UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

MÉMOIRE

PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

NICOLAS FORTIN

B. Sc. A.

CONCEPTION D'OUTILS LOGICIELS D'AIDE À LA DÉCISION APPLIQUÉS AU CYCLE DE VIE DES ANODES D'UNE ALUMINERIE

JUILLET 2008

Bibliothèque
Paul-Émile-Boulet

UQAC
Université du Québec
à Chicoutimi

Mise en garde/Advice

Afin de rendre accessible au plus grand nombre le résultat des travaux de recherche menés par ses étudiants gradués et dans l'esprit des règles qui régissent le dépôt et la diffusion des mémoires et thèses produits dans cette Institution, **l'Université du Québec à Chicoutimi (UQAC)** est fière de rendre accessible une version complète et gratuite de cette œuvre.

Motivated by a desire to make the results of its graduate students' research accessible to all, and in accordance with the rules governing the acceptation and diffusion of dissertations and theses in this Institution, the **Université du Québec à Chicoutimi (UQAC)** is proud to make a complete version of this work available at no cost to the reader.

L'auteur conserve néanmoins la propriété du droit d'auteur qui protège ce mémoire ou cette thèse. Ni le mémoire ou la thèse ni des extraits substantiels de ceux-ci ne peuvent être imprimés ou autrement reproduits sans son autorisation.

The author retains ownership of the copyright of this dissertation or thesis. Neither the dissertation or thesis, nor substantial extracts from it, may be printed or otherwise reproduced without the author's permission.

Résumé

Les décideurs d'aujourd'hui sont constamment confrontés à des problématiques semi-structurées, pour lesquelles il n'existe aucune méthode de résolution reconnue. À ce titre, la modélisation et simulation de systèmes est fréquemment utilisée pour supporter les gestionnaires. Les modèles issus de cette méthode, s'avèrent des outils d'aide à la décision, puisqu'ils permettent de simuler des scénarios de production alternatifs.

L'objectif principal de ce travail de recherche est de démontrer comment modéliser et simuler efficacement un système industriel, dont le fonctionnement repose sur des connaissances de natures diverses. Pour ce faire, nous allons mettre à profit des méthodes d'intelligence artificielles, notamment les systèmes experts, afin de représenter cette connaissance.

Dans un premier temps, la démarche empruntée consiste à effectuer une revue de la littérature, en expliquant entre autre, le lien entre des concepts tels la modélisation, simulation et la prise de décision.

Nous discutons également de l'intelligence artificielle, plus particulièrement, du rôle et du fonctionnement de coquilles de développements de systèmes experts basées sur l'algorithme de Rete et ce, dans un contexte de support à la décision.

Ensuite, nous appliquons notre méthode à un système industriel réel, à savoir le cycle de vie des anodes d'une aluminerie. Nous discutons des forces et faiblesses de notre approche. Il est notamment question de l'intégration de la coquille de système expert Jess au sein du modèle. Nous décrivons aussi le type de problème qui convient d'avantage à notre approche.

Finalement, nous effectuons la validation du modèle, afin de démontrer que notre méthode s'avère viable. Nous réalisons aussi quelques expériences à partir de scénarios de production alternatifs. Nous discutons de l'impact de ces scénarios sur certaines mesures de performance du système.

Remerciements

À tout seigneur tout honneur, je tiens d'abord à remercier mon directeur de maîtrise Sylvain Boivin. Sylvain est indéniablement le directeur qu'il me fallait. Il a su m'encourager, malgré toutes les embuches et mes doutes. Sa grande patience et son flegme ont été particulièrement appréciés. J'aimerais aussi remercier au passage Marc Gravel, directeur des programmes de cycles supérieurs en informatique, pour m'avoir aidé à résoudre un imbroglio au niveau des préalables.

Dans un deuxième temps, je tiens à remercier mes parents, Diane et Robert, qui sont une source d'encouragement et d'inspiration continuelle, depuis que je suis de ce monde. Ils constituent les principaux membres de mon syndicat pour le succès. Je souligne aussi l'importance des autres membres de ma famille, ma sœur Marise, mon beau-frère Alexandre, ma grand-mère Marie-Ange et ma tante Huguette, qui m'ont aussi aidé ou qui ont été avec moi en pensées.

Je remercie mon amie Louise qui m'a beaucoup fait réfléchir sur la vie en général et sur l'importance de m'améliorer en tant qu'individu. Je remercie mon ami Steeve, dont la compagnie est toujours appréciée et qui m'a bien changé les idées quand il le fallait. Évidemment, je remercie tous mes autres ami(e)s qui m'ont accompagné de prêt ou de loin. En ordre alphabétique mais pas en ordre d'importance, je pense plus particulièrement à : Caroline, Claude, Dominique, Josée, Patrick, Stéphane B., Sylvain T. et Yanick.

Je tiens aussi à souligner l'apport de mes amis et collègues de travail, dont la présence, presque quotidienne, est toujours inspirante. Encore une fois, pas en ordre d'importance, je pense notamment à : Dany, Gratien, Mathieu G., Mathieu Ti-G., Mathieu T., Stéphane T et Sylvain L.. Finalement, je remercie mes deux patrons Mario et Sylvain D. qui m'ont permis de mener à bien ce travail de recherche, malgré mon emploi.

Table des matières

Résumé ii
Remerciements iii
Table des matières iv
Liste des figures vii
Liste des tableaux viii
Liste des tableaux viii
Introduction 2
1 Modélisation et outil d'aide à la décision 10
1.1 Prise de décision et modèle 10
1.2 Étude de système et modélisation 11
1.3 Avantages et désavantages de la modélisation 14
1.4 Processus traditionnel de modélisation 16
1.5 Catégories de modèles 20
1.6 La modélisation à événements discrets 22
1.6.1 Terminologie de la modélisation à événements discrets 23
1.6.2 Gestion du temps d'un modèle à événements discrets 26
1.6.2.1 Incrément fixe 27
1.6.2.2 Événement suivant 28
1.6.3 Approche de conception d'un modèle à événements discrets 29
1.6.3.1 Planification d'événements 30
1.6.3.2 Recherche d'activités 31
1.6.3.3 Méthode basée sur des processus 32
1.6.3.4 Méthode en trois phases 34
1.6.4 Architecture classique d'un modèle à événements discrets 35
1.7 L'approche orientée objet et la modélisation 37
1.7.1 Caractéristiques d'un modèle orienté objet 37
1.7.2 Modèle orienté objet, cadre et cadriciel 39
1.7.3 Modèle basé sur des composants logiciels 40
1.7.4 Modélisation et le formalisme DEVS 42
1.8 Systèmes visuels de modélisation interactive 44
1.8.1 Fonctionnement d'un VIMS 45
1.8.2 Caractéristiques d'un VIMS 45
1.8.3 Arena un exemple de VIMS 48
1.8.4 La modélisation classique et les VIMS 49
2 Modélisation basée sur l'intelligence artificielle et les systèmes experts 52
2.1 Principes d'intelligence artificielle 52
2.2 Principes d'un système expert 53
2.2.1 Extraction et représentation des connaissances 56
2.2.2 Traitement par inférence des connaissances 58
2.3 Coquille de développement de systèmes experts 59
2.4 Architecture d'un système expert 59
2.5 Base de connaissances 60
2.5.1 Formalismes de représentations de la connaissance 61
2.5.1.1 Logique prédicative 62
2.5.1.2 Les réseaux sémantiques 63
2.5.1.3 Les cadres et les daemons 63
2.5.1.4 Les objets et les règles de production 64
2.5.2 Réseau d'inférence de règles 65
2.5.3 Acquisition et formalisation des règles 67

2.6 Mémoire de travail et les faits 68
2.6.1 Les valeurs des faits 68
2.6.2 Les types de faits 69
2.7 Moteur d'inférence 70
2.7.1 Moteur à chaînage avant 70
2.7.2 Moteur à chaînage arrière 72
2.8 Assortiment de conditions 74
2.8.1 Assortiment basé sur un algorithme naïf 74
2.8.2 Assortiment basé sur l'algorithme de Rete 76
2.8.2.1 Mise en œuvre de l'algorithme de Rete 78
2.8.2.2 Rétraction et modification selon l'algorithme de Rete 81
2.8.2.3 Optimisation de l'algorithme de Rete 82
2.8.2.4 Analyse de l'algorithme de Rete 83
2.9 Agenda des activations 84
2.9.1 Définition de l'importance des règles 85
2.9.2 Stratégie de résolution de conflits 85
2.10 Moteur d'exécution 86
2.11 Interface utilisateur 86
2.12 Composante d'acquisition des connaissances 87
3 Modélisation du cycle de vie des anodes 90
3.1 Cycle de vie des anodes 90
3.1.1 Formation des anodes 90
3.1.2 Entreposage et manutention des anodes 92
3.1.3 Scellement des anodes 92
3.1.4 Production du métal 93
3.2 Formulation du problème et objectifs généraux 95
3.2.1 Objectifs spécifiques de l'étude 95
3.2.2 Mesures de performances 96
3.2.3 Étendue du modèle 97
3.3 Collecte des données et définition du modèle 97
3.3.1 Configuration du système 97
3.3.2 Procédures d'opération du système 100
3.3.3 Gestion des événements probabilistes 103
3.3.3.1 Variables aléatoires 103
3.3.3.2 Distributions de probabilités 104
3.3.3.3 Pourcentage d'efficacité 107
3.3.3.4 Moyenne échantillonnale 108
3.3.3.5 Approche minimaliste 108
3.3.3.6 Approche retenue 109
3.3.4 Gestion des événements discrets 110
3.3.4.1 Gestion du temps du modèle à événements discrets 110
3.3.4.2 Approche de conception du contrôleur 111
3.4 Conception et mise en œuvre du modèle 113
3.4.1 Modèle UML 114
3.4.1.1 Paquetage connaissance 114
3.4.1.2 Paquetage interface graphique 115
3.4.1.3 Paquetage procédé 116
3.4.1.4 Paquetage simulation 118
3.4.2 Intégration d'un système expert 119
3.4.2.1 Représentation de la connaissance à partir de règles 121
3.4.2.2 Moteur d'inférence avec *chaînage avant* 123
3.4.3 Intégration d'un système expert élémentaire 123

3.4.3.1 Base de connaissances 124
3.4.3.2 Mémoire de travail 124
3.4.3.3 Mécanisme d'inférence 124
3.4.4 Intégration de Jess 125
3.4.4.1 Stratégies d'intégration possibles 126
3.4.4.2 Mécanisme d'inférence 128
3.4.4.3 Stratégie de résolution de conflits 128
3.4.4.4 Optimisation des règles 129
3.4.4.5 Les « shadows facts » 131
3.4.4.6 Module 136
3.4.4.7 Point d'entrée du programme Jess 138
4 Validation, expérimentation et analyse des résultats 141
4.1 Limitation de la validité 141
4.1.1 Hypothèses de modélisation 141
4.1.2 Simplifications du modèle 142
4.1.3 Omissions au sein du modèle 142
4.1.4 Limitations durant la modélisation 143
4.2 Validation en surface 143
4.3 Validation statistique 146
4.3.1.1 Validation de l'entrepôt 147
4.3.1.2 Validation des fours 151
4.3.1.3 Validation de la tour à pâte et du scellement 154
4.4 Expérimentation du modèle 157
4.4.1 Facteur et niveau d'une expérience 158
4.4.2 Type d'expérimentation 158
4.4.3 Scénarios d'expérimentation 160
4.5 Analyse des résultats 161
4.5.1 Expérimentation des cycles de cuisson 162
4.5.2 Quatre expérimentations de situation extrême 162
4.5.3 Expérimentation de la situation optimale 164
4.5.4 Expérimentation des tables de transfert 165
4.6 Conclusions 166
Conclusion 169
Bibliographie 179
Annexe 1 : Glossaire du cycle de vie des anodes de carbone d'une aluminerie 190
Annexe 2 : Phase de manutention et d'entreposage des anodes 202
Annexe 3 : Conception de modèles à l'aide du langage Java 206
Annexe 4 : Base de connaissances formalisée avec des règles du langage Jess 215

Liste des figures

Figure 1 : étude d'un système 12
Figure 2 : prise de décision lors d'une modélisation et simulation 13
Figure 3 : processus de conception d'un modèle 17
Figure 4 : diagramme d'activités 25
Figure 5 : avancement avec incrément fixe 27
Figure 6 : avancement jusqu'à l'événement suivant 28
Figure 7 : planification d'événements 30
Figure 8 : recherche d'activités 30
Figure 9 : méthode basée sur des processus 32
Figure 10 : méthode en trois phases 32
Figure 11 : exemple d'héritage et de polymorphisme 38
Figure 12 : un modèle conçu avec Arena 9.0 49
Figure 13 : représentation d'un système expert élémentaire 55
Figure 14 : architecture d'un système expert à base de règles 60
Figure 15 : réseau d'inférence d'une partie d'un problème 66
Figure 16 : l'algorithme naïf et de Rete pour l'assortiment des conditions 76
Figure 17 : réseau de Rete d'une règle 80
Figure 18 : partage des nœuds de condition et de branchement 83
Figure 19 : diagramme des activités du cycle de vie des anodes 91
Figure 20 : ensembles anodiques (tige métallique scellée à l'anode cuite) 93
Figure 21 : salle des cuves (cellules de réduction de l'aluminium) 93
Figure 22 : électrolyse de l'aluminium dans une cellule de réduction 94
Figure 23 : formation, manutention et entreposage des anodes 98
Figure 24 : lois normales de moyenne μ et de variance σ^2 104
Figure 25 : classes du paquetage connaissance 115
Figure 26 : classes du paquetage interface graphique 116
Figure 27 : classes du paquetage procédé 118
Figure 28 : classes du paquetage simulation 119
Figure 29 : représentation des étapes de *formation, manutention et entreposage des anodes* 145
Figure 30 : évolution de l'*entrepôt* le lundi 147
Figure 31 : évolution de l'*entrepôt* le mardi 148
Figure 32 : évolution de l'*entrepôt* le mercredi 148
Figure 33 : évolution de l'*entrepôt* le jeudi 149
Figure 34 : évolution de l'*entrepôt* le vendredi 149
Figure 35 : évolution de l'*entrepôt* le samedi 150
Figure 36 : évolution de l'*entrepôt* le dimanche 150
Figure 37 : production des *fours* le lundi 151
Figure 38 : production des *fours* le mardi 151
Figure 39 : production des *fours* le mercredi 152
Figure 40 : production des *fours* le jeudi 152
Figure 41 : production des *fours* le vendredi 153
Figure 42 : production des *fours* le samedi 153
Figure 43 : production des *fours* le dimanche 154
Figure 44 : production de la *tour à pâte* et du *scellement* le lundi 154
Figure 45 : production de la *tour à pâte* et du *scellement* le mardi 155
Figure 46 : production de la *tour à pâte* et du *scellement* le mercredi 155
Figure 47 : production de la *tour à pâte* et du *scellement* le jeudi 156
Figure 48 : production de la *tour à pâte* et du *scellement* le vendredi 156
Figure 49 : production de la *tour à pâte* et du *scellement* le samedi 157

Liste des tableaux

Tableau 1 : spécifications pour concevoir des composants 41
Tableau 2 : plage horaire des *arrêts planifiés* 103
Tableau 3 : comparaison du système et du modèle pour le cas de référence 147
Tableau 4 : variation des anodes entrées / sorties selon les cycles de cuisson 162
Tableau 5 : variation des anodes entrées / sorties lors de la désactivation du *vibrocompacteur* .. 163
Tableau 6 : variation des anodes entrées / sorties lors de la désactivation de l'*entrepôt* 163
Tableau 7 : variation des anodes entrées / sorties lors de la désactivation du four *P1* 164
Tableau 8 : variation des anodes entrées / sorties lors de la désactivation du *scellement* 164
Tableau 9 : variation des anodes entrées / sorties lorsque la situation est optimale 165
Tableau 10 : variation des anodes entrées / sorties en fonction du taux de transfert 166
Tableau 11 : horaire des navettes du four *P1* 202
Tableau 12 : horaire des navettes du four *P2* 202
Tableau 13 : horaire des navettes du four *R* avec un cycle de 36 heures 203
Tableau 14 : horaire des navettes du four *R* avec un cycle de 32 heures 203
Tableau 15 : horaire des navettes du four *R* avec un cycle de 40 heures 204
Tableau 16 : plage horaire des arrêts planifiés de l'atelier de scellement 204

INTRODUCTION

Introduction

La *simulation* est une méthode qui est fréquemment utilisée pour étudier des systèmes complexes. Parmi les différents champs d'application de cette méthode citons : la conception et l'analyse de systèmes industriels, l'étude de réseaux de télécommunication ou de logiciels, la conception de systèmes appliqués au domaine du transport, l'évaluation de services offerts par une organisation, la réingénierie de procédés d'affaires, l'analyse de systèmes financiers ou économiques [Law et Kelton, 2000].

L'origine de la *simulation* réside dans la théorie statistique de l'échantillonnage et dans l'analyse des probabilités. Le lien commun entre ces deux concepts est l'utilisation d'échantillons et de nombres aléatoires. Une des premières *simulations* contemporaines, basée sur un échantillon aléatoire, fut effectuée durant la Seconde Guerre mondiale. En effet, les physiciens du projet Manhattan étudièrent la diffusion aléatoire des neutrons nécessaires à une réaction nucléaire en chaîne [MacKeown, 1997]. Comme cette recherche était classée secrète, on lui donna un nom de code lié à l'aspect stochastique de la dispersion des neutrons, c'est-à-dire : Monte Carlo. Cette dénomination tire son origine du fait que la capitale de la principauté de Monaco était à l'époque, l'hôte du plus fameux casino du monde. Depuis ce temps, le terme Monte Carlo perdure, sauf qu'on l'associe désormais exclusivement à un processus de *simulation* basé sur les nombres aléatoires [Bratley et al, 1987].

La *simulation* est une méthode descriptive qui permet d'effectuer des expérimentations à partir d'un *modèle* [Turban et Aronson, 2000]. Son appellation « descriptive » sous-entend que la *simulation* permet d'imiter des comportements, plutôt que de définir le processus de résolution d'une problématique [Hoover et Perry, 1990]. Un *modèle* est une représentation abstraite et simplifiée de la réalité. Les *simulations* sont des expériences qui émulent le fonctionnement d'un système concret représenté par un *modèle* [Law et Kelton, 2000]. Le *modèle* est simplifié puisque d'une part, la réalité est souvent trop complexe à décrire et d'autre part, parce que certains éléments concrets ne valent pas toujours la peine d'être représentés [Turban et Aronson, 2000].

Pour mener à bien la *simulation* d'un système, on a souvent recours à un programme informatique qui implémente le *modèle*. Des données sont donc amassées, afin d'estimer les caractéristiques du *modèle* à concevoir et à mettre en œuvre sous la forme d'un programme [Law et Kelton, 2000]. Le *modèle* est essentiel pour qu'il y ait *simulation*. Cependant, l'inverse est inexact, puisque la conception d'un *modèle* n'implique pas obligatoirement une *simulation*. Par exemple, en architecture on utilise des maquettes afin de représenter un édifice. Bien qu'il s'agisse d'un *modèle* réduit de la réalité, il n'est pas utilisé pour réaliser des *simulations*.

Un *système informatisé d'aide à la décision (SIAD)* est un outil logiciel qui interagit avec un utilisateur, dans le but de résoudre des problèmes *non-structurés* ou *partiellement structurés*, à l'aide de *modèles* et de données. Un problème *non-structuré* a la particularité de ne pas avoir de procédure de résolution standard [Turban et Aronson, 2000]. La *simulation* est couramment utilisée dans l'industrie pour explorer les scénarios opérationnels d'un système. Ainsi, les informations générées par une *simulation* aident les gestionnaires à *prendre des décisions*, en mettant en évidence les actions qui sont susceptibles de contribuer à l'amélioration d'un système [Pidd, 1998]. Par exemple, considérons une manufacture qui étudie un projet d'expansion de ses installations mais qui est incertaine d'un retour sur son investissement. Il est impensable de mettre en œuvre le plan d'expansion, puis de l'abandonner en cours de route, lorsqu'on a réalisé qu'il est injustifié. Il apparaît donc souhaitable, de concevoir un *modèle* qui permette de simuler les opérations actuelles et celles du projet d'expansion, afin d'évaluer la pertinence de l'investissement. Nous allons donc considérer les *modèles* axés sur la méthode de *simulation*, en tant qu'*outils d'aides à la décision* pour le milieu industriel. D'ailleurs, la plupart des *systèmes informatisés d'aide à la décision* incorporent un ou plusieurs *modèles* [Turban et Aronson, 2000]. Dans ce contexte, puisque les termes : *modèle* et *outil d'aide à la décision* sont synonymes, nous allons les utiliser de façon interchangeable.

Problématique

Les gestionnaires d'aujourd'hui sont constamment confrontés à des décisions opérationnelles découlant de systèmes complexes. Le plus souvent, celles-ci ont comme objectif leur amélioration, en minimisant les coûts ou en maximisant les profits. Or, la plupart des problèmes découlant de ce type de décisions sont non-structurés [Turban et Aronson, 2000], puisqu'il n'existe aucune procédure systématique qui permette d'obtenir la solution optimale ou du moins, une qui s'en approche. La résolution de tels problèmes est d'autant plus complexe, puisqu'ils nécessitent un éventail de connaissances pratiques à propos du système à améliorer [Gonzalez et Dankel, 1993].

Objectif

Le présent travail de recherche a pour objectif de démontrer comment *modéliser* efficacement un système, dont le fonctionnement repose sur des connaissances de natures diverses et ce, afin de résoudre de telles problématiques. Nous allons donc présenter une méthode de conception de *modèles*, c'est-à-dire d'*outils d'aide à la décision*, qui intègrent la connaissance nécessaire à la résolution de problèmes semi-structurés. En plus de méthodes de *modélisation* traditionnelle [Banks et al, 1998] [Hoover et Perry, 1990] [Law et Kelton, 2000], nous allons recourir à des notions d'intelligence artificielle et plus particulièrement aux *systèmes experts* [Giarratano et Riley, 1998], reposant sur l'algorithme de Rete [Forgy, 1982]. Ceux-ci permettent de représenter les connaissances qui sont appliquées à la fois par les opérateurs et les automates du système.

Méthodologie

La méthodologie de recherche consiste à expérimenter notre stratégie de conception d'*outils d'aide à la décision*, en l'appliquant à un problème de *modélisation* et de *simulation* d'un système industriel. À partir des résultats issus du *modèle*, nous allons être en mesure de discuter des forces et faiblesses de notre approche, en effectuant entre autre, un comparatif avec d'autres méthodes de *modélisation* classiques [Banks et al, 1998] [Law et Kelton, 2000]. Le système sur lequel

s'appuie notre expérimentation est le cycle de vie des anodes d'une aluminerie [Totten et Mackenzie, 2003].

Modélisation et outil d'aide à la décision

Au premier chapitre, nous expliquons de façon détaillée, en quoi consistent la *prise de décision* et le concept de *modèle*. Nous discutons de l'étude d'un système, des différents types de *modélisation* et des principales méthodes associées aux *modèles symboliques*. Il est aussi question des avantages et désavantages de la *modélisation*.

Nous abordons ensuite les étapes du processus traditionnel de *modélisation* : la formulation du problème, la définition des objectifs et la planification, la collecte des données, la conception et la validation du modèle, la mise en œuvre et la vérification, la conception de scénarios, l'exécution et l'analyse des scénarios, ainsi que la documentation et la présentation des résultats. Nous allons également introduire les différentes catégories de *modèle* : statique, dynamique, déterministe, probabiliste, discret ou continu.

Puisque c'est principalement la *modélisation à événements discrets* qui fait l'objet de notre attention, nous allons expliquer sa terminologie : entité, ressource, système, attribut, classe, file, activité, événement et processus. Nous discutons des mécanismes qui permettent de gérer l'avancement du temps des *modèles à événements discrets*. De plus, il est question de quatre approches de conception de ces *modèles*, à savoir : la planification d'événements, la recherche d'activités, la méthode basée sur des processus et la méthode en trois phases. Ensuite, nous présentons une architecture classique, d'un modèle à événements discrets qui s'appuie sur une approche avec planification d'événements.

Nous discutons de la conception de *modèles orientés objets*, à des fins de *simulation* de procédés : OOS (de l'Anglais « Object Oriented Simulation »). Il est donc question de l'origine de ce type de modélisation, de ses principales caractéristiques (abstraction, encapsulation, héritage et polymorphisme), de ses avantages, des différentes structures ou formes de *modèles orientés objets*, tels les hiérarchies de classes, les cadres ou les cadriciels. Nous vous entretenons aussi

d'un cas particulier d'OOS, où le *modèle* est formé de composants logiciels modulables. Nous allons donc définir la notion de composant et expliquer ses avantages, dans un contexte de *modélisation*. Il est aussi question du formalisme DEVS qui permet de spécifier un *modèle* basé sur des composants.

Finalement, nous abordons les s*ystèmes visuels de modélisation interactive* (SVMI), ou VIMS (de l'Anglais « Virtual Interactive Modeling System »), qui constituent des solutions commerciales, pour concevoir des *modèles*. Sans faire une analyse poussée de ces progiciels, nous vous entretenons sur certaines de leurs fonctionnalités et sur les caractéristiques dont ils doivent être dotés. Nous présentons aussi Arena, un VIMS qui est fréquemment utilisé en *modélisation* de système.

Modélisation basée sur l'intelligence artificielle et les systèmes experts

Nous allons d'abord brièvement introduire la notion d'intelligence artificielle (IA). Nous discutons ensuite des fondements de l'un des domaines d'application de l'IA, à savoir les systèmes experts. Nous démontrons que l'extraction, la représentation et l'inférence de la connaissance d'une problématique, nous permet de modéliser des procédés non-structurés, tel le cycle de vie des anodes d'une aluminerie. Nous vous entretenons aussi sur les outils qui facilitent la conception de systèmes experts.

Comme notre *modèle* est en partie fondé sur les systèmes experts, nous décrivons de façon détaillée leur fonctionnement : la base de connaissances et ses formalismes de représentation, la mémoire de travail et les faits, le moteur d'inférence et les différents types de chaînage, l'assortiment des conditions, l'agenda des activations, le moteur d'exécution, l'interface utilisateur et les composantes d'acquisition des connaissances. L'emphase est particulièrement mise sur les méthodes de chaînage et d'assortiment de conditions, ainsi que sur l'une de leur implémentation : l'algorithme de Rete. Ces méthodes régissent le fonctionnement d'un système expert et font l'objet d'une partie de la discussion des chapitres suivants.

Modélisation du cycle de vie des anodes

Nous analysons d'abord le processus modélisé, soit le cycle de vie des anodes d'une aluminerie. Il est question des ressources, entités et activités qui sont associées aux étapes du procédé, à savoir : la formation des anodes, la manutention et l'entreposage des anodes, le scellement des anodes, ainsi que la production du métal. Ensuite, nous énonçons le problème à l'origine de l'étude du processus, nous précisons les objectifs de l'étude, nous décrivons l'étendue du système à modéliser, ainsi que les mesures de performance qui permettent de sonder l'efficacité des scénarios étudiés.

Nous présentons également les informations qui ont été fournies par les spécialistes du système, en ce qui a trait à la configuration et aux procédures d'opération. La phase de définition du *modèle* est basée sur ces informations. Nous esquissons les principales hypothèses de *modélisation*. Il est notamment question des mécanismes de gestion des événements probabilistes : distributions de probabilités, pourcentage d'efficacité, moyenne échantillonnale et approche minimaliste. Nous abordons aussi quelques principes de gestion des événements discrets : avancement du temps de simulation et approche de conception du contrôleur.

Ensuite, il est question de la conception et de mise en œuvre du *modèle*. Puisque le *modèle* est *orienté objet*, nous le décrivons à l'aide d'un diagramme de classes. Nous expliquons les éléments (entités, ressources, attributs, etc.) qui ont suscités notre intérêt, ainsi que la structure hiérarchique et les relations qui existent entre eux.

Nous discutons aussi du rôle d'un système expert intégré à notre *modèle*. Nous allons expliquer en quoi, des règles empiriques, associées à un mécanisme d'inférence à *chaînage avant*, facilitent la mise en œuvre du *modèle*. Pour ce faire, nous comparons cette approche avec des méthodes de conception classiques. Nous parlons aussi des deux prototypes qui ont été considérés. Le premier est associé à un système expert sur mesure. Le second intègre la coquille de développement de système expert Jess [Jess]. Nous expliquons l'architecture et les avantages de chacun d'eux. Toutefois, l'accent est principalement mis sur les caractéristiques du *modèle* qui

intègre Jess, mentionnons, le mécanisme d'inférence, la stratégie de résolution de conflits, les méthodes d'optimisation des règles, les « shadows facts », les modules et le point d'entrée du programme.

Validation, expérimentation et analyse des résultats

Finalement, nous discutons du processus de validation du modèle. Nous commençons d'abord par décrire les différents aspects qui influent sur la validité [Chung, 2004]. Ensuite, nous discutons de la phase de validation en surface et de la démarche empruntée. Nous effectuons aussi une validation statistique. Ainsi, nous comparons les résultats du modèle et du système, en considérant le cas de référence. Puis, nous expérimentons quatre scénarios de production alternatifs. Nous analysons l'impact de ces expériences sur les mesures de performances du système. Nous terminons en énumérant les principales recommandations issues de l'analyse des expériences.

