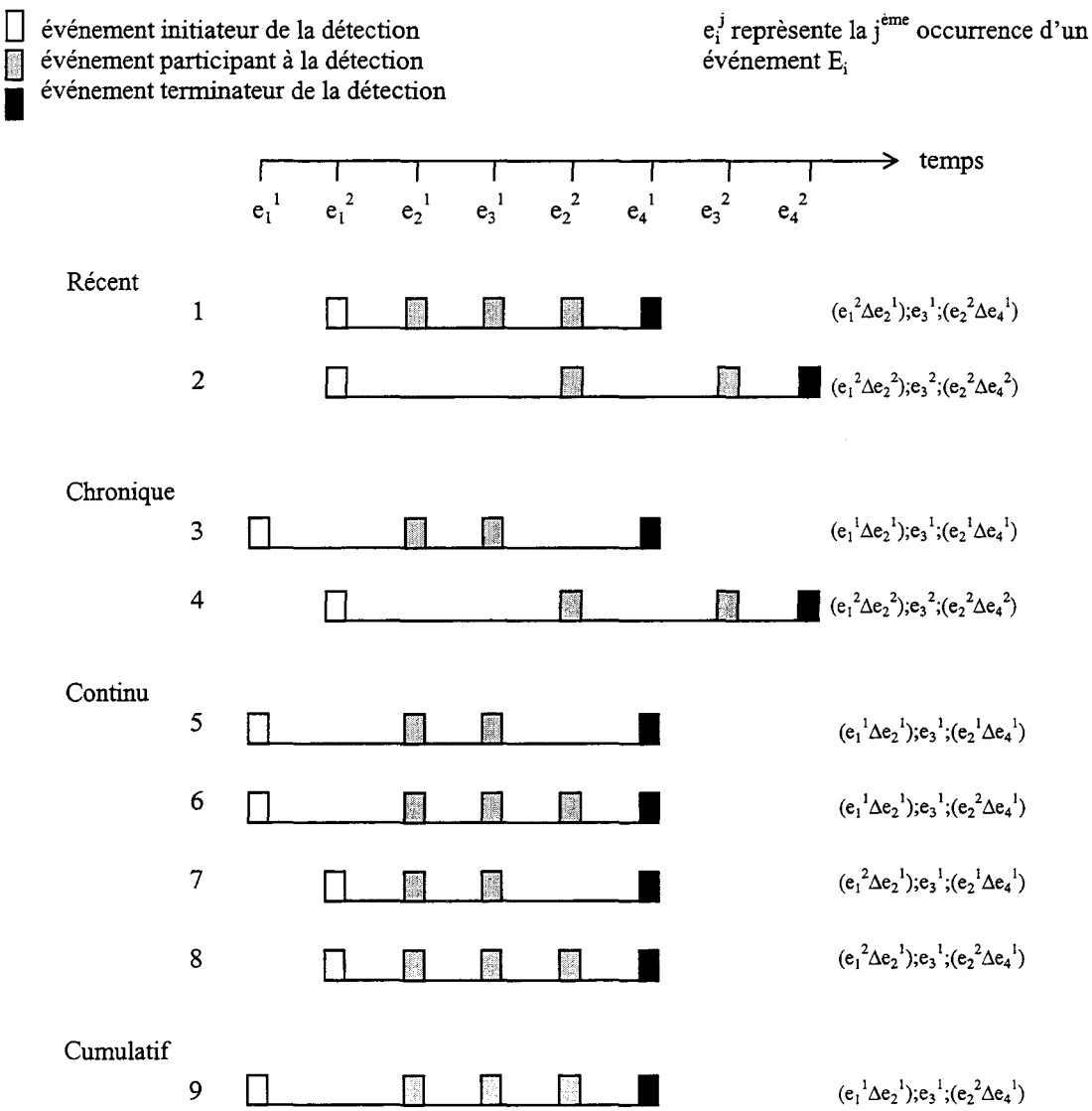


Figure 12: *Snoop* — Détection des événements composés

Si dans *Snoop*, la méthode employée pour la détection des événements composés fait appel aux arbres (chaque noeud de l'arbre est un événement primaire si ce noeud est une feuille, ou un événement composé sinon, la racine représentant la détection de l'événement global), d'autres auteurs ont utilisé les réseaux de Petri comme moyen de détection [GAT 94a] en modélisant les trois opérateurs de composition (disjonction, conjonction, et séquence) de la manière suivante, qui n'est pas sans rappeler le mode « chronique » de *Snoop*:

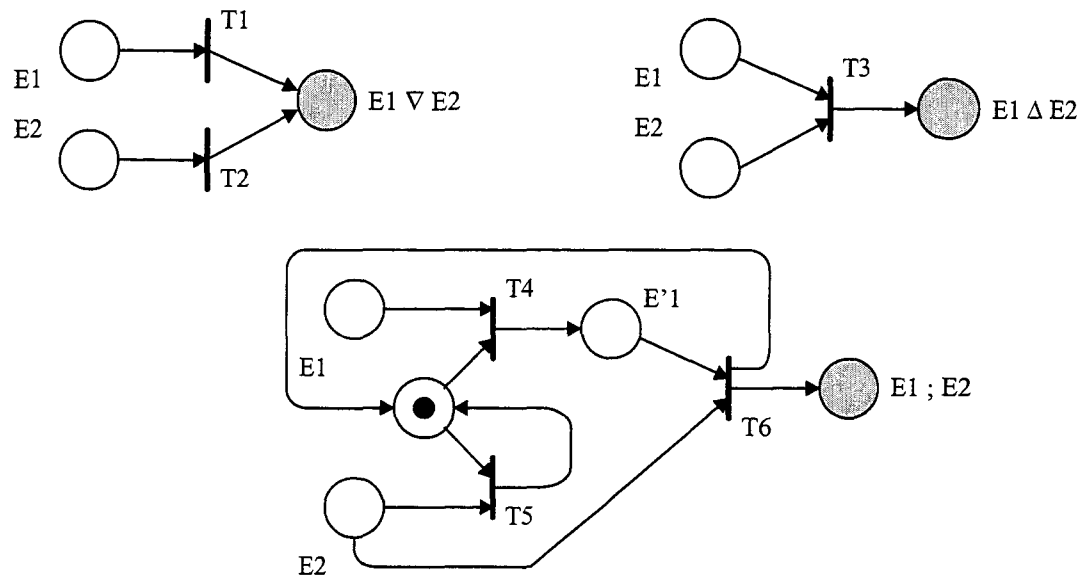


Figure 13: *SAMOS* — Détection des événements composés à l'aide de réseaux de Petri

Nous remarquerons que si l'occurrence d'un événement est instantanée, la détection d'un événement composé rend nécessaire la mémorisation de cette occurrence. Par exemple dans *Snoop*, la mémorisation est réalisée par le marquage d'un noeud de l'arbre, et dans *SAMOS*, par l'ajout d'un jeton dans une place d'un réseau de Petri. On peut aussi imaginer un mécanisme à base de messages grâce auquel l'occurrence d'un événement est « postée » à destination du détecteur d'événements composés. Cette mémorisation disparaît lorsque l'événement ne peut plus intervenir comme initiateur ou comme terminateur d'un événement composé.

II.1.1.3 Exemples

Une règle explicitement tirée suite à un événement concernant une modification dans la base (modèle relationnel) prend la forme générale:

define rule *ReconnaîtreLesNouveauxEmployés*
on insert to *Employé*
if ...
then ...

dans laquelle *Employé* est une relation contenant les informations sur les employés de la société.

La même règle, dans le modèle orienté objet, prend la forme:

define rule *ReconnaîtreLesNouveauxEmployés*
on *Employé.Nouveau()*
if ...
then ...

dans laquelle *Employé.Nouveau* est une méthode définie pour la classe d'objets *Employé*.

Une règle référencée par des événements temporels pourra ressembler à (pour le modèle relationnel):

define rule *PréparerLeCafé*
on every *day* **at** *10 AM*
if ...
then ...

dans laquelle un instant répétitif (*every day*) et un instant absolu (*at 10 AM*) sont présents.

II.1.2 Spécification des conditions

La partie *condition* d'une règle-ECA spécifie un prédicat ou une requête sur les données de la base. La condition sera satisfaite si le prédicat est vrai, ou si la requête a produit au moins une donnée répondant à la condition (i.e. le résultat de la requête n'est pas vide). Dans le cas des règles *event-based*, la condition peut être omise, auquel cas elle est toujours satisfaite.

define rule *TesterNomEmployé*
on insert to *Employé*
if *Employé.Nom* = 'Dupont'
then ...

est un exemple de condition d'une règle à la fois *event-based* et *pattern-based*.

Certains ADBMS possèdent également un mécanisme appelé *condition de transition*, grâce auquel une règle tirée par une modification de données peut faire référence à la donnée modifiée, ou à l'état de la base avant la modification:

define rule *TesterSalaire*
if *Employé.Salaire* > 1.1 × **previous** *Employé.Salaire*
then ...

est un exemple de condition d'une règle *pattern-based* uniquement, associée à une condition de transition: ici la règle est tirée lorsqu'il existe une nouvelle donnée dans la colonne *Salaire* dont la valeur est au moins supérieure de 10 % à la précédente valeur.

II.1.3 Spécification des actions

La partie *action* d'une règle-ECA spécifie les opérations à effectuer lorsque la règle est tirée et que la condition est satisfaite. Une action consiste généralement en une *insertion*, une *destruction*, ou une *modification*, mais nous pouvons également rencontrer n'importe quelle séquence arbitraire de *sélections* ou de modifications. Une action peut

généralement comporter un ordre de fin de transaction: soit pour la valider, soit pour l'annuler.

define rule *FavoriserEmployé*
on insert to *Employé*
then delete from *Employé* **where** *Employé.Nom* = **new**.*Nom*

cette règle *event-based* est tirée lorsqu'un nouvel employé est créé, et son action consiste à supprimer les employés ayant le même nom.

II.1.4 Attachement Evénement-Condition-Action

L'attachement a déjà été cité dans les paragraphes précédents (binding et condition de transition): il consiste à créer un *lien* entre une donnée qui satisfait le prédicat d'une règle et le comportement de l'action de la règle. Au moment de l'exécution de la règle, des valeurs peuvent être référencées dans l'action, qui correspondent à des variables utilisées dans la condition.

Cette notion d'attachement a donné naissance à un grand nombre d'implémentations différentes dans les ADBMS, comme le montre ces deux exemples tirés de *POSTGRES* et de *Starburst*:

define rule *DistribuerSalaire* // POSTGRES
on delete from *employé*
then update *employé* (*employé.salaire* = *employé.salaire* + 0,1 × **old**.*salaire*)

l'action de cette première règle relève de 10 % du salaire de l'employé supprimé le salaire de tous les autres employés.

define rule *MoyenneTropGrande* // Starburst
on update to *employé.salaire*
if (**select avg**(*salaire*) **from new-updated**) > 100
then rollback

l'action de cette seconde règle annule la transaction lorsque la moyenne des salaires mis à jour excède 100.

II.1.5 Les règles-ECA dans les différents systèmes

Les Tableau 1, Tableau 2, et Tableau 3 présentent les particularités exploitées par les différents ADBMS cités en exemple. Nous constatons qu'*Ariel*, *POSTGRES*, et *Starburst* respectent au plus près les spécifications initiales des règles-ECA, bien que quelques opérations de composition soient permises au niveau des événements, alors que *HiPAC*, *Ode*, et *SAMOS* apportent des compléments certains.

Système	Types d'événement			Particularités
	temporel	explicite	composition	
Ariel (relationnel)	énumération d'instants	sur insertion, destruction, ou modification	non	event-based et/ou pattern-based
HiPAC (orienté objet)	1. absolu 2. relatif 3. périodique	1. sur insertion, destruction, ou sélection 2. sur fin de transaction	1. disjonction 2. séquence	événements externes (messages, capteurs)
Ode (orienté objet)	oui	sur insertion, destruction, ou modification	1. relative 2. antériorité 3. séquence 4. chaque (any)	pattern-based uniquement
POSTGRES (relationnel)	date et heure	sur insertion, destruction, modification, ou sélection	disjonction	
SAMOS (orienté objet)	spécification explicite	oui	1. disjonction 2. conjonction 3. séquence 4. absence	
Starburst (relationnel)	non	sur insertion, destruction, ou modification	disjonction	

Tableau 1: Comparatif des événements sur différents systèmes

Système	Types de condition		
	prédicat	requêtes	condition de transitions
Ariel (relationnel)	oui	non	détection d'insertion et de modification
HiPAC (orienté objet)	non	oui	passage de paramètres de l'événement à la condition
Ode (orienté objet)	oui	non	non
POSTGRES (relationnel)	oui	non	non
SAMOS (orienté objet)	oui	non	non
Starburst (relationnel)	oui	non	détection de modification

Tableau 2: Comparatif des conditions sur différents systèmes

Systèmes	Types d'action
Ariel (relationnel)	sélection et/ou modification
HiPAC (orienté objet)	opérations arbitraires sur la base et/ou appel de procédures
Ode (orienté objet)	séquence arbitraire (méthode)
POSTGRES (relationnel)	sélection et/ou modification
SAMOS (orienté objet)	séquence arbitraire (méthode)
Starburst (relationnel)	sélection et/ou modification

Tableau 3: Comparatif des actions sur différents systèmes

II.1.6 Modes de couplage

Les *modes de couplage* [MCA 89] déterminent la manière dont les événements, les conditions, et les actions interagissent avec les transactions. Soient *E*, *C*, et *A* dénotant respectivement les événements, les conditions, et les actions. Deux modes de couplage sont

définis: le *mode de couplage E-C*, qui détermine l'instant où la condition est évaluée par rapport à l'instant où l'événement valide la règle, et le *mode de couplage C-A*, qui détermine l'instant où l'action est exécutée par rapport à l'instant où la condition est évaluée. Chacun des modes de couplage peut être *immédiat*, indiquant une exécution immédiate, *différé*, indiquant une exécution à la fin de la transaction courante, ou *séparé*, indiquant une exécution dans une transaction séparée. Les combinaisons de modes de couplage, résumées dans le Tableau 4, ne sont pas toutes permises: par exemple la combinaison *<couplage E-C séparé, couplage C-A différé>* pose des problèmes évidents de synchronisation entre condition et action. Les modes de couplage et les combinaisons possibles utilisés dans les différents systèmes sont résumés dans le Tableau 5.

	Couplage C-A		
Couplage E-C	*immédiat*	*différé*	*séparé*
immédiat	1. la condition est évaluée et l'action est exécutée après l'événement	2. la condition est évaluée après l'événement et l'action est exécutée à la fin de la transaction	3. la condition est testée après l'événement et l'action est exécutée dans une transaction séparée
différé	4. non autorisé: la condition est évaluée à la fin de la transaction et l'action ne peut plus être exécutée dans la même transaction	5. la condition est évaluée et l'action est exécutée à la fin de la transaction	6. la condition est évaluée à la fin de la transaction et l'action est exécutée dans une transaction séparée
séparé	7. la condition est évaluée et l'action est exécutée dans une transaction séparée	8. non autorisé: l'exécution de l'action à la fin de la transaction courante ne peut pas être synchronisée avec l'évaluation de la condition dans une transaction séparée	9. la condition est testée dans une 1ère transaction séparée et l'action est exécutée dans une 2nde transaction séparée

Tableau 4: Combinaisons des modes de couplage E-C et C-A

Systèmes	Mode de couplage E-C	Mode de couplage C-A	combinaisons selon Tableau 4
Ariel	immédiat (défaut), différé	immédiat, différé	1, 2, 5
HiPAC	immédiat, différé, séparé	immédiat, différé, séparé	1, 2, 3, 5, 6, 7, 9
Ode	immédiat, différé	immédiat, séparé	1 (contraintes fortes) 4 (contraintes lâches) [3] 3 (triggers)
POSTGRES	immédiat	immédiat	1
SAMOS	immédiat, différé, séparé	immédiat, différé, séparé	1, 2, 3, 5, 6, 7, 9
Starburst	différé	différé	5

Tableau 5: Modes de couplages supportés par les différents systèmes

Nous verrons dans le Paragraphe II.2 que les modes de couplage et l'algorithme de traitements des règles-ECA sont intimement liés, au sens où l'algorithme retenu implique l'utilisation obligatoire de telle ou telle combinaison de modes de couplage.

[3] Ode utilise la combinaison 4 non autorisée qui, d'après ses auteurs, offre un fonctionnement équivalent à celui de la combinaison 5.

II.1.7 Priorités

L'un des aspects importants des règles-ECA est la *résolution des conflits*, c'est-à-dire le choix d'une règle particulière à exécuter lorsque plusieurs règles sont tirées simultanément [AGR 91]. Dans beaucoup d'ADBMS, ce choix est fait de façon plus ou moins arbitraire, bien que quelques uns d'entre-eux offrent à l'utilisateur la possibilité d'influencer la résolution des conflits. Cette dernière est essentiellement implémentée en faisant appel à la notion de priorité: une règle sera tirée si et seulement si elle est prioritaire sur une autre règle selon un critère arbitrairement choisi. Le Tableau 6 résume les stratégies utilisées par les différents ADBMS proposés. Nous verrons dans le Paragraphe II.2 que la notion de priorité peut être implicitement utilisée dans les algorithmes de traitement des règles-ECA.

Système	Résolution des conflits (priorités)
Ariel	priorités numériques
HiPAC	exécution concurrentes des règles: aucune priorité entre règles n'est requise
Ode	aucune résolution des conflits prévue
POSTGRES	priorités numériques
SAMOS	processus démon à 3 priorités, par ordre décroissant: règles avec mode de couplage détaché, mode de couplage immédiat, mode de couplage différé
Starburst	ordre partiel: pour 2 règles, l'une possède une priorité supérieure à l'autre, mais aucun ordre n'est requis

Tableau 6: Stratégie de résolution des conflits

II.2 Algorithmes de traitement des règles-ECA

Après avoir défini un ensemble de règles-ECA, il est essentiel de prévoir un mécanisme d'exécution, car le comportement de ces règles peut se révéler complexe. Lorsqu'une règle est éligible, elle peut être déclenchée: il apparaît donc un indéterminisme d'une part sur le choix de la date de déclenchement de la règle, et d'autre part sur le choix de la règle à déclencher (si plusieurs règles sont éligibles simultanément). Or il est clair que l'exécution d'un DBMS (actif ou non) doit être déterministe: l'ordonnancement du déclenchement des règles devient alors nécessaire.

II.2.1 Algorithme de base

L'algorithme de base a été proposé à la suite des travaux concernant les systèmes experts, et est connu sous l'appellation *cycle de reconnaissance et d'action* (recognize-act cycle) [FOR 82]. En utilisant la terminologie associée aux bases de données actives (règle de production: *prédicat* → *action*), il a directement été modifié en:

évaluation initiales des prédicats
tant qu'un prédicat au moins est vérifié
résoudre les conflits (sélectionner une règle de production parmi celles dont le prédicat est vrai)
exécuter l'action de la règle choisie
ré-évaluer les prédicats
fin

Rappelons qu'une règle de production est composée d'un prédicat qui prend sa valeur dans {VRAI, FAUX} et d'une action, tandis qu'une règle-ECA peut être composée d'une condition évaluée par une requête sur les données de la base et d'une action. La requête produisant un ensemble de tuples (si cet ensemble est vide, la condition est fausse, et s'il comporte au moins un tuple, la condition est vraie), cette différence va alors modifier quelque peu l'algorithme de base en:

évaluation initiales des conditions
tant qu'une condition est vraie (une requête au moins produit un ou plusieurs tuples)
résoudre les conflits (sélectionner une règle-ECA parmi celles dont la condition est vraie)
exécuter l'action de la règle choisie pour chacun des tuples de la requête associée à la condition
ré-évaluer les conditions
fin

L'algorithme modifié fait apparaître que l'action d'une règle est exécutée pour toutes les instances de la règle (chacun des tuples), plutôt que sur une seule dans l'algorithme de base. Tirer une règle pour toutes les instances plutôt que pour une seule est connu sous le vocable « *tirer à saturation* » [MAI 88][KIE 90]. Le phénomène de saturation introduit le *tir orienté par les ensembles* (set-oriented firing), qui regroupe les tirs de toutes les instances dans une seule transaction, ce qui semble naturel dans les systèmes relationnels, et que nous retrouverons dans les systèmes orienté objet. Nous appellerons cette nouvelle notion la *granularité* de la règle.

Deux problèmes importants apparaissent dans l'expression de l'algorithme de base: le premier concerne l'influence des modes de couplage, le second la résolution des conflits. Ces deux points ont déjà été abordés dans les Paragraphes II.1.6 et II.1.7 et les solutions mises en oeuvre dans les différents systèmes étudiés sont présentées dans le paragraphe suivant.

II.2.2 Exemples d'implémentation

II.2.2.1 Ariel

Ariel [HAN 92a], utilise le modèle relationnel, dans lequel les données sont mémorisées dans les tables de la base, et les règles dans un catalogue de règles. Les deux particularités principales d'*Ariel* sont le concept de *transition* d'une part et la notion *d'événements logiques* d'autre part (Cf Paragraphe II.1.1.1).

Une transition est définie comme étant une modification dans la base, résultant de l'exécution d'une simple commande ou d'un bloc **do...end** contenant une liste de commandes (l'imbrication des blocs est interdite). Le concepteur a donc un contrôle total sur l'instant où la transition apparaît: s'il encadre toutes les commandes d'une transaction dans un bloc **do...end**, la transaction complète est une transition (la granularité est maximale), sinon chaque commande est considérée comme une transition (la granularité est minimale). Les blocs sont utilisés pour permettre la modification de la base par de multiples commandes en assurant l'intégrité des données (car l'ensemble des instructions contenues dans un bloc **do..end** est alors considéré comme une primitive), qui pourrait être violée pendant une mise à jour.

Le tir des règles *event-based* est effectué à la suite de l'apparition d'événements logiques plutôt que physiques. Le comportement d'un tuple *t* mis à jour par une transition appartient à l'une des quatre catégories illustrées dans le Tableau 7, où *i*, *m*, et *d* représentent respectivement l'insertion, la modification, et la destruction. Mis à part les opérations simples *i*, *m*, et *d*, nous remarquerons que la première opération d'une séquence ne peut être que *i* ou *m* (le tuple n'existe pas ou existe déjà), tandis que la dernière opération ne peut être suivie d'une autre opération s'il s'agit d'une opération *d* (le tuple n'existant plus).

type de mise à jour	description	effet de bord
im^*	insertion de *t* suivie par 0 ou plusieurs modifications	insertion
im^*d	insertion de *t* suivie par 0 ou plusieurs modifications puis par une destruction	pas d'opération
m^+	*t* existant au début de la transition, *t* est modifié une ou plusieurs fois	modification
m^*d	*t* existant au début de la transition, *t* est modifié 0 ou plusieurs fois puis détruit	destruction

Tableau 7: Comportements d'un tuple et effets de bord

Ceci montre comment l'effet de bord d'une séquence de mises à jour d'un tuple peut être résumé à une simple opération d'*insertion*, de *destruction*, ou de *modification* (ou pas d'opération du tout). En règle générale, l'interprétation des événements en événements logiques plutôt que physiques est plus intuitive et plus facile à utiliser, puisque nous sommes uniquement concernés par les effets définitifs sur la base, et non par les effets intermédiaires.

Le traitement des règles est basé sur l'algorithme *recognize-act cycle*. L'étape *évaluation des conditions* recherche l'ensemble des règles qui sont éligibles. L'étape *résolution des conflits* sélectionne une seule règle depuis cet ensemble. Finalement l'étape *exécution de l'action* exécute l'action de la règle. Le cycle se répète jusqu'à ce qu'il n'y ait plus aucune règle éligible, à moins que l'utilisateur ne commande un arrêt explicite.

répéter jusqu'à *aucune règle à tirer* ***ou*** *arrêt explicite demandé*
  évaluation les conditions (des règles éligibles)
  résolution des conflits (sélectionner une règle)
  exécution de l'action (de la règle sélectionnée)
fin

Chaque règle possède une priorité de type numérique. Ces priorités sont utilisées pour aider le système dans l'élection d'une règle lorsque plusieurs d'entre elles sont éligibles: la règle de plus haut niveau de priorité sera retenue lors de la résolution des conflits. Pendant la phase de résolution des conflits, *Ariel* sélectionne une règle à exécuter en utilisant les critères suivants (après chaque étape, s'il ne reste qu'une seule règle éligible, celle-ci est choisie pour être exécutée, sinon l'ensemble des règles encore en considération est transmis à l'étape suivante):

- *Sélectionner la(les) règle(s) dont la priorité est maximale.*
- *S'il y a conflit (i.e. plus d'une règle sélectionnée), sélectionner la(les) règle(s) déclenchée(s) le plus récemment possible.*
- *S'il y a toujours conflit, sélectionner la(les) règle(s) dont la condition est la plus sélective (la sélectivité est estimée au moment où la règle est compilée par l'optimiseur de requêtes de l'ADBMS).*
- *S'il y a toujours conflit, sélectionner une règle au hasard.*

II.2.2.2 HiPAC

Le traitement des règles dans *HiPAC* [CHA 89a] est invoqué à chaque fois qu'un événement arrive et déclenche le tir d'une ou de plusieurs règles, mais *HiPAC* réagit de manière différente, par rapport aux autres ADBMS, lorsque des règles multiples sont tirées simultanément: plutôt que de sélectionner une règle particulière en utilisant un critère quelconque pour résoudre les conflits, *HiPAC* exécute toutes les règles concurremment. Si pendant cette exécution d'autres règles sont tirées, celles-ci s'exécuteront également en parallèle. L'algorithme de traitement devient alors:

- Des événements quelconques déclenchent le tir des règles R_1, R_2, ..., R_n
- Pour chaque règle R_i, ordonnancer une transaction pour:
  $\Rightarrow$ Evaluer la condition de R_i
  $\Rightarrow$ Si la condition est satisfaite, ordonnancer une transaction pour exécuter l'action de R_i

L'ordonnancement des transactions d'évaluation des conditions est basé sur le mode de couplage E-C, tandis que l'ordonnancement des transactions d'exécution des actions l'est sur le mode de couplage C-A: le mode immédiat engendre immédiatement une sous-transaction imbriquée, le mode différé engendre une sous-transaction imbriquée à partir du point de validation de la transaction courante, et le mode détaché engendre une transaction de même niveau. Notons qu'à la fois l'évaluation des conditions et l'exécution des actions peuvent générer des événements qui invoquent l'algorithme de traitement de façon récursive. *HiPAC* utilise également une priorité entre règles de type ordre partiel, qui permet d'influencer l'ordre de sérialisation des sous-transactions dans les files d'attente de l'ordonnanceur.

II.2.2.3 Ode

Ode [GEH 91] permet l'utilisation de deux types de règles: *contraintes* et *triggers*, ayant deux sémantiques d'exécution différentes. Les contraintes sont associées à une classe d'objets, sont tirées par l'invocation d'une méthode quelconque sur un objet quelconque d'une classe, et consistent en une condition et une action (où l'action est exécutée si la condition est fausse). Par exemple la contrainte suivante permet de fixer le salaire d'un employé à 100 si ce salaire est plus grand que 100:

class *Employé*
constraint: $Salaire < 100 \rightarrow Salaire = 100$

L'algorithme de traitement d'un ensemble de contraintes est:

Pour chaque contrainte tirée C_1, C_2, ... C_n:

- Evaluer la condition de C_i
- Si la condition est fausse, exécuter l'action de C_i
- Réévaluer la condition de C_i
- Si la condition est toujours fausse, annuler la transaction

Une contrainte peut recevoir le qualificatif de *forte* ou *lâche*. Les contraintes fortes sont traitées immédiatement à l'invocation de la méthode (couplage E-C immédiat), tandis que les contraintes lâches le sont à la fin de la transaction courante (couplage E-C différé). Par conséquence, les contraintes fortes ont l'avantage de garantir que les objets sont toujours dans un état consistant à tout instant (sauf peut-être pendant une opération de modification), tandis que les contraintes lâches permettent la violation temporaire des contraintes d'intégrité, ce qui arrive fréquemment dans le cas où deux objets sont mutuellement dépendants. En règle générale, les contraintes fortes seront utilisées lorsqu'un seul objet entre en jeu, les contraintes lâches lorsque plusieurs objets apparaîtront.

L'ordre d'évaluation des conditions est arbitraire (i.e. il n'existe pas de priorité). L'exécution des actions peut entraîner l'invocation de méthodes qui appellent alors récursivement l'algorithme de traitement (mais pas forcément pour les mêmes objets et/ou les mêmes contraintes). Notons finalement que la sémantique d'exécution des règles a été particulièrement implémentée pour renforcer les contraintes d'intégrité dans la base: toutes les règles successivement exécutées doivent fournir un état de la base pour lequel toutes les conditions sont vraies, sinon la transaction est annulée. Cependant, rien n'interdit les violations temporaires de contraintes d'intégrité, qui doivent juste être validées par une action correcte avant que la transaction soit elle-même validée.

Les triggers sont constitués comme les contraintes d'une condition et d'une action, mais à la différence des contraintes, ils doivent être *activés* sur un objet particulier. Une fois activé, un trigger peut être déclenché par n'importe quelle méthode relative à l'objet. Lorsque la condition d'un trigger est vraie, la transaction courante engendre une autre transaction pour exécuter l'action du trigger (les triggers utilisent les modes de couplages E-C immédiat et C-A détaché). Les triggers peuvent être désactivés, ou même désignés comme activables une et une seule fois (i.e. ils sont désactivés automatiquement).

Finalement, les triggers seront utilisés lorsqu'il n'existe pas de contrainte d'intégrité à respecter, puisqu'intrinsèquement du fait du mode de couplage C-A détaché, l'action d'un trigger peut être exécutée alors que la transaction à l'origine du déclenchement du trigger aura été depuis longtemps validée.

II.2.2.4 POSTGRES

Dans *POSTGRES* [STO 90a], le traitement des règles est invoqué immédiatement après une quelconque modification portant sur un tuple. Aussi, lorsque l'action d'une règle est une séquence arbitraire d'opérations sur la base, chacune de ces opérations peut (immédiatement) déclencher d'autres règles. Par conséquence, le traitement des règles est intrinsèquement récursif et synchrone (analogue au mécanisme d'appel de procédures), plutôt que séquentiel comme pour l'algorithme *recognize-act cycle*:

- Un tuple ayant été modifié,
- La modification déclenche les règles R_1, R_2, ... R_n, et les conditions de ces règles ayant été satisfaites,
- Pour chaque règle R_i, l'action associée est exécutée.

Il n'existe pas de mécanisme de résolution des conflits: les règles sont exécutées dans un ordre totalement arbitraire. Si dans l'action d'une règle une annulation de transaction intervient, le traitement des règles s'arrête et la transaction complète est annulée.

II.2.2.5 SAMOS

SAMOS [GEP 95] ne gère pas les transactions de manière concurrente (comme *HiPAC*) ou de manière purement séquentielle (comme *Ariel*, *Ode*, *POSTGRES*, *Starburst*), mais utilise le concept de transactions imbriquées: une transaction racine (*top-level transaction*) peut engendrer une ou plusieurs autres transactions appelées sous-transactions, qui peuvent engendrer à leur tour d'autres transactions. Les modifications issues des sous-transactions ne sont réellement prises en compte que lorsque la transaction racine est validée. Si la transaction racine est annulée, les modifications issues des sous-transactions le sont aussi.

Par exemple, considérons un événement primaire e_1 engendrant deux événements composés e_2 et e_3, chacun d'entre-eux étant associé à une règle particulière (r_1, r_2, et r_3). Si pendant l'exécution de l'action de la règle r_1, un événement e_4 déclenche la règle r_4, alors r_4 sera exécutée en tant que sous-transaction de r_1, i.e. avant r_2 et r_3. Autrement dit, bien que la séquence d'événements détectés soit $<e_1, e_2, e_3, e_4>$, la séquence correspondante d'exécution des règles est $<r_1, r_4, r_2, r_3>$:

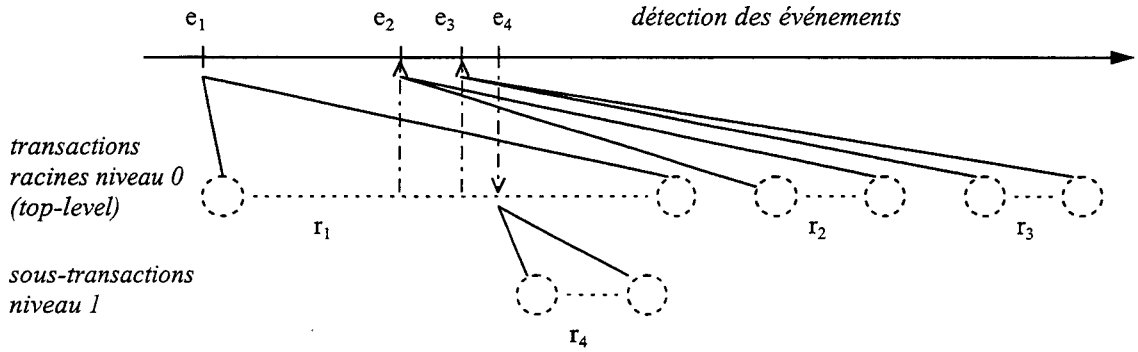


Figure 14: Détection des événements dans *SAMOS* — transactions imbriquées

II.2.2.6 Starburst

Dans *Starburst* [WID 91], le traitement des règles est automatiquement invoqué à la fin de chaque transaction générée par l'utilisateur, et qui a déclenché une ou plusieurs règles. *Starburst* considère comme *Ariel* les effets de bord des ensembles de modifications, mais l'algorithme utilisé est une variante:

évaluation initiales (des règles à tirer, basée sur une modification initiale)
répéter jusqu'à *aucune règle trouvée*
  résolution des conflits (sélectionner une règle)
  évaluer la condition (de la règle sélectionnée)
  exécuter l'action (de la règle sélectionnée)
  réévaluation des règles supplémentaires à tirer (basée sur les nouvelles modifications)
fin

Une différence importante est que la détermination des règles à tirer n'élimine pas les règles dont la condition est fausse (la condition est testée après le choix des règles). Les règles étant associées à des priorités relatives, la résolution des conflits est déterministe si l'ordre des priorités est total, ou complètement arbitraire si les priorités ne sont pas ordonnées.

II.3 Transformation d'un DBMS en ADBMS

Bien que les ADBMS décrits précédemment s'appuient sur des modèles classiques de bases de données (modèles relationnel ou orienté objet), il serait trop rapide de n'y voir qu'une simple adaptation des fonctionnalités et de l'architecture des DBMS existants.

Rappelons ici que les nouvelles fonctionnalités requises par un DBMS pour que celui-ci puisse être qualifié d'« actif » sont [DIT 95]:

- *Un ADBMS doit être avant tout un DBMS.* Tous les concepts requis par un DBMS sont aussi nécessaires pour un ADBMS (facilité de modélisation des données, langage d'interrogation, simultanéité des accès, etc.). Autrement dit, si un utilisateur ignore les fonctionnalités « actives », un ADBMS effectue le même travail qu'un DBMS « passif ».

- *Un ADBMS doit permettre la définition et la gestion des règles-ECA.* Les événements, les conditions, et les actions doivent être définissables par le biais d'un interface quelconque, et l'ADBMS doit savoir gérer le catalogue des règles ainsi définies.
- *Un ADBMS doit être capable de détecter l'occurrence d'un événement.* Ceci est valable pour tous les types d'événements.
- *Un ADBMS doit être capable d'évaluer les conditions et d'exécuter les actions.* Ceci signifie tout simplement qu'un ADBMS doit être capable d'implémenter l'exécution des règles-ECA.
- Optionnellement (mais fortement souhaitable), un ADBMS doit être pourvu d'outils d'aide au développement (par exemple un outil de mise au point capable de tracer le déclenchement des règles).

Les trois approches architecturales utilisables pour construire un ADBMS peuvent alors être choisies parmi:

- Concevoir un ADBMS en repartant de zéro, i.e. tous les composants passifs du système sont également reconstruits. Cette solution est évidemment la plus coûteuse.
- Utiliser un DBMS passif existant, le modifier et étendre ses possibilités de manière interne. Ces changements internes nécessitent la disponibilité du code source. Le prix à payer pour ce degré de liberté est une profonde connaissance de l'architecture et de la conception du DBMS passif, mais cette approche est toutefois envisageable avec l'aide des concepteurs du DBMS passif. Certains DBMS peuvent être modifiés à volonté (*custom*), pour intégrer de nouvelles fonctions. Pour d'autres, il est possible d'utiliser les *toolkits-DBMS*, qui fournissent un certain nombre de fonctionnalités fondamentales d'un DBMS sous forme de bibliothèques: à charge du concepteur d'implémenter le reste.
- Utiliser un DBMS passif, et lui apporter un comportement actif à l'aide d'une couche logicielle supplémentaire. A la différence des autres approches, le DBMS est juste considéré comme une boîte noire et ne peut pas être modifié de manière interne. Cette option, connue sous l'expression *layered-approach*, est vraisemblablement la plus facile à appréhender, car elle offre une totale liberté au concepteur pour construire le noyau de l'ADBMS.

Le Tableau 8 résume l'approche utilisée par les différents systèmes étudiés précédemment.

Systèmes		Architecture
Ariel	(relationnel)	sur-couche logicielle (layered)
HiPAC	(orienté objet)	entièrement ré-écrit (C, Smalltalk, LISP)
Ode	(orienté objet)	modifications internes
POSTGRES	(relationnel)	ajout de nouvelles fonctions (custom)
SAMOS	(orienté objet)	sur-couche logicielle (layered)
Starburst	(relationnel)	ajout de nouvelles fonctions (custom)

Tableau 8: Approches architecturales.

II.4 Conclusion

L'objectif des ADBMS est essentiellement d'apporter la réactivité aux DBMS passifs (relationnel et orienté objet), en utilisant le concept des règles-ECA dérivées des règles de production. Les systèmes décrits dans ce chapitre sont à notre sens les plus représentatifs de cette classe de systèmes de gestion de base de données, mais le lecteur pourra également trouver un complément d'information dans la liste non exhaustive suivante: *ACOOD* [BER 92], *NAOS* [COL 94], *RDL* [SIM 92], *REACH* [BUC 95], *Sentinel* [CHA 94], et *TRIGS* [KAP 94]. Si le point commun de ces systèmes est la prise en compte des événements et la conservation de l'intégrité des données, ils possèdent néanmoins leurs particularités:

- *Ariel* transforme les événements physiques en événements logiques, ce qui est plus intuitif, et recommande l'utilisation des règles *pattern-based* réalisant l'abstraction de l'origine de l'événement qui a déclenché une règle. Ceci est certes restrictif, mais rend certainement ce système le plus simple à appréhender, puisque son fonctionnement reste très séquentiel.
- *POSTGRES* met en avant le synchronisme dans l'exécution des règles, et son fonctionnement n'est pas sans rappeler la récursivité.
- *HiPAC*, au contraire, met l'accent sur le parallélisme total de l'exécution des règles, bien que les modes de couplage permettent d'influencer l'ordonnancement des actions.
- *Ode* se concentre plus sur les différents cas de figure relatifs au respect des contraintes d'intégrité: contraintes lâches (cas général), contraintes fortes (objets mutuellement dépendants), et triggers (pas d'intégrité à respecter).
- *SAMOS* préfère privilégier un temps de réponse réduit à l'arrivée d'un événement extérieur, à l'aide de transactions imbriquées.

Si les fonctionnalités réactives des ADBMS constitue un apport indéniable, il ressort néanmoins que ces systèmes manquent encore d'un outil de modélisation du comportement dynamique d'un système: l'emploi des règles-ECA est certes un atout, mais les concepteurs d'application sont toujours dépendants d'une programmation *impérative* pour la description des conditions et des actions. Nous présentons dans le chapitre III un nouvel outil de modélisation — les réseaux formels — dont la compatibilité avec les concepts des ADBMS sera ensuite exposée dans le chapitre IV.

CHAPITRE III

RESEAUX FORMELS

Relativement au nombre de travaux sur la modélisation des données, nous trouvons assez peu d'études dans la littérature sur les modèles formels de traitements. Mis à part les méthodes spécifiquement orientées objet, la plupart des approches modernes de modélisation du comportement de la dynamique des systèmes d'information utilisent, au moins partiellement, le formalisme des *réseaux de Petri* (RdP). Par exemple, C. Sibertin-Blanc [SIB 90] propose un modèle des systèmes basé sur les objets. Dans ce cadre, un *système* est un ensemble d'objets coopératifs, dont le *comportement dynamique*, ainsi que leurs interactions mutuelles sont modélisées formellement par des RdP à objets. D'autres aspects de l'utilisation des RdP dans les systèmes d'information peuvent être trouvés dans [HAR 93][CAD 97].

Néanmoins, même les RdP de haut niveau habituels — le lecteur pourra trouver une synthèse des RdP et des RdP de haut niveau dans [MUR 89], [JEN 92] ou [LEF 98] — ne disposent pas de tous les concepts utiles pour une modélisation naturelle des systèmes d'information. Notamment, la manipulation d'ensembles comme paramètres et une forme de négation existentielle semblent indispensables. Les réseaux formels [ALL 98] enrichissent le formalisme des RdP de haut niveau en introduisant de nouveaux types d'arcs afin de traduire l'effet d'actions élémentaires, comme l'insertion, la consultation, ou la suppression, ainsi que des contraintes sur les places, les marquages et les transitions.

Une contrainte [JAF 94] est une formule entre les variables d'un problème qui est dite satisfiable s'il existe une instantiation des variables qui la rende vraie. Les contraintes permettent d'exprimer toute sortes de relations sans présager d'une implémentation particulière. La différence entre une expression dans un langage impératif et une expression sous forme contrainte est mise en évidence par le traitement de l'égalité. Les langages impératifs utilisent généralement deux opérateurs distincts pour l'affectation et l'égalité (:= et = en pascal par exemple). Dans une expression de contrainte, l'opérateur d'affectation n'est pas nécessaire, on n'utilise que l'opérateur d'égalité, et c'est la résolution des contraintes qui « affecte » les valeurs des variables qui rendent les relations d'égalité vraies. En particulier, l'expression x := x + 1 d'un langage pascal par exemple est à distinguer avec la contrainte x = x + 1 qui est toujours fausse. Une expression dans les langages impératifs est orientée, c'est-à-dire qu'on fait une différence entre les arguments en entrée et les résultats en sortie. En revanche, cette distinction n'existe pas dans une

contrainte. Par exemple, la contrainte de conversion de degrés Celsius en Fahrenheit (F = 9/5 C + 32) permet de calculer aussi bien F en fonction de C que C en fonction de F. L'ordre d'apparition des instructions dans une expression de contraintes n'a pas d'importance contrairement au cas d'une expression dans un langage impératif. Par exemple, l'expression impérative Y := 2; X := Y; Y := 2 + 1; n'est pas équivalente à Y := 2; Y := 2 + 1, X := Y.

Notre utilisation des contraintes dans les réseaux formels est celle correspondant à l'utilisation du langage Z, introduit au Paragraphe III.2. Il s'agit d'une forme d'expression naturelle pour représenter un problème. En effet, les contraintes permettent des représentations implicites des ensembles. Par exemple E = { x : salaire(x) > 35000 et poste(x) = chef } est l'ensemble de tous les x chef ayant un salaire plus grand que 35000. Les contraintes peuvent ainsi être structurelles, c'est-à-dire décrire la structure des objets, ou fonctionnelles, c'est-à-dire décrire le comportement du système.

Ces deux aspects se retrouvent dans notre formalisme de réseau formel respectivement sur les places et sur les transitions. Quant aux contraintes de marquage, elles s'appliquent aux contenus des places et peuvent être vues comme la caractérisation d'une structure ou d'un comportement selon le point de vue de l'utilisateur. L'abstraction du modèle est garantie par une définition purement formelle du comportement du réseau en utilisant une sémantique basée sur le langage Z. Soulignons néanmoins que la syntaxe Z n'apparaît à aucun moment sur le réseau et que les annotations se limitent à des ensembles de contraintes élémentaires.

Après avoir défini précisément la sémantique des réseaux formels, nous proposerons quelques exemples de modélisations utilisant leurs concepts.

III.1 Eléments de théorie des réseaux

Nous présentons ici brièvement les bases élémentaires de la théorie des réseaux de Petri nécessaires à la compréhension de la suite du texte.

III.1.1 Réseaux élémentaires

Un réseau est un graphe biparti (c'est-à-dire avec deux types de noeuds) orienté:

Définition 1 (*structure d'un réseau*)

> Un *réseau R* est un triplet (P, T, A) tel que:
>
> (i) P et T sont des ensembles finis non vides et disjoints
>
> (ii) $A \subseteq (P \times T) \cup (T \times P)$

Dans le vocabulaire usuel des réseaux de Petri, P est un ensemble de *places*, T un ensemble de *transitions* et A un ensemble d'*arcs*. Un arc peut relier une place à une transition (arc consommation) ou une transition à une place (arc production). Dans le formalisme initial défini par Petri, on parle de conditions et d'événements plutôt que de places et transitions. Graphiquement, on représente en général les places par des cercles, et les transitions par des barres (ou des rectangles).

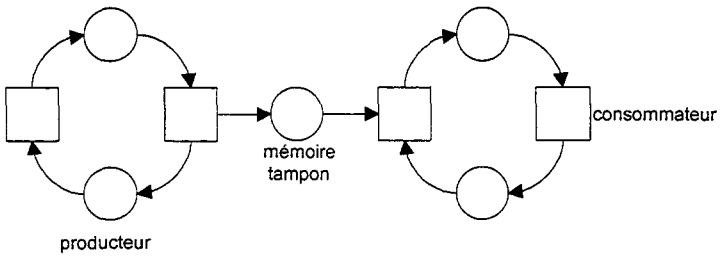


Figure 15: Un réseau simple

Définition 2 (*entrées/sorties*)

Soit un réseau $R = (P, T, A)$. Pour $x \in P \cup T$, on note:
$^\circ x = \{y \mid (y,x) \in A\}$, l'ensemble des *prédécesseurs* de x (ou *entrées* de x)
$x^\circ = \{y \mid (x,y) \in A\}$, l'ensemble des *successeurs* de x (ou *sorties* de x)

x peut être soit une place, soit une transition. Un réseau est un *graphe d'états* si chaque *transition* a exactement une place d'entrée et une place de sortie. On parle de *graphe d'événements* si chaque *place* a exactement une transition d'entrée et une transition de sortie et si le réseau est *marqué*:

Définition 3 (*marquages*)

Soit un réseau $R = (P, T, A)$. On appelle *marquage* une application $M : P \rightarrow \mathbb{N} \cup \{\omega\}$ qui à chaque place de P associe le nombre de *jetons* présents dans cette place (ω dénote un marquage *infini*).

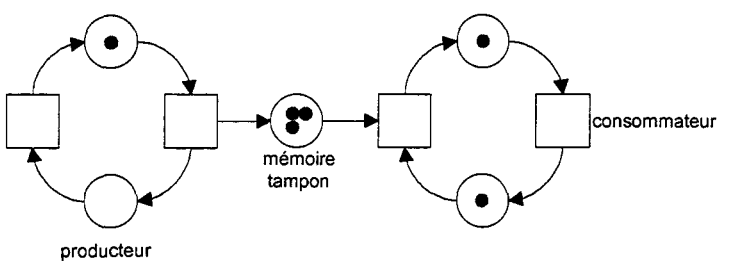


Figure 16: Un réseau de Petri

Dans le formalisme des réseaux condition/événement, le marquage de chaque place contient au plus un jeton. Dans le cas général, on autorise des marquages quelconques, et même les marquages infinis, pour des raisons techniques. Par abus de langage, on appelle réseau de Petri tout réseau marqué (et pas seulement les réseaux condition/événement):

Définition 4 (*réseau de Petri*)

Un tuple (P, T, A, M_0) est un *réseau de Petri* si:

(i) (P, T, A) est un réseau

(ii) $M_0 : P \rightarrow \mathbb{N} \cup \{\omega\}$, est un *marquage initial* du réseau

Dans la suite, on abrégera souvent « Réseau de Petri » par *RdP*.

Le *franchissement* (ou *tir*) d'une transition permet de passer d'un marquage à un autre par une règle très simple: une transition est *validée* si toutes ses places en entrée ont au moins un jeton, et le marquage résultant est obtenu en retirant un jeton de chacune des places d'entrée, et en ajoutant un jeton à chacune des places de sortie.