Conclusion

Nous concluons ce travail en rappelant les principaux objectifs de la recherche et la démarche qui a été empruntée pour les atteindre. Nous soulignons les forces et les faiblesses de notre méthode de modélisation, ainsi que le type de problèmes auxquels elle doit d'avantage s'appliquer.

CHAPITRE 1

MODÉLISATION ET OUTIL D'AIDE À LA DÉCISION

1 Modélisation et outil d'aide à la décision

1.1 Prise de décision et modèle

La *prise de décision* est « un processus qui consiste à sélectionner un ensemble d'actions, afin d'atteindre un ou plusieurs objectifs » NDLR : traduit de l'Anglais [Simon, 1977]. Ainsi une activité de gestion telle la planification implique un éventail de décisions : Que fait-on ?, Quand ?, Comment ? Où ? Avec qui ? D'autres activités telles l'organisation ou le contrôle sont aussi associées à la *prise de décision*.

Les gestionnaires de l'industrie sont constamment amenés à prendre des décisions à propos de différents types de processus ou d'installations. Ces processus ou installations sont appelés des systèmes. Un système est « une collection d'entités, comme des personnes ou des machines, qui interagissent ensembles, dans le but d'atteindre un objectif commun » NDLR : traduit de l'Anglais [Law et Kelton, 2000].

Bien souvent, les décisions des gestionnaires sont prises afin d'améliorer un système. Ainsi, on cherche à accroître ses performances en le rendant plus efficace et efficient [Turban et Aronson, 2000]. L'efficacité sert à mesurer à quel degré les objectifs sont atteints. On l'associe aux extrants du système, c'est-à-dire les produits finis ou les résultats, par exemple, le nombre d'anodes fabriquées par un vibrocompacteur. Quant à l'efficience, il s'agit de la mesure des intrants nécessaires à la génération des extrants, par exemple, la quantité de pâte nécessaire à la fabrication d'une anode.

Un *problème* survient lorsqu'un système ne rencontre pas ses objectifs, ne produit pas les résultats attendus, ou ne fonctionne pas selon les spécifications. Dans ce contexte, les gestionnaires doivent chercher à améliorer la performance du système. Or, bien souvent, les *problèmes* de l'industrie sont dits non-structurés, c'est-à-dire qu'ils ne possèdent pas de méthode de résolution standard [Turban et Aronson, 2000]. Toutefois, on s'entend généralement sur un *processus de décision* générique, formé de quatre phases distinctes, pour solutionner n'importe quel type de problèmes [Simon, 1977].

La première phase du *processus de décision* est l'*intelligence*. Elle consiste à effectuer un examen complet du système, à identifier et définir le problème. Au cours de la seconde phase, dite de *conception*, on met en œuvre un modèle, c'est-à-dire une représentation du système et on propose un ensemble de solutions. La phase suivante, dite du *choix* ou de la *décision*, implique de sélectionner une solution et de la tester à l'aide du modèle, pour s'assurer de sa viabilité. La phase finale, dite d'*implantation*, consiste à appliquer la solution choisie au système réel.

Comme on peut le constater, la *prise de décision* et la *solution du problème* sont des concepts fortement connexes. C'est pourquoi on les utilise généralement de façon interchangeable. Par ailleurs, au cœur même du *processus de décision*, on retrouve la notion de modèle. Un modèle est défini comme « une représentation d'un système concret qui permet d'investiguer les améliorations de ce système, ou de découvrir les effets de différents scénarios sur celui-ci. NDLR : traduit de l'Anglais [Pidd, 1998]. Le modèle repose sur un ensemble d'hypothèses qui prennent la forme de relations mathématiques ou logiques. Il s'agit d'un outil qui permet d'appuyer les gestionnaires au cours de la *prise de décision*. Le présent mémoire s'intéresse donc aux modèles, en tant qu'outils d'aide à la décision, appliqués à un système particulier, en l'occurrence le cycle de vie des anodes.

1.2 Étude de système et modélisation

Il existe principalement deux manières d'effectuer l'étude d'un système : l'*expérimentation réelle* et l'*expérimentation à partir d'un modèle*. « L'expérimentation réelle consiste à modifier certaines propriétés physiques du système et à le laisser opérer sous les nouvelles conditions. » NDLR : traduit de l'Anglais [Law et Kelton, 2000].

L'expérimentation à partir d'un modèle constitue la seconde manière d'effectuer une étude [Law et Kelton, 2000]. Elle implique de concevoir une représentation du système, c'est-à-dire le modèle, qui comme on l'a vu est à la base de la *prise de décision*. Le processus qui mène à la conception du modèle est la modélisation. Elle est divisée en deux branches : *iconique* et *symbolique*.

Les modèles *iconiques* ou *physiques* sont une représentation matérielle du système existant, comme la maquette d'un édifice en architecture [Pidd, 1998]. Les modèles *symboliques*, aussi appelés *mathématiques*, représentent les relations logiques et quantitatives du système réel [Hoover et Perry, 1990]. Un modèle symbolique peut notamment servir à concevoir une *heuristique*, effectuer une *simulation* ou obtenir une *solution analytique* [Law et Kelton, 2000] [Turban et Aronson, 2000]. Le diagramme ci-dessous illustre les différentes manières de mener à bien une étude et les principales utilisations des modèles symboliques.

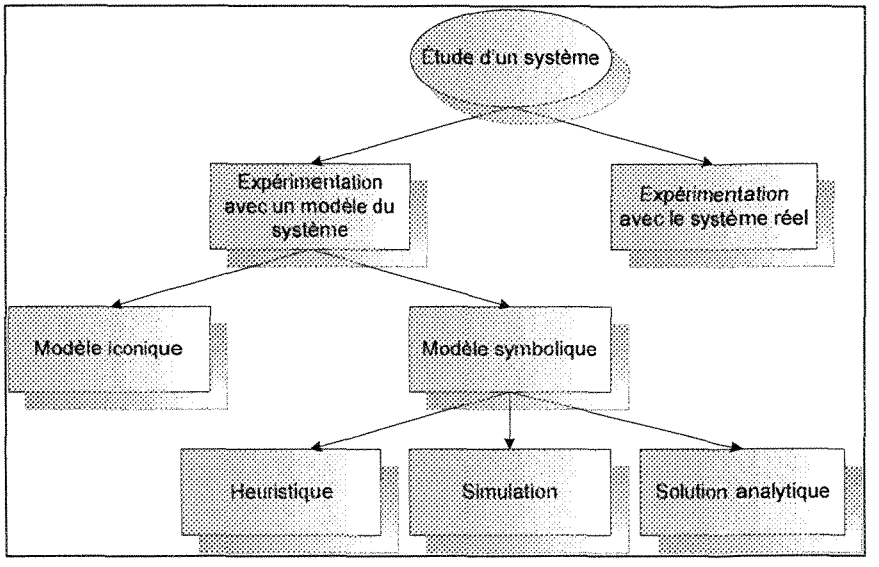

Figure 1 : étude d'un système

Lorsque le modèle est assez simple, il est possible d'utiliser ses relations et un ensemble de valeurs pour obtenir ce qu'on appelle une *solution analytique* [Law et Kelton, 2000]. On peut penser à des méthodes d'optimisation comme la programmation linéaire [Turban et Aronson, 2000]. Celle-ci permet d'estimer un espace de solution à partir d'un modèle, c'est-à-dire un ensemble d'équations et d'inéquations mathématiques [Teghem, 2003]. Cette approche n'a pas été considérée, puisque le type de décision qui nous intéresse découle de problématiques dont on ne connaît pas la procédure de résolution. Quant à l'*heuristique*, on la définie comme suit :

> Une règle empirique ou une méthode établie qui peut aider à trouver une solution, sans pour autant que celle-ci soit assurée, comme ce serait le cas avec un algorithme NDLR : traduit de l'Anglais [Giarratano et Riley, 1998].

Une méthode *heuristique* est donc une astuce qui permet de cibler une solution potentielle. Pour ce faire, on cherche et on évalue des pistes de solution, puis on raffine la recherche, à partir des connaissances acquises [Turban et Aronson, 2000]. Notons que notre modèle ne vise pas à découvrir une solution à partir de nouvelles *heuristiques* ou de méthodes reconnues, telle la recherche avec tabous ou les algorithmes génétiques [Dréo et Siarry, 2003]. Cependant, nous nous sommes intéressés à la *connaissance heuristique* [Friedman-Hill, 2003], découlant du système étudié, soit le cycle de vie des anodes, plus particulièrement, de la possibilité d'adjoindre cette connaissance à un *système expert* [Gonzalez et Dankel, 1993]. Cette question est approfondie au cours des chapitres suivants.

Dans le cadre du mémoire, nous nous intéressons à la modélisation afin d'effectuer des simulations logicielles. Ce type de simulation consiste à « implémenter le modèle sous la forme d'un programme informatique et d'imiter, donc de simuler, le comportement du système réel, en soumettant différents scénarios d'opération » NDLR : traduit de l'Anglais [Pidd, 1998]. Le modèle agit donc comme un véhicule d'expérimentation dans le cadre d'une approche essai et erreur. Ainsi, à la lumière des résultats obtenus, au cours d'une ou plusieurs simulations, les gestionnaires sont en mesures de choisir le scénario le plus profitable. Le diagramme ci-dessous illustre le *processus de décision*, lorsqu'on fait appel à des simulations.

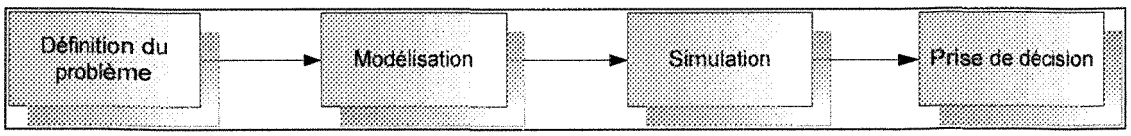

Figure 2 : prise de décision lors d'une modélisation et simulation

Le présent chapitre discute de modélisation et de simulation, en raison de l'intérêt que suscite cette approche au niveau de la prise de décision. Mentionnons qu'afin d'alléger le texte,

nous parlerons de *modélisation* mais qu'on sous-entend aussi la méthode de *simulation* qui lui est associée.

1.3 Avantages et désavantages de la modélisation

La modélisation d'un système n'est pas une panacée. Il s'agit d'un processus qui est habituellement coûteux et long. Dans bien des cas, on l'utilise une fois que d'autres approches, telle la modélisation iconique, les solutions analytiques, voire même l'expérimentation réelle, ont été tentées en vain [Pidd, 1998]. En contrepartie, on aboutit souvent à cette méthode, parce que les systèmes étudiés sont forts complexes [Law et Kelton, 2000]. Mentionnons toutefois que les avancements continuels des technologies de l'information, tant au niveau matériel que logiciel, font de la modélisation une avenue de plus en plus intéressante [Banks et al, 1998]. D'ailleurs, la liste ci-dessous énumère les principaux avantages et désavantages de cette approche.

Les avantages :

- *Compréhension des causes* : L'analyse des causes, à l'origine de certains phénomènes, constitue un des intérêts central des gestionnaires de l'industrie. La modélisation répond à ce besoin, en reconstruisant et en étudiant de façon minutieuse, les scénarios d'opération du système d'intérêt [Banks et al, 1998]. Par exemple, on peut se servir d'un modèle pour identifier des contraintes, telles des goulots d'étranglements, responsables d'un ralentissement des opérations.
- *Compréhension des interactions :* Les installations manufacturières modernes sont forts complexes. Souvent, il apparaît même impossible de considérer toutes les interactions qui se produisent à un moment précis. La modélisation permet de mieux appréhender un système, en analysant les interactions qui surviennent entre leurs variables clés. On peut donc prédire l'effet de ces variables sur sa performance [Banks et al, 1998].
- *Compression et extension du temps* : En comprimant ou dilatant le temps de simulation, le modèle permet d'accélérer ou de décélérer le fonctionnement du système et de l'étudier sur

différentes échelles [Law et Kelton, 2000]. Il est donc possible d'examiner un quart de travail en quelques instants, ou de simuler pendant plusieurs heures, les événements qui se produisent durant une minute.

- *Exploration de scénarios* : Même si la conception d'un modèle est un processus relativement long, une fois à terme, celui-ci permet de simuler le système à partir de paramètres multiples et d'une échelle de temps variable. On peut donc explorer et comparer un vaste éventail de scénarios [Banks et al, 1998].
- *Minimisation des coûts* : La modélisation permet de tester chaque aspect d'une modification du système et ce, sans engager de ressource [Banks et al, 1998]. Ainsi, on évite les problèmes propres à l'expérimentation réelle, puisqu'une fois qu'une décision est appliquée au système existant, un retour à l'arrière peut s'avérer fort coûteux, voire impossible [Pidd, 1998]. À ce titre, l'investissement pour la mise en œuvre d'un modèle, correspond habituellement à 1% du montant attendu pour la conception, ou la réingénierie du système [Banks et al, 1998].
- *Réplétion des simulations* : La modélisation permet de répéter plusieurs fois un même scénario de simulation. Il s'agit d'un avantage certain, car dans la réalité une expérience est rarement répétable [Pidd, 1998].
- *Simulation dynamique et événements transitoires* : Les solutions analytiques sont une avenue intéressante pour étudier un système. Cependant, elles comportent des désavantages lorsqu'on les compare à la modélisation. Notamment, parce qu'elles ne tiennent pas toujours compte de l'aspect dynamique ou des événements transitoires du système [Pidd, 1998]. Par exemple, les actions suivant l'interruption d'un procédé quelconque ne suscitent pas le même intérêt que celles impliquées durant une situation optimale.
- *Simulation des scénarios extrêmes* : L'un des objectifs de la modélisation est de pouvoir simuler les conséquences de scénarios extrêmes [Pidd, 1998]. Or, l'expérimentation de tels scénarios sur un système réel peut être excessivement dangereuse, voire totalement illégale.

Les désavantages :

- *Comparaison des scénarios* : Un modèle est basé sur des variables aléatoires [Baillargeon, 1990] [Banks et al, 1998]. Un scénario simulé produit un estimé des caractéristiques réelles du modèle, pour un ensemble de paramètres d'entrées donnés. Ainsi, pour chaque scénario, plusieurs essais indépendants doivent être effectués, afin d'obtenir un portrait global des caractéristiques du modèle. Pour cette raison, il est difficile de comparer plusieurs scénarios entre eux [Law et Kelton, 2000]. À l'inverse, les méthodes analytiques donnent les caractéristiques précises du modèle, pour un ensemble de paramètres d'entrées donnés.
- *Confiance abusive* : Le nombre important d'informations d'une simulation ou l'impact d'une animation dynamique peuvent générer une confiance excessive en une étude. Ainsi, un modèle invalide, peu importe la vraisemblance de ses résultats, risque de fournir peu d'informations sur les caractéristiques réelles du système [Law et Kelton, 2000].
- *Coût et temps* : La modélisation est une des méthodes les plus coûteuses pour étudier un système, que ce soit en en argent ou en temps. C'est particulièrement vrai lorsqu'on la compare aux méthodes analytiques [Law et Kelton, 2000].
- *Expertise de modélisation* : La conception d'un modèle requiert à la fois des spécialistes du système et du monde de la modélisation. La connaissance de la modélisation ne peut s'acquérir qu'avec le temps et l'expérience [Banks et al, 1998].
- *Interprétation des résultats* : Comme les résultats du modèle sont en fonction de variables aléatoires, il peut apparaître difficile de déterminer si les caractéristiques observées sont le résultat des interactions du modèle, ou de sa nature aléatoire [Banks et al, 1998].

1.4 Processus traditionnel de modélisation

Dans la littérature, des auteurs tels [Banks et al, 1998], [Hoover et Perry, 1990] et [Law et Kelton, 2000], s'entendent sur une démarche systématique commune, afin de mener à bien un projet de modélisation. Les jalons de cette démarche forment le processus de modélisation d'un système. La présente section a pour but d'introduire brièvement les étapes de ce processus.

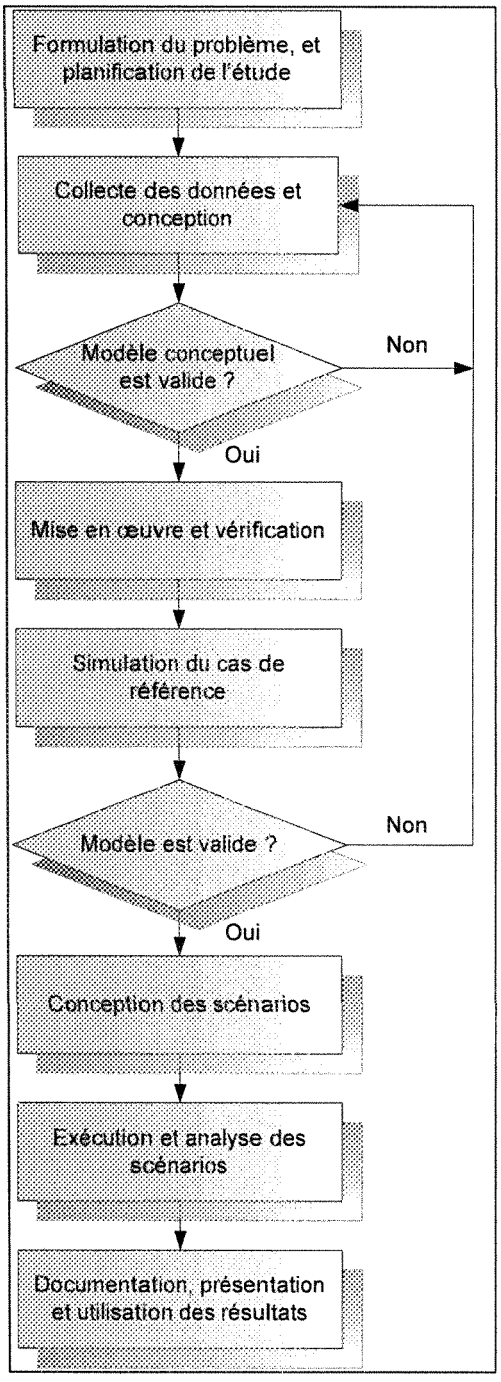

Figure 3 : processus de conception d'un modèle

Formulation du problème et planification de l'étude :

Un projet de modélisation débute par un énoncé du problème à résoudre. Les parties impliquées se réunissent ensuite pour discuter des objectifs généraux de l'étude, des interrogations spécifiques auxquelles elle doit répondre, des mesures de performances, de l'étendue du modèle, de la configuration du système modélisé, des logiciels ou technologies utilisés, de l'échéancier et finalement des ressources assignées.

Collecte des données et conception du modèle :

La seconde étape consiste à collecter des informations à propos du système, par exemple son fonctionnement, ses performances et ses procédures d'opération. À partir de ces indications, on définit des concepts comme les paramètres du modèle, les variables d'état, les entités impliquées, ou les distributions de probabilités utilisées. Ainsi, on peut poser des hypothèses sur le fonctionnement du système. Ces hypothèses constituent l'esquisse conceptuelle du modèle.

Validation du modèle conceptuel :

Les principaux acteurs impliqués dans le projet doivent ensuite valider le modèle conceptuel, c'est-à-dire chacune des hypothèses recensées à l'étape précédente. Ceci permet de s'assurer qu'elles sont conformes et complètes. En plus, cette étape évite d'éventuels retours en arrière, une fois que la mise en œuvre du modèle est entamée.

Mise en œuvre du modèle et vérification :

Au cours de la quatrième étape, le modèle est mis en œuvre à partir d'un langage de programmation quelconque ou d'un logiciel de simulation. Nous discutons de cette dernière possibilité à la section 1.8. Finalement, on doit vérifier l'exactitude du modèle, notamment en effectuant des tests unitaires, pour s'assurer qu'il fonctionne conformément aux hypothèses et pour corriger les défectuosités.

Exécution et validation du modèle :

Cette étape permet de s'assurer que le modèle est une représentation acceptable du système réel. Pour ce faire, on expérimente d'abord un scénario commun au modèle et au système existant. Ce scénario est ce qu'on appelle le cas de base ou de référence. Ensuite, on s'assure qu'il existe une certaine similitude entre les résultats des deux expériences, par exemple, en comparant leurs indicateurs de performances respectifs. Dans certains cas, le système de base est inexistant. Notamment, lorsqu'on réalise un projet de modélisation pour justifier la conception d'un nouveau système. C'est pourquoi il existe des méthodes de validations alternatives [Banks et al, 1998]. Celles-ci dépassent le cadre du présent mémoire. En effet, le type de modèles que nous considérons vise à soutenir la prise de décision, dans une optique d'amélioration d'un système existant. Néanmoins que le système de base soit réel ou non, il importe que les parties impliquées étudient les résultats, afin de valider le modèle. Une analyse de sensibilité [Law et Kelton, 2000] est aussi souhaitable, afin d'établir les paramètres qui ont le plus d'impact sur les mesures de performances. Ces paramètres doivent faire l'objet d'une attention particulière lors des étapes subséquentes.

Conceptions de scénarios :

Pour chaque scénario défini lors de la planification initiale, on doit déterminer des aspects tels la durée d'un essai de simulation (aussi appelée *réplication*) [Banks et al, 1998], le nombre d'essais, ainsi que la valeur des paramètres initiaux du modèle. Nous allons revenir à la notion d'essai un peu plus loin.

Exécution et analyse des scénarios :

Cette étape du processus permet d'expérimenter l'ensemble des scénarios simulés et d'estimer leurs performances respectives. Pour se faire, on s'appuie sur les indicateurs de performance définis lors de la planification. L'analyse des résultats doit également permettre de comparer les scénarios entre eux, afin d'établir celui qui est le plus performant.

Documentation, présentation et utilisation des résultats :

Cette étape consiste à documenter les hypothèses initiales, le fonctionnement du modèle et les résultats de l'étude. L'analyste en modélisation doit aussi présenter les résultats au requérant de l'étude. S'ils sont considérés suffisamment crédibles, les résultats peuvent être utilisés pour appuyer une décision stratégique.

1.5 Catégories de modèles

Comme on l'a mentionné, il existe plusieurs formes de modèles. Toutefois, dans un contexte de simulation, on a souvent recours à des modèles symboliques ou mathématiques. Ainsi, lorsque ce type de modèles est valide, il permet d'étudier le comportement du processus dans une optique d'amélioration [Hoover et Perry, 1990]. La discussion qui suit a pour but de présenter les principales catégories de modèles symboliques.

Modèle statique versus dynamique :

Les modèles statiques sont utilisés lorsqu'on doit considérer une représentation du système, à un moment précis, ou lorsque le temps n'est pas une considération essentielle. C'est pourquoi on a rarement recours à un tel type de modèle. La modélisation de Monte Carlo est la forme de modélisation statique la plus connue [Law et Kelton, 2000]. Elle permet d'estimer le fonctionnement d'un système, en s'appuyant sur les nombres aléatoires. Souvent, on a recours à cette méthode en mathématiques ou en physique, pour évaluer des intégrales définies fort complexes [Bratley et al, 1987]. À l'opposé de cette approche, la modélisation dynamique sert à représenter un processus dont les variables évoluent à travers le temps. On peut penser à un ensemble de clients qui se présentent à intervalle régulier, pour faire la file devant un guichet bancaire.

Modèle déterministe versus probabiliste :

Lorsque les variables d'un modèle n'intègrent aucune notion de probabilité, il est dit déterministe [Hoover et Perry, 1990]. Par exemple, un ensemble d'équations différentielles, décrivant l'air qui s'écoule le long d'une aile d'avion, est déterministe. Une fois que les paramètres

initiaux et les relations, d'un tel modèle, ont été définis, les résultats qu'il génère demeurent fixes. Néanmoins, bien des systèmes doivent être modélisés en tenant compte de la notion de probabilité. Cette notion est intimement liée au concept de variable aléatoire [Baillargeon, 1990]. Une variable aléatoire est définie ainsi :

> Une variable qui ne prendra sa valeur (parmi différentes possibles) qu'après un tirage au hasard, c'est-à-dire après l'aboutissement d'un processus (expérience), dit aléatoire, dont le résultat demeure en dehors du contrôle de l'observateur. [Kakmier, 1982].

En attribuant une probabilité à toutes les valeurs d'une variable aléatoire, on obtient une distribution de probabilité. Or, lorsqu'on modélise un système, certaines valeurs affectées aux variables du modèle peuvent être plus ou moins probables que d'autres. Donc, pour atteindre un certain degré de réalisme, on considère les variables du modèle comme aléatoires et on cherche à estimer leur distribution de probabilité [Law et Kelton, 2000]. Par exemple, on peut penser à un modèle qui émule le comportement d'une file d'attente, où l'intervalle d'arrivée des clients suit une loi de probabilité normale.

Une méthode minimaliste, fréquemment utilisée lorsqu'on conçoit un modèle probabiliste, consiste à simuler un scénario, au cours d'un essai d'une durée arbitraire. Cet essai est ensuite considéré comme un estimé des caractéristiques du modèle [Law et Kelton, 2000]. Toutefois, comme les variables d'un modèle probabiliste sont basées sur des distributions de probabilités [Baillargeon, 1990], il est possible qu'elles aient une très grande variance. Ainsi, l'estimé obtenu au cours d'un essai particulier peut différer grandement des caractéristiques réelles du système. D'où l'importance d'utiliser des techniques statistiques, afin de concevoir et d'analyser une expérience de simulation basée sur de nombreux essais [Hoover et Perry, 1990].

Notons que les modèles de Monte Carlo [Bratley et al, 1987] sont un cas typique de modélisation probabiliste. Il existe également une sous-famille des modèles probabilistes, dite stochastique. En fait, un modèle est stochastique, lorsqu'il est à la fois probabiliste et dynamique. Les variables aléatoires d'un tel modèle varient donc à travers le temps [Processus stochastique].

Modèle discret versus continu :

L'aspect discret ou continu fait référence à l'ensemble des valeurs que peuvent prendre les variables du système. Ainsi, une variable continue peut prendre n'importe quelle valeur réelle comprise dans un intervalle, tandis qu'une variable discrète peut uniquement prendre un ensemble limité de valeurs [Hoover et Perry, 1990]. Dans le cas des modèles dynamiques, cette distinction, entre le discret et le continu, est souvent faite par rapport aux variables qui sont associées au temps. Ainsi, un modèle est discret lorsque ses variables d'état changent instantanément à intervalle régulier [Law et Kelton, 2000]. Par exemple si l'on cherche à modéliser une file d'attente, on devrait normalement opter pour une représentation discrète, car les valeurs des variables d'états, tel le nombre de clients dans la file, sont modifiés aussitôt qu'un nouveau se présente. À l'inverse, les variables d'état d'un système continu changent de façon ininterrompue à travers le temps [Pidd, 1998]. Il importe donc de modéliser tous les instants possibles. Par exemple, le déplacement d'un avion doit être modélisé de façon continue, lorsqu'on considère des variables d'état comme la vitesse ou le carburant consommé. Les modèles continus sont généralement des équations différentielles qui statuent le taux de changement des variables d'états. Toutefois un système continu n'implique pas obligatoirement un modèle continu et vice versa. En fait, cette décision dépend des objectifs de l'étude et des variables qu'on choisit de modéliser [Law et Kelton, 2000].

1.6 La modélisation à événements discrets

Comme nous venons de le mentionner, les variables d'un modèle discret changent au cours d'instants précis et dénombrables. C'est lors de ces instants qu'un événement survient. On définit un événement comme « une occurrence instantanée au cours de laquelle l'état du système peut évoluer » NDLR : traduit de l'Anglais [Law et Kelton, 2000]. Toutefois, les modèles discrets ne comportent pas toujours des événements qui modifient ses variables d'état. Par exemple, un événement peut servir à planifier la fin de la simulation, ou le commencement d'une nouvelle opération. C'est pourquoi on dit que l'état *peut* évoluer lorsqu'un événement est déclenché.

Compte tenu de cette notion événementielle, un constat s'impose, à savoir qu'un modèle avec des événements discrets est implicitement dynamique, car les variables du système changent à travers le temps. De plus, il est généralement probabiliste. Néanmoins, cette notion est facultative, puisque les modèles déterministes peuvent être considérés comme des cas spéciaux des modèles probabilistes [Law et Kelton, 2000]. Dans la littérature, un modèle qui est à la fois discret, dynamique et probabiliste est qualifié comme un *modèle à événements discrets*. Les sections qui suivent visent à approfondir ce concept, car les outils d'aide à la décision que nous avons conçus sont basés sur celui-ci.

1.6.1 Terminologie de la modélisation à événements discrets

Nous allons maintenant définir la terminologie découlant des principales composantes d'un modèle à événements discrets.

Les objets du système :

Entité : « Il s'agit des objets individuels du système qui sont modélisés et dont le comportement ou l'état doit être explicitement suivi. » NDLR : traduit de l'Anglais [Pidd, 1998] Par exemple, chaque anode ou chaque convoyeur qui les déplacent, forment des entités du système de manutention et de scellement d'anodes.

Ressource : « Il s'agit d'une entité statique qui fournit un service à d'autres entités dynamiques. » NDLR : traduit de l'Anglais [Banks et al, 1998] Par exemple, un bassin chargé de refroidir un ensemble d'anodes constitue une ressource.

Organisation des entités :

Lorsque les entités sont modélisées de façon individuelle, il faut généralement les regrouper et les caractériser séparément. Dans ce contexte, voici la terminologie qui est généralement utilisée.

Attribut : « Ce sont des valeurs qui caractérisent une entité. » NDLR : traduit de l'Anglais [Law et Kelton, 2000] L'ensemble des attributs d'entités permet de modéliser l'état du système.

Dans le cadre d'un modèle avec des événements discrets, les attributs changent au cours d'instants précis. Par exemple, chaque convoyeur du cycle de vie des anodes possède un attribut spécifiant s'il est actif ou inactif. Celui-ci est modifié au gré d'événements tels des arrêts planifiés ou non-planifiés.

Classe ou enregistrement : « Ce sont des groupes permanents qui représentent des entités identiques ou similaires » NDLR : traduit de l'Anglais [Pidd, 1998]. Par exemple, chacune des entités *anodes* forment une classe du même nom, puisqu'elles ont toutes les mêmes attributs.

File ou liste : « Ce sont des groupes temporaires d'entités qui possèdent des propriétés communes. ». NDLR : traduit de l'Anglais [Law et Kelton, 2000] On utilise fréquemment ce concept, afin de représenter une file d'entités en attente [Pidd, 1998]. En effet, à mesure que les entités traversent le système, elles requièrent l'utilisation de ressources. Lorsqu'une ressource n'est pas disponible, alors l'entité joint une file. Dans ce contexte, les files peuvent aussi être considérées comme des entités, puisqu'elles possèdent des attributs descriptifs. Par exemple, considérons un *convoyeur*, du cycle de vide des anodes, qui s'avère être une file d'entités *anodes*. Celui-ci possède des attributs tels son *état* (actif / inactif) ou le *nombre d'anodes* dans la file.

Opérations des entités :

Au fur et à mesure qu'une simulation avance, les entités interagissent entre elles et leurs états changent. Une terminologie précise est nécessaire pour décrire ces opérations.

Activité : « Il s'agit d'une opération ou d'une procédure qui est initiée lorsqu'un événement débute » NDLR : traduit de l'Anglais [Pidd, 1998]. C'est donc en raison des activités que l'état des entités est modifié. La durée de l'activité peut être une constante, une valeur aléatoire issue d'une distribution statistique, une valeur calculée à partir d'une équation, etc. [Banks et al, 1998] Par exemple, considérons l'activité *Refroidir anode* qui consiste à faire transiter une anode crue, au sein d'un bassin pour abaisser sa température. Cette activité contribue à faire évoluer l'état de l'entité *Anode* de *Chaude* à *Froide*.

La modélisation discrète implique l'identification des classes d'entités et des activités dans lesquelles elles s'engagent. Ainsi, il est possible de lier ces activités entre elles, afin de visualiser le cycle de vie de chaque classe d'entités. Les diagrammes d'activités [Fowler, 2004] sont une façon de modéliser les interactions entre les classes d'entités [Pidd, 1998]. Ils conviennent particulièrement lorsqu'on les applique à des systèmes dont la structure de base est la file. Ci-dessus se trouve une partie du diagramme des activités du cycle de vie des anodes. Il est formalisé à partir du formalisme UML qui est défini comme suit :

> Une famille de notations graphiques, supportées par un méta-modèle, qui aide à décrire et concevoir des systèmes logiciels, plus particulièrement des systèmes basés sur le formalisme orienté objet. NDLR : traduit de l'Anglais [Fowler, 2004].

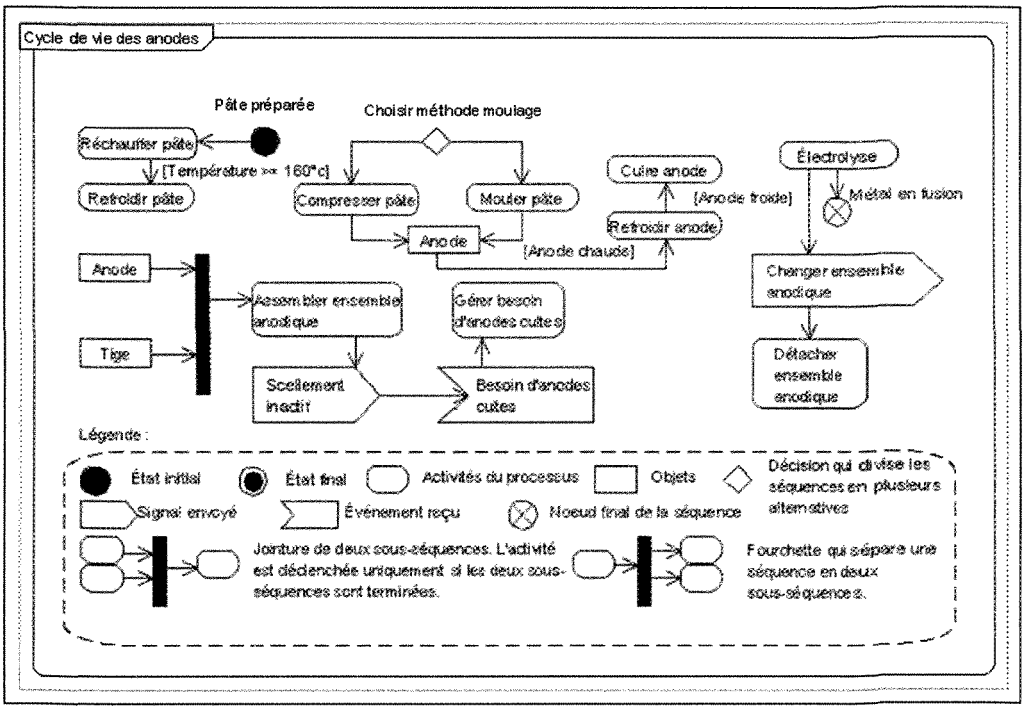

Figure 4 : diagramme d'activités

Il s'agit d'une notation utilisée tout au long du mémoire. Nous nous abstenons de la décrire en détails, comme elle est abondamment documentée, notons par [Fowler, 2004].

Événement : On considère généralement un événement comme « une occurrence de temps qui contribue à l'évolution de l'état du système » NDLR : traduit de l'Anglais [Banks et al, 1998].

Précisions encore une fois qu'un événement n'implique pas nécessairement un changement d'état, bien que ce soit souvent le cas.

Il peut y avoir différentes catégories d'événements. Certains d'entre eux sont *planifiables* et qualifiés de type *B pour « Bound to happen »* [Pidd, 1998]. Par exemple, considérons l'activité *Presser anode*, il est possible de planifier quand l'événement *Fin pressage* survient, en autant qu'on connaisse le moment où l'activité a débuté et sa durée. Notons qu'en plus d'une modification de l'état, les événements planifiables de type *B* vont généralement libérer des ressources et des entités, dans notre exemple : une presse hydraulique et une anode.

En contrepartie, il existe des événements qui ne sont pas dépendants du temps mais plutôt conditionnels à certaines activités. Donc on les qualifie de type *C* pour « *Conditional* » ou « *Cooperative* » [Pidd, 1998]. Par exemple, l'activité *Assembler composantes*, consistant à sceller une anode cuite avec une tige métallique droite, ne peut se produire que si l'événement *Début assemblage* statue que les deux composantes sont disponibles. Souvent, une activité va commencer par un événement de type *C* et se terminer par un de type *B*.

Processus : Il s'agit d'une séquence d'événements regroupés chronologiquement » NDLR : traduit de l'Anglais [Pidd, 1998]. Un processus est souvent utilisé pour décrire tout le cycle de vie d'entités temporaires, c'est-à-dire l'ensemble de leurs activités et de leurs événements. À ce titre, le cycle de vie des anodes d'une aluminerie, dont il est question au cours de ce mémoire, est un bel exemple de processus.

1.6.2 Gestion du temps d'un modèle à événements discrets

La modélisation à événements discrets implique que les changements d'états du système surviennent dans le temps [Pidd, 1998]. C'est pourquoi il importe d'une part, de conserver la valeur du temps présentement simulé, d'autre part, de mettre en œuvre un mécanisme pour avancer le temps de la simulation. La variable du modèle qui donne la valeur du temps actuellement simulé, s'appelle l'*horloge de la simulation* [Law et Kelton, 2000]. Cette valeur est exprimée en une unité de temps particulière, aussi utilisée pour caractériser certains paramètres d'entrée. En ce qui concerne

le mécanisme qui régit l'avancement de cette horloge, on s'entend généralement sur deux approches pour l'implanter : *incrément fixe* et *événement suivant.*

1.6.2.1 Incrément fixe

La méthode avec incrément fixe consiste à augmenter la valeur de l'horloge, par de petits intervalles de temps égaux, aussi appelés pas, d'une valeur de Δt unités. Après chaque mise à jour de l'horloge, correspondant à un multiple de Δt, une vérification est faite, afin de statuer les événements qui sont survenus durant l'intervalle précédent. S'il y a un ou plusieurs événements, le modèle considère qu'ils se sont produits à la fin de l'intervalle et met à jour les variables d'état [Law et Kelton, 2000]. La figure ci-dessous illustre le principe.

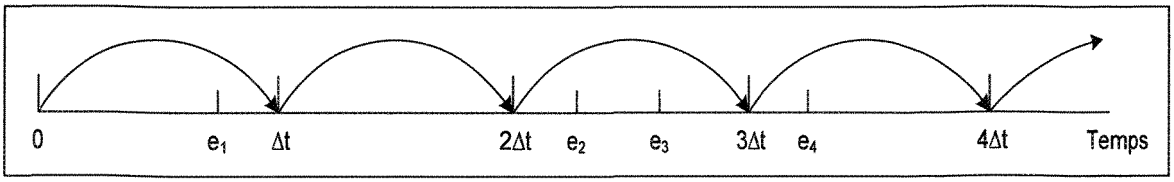

Figure 5 : avancement avec incrément fixe

Les flèches recourbées symbolisent l'avancement du temps de l'horloge. Les paramètres suivants décrivent le fonctionnement de la méthode *incrément fixe* :

n : le nombre d'événements qui surviennent durant la simulation.

e_i : le temps où survient l'$i^{ème}$ événement, où i = [1, n]

Δt : la durée d'un intervalle (pas) de temps.

m : le nombre d'intervalles (pas) de temps simulés.

$j \times \Delta t$: le temps où l'on exécute les événements du $j^{ème}$ intervalle (pas), où j = [1, m].

En étudiant cet exemple, on constate que le premier événement survient en e_1, entre l'intervalle [0, Δt[. Toutefois, comme on l'a mentionné plus haut, le modèle considère uniquement cet événement après le passage de Δt unités de temps.

Un désavantage de cette méthode réside dans le fait qu'il y a toujours des vérifications, même lorsqu'aucun événement ne s'est produit durant un intervalle. [Pidd, 1998] Par exemple, en

considérant [Δt, 2Δt[, on constate qu'aucun événement n'est survenu. Cependant, après 2Δt unités de temps, le modèle vérifie quand même les événements de l'intervalle [Δt, 2Δt[.

Pour contrer ces vérifications inutiles, on peut augmenter la valeur de Δt [Law et Kelton, 2000]. Ainsi, il y a plus de chance qu'un événement survienne à chaque intervalle. Cependant, cette augmentation risque d'occasionner une perte de réalisme. En effet, lorsque plusieurs événements surviennent dans un même intervalle, on doit choisir de façon arbitraire leur ordre d'exécution. Or, l'ordonnancement établi ne correspond pas nécessairement à la réalité.

À l'inverse, une diminution de la valeur de Δt améliore la précision mais accroît aussi la fréquence des vérifications. Le modèle devient donc beaucoup moins performant [Hoover et Perry, 1990]. Pour toutes ces raisons, on utilise surtout l'approche avec *incrément fixe*, lorsque le temps entre les événements est relativement constant et lorsqu'il y a de nombreux changements d'état [Law et Kelton, 2000].

1.6.2.2 Événement suivant

Avec cette méthode, l'horloge du modèle est d'abord initialisée à zéro. Puis, on estime le moment où des occurrences d'événements futurs surviennent. L'horloge est avancée jusqu'au premier événement. À ce point, l'état du modèle est mis à jour et au besoin, on modifie notre connaissance des événements futurs. Le processus se poursuit en avançant l'horloge au prochain événement et ainsi de suite. Normalement, le processus continue jusqu'à ce qu'une condition d'arrêt soit atteinte [Law et Kelton, 2000]. La figure ci-dessus illustre le principe.

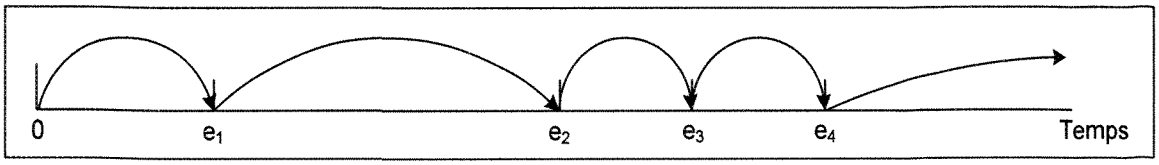

Figure 6 : avancement jusqu'à l'événement suivant

L'approche d'avancement jusqu'à l'*événement suivant* a deux avantages par rapport à l'approche avec *incrément fixe* [Pidd, 1998]. Le premier est que la durée des pas de temps s'ajuste

automatiquement, peu importe le nombre de changements d'état du système. Le fait est qu'en connaissant les événements futurs, on évite les vérifications inutiles pour les intervalles subséquents. De plus, cette méthode met en évidence les événements significatifs d'une simulation. En effet, comme la plupart des modèles discrets mettent à jour leur interface lorsque des événements se produisent, on peut clairement appréhender les principaux jalons qui contribuent à l'évolution de l'état du système.

En contrepartie, l'approche de type *événement suivant* nécessite de maintenir des informations pour contrôler le déroulement d'une simulation, par exemple une liste chaînée des événements futurs [Hoover et Perry, 1990]. De plus, l'écoulement du temps n'est pas sans à-coups, en raison des intervalles variables. Ainsi, les modèles qui implémentent cette méthode peuvent parfois apparaitre plus complexes et confus du point de vue de l'utilisateur.

Finalement, notons que [Law et Kelton, 2000] et [Pidd, 1998] considèrent la méthode *événement suivant* beaucoup plus générique. En effet, lorsque les événements d'un système surviennent à intervalle régulier, un modèle de type *événement suivant* devrait se comporter comme un modèle de type *incrément fixe*. C'est pourquoi on considère parfois cette dernière méthode comme un cas particulier de la méthode *événement suivant*.

1.6.3 Approche de conception d'un modèle à événements discrets

Les modèles à événements discrets sont généralement divisés en trois parties distinctes [Law et Kelton, 2000] [Pidd, 1998]: La première est le contrôleur qui régit le déroulement de la simulation. La seconde est la logique du modèle, à savoir l'expression des activités dans lesquelles les entités ou les ressources s'engagent. La dernière partie est un ensemble d'outils génériques qui permettent entre autre, l'affichage et la saisie d'informations, l'avancement du temps simulé, la génération de valeurs à partir de distributions de probabilités, ainsi que la production de rapports basés sur les mesures de performances.

Il apparaît utile que le contrôleur puisse être réutilisé pour différents types de problèmes [Pidd, 1998]. Cette notion de réutilisabilité dépend en fait de l'approche de conception employée.

Dans la littérature, on s'entend généralement sur quatre approches de conception qui nécessitent une implémentation différente du contrôleur. Ces approches sont : *la planification d'événements*, *la recherche d'activités*, *la méthode basée sur des processus* et *la méthode en trois phases* [Balci, 1988] [Banks et al, 1998]. La présente section a pour but de présenter brièvement chacune de ces approches.

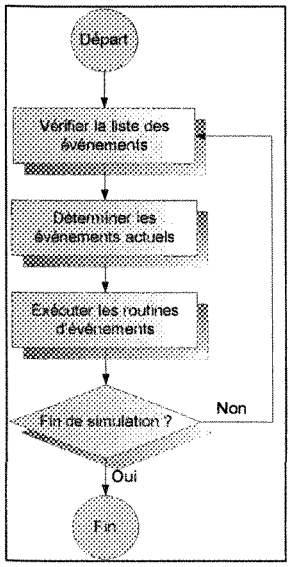

Figure 7 : planification d'événements

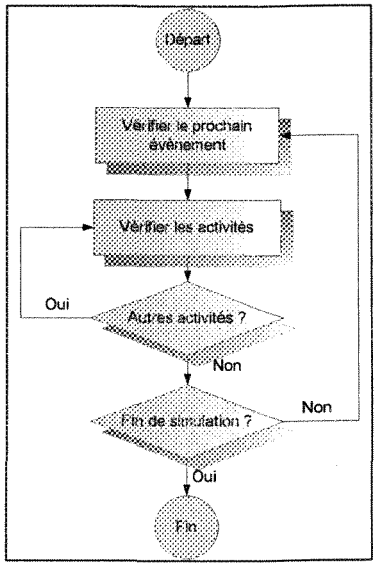

Figure 8 : recherche d'activités

1.6.3.1 Planification d'événements

C'est l'une des approches la plus souvent utilisée en modélisation de systèmes [Law et Kelton, 2000]. Elle fait en sorte que la logique du modèle est exprimée à partir de *routines d'événements*. Il s'agit en fait de programmes qui définissent les *conséquences* ou *actions* des événements, comme la modification des variables d'état ou la planification d'autres événements [Pidd, 1998].

Le contrôleur d'un modèle fondé sur cette approche doit avancer l'horloge de la simulation au prochain événement [Banks et al, 1998]. Ainsi, on assume qu'il maintienne une *liste d'événements*. Chaque élément de la liste contient une référence à la routine événementielle appropriée. Des éléments sont ajoutés lorsqu'un événement est planifié. Le fonctionnement d'un modèle de type

planification d'événements correspond à l'algorithme qui suit [Pidd, 1998] :

```
Répéter jusqu'à ce que la simulation soit terminée :
   1. Vérifier la liste des événements pour déterminer le prochain à se
      produire et déplacer l'horloge de la simulation au moment où il
      survient.
   2. Conserver le temps de l'horloge constant, déterminer tous les
      événements qui se produisent au temps de simulation actuel.
   3. Exécuter toutes les routines des événements actuels, afin de
      déterminer leurs conséquences.
```

Lorsqu'on modélise des systèmes complexes avec cette méthode, il est souvent difficile de considérer toutes les *conséquences* possibles au sein des routines d'événements. Par contre si l'on compare la planification d'événements à d'autres approches, comme la recherche d'activités, on constate que son exécution a tendance à être plus rapide.

1.6.3.2 Recherche d'activités

La recherche d'activités prend pour acquis que tous les événements sont conditionnels à des prémisses. Ainsi, lorsque les prémisses des événements sont rencontrées, des *activités* ou *actions* sont déclenchées [Banks et al, 1998]. Notons que cette approche considère même les événements *planifiables* de type *B* (« Bound to happen ») comme étant conditionnels. En fait, on leur associe une prémisse qui fait en sorte qu'ils se produisent uniquement lorsque l'horloge de la simulation a atteint un instant donné.

Le contrôleur d'un modèle qui implémente cette méthode n'a généralement pas besoin de maintenir une liste d'événements. Il suffit de connaître le moment de la prochaine activité. Toutefois, il est possible d'avoir recours à une liste. Cependant, contrairement à la *planification d'événements*, celle-ci contiendra seulement le temps de déclenchement de l'activité, comme nous n'avons pas besoin de référer à une routine d'événements [Pidd, 1998]. Il faut donc examiner tous les événements, pour déterminer ceux dont les conditions sont respectées. En résumé, le principe d'un contrôleur basé sur la recherche d'activités est le suivant :

```
Répéter jusqu'à ce que la simulation soit terminée :
   1. Vérifier quel est le prochain événement à se produire, en se basant
      sur la liste ou en parcourant tous les événements possibles, afin
```

d'obtenir celui qui a le temps minimal et déplacer l'horloge à l'instant prescrit.
2. Vérifier toutes les activités, pour déterminer celles qui peuvent être déclenchées et le cas échéant exécutez-les. Répétez cette étape, jusqu'à ce qu'il n'y ait plus d'activités à déclencher au temps simulé.

L'exécution d'un modèle conçu à l'aide de cette approche a tendance à être lente, comme on doit tester toutes les conditions des événements [Pidd, 1998]. Sa beauté réside d'avantage dans la simplicité de l'implémentation du contrôleur. Cette méthode est aussi dite à *deux phases*, comme elle est composée de deux étapes distinctes [Banks et al, 1998].

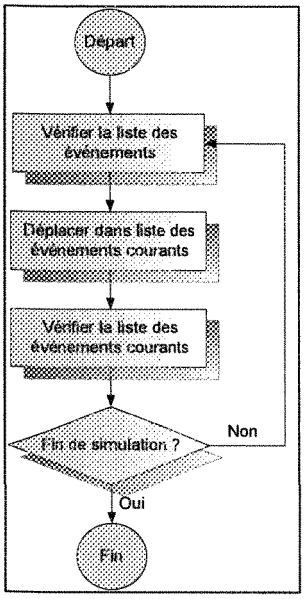

Figure 9 : méthode basée sur des processus

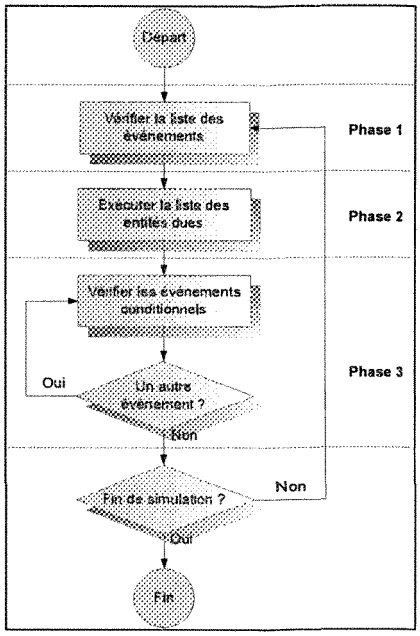

Figure 10 : méthode en trois phases

1.6.3.3 Méthode basée sur des processus

La méthode basés sur des processus fait en sorte que le programme émule le flot des objets à travers le système [Banks et al, 1998]. Contrairement aux autres méthodes qui atomisent le processus en événements ou en activités, celle-ci considère l'ensemble des opérations d'une classe d'entité. Ainsi, lorsqu'une entité est créée, le contrôleur lui applique le processus de sa classe. Par la suite, il se charge d'avancer l'entité à travers la séquence d'opérations du processus.

La progression d'une entité peut être interrompue par des délais *inconditionnels* ou *conditionnels* [Pidd, 1998]. Les premiers impliquent que l'entité s'arrête pour une période de temps quantifiable, par exemple à partir d'une distribution de probabilités. Un tel délai s'apparente donc à un événement de type *B* (« Bound to happen »). Quant aux délais *conditionnels*, ils font en sorte qu'il y a interruption, jusqu'à ce qu'une condition spécifique soit satisfaite. Par exemple, une anode peut demeurer en attente sur un convoyeur, jusqu'à ce qu'elle l'ait entièrement traversé et qu'un poste en aval soit disponible pour la traiter.

Cette notion d'interruption d'un processus oblige le contrôleur à maintenir deux informations : le *temps de réactivation* d'une entité, c'est-à-dire l'instant au cours duquel l'interruption prend fin, ainsi que le *point du processus* où elle se trouve. Ces informations sont enregistrées au sein de deux listes [Balci, 1988]. La première est la *liste des événements futurs*. Elle contient toutes les entités qui sont interrompues par des délais inconditionnels et dont le temps de réactivation est supérieur au temps actuellement simulé. La seconde liste est celle des *événements courants*. Elle est formée des entités interrompues par des délais inconditionnels et avec un temps de réactivation égal au temps actuel de l'horloge. De plus, elle incorpore les entités qui sont assujetties à des délais conditionnels. Ces deux listes permettent au contrôleur d'opérer comme suit :

1. Déterminer le prochain événement à partir de la *liste des événements futurs* et avancer l'horloge de la simulation à l'instant prescrit.
2. Déplacer les entités de la *liste des événements futurs*, dont le temps de réactivation est égal au nouveau temps de l'horloge, vers la *liste des événements courants*.
3. Tenter de faire progresser les entités de la liste des *événements courants* à travers leurs processus respectifs.
   3.1. Les entités qui progressent vont soit compléter leur processus ou s'interrompre, en raison d'un délai *conditionnel* ou *inconditionnel*.
   3.2. Les entités avec un délai *inconditionnel* sont déplacées vers la *liste des événements futurs*. Le contrôleur note leur prochain point de réactivation.

On ne doit pas nécessairement modéliser tous les processus d'un système. Par exemple, cela ne vaut généralement pas la peine de considérer le processus des ressources qui servent d'autres entités [Pidd, 1998]. Par contre, pour certains systèmes complexes, il importe de le faire. Or, lorsqu'on a un système avec plusieurs processus, certains d'entre eux peuvent intervenir sur

les délais *conditionnels* et *inconditionnels* d'autres processus et vice-versa. C'est ce qu'on appelle une approche de type *interaction de processus* [Banks et al, 1998]. Son implémentation est fort complexe et dépasse largement le cadre de celle qui a été décrite plus haut.

1.6.3.4 Méthode en trois phases

Un contrôleur qui repose sur cette méthode maintient trois informations sur chaque entité [Pidd, 1998]. Il y a le *temps cellulaire* (de l'Anglais « Time Cell »), c'est-à-dire l'instant au cours duquel un événement planifiable de type *B* (de l'Anglais « *Bound to happen* ») se produit. Ensuite, on utilise une valeur booléenne pour établir la *disponibilité* de l'entité. Lorsqu'elle est libre, le *temps cellulaire* n'est pas représentatif. Par contre si l'entité n'est pas *disponible*, c'est qu'elle est engagée pour un événement de type *B* survenant au *temps cellulaire.* La dernière information est une référence vers le *prochain événement* planifiable de l'entité. Les trois phases de la méthode sont les suivantes [Balci, 1988] [Banks et al, 1998].

La *phase 1* débute par la recherche du prochain événement à travers la *liste des événements*. L'*horloge de la simulation* est maintenue constante au temps spécifié, jusqu'à la prochaine *phase 1*. En utilisant les trois informations décrites plus haut, le contrôleur recherche les entités *non-disponibles* avec le *temps cellulaire* minimum. Si certaines de ces entités ont un temps cellulaire égal au temps simulé, alors on les ajoute à une *liste d'entités échues* au temps actuel.

La phase 2 consiste à parcourir la *liste des entités échues*, afin d'exécuter leurs événements planifiables respectifs. Le contrôleur doit donc enlever chaque entité de la *liste des entités échues*, affecter leur état de *disponibilité* à vrai et exécuter l'événement de type *B* associé. Ceci peut faire en sorte qu'un nouvel événement soit planifié, pour l'entité traitée ou une autre.

Lors de la phase 3, le contrôleur vérifie les conditions des événements de type *C* (de l'Anglais « *Cooperative* » ou « *Conditional* ») qui sont remplies. Leurs actions sont ensuite exécutées. La vérification des événements conditionnels se poursuit, jusqu'à ce qu'il n'y ait plus aucun d'entre eux qui soient satisfaits.

La *méthode en trois phases* est moins rapide que la *planification d'événements*, car elle oblige un examen de tous les événements conditionnels, de type *C*, à chaque fois qu'un événement planifiable, de type *B*, est déclenché. Par contre, elle est beaucoup plus performante que la recherche d'activités, car les événements planifiables et conditionnels sont différenciés.

La *méthode en trois phases* s'apparente à celle *basée sur des processus*, puisqu'ils considèrent des événements équivalents : *inconditionnel* (i.e. planifiables) et *conditionnels*. Cependant, la *méthode en trois phases* se distingue, car elle exécute les événements *planifiables* (phase 2) d'abord et les *conditionnels* ensuite (phase 3). Ceci peut éviter une situation d'interblocage, où une ressource est en attente d'entités qui doivent être traitées. En effet, comme on l'a mentionné plus tôt, les événements *planifiables* libèrent généralement les entités d'une activité et les *conditionnels* contribuent à les engager [Pidd, 1998].

1.6.4 Architecture classique d'un modèle à événements discrets

L'approche de type *planification d'événements* est généralement utilisée, lorsqu'on conçoit un modèle à événements discrets. Or bien que de tels modèles soient appliqués à différentes catégories de systèmes, il n'en demeure pas moins qu'ils ont souvent les mêmes composantes de bases. Cette section a pour but de présenter les principaux composants d'un modèle à événements discrets axé sur une approche de type *planification d'événements*.