Définition 5 (*comportement d'un RdP*)

Soit (P, T, A, M_0) un réseau de Petri:

(i) une transition $t \in T$ est dite *validée* pour le marquage M ssi $\forall p \in {}^\circ t, M(p) \geq 1$

(ii) le marquage M' résultant du *tir* de la transition t validée pour M est tel que $\forall p \in P$:

$$M'(p) = \begin{cases} M(p) - 1 & \text{si } p \in {}^\circ t \text{ et } p \notin t^\circ \\ M(p) + 1 & \text{si } p \in t^\circ \text{ et } p \notin {}^\circ t \\ M(p) & \text{sinon} \end{cases}$$

(iii) on note alors $M[t>M'$

La Figure 17 illustre le tir de la transition T3. On voit qu'un jeton a été retiré de chaque place en entrée, et qu'un jeton a été ajouté dans la place de sortie. Le comportement de ce réseau schématise le comportement de deux processus (un producteur et un consommateur), reliés par une mémoire tampon où les jetons peuvent s'accumuler en attendant d'être consommés.

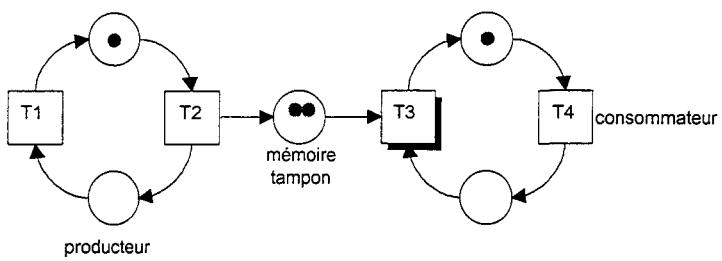


Figure 17: Tir d'une transition

Dans le cas où une place est à la fois en entrée et en sortie de la transition, le marquage ne change pas, puisque le jeton est retiré, puis ajouté. La notion de temps n'intervient pas explicitement dans cette définition. Néanmoins, si on considère un réseau de Petri comme un modèle du comportement dynamique d'un système, les notions de séquentialité et de parallélisme par exemple induisent une dimension temporelle. Dans un réseau de Petri ordinaire, le tir d'une transition est *instantané*, mais rien n'oblige à tirer une transition dès qu'elle est validée. D'autre part, rien n'interdit de tirer *simultanément* plusieurs transitions validées, sauf dans le cas où une transition ne serait plus validée après le tir d'une autre transition validée: on parle alors de *conflit*. Un conflit ne peut se produire que si une place a plus d'une transition en sortie. Un RdP est donc dit *sans conflit* si toute place a au plus une transition de sortie. On parle de RdP à *choix libre* si pour toute place p ayant plus d'une transition en sortie, chacune de ces transitions admet uniquement la place p en entrée.

Dans la définition que nous avons donnée d'un réseau, il ne peut y avoir plus d'un arc entre une place et une transition, ou entre une transition et une place. Les réseaux

place/transition autorisent des arcs multiples. Pour ne pas alourdir la représentation graphique, on conserve un arc unique entre deux noeuds, qu'on annote par un *poids*, représentant le nombre d'arcs entre ces noeuds:

Définition 2 (*RdP place/transition*)

Un tuple (P, T, A, M_0, W) est un réseau place/transition si:

(i) (P, T, A, M_0) est un réseau de Petri

(ii) $W : A \rightarrow \mathbb{N} - \{0\}$ est une fonction qui à chaque arc associe une *pondération* non nulle.

On adapte facilement la règle de tir au comportement des réseaux place/transition:

Définition 7 (*comportement d'un RdP place/transition*)

Soit (P, T, A, M_0, W) un réseau place/transition:

(i) une transition $t \in T$ est dite *validée* pour le marquage M ssi $\forall p \in {}^\circ t, M(p) \geq W(p,t)$

(ii) le marquage M' résultant du *tir* de la transition t validée pour M est tel que $\forall p \in P$:

$$M'(p) = \begin{cases} M(p) - W(p,t) & \text{si } p \in {}^\circ t \text{ et } p \notin t^\circ \\ M(p) + W(t,p) & \text{si } p \in t^\circ \text{ et } p \notin {}^\circ t \\ M(p) - W(p,t) + W(t,p) & \text{si } p \in t^\circ \text{ et } p \in {}^\circ t \\ M(p) & \text{sinon} \end{cases}$$

On notera que les RdP place/transition sont aussi appelés RdP généralisés et de plus que tout RdP généralisé peut être transformé en RdP ordinaire [DAV 92].

Certains auteurs [DAV 92] intègrent dans les RdP place/transition la notion de *capacité*: le tir d'une transition ne doit pas conduire au marquage d'une place qui excéderait sa capacité. Le principe en est simple: il s'agit d'un RdP dans lequel des capacités (nombres entiers strictement positifs) sont associés aux places. En fait, il est toujours possible de se ramener au cas précédent par une transformation simple du réseau (en rajoutant en parallèle à chaque place une nouvelle place avec un nombre de jetons égal à la capacité de la première place [REI 85]).

L'ajout d'*arcs inhibiteurs* apporte aux RdP la puissance d'expression d'une machine de Turing: la présence d'un arc inhibiteur entre une place et une transition bloque le tir de celle-ci dans le cas où la place n'est pas vide [DAV 92].

III.1.2 Réseaux de haut niveau

III.1.2.1 Langage du premier ordre

La modélisation de systèmes réels par des RdP conduit souvent à une explosion combinatoire de la taille du réseau. De plus, il semble naturel de travailler avec des jetons que l'on puisse distinguer (jetons individuels), voire structurer et transformer. Dans un cadre général, on a besoin pour cela d'un *langage* autorisant les notions de *types*, de *variables*, et d'*expressions bien formées* pouvant être *instanciées* et *évaluées*.

Définition 8 (*langage fonctionnel typé de premier ordre*)

On définit un *langage typé* $L = (\Sigma, V, L_\Sigma, L_{\Sigma,V}, Var, Type, \Xi, Eval)$ tel que:

(i) Σ est un ensemble fini de *types*, c'est-à-dire d'ensembles finis non vides. On note $<\Sigma>$ l'ensemble des éléments des types de Σ

(ii) V est un ensemble fini de *variables*, disjoint de $<\Sigma>$

(iii) L_Σ est un ensemble d'*expressions closes bien formées*, tel que $<\Sigma> \subseteq L_\Sigma$

(iv) $L_{\Sigma,V}$ est un ensemble d'*expressions bien formées*, tel que $L_\Sigma \subseteq L_{\Sigma,V}$ et $V \subseteq L_{\Sigma,V}$

(v) $Var : L_{\Sigma,V} \rightarrow V$ est une fonction qui associe à chaque expression bien formée l'ensemble de ses variables

(vi) $Type : L_{\Sigma,V} \rightarrow \Sigma$ est une fonction qui associe un type à chaque expression bien formée

(vii) Ξ est un ensemble de fonctions d'*instanciation* (ou *liens*, ou *substitutions closes*), tel que $\forall\theta \in \Xi$ et $\forall e \in L_{\Sigma,V}$ l'expression close $\theta e \in L_\Sigma$ est l'expression obtenue à partir de e par substitution où *chaque* variable $v \in Var(e)$ a été *substituée* par un élément de $Type(v)$

(viii) $Eval : L_\Sigma \rightarrow <\Sigma>$ est une fonction qui à chaque expression close associe une *valeur* du type de l'expression : $Eval(e) \in Type(e)$

Cette définition d'un langage typé donne les outils suffisants pour définir des *RdP de haut niveau* manipulant des jetons structurés, sans qu'il soit nécessaire d'expliciter le mécanisme de construction des expressions, ou la méthode d'évaluation. Comme exemple de tels langages, on peut citer la logique du premier ordre (RdP prédicat/transition [GEN 91]), les types abstraits algébriques (RdP algébriques [REI 91]), l'algèbre des relations (RdP relationnels [REI 85]), les langages fonctionnels comme ML (RdP colorés [JEN 92]) ou les langages orientés objet (RdP à objets [SIB91a]).

III.1.2.2 RdP de haut niveau

La définition formelle des réseaux de Petri de haut niveau associe un réseau annoté par les expression d'un langage du premier ordre avec un marquage initial, vu comme une fonction d'initialisation du réseau:

Définition 9 (*RdP de haut niveau*)

Un tuple (R, L, C, G, W, I) est un réseau de haut niveau si:

(i) $R = (P, T, A)$ est un réseau

(ii) $L = (\Sigma, V, L_{\Sigma}, L_{\Sigma,V}, Var, Type, \Xi, Eval)$ est un langage typé, tel que le type booléen $\{Vrai, Faux\}$ soit dans Σ

(iii) $C : P \rightarrow \Sigma$ est une fonction qui associe à chaque *place* du réseau un type, appelé *couleur* de la place.

(iv) Le langage L est tel que $\forall p \in P$, l'ensemble des multi-ensembles $Bags(C(p))$ construits sur la base de $C(p)$ est un type de Σ

(v) $W : A \rightarrow L_{\Sigma,V}$ est une fonction qui à chaque *arc* associe une expression de type multi-ensemble:

$$\forall (p,t) \in A,\ Type(W(p,t)) = Bags(C(p))$$

$$\forall (t,p) \in A,\ Type(W(t,p)) = Bags(C(p))$$

(vi) $G : T \rightarrow L_{\Sigma,V}$ associe à chaque *transition* une expression *booléenne*, appelée *garde* (ou *sélecteur*):

$$\forall t \in T,\ Type(G(t)) = \{Vrai, Faux\}$$

(vii) $I : P \rightarrow L_{\Sigma}$ est une fonction d'*initialisation*, qui à chaque *place* associe une expression close de type multi-ensemble:

$$\forall p \in P,\ Type(I(p)) = Bags(C(p))$$

La définition originale de Jensen [JEN 92] autorise plusieurs arcs entre deux noeuds, mais on peut toujours se ramener à la définition précédente par somme symbolique des expressions sur les arcs (qui sont des multi-ensembles). D'autre part, chaque variable libre apparaissant dans la garde d'une transition doit normalement apparaître sur un des arcs arrivant sur cette transition. On peut aussi toujours « déplier » un RdP de haut niveau en un RdP place/transition équivalent (puisque les couleurs sont des ensembles finis [JEN 92]). Evidemment, une telle transformation n'est en général pas envisageable en pratique, mais permet d'utiliser certains résultats théoriques obtenus pour les RdP ordinaires.

Définition 10 (*comportement d'un RdP de haut niveau*)

Soit (R, L, C, G, W, I) un réseau de haut niveau:

(i) un marquage M est une application qui à chaque place du réseau associe un multi-ensemble:

$$\forall p \in P, M(p) \in Bags(C(p))$$

En particulier, le marquage initial M_0 est donné par l'évaluation de la fonction d'évaluation

$$\forall p \in P, M_0(p) = Eval(I(p))$$

(ii) une transition $t \in T$ est dite *validée* pour le marquage M et la substitution θ ssi:

$$\forall p \in {}^\circ t, M(p) \geq Eval(\theta W(p,t))$$

(iii) le marquage M' résultant du *tir* de la transition t validée pour M et θ est tel que $\forall p \in P$:

$$M'(p) = \begin{cases} M(p) - Eval(\theta W(p,t)) & \text{si } p \in {}^\circ t \text{ et } p \notin t^\circ \\ M(p) + Eval(\theta W(t,p)) & \text{si } p \in t^\circ \text{ et } p \notin {}^\circ t \\ M(p) - Eval(\theta W(p,t)) + Eval(\theta W(t,p)) & \text{si } p \in t^\circ \text{ et } p \in {}^\circ t \\ M(p) & \text{sinon} \end{cases}$$

Un RdP, même coloré, devient rapidement difficile à dessiner sur une seule « page » pour la modélisation d'un système complexe. De plus, la compréhension d'un réseau de grande taille peut être problématique. Les *RdP colorés hiérarchiques* [JEN 92] permettent une structuration des réseaux, à l'aide de deux mécanismes de base: la substitution de transitions et la fusion des places. La *substitution des transitions* permet de représenter un sous-réseau par une transition, et de renvoyer à la « page » où est représenté celui-ci, un peu à la manière d'un module en programmation structurée. Une même place peut avoir plusieurs représentations dans le réseau, par le principe de *fusion des places*: lorsque des jetons sont retirés ou ajoutés dans une des instances de la place, ils le sont de même dans toutes les autres instances. On peut toujours transformer un RdP coloré hiérarchique en un RdP non hiérarchique. Pour plus de détails sur les réseaux colorés, on pourra se reporter à [JEN 92].

III.2 Rappels sur le langage Z

Le langage formel Z [ABR 84][LIG 91][SPI 92], basé sur la théorie des ensembles et le calcul des prédicats a été conçu pour la spécification formelle et n'est pas exécutable. Nous nous intéressons ici principalement à sa qualité d'abstraction.

Le *schéma* est la structure fondamentale de Z. Syntaxiquement un schéma se présente sous la forme d'un cadre ouvert à droite et divisé en deux parties horizontalement. On définit dans la première partie les composantes à l'aide des types (types élémentaires, types de base ou types construits). La deuxième partie contient les contraintes logiques sur les composantes définies précédemment. Le nom du schéma apparaît sur la ligne supérieure du cadre:

Période

$début, fin : \mathbb{N}$

$début \leq fin$

Figure 18: Exemple de schéma Z

Ce schéma définit en fait un ensemble en compréhension, comme nous le voyons ci-dessous:

$$Période = \{<début, fin> \in \mathbb{N} \times \mathbb{N} \mid début \leq fin\}$$

Une spécification en Z comporte la définition de l'ensemble des états susceptibles d'être pris en compte par un système en donnant les contraintes qui s'appliquent aux composantes des schémas. L'homogénéité des concepts utilisés dans le langage Z est une qualité remarquable. La notion d'ensemble est le moyen d'expression universel: l'espace des états du système est un ensemble, les types sont des ensembles et même les opérations sont des ensembles, c'est-à-dire des sous-ensembles du produit cartésien de l'ensemble des états. Ainsi il est possible d'effectuer sur les schémas des calculs comme pour les ensembles, ce qui offre une solution élégante pour la décomposition d'un problème.

Pour décrire une opération, Z utilise une notation décorée pour distinguer l'*état après* l'opération de l'*état avant*. La mise en relation de deux états successifs utilise la composition des schémas. Par exemple, si *Période* est l'état avant, *Période'* décrit l'état après, et la contrainte figurant dans la deuxième partie du schéma décrit la relation entre les deux états. On remarquera l'expression $\Delta Période$ dans la première partie du schéma *Décaler10* qui est une notation simplifiée pour déclarer *Période* et *Période'*. Cette opération consiste à décaler la période de 10.

Décaler10

$\Delta Période$

$Période'.début = Période.début + 10$

$Période'.fin = Période.fin + 10$

Figure 19: Exemple d'expression d'une opération en Z avec changement d'état

Les *schémas* Z sont donc capables d'exprimer des changements d'états et peuvent rendre compte du comportement dynamique d'un système. Néanmoins, il n'existe pas de mécanisme spécifique de gestion du contrôle des événements.

III.3 Fondements des réseaux formels

Nous reprenons ici les définitions des réseaux formels [CAD 97] utilisant le formalisme Z. On pourrait également donner une définition purement indépendante de Z, basée sur la théorie des ensembles [GAB 99]. Néanmoins, une telle présentation nécessite un matériel théorique important, et nous ne souhaitons pas alourdir inutilement le texte. Notre principal objectif est en effet l'application aux SGBD actifs que nous développerons au chapitre suivant.

III.3.1 Définition des réseaux formels

Un *réseau formel* est construit graphiquement en reliant des places et des transitions, comme dans un réseau de Petri habituel. La puissance d'expression du modèle réside dans les annotations et les décorations de ces derniers éléments. Le comportement attendu d'un réseau formel, s'il demeure intuitif pour l'utilisateur, devient difficile à définir rigoureusement avec les méthodes habituelles des RdP.

Après avoir examiné plusieurs formalismes, le langage Z, qui permet la spécification statique et dynamique des systèmes à l'aide d'un vocabulaire mathématique standard, nous a semblé être le plus adapté à notre problème. En effet, nous avons besoin d'opérateurs ensemblistes puissants, de variables logiques, et de changements d'états et ces trois points sont rarement traités ensemble dans le même formalisme. Chaque annotation du réseau étant associée à un schéma Z, le réseau complet est alors associé à une spécification Z. Nous donnons dans ce chapitre les principaux modèles de schémas Z correspondant aux annotations de places, d'arcs, et de transitions. La signification des éléments de Z utilisés sera brièvement expliquée quand elle n'est pas évidente.

***Définition** 1 structure d'un réseau formel*

Un tuple $(P, T, A, N, W, D, S_P, C_T, C_M, M_0)$ est un *réseau formel* si:

(i) P, T, et A sont des ensembles finis et disjoints dénotant respectivement les places, les transitions et les arcs du réseau.

(ii) $N : A \rightarrow (P \times T) \cup (T \times P)$ est une fonction qui à chaque arc associe son noeud d'*entrée* et son noeud de *sortie*.

(iii) $W : A \rightarrow \mathbb{N}$ est une fonction qui à chaque *arc* associe une *pondération*.

(iv) $D : A \rightarrow \{$ *consommation*, *information*, *information négative*, *production* $\}$ est une fonction qui à chaque arc associe une *décoration* (un type d'arc).

(v) Entre deux noeuds, il n'y a pas plus d'un arc de chaque type.

(vi) $S_P : P \rightarrow Z_{\Sigma,V}$ est une fonction qui associe à chaque *place* du réseau un *schéma* Z.

(vii) $C_T : T \rightarrow L_{\Sigma,V}$ associe à chaque *transition* une formule d'un langage de premier ordre L, appelée *contrainte de transition*.

(viii) C_M est une formule de L, appelée *contrainte de marquage*.

(ix) M_0 est une fonction d'*initialisation*, qui à chaque place p associe un *ensemble de liaisons* noté $M.\overline{p}$ cohérent avec le schéma de la place p et avec la contrainte de marquage C_M.

Remarques

(i),(ii) On revient ici au point de vue de Jensen [JEN 92], qui autorise plusieurs arcs entre deux noeuds. En fait, l'existence de plusieurs types d'arcs oblige cette convention: il n'est pas possible de contracter les arcs en un seul par sommation des expressions sur les arcs, comme dans les RdP colorés.

(iii) La pondération d'un arc dénote le nombre de variables qui seront générées pour cet arc, d'une manière qui sera décrite plus loin.

(iv) La décoration des arcs permet une grande souplesse dans la définition du comportement des réseaux formels.

(v) Il est nécessaire d'interdire la présence de plusieurs arcs du même type entre deux noeuds, afin d'éviter des confusions au niveau du nommage automatique des variables locales aux arcs.

(vi) Le schéma Z revient d'une certaine manière à définir le *type* des liaisons admises par la place. Néanmoins la définition d'un schéma est plus fine que la déclaration d'une couleur associée à une place, notamment grâce aux *contraintes* qui peuvent annoter le schéma. D'autre part, la notion de *liaison* est dynamique, et permet par exemple l'usage de valeurs incomplètement connues. Les variables locales à un schéma de place peuvent être vues comme des *attributs* de la place. La variable x de la place p est notée $p.x$ à l'extérieur de la place. Les mêmes noms de variables peuvent donc être utilisés dans des places différentes sans risque de confusion.

(vii) Les différentes annotations sur les arcs, et les contraintes de transition permettront de générer automatiquement les schémas Z associés, qui définiront le comportement du réseau.

(viii) La contrainte de marquage est une contrainte globale, qui s'applique à tous les marquages possibles du réseau. Ce type de contrainte permet par exemple d'exprimer des interactions entre différentes places.

(ix) Le marquage initial associe des liaisons aux variables déclarées dans les schémas de place. Un marquage n'est donc pas nécessairement clos.

La définition précédente de la structure générale d'un réseau formel peut être résumée sous la forme d'un schéma:

Réseau formel

$$P : \mathbb{F} Places \ ; \ T : \mathbb{F} \ Transitions \ ; \ A : \mathbb{F} \ Arcs \ ;$$
$$N : A \rightarrow (P \leftrightarrow T) \cup (T \leftrightarrow P)$$
$$W : A \rightarrow \mathbb{N} \cup \{*\}$$
$$D : A \rightarrow \{ \circ\!\!\rightarrowtail , \circ\!\!- , \circ\!\!\rightarrowtail\!\!\!\circ , \rightarrowtail\!\!\circ \}$$

$$\forall a : A \bullet D(a) \in \{ \rightarrowtail\!\!\circ \} \Rightarrow N(a) \in (T \leftrightarrow P)$$
$$\forall a : A \bullet D(a) \in \{ \circ\!\!\rightarrowtail , \circ\!\!- , \circ\!\!\rightarrowtail\!\!\!\circ \} \Rightarrow N(a) \in (P \leftrightarrow T)$$
$$\forall a_1 : A, a_2 : A \bullet N(a_1) = N(a_2) \Rightarrow D(a_1) \neq D(a_2)$$

pour lequel N, W, et D qui associent à chaque arc une entrée/sortie, un poids et un genre sont des fonctions totales, puisqu'elles doivent être définies pour chaque arc. On convient que les arcs consommation, information, et information négative ont pour entrée une place et pour sortie une transition, et que les arcs production ont pour entrée une

transition et pour sortie une place. La dernière contrainte du schéma spécifie que deux arcs différents entre deux mêmes noeuds du réseau ne doivent pas avoir le même genre.

Afin d'illustrer les définitions sur le comportement d'un réseau formel, nous prendrons comme exemple la modélisation de la société commerciale *GENCOD*, dont nous rappelons les grandes lignes (le lecteur pourra se reporter au Paragraphe III.4.3 dans lequel l'application complète est développée):

i. Un *client* passe une commande pour certains *articles* correspondant chacun à un *poste* (une ligne) de la commande.
ii. Si l'article est en *stock*, le poste correspondant de la commande est *facturé*.
iii. Si tous les postes de la commande sont facturés, une *facture* pour la commande est émise.

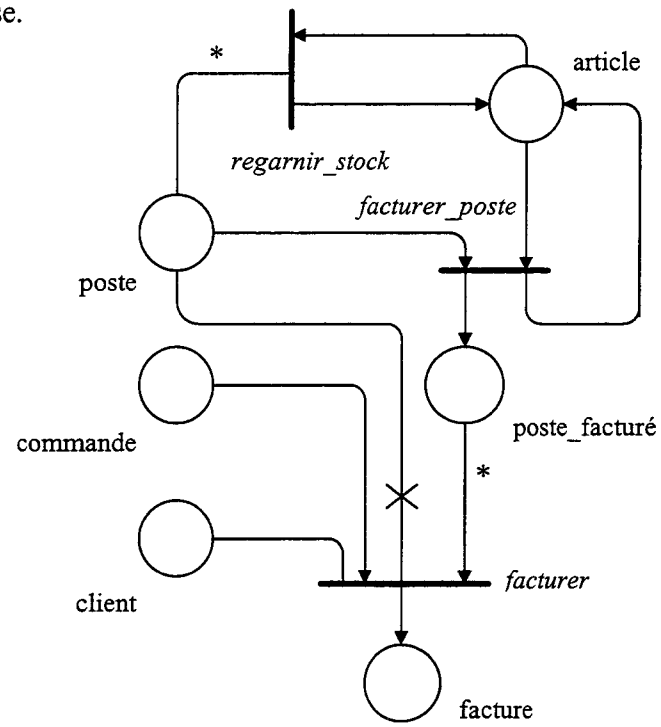


Figure 20: *GENCOD* — Réseau formel

La Figure 20 donne la représentation graphique du réseau formel associé à notre exemple. Nous allons détailler les différents éléments et annotations de ce réseau.

Ainsi, à chaque *place* correspond un schéma Z qui détermine les liaisons possibles:

article

$code\_article : \mathbb{N}$
$prix\_unitaire : \mathbb{R}$
$quantité\_en\_stock : \mathbb{N}$

$code\_article \notin \{ 120, 234, 879, 1053 \}$

On suppose ici par exemple que certains articles sont définitivement épuisés. Cette information supplémentaire est apportée par la contrainte de marquage.

A chaque place p correspond un marquage noté $\overline{p}$ qui représente un ensemble de liaisons (instances) compatibles avec le schéma de place (qui peut être interprété comme un type). Cet ensemble de liaisons peut être écrit en extension sous forme d'une table:

$M.\overline{article}$

code_article	*prix_unitaire*	*quantité_en_stock*
1	235,00	12
2	540,00	20
3	350,00	2
...	...	...

Le marquage de toutes les places correspond au schéma implicite (généré par la structure et le marquage). De façon générale, le schéma de marquage a la forme:

M

$p_1 \wedge ... \wedge p_n$

$\overline{p}_1 : \mathbb{P}\, p_1$

...

$\overline{p}_n : \mathbb{P}\, p_n$

...

Dans notre exemple:

M

$article \wedge ... \wedge facture$

$\overline{article}$: $\mathbb{P}$ article

...

$\overline{facture}$: $\mathbb{P}$ facture

...

où $p_1, ..., p_n$ sont les références de schémas pour les places de P (i.e. les défintitions et variables de ces schémas peuvent être utilisés dans M).

Une contrainte de marquage de la forme

$$\forall j_1, j_2 : p \mid j_1 \in \overline{p}, j_2 \in \overline{p} \bullet j_1.x \neq j_2.x$$

est très courante. On l'appelle *contrainte de clé*, et on convient de l'abréger en soulignant la déclaration de la variable x dans le schéma de place p. Par exemple, si l'on veut imposer que le code d'un article soit unique:

article

$\underline{code\_article} : \mathbb{N}$
$prix\_unitaire : \mathbb{R}$
$quantité\_en\_stock : \mathbb{N}$

$code\_article \notin \{ 120, 234, 879, 1053 \}$

On s'intéresse maintenant plus particulièrement au traitement correspondant à la facturation d'un poste de la commande. Ce traitement consiste à vérifier si l'article commandé est référencé au catalogue. Si la quantité disponible en stock est suffisante pour satisfaire la commande, on facture le poste correspondant et on enlève du stock le nombre d'articles commandés. Bien entendu, les références de l'article et du poste ne sont pas affectés par cette opération. La contrainte correspondante s'écrit formellement:

$$C_{facturer\_poste} = \begin{cases} poste^-.code\_article = article^-.code\_article \\ article^-.quantité\_stock \geq poste^-.quantité\_commandée \\ \\ poste\_facturé^+.numéro\_commande = poste^-.numéro\_commande \\ poste\_facturé^+.code\_article = poste^-.code\_article \\ poste\_facturé^+.montant\_TTC = poste^-.quantité\_commandée \times article^-.prix\_unitaire \\ \\ article^+.code\_article = article^-.code\_article \\ article^+.prix\_unitaire = article^-.prix\_unitaire \\ article^+.quantité\_stock = article^-.quantité\_stock - poste^-.quantité\_commandée \end{cases}$$

L'ordre dans lequel sont écrites les contraintes et les lignes blanches n'ont pas d'importance, si ce n'est pour la lisibilité de la spécification. D'autre part, remarquons qu'il n'y a aucune affectation, mais seulement des contraintes d'égalité.

A chaque arc consommation est associé un schéma d'arc implicite:

Arc[p, t, ⊶, n]

ΔM
$p^{1-}, \ldots p^{n-} : p$

$p^{1-} \in p ; \ldots ; p^{n-} \in p$
$\overline{p}\,' = \overline{p} \setminus \{ p^{1-}, \ldots, p^{n-} \}$

La notation ΔM exprime que le marquage M change d'état et devient M' après l'action; en particulier, $\overline{p}\,'$ représente le marquage $\overline{p}$ de la place p après le changement d'état. Les variables p^{1-}, ..., p^{n-} sont générées à partir du type d'arc (ici qui dénote une consommation), du poids de cet arc et du nom de la place reliée à l'arc. Elles représentent

des liaisons particulières dont le type est une référence au schéma p, et sont contraintes à appartenir chacune au marquage $\overline{p}$ (elles dénotent donc chacune une ligne dans la table du marquage de p). Ces variables doivent également respecter la contrainte de transition, comme nous le verrons plus loin. Le marquage $\overline{p}$ ' après le tir est le marquage $\overline{p}$ privé de l'ensemble des liaisons associées aux variables p^{1-}, ..., p^{n-} de façon comparable à la consommation de jetons dans un RdP.

On convient d'omettre l'indice de la variable générée dans le cas où n=1. Sinon l'arc est annoté par n sur le réseau. Dans notre exemple, le schéma généré par l'arc de *article* à *facturer_poste* est donc:

Arc[article, facturer_poste, ⟜, 1]

ΔM

$article^{-} : article$

$article^{-} \in \overline{article}$

$\overline{article}' = \overline{article} \setminus \{ article^{-} \}$

Attention cependant, ce schéma n'est pas écrit explicitement. Il est inféré à partir de la structure du réseau. L'utilisateur n'a donc jamais à écrire un schéma de ce genre.

On construit de manière analogue les arcs production:

Arc[p, t, ⟝, n]

ΔM

$p^{1+}, \ldots p^{n+} : p$

$p^{1+} \in \overline{p} \; ; \ldots ; p^{n+} \in \overline{p}$

$\overline{p}' = \overline{p} \cup \{ p^{1+}, \ldots, p^{n+} \}$

Dans notre exemple, il suffit encore de générer une seule variable:

Arc[facturer_poste, poste_facturé, ⟝, 1]

ΔM

$poste\_facturé^{+} : poste\_facturé$

$poste\_facturé^{+} \in \overline{poste\_facturé}$

$\overline{poste\_facturé}' = \overline{poste\_facturé} \cup \{ poste\_facturé^{+} \}$

La transition *facturer* introduit trois nouveaux types d'arcs: un arc information, un arc information négative, et un arc consommation étoile.

$$C_{facturer} = \begin{cases} \text{commande}^{-}.\text{numéro\_client} = \text{client}^{?}.\text{numéro\_client} \\ \\ \text{poste}^{-}.\text{numéro\_commande} = \text{commande}^{-}\text{numéro\_commande} \\ \text{poste\_facturé}^{*-}.\text{numéro\_commande} = \text{commande}^{-}.\text{numéro\_commande} \\ \\ \text{facture}^{+}.\text{numéro\_commande} = \text{commande}^{-}.\text{numéro\_commande} \\ \text{facture}^{+}.\text{numéro\_client} = \text{client}^{?}.\text{numéro\_client} \\ \text{facture}^{+}.\text{adresse\_client} = \text{client}^{?}.\text{adresse\_client} \\ \text{facture}^{+}.\text{nom\_client} = \text{client}^{?}.\text{nom\_client} \\ \text{facture}^{+}.\text{montant\_TTC} = \text{SOMME}(\text{poste\_facturé}^{*-}.\text{montant\_TTC}) \end{cases}$$

Contrairement aux arcs précédents, les arcs information ne modifient pas le marquage:

Arc[p, t, ⊶⊣, n]

$$\Xi M$$
$$p^{1?}, \ldots p^{n?} : p$$

$$p^{1?} \in \overline{p} \; ; \ldots ; p^{n?} \in \overline{p}$$

La notation ΞM exprime que le marquage M ne change pas d'état. C'est le cas rencontré pour l'arc entre *client* et *facturer*, dont on désire seulement extraire une information (son numéro par exemple).

Les arcs information négative imposent que les contraintes sur leurs variables *ne soient pas vérifiées* dans le marquage courant:

Arc[p, t, ⊶⊣, n]

$$\Xi M$$
$$p^{1\sim}, \ldots p^{n\sim} : p$$

$$p^{1\sim} \notin \overline{p} \; ; \ldots ; p^{n\sim} \notin \overline{p}$$

C'est le cas de l'arc qui vérifie qu'il ne reste plus aucun poste correspondant à la commande à traiter dans la place *poste*.

Un arc doté d'une pondération * convoie un *ensemble* de liaisons dont la cardinalité n'est pas connue à priori (elle peut même être nulle):

*Arc[p, t, ↦∞, *]*

ΔM

$p^{*-} : p$

$\overline{p}^{*-} \in \overline{p}$

$\overline{p}^{*-} = \{ l : p \mid l \in p \wedge l = \overline{p}^{*-} \bullet l \}$

$\overline{p}' = \overline{p} \setminus \overline{p}^{*-}$

Cette définition est un peu délicate à manipuler. La variable p^{*-} pourra être contrainte dans la transition. $\overline{p}^{*-}$ est un sous-ensemble du marquage de p, tel que chacune des liaisons vérifie la contrainte sur p^{*-}. Dans notre exemple, on souhaite consommer l'ensemble de tous les postes facturés dans la place *poste_facturé* dont le numéro de commande est égal au numéro de la commande traitée par la transition *facturer*. En règle générale, si $\overline{p}$ dénote un ensemble de liaisons du schéma p, on note $p.x$ l'ensemble correspondant des valeurs de la composante x:

$$\overline{p}.x = \{ l : p \mid l \in \overline{p} \bullet l.x \}$$

Dans notre exemple, $\overline{poste\_facturé}^{*-}.montant\_TTC$ représente donc l'ensemble des montants des postes facturés sélectionnés. Il suffit alors de faire la somme sur les valeurs de cet ensemble pour obtenir le montant total de la facture.

Le schéma implicite associé à une transition a pour déclaration la composition respectivement des arcs consommation, information, et production liés à cette transition. De plus le schéma d'une transition intègre la contrainte relative à cette transition:

Facturer

ΔM

$client \xrightarrow{facturer} ; commande \xrightarrow{facturer} ; poste \xrightarrow{facturer} ; poste\_facturé \xrightarrow{facturer} ; facturer \xrightarrow{facture}$

$commande^{-}.numéro\_client = client^{?}.numéro\_client$

$poste^{-}.numéro\_commande = commande^{-}.numéro\_commande$

$poste\_facturé^{*-}.numéro\_commande = commande^{-}.numéro\_commande$

$facture^{+}.numéro\_commande = commande^{-}.numéro\_commande$

$facture^{+}.numéro\_client = client^{?}.numéro\_client$

$facture^{+}.nom\_client = client^{?}.nom\_client$

$facture^{+}.adresse\_client = client^{?}.adresse\_client$

$facture^{+}.montant\_TTC = SOMME(poste\_facturé^{*-}.montant\_TTC)$

III.3.2 Conclusion

Même si les réseaux formels s'inspirent des réseaux de Petri de haut niveau pour la structure des places et des transitions, l'introduction des arcs de types variés nous en éloigne. Seuls les arcs consommation et production sont comparables à ceux qu'on retrouve dans les réseaux de Petri classiques. L'arc information négative peut aussi être comparable à l'arc d'inhibition. Par contre, les arcs information ou ceux de pondération étoile n'ont pas d'équivalence dans les réseaux de Petri habituels. On perd ainsi certaines techniques d'analyse usuelles des réseaux de Petri, mais notre objectif est plus de décrire le plus précisément possible un problème en vue de concrétiser une solution que d'analyser les propriétés du système.

Il est possible de voir dans les réseaux formels un diagramme de flux de données, des références croisées données/traitements ou un support graphique du langage Z. Mais il ne faut pas se tromper, la combinaison de la notion de contrainte avec la règle de déclenchement d'une transition confère à l'ensemble une interprétation rigoureuse prenant en compte tous les aspects dynamiques et fonctionnels de la modélisation des systèmes d'information. Les règles de construction d'un réseau formel guident dans la pratique à une structuration facilitant la compréhension du problème. Mais il apparaît aussi que la structuration des données, c'est-à-dire des places, influence l'expression des contraintes. En effet, on constate qu'une structuration des données issue d'une modélisation relationnelle normalisée conduit à des expressions de contraintes plus simples. Il serait alors utile de définir un guide méthodologique de spécification à l'aide des réseaux formels afin mettre en évidence l'intérêt du modèle en isolant la part d'intuition et d'expertise. Cet aspect n'est pas abordé dans le cadre de cette thèse et fait partie des perspectives ouvertes par le travail présenté dans ce mémoire. La représentation des données est graphiquement moins riche qu'une représentation particulièrement destinée à ce problème comme pour un MCD, mais les réseaux formels permettent d'en capturer toute la spécification.

Notre modèle prend donc clairement en compte dans un cadre unitaire les données (avec leurs contraintes), le comportement dynamique et une description assez fine des processus.

III.4 Exemples détaillés de modélisation

Nous détaillerons ici cinq modélisations issues de problèmes choisis dans les domaines de la gestion, du traitement de l'information, de la planification, de la simulation, et du contrôle de process. Si elles ne prétendent pas définir une liste exhaustive de tout ce qui peut être résolu par les réseaux formels, nous essaierons, au travers de ces exemples, de dégager un début de démarche de modélisation utilisant les réseaux formels. En effet, à ce jour, aucune méthode n'a encore été décrite en vue de construire un réseau formel à partir de spécifications fonctionnelles issues d'un cahier des charges, bien que les réseaux de Petri classiques possèdent les leurs [ZHO 93]. Un des moyens de modéliser un réseau formel est donc de respecter fidèlement les spécifications du cahier des charges [ALL 97]. Il n'est pas très difficile de définir les places et les contraintes de place si nécessaire, les attributs des jetons, et les contraintes de clé. La première approche peut être d'identifier les jetons dans les spécifications, comme nous le ferions à l'aide de la méthode Merise [TAR 86], dans laquelle le marquage d'une place dénote l'état dans lequel les instances d'un certain type peuvent se trouver, et les contraintes de clé font référence aux identifiants. Les transitions dérivent des opérations appliquées sur les jetons. La principale difficulté provient de la modélisation des contraintes de transitions, due en grande partie à l'exclusion mutuelle (parallélisme et séquentialité) intrinsèque aux problèmes mettant en jeu des ressources partagées [ZHO 91].

D'autres études de cas pourront être trouvées dans le cadre de l'étude des HyperNets [LEF 98], comme le *noeud ferroviaire* ou le *centre d'usinage*, ou dans le cadre de la rétroconception [CAD 97], comme le *suivi des dossiers hospitaliers*.

Nous avons volontairement choisi de ne faire apparaître dans ce chapitre que des éléments ayant un rapport direct avec la théorie des réseaux formels ou apportant un supplément d'information. Le chapitre IV dédié à l'outil de développement *NetSPEC* s'intéressera plus particulièrement aux problèmes d'implémentation pratique et abordera plus spécifiquement le comportement des solutions proposées ici. D'autres informations pourront également être trouvées dans les annexes.

III.4.1 Notations

Dans un souci de clarté, et afin de simplifier la lecture des annotations d'un réseau formel, nous introduirons une notation plus suggestive que celle utilisée dans la partie précédente, notamment au niveau de l'expression des contraintes:

- Les <u>attributs</u> (d'un jeton) soumis à des contraintes de clé sont soulignés.
- Les opérateurs booléens ET, OU, et NON sont respectivement notés $\wedge$, $\vee$, et $\neg$.
- Le nombre de variables apparaissant sur un arc quelconque est omis s'il est égal à 1, ou donné explicitement dans le cas d'un arc de pondération n ou *:
  - P^n représente une variable indicée relative à la place P
  - P représente une variable relative à la place P et est équivalent à P^1
  - P* représente un ensemble de variables relatives à la place P, ensemble défini en compréhension

- Les différents types d'arcs, consommation, information, information négative, production sont respectivement matérialisés par les symboles « −, ?, ~, + » et apparaissent après une éventuelle pondération:
  - ⇒ P^- représente un singleton transitant sur un arc consommation
  - ⇒ $P^?$ représente un singleton transitant sur un arc information
  - ⇒ $P^\sim$ représente un singleton transitant sur un arc information négative
  - ⇒ P^+ représente un singleton transitant sur un arc production
- Les attributs des jetons sont désigné à l'aide du sélecteur « . »:
  - ⇒ P^{2-}.x représente l'attribut *x* d'un jeton qui est lui-même le second élément d'un couple transitant sur un arc consommation issu de la place *P*.

III.4.2 Exemple 1: le cas IFIP

Cet exemple, déjà traité à l'aide des réseaux formels dans [CAD 97], résume les fonctionnalités des traitements effectués sur la soumission d'un article en vue de l'organisation d'un congrès organisé par l'« *International Federation for Information Processing* ». Cet exemple a servi de cas de référence pour de nombreuses méthodes et formalismes [PAS 91],[SIB 91]... Succintement, il s'agit, après réception d'un papier, de nommer des examinateurs pour évaluer la qualité du document, de demander si nécessaire des corrections, et de former une session de présentation. Le cahier des charges est présenté sous la forme de quatorze clauses (C1 à C14), dont le résumé figure ci-dessous:

C1: Lorsqu'une lettre d'intention ou un article est reçu, si les auteurs ne sont pas déjà connus de l'organisme scientifique, celui-ci les enregistre.

C2: Seuls les lettres d'intention ou les articles arrivés avant la date de clôture de la remise des papiers seront pris en compte, sauf dérogation accordée par le comité de programme.

C3: Les projets papier (lettre d'intention ou version provisoire) sont répartis entre les membres du comité de programme qui deviennent alors examinateurs des papiers qui leur sont attribués.

C4: Dans le cas d'une lettre d'intention, si la version provisoire du papier n'arrive pas dans le mois qui suit la date de la remise des papiers, le projet de communication est annulé.

C5: Les membres du comité de programme doivent noter les papiers qui leur ont été soumis:
- à travers une note allant de 0 à 10
- pour les critères suivants: intérêt du papier, qualité du papier, qualité de la bibliographie

C6: Après notation, les papiers sont classés en 3 catégories: les papiers acceptés, les papiers refusés, et les papiers en ballottage. Les règles de classement sont définies par le comité de programme de chaque conférence. On souhaite que ces règles puissent être mémorisées par le système d'information, afin de pouvoir élaborer suivant un processus algorithmique, le classement des papiers. Les règles de classement sont de type moyenne pondérée: note moyenne=a_1(note moyenne critère 1)+a_2(note moyenne critère 2)+...+a_n(note

moyenne critère n) tel que $a_1+a_2+...+a_n=1$. Suivant la catégorie du papier, celui-ci est accepté, accepté avec des réserves (corrections demandées, et soumis à décision du comité de programme), ou refusé.

C7: Les papiers acceptés font l'objet de communications dans le cadre de sessions. Le comité de programme se réunit pour: arrêter définitivement la sélection des papiers, planifier les sessions, définir le programme des sessions (affecter les papiers), nommer les présidents de sessions, planifier les autres activités de la conférence.

C8: Le comité de programme veillera à ce que le présentateur d'un papier en soit bien l'auteur.

C9: On notera que l'examinateur d'un papier ne peut en être l'auteur ou le coauteur. Par contre, les membres du comité de programme peuvent soumettre des projets de communication.

C10: Le comité de programme suscite des papiers invités. Les papiers invités ne sont pas revus par les examinateurs, cependant une date limite de remise du papier invité est négociée avec chaque auteur. Si celui-ci ne remplit pas ses engagements, le papier invité ne sera pas programmé. Un papier invité répondant aux conditions requises pour être programmé, le sera suivant les mêmes règles qu'un papier accepté.

C11: Si par suite d'une défaillance d'un ou de plusieurs membres du comité de programme, un papier était noté par moins de 3 examinateurs, alors un ou plusieurs autres examinateurs devraient être nommés.

C12: Dans ce cas, si l'incident touche un nombre significatif de papiers, la date de réunion du comité de programme pourrait être décalée de quelques jours sur décision du comité de programme.

C13: Si l'auteur d'un papier accepté ne remet pas la version définitive de son papier dans les délais imposés par le comité de programme, ce papier pourra être refusé après décision du comité de programme.

C14: La version définitive d'un papier doit intégrer les corrections demandées par le comité de programme. Les demandes de correction sont établies sur la base du papier dans sa version provisoire. Les papiers définitifs pourront être revus par le comité de programme si celui-ci le juge nécessaire. Une règle générale est fixée à ce sujet pour chaque conférence.

III.4.2.1 Modélisation

La démarche que nous adopterons pour modéliser cet exemple est de respecter au mieux les spécifications du problème. Nous devons décrire deux types différents d'annotations pour spécifier complètement le réseau formel: les places et les transitions. Le résumé complet des annotations du cas *IFIP* est donné dans les Table I et Table II, le graphe complet est quant à lui illustré par la Figure 24. Nous ne détaillerons ci-dessous que quelques cas de figure, du plus simple au plus complexe.

Considérant le sous-système relatif aux courriers et aux auteurs, il est clair (clauses C1, C2, C4, et C13) qu'un jeton de la place *courrier* est composé de trois attributs (*nom_auteur*, *date_réception*, et *version*), dont les types sont évidents. Toutefois, étant donné que ces trois attributs sont insuffisants pour déterminer de manière unique un courrier, nous sommes contraints de créer un quatrième attribut, à savoir une référence

(*réf*). Nous devons également associer à cette place une contrainte de place, puisque la version du courrier ne peut prendre que trois valeurs dans l'ensemble {'*lettre*', '*provisoire*', '*définitif*'}. Comme les auteurs inconnus doivent être enregistrés, une place *auteur* doit être présente, dont les jetons n'ont qu'un seul attribut, le *nom* de l'auteur, qui est aussi une contrainte de clé, car il n'est ni nécessaire ni recommandé d'enregistrer plus d'une fois le même auteur. Pour compléter le sous-système, la transition *enregistrer_auteur* relie les places *courrier* et *auteur*, ce qui permet d'effectuer l'enregistrement. Les arcs sont du type information depuis la place *courrier* et production vers la place *auteur*, et la contrainte de transition peut être résumée à $auteur^{+}.nom = courrier^{?}.nom\_auteur$, ce qui semble suffisant pour effectuer l'action correcte.