- *État du système* : « Il s'agit d'une collection de variables qui est utilisée afin de décrire le système à un moment particulier de la simulation ». NDLR : traduit de l'Anglais [Law et Kelton, 2000].
- *Horloge de la simulation* : « C'est une variable qui permet de connaître la valeur courante du temps simulé. » NDLR : traduit de l'Anglais [Pidd, 1998].
- *Liste ou calendrier d'événements* : Sous sa forme la plus simple, il s'agit « d'une liste chronologique contenant le temps de la prochaine occurrence pour chaque type d'événements simulés ». NDLR : traduit de l'Anglais [Law et Kelton, 2000]. Dans le cadre d'un mécanisme *événement suivant*, c'est le balayage de cette liste, à la fin de l'exécution des événements, qui

permet d'avancer l'horloge. Par exemple, considérons un bassin de refroidissement qui traite un ensemble d'anodes acheminées par un convoyeur. D'un point de vue minimaliste, il existe deux types d'événements : l'arrivée et le départ d'une anode. La liste des événements doit donc comporter deux éléments, afin de représenter le temps de la prochaine arrivée et du prochain départ. Sous sa forme plus complexe, il peut s'agir d'une liste contenant toutes les occurrences d'événements futurs. Plusieurs éléments de la liste peuvent donc être associés à un même type d'événements.

- *Compteur statistique* : « Ce sont des variables utilisées pour contenir des informations statistiques à propos de la performance du système » NDLR : traduit de l'Anglais [Law et Kelton, 2000]. Généralement, le contenu des compteurs statistiques est affiché dans l'interface du modèle, afin que l'usager puisse voir l'évolution de la simulation.
- *Routine d'initialisation* : « C'est un sous-programme dont le rôle est d'initialiser l'état du modèle au temps 0 » NDLR : traduit de l'Anglais [Law et Kelton, 2000]. Plus précisément, la routine est chargée de mettre à zéro l'horloge de la simulation, d'initialiser les variables d'état, les compteurs statistiques et la liste des événements.
- *Routine de chronométrage* : « C'est un sous-programme qui détermine le prochain événement de la liste, il est aussi responsable de l'avancement de l'horloge, lorsque cet événement se produit » NDLR : traduit de l'Anglais [Law et Kelton, 2000].
- *Routine d'événements* : « Il s'agit d'un sous-programme qui met à jour l'état du système, lorsqu'un type d'événements particulier survient. Il y a généralement une routine par type d'événements » NDLR : traduit de l'Anglais [Law et Kelton, 2000].
- *Librairie mathématique* : « C'est un ensemble de sous-programmes qui servent à générer le comportement aléatoire d'une simulation, à partir de distributions de probabilités intégrées au modèle » NDLR : traduit de l'Anglais [Law et Kelton, 2000].

- *Générateur de rapport* : « C'est un sous-programme qui calcule les mesures de performances du système, en se basant sur les compteurs statistiques. Il sert aussi à produire un rapport au terme de la simulation » NDLR : traduit de l'Anglais [Law et Kelton, 2000].
- *Contrôleur ou programme principal* : « C'est un sous-programme qui invoque la routine de chronométrage, afin de pouvoir déterminer le prochain événement et transférer le contrôle à la routine d'événements appropriée. Il peut aussi vérifier la fin de la simulation et au besoin appeler le générateur de rapport » NDLR : traduit de l'Anglais [Law et Kelton, 2000].

1.7 L'approche orientée objet et la modélisation

L'utilisation du paradigme objet, en modélisation de systèmes, remonte à l'avènement de Simula, vers la fin des années soixante. Il s'agit du premier langage orienté objet. À l'époque, on utilisait surtout Simula pour concevoir des modèles d'événements discrets [Meyer, 2000]. Aujourd'hui, lorsqu'on fait allusion à cette approche, on parle généralement d'OOS (acronyme de l'Anglais « Object Oriented Simulation »). La OOS est une méthode qui consiste à modéliser un système à partir de ses objets (entités, ressources, etc.) qui interagissent entre eux, à mesure que la simulation avance [Banks et al, 1998].

1.7.1 Caractéristiques d'un modèle orienté objet

Pour qu'un modèle soit considéré comme orienté objet, il doit comporter certaines caractéristiques [Pidd, 1998] [Zobrist et Leonard, 1997], notons : l'*abstraction*, l'*encapsulation*, l'*héritage* et le *polymorphisme*.

L'*abstraction* est un processus qui permet de structurer un problème, sous la forme de types abstraits appelés *classes*. Celles-ci contiennent des *attributs* (données) et des *méthodes* (opérations) qui définissent un ensemble d'*objets*. Plus précisément, on dit qu'un *objet* est une instance de *classe* [Meyer, 2000]. Les *attributs* servent à décrire l'état d'un objet à un moment précis. Quant aux méthodes, elles représentent les opérations qu'un objet peut accomplir [Law et

Kelton, 2000]. Dans un contexte d'OOS, on associe les méthodes aux activités et événements d'un système. Quant aux attributs, ils symbolisent l'état interne du système [Pidd, 1998].

L'*encapsulation* est un principe qui fait en sorte que les *attributs* d'un *objet* peuvent uniquement être *modifiés* par ses propres *méthodes*. Les autres objets, qu'ils aient la même définition de classe ou non, peuvent seulement lire ses *attributs*. Les objets agissent donc comme des entités totalement indépendantes grâce à cette propriété [Pidd, 1998].

La capacité de définir une nouvelle *classe* à partir d'une autre est ce qu'on appelle l'*héritage*. Ce comportement évite de constamment réinventer la roue, puisqu'on peut concevoir une nouvelle classe qui descend d'une *classe* ancêtre prédéfinie. Cette nouvelle *classe* incorpore généralement des caractéristiques additionnelles [Banks et al, 1998] [Zobrist et Leonard, 1997].

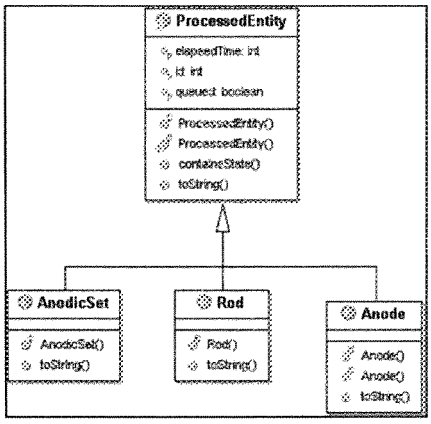

Figure 11 : exemple d'héritage et de polymorphisme

Finalement, la notion de *polymorphisme* fait en sorte que des objets, ayant une même classe ancêtre et des méthodes du même nom, adoptent un comportement différent, lorsqu'on invoque leurs méthodes homonymiques respectives [Law et Kelton, 2000]. Voici un exemple simple d'héritage et de polymorphisme dans le cadre du cycle de vie des anodes. Supposons une classe ancêtre qui représente les entités traitées par le système (*ProcessedEntity*). Les classes *Anode*, *AnodicSet* (Ensemble anodique) et *Rod* (Tige) héritent de la classe ancêtre *ProcessedEntity*. Ces classes ont une méthode *toString()*. Celle-ci retourne une chaîne de caractères qui donne l'état de l'objet. Cependant, cette méthode a un comportement polymorphe, car le format des informations

retournées va être particulier à chaque classe. La représentation UML [Fowler, 2004] d'une telle structure est illustrée ci-dessus.

1.7.2 Modèle orienté objet, cadre et cadriciel

L'idée de modélisation orientée objet est fort intuitive, puisqu'il est facile d'imaginer un système comme un amalgame d'objets [Banks et al, 1998]. Ainsi, on peut concevoir les différentes entités et ressources du cycle de vie des anodes, comme un ensemble d'objets : tour à pâte, bassin, vibrocompacteur, presse, table de transfert, fours, entrepôt, atelier de scellement, convoyeurs, anodes, tiges, ensembles anodiques, etc.

L'approche orientée objet permet d'accroître l'extensibilité et la modularité du modèle, en décentralisant chacune de ses portions, grâce aux notions d'abstraction et d'encapsulation. Ainsi, chaque classe agit comme une entité indépendante. Ceci implique que les changements sur l'une d'elles minimisent les répercussions sur les autres [Meyer, 2000]. Pour les même raisons, cette approche favorise également la réutilisabilité du modèle. En effet, comme les classes d'objets agissent comme des entités autonomes, on peut les réutiliser pour plusieurs domaines d'application connexes [Law et Kelton, 2000]. Par exemple si le contrôleur d'un modèle est encapsulé au sein d'un ensemble de classes, on peut l'utiliser plusieurs fois.

La notion d'héritage accroît aussi la réutilisabilité et la modularité du modèle [Zobrist et Leonard, 1997]. En effet, cette caractéristique permet d'élaborer une hiérarchie de classes. Celle-ci est formée à son sommet par les classes ancêtres qui représentent les objets les plus abstraits du système. Par la suite, on ajoute à cette hiérarchie des classes spécifiques, qui héritent des caractéristiques des classes abstraites et ainsi de suite. Cette notion d'héritage hiérarchique peut être représentée sous la forme d'une structure arborescente, comme celle du diagramme UML de la Figure 11.

Un regroupement de classes, d'un même niveau d'abstraction, est ce qu'on appelle un *cadre* (traduction de l'Anglais « *frame* ») [Banks et al, 1998]. Les *cadres* sont des librairies de classes utilisées lors de la mise en œuvre d'un modèle orienté objet. Contrairement à la classe, le *cadre*

n'est pas nécessairement une structure qui est implémentée dans un langage de programmation. Un *cadre* peut tout simplement être une représentation conceptuelle d'un niveau d'abstraction.

Lorsqu'on modélise à un plus haut niveau, il peut être avantageux de regrouper les *cadres* en *cadriciel* (traduction de l'Anglais « *framework* »). Un *cadriciel* est défini ainsi :

> Un ensemble de classes abstraites collaborant entre elles, pour faciliter la création de tout ou d'une partie d'un système logiciel. Un cadriciel fournit un guide architectural en partitionnant le domaine visé en classes abstraites et en définissant les responsabilités de chacune, ainsi que les collaborations entre classes. [Cadriciel]

Le cadriciel est un peu un coffre à outils générique qui permet d'adresser des problèmes similaires. La principale différence entre une hiérarchie de classes et un cadriciel est que la première est basée sur des relations d'héritage, alors que la seconde, s'appuie aussi bien sur des relations de composition ou de collaboration entre des classes [Banks et al, 1998] [Fowler, 2004].

1.7.3 Modèle basé sur des composants logiciels

En génie logiciel, on définit un composant logiciel comme : « un élément d'un système qui offre un service prédéfini et qui est capable de communiquer avec d'autres composants. » NDLR : traduit de l'Anglais [Software Component]. D'un point de vue minimaliste, un composant doit posséder les caractéristiques suivantes : usage multiple, non-spécifique à un contexte particulier, formé d'autres composants, encapsulé (voir la section 1.7.1), indépendant du déploiement et des différentes versions [Szyperski, 1999].

Les composants sont souvent des objets conçus à partir d'une spécification formelle. Les trois spécifications les plus connues [Young et Tag, 2001] sont : COM (« Component Object Model ») de Microsoft [COM], CORBA (« Common Object Request Broker Architecture ») de l'OMG [CORBA] et JavaBean de Sun Microsystems [JavaBeans].

	COM	CORBA	JavaBeans
Plate-forme	Windows uniquement	Plusieurs dont AIX, Linux, Solaris et Windows	Plusieurs dont Linux Solaris et Windows
Principaux langages de programmation	Plusieurs dont C, C++, Delphi, Visual Basic	Plusieurs dont C, C++, Delphi, Java	Java uniquement
Origine de la spécification	Échange d'information entre applications	Environnement distribué	Machine virtuelle Java

Tableau 1 : spécifications pour concevoir des composants

Les composants basés sur une spécification possèdent généralement une interface définie dans un langage particulier appelé IDL (de l'Anglais « Interface Description Language ») [Software Component]. Cette interface leur permet de se comporter de façon autonome et d'interagir entre eux, quelque soit la plate-forme ou le langage utilisé [Szyperski, 1999].

Pour bien des auteurs, la notion de composant est indissociable de l'approche orientée-objet. Selon les tenants de ce principe : « Un composant est ensemble de classes fortement liées qui agissent comme une unité » [Batory et O'Maley, 1992]. Alors que d'autres affirment que les composants sont des briques logicielles préfabriquées, qui peuvent être assemblées ensembles, un peu comme les composants d'un circuit électronique ou d'un bâtiment. On parle donc d'une notion qui va au-delà de l'approche orienté objet. Cette nouvelle approche est dite *orienté composante* [Software Component]. Dans le contexte de ce mémoire, nous ne comptons pas discourir sur ces aspects sémantiques. Toutefois, nous avons intégré la présente discussion, au sein d'une section sur les modèles orienté objets, car les composants sont souvent implémentés à partir de classes d'objets. Celles-ci sont généralement immutables et capturent un état initial par défaut [Szyperski, 1999].

Les composants sont aussi utilisés en modélisation. D'une part, parce qu'ils facilitent la division d'un système complexe en sous-systèmes élémentaires. Ensuite, parce qu'ils promeuvent la réutilisation logicielle et la simplification de la mise en œuvre du modèle [Pidd et al, 1999]. Ainsi, il devient possible de prendre des composants, répertoriés au sein d'une librairie ou d'un cadriciel et de les intégrer à l'environnement de travail du modélisateur. De tels composants peuvent représenter les objets d'un modèle (ressources, entités, files, etc.), afficher des informations à

l'écran, générer des rapports, effectuer des animations ou retourner une valeur issue d'une distribution de probabilités. Généralement, en modélisation de systèmes, on dit que les composants d'un cadriciel ont les particularités suivantes : capacité d'échanger des informations à l'aide d'un système de messagerie, simplicité de configuration, à partir d'un ensemble de paramètres et capacité d'interconnexion à des fins d'assemblage, grâce à une interface standardisée [Praehofer et al, 1999].

1.7.4 Modélisation et le formalisme DEVS

L'un des pionniers en matière de modélisation basée sur des composants est le docteur Bernard Ziegler de l'Université de l'Arizona. En 1976, il inventa une spécification formelle appelée DEVS (de l'Anglais « Discrete Event System Specification ») [Zeigler, 1995]. DEVS est en fait un formalisme théorique qui sert à spécifier un système discret et ce, au même titre que les équations différentielles peuvent le faire pour un système continu. Ainsi, on peut concevoir des modèles hiérarchiques et modulaires, qui implémentent le formalisme DEVS, à l'aide d'un langage de programmation et de standards logiciels [Zeigler et Sarjoughian, 2005]. DEVS permet de modéliser beaucoup plus facilement et efficacement un système, en le divisant en *composant* de *modèles,* en spécifiant le couplage entre chacun d'eux et en fournissant un cadre spécifique qui facilite la réutilisation des modèles [Young et Tag, 2001].

Il y a deux sortes de modèles du point de vue du formalisme DEVS [Zeigler et Sarjoughian, 2005]. Le premier est le modèle *atomique* ou de *base* qui permet de décrire le comportement dynamique d'un *composant de modèles.* Ainsi, on doit percevoir un *composant de modèles* comme une entité qui interagit avec son environnement, par le biais de *ports entrants* et *sortants.* Plus spécifiquement, lorsque des événements externes au *composant de modèles* sont reçus, à travers ses *ports entrants*, celui-ci détermine une réponse appropriée. De même, lorsque des événements internes contribuent à la modification de son état, ceux-ci doivent être transmis aux *ports sortants*, pour que les autres *composants de modèles* soient notifiés du changement. Le *modèle atomique* (M.A.) du formalisme DEVS possède la structure suivante [Zeigler, 1995] :

M.A. = $\langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$, où

X est l'ensemble des ports pour chaque événement entrant

S est l'ensemble des états

Y est l'ensemble des ports pour chaque événement sortant

$\delta_{int} : S \rightarrow S$ est la fonction de transition interne

$\delta_{ext} : Q \times X \rightarrow S$ est la fonction de transition externe, où

$Q = \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ est appelé l'ensemble de l'état total

e est le temps écoulé depuis la dernière transition

$\lambda : S \rightarrow Y$ est la fonction de sortie

$ta : S \rightarrow R^{+}_{0, \infty}$ est la fonction d'avancement du temps

Le tuple ci-dessus a la signification suivante : à un moment quelconque le système est dans un état $s \in S$. Lorsqu'aucun événement externe ne survient, le système demeure dans un état s pour un temps de $ta(s)$. Lorsque le temps d'inactivité est écoulé (c'est-à-dire $e = ta(s)$), le système retourne la valeur $\lambda(s)$ et change d'état pour $\delta_{int}(s)$. Si un événement externe $x \in X$ survient avant le temps d'expiration, c'est-à-dire le système est dans l'état total (s,e) avec $e \leq ta(s)$, alors le système change son état pour $\delta_{ext}(s,e,x)$. Peu importe le type de transition, interne ou externe, une fois qu'il est survenu, le système tombe dans un nouvel état noté s' pendant $ta(s')$. Le processus se poursuit ainsi par la suite.

Le second type de modèles considéré par DEVS est dit *couplé* [Zeigler et Sarjoughian, 2005]. Il permet de définir comment connecter des *composants de modèles*, pour former un nouveau modèle. Ce modèle peut être utilisé à son tour, comme un *composant* d'un *modèle couplé*, donnant ainsi naissance à une hiérarchie de modèles [Young et Tag, 2001]. Le *modèle couplé* (C.M.) de DEVS est défini comme suit :

C.M. = $\langle X, Y, M, EIC, EOC, IC, SELECT \rangle$, où

X est l'ensemble des ports pour les événements entrants

Y est l'ensemble des ports pour les événements sortants

M est l'ensemble de tous les composants de modèles de DEVS

$EIC \subseteq X \times \cup_i X_i$ est la relation de couplage des entrées externes qui sert à connecter les ports entrants d'un composant, avec ceux du modèle couplé.

$EOC \subseteq \cup_i Y_i \times Y$ est la relation de couplage des sorties externes qui sert à connecter les ports sortants d'un composant, avec ceux du modèle couplé.

$IC \subseteq \cup_i X_i \times \cup_i Y_i$ est la relation de couplage interne qui sert à connecter les ports sortants d'un composant aux ports entrants d'un autre.

$SELECT\ 2^M - \phi \rightarrow M$ est une fonction qui choisit un modèle, lorsque 2 modèles ou plus sont couplés simultanément.

Depuis une dizaine d'années, la spécification DEVS a été utilisée en conjonction avec des standards de composants logiciels. Notons les travaux de [Young et Tag, 2001] avec COM, ceux de [Zeigler et al, 1999] avec CORBA ou encore ceux, de [Praehofer et al, 1999] avec JavaBeans. Ces recherches démontrent la possibilité et les avantages d'une intégration de DEVS avec des technologies logicielles qui comportent des caractéristiques modulaires, multiplateformes et multi-langages.

1.8 Systèmes visuels de modélisation interactive

Jusqu'ici, il a été question d'une approche de modélisation classique qui consiste à concevoir un modèle basé sur des événements discrets et à le mettre en œuvre, sous la forme d'un programme qui simule des scénarios d'opération. Cependant, il existe des logiciels de modélisation commerciaux qui sont utilisés pour effectuer de telles tâches. Ces systèmes, appelés VIMS (de l'Anglais « Visual Interactive Modeling System »), permettent d'élaborer un modèle dans un environnement visuel et interactif et d'exécuter des simulations dans ce même environnement [Pidd, 1998].

Notre recherche n'a pas pour but de faire l'apologie ni le procès de ces systèmes. Cependant, nous comptons montrer qu'il est possible de modéliser un système dont le fonctionnement repose sur des connaissances, en utilisant une approche alternative, dont il est question au cours du chapitre 3. En attendant, la présente section vise à présenter brièvement, le fonctionnement et les caractéristiques des VIMS.

1.8.1 Fonctionnement d'un VIMS

Habituellement, un modèle conçu à partir d'un VIMS débute avec un écran vierge, auquel le modélisateur ajoute des icônes qui représentent les principaux objets du système. Les icônes sont ensuite liés ensembles, pour former un réseau qui capture les interactions logiques des objets du système [Law et Kelton, 2000]. Globalement, il existe deux approches pour modéliser un tel réseau.

La première approche est dite *basée sur les machines* [Pidd, 1998], puisque chaque icône représente une machine, c'est-à-dire une ressource ou un groupe de ressources. Quant aux liens, ils montrent l'itinéraire d'entités passives qui parcourent le système. Par exemple, dans le cadre de la modélisation du cycle de vie des anodes, une ressource équivaut à un poste du système (tour à pâte, bassin de refroidissement, fours, entrepôt, etc.) qui traite des anodes, c'est-à-dire les entités passives. Ce type de réseau convient quand les entités cheminent à travers un processus complexe ou lorsque les ressources effectuent plusieurs activités.

La seconde approche consiste à élaborer un réseau *basé sur des tâches* ou *activités* [Pidd, 1998]. Chaque icône du modèle représente une activité qui implique une ou plusieurs entités ou encore, une ou plusieurs ressources. Ainsi, les tâches sont liées logiquement pour montrer leur degré de dépendance. Cette approche convient lorsque l'aiguillage des entités est relativement simple ou lorsqu'il y a coopération entre plusieurs ressources

1.8.2 Caractéristiques d'un VIMS

Il existe plusieurs caractéristiques souhaitables, lorsqu'il vient le temps de sélectionner un VIMS. En voici quelques-unes selon [Law et Kelton, 2000] :

- Flexibilité de modélisation : Ce critère doit permettre de modéliser un système, peu importe son degré de complexité. Ainsi, il doit être entre autre possible : de définir et changer les attributs d'entités ainsi que les variables globales du modèle, d'avoir recours à des fonctions mathématiques et de créer de nouvelles structures de modélisation.
- Facilité d'utilisation et d'apprentissage : Une interface graphique conviviale et fonctionnelle est de rigueur avec un VIMS. Par exemple, il importe de pouvoir utiliser des constructions visuelles, comme des icônes ou des blocs, qui sont ni trop primitifs ni trop complexes.
- Modélisation hiérarchique : Cette caractéristique est importante afin de pouvoir réutiliser des parties d'un modèle et les combiner en structures hiérarchiques.
- Facilité de débogage : Cet aspect permet de suivre une entité à travers le modèle, de voir l'état du modèle, lorsqu'un événement survient et d'affecter automatiquement une valeur à un attribut ou une variable.
- Vitesse d'exécution : Cette caractéristique intervient principalement durant la phase de simulation.
- Possibilité d'exportation ou d'importation : Il est avantageux d'importer (ou d'exporter) des données d'un (ou vers un) tableur ou une base de données.
- Simulation de scénarios basés sur un paramètre : En changeant un paramètre particulier (comme par exemple, le nombre maximal d'entités d'une file), on peut voir immédiatement son impact, notamment sur certaines mesures de performances (par exemple, le temps moyen d'une entité dans le système).
- Combinaison discret et continu : Pour certains modèles basés sur des événements discrets, il peut être avantageux de simuler certaines parties en tant qu'un phénomène continu.
- Routines externes : Parfois, la logique d'un modèle est si complexe qu'il importe de la mettre en œuvre dans un langage de programmation, d'où l'intérêt de pouvoir référencer des routines externes.

- État initial : Lorsqu'on conçoit un modèle d'un système manufacturier, il est souvent utile de pouvoir l'initialiser dans un état non-vide, par exemple où toutes les ressources traitent des entités et où les files sont à demies pleines.
- Sauvegarde de l'état terminal : Au terme de la simulation, il est souvent utile de préserver l'état final, afin de pouvoir redémarrer à nouveau, à partir de celui-ci.
- Coûts : En plus des coûts associés au VIMS, il importe de considérer ceux attribuables au support technique, à l'installation, à la maintenance, aux mises à jour, au matériel informatique et aux logiciels additionnels.
- Configuration recommandée : Cette considération découle d'aspects tels que : la compatibilité avec un ou plusieurs système(s) d'exploitation(s), la quantité de mémoire vive ou la puissance du processeur qui sont recommandées.
- Animation et graphiques dynamiques : Les objets clés du système doivent idéalement être représentés par des icônes qui changent de position, de couleur et/ou de forme à mesure que la simulation avance. Il existe deux types d'animation : *concurrente* et *post-traitée*. L'animation *concurrente* fait en sorte que l'affichage est rendu en même temps que se déroule la simulation. Ceci peut ralentir l'exécution. Le second type implique que les changements d'état sont préservés sur disque et utilisés pour un affichage subséquent. Lorsque l'animation est *concurrente*, il est utile de pouvoir interrompre la simulation et la redémarrer plus tard.
- Capacités statistiques : Le VIMS doit normalement posséder un générateur de nombres aléatoires. Lorsqu'on l'adjoint à une distribution de probabilité, on peut obtenir des valeurs pour certaines activités ou phénomènes associées au modèle. Si une distribution de probabilité ne peut être utilisée, il est souhaitable de recourir à une distribution empirique, basée sur les données du système. La plupart des VIMS devraient aussi permettre d'établir un intervalle de confiance, pour une mesure de performance, comme le temps passé dans le système [Baillargeon, 1990].

- Support aux usagers et la documentation : Cette caractéristique comprend le support téléphonique ou par courriels, le guide de l'usager, l'aide en ligne avec des exemples, la formation, les démos, les mises à jours, les conférences, etc.
- Génération de rapports et de graphiques : Les rapports doivent être fournis pour les principales mesures de performance du système. Il est souhaitable de pouvoir les configurer. Des graphiques statiques doivent être aussi fournis sous plusieurs formes : histogramme, avec barres, pointe de tarte, courbes, etc.

1.8.3 Arena un exemple de VIMS

Au moment de l'écriture de ce mémoire, Arena de Rockwell Automation [Arena1] est un des VIMS les plus populaires dans le monde de la modélisation de systèmes. Arena est abondamment utilisé pour modéliser les processus de l'industrie, comme les installations manufacturières. Il permet d'analyser la performance actuelle et les améliorations qui peuvent être apportées aux processus [Arena2]. Arena est disponible en plusieurs éditions : *Basic Edition*, *Professional Edition*, *Factory Analyzer*, *Contact Center* et *3D Player* [Arena1]. Cependant, la présente discussion est principalement orientée sur la première édition.

Avec Arena, l'utilisateur construit un modèle expérimental, en plaçant des modules (c'est-à-dire des boîtes de différentes formes) qui représentent les processus ou la logique d'un système. Les connecteurs qui les relient spécifient le flot des entités [Arena2]. Les modules sont configurés à l'aide de boîtes de dialogue. Un modèle peut avoir un nombre illimité de niveaux de hiérarchie. On peut afficher des animations en deux dimensions et des graphiques dynamiques ou statiques (histogrammes, avec barres ou courbes). Arena dispose d'importantes capacités statistiques. Ainsi, il est possible de modéliser des files d'attentes complexes, par exemple à partir de processus non-stationnaires de Poisson (pour les systèmes dont l'arrivée des entités se fait à un taux variable) et de construire des intervalles de confiance pour les mesures de performances [Law et Kelton, 2000].

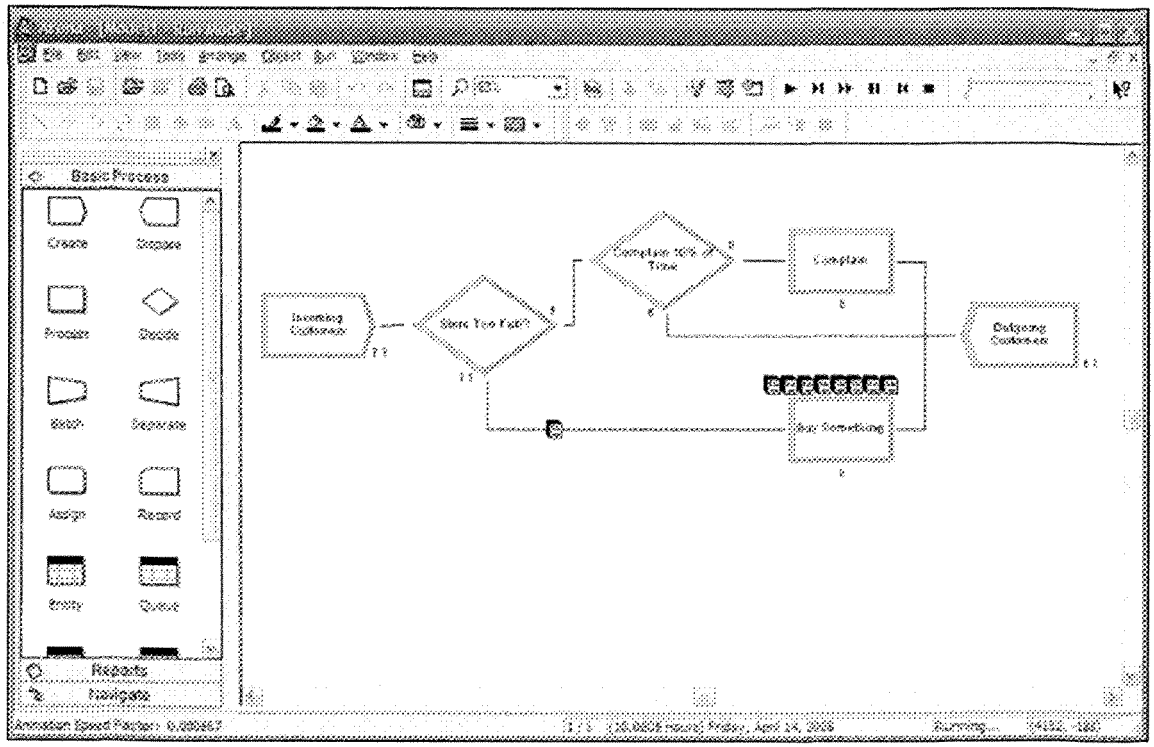

Figure 12 : un modèle conçu avec Arena 9.0

Arena est uniquement compatible avec la famille de systèmes d'exploitation Windows [Arena3]. Conséquemment, il s'intègre aisément aux technologies de Microsoft. Si on a besoin d'algorithmes plus spécifiques, il est possible de recourir à des routines VBA (Visual Basic for Applications) [VBA]. Arena supporte également l'importation de diagrammes Visio, l'importation/exportation de feuilles de calcul Excel ou de bases de données Access et la génération de rapports, à l'aide de Crystal Report. Il est aussi possible d'ajouter des composants COM (ActiveX) [COM] à l'environnement de modélisation [Arena2].