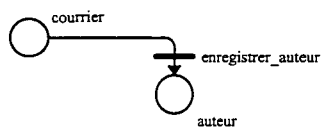


Figure 21: *IFIP* — Sous-système relatif à la transition *enregistrer_auteur*

Une contrainte de transition peut être plus complexe que celle présentée ci-dessus: c'est le cas du sous-système concernant le calcul de la note moyenne assignée à un papier (clause C6). La place *moyenne* contiendra des jetons dont les attributs seront la référence du papier (*réf*), le nom de l'auteur (*nom_auteur*), et la note moyenne (*valeur*). L'attribut *réf* est soumis à une contrainte de clé, puisqu'un papier ne peut pas être noté plusieurs fois. Pour un papier donné (place *papier*), la contrainte de transition doit calculer la note moyenne à partir des notes attribuées par les examinateurs (place *note*), si et seulement si la fin de notation est active (place *fin_notation*) et si le papier a été noté par au moins trois examinateurs. Il nous faut donc un arc information depuis la place *fin_notation*, un arc consommation depuis la place *papier*, un arc consommation de poids * depuis la place *note*, et un arc production vers la place *moyenne*. L'expression de cette contrainte sera alors:

$$
\begin{aligned}
& fin\_notation^{?}.état = \text{'oui'} \\
\wedge\ & note^{*-}.réf = papier^{-}.réf \\
\wedge\ & CARD(note^{*-}) \geq 3 \\
\wedge\ & moyenne^{+}.réf = papier^{-}.réf \\
\wedge\ & moyenne^{+}.nom\_auteur = papier^{-}.nom\_auteur \\
\wedge\ & moyenne^{+}.valeur = MOY(note^{*-}.sujet) \times coef\_sujet \\
& \qquad + MOY(note^{*-}.qualité) \times coef\_qualité \\
& \qquad + MOY(note^{*-}.biblio) \times coef\_biblio
\end{aligned}
$$

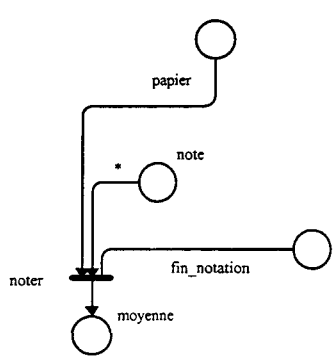


Figure 22: *IFIP* — Sous-système relatif à la transition *noter*

Un dernier exemple concerne le sous-système relatif à la réaffectation d'un examinateur (clause C11). Son intérêt réside dans l'emploi d'un arc de type information négative. Lorsque la notation a pris fin (place *fin_notation*), qu'un papier a reçu moins de trois notes (place *note*), qu'un examinateur n'a pas donné sa note (place *note*), alors un nouvel examinateur doit entrer en jeu (places *examinateur*, *réaffectation*). L'arc information négative apparaît dans le cadre de la vérification de l'absence d'une note de la part d'un examinateur pour un papier donné. La contrainte de transition *réaffecter_examinateur* apparaît alors telle que:

$$
\begin{aligned}
& \mathit{fin\_notation?.état} = \text{'oui'} \\
\wedge\ & \mathit{examinateur\text{-}.réf} = \mathit{note}^{(*)?}.\mathit{réf} \\
\wedge\ & \mathit{CARD}(\mathit{note}^{*?}) < 3 \\
\wedge\ & \mathit{examinateur}^{\sim}.\mathit{réf} = \mathit{note}^{\sim}.\mathit{réf} \\
\wedge\ & \mathit{examinateur}^{\sim}.\mathit{nom\_examinateur} = \mathit{note}^{\sim}.\mathit{nom\_examinateur} \\
\wedge\ & \mathit{réaffectation}^{\sim}.\mathit{réf} = \mathit{examinateur}^{\sim}.\mathit{réf} \\
\wedge\ & \mathit{réaffectation\_examinateur} = \mathit{examinateur}^{\sim}.\mathit{nom\_examinateur} \\
\wedge\ & \mathit{affectation}^{+}.\mathit{réf} = \mathit{réaffectation}^{\sim}.\mathit{réf} \\
\wedge\ & \mathit{affectation}^{+}.\mathit{nom\_examinateur} = \mathit{réaffectation}^{\sim}.\mathit{nom\_examinateur}
\end{aligned}
$$

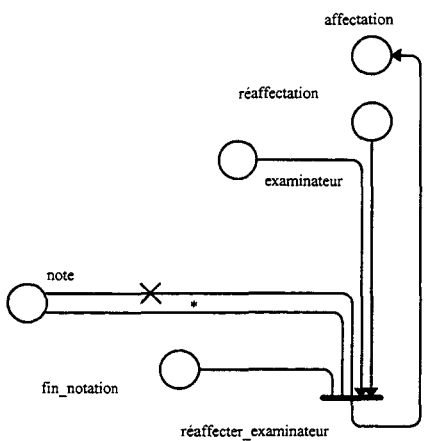


Figure 23: *IFIP* — Sous-système relatif à la transition *réaffecter_examinateur*

III.4.2.2 Graphe, annotation des places et des transitions

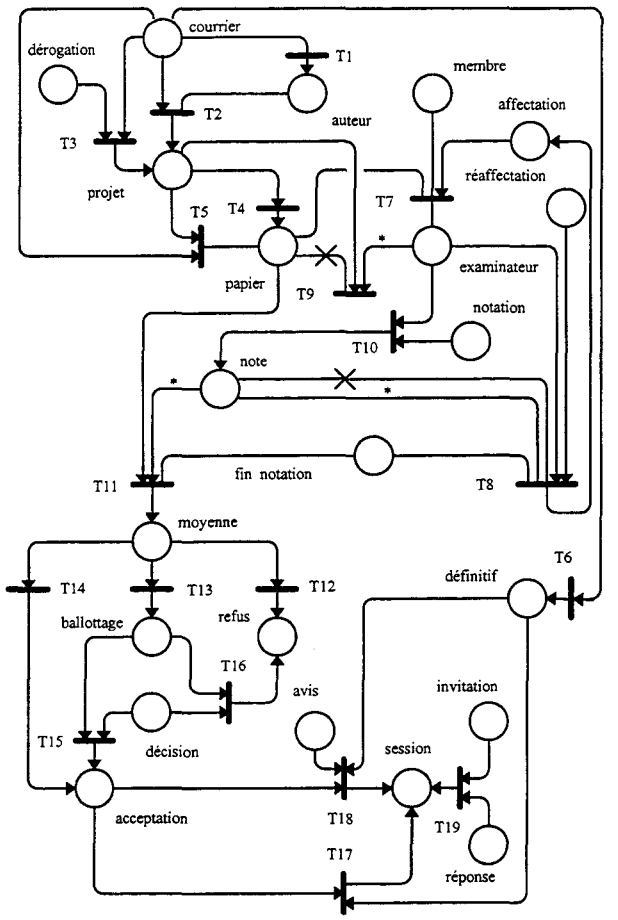


Figure 24: *IFIP* — Graphe complet du réseau formel

PLACES	ATTRIBUTS DES JETONS ET CONTRAINTES DE CLE	CONTRAINTES DE PLACES
courrier	réf: entier nom_auteur: chaîne date_réception: date version: chaîne	version ∈ {'lettre','provisoire','définitif'}
auteur	nom: chaîne	
dérogation	réf: entier accord: chaîne	accord ∈ {'oui','non'}
projet	réf: entier nom_auteur: chaîne date_réception: date version: chaîne	version ∈ {'lettre','provisoire'}
papier	réf: entier nom_auteur: chaîne	
membre	nom: chaîne	
affectation	réf: entier nom_examinateur: chaîne	
examinateur	réf: entier nom: chaîne	
réaffectation	réf: entier nom_ancien_examinateur: chaîne nom_nouveau_examinateur: chaîne	
fin_notation	état: chaîne	état ∈ {'oui','non'}
notation	réf: entier nom_examinateur: chaîne sujet: réel qualité: réel biblio: réel	0 ≤ sujet ≤ 10 0 ≤ qualité ≤ 10 0 ≤ biblio ≤ 10

PLACES	ATTRIBUTS DES JETONS ET CONTRAINTES DE CLE	CONTRAINTES DE PLACES
note	réf: entier nom_examinateur: chaîne sujet: réel qualité: réel biblio: réel	
ballottage	réf: entier nom_auteur: chaîne	
refus	réf: entier nom_auteur: chaîne	
moyenne	réf: entier nom_auteur: chaîne valeur: réel	
acceptation	réf: entier nom_auteur: chaîne	
décision	réf: entier accord: chaîne	accord ∈ {'oui','non'}
avis	réf: entier accord: chaîne	accord ∈ {'oui','non'}
session	réf: entier nom_présentateur: chaîne	
invitation	réf: entier auteur: chaîne date_limite: date	
réponse	réf: entier date_réception: date	
définitif	réf: entier nom_auteur: chaîne date_réception: date	

Table I: *IFIP* — Annotation des places

TRANSITIONS	CONTRAINTES
enregistrer_auteur (T1)	auteur+.nom = courrier?.auteur_nom
recevoir_courrier (T2)	courrier-.date_réception ≤ date_clôture courrier-.version = 'lettre' ∨ courrier-.version = 'provisoire' courrier-.auteur_nom = auteur?.nom projet+.réf = courrier-.réf projet+.nom_auteur = courrier-.nom_auteur projet+.date_réception = courrier-.date_réception projet+.version = courrier-.version
déroger_retard (T3)	courrier-.date_réception > date_retard courrier-.version = 'lettre' ∨ courrier-.version = 'provisoire' dérogation-.réf = courrier-.réf dérogation-.accord = 'oui' projet+.réf = courrier-.réf projet+.nom_auteur = courrier-.nom_auteur projet+.date_réception = courrier-.date_réception projet+.version = courrier-.version
recevoir_provisoire_direct (T4)	projet-.version = 'provisoire' papier+.réf = projet-.réf papier+.nom_auteur = projet-.nom_auteur
recevoir_provisoire_lettre (T5)	courrier-.version = 'provisoire' projet-.version = 'lettre' projet-.réf = courrier-.réf papier+.réf = courrier-.réf papier+.nom_auteur = courrier-.nom_auteur
recevoir_définitif (T6)	courrier-.version = 'définitif' définitif+.réf = courrier-.réf définitif+.nom_auteur = courrier-.nom_auteur définitif+.date_réception = courrier-.date_réception
affecter_examinateur (T7)	affectation-.réf = papier?.réf affectation-.nom_examinateur = membre?.nom affectation-.nom_examinateur ≠ papier?.nom_auteur examinateur+.réf = affectation-.réf examinateur+.nom = affectation-.nom_examinateur
réaffecter_examinateur (T8)	fin_notation = 'oui' réaffectation-.réf = examinateur-.réf réaffectation-.nom_ancien_examinateur = examinateur-.nom examinateur-.réf = note~.réf ∧ examinateur-.nom = note~.nom_examinateur note*?.réf = examinateur-.réf ∧ CARD(note*?) < 3 affectation+.réf = réaffectation-.réf affectation+.nom_examinateur = réaffectation-.nom_nouveau_examinateur
annuler (T9)	projet-.réf = examinateur*-.réf projet-.réf = papier~.réf projet-.version = 'lettre' date_courante - projet-.date_réception > 1 mois

TRANSITIONS	CONTRAINTES
noter (T10)	notation-.réf = examinateur-.réf notation-.nom_examinateur = examinateur-.nom note+.ref = notation-.réf note+.nom_examinateur = notation-.nom_examinateur note+.sujet = notation-.sujet note+.qualité = notation-.qualité note+.biblio = notation-.biblio
calculer_moyenne (T11)	fin_notation = 'oui' note(*)-.réf = papier-.réf CARD(note*-) ≥ 3 moyenne+.réf = papier-.réf moyenne+.nom_auteur = papier-.nom_auteur moyenne+.valeur = MOY(note*-.sujet)×coef_sujet + MOY(note*-.qualité)×coef_qualité + MOY(note*-.biblio)×coef_biblio
classer_refus (T12)	moyenne-.valeur < seuil_refusé refus+.réf = moyenne-.réf refus+.nom_auteur = moyenne-.nom_auteur
classer_ballottage (T13)	moyenne-.valeur ≥ seuil_refusé moyenne-.valeur < seuil_accepté ballottage+.réf = moyenne-.réf ballottage+.nom_auteur = moyenne-.nom_auteur
classer_acceptation (T14)	moyenne-.valeur ≥ seuil_accepté acceptation+.réf = moyenne-.réf acceptation+.nom_auteur = moyenne-.nom_auteur
traiter_ballottage_accepté (T15)	ballottage-.réf = décision-.réf décision-.accord = 'oui' acceptation+.réf = ballottage-.réf acceptation+.nom_auteur = ballottage-.nom_auteur
traiter_ballottage_refusé (T16)	ballottage-.réf = décision-.réf décision-.accord = 'non' refus+.réf = ballottage-.réf refus+.nom_auteur = ballottage-.nom_auteur
organiser_session_définitif (T17)	acceptation-.réf = définitif-.réf définitif-.date_réception ≤ date_limite session+.réf = définitif-.réf session+.nom_présentateur = définitif-.nom_auteur
organiser_session_retard (T18)	acceptation-.réf = définitif-.réf définitif-.date_réception > date_limite définitif-.réf = avis-.réf avis-.accord = 'oui' session+.réf = définitif-.réf session+.nom_présentateur = définitif-.nom_auteur
inviter (T19)	invitation-.réf = réponse-.réf réponse-.date_réception ≤ invitation-.date_limite session+.réf = invitation-.réf session+.nom_présentateur = invitation-.nom_auteur

Table II: *IFIP* — Annotations des transitions

III.4.2.3 Conclusion

Proposer une solution au cas *IFIP* a été relativement aisé. Les spécifications sont précises, ce qui implique un repérage facile des places, des attributs des jetons, et des actions. Toutefois, un effort certain a été déployé pour permettre d'arriver à une modélisation dont toutes les contraintes de transition sont exclusives. Le cas *IFIP* sera redéveloppé dans le chapitre suivant en servant de modèle d'illustration de l'outil *NetSPEC*.

III.4.3 Exemple 2: la gestion commerciale GENCOD

La gestion commerciale *GENCOD* est le cas d'école qui a été référencé dans les chapitres précédents. Ses spécifications sont plus lâches que celles du cas *IFIP* et permettent certainement un plus grand degré de liberté. Cet exemple est intéressant car nous verrons la puissance d'expression s'exprimer, en particulier dans les contraintes de transition. Nous nous intéresserons ici aux annotations du réseau formel, plutôt qu'à la méthode utilisée qui ressemble fortement à celle de l'exemple précédent.

Il s'agit ici de modéliser ce que pourrait être la gestion d'un stock d'articles d'une entreprise, et la gestion de la facturation des commandes. Les *clients* de l'entreprise passent des *commandes* contenant un certain nombre de lignes de commande appelées *postes*. Chaque poste concerne un *article* commercialisé par l'entreprise, article qui est stocké dans un *entrepôt* précis. Le client doit recevoir sa *facture* dès que la totalité de la commande peut être honorée, c'est-à-dire dès qu'aucun article ne manque dans le stock pour satisfaire sa commande. Le stock est regarni par les fournisseurs de l'entreprise. Chaque article reçoit alors un complément qui représente la quantité d'articles manquants pour satisfaire toutes les commandes non honorées, quantité augmentée d'un fond de roulement égal à environ 10%.

III.4.3.1 Graphe, annotation des places et des transitions

Les places *client*, *commande*, *poste*, *article*, et *facture* sont directement issues des spécifications. La place *poste_facturé* est ajoutée à l'ensemble et permet d'éliminer peu à peu les postes honorés. La transition *regarnir_stock* modélise l'alimentation du stock d'article, la transition *facturer_poste* permet de mettre à jour le stock, et la transition *facturer* crée les factures.

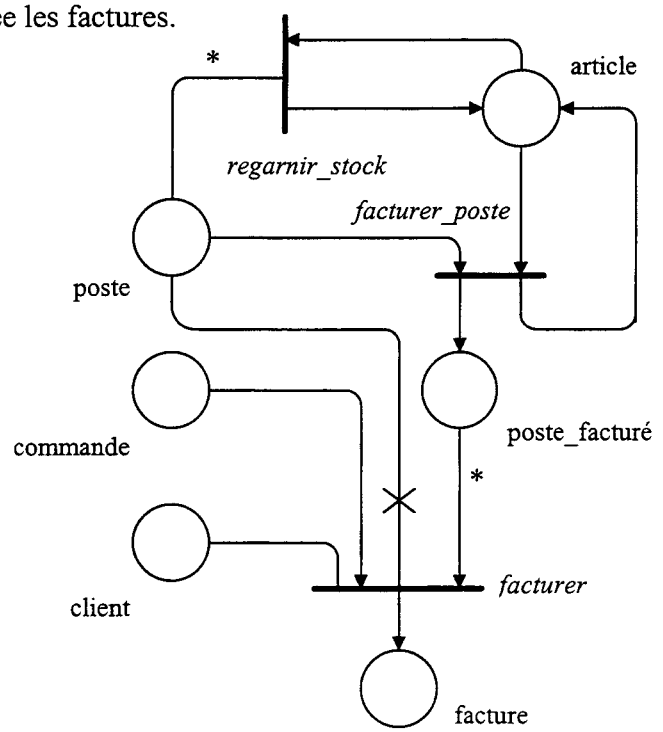


Figure 25: *GENCOD* — Graphe complet de l'application

PLACES	ATTRIBUTS DES JETONS ET CONTRAINTES DE CLE
client	numéro_client: entier adresse_client: chaîne
commande	numéro_commande: entier numéro_client: entier
poste	numéro_commande: entier code_article: entier quantité_commandée: entier

PLACES	ATTRIBUTS DES JETONS ET CONTRAINTES DE CLE
facture	numéro_commande: entier numéro_client: entier adresse_client: chaîne montant_TTC: réel
article	code_article: entier prix_unitaire: réel quantité_stock: entier
poste_facturé	numéro_commande: entier code_article: entier montant_TTC: réel

Table III: *GENCOD* — Annotation des places

TRANSITIONS	CONTRAINTES DE TRANSITION
facturer_poste	$poste^{-}.code\_article = article^{-}.code\_article$ $\wedge\ article^{-}.quantité\_stock \geq poste^{-}.quantité\_commandée$ $\wedge\ poste\_facturé^{+}.numéro\_commande = poste^{-}.numéro\_commande$ $\wedge\ poste\_facturé^{+}.code\_article = poste^{-}.code\_article$ $\wedge\ poste\_facturé^{+}.montant\_TTC = poste^{-}.quantité\_commandée \times article^{-}.prix\_unitaire$ $\wedge\ article^{+}.code\_article = article^{-}.code\_article$ $\wedge\ article^{+}.prix\_unitaire = article^{-}.prix\_unitaire$ $\wedge\ article^{+}.quantité\_stock = article^{-}.quantitée\_stock - poste^{-}.quantité\_commandée$
facturer	$commande^{-}.numéro\_client = client^{?}.numéro\_client$ $\wedge\ poste^{-}.numéro\_commande = commande^{-}numéro\_commande$ $\wedge\ poste\_facturé^{*-}.numéro\_commande = commande^{-}.numéro\_commande$ $\wedge\ facture^{+}.numéro\_commande = commande^{-}.numéro\_commande$ $\wedge\ facture^{+}.numéro\_client = client^{?}.numéro\_client$ $\wedge\ facture^{+}.adresse\_client = client^{?}.adresse\_client$ $\wedge\ facture^{+}.montant\_TTC = SOMME(poste\_facturé^{*-}.montant\_TTC)$
regarnir_stock	$article^{-}.code\_article = poste^{*?}.code\_article$ $\wedge\ article^{-}.quantité\_stock < SOMME(poste^{*?}.quantité\_commandée)$ $\wedge\ article^{+}.code\_article = article^{-}.code\_article$ $\wedge\ article^{+}.prix\_unitaire = article^{-}.prix\_unitaire$ $\wedge\ article^{+}.quantité\_stock = 1{,}1 \times (SOMME(poste^{*?}.quantité\_commandée) - article^{-}.quantité\_stock)$

Table IV: *GENCOD* — Annotation des transitions

III.4.3.2 Conclusion

Cet exemple met en valeur la puissance d'annotation des réseaux formels. En particulier, la contrainte de la transition *facturer* est à rapprocher de l'exemple de requête SQL développée dans le chapitre concernant les DBMS relationnels (Exemple 11 du Paragraphe I.4.1.3). Nous pouvons aussi remarquer que les contraintes des deux transitions *regarnir_stock* et *facturer_poste* ont été exprimées de manière exclusive très naturellement, bien qu'elles opèrent sur une ressource partagée: les articles. Les exemples suivants montreront que l'exclusivité entre transitions peut être beaucoup plus délicate à appréhender.

III.4.4 Exemple 3: les empilements de CUBES

Nous avons vu dans les exemples précédents des cas de modélisation ne faisant appel qu'à une retranscription la plus fidèle possible des spécifications d'un problème. Nous abordons ici un cas pour lequel nous avons choisi de modéliser une stratégie particulière destinée à répondre aux spécifications du problème, bien que cette stratégie n'apparaisse pas dans les spécifications [WIL 88].

On considère un ensemble de *cubes* et une *table*. Chacun des cubes est posé sur un autre cube ou sur la table (considérée comme étant de dimension infinie). Soient deux actions *empiler* et *dépiler*, permettant respectivement :

- de placer un premier cube posé sur la table et sur lequel il n'y a pas d'autre cube, sur un second cube sur lequel il n'y a pas d'autre cube.
- de poser sur la table un premier cube posé sur un second cube et sur lequel il n'y a pas d'autre cube.

Le problème consiste à exécuter une suite d'actions *empiler* et *dépiler* dans un ordre tel que l'on puisse passer d'une configuration initiale à une configuration finale d'empilements, comme il est illustré par la Figure 26.

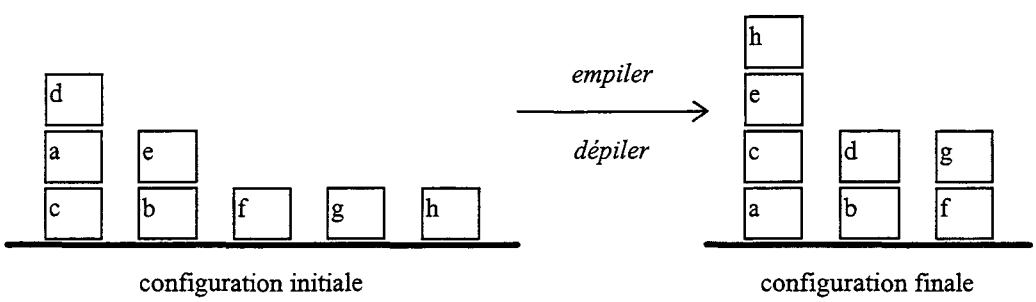


Figure 26: *CUBES* — Position du problème

III.4.4.1 Modélisation

Nous dirons qu'un cube est correctement placé si sa position appartient à la configuration finale (sur la table ou sur un autre cube lui-même correctement placé). Un empilement de cubes sera correctement placé si tous les cubes qui le constituent le sont eux-mêmes.

Les actions *empiler* et *dépiler* ne devront être utilisées que si elles nous permettent de nous rapprocher de la configuration finale. Notamment, il ne sera pas autorisé d'empiler un cube sur un cube mal placé, ou de dépiler un cube correctement placé.

Chaque action *empiler* et *dépiler* se verra associer un gain (intérêt de l'action), dont la valeur sera 2 si un cube est dépilé et que sa position finale est la table, 1 si un cube est empilé sur un cube correctement placé, et 0 si un cube est dépilé et que sa position finale

n'est pas la table (dans tous les autres cas d'actions possibles, le gain sera négatif). La séquence d'actions qui sera effectivement réalisée privilégiera toujours l'action de gain maximal. Ceci nous permettra d'asseoir assez rapidement la base d'un empilement final, tout en évitant de dépiler tous les cubes avant de les empiler correctement (solution naïve).

Pour former une séquence gagnante, les gains de toutes les différentes actions possibles seront évalués, puis l'action de gain maximal sera alors exécutée. Son exécution entraînera vraisemblablement de nouvelles actions possibles, prioritaires sur celles déjà évaluées et le cycle pourra continuer jusqu'à atteindre la configuration finale.

III.4.4.2 Graphe, annotations des places et des transitions

Le graphe du réseau formel est donné en Figure 27. Le réseau ne possède que 3 places: une place *cube* contenant tous les cubes de la configuration courante (qui au départ contient la configuration initiale), une place *but* qui contient la configuration finale, et une place *fin_action* indiquant qu'aucune action de gain positif n'est envisageable.

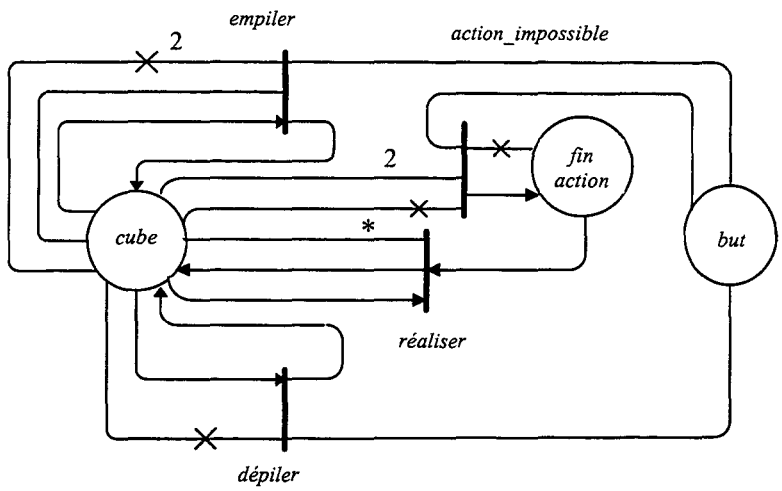


Figure 27: *CUBES* — Graphe du réseau formel

Les annotations des places et des transitions sont décrites dans les Table V et Table VI. Un jeton de la place *cube* possèdent 5 attributs: son *nom* (contrainte de clé), sa *position* (nom du cube sur lequel il est posé ou 'TABLE'), son *placement* ('CORRECT' ou 'INCORRECT'), le *gain* espéré par une éventuelle action le concernant, et la *cible* (nom d'un autre cube ou 'TABLE'). Les différentes contraintes de place indiquent les valeurs possibles des attributs *placement* et *gain*, ainsi que le fait qu'un cube ne peut être placé sur lui-même.

PLACES	ATTRIBUTS DES JETONS ET CONTRAINTES DE CLE	CONTRAINTES DE PLACE
cube	nom: chaîne de caractères position: chaîne de caractères placement: chaîne de caractères gain: entier cible: chaîne de caractères	nom ≠ position nom ≠ cible placement ∈ { 'CORRECT', 'INCORRECT' } gain ∈ { -1, 0, 1, 2 }
but	nom: chaîne de caractères position: chaîne de caractères	nom ≠ position
fin_action	état: chaîne de caractères	

Table V: *CUBES* — Annotations des places

Les contraintes de transition ont été implémentées en considérant qu'un cube posé sur la table ne peut pas être dépilé, et qu'un cube posé sur un autre cube ne peut pas être empilé: les transitions *empiler* et *dépiler* sont donc exclusives et n'ont pas de priorité relative entre elles. Ceci nous permet donc d'éviter un conflit qui consisterait en 2 actions possibles sur le même cube. La transition *réaliser*, dont le but est d'appliquer les actions choisies (c'est-à-dire de modifier le marquage de la configuration courante), est plus délicate à exprimer. En effet, celle-ci devant exécuter l'action de plus haut gain, elle ne peut être tirée que lorsque qu'aucune action empiler ou dépiler de gain positif n'est plus envisageable: la contrainte de la transition *réaliser* doit être exclusive à la fois avec la contrainte de la transition *empiler* (aucun cube ne peut potentiellement être empilé) et avec la contrainte de la transition *dépiler* (aucun cube ne peut potentiellement être dépilé).

Si l'on considère que:

- un cube peut être dépilé si et seulement si:
  - ⇒ Condition A: son placement est incorrect
  - ⇒ Condition B: il n'y a aucun cube sur lui
  - ⇒ Condition C: il n'a été l'objet d'aucune action
  - ⇒ Condition D: sa position actuelle n'est pas la table
- un cube peut être empilé si et seulement si:
  - ⇒ Condition A: son placement est incorrect
  - ⇒ Condition B: il n'y a aucun cube sur lui
  - ⇒ Condition C: il n'a été l'objet d'aucune action
  - ⇒ Condition E: sa position actuelle est la table
  - ⇒ Condition F: sa position finale est un cube correctement placé

alors la contrainte de la transition *réaliser* doit contenir la condition complexe

$$\neg ((A \wedge B \wedge C \wedge (D \vee (E \wedge F))$$

Afin d'alléger l'écriture de la contrainte de la transition *réaliser*, nous avons préféré introduire la place *fin_action* et la transition *action_impossible*. La transition prend en charge la condition complexe vue plus haut et produit un jeton dans la place *fin_action*. La transition *réaliser* teste l'existence de ce jeton et le cas échéant, réalise l'action demandée.

TRANSITIONS	CONTRAINTES DE TRANSITION	COMMENTAIRES
empiler	cube$^-$.position = 'TABLE' ∧ cube$^-$.placement ≠ 'CORRECT' ∧ cube$^-$.nom = cube$^{1\sim}$.position ∧ cube$^-$.gain = -1 ∧ cube$^?$.nom = cube$^{2\sim}$.position ∧ cube$^?$.placement = 'CORRECT' ∧ cube$^-$.nom = but$^?$.nom ∧ but$^?$.position = cube$^?$.nom ∧ cube$^+$.nom = cube$^-$.nom ∧ cube$^+$.position = cube$^-$.position ∧ cube$^+$.placement = 'INCORRECT' ∧ cube$^+$.gain = 1 ∧ cube$^+$.cible = cube$^?$.nom	Choisir un 1er cube posé sur la table, qui est mal placé, sur lequel il n'y a pas d'autre cube, et qui n'a été l'objet d'aucune action. Choisir un 2nd cube sur lequel il n'y a rien, et qui est bien placé. Vérifier si le 1er cube doit être placé sur le 2nd cube. Modifier les attributs du 1er cube en indiquant une action empiler sur la cible représentant le 2nd cube.
dépiler	cube$^-$.placement ≠ 'CORRECT' ∧ cube$^-$.position ≠ 'TABLE' ∧ cube$^-$.nom = cube$^{\sim}$.position	Choisir un cube mal placé, qui n'est pas sur la table, sur lequel il n'y a pas d'autre cube,

	$\wedge$ cube$^-$.gain = -1 $\wedge$ cube$^-$.nom = but$^?$.nom $\wedge$ cube$^+$.nom = cube$^-$.nom $\wedge$ cube$^+$.position = cube$^-$.position $\wedge$ cube$^+$.placement = 'INCORRECT' $\wedge$ ((but$^?$.position = 'TABLE' $\wedge$ cube$^+$.gain = 2) $\vee$ (but$^?$.position $\neq$ 'TABLE' $\wedge$ cube$^+$.gain = 0)) $\wedge$ cube$^+$.cible = 'TABLE'	et qui n'a été l'objet d'aucune action. Rechercher sa configuration finale. Modifier les attributs du cube, en indiquant son gain correct, suivant que sa position finale est sur la table ou sur un autre cube.
action_impossible	fin_action$^-$.état = 'VRAI' $\wedge$ (cube$^{1?}$.placement = 'INCORRECT' $\wedge$ cube$^-$.position = cube$^{1?}$.nom $\wedge$ cube$^{1?}$.gain = -1 $\wedge$ (cube$^{1?}$.position $\neq$ 'TABLE' $\wedge$ but$^?$.nom = cube$^{1?}$.nom) $\vee$ (cube$^{1?}$.position = 'TABLE' $\wedge$ but$^?$.nom = cube$^{1?}$.nom $\wedge$ but$^?$.position = cube$^{2?}$.nom $\wedge$ cube$^{2?}$.placement = 'CORRECT')) $\wedge$ fin_action$^+$.état = 'VRAI'	Expression de la condition d'exclusivité par rapport aux contraintes *dépiler* et *empiler* indiquer qu'aucune action n'est possible
réaliser	fin_action$^-$.état = 'VRAI' $\wedge$ cube$^-$.gain $\neq$ -1 $\wedge$ cube$^-$.gain = MAX(cube$^{*?}$.gain) $\wedge$ cube$^+$.nom = cube$^-$.nom $\wedge$ cube$^+$.position = cube$^-$.cible $\wedge$ ((cube$^-$.gain = 0 $\wedge$ cube$^+$.placement = 'INCORRECT') $\vee$ (cube$^-$.gain $\neq$ 0 $\wedge$ cube$^+$.placement ='CORRECT')) $\wedge$ cube$^+$.gain = -1 $\wedge$ cube$^+$.cible = ''	Tester qu'aucune action n'est possible Choisir un cube qui a été l'objet d'une action, et dont le gain espéré est maximal. Modifier les attributs de ce cube; en changeant sa position selon sa cible, et son placement suivant l'action réalisée.

Table VI: *CUBES* — Annotations des transitions

III.4.4.3 Conclusion

Bien que la stratégie utilisée soit relativement simple, nous voyons que l'expression des contraintes de transition peut devenir complexe très rapidement. Nous verrons dans le chapitre suivant comment s'affranchir d'une telle complexité en introduisant la notion de priorité entre transitions.

Cet exemple est à notre avis intéressant car il permet de montrer que:

- Du fait de l'emploi d'une stratégie opératoire, nous avons montré que les réseaux formels peuvent être utilisés comme « langage de description opérationnelle » d'un algorithme.
- Bien que ne possédant pas de mécanisme de *backtrack* comme en Prolog, qui peut être utilisé pour résoudre le problème, nous avons montré qu'une expression correcte des contraintes de transition permet d'accéder relativement facilement à une solution correcte.

Le résultat de la simulation de ce réseau formel nous donne la suite d'actions réalisées suivante: *empiler g sur f, dépiler d, dépiler a, empiler c sur a, dépiler e, empiler d sur b, empiler e sur c, empiler h sur e.* Cependant, rien ne montre que cette suite est la plus courte (ce n'était pas une contrainte à respecter dans l'énoncé).

III.4.5 Exemple 4: l'architecture pipeline et data-flow de l'IBM360

Cet exemple, tiré de [KOG 81], permet de déterminer le déplacement des données dans un pipeline constituant l'unité d'exécution en virgule flottante de l'unité centrale de l'IBM360, de type data-flow. Il est intéressant à plusieurs titres car:

- Un pipeline devant être sollicité en minimisant les temps morts (i.e. un événement arrivant doit pouvoir être pris en compte le plus rapidement possible), nous verrons qu'il est possible de concevoir des transitions ayant la capacité d'être tirées dès qu'elles sont franchissables.
- L'analyse des résultats permet de représenter facilement le comportement dynamique du pipeline, alors qu'il est à priori difficile de le définir, au vu de la complexité du problème.

Le synoptique de l'unité d'exécution est donné en Figure 29. Cette unité d'exécution comporte 8 composants:

- une file d'attente des opérandes provenant de la mémoire centrale (FPOB1 à FPOB 6) par l'intermédiaire de deux canaux de communication distincts (la mémoire centrale est conçue pour permettre la lecture de deux données simultanément).
- une file d'attente d'instructions (FPOS) provenant de l'unité de recherche des instructions en mémoire centrale.
- un décodeur d'instruction (IDEC) contrôlant les autres composants.
- un ensemble de 4 registres (FPR0 à FPR3) associés à leurs indicateurs: un *busy bit* et un *tag*. Une valeur 0 dans le busy bit indique que le registre contient une donnée exploitable, une valeur 1 indique que le registre attend une donnée dont la source est indiquée dans le tag. La source peut être l'un des FPOB, ou l'un des composants suivants.
- un additionneur pipeline à 2 étages, associé à 3 stations de réservation (RS10 à RS12), elles-mêmes composées d'une partie gauche et d'une partie droite représentant les deux opérandes impliqués dans une opération. Chacune des parties gauche et droite est associée à un tag, dont le rôle est identique au tag des registres.
- un multiplieur/diviseur pipeline à 3 étages, associé à 2 stations de réservation (RS8 et RS9), elles-mêmes composées d'une partie gauche et d'une partie droite représentant les deux opérandes impliqués dans une opération. Chacune des parties gauche et droite est associée à un tag, dont le rôle est identique au tag des registres.
- Une file d'attente de résultats (SDR13 à SDR15) provenant des 2 opérateurs précédents et à destination de la mémoire centrale (canal de communication en écriture). Le tag associé à chaque entrée a un rôle identique au tag des registres.
- On peut considérer le bus commun (CDB) comme le dernier composant de l'unité centrale. Son rôle est de servir de canal de communication interne entre les autres composants.

Ces différents composants font un usage intensif des pipelines (6 au total), et l'architecture peut faire appel au parallélisme lorsque les ressources mises en jeu sont exclusives. L'architecture de type data-flow implique le 'pistage' des données ou plus exactement de leur disponibilité, ce qui est réalisé au moyen des busy bits et des tags.

L'objectif de cette étude est de pouvoir déterminer le nombre de cycles nécessaires pour exécuter le petit programme suivant:

```
100   LOAD  FPR0,A      ; charger le registre FPR0 avec la variable A
101   ADD   FPR0,FPR1   ; additionner FPR0 avec FPR1, ranger le résultat dans FPR0
102   ADD   FPR0,B      ; additionner FPR0 avec la variable B, ranger le résultat dans FPR0
103   ADD   FPR2,FPR3   ; additionner FPR2 avec FPR3, ranger le résultat dans FPR2
104   MPY   FPR1,FPR2   ; multiplier FPR1 avec FPR2, ranger le résultat dans FPR1
105   ST    FPR1,C      ; stocker FPR1 dans la variable C
106   DIV   FPR1,FPR0   ; diviser FPR1 par FPR0, ranger le résultat dans FPR1
```

Quelques hypothèses ont été définies afin de bien se rendre compte de la difficulté d'étudier 'manuellement' un problème de ce type:

- La mémoire centrale étant en asynchronisme total avec l'unité d'exécution: les données reçues par l'unité d'exécution peuvent l'être dans un ordre différent de celui des demandes de lecture. Nous émettrons l'hypothèse que la variable B arrive avant la variable A, alors que la demande de lecture de A (instruction 100) est faite avant celle de B (instruction 102). B arrive juste avant MPY, A arrive juste après. D'autre part, B sera stocké dans le buffer FPOB3, et A dans le buffer FPOB4.
- Les opérateurs de multiplication et de division sont communs, par conséquent il n'est pas possible d'effectuer une multiplication et une division en parallèle.

III.4.5.1 Graphe, annotation des places et des transitions

Nous avons décidé d'adopter une modélisation qui associe à chaque opérateur ou registre une place: *FPOB* (Floating-Point Operand Buffer), *FPOS* (Floating-Point Operation Stack), *IDEC* (Instruction Decoder), *FPR* (Floating-Point Register), *ARS* (Add Reservation Station), *MDRS* (Multiply/Divide Reservation Station), *SDR* (Store Data Register), et *CDB* (Common Data Bus). Deux autres places ont été créées de toute pièce: *CYCLE* qui indique le numéro du cycle machine, et *EVENT* qui indique l'instant d'arrivée des événements extérieurs (variables A et B).

Il apparaît dans ce problème que chaque composant du système est une ressource partagée et critique (par exemple on ne peut pas lire et écrire simultanément dans le même registre). La modélisation des contraintes de transition peut alors devenir inexprimable tant leur complexité dans l'expression de leur exclusivité peut devenir démesurée. Aussi nous avons choisi une modélisation qui consiste à découper chaque cycle machine en plusieurs phases. Chaque phase pourra déclencher le tir d'un groupe de transitions (chaque groupe de transitions sera donc exclusif par rapport à un autre, car manipulant des ressources partagées), tandis que dans un groupe donné, chaque transition sera exclusive par rapport aux autres transitions puisque qu'agissant sur des ressources différentes. Autrement dit, le parallélisme de l'architecture ne peut s'exprimer que dans les limites d'un groupe.

Brièvement, les phases sont décomposées en:

- Phase 1: prise en compte des événements extérieurs
- Phase 2: déplacement des données disponibles vers leurs destinations
- Phase 3: changement d'étage de calcul de l'opérateur pipeline d'addition
- Phase 4: changement d'étage de calcul de l'opérateur pipeline de multiplication
- Phase 5: déplacement des résultats disponibles vers leurs destinations
- Phase 6: armement des opérateurs pipelines d'addition et de multiplication
- Phase 7: recherche d'une instruction
- Phase 8: décodage d'une instruction
- Phase 9: passage au cycle machine suivant

Nous ne donnons pas ici le détail du graphe (28 transitions) et des tables d'annotations et invitons le lecteur à se reporter à l'Annexe D.

III.4.5.2 Conclusion

Par rapport à l'exemple précédent, ce n'est pas ici un algorithme (les instructions à simuler ne sont que des données pour le réseau formel), mais la description fonctionnelle des composants d'une machine et leur comportement face aux événements qui sont ici décrits par les places et les contraintes du réseau formel.

Le résultat de la simulation apparaît sous forme graphique en Figure 28. Concrètement, il fait apparaître (comme nous pouvions nous y attendre lors d'une étude « manuelle » antérieure à la modélisation) que:

- Par rapport au programme fourni est entrée, l'ordre d'exécution des instructions a été modifié en:

```
100     LOAD    FPR0,A
103     ADD     FPR2,FPR3
101     ADD     FPR0,FPR1
104     MPY     FPR1,FPR2
102     ADD     FPR0,B
105     ST      FPR1,C
106     DIV     FPR1,FPR0
```

  Cette modification de l'ordre d'exécution des instructions est l'une des caractéristiques du data-flow: les instructions sont exécutées dès que les opérandes sont disponibles.
- Les transferts inutiles entre registres n'ont pas eu lieu: FPR0 n'a ni reçu la valeur de la variable A ni le résultat de l'instruction 101, et FPR1 n'a pas reçu le résultat de l'instruction 104 (propriétés du data-flow).
- Le nombre de cycles nécessaires à l'exécution du programme est inférieur à la somme du nombre de cycles nécessaires à l'exécution de chaque instruction (propriété du parallélisme).

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
décodeur d'instructions	LOAD FPR0 A	ADD FPR0 FPR1	ADD FPR0 B	ADD FPR2 FPR3	MPY FPR1 FPR2	ST C FPR1	DIV FPR1 FPR0						
unité mult/div (station)							8	8	8		9	9	9
unité add/soustr (station)					12	12	10	10	11	11			
disponibilité des résultats							12		10	8	11		
FPR0 (busy,tag)	0,-	1,4	1,10	1,11	1,11	1,11	1,11	1,11	1,11	1,11	0,-		
FPR1 (busy,tag)	0,-					1,8	1,8	1,9	1,9	1,9	1,9	1,9	1,9
FPR2 (busy,tag)	0,-				1,12	1,12	0,-						
FPR3 (busy,tag)	0,-												
SDR0 (tag)							8	8	8	-			
RS8L (tag)						0	0	0	0	-			
RS8R (tag)						12	0	0	0	-			
RS9L (tag)								8	8	0	0	0	0
RS9R (tag)								11	11	11	0	0	0
RS10L (tag)			4	4	4	0	0	0	-				
RS10R (tag)			0	0	0	0	0	0	-				
RS11L (tag)				10	10	10	10	10	0	0	-		
RS11R (tag)				3	0	0	0	0	0	0	-		
RS12L (tag)					0	0	-						
RS12R (tag)					0	0	-						

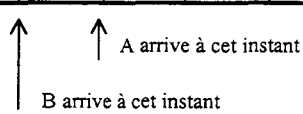


Figure 28: *IBM360* — Comportement dynamique du pipeline

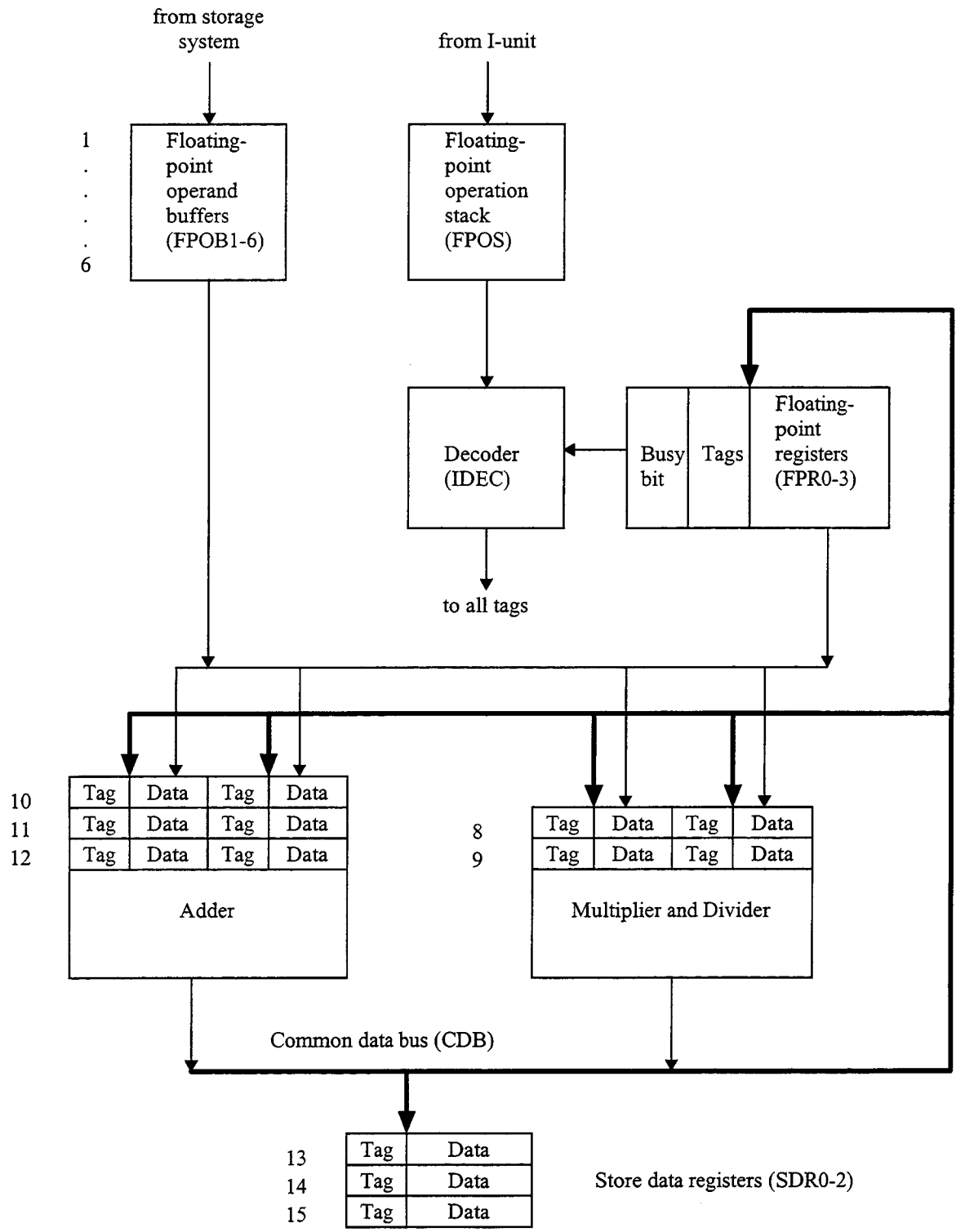


Figure 29: *IBM360* — Synoptique de l'unité d'exécution

III.4.6 Exemple 5: le Steam-Boiler

Le problème du *Steam-Boiler* présenté dans [ABR 94] consiste à concevoir un programme capable de superviser le fonctionnement d'une chaudière: la quantité d'eau contenue dans le réservoir de la chaudière ne doit être ni trop faible ni trop élevée, sinon la chaudière ou la turbine à vapeur pourraient être fortement endommagées. La Figure 30 représente schématiquement le système et son environnement.

Physiquement, le système comprend différentes unités: la chaudière, un équipement pour mesurer la quantité d'eau dans la chaudière, un jeu de quatre pompes pour alimenter la chaudière en eau, un jeu de quatre contrôleurs pour actionner (ouvrir, fermer) les pompes (un pour chaque pompe), un équipement pour mesurer la quantité de vapeur qui sort de la chaudière, une console pour l'opérateur, et un système de transmission de messages entre les différents constituants de l'ensemble.

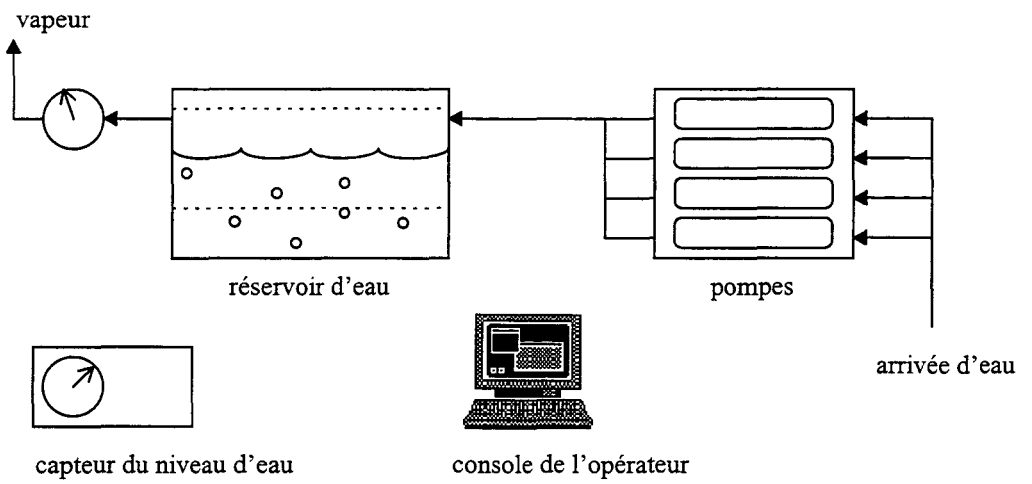


Figure 30: *STEAM-BOILER* — Représentation schématique de la chaudière

La chaudière est caractérisée par:

- une vanne d'évacuation, utilisée pour vidanger la chaudière lors de la phase d'initialisation du système.
- sa capacité C (unité: l).
- les quantités minimale M_1 et maximale M_2 d'eau (unité: l): au dessous de la quantité minimale, la chaudière est en danger de surchauffe, au dessus de la quantité maximale, la production de vapeur n'est pas suffisante pour compenser l'arrivée éventuelle d'eau délivrée par les pompes.
- les quantités normales minimale N_1 et maximale N_2 d'eau (unité: l), qui correspondent à une utilisation en régime continu. $M_1 < N_1$ et $M_2 > N_2$.
- les gradients d'augmentation U_1 et de diminution U_2 de la quantité de vapeur (unité: l/s^2).

Le capteur de niveau d'eau est caractérisé par:

- la quantié d'eau q (unité: l) dans la chaudière.

L'une des quatre pompes est caractérisée par:

- sa capacité P (unité: l/s).
- son mode de fonctionnement (*ouverte* ou *fermée*).

L'un des quatre contrôleurs de pompes est caractérisé par:

- l'état de la pompe associée: l'eau circule ou ne circule pas dans la pompe.

Le capteur de vapeur est caractérisé par:

- la quantité de vapeur v (unité: l/s) sortant de la chaudière.

Le fonctionnement du système est supervisé par un « programme » qui communique avec les différentes unités par l'intermédiaire de messages. En première approximation, le temps de transmission des messages est négligeable. Ce programme boucle indéfiniment et consiste à:

- recevoir les messages provenant des différentes unités.
- analyser les informations reçues et prendre une décision.
- envoyer de nouveaux messages vers les différentes unités.

Le programme opère en différents modes: *initialisation*, *normal*, *dégradé*, *secours*, et *urgence*. L'automate montrant le passage d'un mode à un autre est proposé à la Figure 31:

- mode initialisation: il correspond au démarrage du système. Après réception du message STEAM_BOILER_WAITING venant de toutes les unités, si la quantité de vapeur (message STEAM) est différente de 0 ou si le capteur de niveau d'eau (message LEVEL_FAIL_DET) est en panne, on passe en mode urgence, car le système ne peut pas fonctionner. Si toutes les unités sont prêtes (message PHYS_UNIT_RDY) et si une ou plusieurs autres unités sont en panne (messages PUMP_FAIL_DET, PUMP_CONTROL_FAIL_DET, STEAM_FAIL_DET), on passe en mode dégradé, où l'on attendra que les unités soient réparées. Sinon le mode normal est activé.
- mode normal: si le niveau d'eau devient critique (message LEVEL), la chaudière risque de surchauffer et on passe en mode urgence. Si l'unité de mesure d'eau est en panne (message LEVEL_FAIL_DET) on passe en mode secours où il sera impératif que la réparation ait lieu. Si une autre unité est en panne (messages PUMP_FAIL_DET, PUMP_CONTROL_FAIL_DET, STEAM_FAIL_DET), on passe en mode dégradé. Sinon on reste en mode normal.
- mode dégradé: il réagit de la même manière que le mode normal, mais si les unités en panne sont réparées (messages PUMP_REPAIRED, PUMP_CONTROL_REPAIRED, STEAM_REPAIRED), on repasse en mode normal.
- mode secours: les unités doivent être impérativement réparées pour repasser en mode normal ou en mode dégradé si l'unité de mesure d'eau est toujours en panne. Dans les autres cas, on passe en mode urgence.
- mode urgence: la chaudière ne peut plus fonctionner sans exploser. L'opérateur doit intervenir et commander l'arrêt du système.

D'autres messages existent, en particulier pour synchroniser les différentes opérations, mais ne sont pas ici nécessaires à la bonne compréhension du fonctionnement du système: par exemple le message LEVEL_FAIL_DET envoyé par l'unité de mesure d'eau doit être acquitté par le message LEVEL_FAIL_ACK envoyé par le programme de supervision. Ces messages sont définis plus précisément dans l'Annexe E.

III.4.6.1 Modélisation

Dans notre modélisation, nous avons décidé de rester le plus fidèle possible aux spécifications proposés par J.R. Abrial. La partie opératoire (programme de contrôle) du steam-boiler est donc implémentée ici sous forme d'un automate. En effet, d'après les spécifications, cinq états du mode de fonctionnement du système sont différenciables: initialisation, normal, dégradé, secours, et urgence. Le système passe d'un état à un autre sur réception d'événements introduits sous forme de messages. Cet automate est présenté à la Figure 31. La modélisation des différentes unités du système et leurs interactions par envoi de message est illustré par la Figure 32. Nous voyons apparaître une boîte à messages, seul mécanisme de communication entre les différentes unités. L'opérateur est ici l'unique interfaçage avec l'extérieur du système: il pourra recevoir des messages de la part du système (pannes, acquittements, ...), et produire lui-même des ordres (démarrage, reprise de fonctionnement après réparations, ...). Dans cet exemple nous avons modélisé chaque unité par une place.

Le graphe du réseau formel n'offre ici que peu d'intérêt: chaque place est principalement connectée à la boîte à messages. Nous avons préféré présenter le code source du programme *NetSPEC*, agrémenté de commentaires, en Annexe E. Chaque transition est chargée d'analyser un état particulier du système, et est spécifiée selon son ordre de priorité, qui est ici essentiel. Les transitions sont assemblées en quatre groupes: DISPATCH pour échanger des messages au plus bas niveau (priorité la plus élevée), UNITMSG pour mettre à jour les paramètres des messages, CONTROL pour faire passer l'automate d'un état à un autre, et ACTION pour émuler certaines actions du système automatiquement (priorité la plus faible). Chaque groupe est prioritaire sur le suivant et dans un groupe particulier, une transition est prioritaire sur la suivante, ce qui permet d'alléger l'écriture des contraintes: si une transition est franchissable, nous sommes assurés que les précédentes ne l'étaient pas.

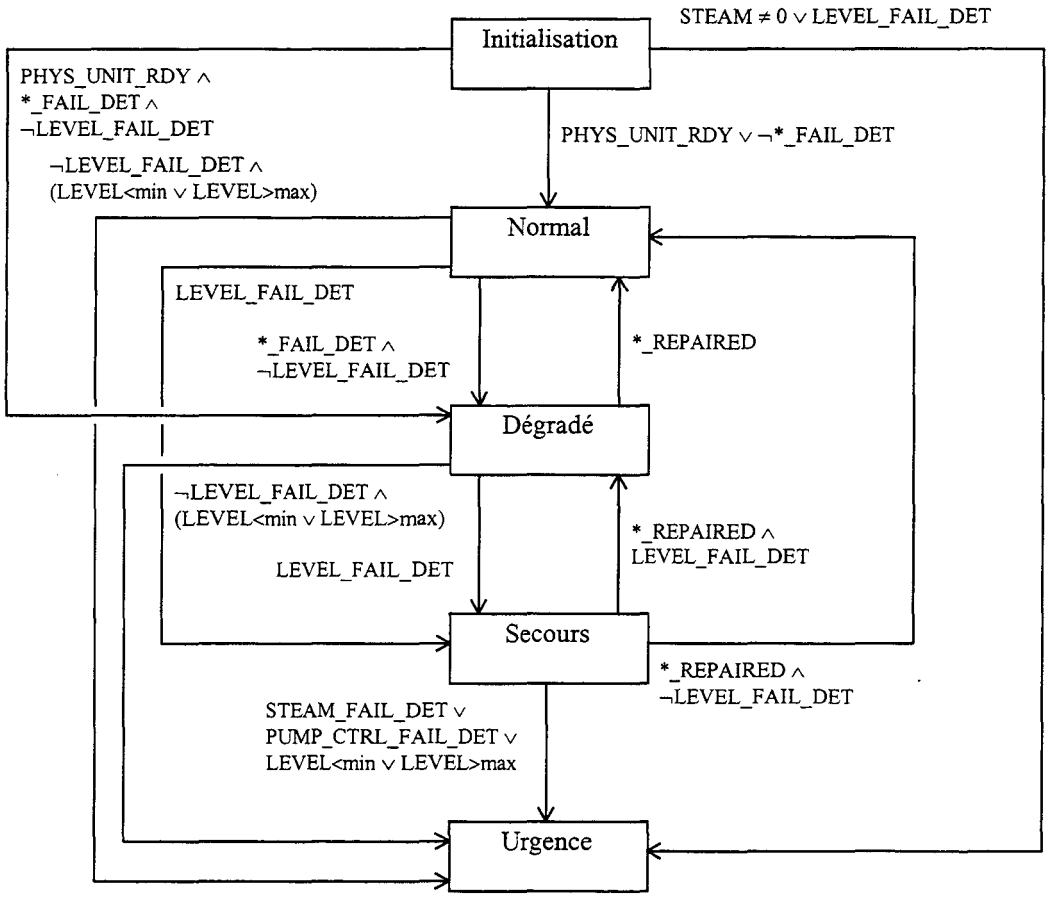


Figure 31: *STEAM-BOILER* — Automate du système

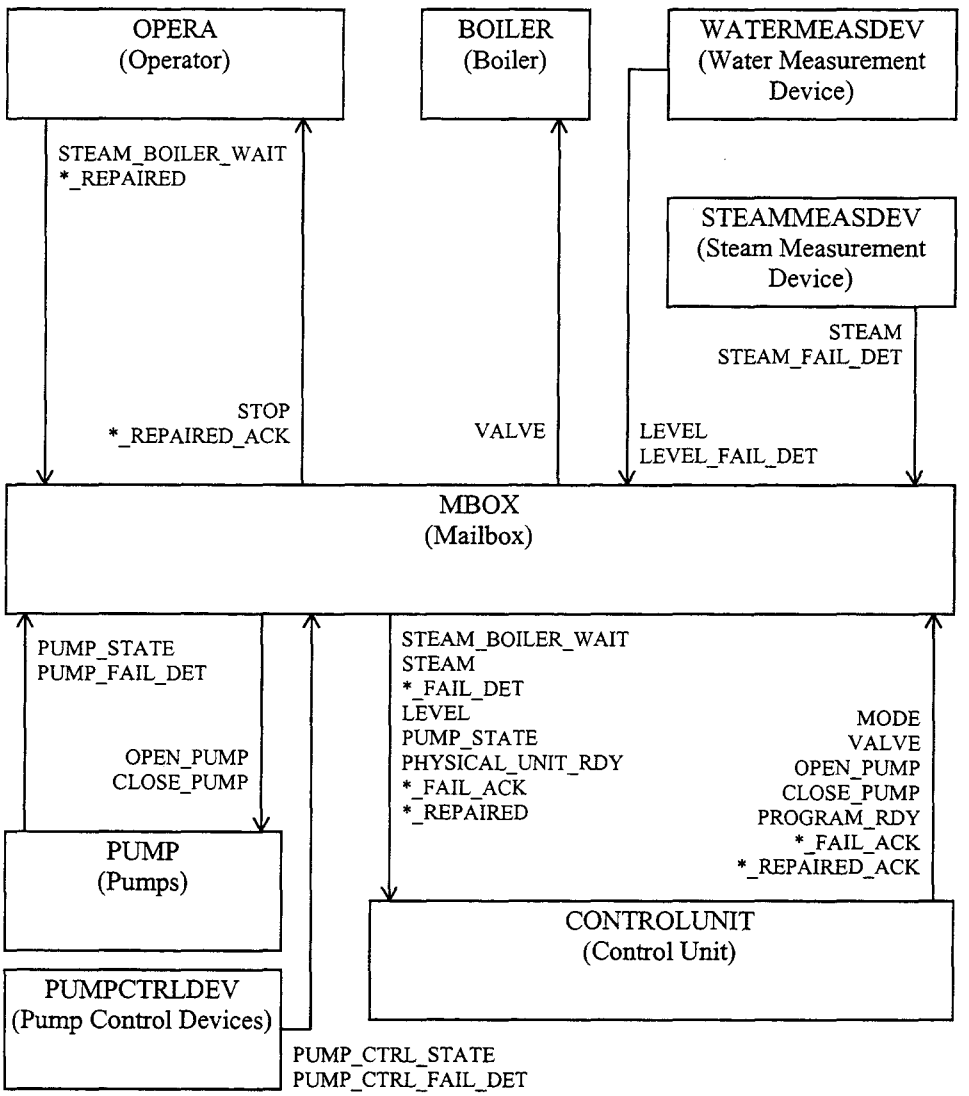


Figure 32: *STEAM-BOILER* — Interactions entre les différentes unités

III.4.6.2 Conclusion

Nous avons illustré, grâce à cet exemple, qu'un système de sécurité devant prendre des décisions sans intervention extérieure peut être implémenté par un réseau formel. L'originalité de cette modélisation repose sur l'emploi d'un automate et d'une structure d'échange de messages (ce n'est certainement pas la seule solution, ni même la meilleure). Cependant, notre solution fait fortement référence à la notion de priorités entre transitions, laquelle n'existait pas dans le modèle théorique, et qui sera introduite plus tard dans l'étude de *NetSPEC*.

III.4.7 Conclusion

En présentant ces cinq modélisations, nous avons montré la diversité des problèmes résolvables à l'aide des réseaux formels. En effet, si *GENCOD* est un cas d'école, les autres exemples appartiennent à des catégories bien connues: le traitement d'information (*IFIP*), la planification (*CUBES*), la simulation (*IBM360*), et le contrôle de process (*STEAM-BOILER*).

Les modèles que nous avons présentés peuvent être considérés comme des *spécifications formelles* des exemples. En effet, les annotations des schémas sont des formules logiques, et le comportement global du système est décrit par la structure du réseau. Aucune notion de programmation impérative n'intervient donc dans cette description.

On peut à ce stade s'interroger sur le passage de la phase de la *spécification* à celle de la *conception* effective d'une implémentation du système modélisé. Nous allons montrer dans le chapitre suivant que ce passage à la conception peut s'effectuer d'une manière totalement *automatique*. Notamment, il ne sera pas nécessaire d'introduire explicitement du code impératif.

L'idée de « transformer » les spécifications d'un réseau formel en programme exécutable destiné à observer le comportement dynamique d'un système nous a donc conduit à développer l'outil *NetSPEC*. Plus précisément, il s'agit maintenant de:

- Concevoir un outil permettant de faire le lien entre une modélisation basée sur un réseau formel et la partie opératoire d'un système d'information. L'emploi d'un SGBD devient alors un élément incontournable de la solution envisagée. Cet outil doit permettre de transformer les spécifications en un code impératif exécutable capable de dialoguer avec une base de données.
- Elaborer un langage de description d'un réseau formel utilisable comme langage de programmation du code source appliqué à cet outil.
- Concevoir une chaîne de développement complète permettant de développer toutes les étapes nécessaires au passage de la description d'un réseau formel à l'observation de son comportement (édition, compilation, mise au point, observation).
- Résoudre certains problèmes relatifs à la phase d'implémentation réelle d'une solution. Ces problèmes concernent principalement l'ordonnancement des tirs des transitions, des priorités relatives des transitions, et les stratégies utilisées pour déterminer le franchissement des transitions (évaluations des contraintes).

Ces différents points sont abordés dans le chapitre suivant, où nous décrivons l'outil *NetSPEC* en détail.

CHAPITRE IV

UNE ALTERNATIVE AUX ADBMS: UTILISATION DES RESEAUX FORMELS

Les chapitres précédents nous ont permis d'introduire les différents éléments nécessaires à la création d'un système de développement utilisable dans le cadre de la conception d'un réseau formel formant le coeur de la partie opératoire d'un système d'information.

A partir des spécifications en langage naturel d'un problème particulier, présentées sous forme d'un cahier des charges, nous désirons ici modéliser une solution faisant intervenir un réseau formel, et concevoir un programme capable de retracer le comportement dynamique — ou réactif — de ce dernier: contenu des places et valeurs des attributs des jetons, évaluations des contraintes de transition, de place, de marquage, et franchissements des transitions avec consommations et productions de jetons.

Bien qu'un tel programme puisse être écrit par le concepteur du réseau formel lui-même — cas de la rétroconception par exemple [CAD 97] — il nous a semblé intéressant de concevoir un système de développement organisé autour d'un outil de génération automatique de code, appelé *NetSPEC*, nous permettant ainsi de consacrer la majeure partie du temps à la conception du réseau formel et nous affranchissant de possibles problèmes d'implémentation.

Afin d'atteindre un degré élevé d'abstraction et d'obtenir le comportement attendu des réseaux formels, nous nous contraignons à ne définir notre réseau formel qu'en termes de places, de transitions, et de contraintes, laissant à *NetSPEC* le soin de compiler le réseau formel en langage impératif. Notre travail s'inscrivant dans l'objectif clair d'obtenir la partie opératoire d'un système d'information, l'emploi d'une base de données comme support du réseau formel est un élément incontournable de la solution envisagée:

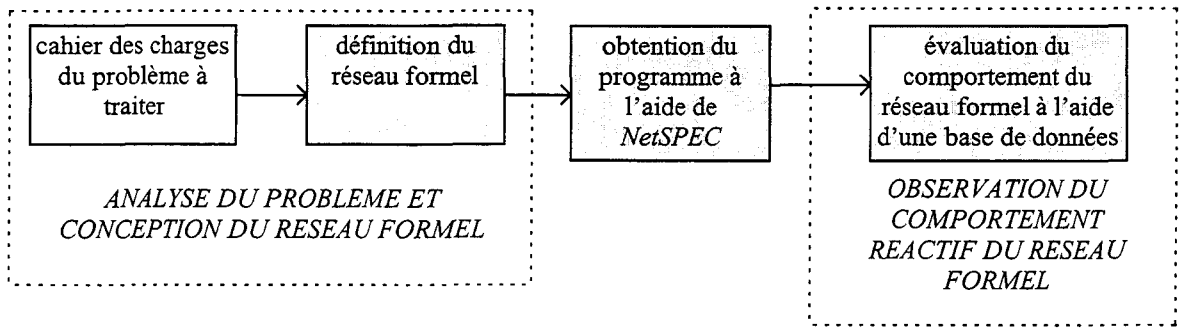


Figure 33: De l'analyse à l'observation

Les chapitres II et III nous ont montré que les réseaux formels et les ADBMS reposent sur des notions communes: dynamique du système définie en termes d'événements, de conditions, et d'actions, en vue de l'observation du comportement réactif de ce dernier. Les travaux antérieurs effectués sur les ADBMS par les précédents groupes de travail ne vont pas toujours dans le sens de l'abstraction désirée puisque le concepteur d'un système d'information organisé autour d'un ADBMS doit posséder une connaissance non négligeable de son système: programmation en C++ pour les ADBMS de type objet comme *SAMOS* ou en Smalltalk comme *HiPAC*, détail du fonctionnement des triggers comme dans *Ode* par exemple.

Les objectifs que nous nous fixons sont donc:

1) de concevoir un outil automatique permettant d'utiliser un réseau formel comme partie opératoire d'un système d'information (sous-système opérant).
2) de permettre à l'utilisateur d'observer le comportement réactif du réseau formel.
3) de conserver un degré d'abstraction élevé dans la définition du réseau formel.
4) de se rapprocher des concepts utilisés dans les ADBMS.
5) de proposer l'usage de cet outil à un utilisateur non-informaticien en rendant transparente la programmation.
6) de concevoir si possible cet outil à l'aide de produits standards.

IV.1 Mise en oeuvre du système NetSPEC

IV.1.1 Architecture du système

Afin de répondre aux différents objectifs cités en introduction, et dans l'optique de proposer une réalisation originale, l'architecture du système envisagé repose principalement sur l'utilisation d'un DBMS passif transformé en ADBMS à l'aide d'une couche logicielle supplémentaire (*Layered Approach* [CHA 92][GEP 95][FRI 97]):

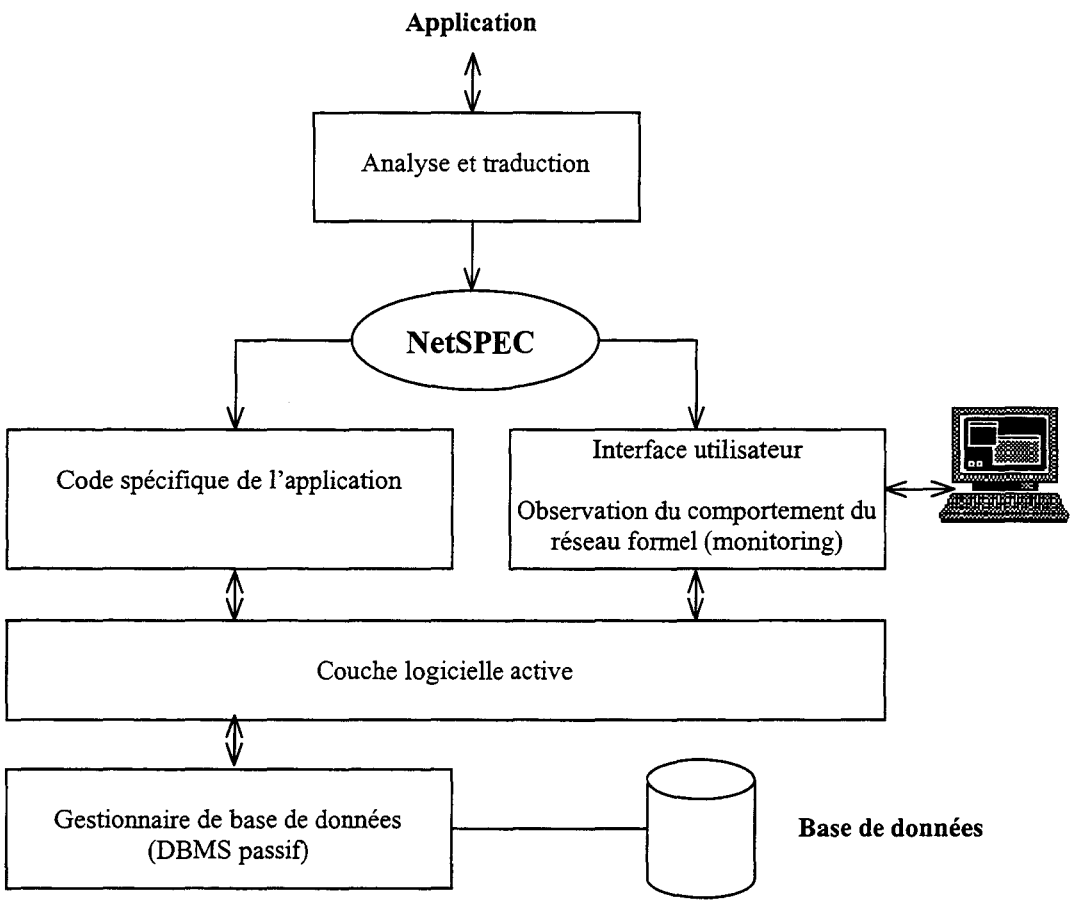


Figure 34: Architecture du système *NetSPEC*

Nous présenterons dans la suite ce chapitre les différents blocs fonctionnels de cette architecture, les choix qui ont été retenus, et les solutions proposées pour résoudre les différents problèmes inhérents aux bases de données actives.

IV.1.2 Analyse et traduction

Un réseau formel étant spécifié en termes de places, de marquages, et de contraintes à l'aide de schémas Z, il serait naturel de ne pas se démarquer de la notation Z. Cependant cette dernière est trop formelle pour pouvoir être utilisée telle quelle, et la nécessité de travailler à l'aide d'une notation plus suggestive apparaît rapidement. Cette nouvelle

notation a été introduite dans les chapitres précédents, notamment dans le but de simplifier la présentation des annotations des places et des transitions.

Moyennant quelques adaptations, cette notation constitue le germe d'un langage de programmation destiné à produire une entrée (code source) suffisamment précise à *NetSPEC* qui génère le programme final. En l'état actuel de nos travaux, ce code source est spécifié directement par le concepteur, mais nous pouvons envisager sa production automatique à l'aide d'autres outils, tel qu'un éditeur graphique, puisqu'un réseau formel peut aussi être construit graphiquement. Le langage de description textuelle proposé possède une syntaxe facilement compréhensible. Il respecte la puissance d'expression d'un réseau formel (annotations et décorations) et permet l'utilisation d'expressions complexes, pour la retranscription des contraintes notamment. La grammaire de ce langage est donnée en Annexe A (notation BNF et diagrammes syntaxiques [AHO 89]) et l'exemple du code source du cas *IFIP* étudié précédemment figure en Annexe C.

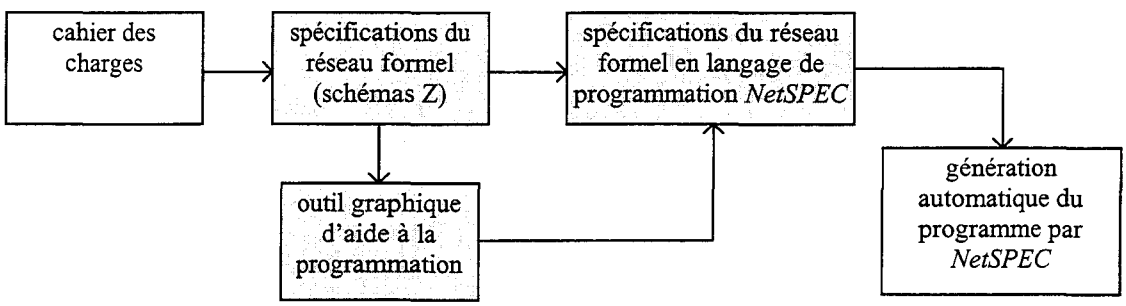


Figure 35: Spécification d'un réseau formel à l'aide d'un langage spécifique

IV.1.3 Génération automatique du programme

A partir du code source décrivant le réseau formel, le générateur de code de *NetSPEC* prend en charge la création de la base de données associée au réseau formel d'une part, et la génération du programme principal destiné, lors de son exécution, à permettre l'observation du comportement du réseau formel d'autre part (code spécifique à l'application, interface utilisateur, couche logicielle active).

A ce stade, le choix du modèle des données doit être clairement défini, car deux des objectifs énoncés plus haut dépendent de ce dernier:

- Mise en oeuvre indépendante d'un langage de programmation classique pour le concepteur (objectif 5).
- Utilisation de produits standards — DBMS passif et langage hôte utilisé par *NetSPEC* — assurant un portage aisé sur différentes plates-formes (objectif 6).

IV.1.3.1 Modèle des données et DBMS passif

A l'heure actuelle, deux standards de modèles des données ont émergé: le modèle relationnel et le modèle orienté objet (Cf. chapitre I). Cependant, le modèle orienté objet ne semble pas en mesure de satisfaire l'objectif 5 puisque les opérations sur la base de données sont décrites essentiellement en terme de méthodes, et nous voyons mal comment soustraire totalement le concepteur de la définition de ces dernières à l'aide d'un langage de programmation classique (généralement C++). L'étude approfondie des ADBMS orientés objet *HiPAC*, *Ode*, et *SAMOS* nous en apporte la preuve (mais leurs objectifs ne sont pas semblables aux nôtres).

Par contre, le modèle relationnel est relativement bien adapté à la réalisation de nos objectifs:

- Un jeton est un tuple, tout comme une rangée d'une relation.
- L'utilisation du langage SQL (Structured Query Language [IBM 91]) comme vecteur d'interrogation de la base de données peut être considérée comme une facilité d'extraction des données. La transformation de la définition d'un réseau formel (définition des places et des contraintes) en commandes ou requêtes SQL semble à priori aisée et peut donc être effectuée de manière automatique, sans l'intervention du concepteur. Finalement, SQL s'interface facilement avec un langage hôte (par exemple C [ANS 89] ou COBOL [COD 78]).
- De nombreux DBMS passifs de type relationnel existent (e.g. DB2 [DAT 89], INGRES [DAT 87] ou ORACLE [CRO 88]) sur différentes plates-formes, du micro au mainframe.

IV.1.3.2 Chaîne de développement

La chaîne de développement de *NetSPEC*[4], illustrée à la Figure 36, est composée:

- Du générateur de code produisant trois programmes. Le premier programme est le résultat de la transformation des spécifications du réseau formel en un programme source spécifique à l'application, et prenant en charge les contraintes de place, les contraintes de transition, et les contraintes de marquage (code mixant le langage hôte C et le langage SQL). Le deuxième programme prend en charge l'interface du système avec l'utilisateur, et permet à ce dernier de contrôler le fonctionnement du réseau formel lors de l'observation de son comportement, mais aussi de modifier et de consulter le contenu des places (code mixant le langage hôte C et le langage SQL). Le troisième et dernier programme est chargé de la création de la base de données et de la compilation des deux programmes précédents (code en langage de commande REXX — Restructured Extended Executor Language [IBM 88]).
- Du préprocesseur SQL du DBMS passif utilisé, qui transforme les différentes requêtes SQL des programmes précédents en appels systèmes à destination du DBMS.

[4] Pour des raisons de disponibilité, le développement initial de *NetSPEC* a été effectué sur une plate-forme IBM OS/2, à l'aide du DBMS passif DB2/2.

- Du compilateur C et de l'éditeur de liens permettant d'obtenir un exécutable du programme complet.
- D'une bibliothèque spécifique à *NetSPEC* (runtime) incluant diverses fonctions utilitaires nécessaires au bon fonctionnement de l'ensemble. Cette bibliothèque, indépendante de l'application, constitue en majeure partie la couche active apportée au DBMS passif pour le transformer en ADBMS.

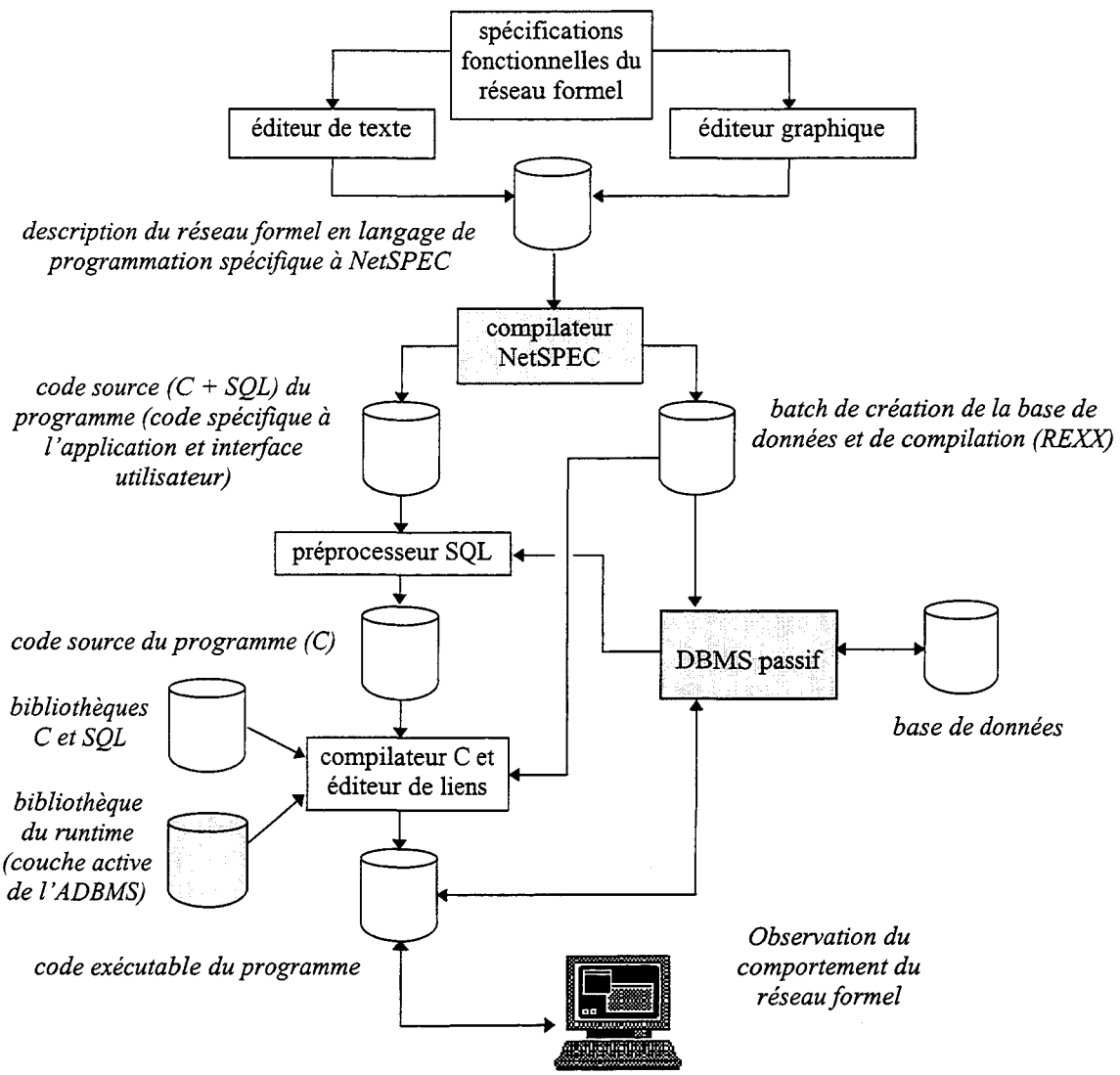


Figure 36: Chaîne de développement du système *NetSPEC*

IV.1.4 Spécifications fonctionnelles

IV.1.4.1 Création de la base de données

Le fichier batch produit par le générateur de code de *NetSPEC*, écrit en langage REXX [IBM 88], prend en charge la création de la base de données associée au réseau formel. Cette création englobe la création de la base de données proprement dite et la création des tables de la base de données.

A chaque place du réseau formel correspond une table particulière, dont chaque rangée (i.e. enregistrement) contient les informations relatives à un jeton. Une rangée contient plusieurs colonnes, chacune d'entre elles représentant un attribut particulier d'un jeton de la place. La possibilité de modéliser des attributs prenant la valeur nulle est également prise en compte lors de la création des tables. La Figure 37 illustre la traduction du schéma Z d'une place (place *projet* du cas *IFIP*) en table de la base de données.

projet

réf : ℕ
nom_auteur : 𝕊
date_réception : date
version : 𝕊

---

version ∈ {'lettre', 'provisoire'}

Table *projet*

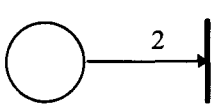
Colonne 1	Colonne 2	Colonne 3	Colonne 4
réf (entier)	*nom auteur* (chaîne)	*date réception* (date)	*version* (chaîne)

Figure 37: Traduction du schéma Z d'une place en table

Il est à noter qu'aucune référence aux contraintes de clé ou de place n'est présente dans la modélisation de la place. Ces contraintes sont intégralement prises en charge par le programme spécifique à l'application[5], afin d'éviter de laisser la gestion des doublons à la discrétion du gestionnaire de la base de données, ce qui pourrait rendre plus délicat les traitements à effectuer en cas de violation de contrainte (contrainte de clé notamment) lors de l'ajout d'une rangée dans la table (production d'un nouveau jeton).

Cependant, *NetSPEC* ajoute automatiquement une colonne supplémentaire dans la table: la clé primaire, dont le rôle est de pouvoir différencier deux jetons ne possédant pas de contrainte de clé, ayant les mêmes valeurs d'attributs, et ne pouvant donc pas être distinguables, comme le montre l'exemple suivant:

Soit:



place P: jeton <x>
avec x ∈ ℕ

transition T:
contrainte $P^{(1)}.x > 0 \wedge P^{(2)}.x > 0$

Un jeton ne pouvant être consommé plus d'une fois, il vient que le jeton $P^{(1)}$ ne peut être que différent du jeton $P^{(2)}$ (bien que leurs attributs respectifs puissent être égaux). Une colonne supplémentaire dans la table (clé primaire) permettra de lever l'ambiguïté du choix, en émettant l'hypothèse implicite que la contrainte soit augmentée de la clause $P^{(1)}.clé\_primaire$

[5] Ce point sera notamment abordé dans le paragraphe détaillant la couche logicielle active.

$\neq P^{(2)}.clé\_primaire$. Depuis la table associée à la place P, représentée avec 3 jetons, le seul couple de jetons vérifiant la contrainte est le couple (⟨2⟩,⟨3⟩) ou (⟨3⟩,⟨2⟩).

Colonne 1	Colonne 2
clé_primaire (entier)	attribut *x* (entier)
1	-1
5	2
100	3

Figure 38: Ajout et utilisation d'une clé primaire

IV.1.4.2 Interface homme/machine

IV.1.4.2.1 Dialogue homme/machine

Son rôle est principalement de permettre à l'utilisateur d'inscrire dans la base de données un marquage initial (avant le premier tir ou entre deux tirs successifs). En l'absence d'interface de type graphique, sa présentation est des plus sommaires, soit une simple liste de commandes destinées à mettre fin au programme, à déclencher et à arrêter un tir, et à produire manuellement des jetons dans les places du réseau. L'annexe B décrit plus précisément cet interface.

IV.1.4.2.2 Mise en oeuvre d'un marquage initial

La mise en oeuvre d'un marquage initial consiste en l'adjonction d'un ou de plusieurs nouveaux jetons dans une ou plusieurs places du réseau formel. La place et les attributs d'un jeton étant obtenus par l'intermédiaire de l'IHM, l'opération consiste uniquement à vérifier si les attributs valident les contraintes de place et les contraintes de clé si elles existent, et à insérer le nouveau jeton dans la place.

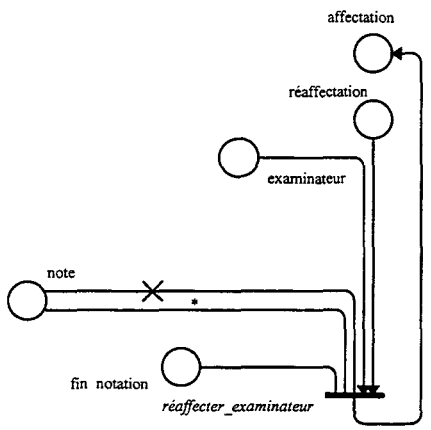
La vérification des contraintes de place est implémentée directement dans le code du programme, sans faire intervenir la base de données. En cas de violation de contrainte, un message d'erreur est affiché.

La vérification des contraintes de clé est implémentée en interrogeant la base de données sur l'existence possible d'un jeton ayant les mêmes valeurs d'attributs (uniquement les attributs soumis aux contraintes de clé) que ceux du nouveau jeton. Si cette existence est validée, un message d'erreur est affiché et le nouveau jeton n'est pas inséré dans la base.

Si toutes les contraintes sont vérifiées, l'insertion du nouveau jeton est réalisée en ajoutant une nouvelle rangée à la table associée à la place.

IV.1.4.3 Couche logicielle active

Parce que nous avons décidé d'utiliser un DBMS passif standard dans notre système, la partie « active » de l'ADBMS doit être construite à l'aide d'une sur-couche logicielle [CHA 92], essentiellement dédiée à un traitement comparable à celui des *règles-ECA* des ADBMS habituels. Les différents aspects de ce traitement — événements-conditions-actions, attachements, modes de couplages, priorités des règles — seront développés dans ce paragraphe en introduisant un parallèle avec les réseaux formels. Nous nous baserons sur le cas *IFIP*, et en particulier sur la transition *réaffecter_examinateur*, pour étayer la discussion de quelques exemples:



Les contraintes de la transition *réaffecter_examinateur* sont:

$fin\_notation^{?}.état = 'oui'$
$examinateur^{-}.réf = note^{*?}.réf$
$CARD(note^{*?}) < 3$
$examinateur^{-}.réf = note^{\sim}.réf$
$examinateur^{-}.nom = note^{\sim}.nom\_examinateur$
$réaffectation^{-}.réf = examinateur^{-}.réf$
$réaffectation^{-}.nom\_ancien\_examinateur = examinateur^{-}.nom$
$affectation^{+}.réf = réaffectation^{-}.réf$
$affectation^{+}.nom\_examinateur = réaffectation^{-}.nom\_examinateur$

Figure 39: Contraintes de la transition *réaffecter_examinateur* du cas *IFIP*

Rappelons que dans un ADBMS, les règles-ECA (Event-Condition-Action) qui déterminent le comportement réactif du système, consistent en des événements, des conditions, et des actions, et que leur sens est:

> *« Lorsqu'un événement arrive, tester une condition et si cette condition est satisfaite, exécuter une action »*

Les réseaux formels s'expriment quant à eux, en termes de contraintes de marquage — contraintes de place, contraintes de clé — et de contraintes de transition. Il s'agit donc ici de déterminer comment exprimer les trois composantes d'une règle-ECA à l'aide de ces contraintes.

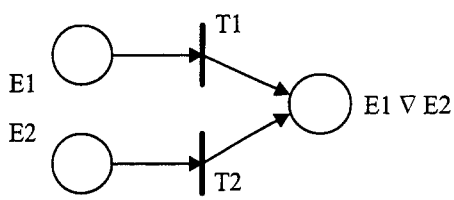
IV.1.4.3.1 La composante Evénement

En dehors du fait qu'un événement puisse être primaire ou composé, nous pouvons affirmer qu'il est essentiellement relatif au marquage du réseau: l'existence ou l'absence d'un jeton dans une place peuvent être considérée comme un événement, les valeurs données aux attributs d'un jeton également. Plus précisément, si nous désirons faire

intervenir le temps, l'instant d'occurrence de la production ou de la consommation d'un jeton référence l'événement.

Cependant, l'une des propriétés d'un réseau de Petri — une transition franchissable sera ou non franchie immédiatement, i.e. il n'y a aucune obligation de franchissement, ce n'est qu'une possibilité [DAV 92] — ne favorise pas la prise en compte instantanée d'un événement, et par conséquent une règle-ECA se rapproche donc plus d'une règle de production classique (règle uniquement basée sur une condition ou *pattern-based*). Cet aspect n'étant toutefois pas en opposition de la définition d'un ADBMS — *Ode* par exemple ne supporte que les règles *pattern-based* — nous avons décidé de ne pas le prendre en compte et de ne pas implémenter les règles de type *event-based*.

Toutefois, la notion d'événement persiste, principalement celle de leur composition: il est aisé de modéliser la disjonction, la conjonction, et la séquence d'événements primaires ou composés à l'aide d'un réseau formel. Le chapitre II a décrit la détection des événements composés à l'aide des réseaux de Petri classiques dans *SAMOS*: nous remarquerons en particulier que la séquence est modélisée plus simplement avec les réseaux formels, étant donnée la puissance que nous apporte l'expression des contraintes:

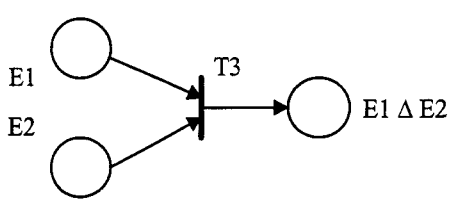


Disjonction: l'événement composé (E1 ∇ E2) arrive dès que E1 arrive ou que E2 arrive.

T1: $(E1 \nabla E2)^{+}.t = E1^{-}.t$

T2: $(E1 \nabla E2)^{+}.t = E2^{-}.t$

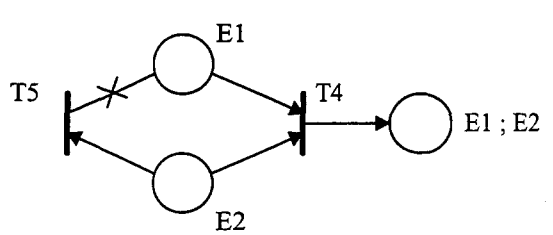
avec *t* instant d'occurrence



Conjonction: l'événement composé (E1 Δ E2) arrive si et seulement si les 2 événements E1 et E2 sont déjà arrivés; l'instant d'occurrence de l'événement composé est déterminé par l'instant d'occurrence du dernier événement le composant (chronologiquement).

T3: $(E1 \Delta E2)^{+}.t = MAX(E1^{-}.t, E2^{-}.t)$

avec *t* instant d'occurrence



Séquence: l'événement composé (E1 ; E2) arrive si l'un des événements le composant (ici E2) arrive après l'autre (E1); la transition puits T5 sert à éliminer le cas pour lequel les événements participant à la détection de l'événement composé arrivent dans le mauvais ordre (E2 avant E1).

T4: $(E1 ; E2)^{+}.t = MAX(E1^{-}.t, E2^{-}.t)$

T5: $E1^{-}.t \leq E2^{-}.t$

avec *t* instant d'occurrence

Figure 40: Détection des événements composés dans un réseau formel

IV.1.4.3.2 La composante Condition

La condition d'une règle-ECA spécifie un prédicat ou une requête sur la base de données. Dans une contrainte de transition, elle permet de faire le lien entre la transition considérée et les places situées en amont de celle-ci: selon le type d'arc reliant la transition aux places amont, le prédicat (et donc la condition) sera vérifié si la requête produit au moins une donnée (i.e. le résultat de la requête n'est pas vide) dans le cas d'un arc information ou consommation, ou si la requête ne produit pas de résultat dans le cas d'un arc information négative. Nous verrons cependant plus loin qu'une partie de la condition peut également porter sur la partie production d'un contrainte.

Pour la transition *réaffecter_examinateur*, la condition s'exprime par la conjonction de quatre prédicats:

$(fin\_notation^{?}.état = 'oui')$ # *la fin de notation est active*
$(examinateur^{-}.réf = note^{\sim}.réf \wedge examinateur^{-}.nom = note^{\sim}.nom\_examinateur)$ # *un examinateur n'a pas donné sa note*
$(réaffectation^{-}.réf = examinateur^{-}.réf \wedge réaffectation^{-}.nom\_ancien\_examinateur = examinateur^{-}.nom)$ # *cet examinateur doit être réaffecté*
$(examinateur^{-}.réf = note^{*?}.réf \wedge CARD(note^{*?}) < 3)$ # *il y a moins de 3 notes pour le papier concernant l'examinateur*

ce qui donnera lieu à quatre requêtes sur la base (la première et la troisième doivent produire au moins une donnée, la deuxième ne doit produire aucune donnée, et la quatrième doit produire moins de trois données):

$\exists$ 1 jeton $\in$ *fin_notation* | *état* = 'oui'
$\exists\neg$ jeton $\in$ *examinateur* | *réf* = *note.réf* $\wedge$ *nom_examinateur* = *note.nom_examinateur*
$\exists!$ jeton $\in$ *réaffectation* | *réf* = *réaffectation.réf* $\wedge$ *nom_examinateur* = *réaffectation.nom_examinateur*
$\exists^{+}$ 2 jetons $\in$ *note* | *réf* = *examinateur.réf*

IV.1.4.3.3 La composante Action

Les opérations à effectuer relativement à la partie action d'une règle-ECA sont la consommation depuis les places en amont de la transition et la production dans les places en aval de la transition. La consommation des variables apparaissant sur les arcs de type consommation arrivant à la transition est implicite, puisque déjà exprimée dans la condition, et correspond à une requête de destruction des jetons concernés. La production de nouveaux jetons consiste à calculer les attributs de ces jetons et à effectuer une requête d'insertion dans la place adéquate.

Pour la transition *réaffecter_examinateur*, un jeton de la place *examinateur* et un jeton de la place *réaffectation* seront consommés (i.e. détruits). Ces jetons sont uniquement ceux qui vérifient la condition exprimée plus haut. La partie production proprement dite de l'action concerne le calcul des attributs d'un nouveau jeton dans la place *affectation*, soit:

affectation+.réf = réaffectation-.réf
affectation+.nom_examinateur = réaffectation-.nom_examinateur

Nous nous apercevons immédiatement qu'il est possible, vu la forme d'une contrainte de production, de satisfaire l'attachement — ou lien — entre les variables utilisées dans la condition et les valeurs calculées dans l'action: ici, cet attachement s'exerce au niveau des attributs *réaffectation.réf* et *réaffectation.nom_examinateur* issus de la condition.

IV.1.4.3.4 Algorithme de traitement des contraintes de transition

La couche logicielle active, qui permet de transformer le DBMS passif en ADBMS, est essentiellement constituée du moteur de tir. Ce dernier, invoqué par l'une des commandes de l'IHM, contrôle seul le franchissement des transitions lorsqu'un tir est déclenché. Partant d'un marquage initial du réseau formel, il procède à l'évaluation du franchissement des transitions (i.e. résolution des contraintes), et le cas échéant, à la modification du marquage du réseau formel. Lorsqu'aucune transition ne peut plus être franchie, le marquage est considéré comme stable, puisqu'il n'y a plus de possibilité d'évolution. Les propriétés d'un réseau formel concernant le franchissement d'une transition sont:

- Une transition est franchissable si chaque place en amont de la transition possède un ensemble de jetons respectant les contraintes associées. La cardinalité de cet ensemble doit être égale à la pondération de l'arc s'il s'agit d'un arc information ou consommation, ou à 0 s'il s'agit d'un arc information négative. Dans le cas particulier où la pondération de l'arc est *, l'ensemble, qui peut être vide, peut également être soumis à une contrainte de cardinalité.
- Une transition est franchissable s'il est possible de produire un ou plusieurs jetons dans les places en aval de la transition (si un ou plusieurs arcs production existent). Les jetons produits peuvent éventuellement être soumis à des contraintes de clé, et à des contraintes de place.
- Il n'y a pas à priori d'obligation de franchir une transition dès que cette dernière est franchissable. Cependant pour obtenir un marquage final stable du réseau formel, une transition franchissable devra être tirée à un moment ou à un autre, afin d'atteindre un stade de non-évolution.
- Les conflits au niveau des contraintes doivent être résolus lors de la définition des contraintes, relativement au problème de l'exclusion mutuelle des ressources partagées (ici les jetons). En effet, un jeton consommable par plusieurs transitions, ne peut être consommé plusieurs fois.
- Si l'on assure l'absence de conflit, une seule transition en aval d'une place peut être franchie à un instant donné, ce qui élimine le parallélisme des franchissements. Les franchissements ne peuvent donc plus faire l'objet que d'une séquentialité, ce qui nous amène à un corollaire de la propriété précédente, à savoir qu'un jeton consommable par plusieurs transitions peut être à l'origine de plusieurs marquages finaux différents du réseau formel, suivant qu'il est consommé par une transition ou par une autre.

La mise en oeuvre du moteur de tir s'appuie sur l'algorithme de traitement des contraintes de transition, qui peut être directement calqué sur l'algorithme de base *recognize-act cycle*:

évaluation initiale des prédicats
tant qu'un prédicat au moins est vrai
  résoudre les conflits (sélectionner une règle de production)
  exécuter l'action de la règle choisie
  ré-évaluer les prédicats
fin

qui est alors transformé en:

évaluation initiale des contraintes de transition
tant qu'une transition au moins est franchissable
  résoudre les conflits (sélectionner une transition)
  appliquer les modifications de marquage (consommations et/ou productions)
  ré-évaluer les contraintes de transition
fin

IV.1.4.3.5 Résolution des conflits - priorité des transitions

En se basant sur les propriétés concernant le franchissement des transitions dans un réseau formel, principalement séquentialité et résolution des conflits, il apparaît une notion de *priorité* entre les transitions, notion déjà abordée lors de l'étude des ADBMS. En effet, la séquentialité implique l'évaluation des contraintes de transition dans un ordre préétabli (ordre partiel), ce qui entraîne une résolution toute naturelle des conflits, puisque la consommation d'un jeton par une transition rendra impossible le franchissement d'une autre transition dans laquelle le jeton aurait également pu être impliqué.