1.8.4 La modélisation classique et les VIMS

Un VIMS, comme Arena, possède plusieurs fonctionnalités qui simplifient le processus de modélisation d'un système, comparativement à une approche classique qui consiste à mettre en œuvre le modèle, sous la forme d'un programme informatique. D'ailleurs selon [Law et Kelton, 2000], un VIMS diminue le temps d'implémentation du modèle, il fournit les éléments à la base de

sa conception (voir la section 1.6.4), en plus de faciliter sa maintenance, sa modification et la détection des erreurs.

En contrepartie, le fonctionnement d'un VIMS n'est pas nécessairement connu par les spécialistes en modélisation. Dans ce contexte, l'apprentissage et la formation d'un ou plusieurs spécialistes sur différents VIMS complexifie le processus de modélisation. Néanmoins, la plupart des spécialistes connaissent au moins un langage de programmation [Law et Kelton, 2000]. Or, il existe des cadriciels de simulation, mis en œuvres dans différentes langages de programmation, qui possèdent des fonctionnalités similaires aux VIMS [L'Écuyer et al., 2002]. Certains de ces cadriciels sont mêmes gratuits. Ainsi, on peut se questionner sur la pertinence des VIMS, lorsqu'on sait qu'un cadriciel ou une librairie de simulation peut convenir dans bien des cas.

Finalement, mentionnons que les VIMS sont beaucoup moins flexibles [Law et Kelton, 2000]. D'ailleurs, avec Arena, on utilise souvent des modules de code VBA [Arena2], pour répondre à des besoins qui ne peuvent être mis en œuvres avec ses fonctionnalités de base. C'est d'ailleurs le cas de notre problématique, puisque une partie du système à modéliser s'appuie sur la notion de *connaissance heuristique* [Friedman-Hill, 2003] qui nécessite de recourir à des notions d'intelligence artificielle. À ce titre, l'utilisation de méthodes d'intelligence artificielle, dans le cadre de la modélisation d'un système, fait l'objet de la discussion du chapitre suivant.

CHAPITRE 2

MODÉLISATION BASÉE SUR L'INTELLIGENCE ARTIFICELLE ET LES SYSTÈMES EXPERTS

2 Modélisation basée sur l'intelligence artificielle et les systèmes experts

2.1 Principes d'intelligence artificielle

Avant d'entrer dans le vif du sujet et de présenter les systèmes experts, une question semble s'imposer d'elle-même : qu'est-ce que l'intelligence artificielle (IA) ? La réponse à cette interrogation est assez complexe, car l'intelligence humaine est en soit plutôt difficile à appréhender, alors que dire de celle d'entités artificielles. En fait, selon [Russell et Norvig, 2003], il existe quatre grandes écoles de pensée, lorsqu'il vient le temps de définir la notion d'intelligence artificielle :

- L'étude des systèmes qui pensent comme des humains.
- L'étude des systèmes qui se comportent comme des humains.
- L'étude des systèmes qui pensent rationnellement. On dit qu'un système est rationnel s'il est capable d'effectuer sa tâche correctement, à partir de ce qu'il connaît.
- L'étude des systèmes qui se comportent rationnellement.

Chacune de ces écoles est associée à des domaines d'application particuliers de l'intelligence artificielle, notons : les algorithmes génétiques, la reconnaissance de la parole, les réseaux de neurones, la robotique, les systèmes experts, les systèmes multi-agents, la traduction de langages, ainsi que la vision et la perception. Pour notre part, nous allons principalement retenir la deuxième définition, puisque nous nous sommes intéressés à concevoir un système capable d'émuler le comportement d'opérateurs humains et d'automates machines.

Historiquement les ordinateurs ont toujours excellés à accomplir des tâches répétitives, telles de complexes calculs arithmétiques, ou le stockage et la recherche d'information à travers une source de données. Ces tâches ont comme dénominateur commun un algorithme. Il est donc possible de les accomplir, par le biais d'une suite d'instructions qui produisent la réponse attendue. Pour leur part, les humains ont de grandes aptitudes pour solutionner des problèmes qui impliquent des abstractions et des symboles, plutôt que des nombres. On peut penser à des problèmes tels, la planification d'activités ou la compréhension d'un texte. De ce point de vue, l'intelligence artificielle est donc la science qui permet aux ordinateurs de représenter et manipuler les symboles des

humains, dans le but de solutionner une problématique, difficilement concevable sous une forme algorithmique [Gonzalez et Dankel, 1993].

La section suivante a pour but d'introduire la notion de système expert, puisqu'il s'agit d'un des domaines de l'IA qui a été appliqué avec le plus de succès dans un contexte de prise de décision [Turban et Aronson, 2000].

2.2 Principes d'un système expert

Les systèmes experts (de l'Anglais « Experts Systems ») ont vu le jour d'un point de vue théorique, vers la fin des années cinquante. Ils se concrétisèrent véritablement à un niveau pratique, à partir des années soixante et soixante-dix. À ce titre, on peut penser à DENDRAL [Lindsay, 1980], un système d'interprétation permettant d'identifier des composants chimiques, ainsi qu'à MYCIN [Buchanan et Shortliffe, 1984] qui diagnostiquait et spécifiait des traitements pour les troubles sanguins. Une nouvelle approche a donc été introduite, pour développer des systèmes d'informations capables de solutionner des problèmes, issus d'un domaine bien précis. Ces systèmes se comportaient en partie comme un spécialiste humain, par exemple, un chimiste en ce qui concerne DENTRAL et un immunologiste dans le cas de MYCIN. Au cours des décennies suivantes, les systèmes experts firent l'objet d'un certain engouement, particulièrement vers la fin des années soixante-dix et au début des années quatre-vingt. Ils permirent de solutionner des problématiques reliées à de nombreux champs d'application, citons : la finance, les sciences naturelles et sociales, l'ingénierie, la conception, ainsi que la santé [Gonzalez et Dankel, 1993].

L'origine des systèmes experts découle d'observations des chercheurs en intelligence artificielle (IA). Ceux-ci constatèrent qu'il était avantageux d'axer leur méthode de résolution d'un problème, sur la connaissance du domaine d'application, plutôt que sur un éventail de connaissances générales, appliquées à plusieurs domaines. On considère souvent à tort, que l'appellation système expert est un synonyme d'intelligence artificielle. Cette perception erronée, s'explique du fait que les systèmes experts forment l'une des branches de l'IA qui a eu le plus de succès. Du moins en ce qui a trait aux applications pratiques.

Le docteur Edward Feigenbaum, un éminent chercheur de l'université Stanford qui est spécialisé en Intelligence artificielle, définit les systèmes expert comme suit :

> Des programmes intelligents qui utilisent la connaissance et des mécanismes d'inférences, pour solutionner des problèmes qui sont difficiles, au point de requérir une expertise particulière, lorsqu'ils sont considérés par des êtres humains. NDLR : traduit de l'Anglais [Feigenbaum, 1982]

Donc, l'appellation de ce type de système d'information, s'explique du fait que les connaissances et les méthodes, qui leur permettent de tirer des conclusions, s'apparentent à celles d'un expert du domaine d'application concerné. Par expert, on entend une personne qui possède des connaissances ou des habiletés très spécifiques, c'est-à-dire inconnues par le commun des mortels. À ce titre, on peut penser au savoir contenu dans un ouvrage spécialisé. Ainsi, dans le cadre de la modélisation et de la simulation de procédés industriels, nous nous intéressons aux systèmes experts, parce qu'ils permettent d'émuler la capacité qu'a un spécialiste humain à prendre des décisions.

Mentionnons que le fait de recourir à un système expert, tout comme n'importe quel domaine de l'IA, n'est pas sans risque, puisqu'il implique un niveau de complexité accru. Par exemple, il faut généralement impliquer des spécialistes en IA [Gonzalez et Dankel, 1993], pour travailler de concert avec l'équipe de modélisation ou encore, former les membres de cette dernière, pour qu'ils puissent intégrer efficacement ces technologies au modèle. C'est pourquoi il importe de déterminer si un problème requiert réellement un système expert. Nous allons donc mettre l'accent, tout au long du mémoire, sur le type de problématique qu'un système expert doit normalement adresser.

Il est notable de préciser que les termes *système à base de connaissances* ou *système expert à base de connaissances* sont souvent utilisés, au détriment de l'appellation *système expert*. En fait, pour certains auteurs, ces substantifs sont tous des synonymes. Cependant, d'autres affirment qu'il s'agit d'un abus de langage, puisqu'un *système expert* doit seulement comporter le savoir particulier d'un spécialiste humain, alors qu'un *système à base de connaissances* peut être associé à la fois à des connaissances générales et spécifiques d'un domaine particulier [Giarratano

et Riley, 1998]. Comme ce débat sémantique déborde du contexte de notre recherche et que de toute façon, nous considérons davantage une expertise spécifique, à savoir celle des spécialistes du cycle de vie des anodes, l'appellation système expert est utilisée.

À la base du système expert, se trouve l'utilisateur qui saisit un ensemble de faits, afin d'obtenir un avis ou une expertise. Le *moteur d'inférence* (de l'Anglais « Inference Engine ») compare les faits saisis, avec le savoir de la *base de connaissances* (de l'Anglais « Knowledge Base ») du système expert, dans le but d'établir des correspondances entre eux. Ainsi, les données de la base de connaissances permettent au moteur d'inférence de tirer les conclusions appropriées, en appliquant un processus de raisonnement, dont il sera question un peu plus loin. Les conclusions sont retournées à l'usager sous la forme d'une expertise [Giarratano et Riley, 1998]. La figure ci-dessous, illustre le fonctionnement élémentaire d'un système expert.

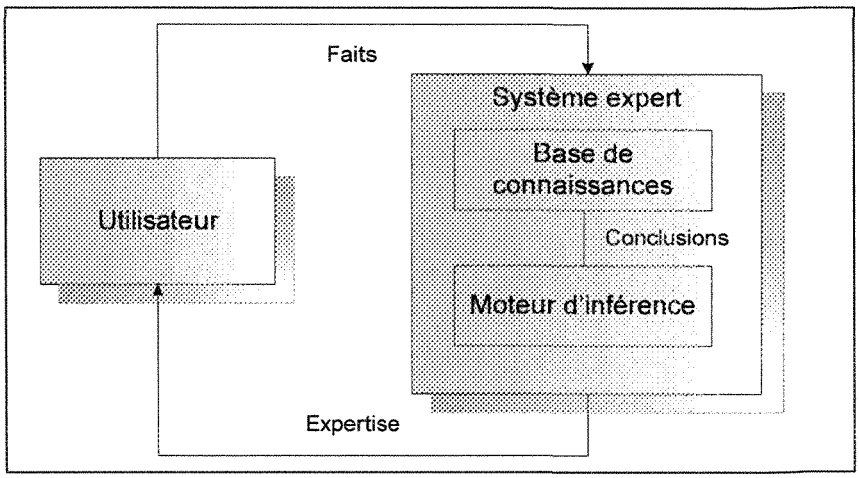

Figure 13 : représentation d'un système expert élémentaire

Au cours des sections suivantes, nous allons donc introduire avec plus de détails, la notion de système expert et leur architecture. Ces sections ont pour objectif d'éclaircir le rôle des systèmes experts, dans un contexte de modélisation de procédés industriels. Ainsi, suite à cette description, nous allons être en mesure de discuter, à partir de la section 3.4.2, de leur intégration avec notre modèle et des avantages qu'ils lui confèrent.

2.2.1 Extraction et représentation des connaissances

La base de connaissances est un élément clé du système expert. Elle contient des informations structurées qui représentent la connaissance spécifique au domaine d'application. L'acquisition des connaissances est effectuée par un spécialiste, appelé *ingénieur de la connaissance* (de l'Anglais « Knowledge Engineer »). C'est à partir d'entretiens exhaustifs, réalisés en compagnie d'un expert du domaine, que ce spécialiste peut extraire le savoir. Par la suite, il devient possible de l'encoder dans le système expert. Ce processus de conception est appelé *ingénierie des connaissances* (de l'Anglais « Knowledge Engineering ») [Gonzalez et Dankel, 1993].

Il existe plusieurs formalismes pour représenter le savoir au sein de la base de connaissances. Chacun d'eux ont leurs avantages et leurs inconvénients, qui doivent être pris en considération par l'ingénieur de la connaissance. Il est cependant notable de mentionner que les règles de type *si ... alors*, s'avèrent être l'une des représentations les plus courantes du savoir de la base de connaissances :

si <Prémisse, antécédent ou *partie gauche* (LHS)>
alors <Action, conséquence ou *partie droite* (RHS)>

Une règle, aussi appelée *règle de production*, est en quelque sorte une commande conditionnelle qui exécute une action dans certaines situations. Les systèmes experts, dont la base de connaissances est conçue à partir de règles, sont appelés des *systèmes à base de règles* ou *systèmes de production* (de l'Anglais « Rule-Based Systems » ou « Production Systems »). Ils permettent d'établir des conclusions selon des *prémisses*, c'est-à-dire la partie conditionnelle, située entre le *si* et le *alors* de la règle. Une prémisse est formée d'une ou plusieurs *conditions*, appelées aussi *motifs* (traduction de l'Anglais « Pattern »). Celles-ci sont généralement séparées par des opérateurs logiques (*et*, *ou*). Souvent, ces conditions contiennent des variables liées à des faits de la mémoire de travail. La prémisse est aussi appelée l'*antécédent*, la *partie gauche* de la règle ou *LHS*, de l'Anglais « Left Hand Side ». Quant aux actions exécutées lorsqu'elle est déclenchée, elles forment la *conséquence*, la *partie droite* ou *RHS* (« Right-Hand Side »).

Généralement, les actions de la RHS servent d'une part, à affirmer ou rétracter des faits et d'autre part, à signaler des informations à l'usager [Friedman-Hill, 2003] [Giarratano et Riley, 1998].

Concernant le procédé étudié, il est possible d'exprimer la connaissance à propos du routage des anodes à l'aide de règles. L'exemple qui suit nous montre une suite de règles génériques formées de plusieurs conditions. Elles permettent de déterminer si une *anode* quelconque (représentée par la variable *A*) traitée par un *poste* (variable *P*), peut être transférée vers un *convoyeur* (variable *C*), situé en aval de ce dernier. Le transfert se réalise une fois que le *temps de traitement requis* sur (*A*) est atteint et si le *poste* (*P*) est *actif* :

```
// Désactivation du poste si un arrêt est planifié.
(R1)  si
            Poste P est "actif" et
            Type d'action TA est un "arrêt" et
            Action planifiée de type TA survient actuellement sur P
      alors
            Rendre poste P "inactif"

// Activation du poste si un arrêt planifié est terminé.
(R2)  si
            Poste P est "inactif" et
            Type d'action TA est un "arrêt" AR et
            Action planifiée de type TA prend actuellement fin sur P
      alors
            Rendre poste P "actif"

// Vérification si l'anode A traitée par le poste P peut être transférée
// vers le convoyeur C.
(R3)  si
            Poste P est "actif" et
            Poste P est "en cours de traitement" et
            Poste P est en train de traiter anode A et
            Convoyeur C en aval du poste P n'a pas
                  atteint sa capacité maximale et
            Le temps de traitement écoulé au poste P sur anode A est
                  supérieur au temps de traitement requis au poste P
      alors
            Rendre le poste P en "arrêt de traitement"
            Retirer anode A du poste P
            Ajouter anode A au convoyeur C
            Mettre à zéro le temps de traitement écoulé au poste P

// Incrémentation du temps de traitement si le poste P est en train de
// traiter une entité quelconque.
(R4)  si
            Poste P est "actif" et
            Poste P est "en cours de traitement"
```

```
alors
      Incrémenter le temps de traitement écoulé de P
```

À ce titre, les règles et les autres représentations de l'expertise font brièvement l'objet de la section 2.5.1. Au cours de celle-ci, nous allons d'avantage discuter du concept de base de connaissances. De plus, à la section 3.4.2.1, nous allons traiter de la représentation de la connaissance à partir de règles de production.

2.2.2 Traitement par inférence des connaissances

Le *moteur d'inférence* est le second concept fondamental d'un système expert. C'est à partir de la connaissance du domaine d'application, que le moteur d'inférence effectue un raisonnement ou infère des conclusions, au même titre qu'un spécialiste humain infèrerait la solution d'un problème. Inférer consiste donc à conclure qu'un ensemble de faits est vrai, en vertu d'un premier ensemble de faits et d'une implication logique, tous deux étant initialement reconnus comme vrais [Gonzalez et Dankel, 1993]. Cette assertion correspond en réalité à une règle logique, appelée *modus ponens*. Elle consiste à déclarer : lorsque les affirmations **p** et $(\mathbf{p} \Rightarrow \mathbf{q})$ sont vraies, alors **q** est aussi vrai [Hu, 1989] [McAllister et Stanat, 1977]. Voici un exemple d'inférence associé à l'enchaînement des étapes d'une anode cuite : si l'entrepôt des anodes est vide (**p**) et qu'une anode cuite est attendue à la phase de scellement (**q**), nous inférons qu'une nouvelle anode sortie du four devra être acheminée directement au centre de scellement (**r**). Donc, si d'une part $(\mathbf{p} \wedge \mathbf{q})$ est vrai et que d'autre part $(\mathbf{p} \wedge \mathbf{q}) \Rightarrow \mathbf{r}$ l'est aussi, alors **r** est vrai.

Il existe plusieurs méthodes afin de mettre en œuvre le mécanisme d'inférence. Celles-ci dépendent généralement de la représentation des connaissances. Dans le cas d'un système à base de règles, les deux méthodes d'inférences les plus communes sont le *chaînage avant* (« Forward Chaining ») et le *chaînage arrière* (« Backward Chaining ») [Giarratano et Riley, 1998] [Gonzalez et Dankel, 1993]. D'ailleurs, lors de la section 2.7 qui aborde le rôle du moteur d'inférence, nous allons définir plus clairement les différents mécanismes d'inférences possibles.

2.3 Coquille de développement de systèmes experts

Une *coquille de développement* est un outil facilitant la conception de systèmes experts (de l'Anglais « Expert System Shell »). Plus précisément, elle est formée d'un moteur d'inférence, de composantes fonctionnelles d'un système expert typique et d'une base de connaissances totalement dépourvue de contenu [Friedman-Hill, 2003]. C'est durant les années soixante-dix qu'on élabora la première coquille de développement : EMYCIN, pour « Empty » MYCIN [Buchanan et Shortliffe, 1984]. Ses concepteurs décidèrent de dépouiller MYCIN de sa connaissance spécifique, à propos des troubles sanguins, afin de pouvoir résoudre des problématiques issues de différents domaines d'applications. Du même coup, on obtenait un outil générique de conception de systèmes experts.

Au cours de ce mémoire, nous allons principalement discuter de deux coquilles de développement de systèmes experts, à savoir le « C Language Integrated Production System » [CLIPS] et le « Java Expert System Shell » [Jess]. Elles sont dotées de fonctionnalités couvrant presque tous les besoins de l'ingénieur de la connaissance et du concepteur de systèmes expert, notons : l'affirmation, la rétraction et la modification de faits ou encore, la définition de nouvelles règles. Il est même possible de les intégrer à des applications C/C++ (CLIPS) et Java (Jess), puisqu'elles disposent toutes deux d'une interface de programmation (API) [Giarratano et Riley, 1998] [Jess 7.0 Manual]. D'ailleurs à la section 3.4.2, nous allons discuter d'avantage des principales fonctionnalités qui justifient leur usage dans le cadre de la simulation de processus.

2.4 Architecture d'un système expert

Les éléments d'un système expert typique sont décrits au sein de la Figure 14 [Friedman-Hill, 2003] [Luger et Stubblefield, 1998]. Notons que dans ce cas-ci, puisque le concept qui représente le savoir s'avère être des règles, la figure illustre un *système à base de règles* (« Rule-Based System ») ou *système de production* (« Production System »).

Globalement, son but est d'apparier les règles de la base de connaissances, aux faits de la mémoire de travail, afin d'inférer des conclusions. Lorsque toutes les conditions d'une règle sont

satisfaites par un ensemble de faits, on dit qu'elle devient *active*. Par la suite, une priorité est assignée à chacune des règles activées, afin de spécifier leur séquence d'exécution. Cet ensemble de règles ordonnées s'appelle l'*agenda des activations*. Le fonctionnement que nous venons de mentionner est celui d'un moteur d'inférence à *chaînage avant*. Normalement, celui-ci doit effectuer un ou plusieurs *cycles d'inférences*. Un cycle consiste à réaliser chacune des étapes que nous venons de décrire. Les sections suivantes ont pour but d'éclaircir le rôle des différentes composantes qui interviennent lors d'un cycle d'inférence.

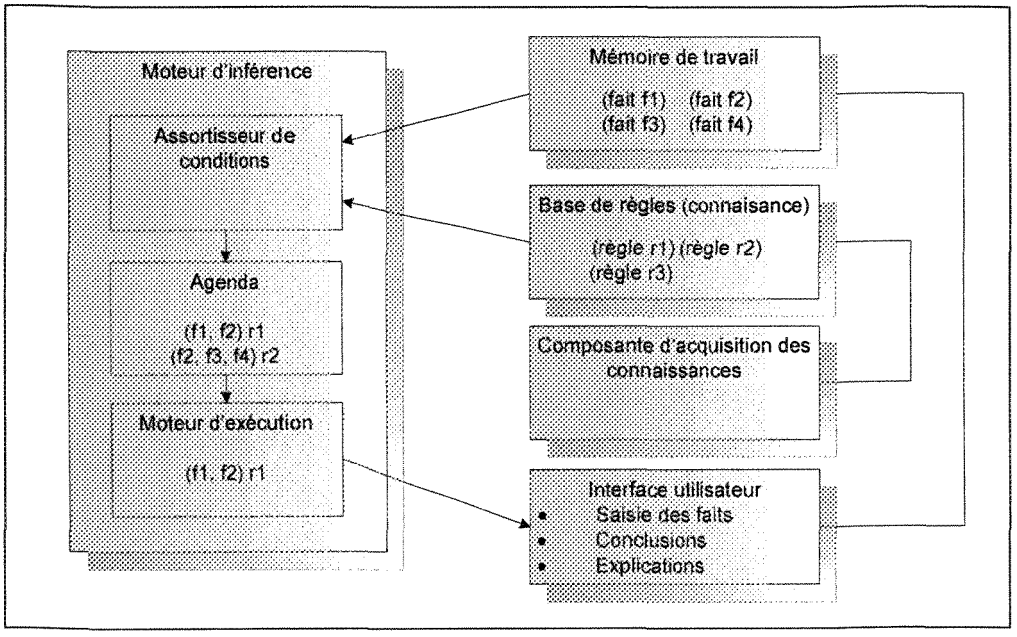

Figure 14 : architecture d'un système expert à base de règles

2.5 Base de connaissances

La *base de connaissances* (« Knowledge Base ») englobe tout le savoir inhérent au domaine d'application, afin de solutionner des problèmes spécifiques. Il existe trois catégories de connaissances [Krishnamoorthy et Rajeev, 1996] :

- Compilée : c'est-à-dire issue d'experts du domaine, de manuels spécialisés, d'études scientifiques, de spécifications reconnues, etc.

- Qualitative : basée sur des méthodes empiriques, des théories approximatives, des modèles causals, des heuristiques, le bon sens, etc.
- Quantitative : appuyée sur des théories mathématiques, des techniques numériques, etc.

Les deux premières catégories de connaissances sont aussi divisées en sous-groupes : *déclarative* et *procédurale* [Turban et Aronson, 2000]. La connaissance déclarative adresse les propriétés physiques du domaine, elle permet donc de le décrire. La connaissance procédurale réfère aux approches de résolution employées dans le cadre d'un problème. C'est le domaine d'application qui détermine la nature exacte et les interactions possibles de la connaissance. Par exemple, la connaissance qui permet de déterminer l'état du système de manutention des anodes, tel les postes arrêtés, les convoyeurs à pleine capacités, est dite déclarative. Quant à celle impliquant des prises de décisions, comme les actions qui permettent de combler un manque d'anode à la phase de scellement, elle est dite procédurale.

La *connaissance heuristique* est une catégorie spéciale, parfois conjointe aux trois précédentes. Elle est basée sur une ou plusieurs heuristiques, par exemple une règle empirique généralement reconnue [Gonzalez et Dankel, 1993]. Cette connaissance s'avère fondamentale, puisque généralement les problèmes soumis aux systèmes experts n'ont pas de solutions algorithmiques connues. De toute façon, l'usage d'un système expert devrait être reconsidéré, lorsque l'on sait qu'un problème a une solution exacte, telle un algorithme [Friedman-Hill, 2003].

2.5.1 Formalismes de représentations de la connaissance

Quelque que soit la catégorie de la connaissance impliquée dans la résolution d'une problématique, il prime avant tout de la représenter sous une forme compréhensible par un programme informatique. Il existe plusieurs formalismes de représentation. Parmi les plus courants notons [Gonzalez et Dankel, 1993] [Krishnamoorthy et Rajeev, 1996] : la *logique prédicative*, les *réseaux sémantiques*, les *cadres* (« Frames ») et les *règles de production*, aussi simplement appelées *règles*.

2.5.1.1 Logique prédicative

La *logique prédicative* ou *logique du premier ordre* est basée sur des faits, aussi nommés axiomes, c'est-à-dire des propositions qui sont soient vraies ou fausses [First-order logic]. Une proposition exprimant une relation est un prédicat. Il peut s'agir d'une relation entre plusieurs objets. Voici quelques exemples de prédicats :

```
<nom du prédicat>(<objet> +) :

        etat(four-R, actif)
        evenement(four-R, inactif, 245)
        temps-simule(245)
```

Une base de connaissances recourant à la logique prédicative contient des implications conditionnelles qui agissent à titre de méthodes d'inférences, citons la déduction, l'abduction et l'induction. Ces méthodes appliquées aux prédicats permettent de déduire des axiomes additionnels. Par exemple, à partir des prédicats ci-dessus et de la proposition suivante :

```
∀ P, ∀ T,
        [etat(P, actif) ∧ evenement(P, inactif, T) ∧ temps-simule(T)
        ⇒ etat(P, inactif)]
```

L'axiome qui suit est déduit :

```
etat(four-R, inactif)
```

Un des désavantages de la logique prédicative est qu'elle est entièrement basée sur des valeurs booléennes. Il n'y a donc aucun mécanisme pour gérer l'incertitude. Aussi, on constate qu'une fois qu'un axiome est déduit, il n'est plus possible de le modifier ou de le supprimer. La base de connaissances de tels systèmes a donc tendance à s'accroire énormément [Krishnamoorthy et Rajeev, 1996]. Certains systèmes experts qui s'appuient sur PROLOG [Clocksin et Mellish, 1987], une implémentation de la logique prédicative, résolvent ce problème en permettant la rétraction d'axiomes. Cependant, on ne peut plus véritablement parler de logique prédicative pure.

La logique prédicative, telle qu'implémentée par PROLOG, est implicitement déclarative. L'ingénieur de la connaissance n'a donc pas un droit de regard sur la méthodologie de recherche et d'appariements d'axiomes. En fait, on utilise un mécanisme d'inférence à *chaînage arrière*, dont il

est question à section 2.7.2. Or, dans certaines situations, celui-ci peut occasionner un temps d'exécution accru [Gonzalez et Dankel, 1993].

2.5.1.2 Les réseaux sémantiques

Les *réseaux sémantiques* furent introduits par [Quillian, 1968]. Un réseau sémantique est un graphe dirigé [Cormen et al., 1994] représentant la connaissance d'un domaine particulier. Chaque nœud du réseau sert à décrire les concepts ou les entités du domaine. Quant aux arêtes, elles illustrent les associations sémantiques qui interviennent entre les entités [Gonzalez et Dankel, 1993]. Les réseaux sémantiques servent d'avantage à exprimer la connaissance déclarative du domaine, afin de connaître ses propriétés physiques et sa taxonomie [Krishnamoorthy et Rajeev, 1996]. Il s'agit d'un inconvénient, puisqu'il est difficile de les utiliser dans un contexte où la connaissance procédurale est requise. Comme on l'a vu, ce type de connaissance est impliqué dans la résolution d'une problématique. Mentionnons que l'exploration d'un vaste réseau sémantique peut être assez longue, notamment lorsqu'il y a de nombreuses relations [Gonzalez et Dankel, 1993].

2.5.1.3 Les cadres et les daemons

Les *cadres*, de l'Anglais « frames » sont un autre formalisme de représentation de la connaissance. Ils furent introduits par [Minsky, 1974]. Ils regroupent et décrivent des faits, c'est-à-dire les entités d'un domaine, à partir d'attributs communs. Ces attributs descriptifs sont appelées les *fentes* (« slots »). On leur affecte une *valeur* ou un *remplisseur* (« filler »). Les fentes peuvent être subdivisées en *facettes* (« facets »). Ces dernières spécifient des informations comme : l'ensemble des valeurs possibles, le type d'une fente ou sa valeur par défaut [Durkin, 1994].

Également, les facettes référencent presque toujours des méthodes qui retournent ou affectent les valeurs des fentes. Ces méthodes sont appelées *daemons* [Gonzalez et Dankel, 1993]. Le rôle des daemons s'apparente à celui des règles d'un système de production, puisqu'il exprime à la fois le savoir déclaratif et procédural, en effectuant diverses manipulations à partir des

valeurs de fentes. Notons qu'on distingue deux types de cadres : les classes, décrivant les caractéristiques générales et les instances, c'est-à-dire un ensemble de cadres affiliés à une même classe. Ordinairement, les fentes de classes n'ont pas de valeur assignée, alors que celles des instances en ont. Parce que les instances héritent des attributs de leur classe ancêtre, il devient possible de construire une hiérarchie de cadres [Durkin, 1994].

Comme on peut le constater la notion de cadre [Krishnamoorthy et Rajeev, 1996], s'apparente à celle d'objet [Fowler, 2004] [Gilbert et McCarthy, 2001]. D'ailleurs, l'approche orientée objet est en grande partie basée sur les travaux effectués par des chercheurs en intelligence artificielle, durant les années soixante-dix [Champeaux et al, 1993].

Bien que la connaissance issue d'un système à base de cadres soit beaucoup plus structurée et organisée que celle des autres formalismes, son usage implique certains désavantages. Notamment, parce qu'il est difficile de décrire la connaissance heuristique à partir de cadres. En fait, les règles de production sont un mode de représentation beaucoup plus naturel, pour cette catégorie de connaissance [Gonzalez et Dankel, 1993].

2.5.1.4 Les objets et les règles de production

La connaissance basée sur les cadres implique que leurs fentes (attributs) et leurs daemons (méthodes) soient considérés comme deux concepts indépendants. Ainsi, une fente quelconque peut être manipulée par un daemon qui se trouve dans un autre cadre. L'interprétation d'un attribut spécifique peut donc différée d'un cadre ou d'un daemon à l'autre [Gonzalez et Dankel, 1993].

Par définition les *objets* ont une fonction beaucoup plus unificatrice que les cadres, puisqu'ils allient à la fois les *données* (attributs) et les *opérations* (méthodes), découlant d'un problème particulier [Müller, 1996]. En plus, le paradigme orienté objet comporte des caractéristiques intéressantes dans un contexte de représentation de connaissances. Il a été question de celles-ci à la section 1.7.1, à savoir : l'abstraction, l'encapsulation, l'héritage et le polymorphisme [Müller, 1996]. D'ailleurs, l'encapsulation permet d'éviter certaines incohérences associées aux cadres. En effet, les fentes d'un cadre peuvent être manipulées par n'importe quel daemon. Or, dans un

contexte de formalisation objet, l'encapsulation résout ce problème, car chaque attribut est référencé individuellement par des méthodes de leurs classes. Du même coup, tous les attributs ont une signification unique.

Toutefois, tout comme pour les cadres, le problème de la représentation de la connaissance heuristique demeure. C'est pourquoi la plupart des coquilles de systèmes experts ont une approche de représentation hybride, c'est-à-dire qu'ils recourent à la fois à des objets et des règles de production [Durkin, 1994]. Celles-ci construisent leurs conditions à partir des attributs de classes. Ces conditions sont appariées à des objets qui symbolisent les faits de la mémoire de travail. Les règles ont donc une structure similaire au pseudo-code suivant :

```
si
      poste.etat = "actif" et
      poste.nom = action.applique_a et
      action.type = "arrêt"
alors
      poste.etat = "désactiver"
```

La coquille de système expert CLIPS illustre bien cette approche hybride. Elle possède une extension appelée COOL (CLIPS Object-Oriented Language) [Giarratano, 2002] [Riley et al, 2005]. Elle permet d'implémenter des objets et de les référencer à partir de prémisses de règles. Quant à la coquille Jess [Jess 7.0 Manual], elle offre d'encore plus grandes possibilités d'interaction entre les règles et les objets. Nous en reparlerons d'ailleurs à la section 3.4.4.

Finalement, mentionnons qu'en ce qui concerne notre modèle, les règles sont un choix tout à fait naturel, pour représenter le fonctionnement du cycle de vie des anodes. Nous allons d'ailleurs revenir sur les forces de ce formalisme à la section 3.4.2.1. Puisque les règles de production font d'avantage l'objet de notre intérêt, le reste de la discussion traite principalement de ce formalisme.

2.5.2 Réseau d'inférence de règles

Un graphe s'avère particulièrement utile pour illustrer une base de connaissances formée de règles. En effet, une telle structure permet au spécialiste d'organiser la connaissance acquise, de rechercher les relations entre ses différentes sources, de vérifier sa cohérence et de la convertir

facilement sous une forme abstraite, reconnue par le système expert [Krishnamoorthy et Rajeev, 1996].

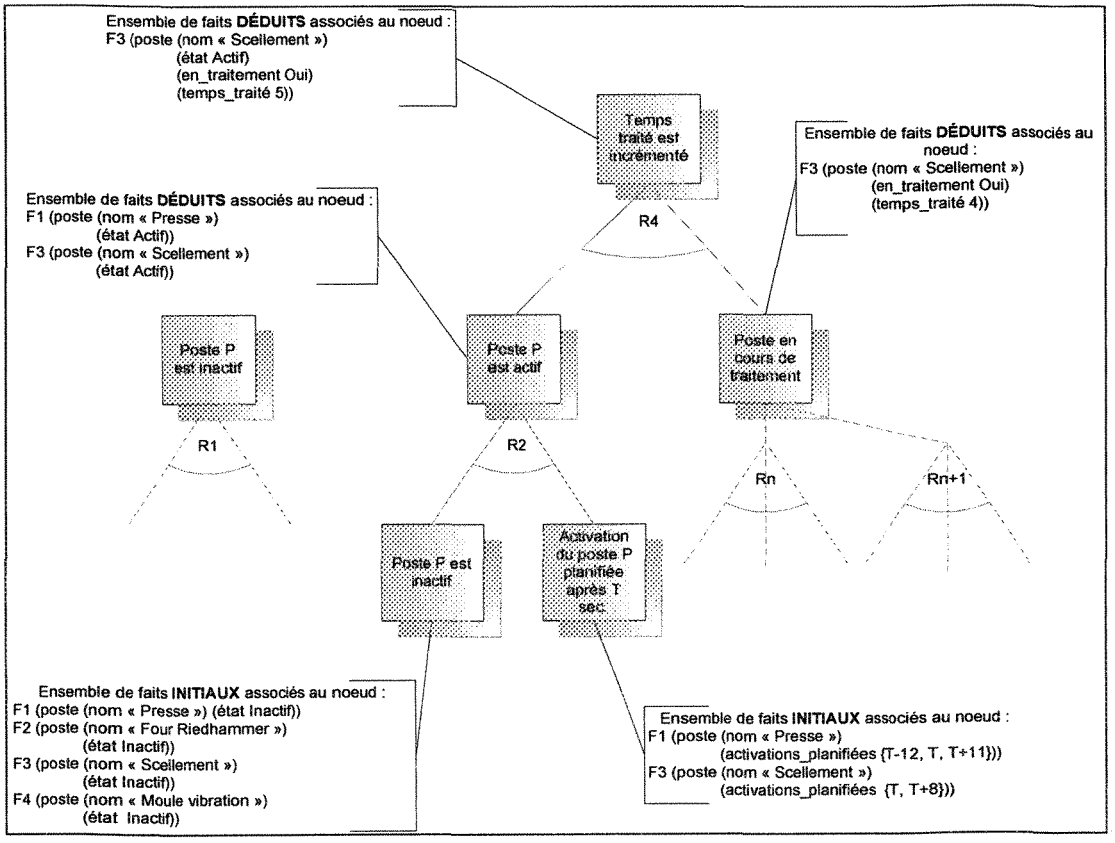

Figure 15 : réseau d'inférence d'une partie d'un problème

Puisque les conclusions tirées à partir des règles sont souvent des faits qui peuvent satisfaire les prémisses d'autres règles, il devient possible de visualiser ce graphe comme un réseau de règles et de faits interconnectés. On appelle cette structure un *réseau d'inférence* (de l'Anglais « Inference Network » ou « Inference Net ») [Gonzalez et Dankel, 1993]. Chaque nœud du graphe correspond à un paramètre associé à un ensemble de faits initiaux ou déduits durant le processus d'inférence. Ainsi, les nœuds peuvent d'une part, constituer la *partie gauche* ou *droite* d'une règle et d'autre part, permettre de référencer certains faits associés à des aspects ponctuels du problème. Quant aux jonctions entre les nœuds, elles servent à représenter les règles. La figure

ci-dessus nous montre le réseau d'inférence, illustrant une partie de l'exemple tiré de la section 2.2.1.

2.5.3 Acquisition et formalisation des règles

Bien que le réseau d'inférence soit une représentation tout à fait commode pour comprendre un problème et son domaine d'application, il ne correspond pas exactement à la représentation interne du système expert. Bien souvent une base de connaissances formée de règles est adjointe d'un *compilateur* [Friedman-Hill, 2003]. Celui-ci permet de restructurer la représentation des règles du système expert. Ainsi, le moteur d'inférence est en mesure de les traiter plus efficacement. C'est pourquoi la structure logicielle de la connaissance dépend en grande partie de l'algorithme du mécanisme d'inférence. L'algorithme de Rete [Forgy, 1982] constitue une des implémentations les plus courantes. Il nécessite que le compilateur construise une structure indexée, semblable à un graphe, appelée *réseau de Rete*. Chaque nœud entrant du réseau correspond à un motif d'une règle, à laquelle on peut associer des faits. Nous allons discuter de cet algorithme à la section 2.8.2.

Mentionnons qu'il existe différentes façons de formaliser des règles. L'une d'elle consiste à les programmer dans un langage spécifique. Grâce à ce programme source, le compilateur de la base de connaissances et un analyseur syntaxique sont en mesures de construire la structure en un réseau indexé, constituant la représentation interne des règles [Friedman-Hill, 2003]. Les coquilles de développement, dont il est question dans ce mémoire, à savoir CLIPS et Jess (voir la section 2.3), ont recours à la syntaxe LISP pour programmer les règles [Anderson et al, 1987]. L'acronyme LISP signifie « LISt Processing ». C'est un langage procédural qui, contrairement à PROLOG, est explicitement déclaratif. Il a été développé par John McCarthy durant les années 50. D'ailleurs, LISP et PROLOG ont été largement utilisés dans plusieurs domaines qui touchent l'intelligence artificielle [Luger et Stubblefield, 1998]. La structure de base du langage LISP est la liste, c'est-à-dire une séquence de symboles, séparés par des blancs et regroupés entre parenthèses :

```
(liste_de_symboles (voici une liste imbriquée de symboles en lisp))
(+ 2 4)
```

Dans le cas de CLIPS et de Jess, le premier symbole de la liste (la *tête*) a une signification particulière, lorsqu'il évoque un mot réservé reconnu par leur analyseur syntaxique. Ainsi, pour définir une règle quelconque, il suffit d'utiliser l'expression **defrule** et la syntaxe suivante :

```
(defrule <nom-de-la-regle> ["<Commentaire>"].
      <antecedent-premisse>*
=>
      <action-consequence>*
)
```

2.6 Mémoire de travail et les faits

La mémoire de travail (« Working Memory »), appelée aussi base de faits (« Facts Base »), contient un ensemble d'informations qui décrivent l'état ponctuel d'un système. D'une part, la mémoire de travail contient des faits assortis à des prémisses de règles et d'autre part, ces faits peuvent être affirmés, modifiés ou rétractés, par des actions imputables aux règles, c'est-à dire les conséquences. Afin de simplifier la recherche de faits, particulièrement lors des modifications et des rétractations, les systèmes experts modernes maintiennent des index [Friedman-Hill, 2003], similaires à ceux d'une base de données relationnelle [Elmasri et Navathe, 1994] [Gardarin, 2001].

2.6.1 Les valeurs des faits

Généralement la mémoire de travail permet de stocker des faits, dont les valeurs correspondent à des types élémentaires, tels des booléens, des chaînes de caractères, des nombre entiers ou réels. Certaines coquilles de systèmes experts offrent même la possibilité d'incorporer des objets complexes. C'est le cas de Jess, puisque sa mémoire de travail peut facilement référencer des instances de classes Java [Friedman-Hill, 2003]. Généralement, les outils de conception de systèmes experts, comme CLIPS et Jess, utilisent la syntaxe LISP pour programmer les faits de la mémoire de travail [Giarratano, 2002] [Jess 7.0 Manual]. Les valeurs de ces faits correspondent à l'ensemble des symboles qui suivent la *tête* d'une liste. Par exemple, la

tête du fait ci-dessous est `poste`. Les trois valeurs de ce fait sont respectivement de type : chaîne de caractères (`"Presse"`), symbole (`actif`) et nombre entier (`254`).

```
(poste "Presse" actif 254)
```

2.6.2 Les types de faits

Les coquilles de développement de systèmes experts modernes, comme Jess et CLIPS, mettent en œuvre deux types de faits : *non-ordonnés* et *ordonnés*. Les premiers sont formés de *fentes* (« slots »), c'est-à-dire les attributs qui décrivent le fait, par exemple : le nom, l'état, le temps traité, le temps de traitement, le temps d'opération, etc. [Giarratano, 2002] [Jess 7.0 Manual]. Ce type de fait ressemble à un enregistrement d'une table de base de données relationnelle [Elmasri et Navathe, 1994], où les fentes correspondent aux champs de la table. Afin de spécifier explicitement les attributs des faits non-ordonnés, CLIPS et Jess possèdent une instruction de définition de *modèles de faits* (« template »), à savoir le mot réservé **`deftemplate`** :

```
(deftemplate poste
      (slot nom (type STRING))
      (slot etat (type STRING)
      (slot temps_traite (default 0) (type INTEGER)))
```

Ainsi, on peut définir des faits non-ordonnés qui agissent comme des instances du modèle, par exemple :

```
(poste (nom "Presse") (etat actif) (temps_traite 254))
(poste (nom "Four R") (etat inactif) (temps_traite 254))
```

Quant aux faits ordonnés, ils sont utilisés lorsque l'information est dépourvue de toute forme de structure. En un mot, ils correspondent à une simple énumération de valeurs :

```
(poste "Presse" actif 254)
(actif inactif "arret planifie" "arret non-planifie")
```

Les coquilles CLIPS ou Jess utilisent le mot réservé **`assert`** [Giarratano, 2002] [Jess 7.0 Manual], pour signaler à leur analyseur syntaxique qu'un nouveau fait est affirmé, c'est-à-dire ajouté à la mémoire de travail. Éventuellement, les valeurs ou *fentes* des faits peuvent être assortis aux variables des conditions de règles (**`defrule`**) et contribuer à l'activation de ces dernières.

L'exemple ci-dessous montre l'affirmation de trois faits non-ordonnés `poste`. Le premier et le dernier satisfont l'unique condition de la règle `poste-actif`. Notons que la variable `?nom_poste` est liée à la *fente* `nom`. Cette *fente* est associée à des faits dont la tête est de type `poste`.

```
(assert (poste (nom "Presse") (etat actif) (temps_traite 254))
(assert (poste (nom "Moule") (etat inactif) (temps_traite 50))
(assert (poste (nom "Tour à pâte") (etat actif) (temps_traite 306))

(defrule poste-actif "Affiche à l'écran les postes actifs"
     (poste (nom ?nom_poste) (etat actif))
=>
     (printout t "Le poste " ?nom_poste " est actif." crlf)
)
```

On a comme sortie l'affichage suivant :

```
Le poste Presse est actif.
Le poste Tour à pâte est actif.
```

2.7 Moteur d'inférence

Le *moteur d'inférence* est la composante chargée d'interpréter le savoir de la base de connaissances. Il est divisé en trois parties : l'*assortisseur de condition*, l'*agenda des activations* et le *moteur d'exécution*. Il examine le contenu de la base de connaissances et les faits accumulés à propos d'un problème, afin de tirer des conclusions, établir des solutions ou obtenir des réponses. Toutefois, les mécanismes qui contribuent à atteindre ces résultats peuvent varier de façon significative. Notamment en raison des différents formalismes de la connaissance et des types de raisonnements pouvant être appliqués à celle-ci. Dans le cas d'un système expert formalisé à partir de règles, le fonctionnement du moteur d'inférence est basé sur deux catégories de raisonnements : le *chaînage avant* et le *chaînage arrière* [Gonzalez et Dankel, 1993].

2.7.1 Moteur à chaînage avant

Le *chaînage avant* est une approche de résolution de problème dite *de bas en haut*. Son appellation anglaise est « bottom-up ». En effet, elle consiste à d'abord s'appuyer sur des preuves de bas niveau, à savoir les faits, pour pouvoir établir des conclusions de niveau supérieur [Giarratano et Riley, 1998]. On dit aussi que c'est une approche *conduite par les données* (« data

driven »), car ce sont les faits qui mènent au résultat final. C'est pourquoi le *chaînage avant* convient d'avantage, lorsque la plupart des faits en présences sont nécessaires à la résolution de la problématique. On l'utilise aussi dans des situations où il existe plusieurs conclusions acceptables et dans le cadre de problèmes impliquant la synthèse de données, par exemple : le contrôle, la planification, la simulation, la surveillance, etc.

En pratique, ce type de raisonnement implique qu'il faut vérifier chaque règle de la base de connaissances, afin de déterminer si les faits observés satisfont pleinement leurs prémisses. Les règles ainsi satisfaites sont dites *actives* et doivent être exécutées par le moteur d'inférence, afin de déduire de nouveaux faits. À leur tour, ceux-ci pourront déclencher d'autres règles qui déduiront des faits additionnels et ainsi de suite. Normalement, le processus itératif que nous venons de décrire, se poursuit jusqu'à ce qu'il n'y ait plus aucune règle active. Ainsi, un moteur d'inférence à *chaînage avant* doit itérer plusieurs cycles. Ceux-ci impliquent les étapes qui suivent [Friedman-Hill, 2003] :

1. Assortiment :
   1.1. Comparer toutes les règles avec les faits de la *mémoire de travail*, grâce à l'*assortisseur de conditions*.
   1.2. Activer les règles dont toutes les conditions sont satisfaites par un ensemble de faits.
   1.3. Construire une liste non ordonnée, i.e. l'*ensemble des conflits*, à partir des règles activées durant ce cycle et les cycles antérieurs.
2. Résolution des conflits :
   2.1. Ordonner les règles conflictuelles à l'aide de la stratégie mise en œuvre par le *solutionneur de conflits*.
   2.2. Construire l'*agenda des activations*, i.e. l'ensemble des règles actives, ordonnées par la stratégie de résolution de conflits.
3. Exécution :
   3.1. Sélectionner la première règle active de l'*agenda*.
   3.2. Déclencher la règle sélectionnée.
   3.3. Apporter les changements à la *mémoire de travail* si l'exécution provoque des modifications, e.g. s'il y a des affirmations ou des rétractions de faits.
   3.4. Effectuer un autre cycle en allant à 1.1.

À première vue, de telles répétitions semblent impliquer des actions redondantes. On a qu'à penser à la première étape (assortiment) qui consiste à comparer les faits et les règles à chaque itération. Cependant, plusieurs systèmes experts utilisent des algorithmes pour éliminer cette

redondance. Notamment, en préservant les résultats issus de l'assortisseur de conditions (les étapes 1.1, 1.2 qui sont détaillées à partir de 2.8) et du solutionneur de conflits de l'agenda (les étapes 2.1, 2.2, dont il est question à la section 2.9).

Finalement, mentionnons que le mécanisme que nous venons de décrire, s'apparente à un parcours en largeur d'un graphe (de l'Anglais « Breadth-First Search ») [Shaffer, 1997] [Cormen et al., 1994], lorsqu'on l'applique au réseau d'inférence, illustrant la connaissance d'un processus particulier. C'est pourquoi le raisonnement avec *chaînage avant* convient d'avantage, pour des problèmes impliquant un réseau d'inférence assez large et peu profond [Giarratano et Riley, 1998].

2.7.2 Moteur à chaînage arrière

À l'inverse, le *chaînage arrière* est une approche de résolution *du haut vers le bas*. Elle implique un raisonnement à partir de structures de hauts niveaux, c'est-à-dire des hypothèses, vers des faits de bas niveaux qui supportent ces dernières. On dit aussi que le *chaînage arrière* est axé vers la *recherche d'objectifs* (« goal seeking ») [Giarratano et Riley, 1998]. C'est pourquoi, on la privilégie lorsqu'une quantité limitée d'hypothèse (ou d'objectifs) est initialement déterminée. Aussi, cette méthode d'inférence est fort appropriée si la problématique comporte un grand nombre de faits et si seulement une partie d'entre eux sont pertinents dans le cadre de la résolution. D'ailleurs, il arrive qu'un système expert avec *chaînage arrière* demande à l'usager de spécifier des faits absents de la mémoire de travail, afin de supporter les hypothèses qu'il a émises.

Pour toutes les raisons que nous venons d'énumérer, cette approche semble idéale pour les problèmes de diagnostic [Gonzalez et Dankel, 1993]. Par exemple, un spécialiste de la santé confronté à un malade doit se limiter à des symptômes spécifiques (les faits significatifs), plutôt qu'à l'état global du patient (tous les faits en présences), afin d'établir et valider un nombre minimal de causes (les hypothèses). D'autre part, on note qu'une méthode d'inférence avec *chaînage arrière* est similaire à un parcours en profondeur d'un graphe (de l'Anglais « Depth-First Search ») [Shaffer, 1997] [Cormen et al., 1994]. C'est pourquoi on l'utilise d'avantage lorsque le réseau d'inférence de la problématique est étroit et profond [Giarratano et Riley, 1998].

Tout comme un système expert basé sur un mécanisme d'inférence à *chaînage avant*, la base de connaissances des systèmes à *chaînage arrière* est formée de règles de type *si ... alors*. Cependant, leur mode de raisonnement est beaucoup plus complexe que la simple exécution des règles activées. En effet, dans un premier temps, il faut déterminer un ensemble d'objectifs (ou hypothèses) à atteindre. Puis, le moteur d'inférence va activement chercher à satisfaire les prémisses de certaines règles qui peuvent supporter ces objectifs, en considérant les faits en présences. Éventuellement, les règles ne pouvant être satisfaites permettront de déterminer des sous objectifs à atteindre [Friedman-Hill, 2003]. Les étapes suivantes décrivent de manière plus détaillée, le fonctionnement d'un moteur d'inférence à *chaînage arrière* [Gonzalez et Dankel, 1993]:

1. Construire une pile, (i.e. *l'ensemble des hypothèses)* composée des objectifs à atteindre. Ceux-ci sont supportés par des faits de la mémoire de travail.
2. Sélectionner le premier objectif de la pile, trouvez toutes les règles qui l'appuient (i.e. celles dont l'exécution permet d'affirmer des faits supportant l'objectif).
3. Pour chacune des règles qui valident l'objectif, examiner leurs prémisses :
   - 3.1. Si la prémisse d'une règle est satisfaite (i.e. toutes ses conditions sont assorties à des faits de la mémoire de travail), alors :
     - 3.1.1. Exécuter la règle satisfaite.
     - 3.1.2. Apporter les changements à la *mémoire de travail* si l'exécution provoque des modifications.
     - 3.1.3. Enlever le premier objectif de la pile et retourner à l'étape 2.
   - 3.2. Si la prémisse d'une règle n'est pas satisfaite (i.e. une ou plusieurs de ses conditions ne sont pas assorties à des faits de la mémoire de travail), alors :
     - 3.2.1. Rechercher une règle, dont l'exécution permet d'affirmer un fait capable de satisfaire la condition manquante.
     - 3.2.2. Si une telle règle existe, considérez que la condition assortie au fait manquant est un sous objectif à atteindre. Ajoutez-le à la pile et retournez à l'étape 2.
   - 3.3. Si l'étape 3.2 ne réussit pas à déterminer une règle, dont l'exécution affirme un fait satisfaisant la condition manquante, alors demandez à l'usager si ce fait est vrai.
     - 3.3.1. Si sa réponse est positive, affirmez le fait, considérez que la condition manquante est maintenant satisfaite et passez à la condition suivante de cette règle (i.e. retournez à l'étape 3.2.1).
     - 3.3.2. Si sa réponse est négative, considérez une autre règle (i.e. retournez à l'étape 3.1).

```
3.4.  Si toutes les règles qui peuvent appuyer l'objectif actuel ont
      été essayées sans succès, celui-ci demeure indéterminé. Enlevez
      l'objectif de la pile.
   3.4.1.  Passez à l'objectif suivant s'il y en a un (i.e. retournez
           à 2).
   3.4.2.  S'il n'y a plus d'objectif dans la file, le moteur
           d'inférence a complété son travail.
```

2.8 Assortiment de conditions

Les *moteurs d'inférences* des systèmes experts doivent mettre en œuvre un algorithme particulier, afin d'assortir les faits de la mémoire de travail, avec les conditions contenues dans chacune des règles. Cette tâche incombe à l'*assortisseur de condition* ou *assortisseur de motif* (traduction de l'Anglais « Pattern Matcher »). Par la suite, les règles dont toutes les conditions sont assorties vont être ajoutées à l'agenda des activations et éventuellement déclenchées. Cependant, l'implémentation de l'algorithme d'assortiment des motifs dépend du type de raisonnement imputable au moteur d'inférence. En ce qui concerne le modèle que nous avons conçu, il utilise un mode de raisonnement à *chaînage avant*, pour les raisons qui sont mentionnées à la section 3.4.2.2. La discussion qui suit est donc axée sur l'une de ses implémentations les plus connues, à savoir l'algorithme de Rete développé par le docteur Charles L. Forgy de l'université Carnegie Mellon [Forgy, 1982]. L'appellation Rete signifie *filet* en Latin ou « net » en Anglais, au sens d'un réseau (c'est-à-dire « Network »), puisqu'il consiste à construire un graphe, similaire au réseau d'inférence dont il a été question à la section 2.5.2.

2.8.1 Assortiment basé sur un algorithme naïf

Afin de mieux comprendre en quoi Rete est une solution appropriée au problème d'assortiment de conditions, considérons d'abord l'implémentation naïve, du raisonnement à *chaînage avant* que nous avons décrite à la section 2.7.1. Cette solution triviale consiste à conserver une liste de règles et vérifier que toutes leurs conditions sont assorties à des faits de la mémoire de travail. On peut ainsi former un ensemble d'activations pour toutes les règles assorties. Après avoir sélectionné et exécuté une règle de l'ensemble, on la retire de ce dernier et on passe

au cycle d'inférence suivant. Ceci implique une nouvelle tentative d'assortiment de conditions. Cette approche, où les règles « recherchent » des faits, est clairement inefficace. En effet, à chaque cycle, il faut revérifier les activations, car le déclenchement d'une règle peut avoir modifié la mémoire de travail [Friedman-Hill, 2003 et 2005] [Giarratano et Riley, 1998].

Maintenant considérons la règle suivante (`R1'`), une simplification de `R1` définie à la section 2.2.1, afin d'analyser quelle pourrait être la complexité de cet algorithme.

```
(R1') si
        Poste P est "actif" et
        Action planifiée de type "arrêt" survient actuellement sur P
     alors
        Rendre poste P "inactif"
```

Supposons qu'il y ait **p** faits décrivant les *postes* et **a** faits associés aux *actions planifiées*. En appliquant l'algorithme naïf à cette règle, dans un premier temps, on doit rechercher tous les *postes actifs* (où `P` représente l'un d'eux) et toutes les *actions planifiées* de type *arrêt* actuellement en vigueur. Dans un deuxième temps, à cause de la jointure découlant de la variable `P` des deux conditions, on estime les *arrêts* qui s'appliquent aux *postes actifs*. Nous pouvons donc affirmer que dans ce cas, l'algorithme naïf devrait avoir une performance $\mathbf{O((a+p)^2)}$. La complexité de la solution semble donc être dépendante d'une part, du nombre de conditions formant la prémisse (**2** dans `R1'`) et d'autre part, du nombre de faits qui leur sont associées (**a** et **p**). Néanmoins, il est plutôt difficile d'estimer la complexité exacte de cette implémentation, lorsqu'on applique à toutes les règles de la base de connaissances. Notamment parce que chacune d'elles n'ont pas le même nombre de motifs (conditions) et que ceux-ci ne sont pas nécessairement liés par des variables (`P` dans `R1'`) [Jess 7.0 Manual]. Par contre, nous pouvons affirmer qu'en moyenne, la performance de la solution naïve est de l'ordre de $\mathbf{O(R \cdot F^C)}$ où,

1. **R** = nombre de règles de la base de connaissances
2. **F** = nombre de faits de la mémoire de travail
3. **C** = nombre moyen de conditions par règle.

Le nombre moyen de conditions par règle cause donc un accroissement exponentiel de la complexité, au fur et à mesure que des faits sont ajoutés. Cet algorithme est d'autant plus inefficient, puisque que l'on doit l'appliquer à chaque cycle d'inférence.