Les paragraphes suivants décrivent quelques méthodes envisageables pour implémenter une priorité [CRO 75] aux franchissements des transitions.

IV.1.4.3.5.1 Evaluation avec priorité aléatoire

L'hypothèse est ici de donner une priorité égale à toutes les transitions, leur offrant ainsi la même chance d'être tirées. L'évaluation aléatoire des contraintes de transition nous impose de définir d'abord une fonction aléatoire, de distribution uniforme, et si possible non cyclique, ce qui est relativement aisé: par exemple, soit la suite $X_{n+1} = aX_n$ *modulo* c, avec a et c entiers positifs, alors $U_n = X_n / c$ est distribué uniformément dans l'intervalle [0,1[, et pour éviter un cycle dans une période plus petite que c, a doit être choisi premier relativement à c [KLO 92].

Cependant, cette méthode ne fait qu'associer l'évaluation des contraintes d'une transition à un nombre aléatoire, non pas son franchissement effectif, puisque l'évaluation peut aboutir à un échec, ce qui peut être le cas le plus courant dans un problème particulier. Si théoriquement l'évaluation des contraintes de transition a une durée nulle, il n'en est pas de même pratiquement, et nous pouvons aboutir à des durées de tir différentes, voire prohibitives, comme le montre l'exemple suivant:

Considérons 4 transitions, associées respectivement aux 4 entiers {1,2,3,4}, et un générateur de nombres aléatoires de distribution uniforme dans l'intervalle [1..4]. Les deux graphes de gauche représentent deux distributions de nombres aléatoires, ceux de droite les franchissements effectifs des transitions, en émettant l'hypothèse que la transition 1 est franchissable 2 fois et que la transition 4 est franchissable 1 fois.

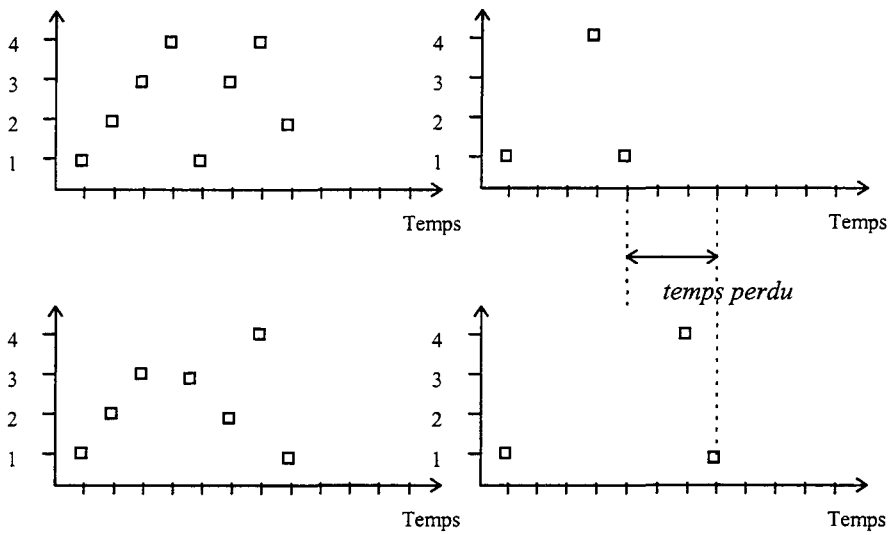


Figure 41: Priorité aléatoire

IV.1.4.3.5.2 Evaluation avec priorité fixe

L'évaluation des contraintes de transition est effectuée dans un ordre préétabli et inaltérable. Si l'on considère les 4 transitions de l'exemple précédent, et en supposant que T1 possède la priorité maximale et T4 la priorité minimale, l'évaluation et le franchissement seront ordonnés selon la Figure 42:

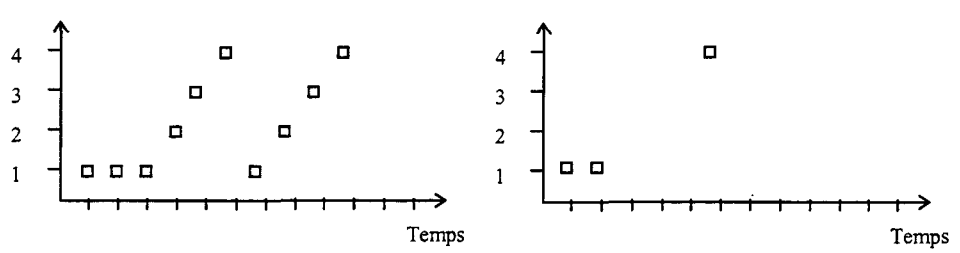


Figure 42: Priorité fixe

L'inconvénient majeur de ce type de gestion est qu'une transition de forte priorité, constamment franchissable, interdit le franchissement effectif de toutes les transitions de priorités inférieures. Cet inconvénient peut toutefois devenir un avantage si l'on se place dans le cas particulier d'une transition ne devant être franchie que lorsqu'une autre transition ne l'est plus, et ce, sans avoir à spécifier une quelconque exclusivité dans les deux contraintes de transition. Néanmoins cette méthode semble être une solution correcte pour de nombreux problèmes.

IV.1.4.3.5.3 Evaluation avec priorité circulaire

Cette méthode d'évaluation permet de mettre en oeuvre une priorité égale entre les transitions: une transition venant d'être franchie ne le sera pas une seconde fois immédiatement si une autre transition est franchissable. Par conséquent, une transition franchie se verra assigner la priorité la plus faible. Nous sommes assurés dans ce cas que chaque transition sera tirée en un temps fini:

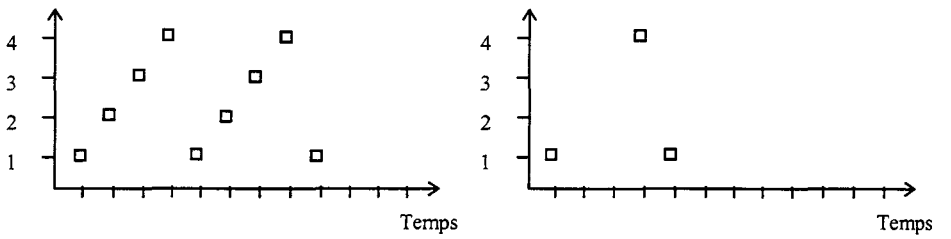


Figure 43: Priorité circulaire

Cette méthode possède également un inconvénient: si certaines transitions (en particulier les dernières à être évaluées) dépendent fortement du franchissement d'autres transitions, il peut apparaître une perte de temps non négligeable, puisqu'il y aura beaucoup d'échecs.

IV.1.4.3.5.4 Evaluation avec priorité itérative

Cette méthode d'évaluation privilégie le franchissement de la transition la plus récemment utilisée. Dans le cas le plus couramment rencontré où il existe un arc consommation en amont de la transition, l'effet obtenu est de consommer tous les jetons des places amont avant de passer à une autre transition, comme l'illustre la Figure 44.

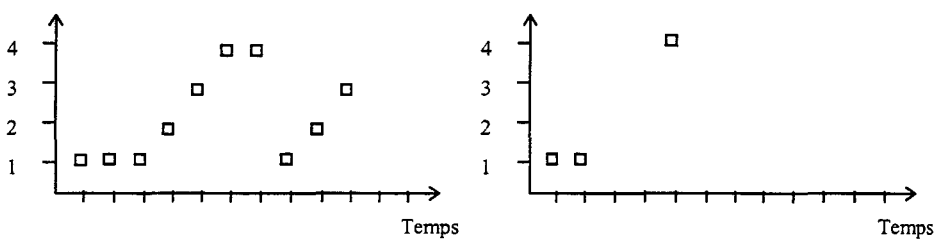


Figure 44: Priorité itérative

Cette méthode permet d'implémenter facilement le tir orienté par les ensembles.

IV.1.4.3.5.5 Choix d'une priorité en fonction du problème à résoudre

Etant donné que chacun des types de priorité définis plus haut possède des caractéristiques propres (avantages ou inconvénients), il serait intéressant de pouvoir formuler quelques règles de base en vue du choix le plus judicieux pour le problème à résoudre.

Cinq exemples de réseaux formels ont été développés, puis simulés en utilisant les trois priorités à étudier, respectivement priorité fixe, circulaire, et itérative. La priorité aléatoire a été volontairement écartée, n'étant pas jugée très satisfaisante (Cf Figure 41). Il a été procédé à un tir de chaque réseau formel à partir du même jeu de données (marquage initial), et il a bien entendu été vérifié que les marquages finaux étaient cohérents. Ces exemples ont déjà été décrits dans le chapitre précédent et nous avions alors fait allusion à leur simulation. Toutefois, l'exemple *GENCOD* n'est pas traité ici parce que trop peu significatif pour illustrer nos propos.

Exemple 1: Réseau formel à 4 transitions T1 à T4 dans lequel T4 est franchissable 1 fois dès que T1 a été franchie 2 fois (cas *SIMPLE*, dérivé de l'exemple précédent).

Exemple 2: Réseau formel modélisant le cas *IFIP* à 19 transitions (Annexe C). Ce réseau formel possède intrinsèquement un 'sens de parcours', i.e. un jeton du marquage initial (place *courrier*) se déplace dans le réseau jusqu'à atteindre une place précise du marquage final (place *session*).

Exemple 3: Réseau formel modélisant l'architecture data-flow de l'unité d'exécution en virgule flottante de l'*IBM 360* (Annexe D). Etant donnée l'architecture fortement pipeline, il est à priori extrêmement difficile de prévoir l'ordre de tir des transitions.

Exemple 4: Réseau formel modélisant le *STEAM-BOILER* (Annexe E) à 46 transitions. Ce réseau formel traite principalement de l'émission et de la réception de messages, avec prises de décision. Un marquage final n'est en théorie jamais atteint puisqu'il s'agit de modéliser un système de sécurité bouclant indéfiniment, mais la simulation a été arrêtée lorsque le système a cessé d'évoluer (d'après le jeu de données fourni).

Exemple 5: Réseau formel modélisant le problème des *CUBES* (Annexe F) à 4 transitions. Ce réseau formel comporte un indéterminisme certain, puisque plusieurs séquences d'actions sont possibles pour arriver à la configuration finale d'empilements.

Les résultats concernant le nombre d'évaluations de franchissement et le nombre de tirs effectifs sont regroupés dans le Tableau 9.

	SIMPLE	*IFIP*	*STEAM-BOILER*	*IBM360*	*CUBES*
priorité fixe	10 / 3	388 / 44	238 / 10	2450 / 129	70 / 25
priorité circulaire	12 / 3	207 / 44	189 / 10	785 / 129	35 / 25
priorité itérative	10 / 3	98 / 44	198 / 10	942 / 129	59 / 25

Tableau 9: Nombre d'évaluations de transitions / nombre de tirs effectifs

La Figure 45 est une représentation de la performance atteinte par les différents types de priorités, relativement à la performance de la priorité fixe (performance=1). Par exemple le cas *IFIP* demande 388 évaluations en priorité fixe, 207 en priorité circulaire soit un rapport (388 / 207) proche de 2, et 98 en priorité itérative soit un rapport (388 / 98) proche de 4. La performance de la priorité circulaire est donc double de celle de la priorité fixe et la performance de la priorité itérative est quadruple.

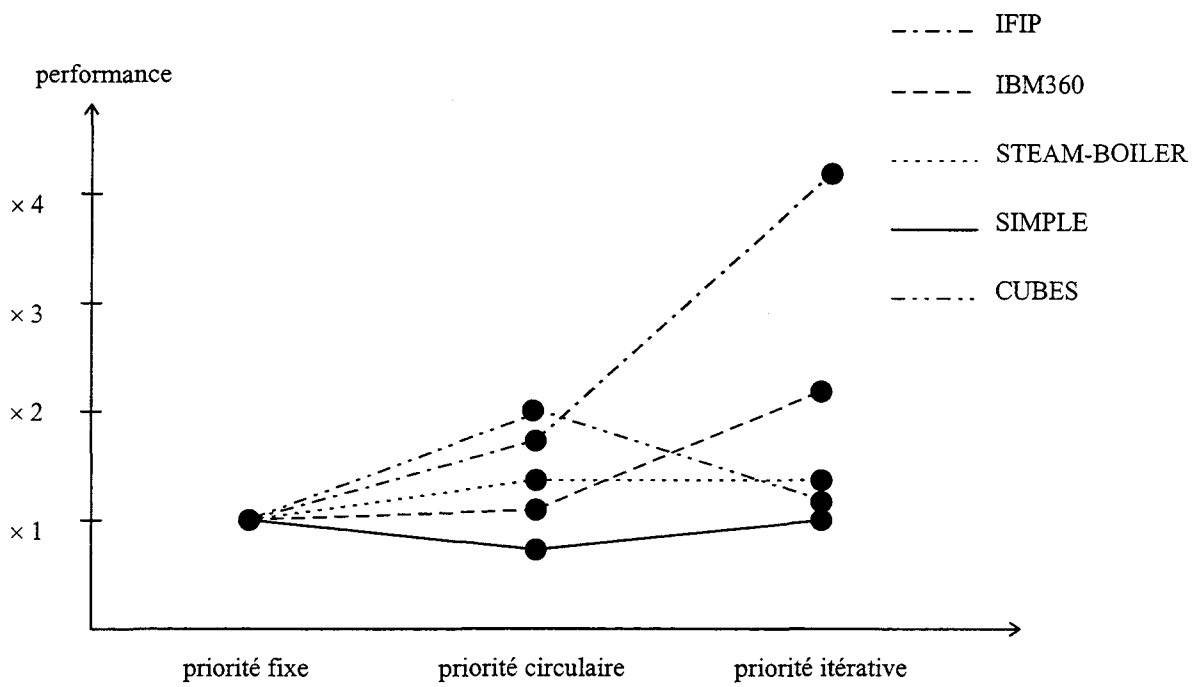


Figure 45: Performances suivant les types de priorités

Nous pouvons mentionner à partir des exemples précédents quelques conclusions empiriques, puisqu'uniquement basées sur l'observation.

La priorité fixe donne rarement de bonnes performances, ce qui semble correspondre au fait que les transitions des réseaux formels étudiés n'ont pas à priori de priorités implicites entre elles, sauf pour le cas *SIMPLE* dans lequel T1 est visiblement prioritaire sur T4.

La priorité circulaire donne des performances médiocres pour le cas *SIMPLE* (T4 ne peut pas être tirée immédiatement après le premier tir de T1), des performances moyennes pour le cas *IFIP* (un jeton empruntant un chemin bien précis dans le réseau formel, les transitions amont doivent être tirées avant les transitions aval), et de bonnes performances pour le cas *STEAM-BOILER* (une émission de message devant en principe être suivie par une réception) et pour le cas *IBM360* (il est impossible d'injecter trop de données dans un pipeline sans que celles-ci n'y circulent). D'excellentes performances sont trouvées pour le cas *CUBES* (dans la plupart des cas, il n'y a qu'une seule action possible qui doit donc être immédiatement réalisée).

La priorité itérative donne d'excellentes performances pour le cas *SIMPLE* (priorité implicite de T1 sur T4) et pour le cas *IFIP* (tir d'un maximum de transitions amont avant les transitions aval), de bonnes performances pour le cas *STEAM-BOILER* (émission de plusieurs messages avant la première réception), des performances moyennes pour le cas *IBM360* (la priorité est donnée à l'avancement des données dans le pipeline, mais sans garantie d'un fonctionnement optimal de ce dernier), et des performances peu intéressantes pour le cas CUBES (s'il n'est pas rare de pouvoir dépiler plusieurs cubes à la suite, l'inverse — empiler plusieurs cubes à la suite — n'est pas forcément vrai).

Les règles empiriques pouvant donc être appliquées au développement d'un réseau formel pourraient être:

- *règle 1:* si les transitions sont majoritairement prioritaires les unes par rapport aux autres, la priorité fixe s'impose.
- *règle 2:* si le réseau formel possède intrinsèquement un 'sens' de parcours, la priorité itérative pourra être envisagée.
- *règle 3:* si les transitions sont dépendantes les unes des autres, la priorité circulaire semble de loin la meilleure.

Pour ces raisons, il a été décidé d'implémenter les trois types de priorité dans le moteur de tir, laissant le choix de la priorité à l'utilisateur. Le pseudo-code du moteur de tir peut être présenté comme étant:

```
itérer ← 0
tant que itérer < nombre de transitions faire
    si une transition est franchissable (la recherche est effectuée par ordre de priorité décroissante)
            alors
                appliquer les modifications de marquage
                itérer ← 0
            sinon
                itérer ← itérer + 1
    fsi
    suivant la priorité choisie, appliquer les nouvelles priorités aux transitions
fin tant que
```

Figure 46: Pseudo-code du moteur de tir

IV.1.4.3.6 Transactions et modes de couplage

Les modes de couplage (couplage E-C et couplage C-A) déterminent la manière dont les événements, les conditions, et les actions interagissent avec les transactions. Dans notre cas, les événements — au sens d'une règle-ECA — n'étant pas exprimés dans les contraintes de transition, seul mode le couplage C-A possède un intérêt.

Nous avons décidé dans notre système de privilégier l'intégrité des données: à tout instant, nous voulons garantir que tous les objets sont dans un état consistant. Le mode de couplage C-A choisi est par conséquent du type immédiat. Le tir d'une transition n'est pas un tir orienté ensemble, même si l'évaluation de la condition produit plusieurs tuples: un seul tuple sera concerné dans un unique tir de la transition, comme dans un réseau de Petri classique. Dans le moteur de tir présenté à la Figure 46, la transaction débute au moment de l'évaluation de la partie condition de la contrainte de transition et prend fin après les modifications de marquage issues de la même transition.

IV.1.4.4 Programme spécifique à l'application

Le code spécifique à l'application prend en charge l'évaluation des contraintes de transition du réseau formel afin de déterminer si un franchissement de transition est possible. L'évaluation d'une contrainte comprend la partie amont de la transition (arcs information, information négative, consommation), la partie aval de la transition (arcs

production), ainsi que la partie relative au marquage (contraintes de clé et contraintes de place).

IV.1.4.4.1 Partie amont des contraintes de transition

L'évaluation des contraintes de transition en amont d'une transition consiste à construire un ensemble de jetons connus, vérifiant les contraintes les plus simples, puis à affiner cet ensemble au fur et à mesure que les jetons intervenant dans les contraintes plus complexes puissent être déterminés. Nous séparerons ces contraintes de transition en trois groupes:

- Les contraintes faisant intervenir des arcs de poids 1 à n, n étant fini.
- Les contraintes faisant intervenir des arcs de poids *.
- Les contraintes faisant intervenir la cardinalité d'ensembles de jetons.

Les paragraphes suivants décrivent les opérations à effectuer pour mener à bien l'évaluation des contraintes de transition pour les places en amont. La transition *réaffecter_examinateur* prise en exemple sera détaillée en se basant sur le marquage initial suivant:

fin_notation	état
	'oui'

examinateur	réf	nom
	1	'Jean'
	1	'Paul'
	1	'Pierre'
	2	'Jacques'
	2	'Louis'
	2	'Marcel'
	2	'Pierre'
	3	'Jacques'
	3	'Marcel'
	3	'Patrick'

affectation	réf	nom_examinateur

réaffectation	réf	nom_ancien_examinateur	nom_nouveau_examinateur
	1	'Paul'	'Jacques'
	2	'Jacques'	'Patrick'

note	réf	nom_examinateur	sujet	qualité	biblio
	1	'Jean'	10	10	8
	1	'Pierre'	9	10	9
	2	'Louis'	10	10	10
	2	'Marcel'	10	9	9
	2	'Pierre'	10	9	10
	3	'Jacques'	8	9	9
	3	'Marcel'	10	10	9

Exemple 14: Marquage initial du sous-système *réaffecter_examinateur*

IV.1.4.4.1.1 Evaluation des contraintes 1-n

Elle consiste en l'évaluation de toutes les contraintes pouvant être directement vérifiées: il s'agit des contraintes relatives aux arcs de type consommation, information, et information négative, ne faisant référence à aucune variable étoilée, mais pouvant faire référence à des variables pondérées autrement que par un poids égal à 1. Elle génère un ensemble de *m* tuples (jetons), *m* représentant ici le nombre de jetons intervenant dans les contraintes.

Pour la transition *réaffecter_examinateur*, les contraintes 1-n sont:

fin_notation$^{?}$*.état = 'oui'* # *la fin de notation est active*
examinateur^{-}*.réf = note*$^{\sim}$*.réf* # *un examinateur n'a pas donné sa note*
examinateur^{-}*.nom = note*$^{\sim}$*.nom_examinateur*
réaffectation^{-}*.réf = examinateur*$^{-}$*.réf* # *cet examinateur doit être réaffecté*
réaffectation^{-}*.nom_ancien_examinateur = examinateur*$^{-}$*.nom*

Ce qui peut se traduire par:

$\exists$ 1 jeton $\in$ *fin_notation* | *état* = 'oui'
$\exists\neg$ jeton $\in$ *examinateur* | *réf* = *note.réf* $\wedge$ *nom_examinateur* = *note.nom_examinateur*
$\exists!$ jeton $\in$ *réaffectation* | *réf* = *réaffectation.réf* $\wedge$ *nom_examinateur* = *réaffectation.nom_examinateur*

Le résultat de l'évaluation de ces contraintes sera la génération d'un ensemble d'examinateurs qui n'ont pas donné leur note. Cet ensemble peut être vide, mais pour que la transition soit franchissable, il doit contenir au moins un jeton (ici un examinateur).

Les deux tuples générés dans notre exemple seront:

fin_notation	**examinateur**		**réaffectation**		
état	***réf***	***nom***	***réf***	***ancien_nom_examinateur***	***nouveau_nom_examinateur***
'oui'	1	'Paul'	1	'Paul'	'Jacques'
'oui'	2	'Jacques'	2	'Jacques'	'Marcel'

Exemple 15: Résultat de l'évaluation des contraintes 1-n

IV.1.4.4.1.2 Evaluation des contraintes *

Elle consiste en l'évaluation des contraintes faisant intervenir des variables étoilées, transitant sur des arcs de type information et/ou consommation, mais ne faisant pas encore intervenir une possible cardinalité. Cette évaluation relie un nouvel ensemble de jetons à un tuple de jetons sélectionné précédemment.

Pour la transition *réaffecter_examinateur*, la contrainte * est:

examinateur-.réf = note$^{*?}$*.réf*

Ce qui se traduira par la construction d'un ensemble de jetons appartenant à la place *note* tel que *réf = examinateur.réf* , *examinateur.réf* appartenant lui-même à l'ensemble construit précédemment. Autrement dit, nous sélectionnerons toutes les notes du papier pour lequel l'examinateur n'a pas donné sa note.

Toutefois, il ne s'agit pas à strictement parler d'une évaluation de contrainte impliquant la possibilité ou l'impossibilité du franchissement de la transition, mais uniquement de la construction d'un ensemble de jetons, qui peut d'ailleurs être vide.

groupe	note				
	réf	*nom_examinateur*	*sujet*	*qualité*	*biblio*
1er	1	'Jean'	10	10	8
	1	'Pierre'	9	10	9
2nd	2	'Louis'	10	10	10
	2	'Marcel'	10	9	9
	2	'Pierre'	10	9	10

Exemple 16: Résultat de l'évaluation des contraintes *

IV.1.4.4.1.3 Evaluation des contraintes de cardinalité

Elle consiste en l'évaluation des contraintes faisant intervenir des cardinalités, opérant sur un ou plusieurs ensembles de jetons créés précédemment lors de l'évaluation des contraintes *.

Pour la transition *réaffecter_examinateur*, la contrainte de cardinalité est:

$$CARD\,(note^{*?}) < 3$$

Ce qui se traduira par l'obligation pour l'ensemble créé précédemment de posséder moins de trois notes pour un papier donné. Si tel est le cas la transition est franchissable.

Dans notre exemple, seul l'ensemble des notes dont la référence est égale à 1 satisfait la contrainte de cardinalité. Par conséquent, le seul examinateur pouvant être réaffecté est *'Paul'* pour la référence *1*.

IV.1.4.4.2 Partie aval des contraintes de transition

IV.1.4.4.2.1 Evaluation des contraintes de production

Elle concerne le calcul des attributs des jetons soumis aux contraintes de production, si elles existent (une transition peut être une transition puits). Elle est vraisemblablement l'étape la plus délicate à implémenter, car les contraintes de production peuvent présenter trois formes différentes, et doivent être évaluées dans un ordre précis:

- la forme de production dite *effective*
- la forme de production dite *conditionnelle*
- la forme de production dite *relationnelle*

Une forme de production *effective* est du type $P^{(n)+}.x$ = *expression calculable fonction des attributs de jetons issus d'arcs consommation, information, de poids 1 à n, y compris * si les attributs sont calculés à l'aide de fonctions comme la moyenne par*

exemple. Autrement dit, le terme de gauche $P^{(n)+}.x$ est le seul terme production de la contrainte et doit être inconnu, le terme de droite ne faisant intervenir que des attributs connus ou valeurs calculables, c'est-à-dire n'étant plus contraints. L'effet de l'évaluation de cette forme de production est d'assigner une valeur à l'attribut du jeton produit. Cette évaluation ne peut donc pas produire d'échec au franchissement de la transition.

Une forme de production *conditionnelle* permet de calculer un attribut production, dont la valeur est soumise à des conditions. La forme usuelle est du type $(condition^1 \wedge production\ effective^1) \vee \ldots \vee (condition^n \wedge production\ effective^n)$. Chaque production effective doit désigner le même attribut (celui qui est calculé) qui est ici encore inconnu. Chaque condition est une expression calculable, relative à des attributs déjà connus (attributs des jetons issus d'arcs consommation ou information, voire d'attributs de jetons issus d'arc production et déjà calculés par des productions effectives ou conditionnelles précédentes). Les conditions doivent également être exclusives (une seule d'entre elles est vérifée, les autres sont fausses), et l'attribut final est celui issu de la production effective associée à la condition vérifiée.

Une forme de production *relationnelle* possède la même forme qu'une forme de production effective, mais le terme de gauche doit être connu (c'est-à-dire qu'il a déjà été calculé par une forme effective ou par une forme conditionnelle). L'effet de l'évaluation de cette forme de production est de comparer le terme de gauche à celui de droite, et peut donc aboutir à un échec de franchissement de la transition. Nous noterons ici qu'une forme de production relationnelle peut très bien être considérée comme étant l'un des prédicats de la partie condition d'une règle-ECA.

type de production	exemple	commentaires
effective	$A^+.x = B^-.x + 1$	$B^-.x$ étant connu, $A^+.x$ qui est alors inconnu reçoit une valeur
conditionnelle	$(B^-.y = 0 \wedge A^+.y = 0)$ $(B^-.y \neq 0 \wedge A^+.y = 2 \times B^-.y)$	$B^-.y$ étant connu, $A^+.y$ (inconnu) reçoit 0 si $B^-.y$ est nul, sinon $A^+.y$ reçoit le double de $B^-.y$
relationnelle	$(A^+.x < A^+.y)$	la transition est franchissable si et seulement si $A^+.x$, maintenant connu, est inférieur à $A^+.y$, également connu

Tableau 10: Les trois types de contraintes de production

IV.1.4.4.2.2 Evaluation des contraintes de clé et de place

Une fois tous les attributs des jetons produits calculés, et si aucune production relationnelle n'a conduit à un échec, les contraintes de clé et les contraintes de place, si elles existent, sont évaluées à leur tour, pouvant également entraîner un échec du franchissement de la transition.

Dans l'exemple de la transition *réaffecter_examinateur*, il n'existe que des productions effectives, associée à une contrainte de clé:

$affectation^+.réf = réaffectation^-.réf$
$affectation^+.nom\_examinateur = réaffectation^-.nom\_examinateur$

Ce qui se traduira par la production d'un jeton dans la place *affectation*, dont les attributs sont ceux du nouvel examinateur envisagé, si et seulement si les deux attributs *réf* et *nom_examinateur* n'existent pas dans la table *affectation*.

affectation	réf	nom_examinateur
	1	'Paul'

Exemple 17: Résultat des productions

IV.1.4.4.3 Récapitulatif des différentes étapes

Le pseudo-code suivant résume globalement les actions à effectuer pour valider ou non le franchissement d'une transition:

```
[Etape 1] : évaluation des contraintes de transition relatives aux arcs de type consommation,
            information, et information négative, de poids compris entre 1 et n (évaluation des
            contraintes 1-n)
si ces contraintes sont toutes vérifiées
  alors
    [Etape 2] : considérant un tuple de l'étape 1, évaluation des contraintes de transition relatives
                aux arcs de type consommation et information de poids * (évaluation des
                contraintes *)
    si ces contraintes sont toutes vérifiées
      alors
        [Etape 3] : considérant chacun des ensembles de tuples produits à l'étape 3, évaluation
                    des contraintes faisant intervenir des cardinalités
        si ces contraintes sont toutes vérifiées
          alors
            [Etape 4] : calcul des attributs des jetons produits, à l'aide des contraintes de
                        productions effectives et de productions conditionnelles; les productions
                        relationnelles, les contraintes de place et les contraintes de clé sont
                        évaluées
            si ces contraintes sont toutes vérifiées
              alors
                [Etape 5] : préparation des productions et des consommations
                /* le franchissement est possible */
            fin si
        fin si
    fin si
fin si
/* le franchissement est impossible */
```

Pratiquement, le pseudo-code (expurgé de toute particularité due à l'implémentation) généré par NetSPEC se présente comme:

```
/* Transition REAFFECTER_EXAMINATEUR */
   // Par rapport à la définition du réseau formel, NetSPEC nomme explicitement les arcs pour son usage interne.
/*    (zz0061.FIN_NOTATION(1)?.ETAT)==('oui') */
/*  & (zz0064.REAFFECTATION(1)-.REF)==(zz0063.EXAMINATEUR(1)-.REF) */
/*  & (zz0064.REAFFECTATION(1)-.ANCIEN_EXAMINATEUR)==(zz0063.EXAMINATEUR(1)-.NOM) */
/*  & ((zz0063.EXAMINATEUR(1)-.REF)==(zz0062.NOTE(1)~.REF))&((zz0063.EXAMINATEUR(1)-
                                                              .NOM)==(zz0062.NOTE(1)~.NOM_EXAMINATEUR)) */
/*  & (zz0065.NOTE(*)?.REF)==(zz0063.EXAMINATEUR(1)-.REF) */
/*  & (CARD({zz0065.NOTE(*)?}))<(3) */
/*  & (zz0066.AFFECTATION(1)+.REF)==(zz0064.REAFFECTATION(1)-.REF) */
/*  & (zz0066.AFFECTATION(1)+.NOM_EXAMINATEUR)==(zz0064.REAFFECTATION(1)-.NOUVEAU_EXAMINATEUR)
*/

// Pseudo-code de la transition REAFFECTER_EXAMINATEUR

fonction réaffecter_examinateur
   transition franchissable: booléen
   début
      transition_franchissable ← FAUX

// Etape 1: évaluation des contraintes de transition relatives aux arcs de type consommation, information,
// et information négative, de poids compris entre 1 et n. Un curseur est créé à cet effet. NetSPEC utilise des noms
// de corrélation pour désigner les tables (ZZCN). Le numéro du nom de corrélation est celui de l'arc, et le suffixe (lettre)
// permet de différencier les variables indicées (ici c'est inutile car tous les arcs sont de poids 1).

      DECLARE CURSOR C17 AS SELECT *
            FROM FIN_NOTATION ZZCN0061A,EXAMINATEUR ZZCN0063A,REAFFECTATION ZZCN0064A
            WHERE (ZZCN0061A.ETAT='oui')
               AND (ZZCN0064A.REF=ZZCN0063A.REF)
               AND (ZZCN0064A.ANCIEN_EXAMINATEUR=ZZCN0063A.NOM)
               AND (NOT EXISTS(SELECT * FROM NOTE ZZCN0062A
                                    WHERE (ZZCN0063A.REF=ZZCN0062A.REF)
                                       AND (ZZCN0063A.NOM=ZZCN0062A.NOM_EXAMINATEUR)))

// Etape 2: considérant un tuple de l'étape 1, évaluation des contraintes de transition relatives aux arc
// de type consommation et information de poids *.

      OPEN C17
      tant que ¬EOF(C17) et ¬transition_franchissable faire
         // sélectionner un tuple; les attributs des jetons sont placés dans des variables « host »; les variables utilisées sont
         // zz0138:FIN_NOTATION.ZZPK, zz0139: FIN_NOTATION.ETAT, zz0198: EXAMINATEUR.ZZPK,
         // zz0199: EXAMINATEUR.REF, zz0200:EXAMINATEUR.NOM, zz0232: REAFFECTATION.ZZPK,
         // zz0233: REAFFECTATION.REF, zz0234: REAFFECTATION.ANCIEN_EXAMINATEUR,
         // zz0235: REAFFECTATION.NOUVEAU_EXAMINATEUR; les attributs ZZPK représentent les clés primaires
         // automatiquement ajoutées par NetSPEC.
         FETCH C17 INTO :zz0138,:zz0139,:zz0198,:zz0199,:zz0200,:zz0232,:zz0233,:zz0234,:zz0235
         // Etape 3: Evaluation des contraintes faisant intervenir des cardinalités
         DECLARE CURSOR C18 AS SELECT
                     COUNT(*)
                        FROM NOTE
                        WHERE (NOTE.REF=:zz0199) // zz0199: EXAMINATEUR.REF
         OPEN C18
         FETCH C18 INTO :zz0097 // zz0097 contient la cardinalité de l'ensemble construit depuis la place NOTE
         CLOSE C18
         si zz0097 < 3 alors // vérification de la contrainte de cardinalité
            // Etape 4: calcul des attributs des jetons produits
            zz0243 ← zz0233 // AFFECTATION.REF ← REAFFECTATION.REF
            zz0244 ← zz0235 // AFFECTATION.NOM_EXAMINATEUR ← REAFFECTATION.NOUVEAU_EXAMINATEUR
            // Etape 5: préparation des productions
            INSERT INTO AFFECTATION VALUES(:zz0120,:zz0243,:zz0244)
            zz0120 ← zz0120 + 1 // zz0120 est une variable globale représentant la clé primaire ZZPK de la table
            // Etape 5: préparation des consommations
            DELETE FROM EXAMINATEUR WHERE ZZPK=:zz0198
            DELETE FROM REAFFECTATION WHERE ZZPK=:zz0232
            transition_franchissable ← VRAI
         fsi
      ftq
      CLOSE C17
   fin
```

IV.2 Conclusion

Nous avons consacré ce chapitre à la mise en oeuvre de notre approche. En partant des éléments de la théorie des réseaux formels, nous avons détaillé notre méthode de résolution des contraintes de place, de clé, et de transition. Nous avons également ajouté les concepts relatifs aux ADBMS comme la gestion des conflits (priorités) et les modes de couplage.

L'outil *NetSPEC* respecte les autres objectifs que nous nous étions fixés, notamment l'automatisation de la création de la partie opératoire d'un système d'information, en proposant au concepteur l'utilisation d'un langage de modélisation simple, faisant abstraction de toute forme de programmation classique.

La conception de *NetSPEC* a cependant mis en évidence des problèmes qui n'existent pas à priori dans la théorie des réseaux formels comme, par exemple:

- Le modèle formel peut comporter des transitions concurrentes dont les contraintes ne sont pas exclusives. Une telle configuration conduit à l'apparition d'*indéterminismes* dans le comportement du réseau. *NetSPEC* se rapproche de la méthode utilisée dans les ADBMS pour résoudre ces conflits en proposant l'utilisation des priorités entre les transitions (par exemple, si nous utilisons la priorité fixe pour le problème des empilements de cubes, la transition *action_impossible* peut être supprimée). Il en découle que le « style de programmation » influence de manière non négligeable la modélisation d'une solution.
- La nécessité de concevoir une chaîne de développement complète, incluant notamment un outil de mise au point, est primordiale. En effet, les tirs des transitions d'un réseau formel induisent les mêmes conséquences que les déclenchements des règles-ECA des ADBMS: le comportement des transitions (des règles-ECA) peut devenir complexe voire difficilement maîtrisable, à cause de l'indéterminisme. Le concepteur d'une application en phase de développement doit alors pouvoir directement influencer le comportement des transitions, notamment pour résoudre les problèmes de terminaison ou d'interblocage.

CONCLUSION GENERALE

Notre travail a donc conduit à la réalisation d'un système opérationnel de gestion de bases de données actives basé sur un modèle graphique et déclaratif. Nous avons également analysé l'efficacité de l'outil sur des exemples significatifs issus de domaines différents. Nous avons notamment étudié l'effet de différentes stratégies de résolution des conflits sur l'efficacité globale du système. A l'issue de notre réflexion, nous estimons qu'un tel outil peut diminuer sensiblement le temps de conception et de réalisation d'un système d'information. En effet, nous avons montré que le passage de la phase de modélisation à la phase de réalisation d'un prototype opérationnel peut être entièrement automatisée avec l'outil *NetSPEC*. Bien entendu, notre étude ne prend pas en compte certains points importants, comme la prise en compte de l'architecture, ou les aspects organisationnels. Néanmoins, l'adaptation de notre outil, par exemple l'intégration à une architecture client/serveur, ne devrait pas poser de difficultés fondamentales.

Nous n'avons pas abordé dans ce mémoire la justification des propriétés des algorithmes employés : il serait souhaitable de démontrer rigoureusement la correction, la complétude ou la terminaison de ces algorithmes. Néanmoins, ce genre de preuves sur des algorithmes aussi complexes pose des difficultés non négligeables, et la plupart des algorithmes usuels sur les ADBMS sont donnés sans ces justifications. D'autre part, la génération des requêtes SQL correspondant aux annotations des transitions est souvent très délicate. Nous n'avons pas développé ce point dans le texte, et il y aurait sans doute plusieurs optimisations possibles dans l'application de ce processus.

Nous avons abordé dans la présentation des exemples l'influence d'un *style* dans la description du réseau sur l'efficacité du système final. Il serait nécessaire de poursuivre cette réflexion sur l'élaboration d'une véritable *méthode* de modélisation et de conception, intégrant dans notre démarche la conception préalable du modèle conceptuel de données, la validation de propriétés formelles (invariants, absence de blocages, ...), l'organisation des traitements. Ce travail fait actuellement l'objet d'une thèse dans l'équipe, incluant notamment une utilisation de la méthode B (Philippe Bon).

Enfin, Arnaud Lefort a montré [LEF 98] l'utilité de prendre en compte des *marquages partiellement connus*. L'intégration à *NetSPEC* de tels concepts pourrait permettre la conception efficace de *bases de données à contraintes*. Ici encore, la réalisation pratique nécessite la résolution de problèmes théoriques et pratiques non triviaux : intégration d'un solveur de contraintes dans le moniteur des tirs, définition d'ensembles infinis (e.g. intervalles), ...

Le développement d'un environnement complet incluant une méthode, un outil de validation et la prise en compte de l'architecture est une tâche considérable. Nous espérons qu'elle pourra être menée à bien dans les années qui viennent.

BIBLIOGRAPHIE

[ABR 84] J.R. Abrial: *Spécifier, ou comment matérialiser l'abstrait.* Techniques et Sciences Informatiques, Vol. 3, No. 3, 1984. pp 210-219.

[ABR 94] J.R. Abrial: *Steam-Boiler Control Specification Problem.* Technical Report, 1994. 8 p.

[AHO 89] A. Aho, R. Sethi, J. Ullman: *COMPILATEURS. Principes, techniques et outils.* InterEditions, Paris, 1989. 875 p.

[AGR 89] R. Agrawal, N.H. Gehani: *ODE (Object Database and Environment): The Language and the Data Model.* In Proceedings of the ACM-SIGMOD International Conference on Management of Data, Portland, Oregon, May-June 1989. pp 36-45.

[AGR 91] R. Agrawal, R.J. Cochrane, B. Lindsay: *On Maintaining Priorities in a Production Rule System.* In Proceedings of the 17th International Conference on Very Large Data Bases (VLDB), Barcelona, Spain, September 1991. pp 479-487.

[ALL 97] L. Allain, A. Hébrard: *From Formal Nets to Code Generation.* In IEEE International Conference on Systems, Man and Cybernetics, Orlando, Florida, October 1997. 6 p.

[ALL 98] L.Allain, C. Cadivel, A. Lefort, P. Yim: *Les réseaux formels: un outil pour la modélisation des traitements.* Soumis à la revue Ingénierie des Systèmes d'Information 1998. 36 p.

[ANS 75] ANSI/X3/SPARC Study Group On Data Base Management Systems: *Interim Report.* ACM SIGMOD Bulletin 7, No. 2, 1975.

[ANS 89] ANSI C, American National Standard for Information Systems: *Programming Language C.* 1989.

[ATK 89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik: *The Object-Oriented Database System Manifesto.* In Proceedings of the 1st DOOD Conference, Kyoto, Japan, December 1989.

[BAC 69] C.W. Bachman: *Data Structure Diagrams.* Data Base, ACM SIGBDP, Vol.1 No.2, 1969.

[BER 92] M. Berndtsson, B. Lings: *On Developing Reactive Object-Oriented Databases.* In Active Databases, Special Issue of the Bulletin of the IEEE Transactions on Data Engineering Vol. 15, 1992.

[BOO 91] G. Booch: *Conception orientée objet et applications.* Addison-Wesley, Paris, 1991. 588 p.

[BOU 78] F. Bouille: *A Survey on The Hypergraph-Based Data Structure and its Applications to Cartography and Mapping.* User's Conference on Computer Mapping Software and Data Bases, Harvard, USA, July 1978.

[BRO 85] L. Brownston, R. Farrell, E. Kant, N. Martin: *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*. Addison-Wesley, Reading, Massachusetts, 1985.

[BUC 94] A.P. Buchmann: *Active Object Systems*. In Advances in Object-Oriented Database Systems. Computer and System Sciences Vol. 130, Springer, 1994.

[BUC 95] A.P. Buchmann, J. Blakeley, J.A. Zimmermann, D.L. Wells: *Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions*. In Proceedings of th 11th International Conference on Data Engineering, Taipei, Taïwan, March 1995.

[CAD 97] C. Cadivel: *Contribution à la Spécification des Systèmes d'Information*. Thèse de Doctorat en Informatique et Automatique Appliquées, INSA Lyon, France, 1997.

[CAT 95] R. G. G. Cattell: *The Object Database Standard: ODMG*. Morgan Kaufmann, 1995.

[CHA 76] D. Chamberlain: *SEQUEL 2. A Unified Approach to Data Definition Manipulation and Control*. IBM Journal of Research, Vol. 20, No. 6, 1976.

[CHA 89a] S. Chakravarthy: *Rule Management and Evaluation: An Active DBMS perspective*. In ACM-SIGMOD Record Vol. 18 No. 3, September 1989. pp 20-28.

[CHA 89b] S. Chakravarthy: *HiPAC: A Research Project in Active, Time-Constrained Database Management, Final Report*. Technical Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, Massachussets, August 1989.

[CHA 92] S. Chakravarthy: *Architectures and Monitoring Techniques for Active Databases: an Evaluation*. Technical Report UF-CIS-TR-92-041, CIS Department, University of Florida, 1992. 28 p.

[CHA 93a] S. Chakravarthy: *A Comparative Evaluation of Active Relational Databases*. Technical Report UF-CIS TR-93-002, CIS Department, University of Florida, January 1993. 7 p.

[CHA 93b] S. Chakravarthy, D. Mishra: *Snoop: An Expressive Event Specification Language for Active Databases*. Technical Report UF-CIS TR-93-007, CIS Department, University of Florida, March 1993. 34 p.

[CHA 93c] S. Chakravarthy, V. Krishnaprasad, E. Anwar, S.K. Kim: *Anatomy of a Composite Event Detector*. Technical Report UF-CIS TR-93-039, CIS Department, University of Florida, December 1993. 41 p.

[CHA 94] S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, R.H. Badani: *ECA Rule Integration into OODBMS: Architecture and Implementation*. Technical Report UF-CIS TR-94-023, CIS Department, University of Florida, May 1994. 22 p.

[CHE 76] P. P. S. Chen: *The Entity-Relationship Model: Toward a unified view of data*. In ACM Transactions on Database Systems, Vol. 1, No. 1, March 1976. pp 9-36.

[COD 70] E.F. Codd: *A Relational Data Model for Large Shared Data Banks*. In ACM, Vol. 13 No 3, June 1970. pp 377-387.

[COD 71] CODASYL DATA BASE TASK GROUP: *Report*. ACM, New-York, April 1971.

[COD 72] E.F. Codd: *Further Normalization of the Database Relational Model*. In Database Systems, R. Rustin (ed)., Prentice-Hall, 1971.

[COD 74] E.F. Codd: *Recent Investigations in Relational Database Systems*. IFIP Congrès 1974, North-Holland.

[COD 78] CODASYL Programming Language Committee: *COBOL Journal of Development*, 1978.

[COL 94] C. Collet, T. Coupaye, T. Svensen: *NAOS: Efficient and Modular Reactive Capabilities in an Object-Oriented Database System*. In Proceedings of the 20th International Conference on Very Large Data Bases, Santiago, Chile, September 1994.

[COU 90] M. Courvoisier, M. Paludetto, R. Valette: Génération de code ADA à partir d'une approche orientée objets HOOD/Réseaux de Petri. Actes des 3èmes journées internationales, Le Génie Logiciel et ses Applications, Toulouse, Décembre 1990. Nanterre: EC2, 1990, pp 795-826.

[CRO 75] Nom collectif CROCUS: *Systèmes d'Exploitation des Ordinateurs*. Bordas, Paris, 1975. 363 p.

[CRO 88] D.J. Cronin: *Mastering ORACLE: featuring ORACLE's SQL Standard*. 1st Ed. Hayden Books, Indianapolis, 1988, 1989.

[DAT 71] C.J. Date, P. Hopewell: *File Definition and Logical Data Independance*. ACM SIGFIDET 1971, Workshop on Data Description Access and Control.

[DAT 87] C.J. Date: *A guide to INGRES: A User's Guide to the INGRES Product from Relational Technology Inc*. Addison-Wesley, Reading, Massachussets, 1987.

[DAT 89] C.J. Date, C.J. White: *A guide to DB2*. 3rd Ed. Addison-Wesley, Reading, Massachussets, 1987.

[DAV 92] R. David, H. Alla: *Du Grafcet aux Réseaux de Petri*. Hermès, Paris, 1992.

[DIT 95] K.R. Dittrich, S. Gatziu, A. Geppert: *The Active Database Management System Manifesto: A Rulebase of ADBMS Features*. In Proceedings of the 2nd International Workshop on Rules in Databases Systems (RIDS), Athens, Greece, September 1995. 15 p.

[DON 77] A. Donald: *The ANSI/SPARC DBMS MODEL*. Jardine Ed, 1977, North Holland.

[ESW 76] K.P. Eswaran: *Specifications, Implementations and Interactions of a Trigger Subsystem in an Integrated Database System*. Technical Report. IBM Research Laboratory, San Jose, California, 1976.

[FOR 82] C.L. Forgy: *RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem*. Artificial Intelligence, Vol. 19, 1982.

[FRI 97] H. Fritschi, S. Gatziu, K.R. Dittrich: *FRAMBOISE - an Approach to construct Active Database Mechanisms*. Technical Report 97-04, Institüt für Informatik, Universität Zürich, Switzerland, April 1997. 28 p.

[GAB 99] K. Gaber: Programmation Relationnelle et Relations Dynamiques. Thèse de Doctorat en Informatique et Automatique Appliquées, INSA Lyon, France, 1999. En préparation.

[GAR 79] G. Gardarin, R. Gomez de Cedron: *Gestion des fichiers*. Techniques de l'ingénieur, H3120, 3. 1979.

[GAR 82] G. Gardarin: *Bases de données: les systèmes et leurs langages*.Eyrolles Paris, 1982. 265 p.

[GAR 90] G. Gardarin, P. Valduriez: *Bases de données objets, déductives, réparties*.Eyrolles Paris, 1990. 255 p.

[GAT 94a] S. Gatziu, K.R. Dittrich: *Detecting Composite Events in an Active Database Systems Using Petri Nets*. In Proceedings of the 4th International Workshop on Research Issues in Data Engineering: Active Database Systems, Houston, Texas, February 1994. pp 2-9.

[GAT 94b] S. Gatziu, A. Geppert, K.R. Dittrich: *The SAMOS Active DBMS Prototype*. Technical Report 94.16, Institut für Informatik, Universität Zürich Switzerland, 1994. 11 p.

[GEH 91] N.H. Gehani, H.V. Jagadish: *Ode as an Active Database: Constraints and Triggers*. In Proceedings of the 17th International Conference on Very Large Data Bases (VLDB), Barcelona, Spain, September 1991. pp 327-336.

[GEN 91] H.J. Genrich: Predicate/Transition Nets. In High-level Petri Nets: theory and application. Springer-Verlag, 1991. pp 3-43.