2.8.2 Assortiment basé sur l'algorithme de Rete

L'amélioration du mécanisme d'assortiment de motifs est basée sur une constatation, à savoir que la plupart des systèmes experts ont une mémoire de travail avec un contenu relativement fixe. Ainsi, bien que des faits soient affirmés, modifiés ou rétractés, leur pourcentage demeure relativement négligeable [Friedman-Hill, 2003] [Giarratano et Riley, 1998]. L'implémentation que nous avons décrite, où les règles « recherchent » les faits, est donc inefficace, puisque l'ensemble des assortiments de motifs demeure sensiblement le même pour chaque cycle d'inférence.

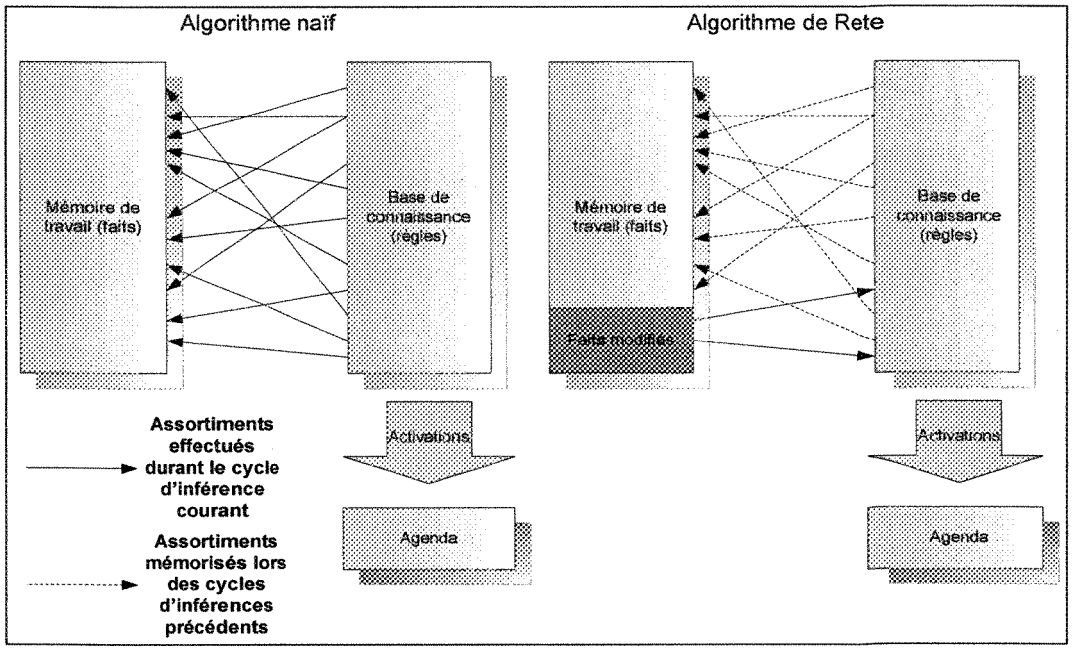

Figure 16 : l'algorithme naïf et de Rete pour l'assortiment des conditions

L'algorithme de Rete apparaît comme une amélioration, car il mémorise les assortiments des cycles précédents et les met à jour, uniquement lorsque des faits ont été changés. Cette solution semble donc moins coûteuse, car ce sont uniquement les faits modifiés qui « recherchent » les

règles. La figure ci-dessus illustre les deux algorithmes d'assortiment de motifs, dont il a été question jusqu'ici. Notons les faits modifiés dans la zone foncée et la faible proportion des assortiments devant être à nouveau vérifiées, c'est-à-dire les flèches pleines, lorsqu'on considère Rete.

La mémorisation des assortiments implique que chacune des règles connaissent les faits qui satisfont leurs conditions, en considérant les restrictions découlant des variables qui les lient. Donc, si un fait modifié était assorti à la troisième condition d'une règle, l'information provenant de l'assortiment des deux conditions précédentes doit être disponible, afin de compléter le processus d'appariement. Cette information indiquant l'ensemble des faits qui satisfont un groupe de conditions successives, en considérant les variables qui les lient, s'appelle une *association partielle* (de l'Anglais « Partial Match ») [Giarratano et Riley, 1998]. Par exemple, soit la règle `R1'` :

```
(R1') si
        Poste P est "actif" et
        Action planifiée de type "arrêt" survient actuellement sur P
     alors
        Rendre poste P "inactif"
```

En ayant recours à la syntaxe LISP, nous avons la règle suivante :

```
(R1') (defrule desactiver-poste

        ; Première condition de la règle pour poste.
        ; (n.b. ?fait_poste contient un fait associé à la condition.)
        ?fait_poste <-
              (poste (nom ?nom_poste) (etat "actif"))

        ; Deuxième condition de la règle pour action planifiée.
              (action_planifiee (type "arrêt")
                        (poste ?nom_poste) (debute_a ?t_actuel))
     =>
        ; Modification de le l'attribut état du fait poste.
        (modify ?fait_poste (etat "inactif"))
```

Supposons que la mémoire de travail contienne les faits suivants et que le temps de simulation actuel (variable `?t_actuel`) soit de 245 secondes :

```
F1 : (poste (nom "Presse") (etat "actif"))
F2 : (poste (nom "Tour à pâte") (etat "inactif"))
F3 : (poste (nom "Moule") (etat "actif"))
F4 : (poste (nom "Four R") (etat "actif"))
```

```
F5 : (action_planifiee
        (type "démarrage") (poste "Tour à pâte") (debute_a
     250))
F6 : (action_planifiee
        (type "arrêt") (poste "Presse") (debute_a 245))
F7 : (action_planifiee
        (type "arrêt") (poste "Moule") (debute_a 245))
F8 : (action_planifiee
        (type "arrêt") (poste "Scellement") (debute_a 245))
```

Dans ce cas-ci, les *associations partielles* correspondent aux ensembles de faits suivants :

1ère condition : {F1} et {F3}
1ère et 2ième condition : {F1, F6} et {F3, F7}

Notons qu'une *association partielle* de toutes les conditions d'une règle forme une *activation*, comme c'est le cas pour les ensembles suivants : {F1, F6} et {F3, F7}. Également, il existe un autre type d'information, appelée *association de condition* (en Anglais « Pattern Match »), qui doit être prise en compte par l'algorithme de Rete. Elle survient lorsqu'une condition de règle est satisfaite par un fait, peu importe les variables des autres motifs limitant le processus d'assortiment [Giarratano et Riley, 1998]. En considérant l'exemple précédent, les *associations de conditions* de `R1'` sont :

1ère condition : F1, F3 et F4
2ième condition : F6, F7 et F8

Mentionnons que ces faits se bornent à effectuer une association avec la 1ère et 2ième condition, sans se soucier de la variable `P` (`?nom_poste`) qui lie les deux motifs.

2.8.2.1 Mise en œuvre de l'algorithme de Rete

Comme on l'a mentionné, l'algorithme de Rete est mis en œuvre à l'aide d'un réseau de nœuds interconnectés, voir la Figure 17 plus bas. Chacun d'eux représentent généralement un ou plusieurs tests à effectuer. Ceux-ci résultent des diverses conditions de la partie gauche des règles. Chaque nœud du réseau a soit une ou deux *entrées* et une seule *sortie*. Les faits de la mémoire de travail qu'on affirme, modifie ou rétracte sont donc testés par certains nœuds du réseau. Lorsque le test d'un nœud est fructueux, les faits résultants sont envoyés au suivant. Une activation est

effectuée lorsqu'un ensemble de faits a entièrement traversé le réseau. Le nœud se trouvant à la toute fin du réseau s'appelle le *nœud terminal* [Friedman-Hill, 2003] [Jess 7.0 Manual].

La mise en œuvre de l'algorithme de Rete, s'effectue en deux étapes [Giarratano et Riley, 1998]. Dans un premier temps, lorsque des changements au sein de la mémoire de travail ont eu lieu, la vérification des *associations de conditions* mémorisées doit être faite, pour estimer celles qui sont toujours valides. Dans un deuxième temps, la comparaison des variables liant les conditions doit être effectuée, afin de déterminer les groupes de motifs dont les *associations partielles* demeurent effectives.

L'algorithme de Rete réalise la première étape grâce à un arbre appelé le *réseau de condition* (traduction de l'Anglais « Pattern Network »). À son sommet se trouve un ou plusieurs nœuds à *entrée unique*, appelés *nœuds de condition* (de l'Anglais « Pattern Node »). Ceux-ci permettent de filtrer les faits selon leur *tête* et d'envoyer au nœud suivant les faits résultants [Friedman-Hill, 2003]. Par exemple, dans le cas de `R1'`, les faits *poste* et *actions planifiées* empruntent des chemins totalement différents à travers le *réseau de condition*. En plus, les *nœuds de condition* de cet arbre permettent de représenter les contraintes individuelles d'attributs (« slots »), en autant qu'ils ne soient pas liés à d'autres motifs [Giarratano et Riley, 1998]. Cette représentation d'un motif non lié, s'explique du fait que le réseau mémorise les *associations de conditions* qui, comme on l'a vu, ne tiennent pas compte des liens entre les motifs. Ainsi, en considérant `R1'`, parce que l'attribut *état* de la première condition impose que sa valeur soit « *actif* », un nœud devrait être ajouté au *réseau de condition*. Ce nœud permet d'effectuer le test qui découle uniquement du premier motif.

Quant à la seconde étape, elle est accomplie à l'aide du *réseau de branchement* (de l'Anglais « Join Network »). Celui-ci est adjoint au *réseau de condition*. Donc, une fois que les conditions ont été appariées aux faits, la comparaison des variables qui les joignent doit être effectuée, pour s'assurer que celles utilisées dans d'autres motifs possèdent des valeurs cohérentes [Giarratano et Riley, 1998]. Le *réseau de branchement* est formé de nœuds à double

entrées appelés *nœuds de branchement* (traduction de l'Anglais « Join Node »). Généralement, ils testent une jointure entre deux conditions. De plus, ils sont utilisés pour mémoriser l'ensemble des faits provenant des deux *entrées*. Les nœuds utilisent deux unités de stockages pour préserver les faits entrants, à savoir la *mémoire alpha* pour l'arête gauche et la *bêta* pour l'arête droite.

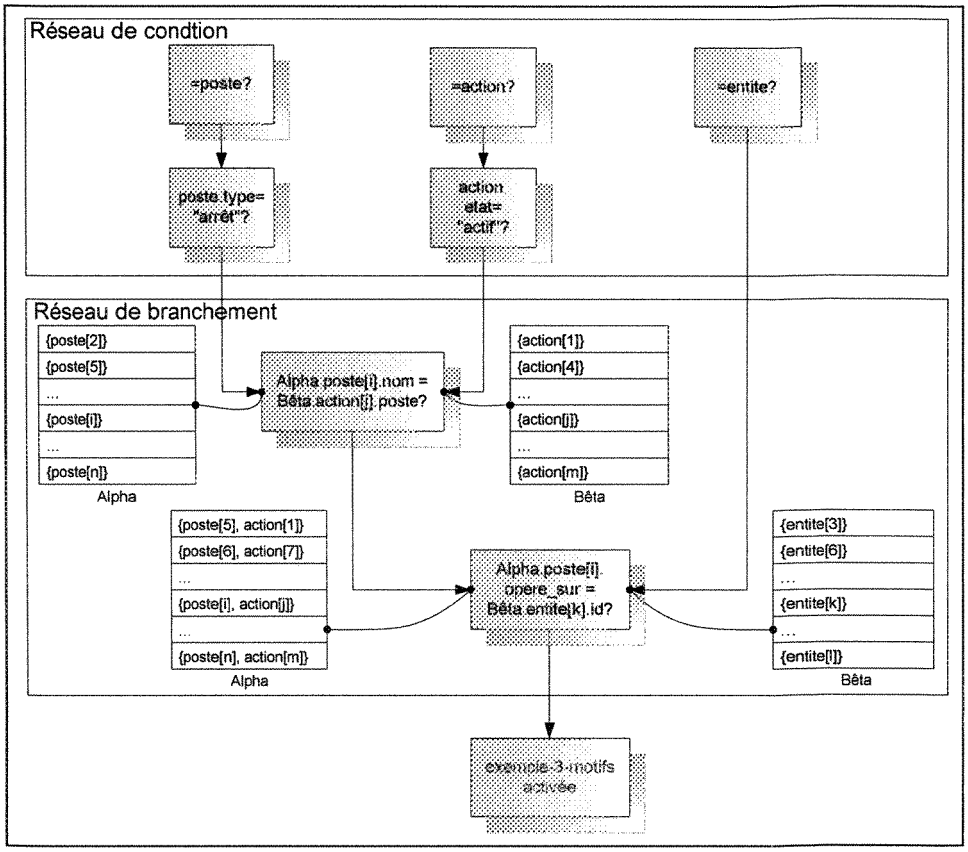

Figure 17 : réseau de Rete d'une règle

Généralement, on utilise une structure de donnée indexée, telle une table de hachage [Cormen et al, 1994], pour représenter les deux mémoires et ainsi faciliter la recherche des faits et des règles appariées [Friedman-Hill, 2003]. Le *réseau de branchement* est construit de manière à ce que *l'entrée* gauche d'un nœud reçoive des faits ayant des *têtes* identiques ou différentes. Quant à l'entrée de droite, elle obtient des faits qui ont tous la même *tête*. Les deux *entrées* ont des fonctions distinctes, puisque d'une part, les faits provenant de l'arête droite découlent d'un motif unique *m* (supposons le $n^{ième}$ d'une règle quelconque) et d'autre part, ceux issus de l'arête gauche

résultent des assortiments des *n-1* motifs qui précèdent *m*, autrement dit un intervalle de motifs *[1..n-1]* [Friedman-Hill, 2003]. La Figure 17 illustre le réseau de Rete de la règle suivante :

```
(defrule exemple-3-motifs
      (poste (nom ?nom_poste) (etat "actif") (opere_sur ?id_entite))
      (action (type "arrêt") (poste ?nom_poste))
      (entite (id ?id_entite))
=>
      )
```

Lorsqu'on analyse son *réseau de branchement*, on constate que *l'entrée* de droite du premier nœud dérive des assortiments du second motif (*n = 2*), tandis que celle de gauche découle du motif précédent (où l'on a *[1..n-1]* ≡ 1er motif). La même logique est applicable au niveau du second nœud, puisque son arête de droite reçoit les faits associés au troisième motif (*n = 3*), alors que celle de gauche résulte des associations des deux motifs précédents (où l'on a *[1..n-1]* ≡ 1er et 2^{ième} motif). C'est d'ailleurs pourquoi chaque cellule de la mémoire *alpha* de ce nœud est formée d'un couple de faits `poste` (1er motif) et `action` (2^{ième} motif), contrairement à *bêta* qui contient seulement des faits dont la *tête* est `entite`.

2.8.2.2 Rétraction et modification selon l'algorithme de Rete

Jusqu'ici nous avons vu comment l'algorithme de Rete met en œuvre l'assortiment de faits lors d'affirmations. Cependant, il est notable de mentionner que la rétraction utilise un principe similaire. En réalité, Rete ne propage pas des faits à travers le réseau mais plutôt des jetons. Un jeton est composé d'un ou plusieurs faits et d'une commande spécifiant l'action à exécuter : affirmation ou rétraction. Ainsi, lorsqu'un jeton avec la commande de rétraction parvient à un *nœud de branchement*, Rete recherche un fait identique au sein de la mémoire appropriée. Si la recherche est fructueuse, le fait est détruit. Puis, pour tous les faits associés, se trouvant dans la mémoire opposée, on crée des jetons avec la commande de suppression. Ceux-ci sont à leur tour envoyés à la *sortie* du *nœud de branchement*. Finalement, lorsqu'un *nœud terminal* du réseau de Rete reçoit un jeton avec la commande de suppression, l'activation correspondante est retirée de l'agenda [Friedman-Hill, 2003].

Concernant la commande de modification, bien qu'elle ne soit pas précisée dans la description originale de Rete [Forgy, 1982], la plupart des systèmes experts la mettent en œuvre en rétractant le fait, puis en changeant les valeurs d'attributs et en réaffirmant le fait modifié [Giarratano, 2002] [Jess 7.0 Manual].

2.8.2.3 Optimisation de l'algorithme de Rete

Il existe différentes méthodes pour optimiser l'algorithme de Rete. Deux des plus simples consistent à effectuer un *partage des nœuds* [Friedman-Hill, 2003] [Jess 7.0 Manual]. Le premier type de partage implique la mise en commun des nœuds du *réseau de condition*. Donc, en considérant la règle suivante :

```
(defrule exemple-2-motifs
      (poste (nom ?nom_poste) (etat "actif"))
      (action (type "arrêt") (poste ?nom_poste))
=>
      )
```

Ainsi que `exemple-3-motifs`, décrite à la section 2.8.2.1, on constate que quatre nœuds du *réseau de condition* peuvent être mis en commun : `=poste?`, `=action?` (pour la vérification des *têtes*), ainsi que `poste.etat="actif"?` et `action.type="arrêt"` (pour la vérification des valeurs d'attributs).

Cependant, en ce qui concerne cet exemple, on peut éviter encore plus de redondance, en mettant aussi en commun le *nœud de branchement* qui effectue le test suivant : `Alpha.poste[i].nom = Bêta.action[j].poste?`, pour les deux règles. La figure ci-dessous montre le réseau de Rete partageant à la fois les *nœuds de condition* et de *branchement*.

Puisque l'opération de jointure implique la comparaison de plusieurs conditions, les tests du *réseau de branchement* risquent d'être en plus grand nombre que ceux du *réseau de condition*. C'est pourquoi la jointure des variables de motifs compte pour une grande part du temps d'exécution de l'algorithme. Il s'avère donc fort avantageux de partager les *nœuds de branchement* autant que possible. Ainsi, en mettant en commun le premier *nœud de branchement* dans la figure

suivante, on doit s'attendre à ce que la performance de l'algorithme soit doublée [Friedman-Hill, 2003].

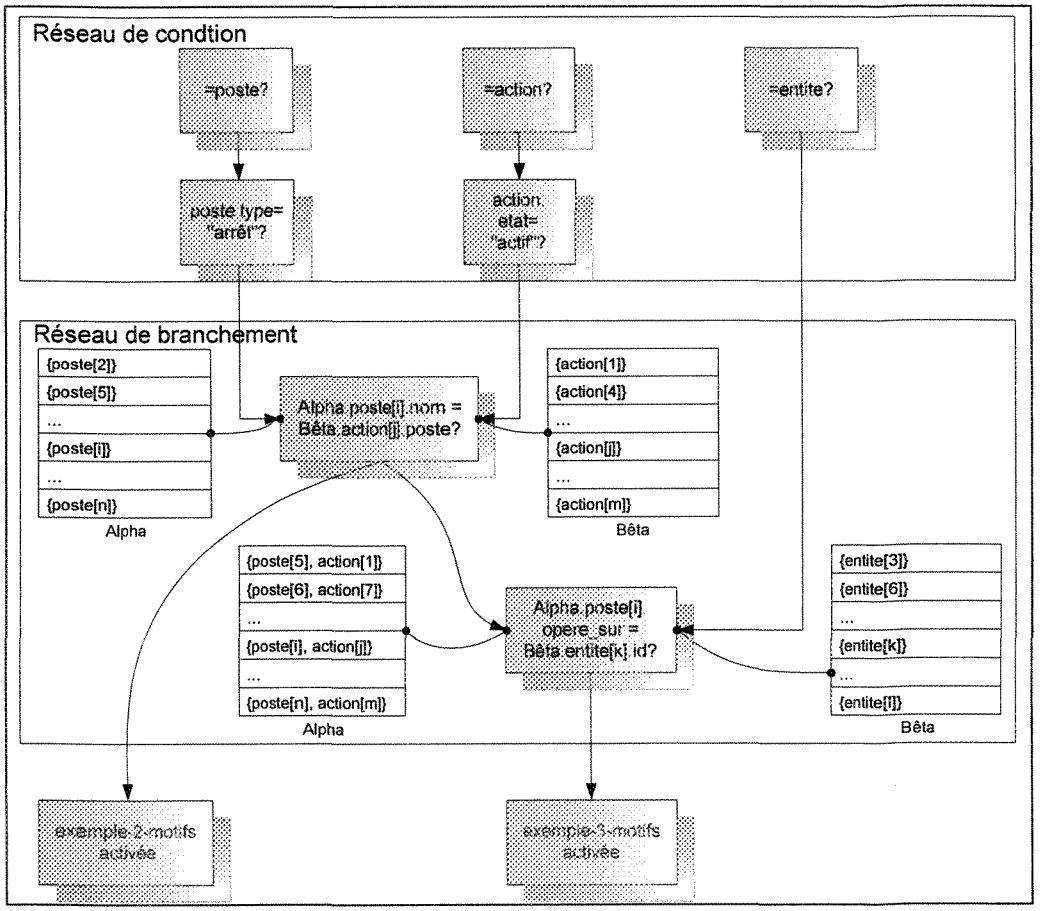

Figure 18 : partage des nœuds de condition et de branchement

2.8.2.4 Analyse de l'algorithme de Rete

À l'instar de la solution naïve, la complexité de l'algorithme de Rete, lors du premier cycle d'inférence est aussi $\mathbf{O(R \cdot F^C)}$. En effet, puisqu'il n'y a aucun résultat antérieur mémorisé, il faut effectuer les *associations de conditions*. Concernant les cycles d'inférences suivants, dans le pire des cas, c'est-à-dire lorsque que tous les faits changent et qu'il n'y a pas de nœuds partagés, la performance demeure sensiblement la même que l'algorithme naïf [Friedman-Hill, 2003].

L'amélioration la plus notable se situe au niveau du cas moyen, alors que peu de faits sont modifiés. Dans cette situation on s'attend à une complexité $\mathbf{O(R' \cdot F'^{C'})}$ où,

1. **R'** (nombre de règles associées aux faits modifiés) < **R** (nombre de règles)
2. **F'** (nombre de faits modifiés) < **F** (nombre de faits)
3. **C'** (nombre moyen de conditions des règles associées aux faits modifiés) < **C** (nombre moyen de conditions des règles).

Le principal désavantage de l'algorithme de Rete est qu'il utilise beaucoup de mémoire. En effet, la préservation de plusieurs milliers d'*associations partielles* et d'*associations de conditions* peut s'avérer coûteuse en espace de stockage. Toutefois, le compromis entre le temps d'exécution et la quantité de mémoire vaut souvent la peine. Mentionnons qu'il existe tout de même des approches pour optimiser l'utilisation de la mémoire. Notamment en réécrivant les règles, de manière à ce que Rete ait beaucoup moins d'*associations partielles* à effectuer [Giarratano et Riley, 1998] [Jess 7.0 Manual].

2.9 Agenda des activations

Au cours de chaque cycle d'inférence, la liste des règles actives, pouvant potentiellement être déclenchées, doit être maintenue. Cette liste de règles éligibles est appelée l'*ensemble des conflits*. Cependant, il importe d'établir leur ordre de déclenchement. Pour ce faire, une *stratégie* préétablie est utilisée, afin d'affecter une priorité d'exécution à chacune des règles de l'*ensemble des conflits*. Ce processus d'ordonnancement est appelé la *stratégie de résolution de conflits* (de l'Anglais « Conflict Resolution Strategy ») [Giarratano, 2002]. La liste des règles, issues du processus de résolution de conflits, constitue l'*agenda des activations* [Friedman-Hill, 2003]. Une application adjointe à un système expert, se termine une fois que son *agenda des activations* est vide.

Il peut arriver qu'à chaque cycle d'inférence, l'*agenda des activations* ne contienne pas plus d'une règle. La *stratégie de résolution de conflits* est alors superflue. Du même coup, recourir à un système expert s'avère tout aussi inutile. En effet, cette conjoncture implique qu'il est possible de

résoudre la problématique à l'aide d'un programme séquentiel, écrit dans un autre langage de troisième génération [Giarratano, 2002]. Il est donc fondamental de vérifier si le problème a une solution algorithmique séquentielle, avant même de considérer une approche basée sur un système expert.

2.9.1 Définition de l'importance des règles

La plupart des coquilles de développement permettent d'affecter à chaque règle une priorité dans l'*agenda des activations* [Giarratano, 2002] [Jess 7.0 Manual]. Pour ce faire, on assigne à une propriété particulière l'*importance* relative de la règle (de l'Anglais « Salience »). Plus la valeur de cette propriété est élevée, plus l'*importance* de la règle est accrue. Lorsque deux règles ont des *importances* différentes, la stratégie de résolution de conflits n'entre pas en ligne de compte. Cependant, en affectant des priorités distinctes à plusieurs règles, on risque de détériorer les performances du système expert, du moins en ce qui concerne la *stratégie de résolution de conflits*. De plus, lorsque l'ordre d'activation des règles est fortement établi, l'usage d'un système expert doit être reconsidéré, puisqu'une fois encore, le problème peut aussi bien être résolu par un programme séquentiel [Friedman-Hill, 2003].

2.9.2 Stratégie de résolution de conflits

Chaque coquille de développement de système expert a sa propre *stratégie de résolution de conflits*. Généralement, une *stratégie* efficace doit considérer la spécificité ou la complexité de chaque règle et le moment de leur activation. CLIPS [Giarratano, 2002] et Jess [Jess 7.0 Manual] utilisent par défaut une *stratégie de résolution de conflits* dite *en profondeur* (« Depth ») ou FIFO (« First In First Out »). Celle-ci signifie qu'à *importance* égale, les règles les plus récemment activées sont d'abord déclenchées. L'agenda agit donc comme une pile de règles. Les deux coquilles de développement mettent aussi en œuvre une *stratégie* dite *en largeur* (« Breadth ») ou LIFO (« Last In First Out »). Elle considère que l'agenda est une file, où les règles sont déclenchées dans leur ordre d'activation. CLIPS intègre également trois *stratégies* alternatives

[Giarratano, 2002]. Quant à Jess, il permet uniquement d'utiliser les stratégies FIFO en *profondeur* et LIFO en *largeur*. Toutefois, l'élaboration de *stratégies* personnalisées est possible.

L'approche *en profondeur* s'avère suffisante pour résoudre la plupart des problèmes. Certaines situations délicates peuvent toutefois survenir. C'est notamment le cas, lorsque toutes les règles déclenchées activent d'autres règles. Les activations les plus anciennes sont alors repoussées dans l'agenda et n'ont jamais l'occasion de se déclencher [Friedman-Hill, 2003]. À ce titre, nous allons discuter de la *stratégie de résolution de conflits* que nous avons opté à la section 3.4.4.3.

2.10 Moteur d'exécution

Il s'agit d'une composante qui est responsable de l'exécution des actions de la *partie droite* (RHS) d'une règle. Sous sa forme la plus simple, cette tâche revient à affirmer, retirer ou modifier des faits. Cependant, pour la plupart des systèmes experts modernes, le déclenchement d'une règle implique des actions beaucoup plus complexes. Ainsi, certaines coquilles permettent de définir les actions dans un langage de programmation structuré. Dans ce cas, le *moteur d'exécution* représente l'environnement dans lequel ce langage de programmation s'exécute. Pour d'autres systèmes, le *moteur d'exécution* agit comme un interpréteur de langage ou encore, comme intermédiaire responsable d'invoquer un programme compilé ou une librairie externe [Friedman-Hill, 2003].

2.11 Interface utilisateur

L'*interface utilisateur* permet d'échanger des informations, entre le système expert et l'usager. Pour ce faire, elle met à la disposition de l'utilisateur, un ensemble de moyens d'interactions tels : des menus, des boîtes de textes, le langage naturel ou un affichage graphique. Plus précisément, voici quelques fonctions qui lui sont propres [Gonzalez et Dankel, 1993] :

- Permettre à l'utilisateur de fournir au système expert des informations sur l'état du problème. Par exemple, le moteur d'inférence à *chaînage arrière* peut demander à l'usager de saisir des

données supplémentaires pour atteindre un objectif. Également, le système expert peut requérir un ensemble de faits initiaux, avant le commencement du processus d'inférence.

- Fournir des explications pour justifier certaines demandes d'informations faites par le système expert.
- Permettre à l'utilisateur de formuler des questions quant aux conclusions qui sont émises.
- Afficher les conclusions obtenues sous forme de texte, de graphiques, etc.
- Permettre à l'utilisateur de préserver ou d'imprimer les conclusions, une fois le processus d'inférence terminé.

2.12 Composante d'acquisition des connaissances

La *composante d'acquisition* est un outil qui assiste le spécialiste, lors l'élaboration de la base de connaissances. Comme on l'a mentionné, ce spécialiste interagit généralement avec l'expert du domaine, afin d'acquérir le savoir et de l'encoder sous une forme particulière. Citons en exemple, des règles rédigées selon la syntaxe LISP et utilisables par une coquille de développement comme CLIPS ou Jess. Sous sa forme la plus simple, cet outil agit comme un éditeur de la base de connaissances. Ainsi, il fournit au spécialiste une vue de la base de connaissances et un moyen d'y apporter des changements. Sous sa forme la plus complexe, la *composante d'acquisition* peut disposer d'un éventail de fonctionnalités [Gonzalez et Dankel, 1993], en voici quelques-unes :

- Localiser les problèmes associés aux éléments de la base de connaissances.
- Comparer les éléments qui forment la base de connaissances.
- Conserver les modifications qui ont été faites.
- Fournir des fonctionnalités d'éditions et de saisies avancées, etc.