[GEP 95] A. Geppert, S. Gatziu, K.R. Dittrich, H. Fritschi, A. Vaduva: *Architecture and Implementation of the Active Object-Oriented Database Management System SAMOS: the Layered Approach*. Technical Report 95.29, Institut für Informatik, Universität Zürich Switzerland, December 1995. 23 p.

[HAN 91] E.N. Hanson: *The Design and Implementation of the Ariel Active Database Rule System*. Technical Report UF-CIS 018-92, CIS Department, University of Florida, September 1991. 39 p.

[HAN 92a] E.N. Hanson: *Rule Condition Testing and Action Execution in Ariel.*, In Proceedings of the ACM-SIGMOD International Conference, June 1992. pp 49-58.

[HAN 92b] E.N. Hanson, J. Widom: *An Overview of Production Rules in Database Systems*. Technical Report UF-CIS 92-031, CIS Department, University of Florida, October 1992. 30 p.

[HAR 93] G. Harhalakis, F.B. Vernadat: *Petri Nets for Manufacturing Information Systems*. In Practice of Petri Nets in Manufacturing. Chapman & Hall, Londres, 1993. 295p.

[HEW 77] C. Hewitt: *Viewing Control Structures as Patterns of Passing Messages*. In Artificial Intelligence, No. 8, 1977. pp 323-364.

[IBM 75] IBM Corporation: *Information Management System/Virtual Storage*. General Information Manual, IBM No. GH20.1260-3.

[IBM 81] IBM: *SQL/Data System Concepts and Facilities*. IBM Corporation, 1981.

[IBM 88] IBM: *System Product Interpreter User's Guide (REXX)*. IBM Corporation, 1988.

[IBM 91] IBM: *Structured Query Language (SQL) Reference*. IBM Corporation, 1991.

[JAF 94] J. Jaffar, M.J. Maher: *Constraint Logic Programming: A Survey*. Journal of Logic Programming, Vol. 19/20, May/June 1994. pp 503-582.

[JEN 92] K. Jensen: *Coloured Petri Nets*. Springer, Berlin, Vol. 1, 1992. 234 p.

[JOU 77] C. Jouffroy, C. Letang: *Les fichiers*. Dunod, Paris 1977.

[KAP 94] G. Kappel, S. Rausch-Schott, W. Retschitzegger, S. Viewweg: *TriGS: Making a Passive Object-Oriented Database System Active*. Journal of Object-Oriented Programming Vol. 7, No 4, July 1994.

[KIE 90] G. Kiernan, C. de Maindreville, E. Simon: *Making Deductive Database a Practical Technology: A Step Forward.* In Proceedings of the ACM SIGMOD International Conference on Management of Data, May 1990.

[KLO 92] P. E. Kloeden, E. Platen: *Numerical Solution of Stochastics Differential Equations.* Springer-Verlag Berlin Heidelberg, 1992.

[KOG 81] P. M. Kogge: *The Architecture of Pipelined Computers.* McGraw-Hill Book Company, 1981. 334 p.

[LAP 90] G. Lapis, G.M. Lohman, H. Pirahesh: *A Starburst is born (relational DBMS).* In ACM-SIGMOD Record Vol. 19 No. 2, June 1990.

[LEF 98] A. Lefort: *Les HyperNets: un outil de modélisation et de spécification.* Thèse de Doctorat en Productique, Automatique, et Informatique Industrielle, Ecole Centrale de Lille, France, 1998.

[LIG 91] D. Lightfoot: *Formal Specification Using Z.* The Macmillan Press, London, 1991. 191 p.

[LYO 90] L. Lyon: *The IMS/VS expert's guide.* Van Nostrand Reinhold, New York, 1990.

[MAI 88] C. de Maindreville, E. Simon: *A Production Rule Based Approach to Deductive Databases.* In Proceedings of the 2nd International Conference on Expert Database Systems, April 1988.

[MCA 89] D.R. McCarthy, U. Dayal: *The Architecture of an Active, Object-oriented Database Management System.* In Proceedings of the ACM-SIGMOD International Conference on Management of Data Vol. 18 No. 2, Portland, Oregon, May/June 1989. pp 215-224.

[MEY 79] B. Meyer, M. Demuyinck: *Les langages de spécification.* In EDF, Bulletin de la Direction des Etudes et Recherches, Série C, Mathématique, Informatique, No. 1, 1979. pp 39-60.

[MRI 72] MRI System Corp.: *System 2000 General Information Manual.* Austin, Texas, 1972.

[MUR 89] T. Murata: *Petri Nets: Properties, Analysis and Applications.* In Proceeding of the IEEE, Vol. 77, No. 4, April 1989. pp 171-182.

[PAS 91] D. Pascot, D. Ridjanovic; *DATARUN: La modélisation des traitements au sein des modèles de données. Application au cas IFIP.* Paris: AFCET, 4 octobre 1991. Document de travail Groupe AFCET 135 - Conception des systèmes d'informations.

[REI 85] W. Reisig: *Petri Nets, an Introduction.* EATCS Monographs on theorical computer science, Springer-Verlag, 1985. 161 p.

[REI 91] W. Reisig: *Petri Nets and Algebraic Specifications.* In High-level Petri Nets: theory and application. Springer-Verlag, 1991. pp 137-170.

[RIS 92] N. Rishe: *Database Design: The Semantic Modeling Approach.* McGraw-Hill, 1992.

[SCH 91] U. Schreier, H. Pirahesh, R. Agrawal, C. Mohan: *Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS.* In Proceedings of the 17th International Conference on Very Large Data Bases (VLDB), Barcelona, Spain, September 1991. pp 469-478.

[SIB 90] C. Sibertin-Blanc: *La modélisation du fonctionnement des systèmes d'information par réseaux de Petri à objets*". Document de travail Groupe AFCET 135 - conception des systèmes d'information. Paris : AFCET, 2 Avril 1990, 15 p.

[SIB 91] C. Sibertin-Blanc: *Cooperative Objects for the Conceptual Modelling of Organizational Information Systems*. In Proceedings of IFIP TC8 Working Conference on The Object Oriented Approach in Information Systems, Quebec City (Canada), October 28-31, 1991. 26p.

[SIM 92] E. Simon, J. Kiernan, C. De Maindreville: *Implementing High Level Active Rules on Top of a Relationnal DBMS*. In Proceedings of the 18th International Conference on Very Large Data Bases, Vancouver, Canada, August 1992.

[SPI 92] J.M. Spivey: *The Z Notation: A Reference Manual*. Prentice Hall, London, 1992. 155 p.

[STO 74] M. Stonebraker: *A Functional View of Data Independance*. ACM SIGMOD, Workshop on Data Description, Access, and Control, 1971.

[STO 90a] M. Stonebraker, L. Rowe, M. Hirohama: *The Implementation of POSTGRES*. In IEEE Transactions on Knowledge and Data Engineering Vol. 2 No. 1, March 1990. pp 125-142.

[STO 90b] M. Stonebraker: *POSTGRES DBMS*. In ACM-SIGMOD Record Vol. 19 No. 2, June 1990.

[TAR 85] H. Tardieu, D. Naci, D. Pascot: *Conception d'un système d'information - construction de la base de données*. Les Editions d'Organisation. Paris, 1985. 204 p.

[TAR 86] H. Tardieu, A. Rochfeld, R. Colletti: *La Méthode MERISE*. Les Editions d'Organisation, Paris, 1986. 318 p.

[TAY 76] R.W. Taylor, R.L. Frank: *CODASYL Data-Base Management Systems*. ACM Computing Surveys, Vol. 8 No. 1, March 1976.

[TSI 76] D.C. Tsichritzis, F.H. Lochovsky: *Hierarchical Database Management: A Survey*. ACM Computing Surveys, Vol. 8 No. 1, March 1976.

[TSI 78] D.C. Tsichritzis, A. Klug: *The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Data Base Management Systems*. Insformation Systems Vol. 3, 1978.

[WID 77] G. Widerhold: *Database Design*. McGraw-Hill, 1977.

[WID 91] J. Widom, R.J. Cochrane, B.G. Lindsay: *Implementing Set-Oriented Production Rules As An Extension To Starburst*. In Proceedings of the 17th International Conference on Very Large Data Bases (VLDB), Barcelona, Spain, 1991. pp 275-285.

[WIL 88] D. Wilkins: *Practical Planning: Extending the classical AI planning paradigm*. Morgan Kaufmann, 1988.

[YIM 95] P. Yim, C. Cadivel, A. Lefort: *Process Modeling in Information System by Formal Net*. In IEEE International Conference on Systems, Man and Cybernetics, Vancouver, Canada, October 1995. pp 2263-2270.

[ZHO 91] M.C. Zhou, F. DiCesare: *Parallel and Sequential Mutual Exclusions for Petri Net Modeling for Manufacturing Systems with Shared Resources*. In IEEE Transactions on Robotics Automation Vol. 7 No. 7, 1991. pp 515-527.

[ZHO 93] M.C. Zhou, K. McDermott, P.A. Patel: *Petri Net Synthesis and Analysis of a Flexible Manufacturing System Cell*. In IEEE Transactions on Systems, Man, and Cybernetics Vol. 23 No. 2, March/April 1993. pp 523-531.

GLOSSAIRE

ADBMS	Active *DBMS*.
Cardinalité	Nombre de fois qu'une occurrence d'un objet est impliquée dans les occurrences d'une relation.
COMMIT	Validation d'une *transaction*.
DBMS	DataBase Management System. Voir *SGBD*.
DDL	Data Definition Language. Langage de définition des données.
DML	Data Manipulation Language. Langage de manipulation des données.
MCD	Modèle Conceptuel des Données. Voir *Merise*.
MCT	Modèle Conceptuel des Traitements. Voir *Merise*.
Merise	Méthode de conception et de développement des systèmes d'information.
MLD	Modèle Logique des Données. Voir *Merise*.
Normalisation	Processus de décomposition d'une relation en formes normales.
RdP	Réseau de Petri.
Requête	Opération simple sur une base de données (insertion, destruction, modification, sélection).
ROLLBACK	Annulation d'une *transaction*.
SGBD	Système de Gestion de Bases de Données.
SQL	Structured Query Language. Voir *DML*.
SI	Système d'Information.
Transaction	Unité de travail comportant plusieurs *requêtes* sur une base de données.
Tuple	Ligne (ou rangée) d'une relation.

ANNEXE A

GRAMMAIRE DU LANGAGE DE PROGRAMMATION NETSPEC

La grammaire du langage de programmation de l'outil *NetSPEC* est donnée d'une part en notation BNF (Backus-Naur Form) et d'autre part sous forme de diagrammes syntaxiques. Ces deux présentations sont complémentaires l'une de l'autre. Il est sous-entendu que:

- le format du texte est libre (espaces, tabulations, retours à la ligne)
- des commentaires peuvent être ajoutés et commencent par le symbole # et se terminent à la fin de la ligne
- les terminaux (mots clés, ponctuation, ...) sont représentés en caractères gras
- les mots clés ne peuvent pas être employés comme identificateurs
- les mots clés SQL sont réservés et ne devraient pas être employés comme identificateurs
- ZZPK est un identificateur réservé
- dans les notes, [SQL] indique une compatibilité SQL, [C] indique une compatibilité avec le langage C

A.1: Liste des mots clés NetSPEC

AVG	BAG	CARD	CONS	CONST
CURRENT_DATE	CURRENT_TIME	DATE	GLOBAL_TICKS	INFO
INT	KEY	LENGTH	LOCAL_TICKS	MAX
MIN	NEG	NULL	PROD	REAL
STRING	SUBSTR	SUM	TIME	TRANS
WITH				

A.2: Liste des mots réservés SQL

ADD	ALL	ALTER	AND	ANY
AS	ASC	AVG	BEGIN	BETWEEN
BINDADD	BIT	BY	CASCADE	CHAR
CHARACTER	CHECK	CLOSE	COLUMN	COMMENT
COMMIT	CONNECT	CONTINUE	CONTROL	COUNT
CREATE	CREATETAB	CURRENT	CURSOR	DATA
DATABASE	DATE	DAY	DAYS	DBADM
DEC	DECIMAL	DECLARE	DELETE	DESC
DESCRIBE	DESCRIPTOR	DISTINCT	DROP	END
EXCEPT	EXCLUSIVE	EXEC	EXECUTE	EXISTS
FETCH	FLOAT	FOR	FOREIGN	FOUND
FROM	GO	GOTO	GRANT	GROUP
HAVING	HOLD	HOUR	IMMEDIATE	IN

INCLUDE	INDEX	INSERT	INT	INTEGER
INTERSECT	INTO	IS	KEY	LENGTH
LIKE	LOCK	LONG	MAX	MICROSECOND
MIN	MINUTE	MODE	MONTH	NULL
NOT	OF	ON	ONLY	OPEN
OPTION	OR	ORDER	PACKAGE	PREPARE
PRIMARY	PRIVILEGES	PROGRAM	PUBLIC	REFERENCES
RESET	RESTRICT	REVOKE	ROLLBACK	SECOND
SECTION	SELECT	SET	SHARE	SMALLINT
SOME	SQL	SQLCA	SQLDA	SQLERROR
SQLWARNING	START	STOP	SUBSTR	SUM
TABLE	TIME	TIMESTAMP	TO	TRANSLATE
UNION	UNIQUE	UPDATE	USER	USING
VARCHAR	VALUES	VIEW	WHENEVER	WHERE
WITH	WORK	YEAR		

A.3: Liste des symboles et des ponctuations

,	séparateur de liste
:	séparateur de contrainte
;	terminateur d'instruction
.	point décimal, sélecteur de champ
?	spécificateur d'entrée, spécificateur d'arc information
~	test de nullité, spécificateur d'arc information négative
^	spécificateur de nullité
'	délimiteur de chaîne de caractères
[ ]	délimiteurs de dimension de tableau
{ }	délimiteurs de début et de fin de bloc
( )	délimiteurs de début et de fin de liste, parenthésage d'expression
+	addition, concaténation, spécificateur d'arc production
-	soustraction, moins unaire, spécificateur d'arc consommation
*	multiplication, poids d'arc
/	division
!	non logique
&	et logique
|	ou logique
= =	égalité
!=	différence
<	inférieur
<=	inférieur ou égal
>	supérieur
>=	supérieur ou égal

A.4: Grammaire en notation BNF

<programme> ::=
    /* vide */
  | <programme> <déclaration_constante>
  | <programme> <déclaration_place>
  | <programme> <déclaration_transition>

<déclaration_constante> ::=
    CONST <spécificateur_type> <liste_initialisation_constante> ;

<liste_initialisation_constante> ::=
    <initialisation_constante>
  | <liste_initialisation_constante> , <initialisation_constante>

```
<initialisation_constante> ::=
        <identificateur> = <expr_constante>

<déclaration_place> ::=
        BAG <identificateur> <spécificateur_entrée>
            { <liste_déclaration_champ> } <contrainte_place> ;

<spécificateur_entrée> ::=
        /* vide */
    |   ?

<liste_déclaration_champ> ::=
        <déclaration_champ>
    |   <liste_déclaration_champ> <déclaration_champ>

<déclaration_champ> ::=
        <spécificateur_clé> <spécificateur_type> <liste_déclarateur_champ> ;

<spécificateur_clé> ::=
        /* vide */
    |   KEY

<liste_déclarateur_champ> ::=
        <déclarateur_champ>
    |   <liste_déclarateur_champ> , <déclarateur_champ>

<déclarateur_champ> ::=
        <identificateur>
    |   ^ <identificateur>

<contrainte_place> ::=
        /* vide */
    |   WITH <expr_logique_contrainte_place>

<expr_logique_contrainte_place> ::=
        <expr_unaire_contrainte_place>
    |   <expr_logique_contrainte_place> <op_logique> <expr_unaire_contrainte_place>

<expr_unaire_contrainte_place> ::=
        <expr_primaire_contrainte_place>
    |   ! <expr_primaire_contrainte_place>

<expr_primaire_contrainte_place> ::=
        <expr_prédicat_place>
    |   ( <expr_logique_contrainte_place> )

<expr_prédicat_place> ::=
        ^ <identificateur>
    |   <expr_addition_place> <op_condition> <expr_addition_place>

<expr_addition_place> ::=
        <expr_multiplication_place>
    |   <expr_addition_place> <op_addition> <expr_multiplication_place>

<expr_multiplication_place> ::=
        <expr_unaire_place>
    |   <expr_multiplication_place> <op_multiplication> <expr_unaire_place>
```

<expr_unaire_place> ::=
<expr_primaire_place>
| - <expr_primaire_place>

<expr_primaire_place> ::=
<expr_fonction_place>
| <expr_constante>
| <expr_variable>
| <identificateur>
| (<expr_addition_place>)

<expr_fonction_place> ::=
<fonction_scalaire> (<liste_paramètre_fonction_place>)

<liste_paramètre_fonction_place> ::=
<expr_addition_place>
| <liste_paramètre_fonction_place> , <expr_addition_place>

<déclaration_transition> ::=
TRANS <identificateur> (<liste_déclaration_arc>) <contrainte_transition> ;

<liste_déclaration_arc> ::=
<déclaration_arc>
| <liste_déclaration_arc> , <déclaration_arc>

<déclaration_arc> ::=
<spécificateur_arc> <poids_arc> <identificateur>

<contrainte_transition> ::=
/* vide */
| **WITH** <liste_contrainte_transition>

<liste_contrainte_transition> ::=
<expr_logique_contrainte_transition>
| <liste_contrainte_transition> , <expr_logique_contrainte_transition>
| <liste_contrainte_transition> : <expr_logique_contrainte_transition>

<expr_logique_contrainte_transition> ::=
<expr_unaire_contrainte_transition>
| <expr_logique_contrainte_transition> <op_logique>
<expr_unaire_contrainte_transition>

<expr_unaire_contrainte_transition> ::=
<expr_primaire_contrainte_transition>
| ! <expr_primaire_contrainte_transition>

<expr_primaire_contrainte_transition> ::=
<expr_prédicat_transition>
| (<expr_logique_contrainte_transition>)

<expr_prédicat_transition> ::=
^ <identificateur_composé>
| <expr_addition_transition> <op_condition> <expr_addition_transition>

<expr_addition_transition> ::=
<expr_multiplication_transition>
| <expr_addition_transition> <op_addition> <expr_multiplication_transition>

```
<expr_multiplication_transition> ::=
                <expr_unaire_transition>
        |       <expr_multiplication_transition> <op_multiplication> <expr_unaire_transition>

<expr_unaire_transition> ::=
                <expr_primaire_transition>
        |       - <expr_primaire_transition>

<expr_primaire_transition> ::=
                <expr_fonction_transition>
        |       <expr_fonction_statistique_transition>
        |       <expr_cardinalité_transition>
        |       <expr_constante>
        |       <expr_variable>
        |       <identificateur_composé>
        |       ( <expr_addition_transition> )

<expr_fonction_transition> ::=
                <fonction_scalaire> ( <liste_paramètre_fonction_transition> )

<liste_paramètre_fonction_transition> ::=
                <expr_addition_transition>
        |       <liste_paramètre_fonction_transition> , <expr_addition_transition>

<expr_fonction_statistique_transition> ::=
                <fonction_statistique> ( { <identificateur_composé> } )

<expr_cardinalité_transition> ::=
                CARD ( { <identificateur> ( * ) <spécificateur_arc_cons_info> } )

<identificateur_composé> ::=
                <identificateur> <poids_arc> <spécificateur_arc_cons_info_neg_prod> .
                                                                        <identificateur>

<spécificateur_arc_cons_info> ::=
                -
        |       ?

<spécificateur_arc_cons_info_neg_prod> ::=
                <spécificateur_arc_cons_info>
        |       ~
        |       +

<spécificateur_arc> ::=
                CONS
        |       INFO
        |       NEG
        |       PROD

<poids_arc> ::=
                /* vide */
        |       ( <constante_entière> )
        |       ( * )

<expr_constante> ::=
                <constante_entière>
        |       <constante_réelle>
        |       <constante_chaîne>
```

<spécificateur_type> ::=
INT
| **REAL**
| **TIME**
| **DATE**
| **STRING** [<constante_entière>]

<op_multiplication> ::=
*
| /

<op_addition> ::=
+
| -

<op_logique> ::=
&
| |

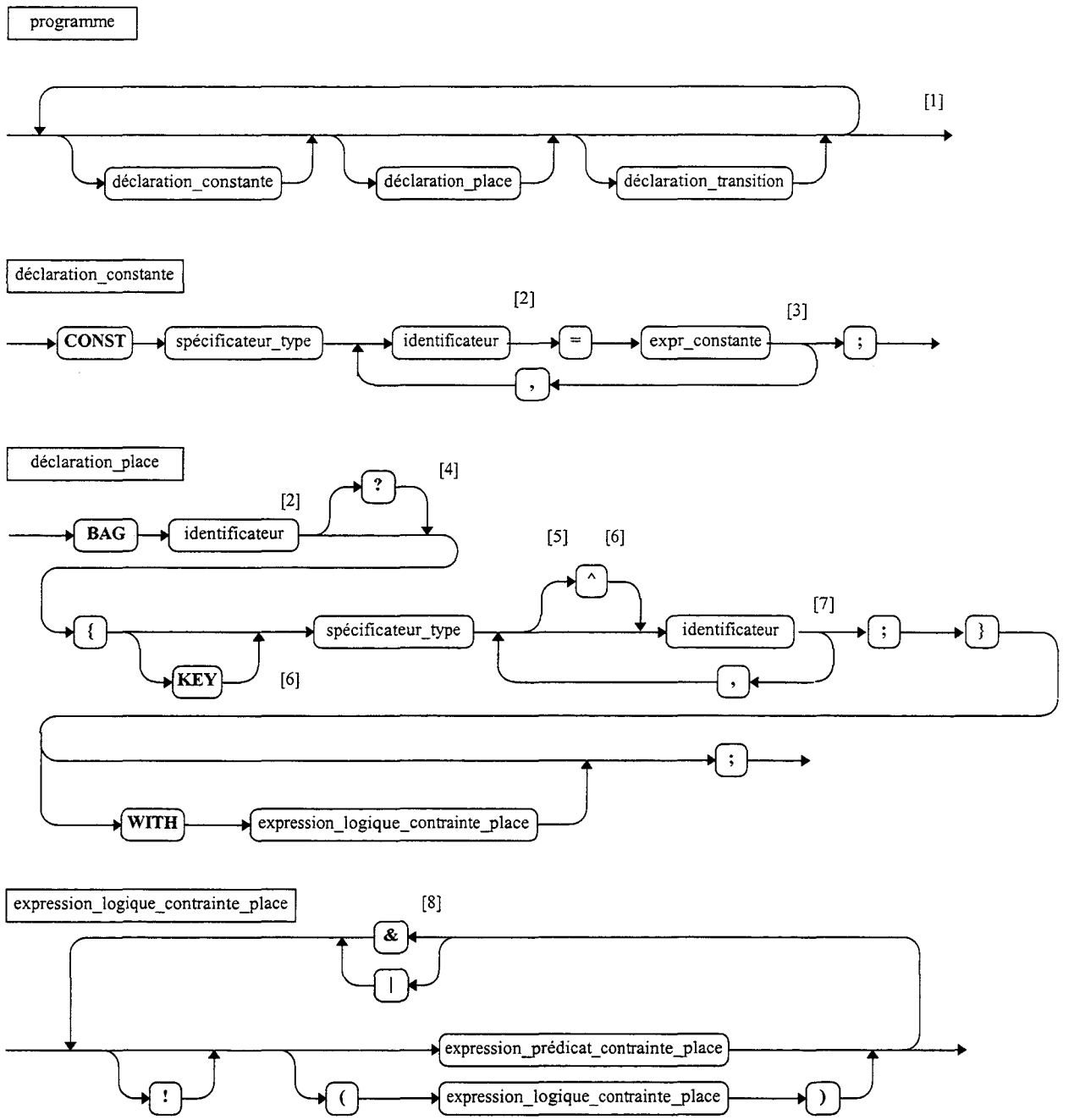
<op_condition> ::=
==
| **!=**
| **<**
| **<=**
| **>**
| **>=**

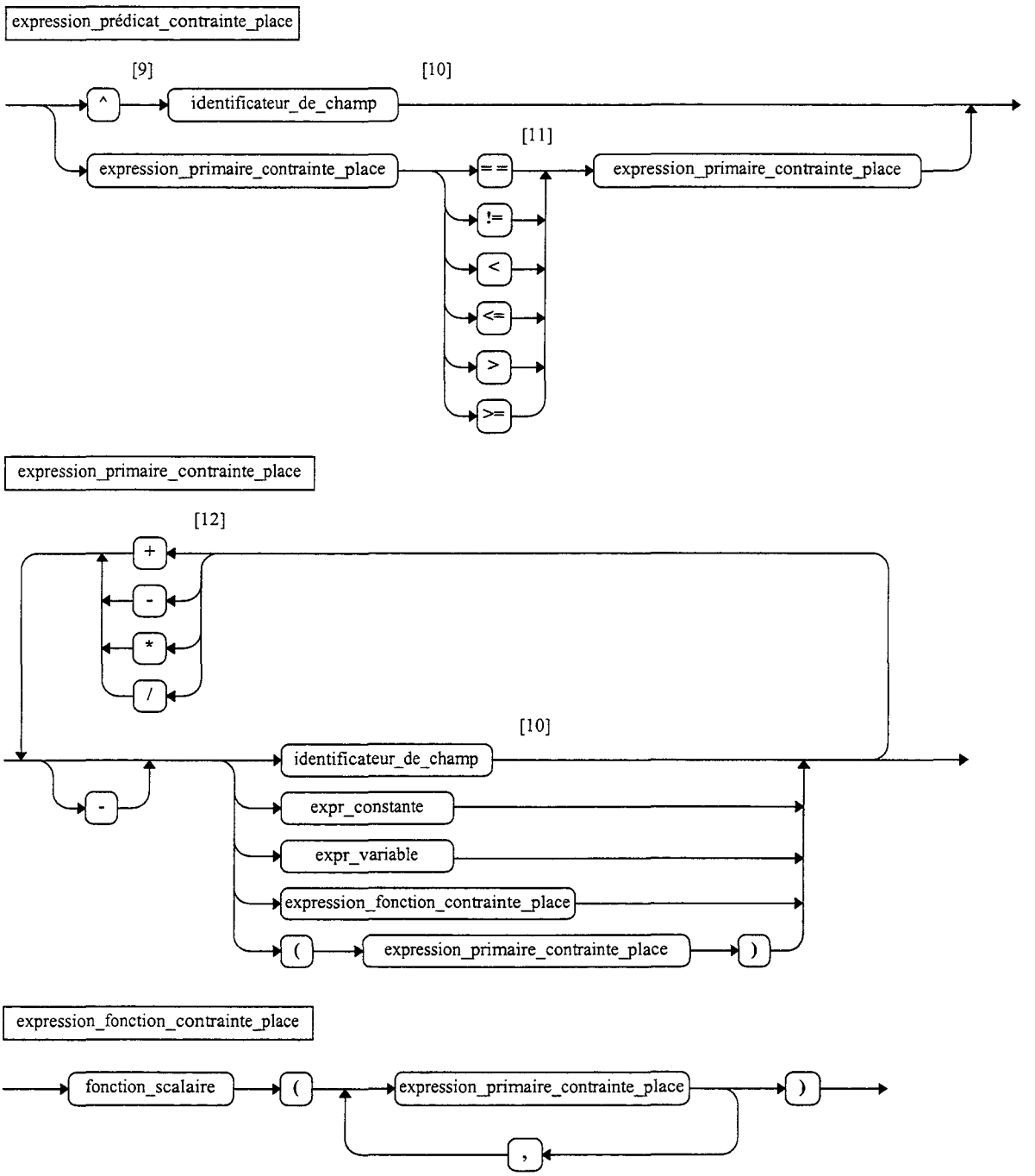
<fonction_scalaire> ::=
LENGTH
| **SUBSTR**

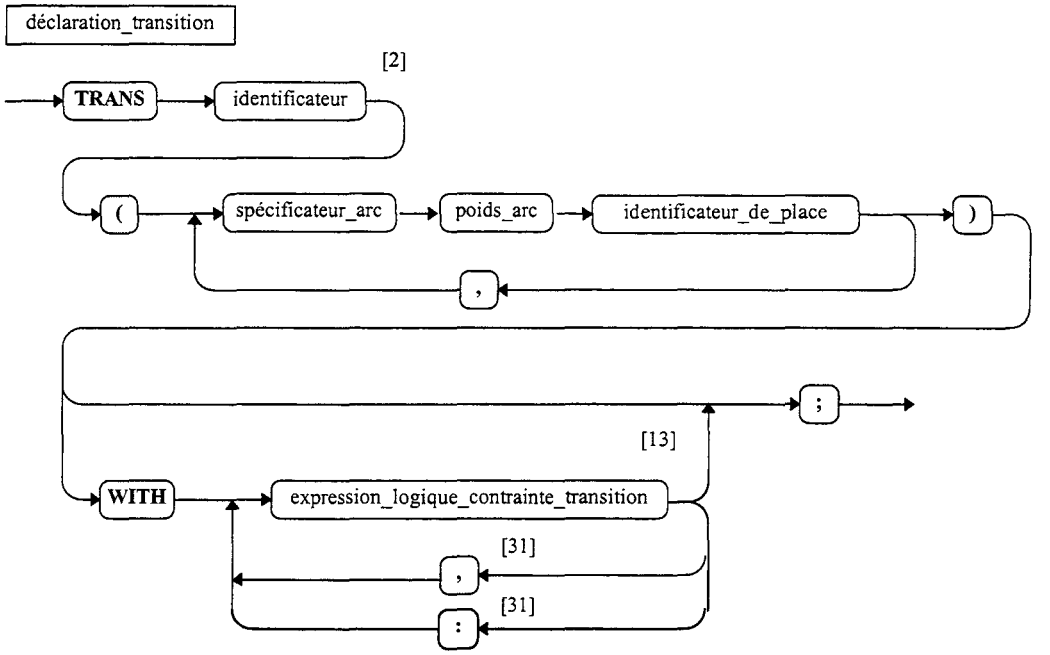
<fonction_statistique> ::=
AVG
| **SUM**
| **MIN**
| **MAX**

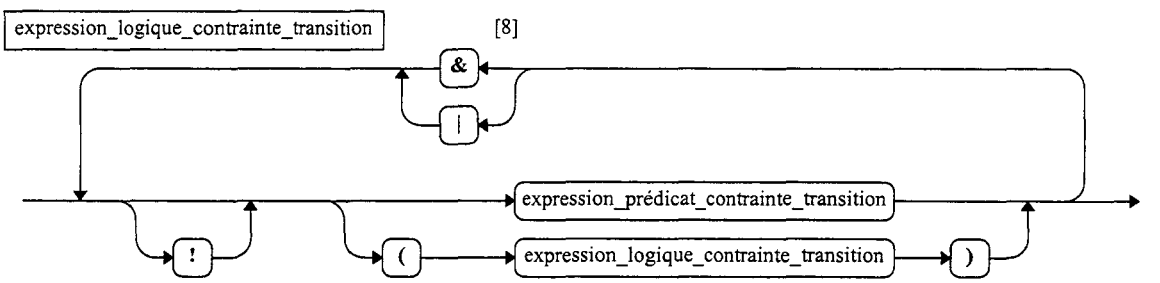
<expr_variable> ::=
CURRENT_TIME
| **CURRENT_DATE**
| **GLOBAL_TICKS**
| **LOCAL_TICKS** (identificateur)
| **NULL**

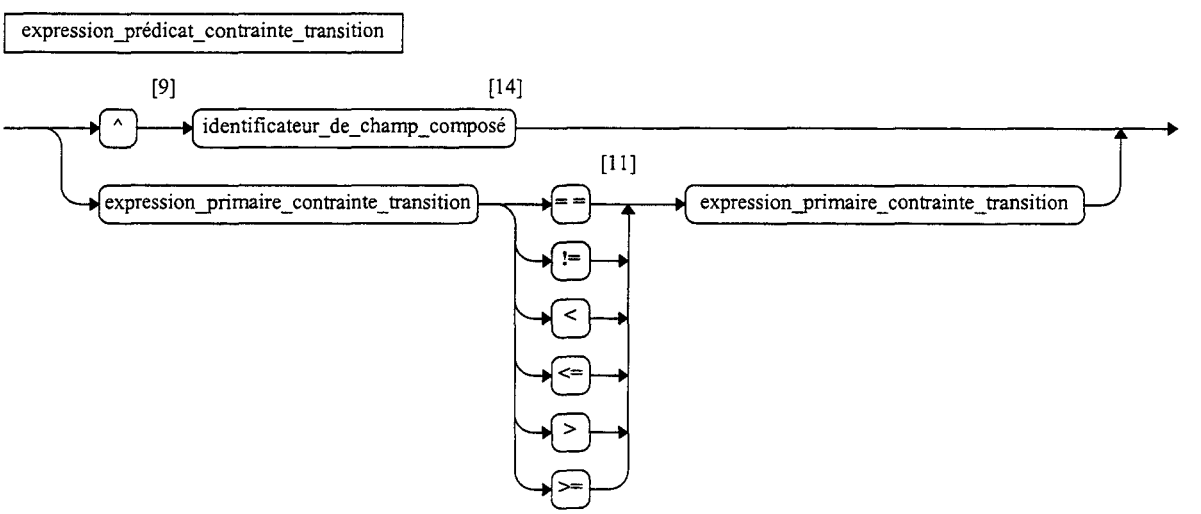
A.5: Grammaire en diagrammes syntaxiques



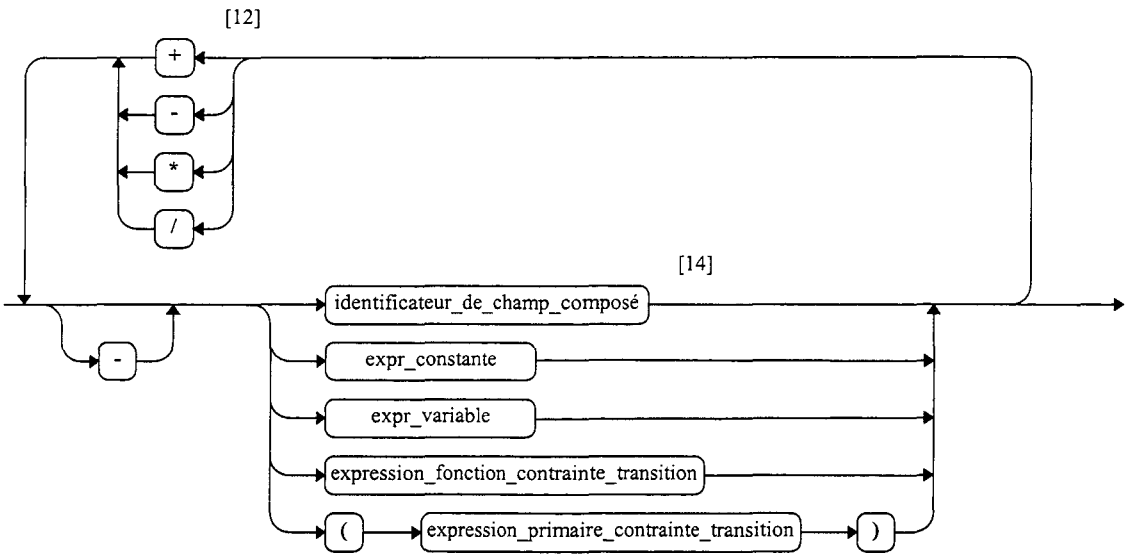

expression_prédicat_contrainte_place
[9]
[10]
^
identificateur_de_champ
expression_primaire_contrainte_place
[11]
==
!=
<
<=
>
>=
expression_primaire_contrainte_place
expression_primaire_contrainte_place
[12]
+
-
*
/
[10]
identificateur_de_champ
-
expr_constante
expr_variable
expression_fonction_contrainte_place
(
expression_primaire_contrainte_place
)
expression_fonction_contrainte_place
fonction_scalaire
(
expression_primaire_contrainte_place
)
,


déclaration_transition
[2]
TRANS
identificateur
(
spécificateur_arc
poids_arc
identificateur_de_place
)
,
;
[13]
WITH
expression_logique_contrainte_transition
[31]
,
[31]
:

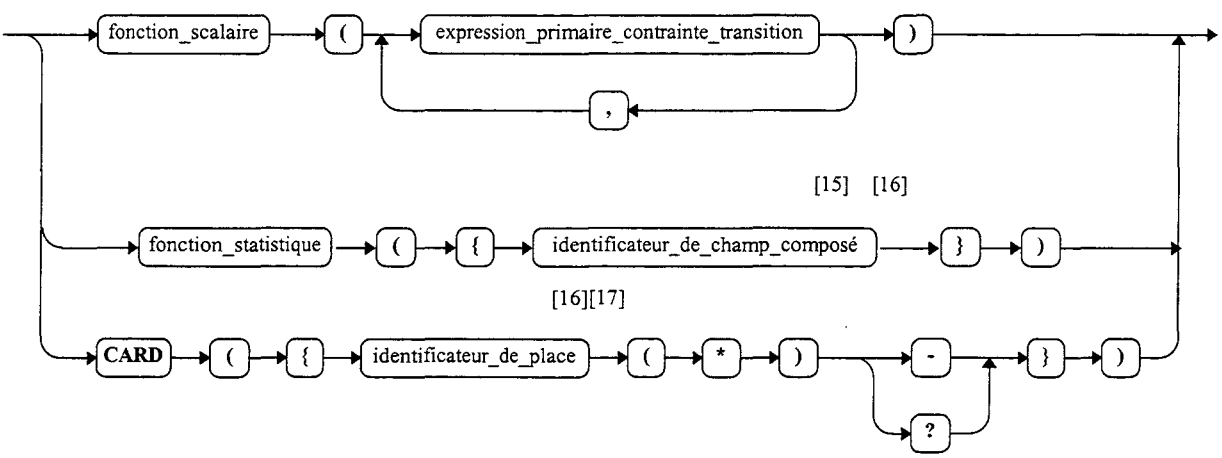

expression_logique_contrainte_transition
[8]
&
|
expression_prédicat_contrainte_transition
!
(
expression_logique_contrainte_transition
)


expression_prédicat_contrainte_transition
[9]
[14]
^
identificateur_de_champ_composé
[11]
expression_primaire_contrainte_transition
==
expression_primaire_contrainte_transition
!=
<
<=
>
>=

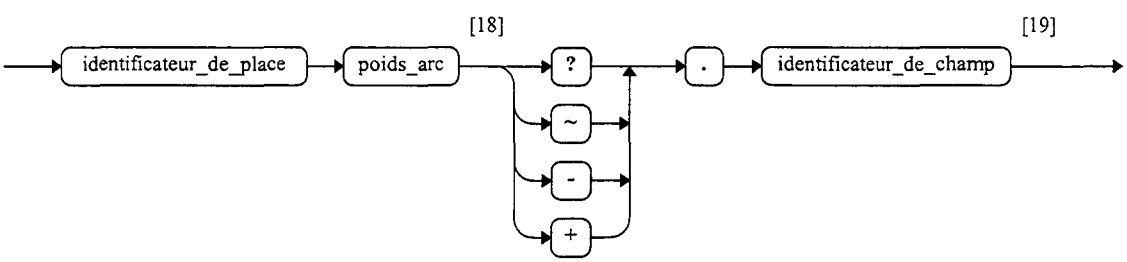
expression_primaire_contrainte_transition

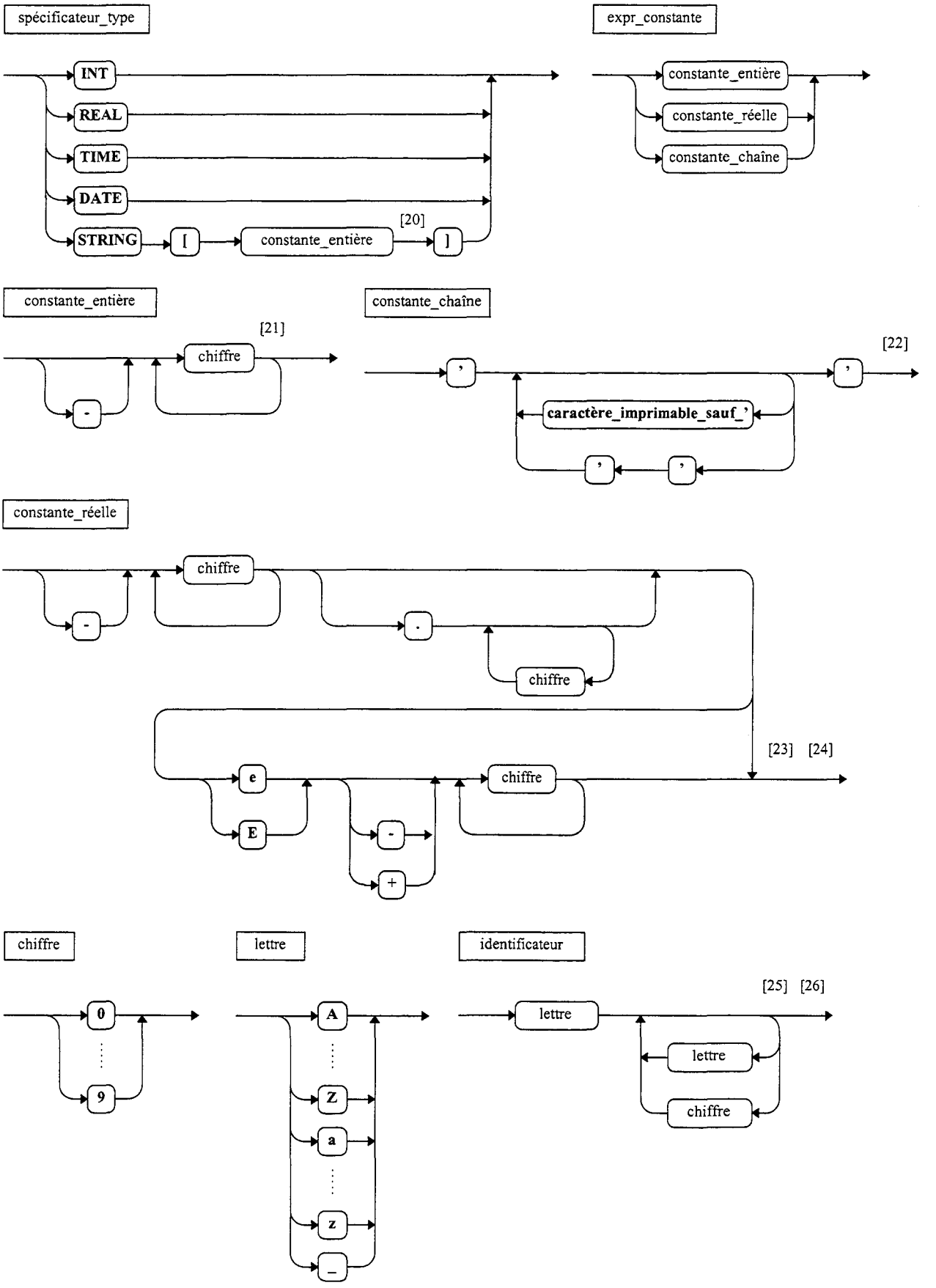


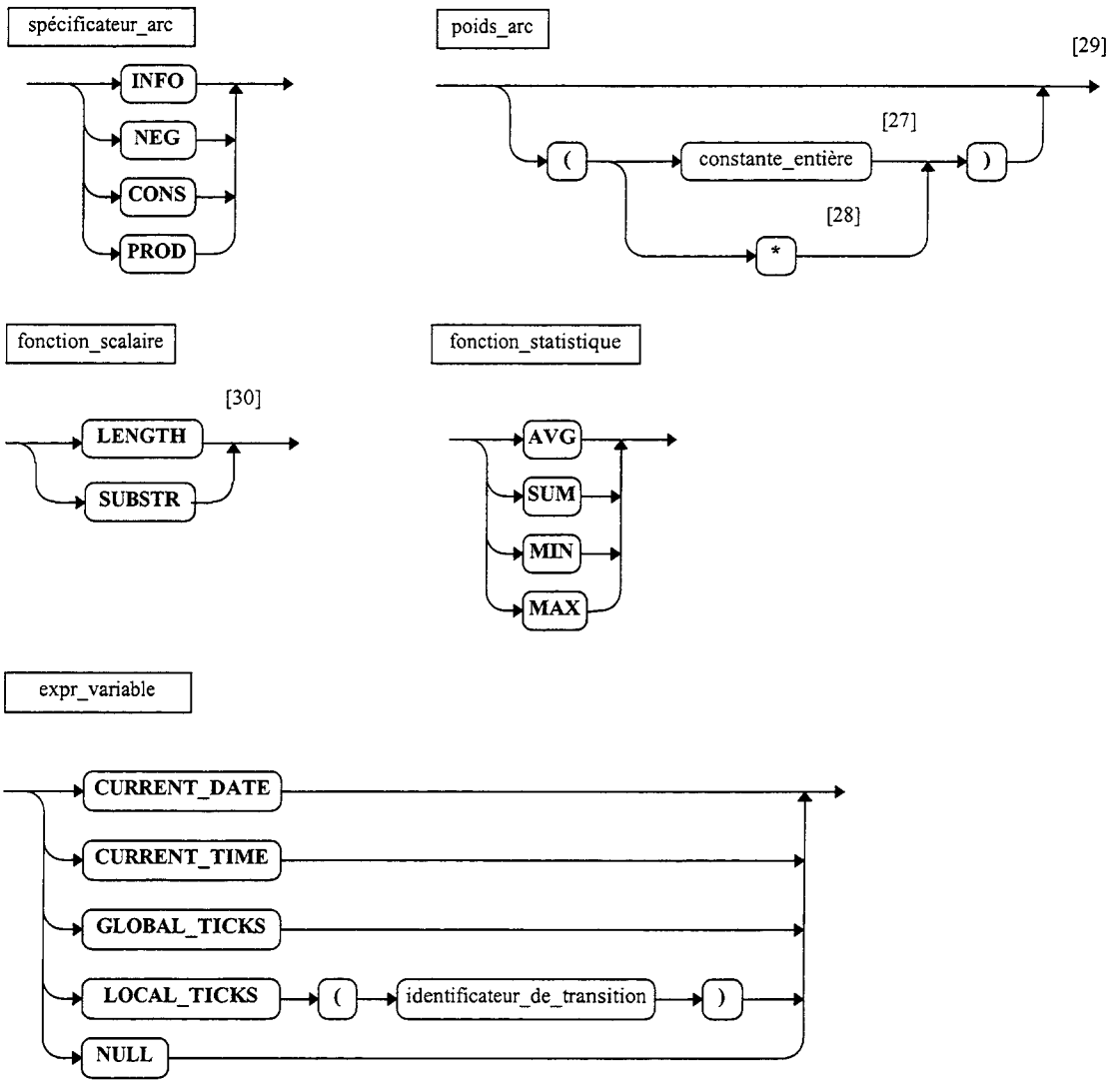
expression_fonction_contrainte_transition



identificateur_de_champ_composé




spécificateur_type
INT
REAL
TIME
DATE
STRING
[
constante_entière
]
[20]
expr_constante
constante_entière
constante_réelle
constante_chaîne
constante_entière
-
chiffre
[21]
constante_chaîne
'
caractère_imprimable_sauf_'
'
'
[22]
constante_réelle
-
chiffre
.
chiffre
e
E
-
+
chiffre
[23] [24]
chiffre
0
9
lettre
A
Z
a
z
_
identificateur
lettre
lettre
chiffre
[25] [26]



[1] les références en avant ne sont pas autorisées

[2] *identificateur* doit être unique

[3] le type de *expr_constante* doit être compatible avec *spécificateur_type*

type	*constante*
INT	INT ou REAL (arrondi à la valeur entière inférieure)
REAL	REAL ou INT
TIME	STRING (8 caractères au format hh:mm:ss) [SQL]
DATE	STRING (10 caractères au format jj/mm/aaaa) [SQL]
STRING	STRING (de longueur inférieure ou égale)

[4] l'opérateur *?* indique que la place est sujette à une entrée de jeton (autre qu'une production, i.e. interface homme/machine)

[5] l'opérateur ^ indique que le champ (attribut) peut prendre une valeur nulle

[6] *KEY* et ^ sont exclusifs

[7] *identificateur* doit être unique entre *{* et *}*

[8] la priorité de l'opérateur & est supérieure à celle de l'opérateur |

[9] l'opérateur ^ est un test de nullité

[10] *identificateur_de_champ* doit désigner un champ de la place pour laquelle on définit la contrainte

[11] les types des parties gauche et droite des opérateurs doivent être compatibles (Cf [3])

[12] les priorités des opérateurs sont dans l'ordre décroissant (* /) et (+ -)

[13] les différentes expressions de contrainte sont implicitement combinées avec l'opérateur &

[14] *identificateur_de_champ_composé* doit désigner un champ d'une place reliée à la transition pour laquelle on définit la contrainte

[15] *identificateur_de_champ_composé* désigne un champ d'une place reliée à la transition par un arc information ou consommation

[16] le poids de l'arc doit être *

[17] *identificateur_de_place* désigne une place reliée à la transition par un arc information ou consommation

[18] *poids_arc* doit être compris entre 1 et le poids de l'arc qui relie la place à la transition, sauf si le poids de ce dernier est *

[19] *identificateur_de_champ* doit désigner un champ de la place repérée par *identificateur_de_place*
[20] *constante_entière* doit être supérieure ou égale à 1 et inférieure à 254 [SQL]
[21] la dynamique s'étend de -2147483648 à +2147483647 (entier signé sur 32 bits) [C] [SQL]
[22] la longueur de la chaîne doit être comprise entre 0 et 254 caractères [SQL]
[23] la dynamique en valeur absolue s'étend de 10^{-308} à 10^{+308} (réel double-précision en codage IEEE) [C] [SQL]
[24] au moins l'une des partie fractionnaire ou exposant doit exister
[25] majuscules et minuscules sont équivalentes et sont traîtées comme des majuscules [SQL]
[26] la longueur d'un identificateur doit être inférieure ou égale à 18 caractères [SQL] ; ne s'applique pas aux identificateurs de transition
[27] *constante_entière* doit être supérieure ou égale à 1
[28] ne peut pas s'appliquer aux arcs information négative
[29] par défaut, le poids de l'arc est égal à 1
[30] se référer au manuel SQL pour les types et les caractéristiques des paramètres [SQL]
[31] la virgule sépare les contraintes reliées par la clause implicite &, les deux-points séparent les contraintes reliées par la clause implicite |.

ANNEXE B

MANUEL D'UTILISATION DE NETSPEC

B.1: Configuration logicielle

NetSPEC est conçu pour fonctionner sur une plate-forme de type Personal Computer (Pentium II, RAM 32 M, Disque 250 M). La base logicielle pré-installée doit comporter:

- Le système d'exploitation **IBM OS/2 2.11** ou **OS/2 WARP 3.0** avec les utilitaires **REXX**.
- Le gestionnaire de base de données **IBM Database Manager DB2/2 1.2** en configuration locale ou client/serveur.
- Le Toolkit **IBM OS/2 2.0**.
- Le compilateur **IBM C Set/2 2.0**.
- Un éditeur de texte quelconque.
- L'exécutable et le runtime de *NetSPEC* (**netspec.exe rtlib.obj run.exe**).
- La ligne **SET NETSPEC=C:\NETSPEC** devrait être ajoutée au fichier **config.sys**.

B.2: Compilation d'un réseau formel

Le code source du réseau formel (**fichier.z**) est compilé par *NetSPEC* en invoquant la ligne de commande:

netspec [-v] [-pE|-pF|-pL] [-h|-?] nom_du_fichier_source

Les options de compilation sont:

- **-v** détaille à l'écran les étapes de compilation
- **-pE** implémente la priorité circulaire
- **-pF** implémente la priorité fixe
- **-pL** implémente la priorité itérative
- **-h-?** liste les options de compilation

Le code source produit par *NetSPEC* doit ensuite être compilé en invoquant la ligne de commande:

fichier.cmd [-c]

dans laquelle l'option **-c** permet de créer la base de donnés (cette option n'est nécessaire qu'à la première compilation). L'exécution du réseau formel est ensuite déclenchée par:

run.exe fichier

B.3: Observation du comportement du réseau formel

L'écran est divisé en trois fenêtres standards:

- Une fenêtre principale réalise l'interface avec l'utilisateur. Ce dernier peut entrer manuellement des jetons et leurs attributs dans les places du réseau formel. Il peut également déclencher ou arrêter le moteur de tir.
- Une fenêtre secondaire indique le résultat des commandes de l'utilisateur (elle correspond à la partie de code implémentant l'insertion de nouveaux jetons dans la base de données).
- Une fenêtre tertiaire trace les différents tirs effectués (elle correspond à la partie de code implémentant le moteur de tir).

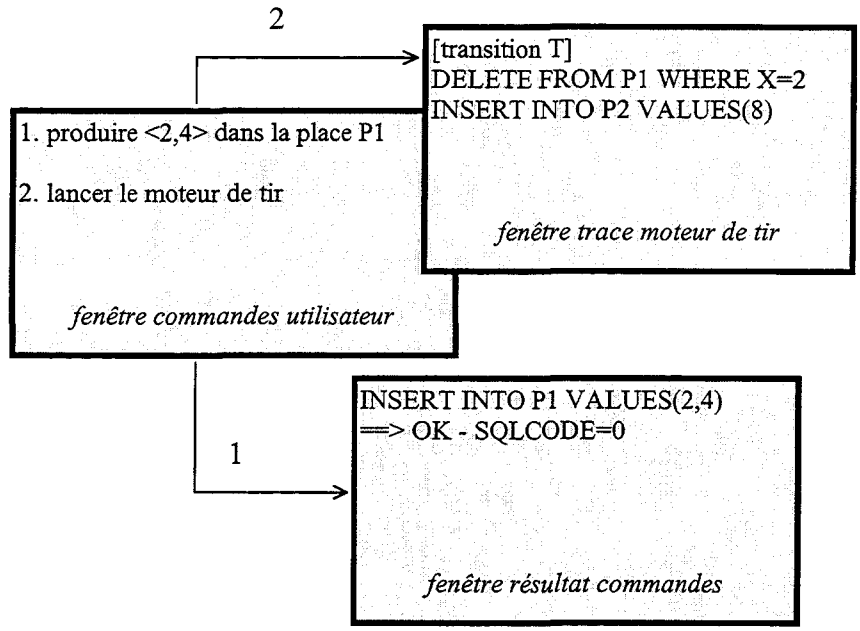


Figure 47: Ecran type d'une session NetSPEC

ANNEXE C

CODE SOURCE DU CAS IFIP

```
# Code source du réseau formel du cas IFIP

# Déclaration des constantes globales

const date date_cloture='31/10/1996', date_retard='31/12/1996';
const real seuil_accepte=9.0, seuil_refuse=8.0;
const real coef_sujet=0.5, coef_qualite=0.3, coef_biblio=0.2;

#
# Déclaration des places, attributs des jetons, contraintes de clé, et contraintes de
# places

bag courrier ? {
        int             ref;
        string[31]      nom_auteur;
        date            date_arrivee;
        string[10]      version;
} with (version=='lettre')|(version=='provisoire')|(version=='definitif');

bag auteur {
        key string[31]  nom;
};

bag derogation ? {
        key int         ref;
        string[3]       reponse;
} with (reponse=='oui')|(reponse=='non');

bag projet {
        key int         ref;
        string[31]      nom_auteur;
        date            date_arrivee;
        string[10]      version;
} with (version=='lettre')|(version=='provisoire');

bag papier {
        key int         ref;
        string[31]      nom_auteur;
};

bag membre ? {
        key string[31]  nom;
};

bag examinateur {
        key int         ref;
        key string[31]  nom;
};

bag affectation ? {
        key int         ref;
        key string[31]  nom_examinateur;
```

```
};
bag reaffectation ? {
        key int          ref;
        key string[31]   ancien_examinateur,nouveau_examinateur;
};

bag notation ? {
        key int          ref;
        key string[31]   nom_examinateur;
        real             note_sujet,note_qualite,note_biblio;
} with (note_sujet>=0.0)&(note_sujet<=10.0)&(note_qualite>=0.0)&(note_qualite<=10.0)&
       (note_biblio>=0.0)&(note_biblio<=10.0);

bag fin_notation ? {
        string[3]        etat;
} with (etat=='oui')|(etat=='non');

bag note {
        key int          ref;
        key string[31]   nom_examinateur;
        real             note_sujet,note_qualite,note_biblio;
};

bag moyenne {
        key int          ref;
        string[31]       nom_auteur;
        real             valeur;
};

bag ballottage {
        key int          ref;
        string[31]       nom_auteur;
};

bag refus {
        key int          ref;
        string[31]       nom_auteur;
};

bag acceptation {
        key int          ref;
        string[31]       nom_auteur;
};

bag definitif {
        key int          ref;
        string[31]       nom_auteur;
        date             date_arrivee;
};

bag decision ? {
        key int          ref;
        string[3]        accord;
} with (accord=='oui')|(accord=='non');

bag avis ? {
        key int          ref;
        string[3]        accord;
} with (accord=='oui')|(accord=='non');

bag session {
        key int          ref;
        string[31]       nom_presentateur;
};

bag invitation ? {
        key int          ref;
        string[31]       nom_invite;
        date             date_limite;
};

bag reponse ? {
        key int          ref;
        date             date_remise;
};
```

```
#
# Déclaration des transitions, des arcs, et des contraintes de transitions
#

trans enregistrer_auteur(info courrier,prod auteur)
    with (auteur+.nom == courrier?.nom_auteur);

trans recevoir_courrier(cons courrier,info auteur,prod projet)
    with (courrier-.date_arrivee <= date_cloture)
       , (courrier-.version != 'definitif')
       , (courrier-.nom_auteur == auteur?.nom)
       , (projet+.ref == courrier-.ref)
       , (projet+.nom_auteur == courrier-.nom_auteur)
       , (projet+.date_arrivee == courrier-.date_arrivee)
       , (projet+.version == courrier-.version);

trans deroger_retard(cons courrier,cons derogation,prod projet)
    with (courrier-.date_arrivee > date_cloture)
       , (courrier-.version != 'definitif')
       , (courrier-.ref == derogation-.ref)
       , (derogation-.reponse == 'oui')
       , (projet+.ref == courrier-.ref)
       , (projet+.nom_auteur == courrier-.nom_auteur)
       , (projet+.date_arrivee == courrier-.date_arrivee)
       , (projet+.version == courrier-.version);

trans recevoir_papier_direct(cons projet,prod papier)
    with (projet-.version == 'provisoire')
       , (papier+.ref == projet-.ref)
       , (papier+.nom_auteur == projet-.nom_auteur);

trans recevoir_papier_lettre(cons projet,cons courrier,prod papier)
    with (courrier-.version == 'provisoire')
       , (projet-.version == 'lettre')
       , (courrier-.ref == projet-.ref)
       , (papier+.ref == courrier-.ref)
       , (papier+.nom_auteur == courrier-.nom_auteur);

trans annuler(neg papier,cons projet,cons(*) examinateur)
    with (projet-.version == 'lettre')
       , (current_date > projet-.date_arrivee + 30)
       , (projet-.ref == examinateur(*)-.ref)
       , (projet-.ref == papier~.ref);

trans recevoir_definitif(cons courrier,prod definitif)
    with (courrier-.version == 'definitif')
       , (definitif+.ref == courrier-.ref)
       , (definitif+.nom_auteur == courrier-.nom_auteur)
       , (definitif+.date_arrivee == courrier-.date_arrivee);

trans affecter_examinateur(info papier,info membre,cons affectation,prod examinateur)
    with (affectation-.ref == papier?.ref)
       , (affectation-.nom_examinateur == membre?.nom)
       , (affectation-.nom_examinateur != papier?.nom_auteur)
       , (examinateur+.ref == affectation-.ref)
       , (examinateur+.nom == affectation-.nom_examinateur);

trans reaffecter_examinateur(info fin_notation,neg note,cons examinateur,
                             cons reaffectation,info(*) note,prod reaffectation)
    with (fin_notation?.etat == 'oui')
       , (reaffectation-.ref == examinateur-.ref)
       , (reaffectation-.ancien_examinateur == examinateur-.nom)
       , (examinateur-.ref == note~.ref & examinateur-.nom == note~.nom_examinateur)
       , (note(*)?.ref == examinateur-.ref)
       , (card({note(*)?}) < 3)
       , (reaffectation+.ref == reaffectation-.ref)
       , (reaffectation+.nom_examinateur == reaffectation-.nouveau_examinateur);

trans noter(cons examinateur,cons notation,prod note)
    with (notation-.ref == examinateur-.ref)
       , (notation-.nom_examinateur == examinateur-.nom)
       , (note+.ref == notation-.ref)
       , (note+.nom_examinateur == notation-.nom_examinateur)
       , (note+.note_sujet == notation-.note_sujet)
```

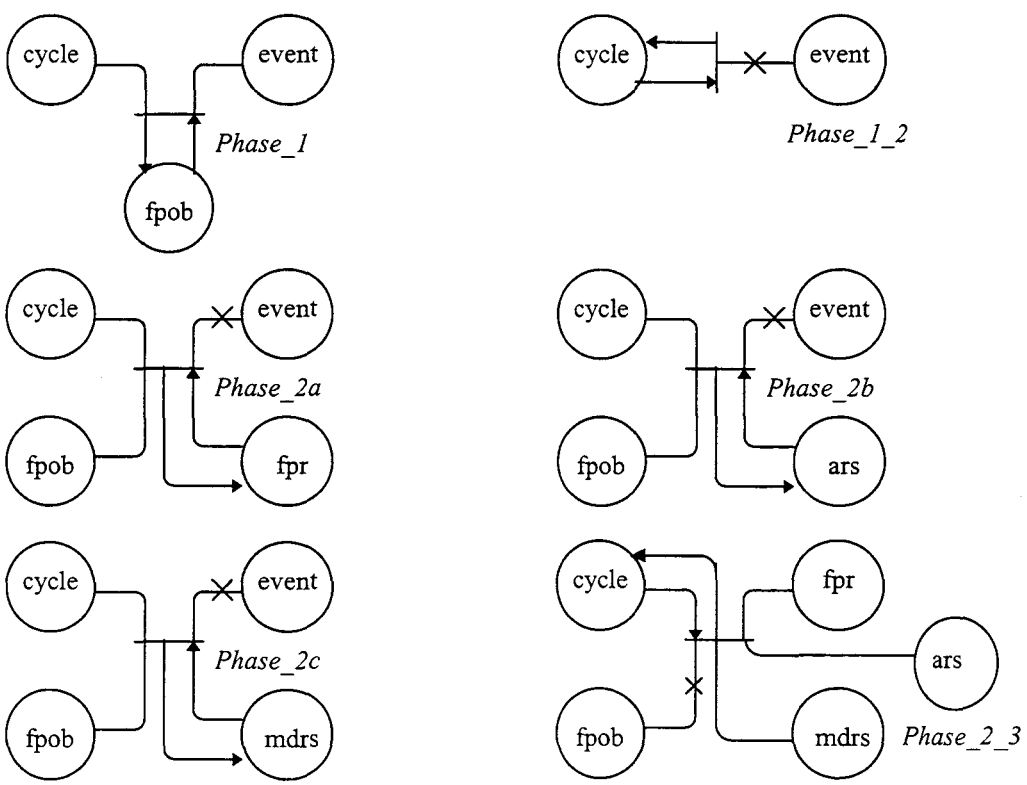
```
       , (note+.note_qualite == notation-.note_qualite)
       , (note+.note_biblio == notation-.note_biblio);

trans calculer_moyenne(cons papier,cons(*) note,info fin_notation,prod moyenne)
    with (fin_notation?.etat == 'oui')
       , (note(*)-.ref == papier-.ref)
       , (card({note(*)-}) >= 3)
       , (moyenne+.ref == papier-.ref)
       , (moyenne+.nom_auteur == papier-.nom_auteur)
       , (moyenne+.valeur == avg({note(*)-.note_sujet}) * coef_sujet)
                           + avg({note(*)-.note_qualite}) * coef_qualite
                           + avg({note(*)-.note_biblio}) * coef_biblio);

trans classer_refus(cons moyenne,prod refus)
    with (moyenne-.valeur < seuil_refuse)
       , (refus+.ref == moyenne-.ref)
       , (refus+.nom_auteur == moyenne-.nom_auteur);

trans classer_acceptation(cons moyenne,prod acceptation)
    with (moyenne-.valeur >= seuil_accepte)
       , (acceptation+.ref == moyenne-.ref)
       , (acceptation+.nom_auteur == moyenne-.nom_auteur);

trans classer_ballottage(cons moyenne,prod ballottage)
    with (moyenne-.valeur >= seuil_refuse)
       , (moyenne-.valeur < seuil_accepte)
       , (ballottage+.ref == moyenne-.ref)
       , (ballottage+.nom_auteur == moyenne-.nom_auteur);

trans traiter_ballottage_accepte(cons ballottage,cons decision,prod acceptation)
    with (ballottage-.ref == decision-.ref)
       , (decision-.accord == 'oui')
       , (acceptation+.ref == ballottage-.ref)
       , (acceptation+.nom_auteur == ballottage-.nom_auteur);

trans traiter_ballottage_refuse(cons ballottage,cons decision,prod refus)
    with (ballottage-.ref == decision-.ref)
       , (decision-.accord == 'non')
       , (refus+.ref == ballottage-.ref)
       , (refus+.nom_auteur == ballottage-.nom_auteur);

trans organiser_session_definitif(cons acceptation,cons definitif,prod session)
    with (acceptation-.ref == definitif-.ref)
       , (definitif-.date_arrivee <= date_retard)
       , (session+.ref == acceptation-.ref)
       , (session+.nom_presentateur == acceptation-.nom_auteur);

trans organiser_session_retard(cons acceptation,cons definitif,cons avis,prod session)
    with (acceptation-.ref == definitif-.ref)
       , (definitif-.date_arrivee > date_retard)
       , (definitif-.ref == avis-.ref)
       , (avis-.accord == 'oui')
       , (session+.ref == acceptation-.ref)
       , (session+.nom_presentateur == acceptation-.nom_auteur);

trans inviter(cons invitation,cons reponse,prod session)
    with (invitation-.ref == reponse-.ref)
       , (reponse-.date_remise <= invitation-.date_limite)
       , (session+.ref == invitation-.ref)
       , (session+.nom_presentateur == invitation-.nom_invite);

# fin des specifications
```

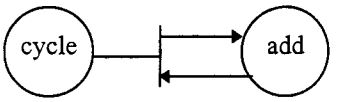
ANNEXE D

L'ARCHITECTURE PIPELINE DE L'IBM360 GRAPHE ET ANNOTATIONS

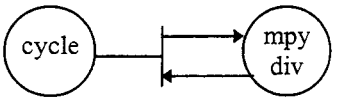
D.1: Graphe du réseau formel

Le graphe du réseau formel, qui comprend 12 places et 28 transitions, est donné en Figure 48. Nous avons préféré le présenter sous une forme éclatée pour faciliter la compréhension, car il est délicat de représenter un réseau d'une telle complexité en un graphe unique (Cf Paragraphe III.1.2.2). Nous nous sommes donc permis cette dérogation car aucune transition, selon notre modélisation, n'a d'interaction ou n'est dépendante d'une autre transition. Les différents sous-graphes élémentaires pourront alors être utilisés comme aide à la lecture des tables d'annotation. Nous retrouvons dans ces sous-graphes les places représentant les différents composants de l'unité d'exécution (*fpos*, *fpr*, *sdr*, *ars*, *mdrs*, *idec*, *add*, *mpydiv*, *cdb*) ainsi que 2 places créées de toutes pièces pour effectuer une simulation (*cycle*, *event*). Les transitions correspondent aux événements internes des différents composants. Les annotations sont disponibles en Table VII et Table VIII. Pour simplifier le problème, notons qu'il n'est fait aucune référence à d'éventuelles contraintes de place ou de clé.

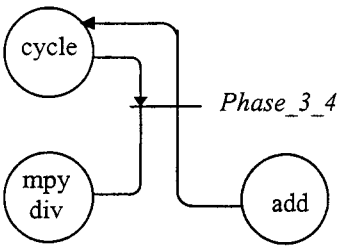


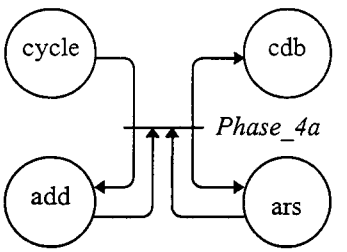


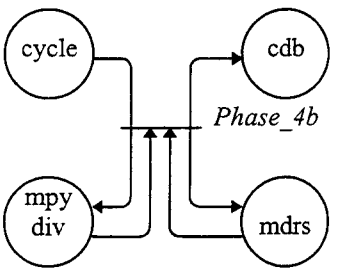
Phase_3a

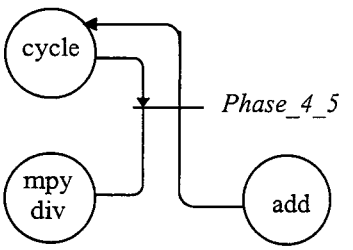


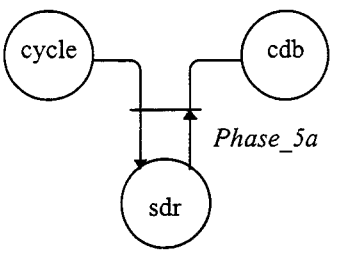
Phase_3b

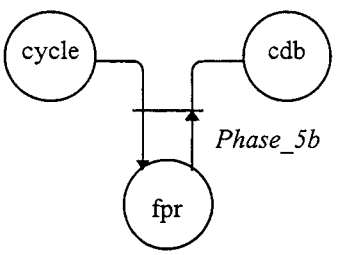


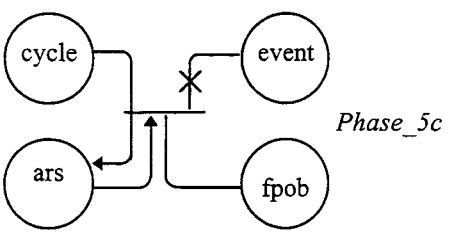


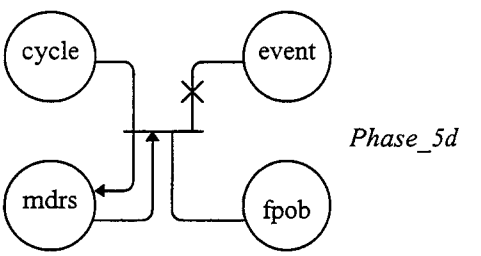


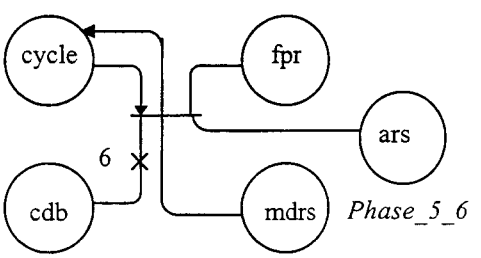


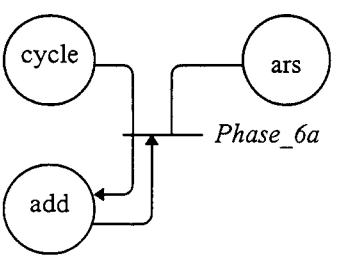












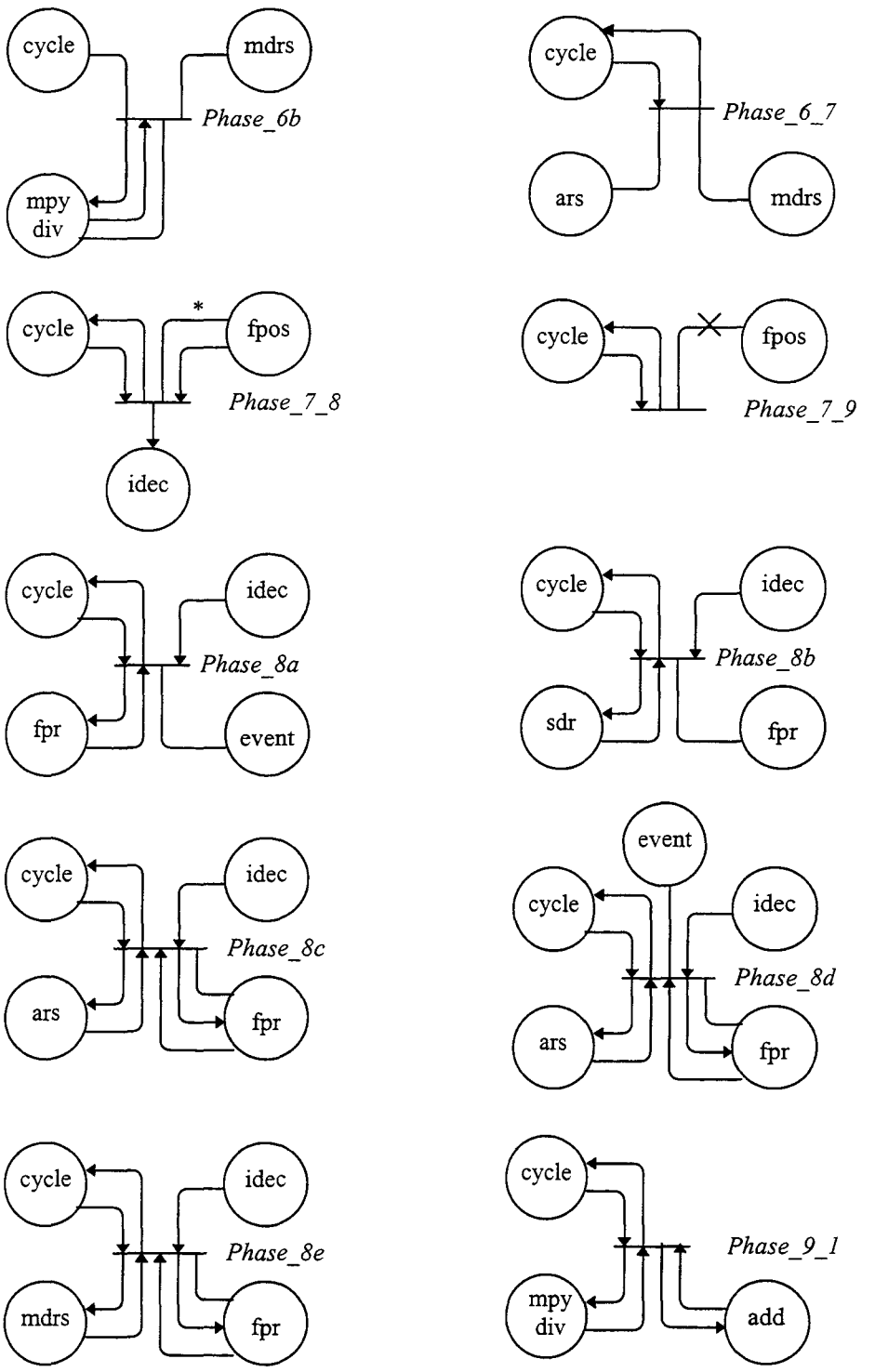


Figure 48: *IBM360* — Graphe du réseau formel

D.2: Annotation des places

PLACE	ATTRIBUTS DES JETONS	COMMENTAIRES
cycle	number: entier	numéro du cycle machine
	phase: entier	numéro de la phase dans le cycle
event	cycle: entier	cycle machine au début duquel arrive l'événement
	varname: chaîne de caractères	nom de la variable provenant de la mémoire centrale
	obid: entier	destination de la variable (numéro du FPOB)
	value: réel	valeur de la variable
fpob	id: entier	numéro du buffer (1..6)
	value: réel	valeur du buffer
fpos	inum: entier	numéro de l'instruction
	mnem: chaîne de caractères	code opération de l'instruction
	left_op: chaîne de caractères	opérande gauche de l'instruction
	right_op: chaîne de caractères	opérande droit de l'instruction
fpr	regname: chaîne de caractères	nom du registre (FPR0..FPR3)
	busy: entier	bit d'occupation (VRAI si le registre attend une valeur, FAUX sinon)
	tag: entier	source de l'opérande (1 à 12 si le registre attend une valeur, 0 sinon)
	value: réel	valeur du registre (si busy = FAUX)
sdr	id: entier	id du registre (13..15)
	busy: entier	bit d'occupation (VRAI si le registre attend une valeur, FAUX sinon)
	tag: entier	source de l'opérande (0 à 3 si le registre attend une valeur, -1 sinon)
	value: réel	valeur du registre (si busy = FAUX)
ars	id: entier	identification de la station (10..12)
	left_tag: entier	source de l'opérande droit
	left_value: réel	valeur de l'opérande droit
	right_tag: entier	source de l'opérande gauche
	right_value: réel	valeur de l'opérande gauche
	need_service: entier	bit de service (FAUX si aucun des opérandes n'est présent, VRAI sinon)
mdrs	id: entier	identification de la station (8..9)
	left_tag: entier	source de l'opérande gauche
	left_value: réel	valeur de l'opérande gauche
	right_tag: entier	source de l'opérande droit
	right_value: réel	valeur de l'opérande droit
	need_service: entier	bit de service (FAUX si aucun des opérandes n'est présent, VRAI sinon)
	mnem: chaîne de caractères	code opération de l'instruction (MPY, DIVIDE)
idec	mnem: chaîne de caractères	code opération de l'instruction
	left_op: chaîne de caractères	opérande gauche de l'instruction
	right_op: chaîne de caractères	opérande droit de l'instruction
adder	stage: entier	étage en cours d'utilisation (0..3)
	tag: entier	identification de la station (10..12)
	freeze: entier	bit de disponibilité
mpydiv	stage: entier	étage en cours d'utilisation (0..4)
	tag: entier	identification de la station (8..9)
	freeze: entier	bit de disponibilité
	mnem: chaîne de caractères	code opération de l'instruction (MPY, DIVIDE)
cdb	id: entier	identification du résultat disponible sur le bus commun (1..12)
	value: réel	valeur du résultat

Table VII: *IBM360* — Annotations des places

D.3: Annotation des transitions

TRANSITION	CONTRAINTES DE TRANSITION
Phase_1 Tester un événement externe (arrivée d'une variable de la mémoire centrale), uniquement en phase 1	(cycle?.phase = 1) (cycle?.number = event-.cycle) (event-.obid = fpob-.id) (fpob+.id = fpob-.id) (fpob+.value = event-.value)
Phase_1_2 Aucun événement externe, passer en phase 2	(cycle-.phase = 1) (cycle-.number = event~.cycle) (cycle+.number = cycle-.number) (cycle+.phase = 2)
Phase_2a Un FPR attend le contenu d'un FPOB	(cycle?.phase = 2) (fpr-.tag = fpob?.id) (fpob?.id = event~.obid) (fpr+.regname = fpr-.regname) (fpr+.busy = FALSE) (fpr+.tag = 0) (fpr+.value = fpob?.value)
Phase_2b Un ARS (gauche ou droit) attend le contenu d'un FPOB	(cycle?.phase = 2) (ars-.left_tag = fpob?.id ∨ ars-.right_tag = fpob?.id) (fpob?.id = event~.obid) (ars+.id = ars-.id) ((ars-.left_tag = fpob?.id ∧ ars+.left_tag = 0) ∨ (ars-.left_tag ≠ fpob?.id ∧ ars+.left_tag = ars-.left_tag)) ((ars-.left_tag = fpob?.id ∧ ars+.left_value = fpob?.value) ∨ (ars-.left_tag ≠ fpob?.id ∧ ars+.left_value = ars-.left_value)) ((ars-.right_tag = fpob?.id ∧ ars+.right_tag = 0) ∨ (ars-.right_tag ≠ fpob?.id ∧ ars+.right_tag = ars-.right_tag)) ((ars-.right_tag = fpob?.id ∧ ars+.right_value = fpob?.value) ∨ (ars-.right_tag ≠ fpob?.id ∧ ars+.right_value = ars-.right_value)) (ars+.need_service = TRUE)
Phase_2c Un MDRS (gauche ou droit) attend le contenu d'un FPOB	(cycle?.phase = 2) (mdrs-.left_tag = fpob?.id ∨ mdrs-.right_tag = fpob?.id) (fpob?.id = event~.obid) (mdrs+.id = mdrs-.id) (mdrs+.mnem = mdrs-.mnem) ((mdrs-.left_tag = fpob?.id ∧ mdrs+.left_tag = 0) ∨ (mdrs-.left_tag ≠ fpob?.id ∧ mdrs+.left_tag = mdrs-.left_tag)) ((mdrs-.left_tag = fpob?.id ∧ mdrs+.left_value = fpob?.value) ∨ (mdrs-.left_tag ≠ fpob?.id ∧ mdrs+.left_value = mdrs-.left_value)) ((mdrs-.right_tag = fpob?.id ∧ mdrs+.right_tag = 0) ∨ (mdrs-.right_tag ≠ fpob?.id ∧ mdrs+.right_tag = mdrs-.right_tag)) ((mdrs-.right_tag = fpob?.id ∧ mdrs+.right_value = fpob?.value) ∨ (mdrs-.right_tag ≠ fpob?.id ∧ mdrs+.right_value = mdrs-.right_value)) (mdrs+.need_service = TRUE)
Phase_2_3 Rien n'attend le contenu d'un FPOB, passer en phase 3	(cycle-.phase = 2) (fpob~.id = fpr?.tag) (fpob~.id = ars?.left_tag) (fpob~.id = ars?.right_tag) (fpob~.id = mdrs?.left_tag) (fpob~.id = mdrs?.right_tag) (cycle+.number = cycle-.number) (cycle+.phase = 3)
Phase_3a Si l'additionneur pipeline est utilisé, passer à l'étage suivant	(cycle?.phase = 3) (adder-.freeze = FALSE) (adder-.stage ≥ 1 ∧ adder-.stage ≤ 2) (adder+.stage = adder-.stage + 1) (adder+.tag = adder-.stage) (adder+.freeze = TRUE)
Phase_3b Si le multiplieur/diviseur pipeline est utilisé, passer à l'étage suivant	(cycle?.phase = 3) (mpydiv-.freeze = FALSE) (mpydiv-.stage ≥ 1 ∧ mpydiv-.stage ≤ 3) (mpydiv+.stage = mpydiv-.stage + 1) (mpydiv+.tag = mpydiv-.stage) (mpydiv+.freeze = TRUE) (mpydiv+.mnem = mpydiv-.mnem)

Phase_3_4 Aucun opérateur pipeline n'est utilisé, passer en phase 4	(cycle-.phase = 3) (adder?.freeze = TRUE ∨ adder?.stage < 1 ∨ adder-.stage > 2) (mpydiv?.mnem = '' ∨ mpydiv?.freeze = TRUE ∨ mpydiv?.stage) (cycle+.number = cycle-.number) (cycle+.phase = 4)
Phase _4a L'opérateur d'addition a produit un résultat	(cycle?.phase = 4) (adder-.stage = 3) (ars-.id = adder-.tag) (adder+.stage = 0) (adder+.tag = 0) (adder+.freeze = FALSE) (cdb+.id = adder-.tag) (cdb+.value = ars-.left_value + ars-.right_value) (ars+.id = ars-.id) (ars+.left_tag = ars-.left_tag) (ars+.left_value = ars-.left_value) (ars+.right_tag = ars-.right_tag) (ars+.right_value = ars-.right_value) (ars+.need_service = FALSE)
Phase _4b L'opérateur de multiplication/division a produit un résultat	(cycle?.phase = 4) (mpydiv-.stage = 4) (mdrs-.id = mpydiv-.tag) (mpydiv+.stage = 0) (mpydiv+.tag = 0) (mpydiv+.freeze = FALSE) (mpydiv+.mnem = '') (cdb+.id = mpydiv-.tag) ((mpydiv-.mnem = 'MPY' ∧ cdb+.value = mdrs-.left_value × mdrs-.right_value) ∨ (mpydiv-.mnem = 'DIVIDE' ∧ cdb+.value = mdrs-.left_value ÷ mdrs-.right_value)) (mdrs+.id = mdrs-.id) (mdrs+.mnem = mdrs-.mnem) (mdrs+.left_tag = mdrs-.left_tag) (mdrs+.left_value = mdrs-.left_value) (mdrs+.right_tag = mdrs-.right_tag) (mdrs+.right_value = mdrs-.right_value) (mdrs+.need_service = FALSE)
Phase_4_5 Aucun pipeline n'a produit de résultat, passer en phase 5	(cycle-.phase = 4) (adder?.stage ≠ 3) (mpydiv?.mnem = '' ∨ mpydiv?.stage ≠ 4) (cycle+.number = cycle-.number) (cycle+.phase = 5)
Phase _5a Un SDR attend un résultat sur le CDB	(cycle?.phase = 5) (sdr-.tag = cdb?.id) (sdr+.id = sdr-.id) (sdr+.value = cdb?.value) (sdr+.tag = 0) (sdr+.busy = FALSE)
Phase _5b Un FPR attend un résultat sur le CDB	(cycle?.phase = 5) (fpr-.tag = cdb?.id) (fpr+.regname = fpr-.regname) (fpr+.value = cdb?.value) (fpr+.tag = 0) (fpr+.busy = FALSE)
Phase_5c Un ARS gauche ou droit attend un résultat sur le CDB	(cycle?.phase = 5) (ars-.left_tag = cdb?.id ∨ ars-.right_tag = cdb?.id) (ars+.id = ars-.id) ((ars-.left_tag = cdb?.id ∧ ars+.left_value = cdb?.value) ∨ (ars-.left_tag ≠ cdb?.id ∧ ars+.left_value = ars-.left_value)) ((ars-.left_tag = cdb?.id ∧ ars+.left_tag = 0) ∨ (ars-.left_tag ≠ cdb?.id ∧ ars+.left_tag = ars-.left_tag)) ((ars-.right_tag = cdb?.id ∧ ars+.right_value = cdb?.value) ∨ (ars-.right_tag ≠ cdb?.id ∧ ars+.right_value = ars-.right_value)) ((ars-.right_tag = cdb?.id ∧ ars+.right_tag = 0) ∨ (ars-.right_tag ≠ cdb?.id ∧ ars+.right_tag = ars-.right_tag)) (ars+.need_service = TRUE)
Phase_5d Un MDRS gauche ou droit attend un résultat sur le CDB	(cycle?.phase = 5) (mdrs-.left_tag = cdb?.id ∨ mdrs-.right_tag = cdb?.id) (mdrs+.id = mdrs-.id) (mdrs+.mnem = mdrs-.mnem)

	((mdrs-.left_tag = cdb?.id ∧ mdrs+.left_value = cdb?.value) ∨ (mdrs-.left_tag ≠ cdb?.id ∧ mdrs+.left_value = mdrs-.left_value)) ((mdrs-.left_tag = cdb?.id ∧ mdrs+.left_tag = 0) ∨ (mdrs-.left_tag ≠ cdb?.id ∧ mdrs+.left_tag = mdrs-.left_tag)) ((mdrs-.right_tag = cdb?.id ∧ mdrs+.right_value = cdb?.value) ∨ (mdrs-.right_tag ≠ cdb?.id ∧ mdrs+.right_value = mdrs-.right_value)) ((mdrs-.right_tag = cdb?.id ∧ mdrs+.right_tag = 0) ∨ (mdrs-.right_tag ≠ cdb?.id ∧ mdrs+.right_tag = mdrs-.right_tag)) (mdrs+.need_service = TRUE)
Phase_5_6 Aucune station n'attend un résultat du CDB, passer en phase 6	(cycle-.phase = 5) (cdb(1)~.id = sdr?.tag) (cdb(2)~.id = fpr?.tag) (cdb(3)~.id = ars?.left_tag) (cdb(4)~.id = ars?.right_tag) (cdb(5)~.id = mdrs?.left_tag) (cdb(6)~.id = mdrs?.right_tag) (cdb(*)-.id ≠ 0) (cycle+.number = cycle-.number) (cycle+.phase = 6)
Phase_6a Le pipeline d'addition est requis	(cycle?.phase = 6) (ars?.left_tag = 0 ∧ ars?.right_tag = 0) (ars?.need_service = TRUE) (adder-.freeze = FALSE) (adder-.stage = 0) (adder+.stage = 1) (adder+.freeze = TRUE) (adder+.tag = ars?.id)
Phase_6b Le pipeline de multiplication/division est requis	(cycle?.phase = 6) (mdrs?.left_tag = 0 ∧ mdrs?.right_tag = 0) (mdrs?.need_service = TRUE) (mpydiv-.freeze = FALSE) (mpydiv-.stage = 0) (mpydiv+.stage = 1) (mpydiv+.freeze = TRUE) (mpydiv+.tag = mdrs?.id) (mpydiv+.mnem = mdrs?.mnem)
Phase_6_7 Aucun pipeline n'est requis, passer en phase 7	(cycle-.phase = 6) (ars?.need_service = FALSE ∨ ars?.left_tag ≠ 0 ∨ ars?.right_tag ≠ 0) (mdrs?.need_service = FALSE ∨ mdrs?.left_tag ≠ 0 ∨ mdrs?.right_tag ≠ 0) (cycle+.number = cycle-.number) (cycle+.phase = 7)
Phase_7_8 Rechercher une instruction dans FPOS	(cycle-.phase = 7) (fpos-.inum = MIN (fpos(*)?.inum)) (idec+.mnem = fpos-.mnem) (idec+.left_op = fpos-.left_op) (idec+.right_op = fpos-.right_op) (cycle+.number = cycle-.number) (cycle+.phase = 8)
Phase_7_9 Plus d'instruction à exécuter	(cycle-.phase = 7) (fpos~.inum > 0) (cycle+.number = cycle-.number) (cycle+.phase = 9)
Phase_8a Décoder l'instruction LOAD reg,mem	(cycle-.phase = 8) (idec-.mnem = 'LOAD') (idec-.left_op = fpr-.regname) (idec-.right_op = event~.varname) (fpr+.regname = fpr-.regname) (fpr+.busy = TRUE) (fpr+.tag = event?.obid) (fpr+.value = fpr-.value) (cycle+.number = cycle-.number) (cycle+.phase = 9)
Phase_8b Décoder l'instruction STORE mem,reg	(cycle-.phase = 8) (idec-.mnem = 'STORE') (idec-.right_op = fpr?.regname) (sdr-.busy = FALSE) (sdr+.id = sdr-.id) (sdr+.busy = TRUE)

	(sdr+.tag = fpr?.tag) ((fpr?.busy = FALSE ∧ sdr+.value = fpr?.value) ∨ (fpr?.busy = TRUE ∧ sdr+.value = sdr-.value)) (cycle+.number = cycle-.number) (cycle+.phase = 9)
Phase_8c Décoder l'instruction ADD reg,reg	(cycle-.phase = 8) (idec-.mnem = 'ADD') (idec-.left_op = fpr-.regname) (idec-.right_op = fpr?.regname) (ars-.left_tag < 0 ∧ ars-.right_tag < 0) (ars+.id = ars-.id) (ars+.left_tag = fpr-.tag) ((fpr-.busy = FALSE ∧ ars+.left_value = fpr-.value) ∨ (fpr-.busy = TRUE ∧ ars+.left_value = ars-.left_value)) (ars+.right_tag = fpr?.right_tag) ((fpr?.busy = FALSE ∧ ars+.right_value = fpr?.value) ∨ (fpr?.busy = TRUE ∧ ars+.right_value = ars-.right_value)) (ars+.need_service = TRUE) (fpr+.regname = fpr-.regname) (fpr+.busy = TRUE) (fpr+.tag = ars-.id) (fpr+.value = fpr-.value) (cycle+.number = cycle-.number) (cycle+.phase = 9)
Phase_8d Décoder l'instruction ADD reg,mem	(cycle-.phase = 8) (idec-.mnem = 'ADD') (idec-.left_op = fpr-.regname) (idec-.right_op = event?.varname) (ars-.left_tag < 0 ∧ ars-.right_tag < 0) (ars+.id = ars-.id) (ars+.left_tag = fpr-.tag) ((fpr-.busy = FALSE ∧ ars+.left_value = fpr-.value) ∨ (fpr-.busy = TRUE ∧ ars+.left_value = ars-.left_value)) (ars+.right_tag = event?.obid) (ars+.right_value = ars-.right_value) (ars+.need_service = TRUE) (fpr+.regname = fpr-.regname) (fpr+.busy = TRUE) (fpr+.tag = ars-.id) (fpr+.value = fpr-.value) (cycle+.number = cycle-.number) (cycle+.phase = 9)
Phase_8e Décoder l'instruction MPY/DIVIDE reg,reg	(cycle-.phase = 8) (idec-.mnem = 'MPY' ∨ idec-.mnem = 'DIVIDE') (idec-.left_op = fpr-.regname) (idec-.right_op = fpr?.regname) (mdrs-.left_tag < 0 ∧ mdrs-.right_tag < 0) (mdrs+.id = mdrs-.id) (mdrs+.mnem = idec-.mnem (mdrs+.left_tag = fpr-.tag) ((fpr-.busy = FALSE ∧ mdrs+.left_value = fpr-.value) ∨ (fpr-.busy = TRUE ∧ mdrs+.left_value = mdrs-.left_value)) (mdrs+.right_tag = fpr?.right_tag) ((fpr-.busy = FALSE ∧ mdrs+.right_value = fpr-.value) ∨ (fpr-.busy = TRUE ∧ mdrs+.right_value = mdrs-.right_value)) (mdrs+.need_service = TRUE) (fpr+.regname = fpr-.regname) (fpr+.busy = TRUE) (fpr+.tag = mdrs-.id) (fpr+.value = fpr-.value) (cycle+.number = cycle-.number) (cycle+.phase = 9)
Phase 9_1 Passage au cycle machine suivant	(cycle-.phase = 9) (adder+.freeze = 0) (adder+.stage = adder-.stage) (adder+.tag = adder-.tag) (mpydiv+.freeze = 0) (mpydiv+.stage = mpydiv-.stage) (mpydiv+.tag = mpydiv-.tag) (mpydiv+.mnem = mpydiv-.mnem)

	(cycle+.number = cycle-.number + 1) (cycle+.phase = 1)

Table VIII: *IBM360* — Annotation des transitions

ANNEXE E

CODE SOURCE DU STEAM-BOILER

```
# Steam-Boiler Control Specification.

# Definitions

const int   CLOSED          = 0 ,            # pompe ouverte
            OPEN            = 1 ;            # pompe fermée

const int   STOPPED         = OFF ,          # pompe arrêtée
            POURING         = ON ;           # pompe débite

const string[9]
            INIT_MODE       = 'INIT'      ,  # mode INITIALISATION
            NORMAL_MODE     = 'NORMAL'    ,  # mode NORMAL
            DEGRADED_MODE   = 'DEGRADED'  ,  # mode DEGRADE
            RESCUE_MODE     = 'RESCUE'    ,  # mode SECOURS
            EMERGENCY_MODE  = 'EMERGENCY' ;  # mode URGENCE

const real  RATIO           = 0.1 ;          # valeur expansion vapeur (?)

# messages: chaque message possède un nom; les commentaires indiquent le ou les paramètres
#           des messages

const string[22]
            STOP                    = 'STOP' ,
            STEAM_BOILER_WAIT       = 'STEAM_BOILER_WAIT' ,
            PHYSICAL_UNITS_RDY      = 'PHYSICAL_UNITS_RDY' ,
            PUMP_STATE              = 'PUMP_STATE' ,         # pompe, état
            PUMP_CTRL_STATE         = 'PUMP_CTRL_STATE' ,    # pompe, état
            LEVEL                   = 'LEVEL' ,              # niveau d'eau
            STEAM                   = 'STEAM' ,              # vapeur de vapeur
            PUMP_REPAIRED           = 'PUMP_REPAIRED' ,      # pompe
            PUMP_CTRL_REPAIRED      = 'PUMP_CTRL_REPAIRED' ,# pompe
            LEVEL_REPAIRED          = 'LEVEL_REPAIRED' ,
            STEAM_REPAIRED          = 'STEAM_REPAIRED' ,
            PUMP_FAIL_ACK           = 'PUMP_FAIL_ACK' ,      # pompe
            PUMP_CTRL_FAIL_ACK      = 'PUMP_CTRL_FAIL_ACK' ,# pompe
            LEVEL_FAIL_ACK          = 'LEVEL_FAIL_ACK' ,
            STEAM_FAIL_ACK          = 'STEAM_FAIL_ACK' ,
            MODE                    = 'MODE' ,               # mode
            PROGRAM_RDY             = 'PROGRAM_RDY' ,
            VALVE                   = 'VALVE' ,
            OPEN_PUMP               = 'OPEN_PUMP' ,          # pompe
            CLOSE_PUMP              = 'CLOSE_PUMP' ,         # pompe
            PUMP_FAIL_DET           = 'PUMP_FAIL_DET' ,      # pompe
            PUMP_CTRL_FAIL_DET      = 'PUMP_CTRL_FAIL_DET' ,# pompe
            LEVEL_FAIL_DET          = 'LEVEL_FAIL_DET' ,
            STEAM_FAIL_DET          = 'STEAM_FAIL_DET' ,
            PUMP_REPAIRED_ACK       = 'PUMP_REPAIRED_ACK' , # pompe
            PUMP_CTRL_REPAIRED_ACK  = 'PUMP_CTRL_REPAIRED_ACK' , # pompe
            LEVEL_REPAIRED_ACK      = 'LEVEL_REPAIRED_ACK' ,
            STEAM_REPAIRED_ACK      = 'STEAM_REPAIRED_ACK' ;
```

```
##############################################################################
# Place OPERA:
#       La place OPERA représente l'interface entre l'utilisateur, qui tient le rôle
#       de l'opérateur, et le système. Lorsque l'opérateur émet un message à destination
#       du système, il le produit dans cette place. Le système envoie des messages à
#       destination de l'opérateur dans cette place, et l'opérateur doit les consommer
#       lui-même.
#           Liste possible des messages émis par l'opérateur:
#               STEAM_BOILER_WAIT           démarrage du systeme
#               PUMP_REPAIRED(n)            réparation d'une pompe
#               PUMP_CTRL_REPAIRED(n)       réparation d'un contrôleur de pompe
#               LEVEL_REPAIRED              réparation de l'unité de niveau d'eau
#               STEAM_REPAIRED              réparation de l'unité de niveau vapeur
#           Liste possible des messages reçus par l'opérateur:
#               PUMP_REPAIRED_ACK(n)        acquittement de réparation d'une pompe
#               PUMP_CTRL_REPAIRED_ACK(n)   acquittement de réparation d'un contrôleur
#               LEVEL_REPAIRED_ACK          acquittement de réparation de l'unité d'eau
#               STEAM_REPAIRED_ACK          acquittement de réparation de l'unité de vapeur
#               STOP                        arrêt d'urgence demandé par le système

bag Opera ? {

        string[22]  Msg ;           # message
        int         ^Param ;        # paramètre optionnel
    } ;

# Place MBOX:
#       comme il est dit dans les spécifications que les durées de transmission des
#       messages sont négligeables (i.e. nulles), plutôt que de dispatcher les messages
#       vers les différentes unités, on préfère les concentrer dans une même structure: la
#       boîte à messages; toutes les unités sont capables d'interroger, de consommer, et de
#       produire dans la boîte; chaque message est composé de son type et de 4 paramètres
#       optionnels.
#           Liste possible des messages émis par l'opérateur:
#               STEAM_BOILER_WAIT
#               PUMP_REPAIRED(n)
#               PUMP_CTRL_REPAIRED(n)
#               LEVEL_REPAIRED
#               STEAM_REPAIRED
#           Liste possible des messages émis par les unités:
#               PHYSICAL_UNITS_RDY          émulation automatique, réponse à PROGRAM_RDY
#               PUMP_STATE(n)               état d'une pompe
#               PUMP_CTRL_STATE(n)          état d'un contrôleur de pompe
#               LEVEL(v)                    niveau d'eau
#               STEAM(v)                    niveau de vapeur
#               STEAM_FAIL_DET              détection panne niveau de vapeur
#               LEVEL_FAIL_DET              détection panne niveau d'eau
#               PUMP_FAIL_DET(n)            détection panne pompe
#               PUMP_CTRL_FAIL_DET(n)       détection panne contrôleur pompe
#           Liste possible des messages reçus par les unités:
#               MODE(m)                     état du système
#               PROGRAM_RDY                 progamme prêt après initialisation
#               VALVE                       commande de vidange
#               OPEN_PUMP(n)                ouverture d'une pompe
#               CLOSE_PUMP(n)               fermeture d'une pompe

bag Mbox ? {

        string[22]  Msg ;           # message
        int         ^Param1 ;       # 1er paramètre optionnel
        int         ^Param2 ;       # 2ème paramètre optionnel
        real        ^Param3 ;       # 3ème paramètre optionnel
        string[9]   ^Param4 ;       # 4ème paramètre optionnel
    } ;

# Place BOILER:
#       caractéristiques de la chaudière: ces caractéristiques peuvent être employées par
#       toutes les unités, et aussi par l'opérateur; on remarque que le niveau d'eau n'en
#       fait pas partie, puisqu'il est variable; si une unité veut consulter le niveau
#       d'eau, elle devra le faire par l'intermédiaire du message LEVEL de la place MBOX.

bag Boiler ? {

        real    WMaxCap ;       # capacité totale (C)
        real    WMinLim ;       # limite minimale de quantité d'eau (M1)
```

```
        real    WMaxLim ;       # limite maximale de quantité d'eau (M2)
        real    WMinNorm ;      # limite minimale normale de quantité d'eau (N1)
        real    WMaxNorm ;      # limite maximale normale de quantité d'eau (N2)
        real    SMaxQty ;       # quantité maximale de vapeur (W)
        real    SMaxGradInc ;   # gradient maximal d'augmentation (U1)
        real    SMaxGradDec ;   # gradient minimal de diminuation (U2)

    } with  ( 0.0 < WMinLim )
        &   ( WMinLim < WMinNorm )
        &   ( WMinNorm < WMaxNorm )
        &   ( WMaxNorm < WMaxLim )
        &   ( WMaxLim < WMaxCap ) ;

# Place WATERMEASDEV:
#       Unité de mesure de la quantité d'eau dans la chaudière.

bag WaterMeasDev ? {

        real    WQuantity ;     # quantité d'eau (q)

    } with  ( WQuantity >= 0.0 )
;


# Place STEAMMEASDEV:
#       Unité de mesure de la quantité de vapeur sortant de la chaudière.

bag SteamMeasDev ? {

        real    SQuantity ;     # quantité de vapeur (v)

    } with  ( SQuantity >= 0.0 )
;

# Place PUMP:
#       Caractéristiques et état de chacune des 4 pompes.

bag Pump ? {

        key int PNum ;          # numéro de la pompe
        real    PMaxCap ;       # capacité (P)
        int     PFuncMode ;     # mode de fonctionnement (OPEN/CLOSED)

    } with  ( PNum >= 1 & PNum <= 4 )
        &   ( PMaxCap >= 0.0 )
        &   ( PFuncMode == OPEN | PFuncMode == CLOSED )
;

# Place PUMPCTRLDEV:
#       Unités de contrôle des 4 pompes.

bag PumpCtrlDev ? {

        key int CNum ;          # numéro du contrôleur
        int     CWater ;        # l'eau circule ou ne circule pas

    } with  ( CNum >= 1 & CNum <= 4 )
        &   ( CWater == POURING | CWater == STOPPED )
;

##############################################################################

trans DISPATCH_Steam_Boiler_Wait
        (
            cons    Opera ,
            prod    MBox
        )
    with
        # l'opérateur a produit le message STEAM_BOILER_WAIT pour lancer le système: il
        # faut le recopier dans la boîte à messages.
            ( Opera-.Msg == STEAM_BOILER_WAIT )
        ,   ( MBox+.Msg == STEAM_BOILER_WAIT )
;

trans DISPATCH_Repaired
```

```
        (
            cons    Opera ,
            prod    MBox
        )
    with
        # l'opérateur a produit les messages *_REPAIRED pour indiquer que les unités en
        # panne ont été réparées: il faut les recopier dans la boîte à messages.
            ( Opera-.Msg == LEVEL_REPAIRED
                | Opera-.Msg == STEAM_REPAIRED
                | Opera-.Msg == PUMP_REPAIRED
                | Opera-.Msg == PUMP_CTRL_REPAIRED )
        ,   ( MBox+.Msg == Opera-.Msg )
        ,   ( Opera-.Msg == PUMP_REPAIRED & MBox+.Param1 == Opera-.Param )
            |
            ( Opera-.Msg == PUMP_CTRL_REPAIRED & MBox+.Param1 == Opera-.Param )
;

trans UNITMSG_Create_Mode_Msg
        (
            neg     MBox ,
            prod    MBox
        )
    with
        # à l'initialisation du système, le message MODE doit être créé.
            ( MBox~.Msg == MODE )
        ,   ( MBox+.Msg == MODE )
        ,   ( MBox+.Param4 == INIT_MODE )
;

trans UNITMSG_Create_Level_Msg
        (
            neg     MBox ,
            prod    MBox
        )
    with
        # à l'initialisation du système, le message LEVEL doit être créé.
            ( MBox~.Msg == LEVEL )
        ,   ( MBox+.Msg == LEVEL )
        ,   ( MBox+.Param3 == 0.0 )
;

trans UNITMSG_Update_Level_Msg
        (
            cons    MBox ,
            prod    MBox ,
            info    WaterMeasDev
        )
    with
        # en cours de fonctionnement, le paramètre du message LEVEL doit être mis à
        # jour s'il ne correspond pas au relevé de l'unité de mesure de niveau d'eau.
            ( MBox-.Msg == LEVEL )
        ,   ( MBox-.Param3 != WaterMeasDev?.Wquantity )
        ,   ( MBox+.Msg == LEVEL )
        ,   ( MBox+.Param3 == WaterMeasDev?.WQuantity )
;

trans UNITMSG_Create_Steam_Msg
        (
            neg     MBox ,
            prod    MBox
        )
    with
        # à l'initialisation du système, le message STEAM doit être créé.
            ( MBox~.Msg == STEAM )
        ,   ( MBox+.Msg == STEAM )
        ,   ( MBox+.Param3 == 0.0 )
;

trans UNITMSG_Update_Steam_Msg
        (
            cons    MBox ,
            prod    MBox ,
            info    SteamMeasDev
        )
    with
```

```
        # en cours de fonctionnement, le paramètre du message STEAM doit être mis à
        # jour s'il ne correspond pas au relevé de l'unité de mesure de vapeur.
            ( MBox-.Msg == STEAM )
        ,   ( MBox-.Param3 != SteamMeasDev?.Squantity )
        ,   ( MBox+.Msg == STEAM )
        ,   ( MBox+.Param3 == SteamMeasDev?.SQuantity )
;

trans UNITMSG_PumpState_Open_Msg
        (
            cons    MBox ,
            cons    Pump ,
            prod    Pump
        )
    with
        # si une pompe reçoit le message OPEN_PUMP, l'état de la pompe est mis à jour.
            ( MBox-.Msg == OPEN_PUMP )
        ,   ( MBox-.Param1 == Pump-.PNum )
        ,   ( Pump+.PNum == Pump-.PNum )
        ,   ( Pump+.PFuncMode == OPEN )
        ,   ( Pump+.PMaxCap == Pump-.PMaxCap )
;

trans UNITMSG_PumpState_Close_Msg
        (
            cons    MBox ,
            cons    Pump ,
            prod    Pump
        )
    with
        # si une pompe reçoit le message CLOSE_PUMP, l'état de la pompe est mis à jour.
            ( MBox-.Msg == CLOSE_PUMP )
        ,   ( MBox-.Param1 == Pump-.PNum )
        ,   ( Pump+.PNum == Pump-.PNum )
        ,   ( Pump+.PFuncMode == CLOSED )
        ,   ( Pump+.PMaxCap == Pump-.PMaxCap )
;

trans UNITMSG_Create_PumpState_Msg
        (
            neg     MBox ,
            prod    MBox ,
            info    Pump
        )
    with
        # à l'initialisation du système, les messages PUMP_STATE relatifs aux quatre
        # pompes doivent être créés.
            ( MBox~.Msg == PUMP_STATE & MBox~.Param1 == Pump?.PNum )
        ,   ( MBox+.Msg == PUMP_STATE )
        ,   ( MBox+.Param1 == Pump?.PNum )
        ,   ( MBox+.Param2 == Pump?.PFuncMode )
;

trans UNITMSG_Create_PumpCtrlState_Msg
        (
            neg     MBox ,
            prod    MBox ,
            info    PumpCtrlDev
        )
    with
        # à l'initialisation du système, les messages PUMP_CTRL_STATE relatifs aux
        # quatre contrôleurs de pompe doivent être créés.
            ( MBox~.Msg == PUMP_CTRL_STATE & MBox~.Param1 == PumpCtrlDev?.CNum )
        ,   ( MBox+.Msg == PUMP_CTRL_STATE )
        ,   ( MBox+.Param1 == PumpCtrlDev?.CNum )
        ,   ( MBox+.Param2 == PumpCtrlDev?.CWater )
;

trans UNITMSG_Create_PhysRdy_Msg
        (
            info    MBox ,
            neg     MBox ,
            prod    MBox
        )
    with
```

```
        # lorsque le système a terminé son initialisation, le message PROGRAM_RDY
        # est émis vers toutes les unités, et celles-ci doivent répondre par le
        # message PHYSICAL_UNIT_RDY; en l'absence de spécifications plus précises, on
        # a décidé de simuler automatiquement la réponse des unités.
            ( MBox?.Msg == PROGRAM_RDY )
        ,   ( MBox~.Msg == PHYSICAL_UNITS_RDY )
        ,   ( MBox+.Msg == PHYSICAL_UNITS_RDY )
;

trans CONTROL_Init_Check_No_Steam
        (
            cons(2) MBox ,
            info    MBox ,
            prod    MBox
        )
    with
        # Mode INIT: si le message STEAM envoyé par l'unité de mesure de vapeur indique
        # qu'il y a de la vapeur (ce qui est anormal puisque qu'il n'y a pas d'eau dans la
        # chaudière), passer en mode EMERGENCY.
            ( MBox(1)-.Msg == MODE )
        ,   ( MBox(1)-.Param4 == INIT_MODE )
        ,   ( MBox(2)-.Msg == STEAM_BOILER_WAIT )
        ,   ( MBox?.Msg == STEAM )
        ,   ( MBox?.Param3 != 0.0 )
        ,   ( MBox+.Msg == MODE )
        ,   ( MBox+.Param4 == EMERGENCY_MODE )
;

trans CONTROL_Check_Level_Failure
        (
            cons(3) MBox ,
            prod(2) MBox
        )
    with
        # Mode INIT: si l'unité de mesure d'eau est en panne, passer en mode EMERGENCY,
        # et émettre le message d'acquittement.
            ( MBox(1)-.Msg == MODE )
        ,   ( MBox(1)-.Param4 == INIT_MODE )
        ,   ( MBox(2)-.Msg == STEAM_BOILER_WAIT )
        ,   ( MBox(3)-.Msg == LEVEL_FAIL_DET )
        ,   ( Mbox(1)+.Msg == MODE )
        ,   ( MBox(1)+.Param4 == EMERGENCY_MODE )
        ,   ( MBox(2)+.Msg == LEVEL_FAIL_ACK )
;

trans CONTROL_Init_Too_Much_Water
        (
            info(3) MBox ,
            info    Boiler ,
            neg     MBox ,
            prod    MBox
        )
    with
        # Mode INIT: si le message LEVEL envoyé par l'unité de mesure du niveau d'eau
        # indique qu'il y a trop d'eau, envoyer le message VALVE pour vidanger la
        # chaudière.
            ( MBox(1)?.Msg == MODE )
        ,   ( MBox(1)?.Param4 == INIT_MODE )
        ,   ( MBox(2)?.Msg == STEAM_BOILER_WAIT )
        ,   ( MBox(3)?.Msg == LEVEL )
        ,   ( MBox(3)?.Param3 > Boiler?.WMaxNorm )
        ,   ( MBox~.Msg == VALVE )
        ,   ( MBox+.Msg == VALVE )
;

trans CONTROL_Init_Not_Enough_Water
        (
            info    Boiler ,
            info    Pump ,
            info(3) MBox ,
            neg(2)  MBox ,
            prod    MBox
        )
    with
        # Mode INIT: si le message LEVEL envoyé par l'unité de mesure du niveau d'eau
```

```
        # indique qu'il n'y a pas assez d'eau, envoyer un message OPEN_PUMP à
        # destination d'une pompe en état de fonctionner (i.e. il n'y a pas de message
        # PUMP_FAIL_DET sur cette pompe), et si cette pompe ne débite pas déjà.
            ( MBox(1)?.Msg == MODE )
        ,   ( MBox(1)?.Param4 == INIT_MODE )
        ,   ( MBox(2)?.Msg == STEAM_BOILER_WAIT )
        ,   ( MBox(3)?.Msg == LEVEL )
        ,   ( MBox(3)?.Param3 < Boiler?.WMinNorm )
        ,   ( MBox(1)~.Msg == PUMP_FAIL_DET & MBox(1)~.Param1 == Pump?.PNum )
        ,   ( MBox(2)~.Msg == PUMP_STATE & MBox(2)~.Param1 == Pump?.PNum
                                         & MBox(2)~.Param2 == OPEN )
        ,   ( MBox+.Msg == OPEN_PUMP )
        ,   ( MBox+.Param1 == Pump?.PNum )
;

trans CONTROL_Init_Water_OK
        (
            info    Boiler ,
            info(3) MBox ,
            neg     MBox ,
            prod    MBox
        )
    with
        # Mode INIT: dès que le message LEVEL indique une quantité d'eau adéquate
        # indiquer aux autres unités que le programme est prêt.
            ( MBox(1)?.Msg == MODE )
        ,   ( MBox(1)?.Param4 == INIT_MODE )
        ,   ( MBox(2)?.Msg == STEAM_BOILER_WAIT )
        ,   ( MBox(3)?.Msg == LEVEL )
        ,   ( MBox(3)?.Param3 >= Boiler?.WMinNorm )
        ,   ( MBox(3)?.Param3 <= Boiler?.WMaxNorm )
        ,   ( MBox~.Msg == PROGRAM_RDY )
        ,   ( MBox+.Msg == PROGRAM_RDY )
;

trans CONTROL_Init_Unit_Failure
        (
            cons(4) MBox ,
            prod(2) MBox
        )
    with
        # Mode INIT: sur réception du message PHYSICAL_UNITS_RDY, passer en mode
        # DEGRADED si une détection de dysfonctionnement est présente sur l'une des unités
        # (sauf LEVEL_FAIL_DET déjà pris en compte plus haut), et émettre les messages
        # d'acquittement automatiquement pour ne pas surcharger l'opérateur.
            ( MBox(1)-.Msg == MODE )
        ,   ( MBox(1)-.Param4 == INIT_MODE )
        ,   ( MBox(2)-.Msg == STEAM_BOILER_WAIT )
        ,   ( MBox(3)-.Msg == PHYSICAL_UNITS_RDY )
        ,   ( MBox(4)-.Msg == PUMP_FAIL_DET
                | MBox(4)-.Msg == PUMP_CTRL_FAIL_DET
                | MBox(4)-.Msg == STEAM_FAIL_DET )
        ,   ( MBox(4)-.Msg == PUMP_FAIL_DET
                                & MBox(1)+.Msg == PUMP_FAIL_ACK )
          |
            ( MBox(4)-.Msg == PUMP_CTRL_FAIL_DET
                                & MBox(1)+.Msg == PUMP_CTRL_FAIL_ACK )
          |
            ( MBox(4)-.Msg == STEAM_FAIL_DET
                                & MBox(1)+.Msg == STEAM_FAIL_ACK )
        ,   ( MBox(4)-.Msg == PUMP_FAIL_DET
                                & MBox(1)+.Param1 == MBox(4)-.Param1 )
          |
            ( MBox(4)-.Msg == PUMP_CTRL_FAIL_DET
                                & MBox(1)+.Param1 == MBox(4)-.Param1 )
        ,   ( MBox(2)+.Msg == MODE )
        ,   ( MBox(2)+.Param4 == DEGRADED_MODE )
;

trans CONTROL_Init_To_Normal
        (
            cons(2) MBox ,
            info    MBox ,
            prod    MBox
        )
```

```
    with
        # Mode INIT: si tout est OK, passer en mode NORMAL.
            ( MBox(1)-.Msg == MODE )
        ,   ( MBox(1)-.Param4 == INIT_MODE )
        ,   ( MBox(2)-.Msg == STEAM_BOILER_WAIT )
        ,   ( MBox?.Msg == PHYSICAL_UNITS_RDY )
        ,   ( MBox+.Msg == MODE )
        ,   ( MBox+.Param4 == NORMAL_MODE )
;

trans CONTROL_Normal_Water_Failure
        (
            cons(2) MBox ,
            prod(2) MBox
        )
    with
        # Mode NORMAL: si le message LEVEL_FAIL_DET est présent (l'unité de mesure du
        # niveau d'eau est en panne), passer en mode RESCUE; le message d'acquittement
        # est émis.
            ( MBox(1)-.Msg == MODE )
        ,   ( MBox(1)-.Param4 == NORMAL_MODE )
        ,   ( MBox(2)-.Msg == LEVEL_FAIL_DET )
        ,   ( MBox(1)+.Msg == LEVEL_FAIL_ACK )
        ,   ( MBox(2)+.Msg == MODE )
        ,   ( MBox(2)+.Param4 == RESCUE_MODE )
;

trans CONTROL_Normal_Water_Critical
        (
            cons    MBox ,
            info    MBox ,
            info    Boiler ,
            prod    MBox
        )
    with
        # Mode NORMAL: si le message LEVEL envoyé par l'unité de mesure du niveau d'eau
        # indique que le niveau d'eau est en passe de devenir critique, passer en mode
        # EMERGENCY; pour des raisons de simplicité, on ne regarde que le dépassement des
        # limites inférieure et supérieure.
            ( MBox-.Msg == MODE )
        ,   ( MBox-.Param4 == NORMAL_MODE )
        ,   ( MBox?.Msg == LEVEL )
        ,   ( MBox?.Param3 < Boiler?.WMinLim
                | MBox?.Param3 > Boiler?.WMaxLim )
        ,   ( MBox+.Msg == MODE )
        ,   ( MBox+.Param4 == EMERGENCY_MODE )
;

trans CONTROL_Normal_Unit_Failure
        (
            cons(2) MBox ,
            prod(2) MBox
        )
    with
        # Mode NORMAL: si l'une ou plusieurs autres unités sont en panne, passer
        # en mode DEGRADED (sur reception de *_FAIL_DET), et envoyer les messages
        # d'acquittement.
            ( MBox(1)-.Msg == MODE )
        ,   ( MBox(1)-.Param4 == NORMAL_MODE )
        ,   ( MBox(2)-.Msg == PUMP_FAIL_DET
                | MBox(2)-.Msg == PUMP_CTRL_FAIL_DET
                | MBox(2)-.Msg == STEAM_FAIL_DET)
        ,   ( MBox(2)-.Msg == PUMP_FAIL_DET
                                & MBox(1)+.Msg == PUMP_FAIL_ACK )
          |
            ( MBox(2)-.Msg == PUMP_CTRL_FAIL_DET
                                & MBox(1)+.Msg == PUMP_CTRL_FAIL_ACK )
          |
            ( MBox(2)-.Msg == STEAM_FAIL_DET
                                & MBox(1)+.Msg == STEAM_FAIL_ACK )
        ,   ( MBox(2)-.Msg == PUMP_FAIL_DET
                                & MBox(1)+.Param1 == MBox(2)-.Param1 )
          |
            ( MBox(2)-.Msg == PUMP_CTRL_FAIL_DET
                                & MBox(1)+.Param1 == MBox(2)-.Param1 )
```

```
        ,   ( MBox(2)+.Msg == MODE )
        ,   ( MBox(2)+.Param4 == DEGRADED_MODE )
;

trans CONTROL_Normal_Not_Enough_Water
        (
            info    Boiler ,
            info    Pump ,
            info(2) MBox ,
            neg(2)  MBox ,
            prod    MBox
        )
    with
        # Mode NORMAL: si le message LEVEL envoyé par l'unité de mesure du niveau d'eau
        # indique qu'il n'y a pas assez d'eau, envoyer un message OPEN_PUMP à destination
        # d'une pompe en état de fonctionner si cette pompe ne débite pas déjà.
            ( MBox(1)?.Msg == MODE )
        ,   ( MBox(1)?.Param4 == NORMAL_MODE )
        ,   ( MBox(2)?.Msg == LEVEL )
        ,   ( MBox(2)?.Param3 < Boiler?.WMinNorm )
        ,   ( MBox(1)~.Msg == PUMP_FAIL_DET & MBox(1)~.Param1 == Pump?.PNum )
        ,   ( MBox(2)~.Msg == PUMP_STATE & MBox(2)~.Param1 == Pump?.PNum
                                         & MBox(2)~.Param2 == OPEN )
        ,   ( MBox+.Msg == OPEN_PUMP )
        ,   ( MBox+.Param1 == Pump?.PNum )
;

trans CONTROL_Normal_Too_Much_Water
        (
            info(2) MBox ,
            info    Boiler ,
            neg(2)  MBox ,
            info    Pump ,
            prod    MBox
        )
    with
        # Mode NORMAL: si le message LEVEL envoyé par l'unité de mesure du niveau d'eau
        # indique qu'il y a trop d'eau, envoyer un message CLOSE_PUMP à destination d'une
        # pompe ouverte.
            ( MBox(1)?.Msg == MODE )
        ,   ( MBox(1)?.Param4 == NORMAL_MODE )
        ,   ( MBox(2)?.Msg == LEVEL )
        ,   ( MBox(2)?.Param3 > Boiler?.WMaxNorm )
        ,   ( MBox(1)~.Msg == PUMP_FAIL_DET & MBox(1)~.Param1 == Pump?.PNum )
        ,   ( MBox(2)~.Msg == PUMP_STATE & MBox(2)~.Param1 == Pump?.PNum
                                         & MBox(2)~.Param2 == CLOSED )
        ,   ( MBox+.Msg == CLOSE_PUMP )
        ,   ( MBox+.Param1 == Pump?.PNum )
;

trans CONTROL_Degraded_Pump_Repaired
        (
            info    MBox ,
            cons(2) MBox ,
            prod    Opera
        )
    with
        # Mode DEGRADED: on ne peut quitter le mode DEGRADED qu'en passant en mode RESCUE
        # ou en mode EMERGENCY si la situation se dégrade encore; pour revenir en mode
        # NORMAL, il faut que les réparations aient eu lieu, sinon on reste en mode
        # DEGRADED; il faut donc commencer par recevoir les messages *_REPAIRED, les
        # consommer, et envoyer les accusés de réception à l'opérateur; lorsque l'opérateur
        # aura consommé à son tour les accusés de réception, on pourra repasser en mode
        # NORMAL si tout le reste fonctionne (Cf plus loin).
            ( MBox?.Msg == MODE )
        ,   ( MBox?.Param4 == DEGRADED_MODE )
        ,   ( MBox(1)-.Msg == PUMP_FAIL_ACK )
        ,   ( MBox(2)-.Msg == PUMP_REPAIRED )
        ,   ( MBox(1)-.Param1 == MBox(2)-.Param1 )
        ,   ( Opera+.Msg == PUMP_REPAIRED_ACK )
        ,   ( Opera+.Param == MBox(2)-.Param1 )
;

trans CONTROL_Degraded_PumpCtrl_Repaired
        (
```

```
            info    MBox ,
            cons(2) MBox ,
            prod    Opera
        )
    with
            ( MBox?.Msg == MODE )
        ,   ( MBox?.Param4 == DEGRADED_MODE )
        ,   ( MBox(1)-.Msg == PUMP_CTRL_FAIL_ACK )
        ,   ( MBox(2)-.Msg == PUMP_CTRL_REPAIRED )
        ,   ( MBox(1)-.Param1 == MBox(2)-.Param1 )
        ,   ( Opera+.Msg == PUMP_CTRL_REPAIRED_ACK )
        ,   ( Opera+.Param == MBox(2)-.Param1 )
;

trans CONTROL_Degraded_Steam_Repaired
        (
            info    MBox ,
            cons(2) MBox ,
            prod    Opera
        )
    with
            ( MBox?.Msg == MODE )
        ,   ( MBox?.Param4 == DEGRADED_MODE )
        ,   ( MBox(1)-.Msg == STEAM_FAIL_ACK )
        ,   ( MBox(2)-.Msg == STEAM_REPAIRED )
        ,   ( Opera+.Msg == STEAM_REPAIRED_ACK )
;

trans CONTROL_Degraded_Water_Failure
        (
            cons(2) MBox ,
            prod(2) MBox
        )
    with
        # Mode DEGRADED: si l'unité de mesure du niveau d'eau ne fonctionne pas, passer en
        # mode RESCUE et émettre le message d'acquittement.
            ( MBox(1)-.Msg == MODE )
        ,   ( MBox(1)-.param4 == DEGRADED_MODE )
        ,   ( MBox(2)-.Msg == LEVEL_FAIL_DET )
        ,   ( MBox(1)+.Msg == LEVEL_FAIL_ACK )
        ,   ( MBox(2)+.Msg == MODE )
        ,   ( MBox(2)+.Param4 == RESCUE_MODE )
;

trans CONTROL_Degraded_Unit_Failure
        (
            info    MBox ,
            cons    MBox ,
            prod    MBox
        )
    with
        # Mode DEGRADED: si c'est une autre unité qui tombe en panne, rester en mode
        # DEGRADED.
            ( MBox?.Msg == MODE )
        ,   ( MBox?.Param4 == DEGRADED_MODE )
        ,   ( MBox-.Msg == STEAM_FAIL_DET
                | MBox-.Msg == PUMP_FAIL_DET
                | MBox-.Msg == PUMP_CTRL_FAIL_DET )
        ,   ( MBox-.Msg == PUMP_FAIL_DET
                            & MBox+.Msg == PUMP_FAIL_ACK )
          |
            ( MBox-.Msg == PUMP_CTRL_FAIL_DET
                            & MBox+.Msg == PUMP_CTRL_FAIL_ACK )
          |
            ( MBox-.Msg == STEAM_FAIL_DET
                            & MBox+.Msg == STEAM_FAIL_ACK )
        ,   ( MBox-.Msg == PUMP_FAIL_DET
                            & MBox+.Param1 == MBox-.Param1 )
          |
            ( MBox-.Msg == PUMP_CTRL_FAIL_DET
                            & MBox+.Param1 == MBox-.Param1 )
;

trans CONTROL_Degraded_Water_Critical
        (
```

```
            cons    MBox ,
            info    MBox ,
            info    Boiler ,
            prod    MBox
        )
    with
        # Mode DEGRADED: si le niveau d'eau est critique, passer en mode EMERGENCY.
            ( MBox-.Msg == MODE )
        ,   ( MBox-.Param4 == DEGRADED_MODE )
        ,   ( MBox?.Msg == LEVEL )
        ,   ( MBox?.Param3 < Boiler?.WMinLim
                | MBox?.Param3 > Boiler?.WMaxLim )
        ,   ( MBox+.Msg == MODE )
        ,   ( MBox+.Param4 == EMERGENCY_MODE )
;

trans CONTROL_Degraded_To_Normal
        (
            cons    MBox ,
            neg     Opera ,
            prod    MBox
        )
    with
        # Mode DEGRADED: s'il n'y a plus aucune unité en panne (aucun message ?_FAIL_ACK
        # n'existe, pas plus qu'un message ?_FAIL_DET), et si les accusés de réception
        # ?_REPAIRED_ACK ont été pris en compte, repasser en mode NORMAL.
            ( MBox-.Msg == MODE )
        ,   ( MBox-.Param4 == DEGRADED_MODE )
        ,   ( Opera~.Msg == PUMP_REPAIRED_ACK
                | Opera~.Msg == PUMP_CTRL_REPAIRED_ACK
                | Opera~.Msg == STEAM_REPAIRED_ACK )
        ,   ( MBox+.Msg == MODE )
        ,   ( MBox+.Param4 == NORMAL_MODE )
;

trans CONTROL_Degraded_Not_Enough_Water
        (
            info(2) MBox ,
            info    Boiler ,
            info    Pump ,
            neg(2)  MBox ,
            prod    MBox
        )
    with
        # Mode DEGRADED: si le message LEVEL envoyé par l'unité de mesure de l'eau
        # indique qu'il n'y a pas assez d'eau, envoyer un message OPEN_PUMP
        # à destination d'une pompe en état de fonctionner.
            ( MBox(1)?.Msg == MODE)
        ,   ( MBox(1)?.Param4 == DEGRADED_MODE )
        ,   ( MBox(2)?.Msg == LEVEL )
        ,   ( MBox(2)?.Param3 < Boiler?.WMinNorm )
        ,   ( MBox(1)~.Msg == PUMP_FAIL_DET & MBox(1)~.Param1 == Pump?.PNum )
        ,   ( MBox(2)~.Msg == PUMP_STATE & MBox(2)~.Param1 == Pump?.PNum
                                         & MBox(2)~.Param2 == OPEN )
        ,   ( MBox+.Msg == OPEN_PUMP )
        ,   ( MBox+.Param1 == Pump?.PNum )
;

trans CONTROL_Degraded_Too_Much_Water
        (
            info(2) MBox ,
            info    Boiler ,
            neg(2)  MBox ,
            info    Pump ,
            prod    MBox
        )
    with
        # Mode DEGRADED: si le message LEVEL envoyé par l'unité de mesure du niveau d'eau
        # indique qu'il y a trop d'eau, envoyer un message CLOSE_PUMP à destination d'une
        # pompe ouverte.
            ( MBox(1)?.Msg == MODE )
        ,   ( MBox(1)?.Param4 == DEGRADED_MODE )
        ,   ( MBox(2)?.Msg == LEVEL )
        ,   ( MBox(2)?.Param3 > Boiler?.WMaxNorm )
        ,   ( MBox(1)~.Msg == PUMP_FAIL_DET & MBox(1)~.Param1 == Pump?.PNum )
```

```
        ,   ( MBox(2)~.Msg == PUMP_STATE & MBox(2)~.Param1 == Pump?.PNum
                                         & MBox(2)~.Param2 == CLOSED )
        ,   ( MBox+.Msg == CLOSE_PUMP )
        ,   ( MBox+.Param1 == Pump?.PNum )
;

trans CONTROL_Rescue_Unit_Failure
        (
            cons(2) MBox ,
            prod(2) MBox
        )
    with
        # Mode RESCUE: on arrive normalement en mode RESCUE sur LEVEL_FAIL_ACK; si d'autres
        # unités tombent en panne, on passe en mode EMERGENCY, sinon on reste en mode
        # RESCUE jusqu'à ce que l'unité de mesure du niveau d'eau soit reparée; cependant,
        # tant qu'elle n'est pas reparée, on peut toujours passer en mode EMERGENCY si le
        # niveau devient critique.
            ( MBox(1)-.Msg == MODE )
        ,   ( MBox(1)-.Param4 == RESCUE_MODE )
        ,   ( MBox(2)-.Msg == STEAM_FAIL_DET
                | MBox(2)-.Msg == PUMP_FAIL_DET
                | MBox(2)-.Msg == PUMP_CTRL_FAIL_DET )
        ,   ( MBox(2)-.Msg == STEAM_FAIL_DET
                              & MBox(1)+.Msg == STEAM_FAIL_ACK )
          |
            ( MBox(2)-.Msg == PUMP_FAIL_DET
                              & MBox(1)+.Msg == PUMP_FAIL_ACK )
          |
            ( MBox(2)-.Msg == PUMP_CTRL_FAIL_DET
                              & MBox(1)+.Msg == PUMP_CTRL_FAIL_ACK )
        ,   ( MBox(2)-.Msg == PUMP_FAIL_DET
                              & MBox(1)+.Param1 == MBox(2)-.Param1 )
          |
            ( MBox(2)-.Msg == PUMP_CTRL_FAIL_DET
                              & MBox(1)+.Param1 == MBox(2)-.Param1 )
        ,   ( MBox(2)+.Msg == MODE )
        ,   ( MBox(2)+.Param4 == EMERGENCY_MODE )
;

trans CONTROL_Rescue_To_Normal
        (
            cons(3) MBox ,
            prod    Opera ,
            prod    MBox
        )
    with
        # Mode RESCUE: si l'unité de mesure du niveau d'eau a été reparée, on repasse en
        # mode NORMAL, puisque rien d'autre n'est en panne.
            ( MBox(1)-.Msg == MODE )
        ,   ( MBox(1)-.Param4 == RESCUE_MODE )
        ,   ( MBox(2)-.Msg == LEVEL_FAIL_ACK )
        ,   ( MBox(3)-.Msg == LEVEL_REPAIRED )
        ,   ( Opera+.Msg == LEVEL_REPAIRED_ACK )
        ,   ( MBox+.Msg == MODE )
        ,   ( MBox+.Param4 == DEGRADED_MODE )
;

trans CONTROL_Rescue_Water_Critical
        (
            cons    MBox ,
            info    MBox ,
            info(*) Pump ,
            prod    MBox
        )
    with
        # Mode RESCUE: sinon on calcule la criticité du niveau d'eau; pour plus de
        # simplicité, il faut que la quantité de vapeur soit égale au débit des pompes
        # que multiplie le taux; si c'est différent on passe en mode EMERGENCY.
            ( MBox-.Msg == MODE )
        ,   ( MBox-.Param4 == RESCUE_MODE )
        ,   ( MBox?.Msg == STEAM )
        ,   ( Pump(*)?.PFuncMode == OPEN )
        ,   ( CARD ( { Pump(*)? } ) > 0 )
        ,   ( MBox?.Param3 != RATIO * SUM ( { Pump(*)?.PMaxCap } ) )
        ,   ( MBox+.Msg == MODE )
```

```
        ,   ( MBox+.Param4 == EMERGENCY_MODE )
;

trans CONTROL_Emergency_Stop
        (
            info    MBox ,
            neg     Opera ,
            prod    Opera
        )
    with
        # Mode EMERGENCY: on ne fait rien: au mieux tout est cassé, au pire l'opérateur
        # n'est plus là pour le savoir car tout a explosé; le message STOP est émis vers
        # l'opérateur qui doit arrêter le système.
            ( MBox?.Msg == MODE )
        ,   ( MBox?.Param4 == EMERGENCY_MODE )
        ,   ( Opera~.Msg == STOP )
        ,   ( Opera+.Msg == STOP )
;

# transition ACTION:
#       Effectue les actions liées à la réception des différents messages par les unités;
#       la transition a la plus faible priorité afin que les autres transitions puissent
#       prendre en compte les modifications du système une à une, en particulier les
#       dépassements de limites, les mises à jour de paramètres de message, etc...

trans ACTION_Purge_boiler
        (
            cons    MBox ,
            info    Boiler ,
            cons    WaterMeasDev ,
            prod    WaterMeasDev
        )
    with
        # sur réception du message VALVE, l'action à réaliser est la vidange de la
        # chaudière; cette action est une simulation automatique; afin de tester le
        # remplissage futur de la chaudière, la vidange est effectuée jusqu'à un niveau
        # compris entre WMinNorm et WMaxNorm.
            ( MBox-.Msg == VALVE )
        ,   ( WaterMeasDev-.WQuantity > Boiler?.WMaxNorm )
        ,   ( WaterMeasDev+.WQuantity == Boiler?.WMinNorm
                    + ( Boiler?.WMaxNorm - Boiler?.WMinNorm ) / 2 )
;

trans ACTION_Fill_Boiler
        (
            cons    WaterMeasDev ,
            prod    WaterMeasDev ,
            info(*) Pump
        )
    with
        # si une pompe est en marche, la chaudière doit se remplir; pour une certaine
        # dynamique de remplissage, le niveau est augmenté de la capacité de la pompe à
        # chaque fois que la transition est franchie; il ne faudrait donc pas donner de
        # valeur trop faible à PMaxCap car cela risquerait de durer assez longtemps.
            ( Pump(*)?.PFuncMode == OPEN )
        ,   ( CARD ( { Pump(*)? } ) > 0 )
        ,   ( WaterMeasDev+.WQuantity == WaterMeasDev-.WQuantity
                                        + SUM ( { Pump(*)?.PMaxCap } ) )
        ,   ( WaterMeasDev+.WQuantity != WaterMeasDev-.WQuantity )
;

trans ACTION_Steam_Production
        (
            info    ControlUnit ,
            info    Boiler ,
            cons    SteamMeasDev ,
            prod    SteamMeasDev
        )
    with
        # simulation du début de la production de vapeur
            ( ControlUnit?.PMode == NORMAL_MODE )
        ,   ( SteamMeasDev-.SQuantity == 0.0 )
        ,   ( SteamMeasDev+.SQuantity == Boiler?.SMaxGradInc )
;
```

```
#trans ACTION_Unfill_Boiler
#        (
#            info    Boiler ,
#            info    SteamMeasDev ,
#            cons    WaterMeasDev ,
#            prod    WaterMeasDev
#        )
#    with
#        # simulation de la diminution du niveau d'eau
#            ( SteamMeasDev?.SQuantity > 0.0 )
#        ,   ( WaterMeasDev-.Wquantity == WaterMeasDev-.WQuantity )
#        ,   ( WaterMeasDev+.WQuantity == Boiler?.WMinNorm
#                                    - SteamMeasDev?.SQuantity )
```

ANNEXE F

TRACE D'OBSERVATION NETSPEC (CUBES)

Nous présentons dans cette annexe la trace d'observation du comportement du réseau formel du problème des empilements de cubes. Le travail préalable à l'obtention de cette trace est la compilation du réseau formel présenté au chapitre III (en priorité circulaire), et l'initialisation dans la base de données des tables modélisant les places *cube*, *but*, et *fin_action* selon le jeu de données étudié. Notons dans cette trace que *ZZPK* représente la clé primaire automatiquement ajoutée par *NetSPEC* et que la valeur d'attribut '@' représente la table.

```
[EMPILER]
>>> DELETE FROM CUBE WHERE ZZPK=-7 AND NOM='g' AND POSITION='@' AND PLACEMENT=0 AND GAIN=-1
AND CIBLE=''
>>> INSERT INTO CUBE VALUES(1,'g','@',0,1,'f')

[DEPILER]
>>> DELETE FROM CUBE WHERE ZZPK=-4 AND NOM='d' AND POSITION='a' AND PLACEMENT=0 AND GAIN=-1
AND CIBLE=''
>>> INSERT INTO CUBE VALUES(2,'d','a',0,0,'@')

[ACTION_IMPOSSIBLE]
>>> INSERT INTO FIN_ACTION VALUES(3,1)

[REALISER]
>>> DELETE FROM FIN_ACTION WHERE ZZPK=3 AND ETAT=1
>>> DELETE FROM CUBE WHERE ZZPK=1 AND NOM='g' AND POSITION='@' AND PLACEMENT=0 AND GAIN=1
AND CIBLE='f'
>>> INSERT INTO CUBE VALUES(4,'g','f',1,-1,'')

[DEPILER]
>>> DELETE FROM CUBE WHERE ZZPK=-5 AND NOM='e' AND POSITION='b' AND PLACEMENT=0 AND GAIN=-1
AND CIBLE=''
>>> INSERT INTO CUBE VALUES(5,'e','b',0,0,'@')

[ACTION_IMPOSSIBLE]
>>> INSERT INTO FIN_ACTION VALUES(6,1)

[REALISER]
>>> DELETE FROM FIN_ACTION WHERE ZZPK=6 AND ETAT=1
>>> DELETE FROM CUBE WHERE ZZPK=2 AND NOM='d' AND POSITION='a' AND PLACEMENT=0 AND GAIN=0
AND CIBLE='@'
>>> INSERT INTO CUBE VALUES(7,'d','@',0,-1,'')

[DEPILER]
>>> DELETE FROM CUBE WHERE ZZPK=-1 AND NOM='a' AND POSITION='c' AND PLACEMENT=0 AND GAIN=-1
AND CIBLE=''
>>> INSERT INTO CUBE VALUES(8,'a','c',0,2,'@')

[ACTION_IMPOSSIBLE]
>>> INSERT INTO FIN_ACTION VALUES(9,1)

[REALISER]
>>> DELETE FROM FIN_ACTION WHERE ZZPK=9 AND ETAT=1
```

```
>>> DELETE FROM CUBE WHERE ZZPK=8 AND NOM='a' AND POSITION='c' AND PLACEMENT=0 AND GAIN=2
AND CIBLE='@'
>>> INSERT INTO CUBE VALUES(10,'a','@',1,-1,'')

[EMPILER]
>>> DELETE FROM CUBE WHERE ZZPK=-3 AND NOM='c' AND POSITION='@' AND PLACEMENT=0 AND GAIN=-1
AND CIBLE=''
>>> INSERT INTO CUBE VALUES(11,'c','@',0,1,'a')

[ACTION_IMPOSSIBLE]
>>> INSERT INTO FIN_ACTION VALUES(12,1)

[REALISER]
>>> DELETE FROM FIN_ACTION WHERE ZZPK=12 AND ETAT=1
>>> DELETE FROM CUBE WHERE ZZPK=11 AND NOM='c' AND POSITION='@' AND PLACEMENT=0 AND GAIN=1
AND CIBLE='a'
>>> INSERT INTO CUBE VALUES(13,'c','a',1,-1,'')

[ACTION_IMPOSSIBLE]
>>> INSERT INTO FIN_ACTION VALUES(14,1)

[REALISER]
>>> DELETE FROM FIN_ACTION WHERE ZZPK=14 AND ETAT=1
>>> DELETE FROM CUBE WHERE ZZPK=5 AND NOM='e' AND POSITION='b' AND PLACEMENT=0 AND GAIN=0
AND CIBLE='@'
>>> INSERT INTO CUBE VALUES(15,'e','@',0,-1,'')

[EMPILER]
>>> DELETE FROM CUBE WHERE ZZPK=7 AND NOM='d' AND POSITION='@' AND PLACEMENT=0 AND GAIN=-1
AND CIBLE=''
>>> INSERT INTO CUBE VALUES(16,'d','@',0,1,'b')

[ACTION_IMPOSSIBLE]
>>> INSERT INTO FIN_ACTION VALUES(17,1)

[REALISER]
>>> DELETE FROM FIN_ACTION WHERE ZZPK=17 AND ETAT=1
>>> DELETE FROM CUBE WHERE ZZPK=16 AND NOM='d' AND POSITION='@' AND PLACEMENT=0 AND GAIN=1
AND CIBLE='b'
>>> INSERT INTO CUBE VALUES(18,'d','b',1,-1,'')

[EMPILER]
>>> DELETE FROM CUBE WHERE ZZPK=15 AND NOM='e' AND POSITION='@' AND PLACEMENT=0 AND GAIN=-1
AND CIBLE=''
>>> INSERT INTO CUBE VALUES(19,'e','@',0,1,'c')

[ACTION_IMPOSSIBLE]
>>> INSERT INTO FIN_ACTION VALUES(20,1)

[REALISER]
>>> DELETE FROM FIN_ACTION WHERE ZZPK=20 AND ETAT=1
>>> DELETE FROM CUBE WHERE ZZPK=19 AND NOM='e' AND POSITION='@' AND PLACEMENT=0 AND GAIN=1
AND CIBLE='c'
>>> INSERT INTO CUBE VALUES(21,'e','c',1,-1,'')

[EMPILER]
>>> DELETE FROM CUBE WHERE ZZPK=-8 AND NOM='h' AND POSITION='@' AND PLACEMENT=0 AND GAIN=-1
AND CIBLE=''
>>> INSERT INTO CUBE VALUES(22,'h','@',0,1,'e')

[ACTION_IMPOSSIBLE]
>>> INSERT INTO FIN_ACTION VALUES(23,1)

[REALISER]
>>> DELETE FROM FIN_ACTION WHERE ZZPK=23 AND ETAT=1
>>> DELETE FROM CUBE WHERE ZZPK=22 AND NOM='h' AND POSITION='@' AND PLACEMENT=0 AND GAIN=1
AND CIBLE='e'
>>> INSERT INTO CUBE VALUES(24,'h','e',1,-1,'')

[ACTION_IMPOSSIBLE]
>>> INSERT INTO FIN_ACTION VALUES(25,1)
```

Résumé

Depuis quelques années, des recherches ont été entreprises afin de modéliser l'aspect dynamique du comportement d'un système conçu autour d'une base de données. Différents projets ont donc conduit à proposer des systèmes de gestion de bases de données actifs, mais les concepteurs d'applications ne peuvent cependant pas encore s'affranchir d'une spécification des traitements peu différente de la programmation classique.

Nous proposons de définir un modèle de spécification compréhensible par l'utilisateur, aussi bien adapté à la modélisation des données qu'à celle des traitements, et de concevoir un outil de génération automatique de code de la partie opératoire d'un système d'information, s'appuyant sur l'utilisation des *réseaux formels*. Ce modèle tire les avantages graphiques des réseaux de Petri pour servir de support au dialogue avec l'utilisateur, et hérite de la rigueur des langages formels basés sur la théorie des ensembles en décrivant sans ambiguïté les règles de gestion du système d'information. L'outil qui en est dérivé, baptisé *NetSPEC*, apporte au concepteur la possibilité de concentrer ses efforts sur la conception de son application, non sur son implémentation, puisque le système effectue lui-même la transformation des spécifications d'un réseau formel en un programme utilisant les langages classiques associés aux bases de données.

Mots clé: Spécification des traitements; Réseaux de Petri; Contraintes; Bases de données; Modélisation; Génération de code;

Abstract

Over the past few years, modeling the dynamic behavior of systems designed around databases has become an important research area. Several studies have proposed active database management system prototypes, but application designers cannot yet use working specifications different from classical programming.

This work defines a specification model understandable by the user, adapted to data modeling and to computational modeling. Then, a tool based on formal nets, automatically generating the imperative code of the computational part of an information system is presented.

This model uses the graphical advantages of Petri nets as a visual interface with the user. It inherits the precision of formal languages based on the set theory by describing unambiguously management rules of information systems. The derived tool, called *NetSPEC*, provides the designer with the ability to focus on the design rather than the implementation: *NetSPEC* translates formal net specifications into programs written in classical database languages.

Keywords: Working Specification; Petri Nets; Constraints; Databases ; Modeling; Code Generation;