Nous terminons ainsi le second chapitre. Au cours de celui-ci, nous avons vu en détail les fondements des systèmes experts et pourquoi ils font l'objet de notre intérêt. Nous allons maintenant présenter la problématique à laquelle nous avons appliqué notre stratégie de

modélisation. Nous allons également montrer l'architecture du modèle qui nous a permise de résoudre cette problématique.

CHAPITRE 3

MODÉLISATION DU CYCLE DE VIE DES ANODES

3 Modélisation du cycle de vie des anodes

3.1 Cycle de vie des anodes

La présente section introduit le fonctionnement du *cycle de vie des anodes* d'une aluminerie. Ce processus est divisé en quatre étapes : *formation des anodes*, *manutention et entreposage d'anodes*, *scellement d'anodes* et *production du métal*. Le *cycle de vie des anodes* comprend un ensemble de ressources, appelées « postes », reliés par un réseau de files, nommées « convoyeurs ». Les postes sont associés à des activités qui contribuent à faire évoluer l'état des anodes. Ils permettent aussi d'aiguiller les anodes sur les bons convoyeurs, en se basant sur des règles de routage. Ces règles peuvent être modifiées selon les messages qui sont émis par d'autres postes. Nous allons décrire brièvement chacune des étapes du processus. L'Annexe 1 contient un glossaire des principaux termes du *cycle de vie des anodes*.

3.1.1 Formation des anodes

Les anodes qui sont produites dans une aluminerie sont en fait des blocs rectangulaires formés à partir d'une pâte chaude. Cette *pâte d'anode* est composée de coke de pétrole, de goudron liquide et de résidus d'anodes broyées [Totten et Mackenzie, 2003]. Une fois suffisamment refroidie, la pâte est coulée dans des moules. Elle prend sa forme rectangulaire, sous l'action d'une presse hydraulique ou d'un vibrocompacteur mécanique [Beghein et al., 2003]. Le produit obtenu, par l'une ou l'autre de ces méthodes, est une *anode crue*. Celle-ci est refroidie, en transitant à travers un bassin d'eau. La Figure 19 décrit en détail l'enchaînement de ces activités.

Une fois formée, les *anodes crues* peuvent être entreposées ou cuites. La cuisson s'effectue sous l'effet de la chaleur radiante, émise par des gaz à haute température. Cette chaleur circule à travers les cloisons d'un gigantesque four. Le four est subdivisé en deux rangées de chambres de cuisson. Les anodes crues sont empilées dans les alvéoles de ces chambres [Totten et Mackenzie, 2003].

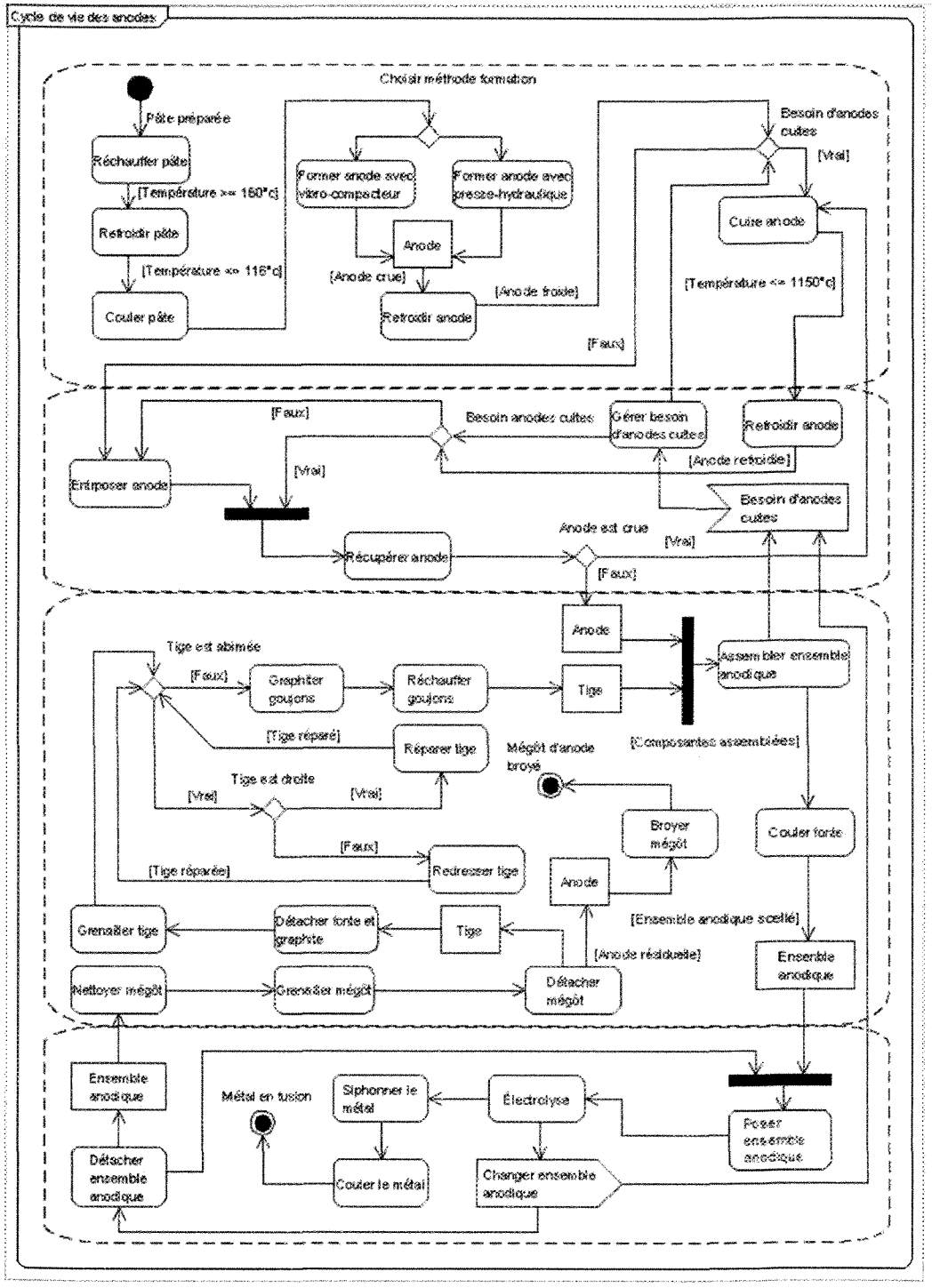

Figure 19 : diagramme des activités du cycle de vie des anodes

La cuisson permet d'éliminer les impuretés des anodes, d'améliorer leur conductivité et leur solidité. L'extrant de cette activité est ce qu'on appelle une *anode cuite* ou *précuite*. En effet, il existe aussi des alumineries qui font appel à des anodes dites *auto-cuites* ou *Söderberg*. Ces anodes sont composées d'une pâte qui est contenue dans un caisson d'acier. Elles sont envoyées à l'étape de *production du métal*, sans avoir été préalablement cuites dans un four [World-Aluminium.org]. Dans le cadre du mémoire, nous nous intéressons uniquement au procédé basé sur les anodes *précuites*.

3.1.2 Entreposage et manutention des anodes

Les *anodes crues* et *cuites* qui transitent en aval de l'étape de *formation des anodes* peuvent être accumulées dans un entrepôt sec, à l'aide de grues gerbeuses. L'opération d'entreposage des *anodes crues* survient lorsque les fours ont cessé d'opérer ou lorsqu'ils n'arrivent plus à répondre à la demande. Quant à l'opération d'entreposage des *anodes cuites*, elle se s'effectue lors de l'arrêt du secteur de *scellement*. L'entrepôt agit donc comme un réservoir d'*anodes cuites* et *crues*.

Par ailleurs, les grues de l'entrepôt ont aussi la possibilité de récupérer des anodes *crues* ou des anodes *cuites* [Poirier, 1997]. La première option implique le routage des *anodes crues* vers les fours, lorsque ces derniers n'en reçoivent plus suffisamment. Une fois cuites, les *anodes* retraversent le secteur de la manutention. Quant à la seconde option, où l'on récupère des *anodes cuites*, elle consiste à les expédier directement à l'atelier de *scellement* s'il ne fonctionne pas à sa pleine capacité.

3.1.3 Scellement des anodes

L'étape de scellement doit permettre de produire ce qu'on appelle un *ensemble anodique*. Il est formé d'une *anode cuite*, fixée aux *goujons* d'une *tige* métallique, par de la *fonte de fer*. Les intrants du secteur du scellement sont donc : une *anode cuite* et un *ensemble anodique*, dont l'anode forme un *mégot*. Le *mégot* est une anode résiduelle issue de l'électrolyse [Totten et Mackenzie, 2003]. Il est question de cette activité un peu plus loin. L'*anode cuite* provient de l'étape

de *manutention et d'entreposage*. Quant à l'*ensemble anodique* résiduel, il s'agit d'un extrant de l'étape de *production du métal*.

Durant l'étape de *scellement*, le *mégot* est nettoyé en projetant des grenailles, puis détaché de la *tige*, pour être finalement broyé. On enlève ensuite la *fonte de fer* qui est fixée aux *goujons* de la *tige*. La *tige* est également nettoyée par grenaillage. La *tige* est redressée si elle est recourbée et ses *goujons* sont changés s'ils sont abîmés. Une fois que la *tige* est réparée, on enduit ses *goujons* avec du graphite et on les réchauffe. Puis, la *tige* et l'*anode cuite* sont assemblées et scellées par la *fonte de fer* [Beghein et al., 2003].

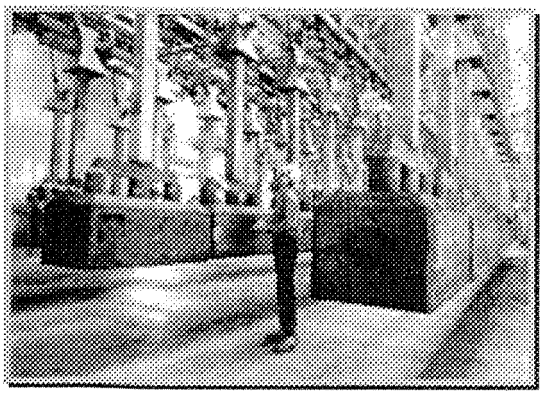

Figure 20 : ensembles anodiques (tige métallique scellée à l'anode cuite)

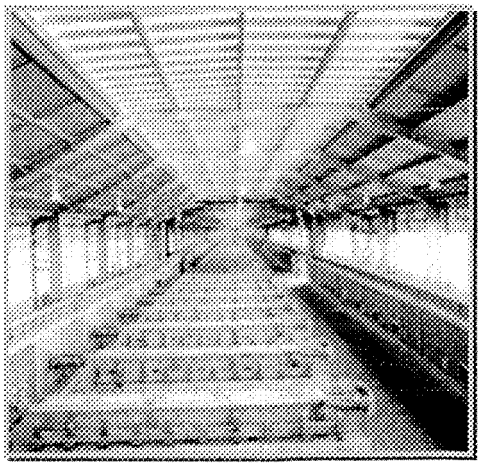

Figure 21 : salle des cuves (cellules de réduction de l'aluminium)

3.1.4 Production du métal

Le procédé Bayer est une suite d'étapes mécaniques et chimiques qui permet de raffiner un minerai appelé bauxite, afin d'en extraire l'alumine (Al_2O_3). L'alumine est exploitée industriellement pour obtenir de l'aluminium (Al) à partir du procédé Hall-Héroult [Totten et Mackenzie, 2003]. Ce procédé consiste à dissoudre l'alumine dans un électrolyte, aussi appelé solution ou bain électrolytique. L'électrolyte contient plusieurs composants chimiques ioniques. Le principal composant est la cryolite (Na_3AlF_6). On ajoute également d'autres composants, tels du LiF, AlF_3 et MgF_2, pour abaisser la température de fusion et ainsi accroître le rendement en aluminium. On

diminue aussi la température pour des questions environnementales. En effet, des gaz fluorés sont émis, lorsque la température est trop élevée. Le bain est contenu dans un réceptacle de matériaux carbonés contenus dans une cuve d'acier, aussi appelée cellule de réduction. L'aluminium ayant une densité supérieure au bain, il se dépose au fonde de la cuve. Il fait ainsi office de cathode [Beghein et al., 2003].

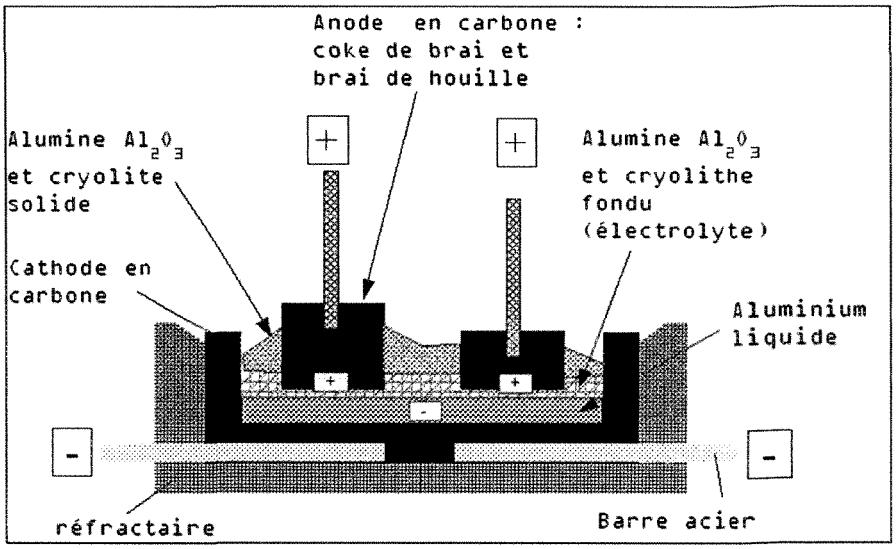

Figure 22 : électrolyse de l'aluminium dans une cellule de réduction

Pour qu'une réaction soit initiée, on effectue l'électrolyse du bain, en faisant passer un courant électrique à travers l'électrolyte. L'ensemble anodique est suspendu au-dessus de la cuve et l'anode trempe dans le bain électrolytique. Un courant électrique continu de plus de 300 kiloampères traverse l'électrolyte, en partant de la tige et de l'anode cuite (électrode positive), jusqu'à l'aluminium (électrode négative). La réduction de l'aluminium se produit sous l'action du courant électrique, le métal liquide se dépose au fond du bain. L'anode est graduellement consumée, puisqu'elle intervient dans la réaction de réduction, la réaction globale du procédé Hall-Héroult étant [Zumdahl, 1988] :

$$2Al_2O_3 + 3C \rightarrow 4Al + 3CO_2$$

Le métal en fusion est siphonné au fond de la cuve et conservé dans des creusets. Parfois, l'aluminium est mélangé à d'autres métaux pour former des alliages. Généralement, l'aluminium est

coulé pour former des lingots ou des billettes. Au terme de l'électrolyse, l'anode cuite devient un mégot et doit être remplacée. La tige de l'ensemble anodique est détachée et un nouvel ensemble anodique est suspendu au dessus de la cellule de réduction.

3.2 Formulation du problème et objectifs généraux

Nous avons appliqué notre stratégie de conception d'*outils d'aide à la décision*, à une problématique réelle de *modélisation* d'un système. Le problème s'inscrit dans le cadre d'un projet de restructuration des opérations d'une aluminerie. Cette restructuration a des impacts au niveau des étapes de *formation, manutention et entreposage des anodes*. Il devenait donc impératif pour les gestionnaires de connaître quels sont les possibilités d'amélioration de ce système. Nous avons donc conçu un modèle qui reproduit les principales activités et procédures opérationnelles. Cette section discute des aspects qui ont fait l'objet de notre attention, dans le cadre de l'application de notre stratégie aux étapes de *formation, manutention et entreposage des anodes*.

3.2.1 Objectifs spécifiques de l'étude

Afin de s'assurer de l'efficacité de notre stratégie de conception d'*outils d'aide à la décision*, nous avons défini une suite d'objectifs tangibles que le modèle doit nous permettre d'atteindre :

- Pouvoir simuler les opérations du système, sur une période allant d'une minute à sept jours, c'est-à-dire une semaine complète de production. Le début d'une simulation se fait à 00h00, dans la nuit de dimanche à lundi.
- Pouvoir assigner des *activités planifiées* et *non-planifiées* à chacun des postes du système, afin d'étudier leurs impacts respectifs :
    a. *Activités planifiés* : Ce sont des actions auxquelles les postes sont assujettis en vertu d'une planification précise. Par exemple, les *arrêts planifiés*, imputables à la maintenance des machines ou à l'horaire des employés, font partie de ce type d'activité.
    b. *Activités non-planifiés* : Ce sont des actions auxquelles les postes sont assujettis et dont le temps d'occurrence et la durée sont aléatoires. Ces activités doivent être conformes

aux distributions de probabilités produites à partir d'observations du système. À titre d'exemple, citons les *arrêts* imputables à des bris mécaniques ou des accidents. Le modèle doit pouvoir ignorer les *arrêts non-planifiés*, pour pouvoir étudier le cas optimum, où tous les postes sont fiables à 100 %.

- Pouvoir assigner différentes vitesses aux convoyeurs du système, afin d'étudier leurs impacts respectifs.
- Pouvoir assigner différents horaires aux fours du système, afin de faire varier leurs cycles de cuissons.

3.2.2 Mesures de performances

Les mesures de performances utilisées pour évaluer l'efficacité des scénarios sont les suivantes :

- Pour chaque poste, on doit afficher de façon continue :
  a. Son état, parmi les suivants : en traitement, en attente d'anode(s) ou en arrêt (planifié ou non-planifié).
  b. Le nombre d'anodes qui sont entrées et sorties par le poste.
- Pour les postes qui sont des fours, on doit connaître en continu, le nombre d'anodes qui sont cuites, c'est-à-dire prêtes à être écoulées en aval du processus, vers l'entrepôt ou le scellement. De plus, il faut afficher un état additionnel : *bloqué*. Celui-ci permet de savoir si le four peut écouler les anodes qui viennent d'être cuites. Un blocage survient lorsque toutes les files en aval d'un four sont à leur pleine capacité.
- Pour l'entrepôt, on doit connaître de façon continue, le nombre d'anodes *cuites* et *crues* (*non-cuites*). Ces informations doivent aussi être préservées pour chaque jour d'opération de la phase de *manutention et d'entreposage d'anodes*.
- Pour chaque file, on doit savoir sa capacité maximale et afficher de façon instantanée le nombre d'anodes actuellement en transit.

3.2.3 Étendue du modèle

Le modèle ne doit pas couvrir toutes les opérations du *cycle de vie des anodes* mais plutôt celles qui s'étalent de la sortie des anodes de la *tour à pâte*, en passant par les *fours* et le secteur de la *manutention*, jusqu'à leur entrée du centre de *scellement*. Le système modélisé correspond donc aux étapes de *formation, manutention et entreposage des anodes*. Celles-ci sont décrites avec plus de précision à la section suivante.

3.3 Collecte des données et définition du modèle

Au cours de cette section, nous allons discuter des principales informations qui ont été fournies par l'utilisateur du modèle, c'est-à-dire la configuration et les procédures d'opérations du système. Dans un deuxième temps, nous allons présenter les principales hypothèses de modélisation. Il est question des différentes approches probabilistes possibles et de celle qui a été retenue. Également, nous allons aborder la stratégie d'intégration des événements discrets. Plus particulièrement, nous discutons de la gestion du temps et de l'approche de conception du contrôleur.

3.3.1 Configuration du système

L'utilisateur du modèle nous a fourni des documents qui illustrent la configuration et le fonctionnement du processus. Ceux-ci traitent surtout des postes des secteurs de la *formation, de la manutention et de l'entreposage des anodes*. Ainsi, nous avions entre nos mains, une étude qui décrit les stratégies de gestion de l'entrepôt [Poirier, 1997]. Celle-ci nous a permis de définir les règles de fonctionnement d'une partie du modèle.

Comme nous avons mentionné à la section 3.2, nous allons uniquement considérer les étapes de *formation, manutention et entreposage des anodes*. Cependant, pour des fins de modélisation, certaines activités, telles que décrites à la Figure 19, ont soient été délibérément regroupées au sein d'un même poste, soient modélisées à un niveau de détail moindre que la

réalité ou encore, tout simplement omises, lorsqu'elles étaient jugées non-pertinentes. Le diagramme de la Figure 23 (voir ci-dessous) montre les principales activités qui ont été retenues.

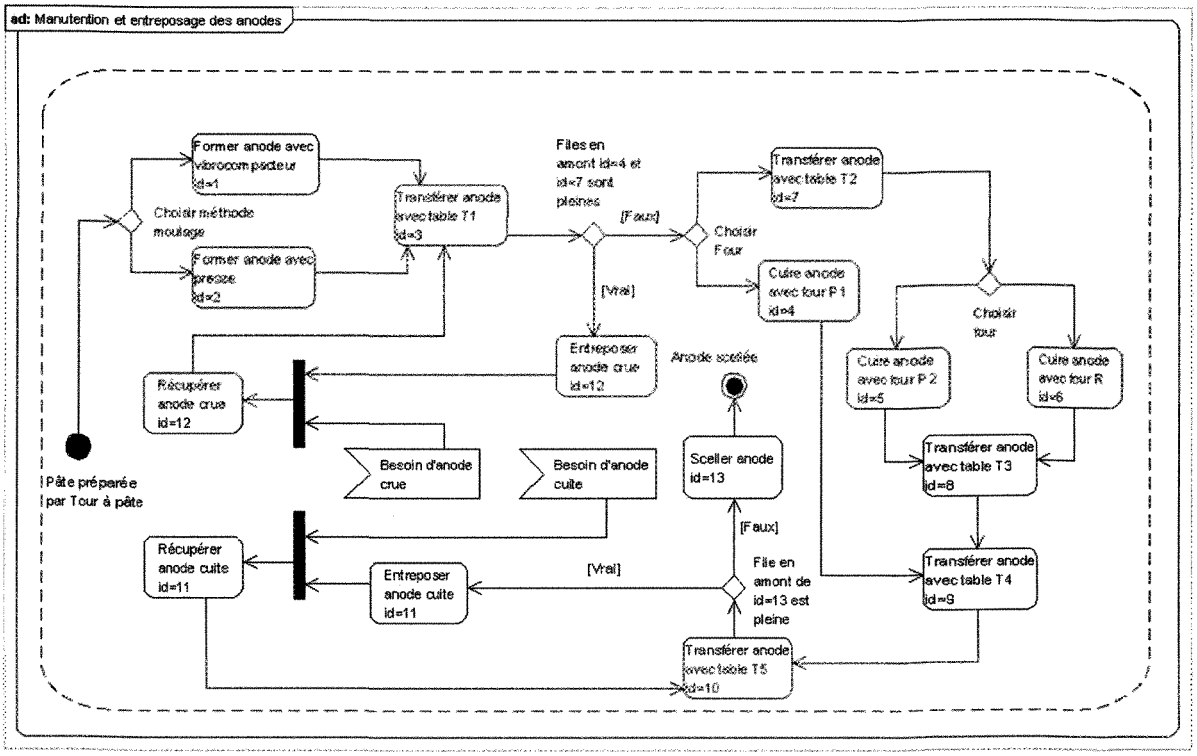

Figure 23 : formation, manutention et entreposage des anodes

Chacune des activités, à l'exception de l'entrepôt, symbolisent un poste qui contribue à la transformation ou à l'aiguillage des anodes. Le cas particulier de l'entrepôt est discuté un peu plus loin. La *tour à pâte* n'est pas représentée en tant qu'une ressource du modèle. En fait, son temps de traitement est additionné aux postes *vibrocompacteur* (*id=1*) et *presse* (*id=2*). Le processus débute donc avec le choix de la méthode de formation de l'anode crue, c'est-à-dire à partir du *vibrocompacteur* ou de la *presse*. Puis, les anodes transitent à travers une file, c'est-à-dire un bassin de refroidissement. Le bassin des anodes compactées a un temps de transit supérieur à celui des anodes pressées. C'est d'ailleurs pourquoi il a fallu créer deux postes distincts, *vibrocompacteur* (*id=1*) et *presse* (*id=2*), bien qu'ils aient le même temps de traitement. Si un convoyeur en amont du *four P1* (*id=4*) ou de la *table T2* (*id=7*) n'a pas atteint sa pleine capacité,

alors les anodes de la *table T1* (*id=3*) sont aiguillées vers celui-ci. Dans le cas contraire, elles sont envoyées à l'*entrepôt* (*récupérer anode crue id=12*) qui se charge de les accumuler.

L'*entrepôt* est desservi par les grues *G1* (*id=11*) et *G2* (*id=12*). Elles permettent de *décharger* ou de *charger* les anodes sur les convoyeurs adjacents. Le *chargement* d'un convoyeur implique la *récupération* d'un certain nombre d'anodes de l'*entrepôt*, pour les expédier vers les *fours* (anodes crues *id=12*) ou le *scellement* (anodes cuites *id=11*). Quant au *déchargement* d'un convoyeur, il nécessite l'*entreposage* des anodes provenant de la *table de transfert T1* (anodes crues *id=12*) ou des *fours* (anodes cuites *id=11*). Conséquemment, l'*entrepôt* agit comme un réservoir d'anodes crues et d'anodes cuites. Il est divisé en deux sections et une grue est affectée à chacune d'elle. Une section a une capacité de 7922 anodes, pour une capacité totale d'*entreposage* de 15984 anodes.

La grue *G1* (*id=11*) est assignée prioritairement au traitement des anodes cuites et la grue *G2* (*id=12*) à celui des anodes crues. Les grues peuvent répondre à quatre signaux différents. Certains signaux ont préséances sur d'autres, afin d'éviter que le scellement manque d'anodes *cuites*. L'ordre de priorité est le suivant : *récupérer* les anodes *cuites*, *entreposer* les anodes *cuites*, *entreposer* les anodes *crues* et *récupérer* les anodes *crues*. Comme nous allons le voir plus loin, les signaux sont émis selon l'état des *fours* (*id=4*, *id=5*, *id=6*) et du *scellement* (*id=13*). Sur la Figure 23, chacun des signaux est associé à une activité. Mentionnons toutefois que cette assignation peut changer au gré de la disponibilité des *grues* et des différents signaux en présences.

L'étape *de formation des anodes* inclut trois fours, à savoir le four *P1* (*id=4*), le four *P2* (*id=5*) et le four *R* (*id=6*). Les fours fonctionnent selon des cycles de cuissons de 32, 36 ou 40 heures. Le cycle de 36 heures est celui qui est considéré pour le scénario de référence. Au terme d'un cycle de cuisson, les anodes sont cuites et prêtes à être écoulées en aval, soit vers l'*entrepôt* (*récupérer anode cuite id=11*) ou vers le *scellement* (*id=13*). Un four est divisé en plusieurs chambres. Les anodes sont placées dans les alvéoles des chambres de cuisson. Chaque four est desservi par un train. Celui-ci fait la navette entre l'entrée du four et chacune des chambres de cuissons. Ainsi, on

peut récupérer les anodes cuites et alimenter les alvéoles des fours en anodes crues. Les anodes récupérées sont écoulées une à une, en aval du procédé. Mentionnons que les activités d'alimentation et de récupération n'ont pas été explicitement modélisées. Nous les avons intégrées en une activité globale *cuire anode* (*id=4*, *id=5* et *id=6*), parce que cela permettait de simplifier le modèle, sans affecter son réalisme.

Le *scellement* (*id=13*) est l'activité terminale du procédé. Il est crucial que le *scellement* soit continuellement alimenté par un des fournisseurs d'anodes cuites, c'est-à-dire les fours *P1* (*id=4*), *P2* (*id=5*), *R* (*id=6*) et l'*entrepôt* (*récupérer anode cuite id=11*). Tel que nous avons discuté à la section 3.1.3, les activités de l'atelier de *scellement* permettent de fixer l'anode cuite à une tige métallique, d'où l'état final *Anode scellée* sur la Figure 23. Toutefois, nous nous sommes contentés de modéliser ces activités en un seul poste, puisqu'un niveau de détail accru n'était pas nécessaire pour répondre aux objectifs de l'étude.

3.3.2 Procédures d'opération du système

Procédure d'alimentation et de récupération des fours

Les trains qui alimentent les chambres de cuissons ont une capacité de 15 anodes pour le four *R* et de 18 anodes pour les fours *P1* et *P2*. Ces navettes suivent un horaire qui varie selon le cycle de cuisson du four. Lorsqu'une anode crue arrive d'un poste en amont, elle est empilée à l'entrée du four par une grue gerbeuse. Puis, le train d'anodes cuites quitte les chambres de cuisson à l'instant prescrit par son horaire, c'est-à-dire au terme d'un cycle complet de cuisson. Les anodes cuites sont alors déchargées et empilées à l'entrée du four par une autre grue. Elles sont ensuite écoulées une à une, vers les postes en aval, soit l'*entrepôt* (*récupérer anode cuite id=11*) ou le *scellement* (*id=13*). Tout de suite après le déchargement des anodes cuites, les anodes crues sont chargées à leur tour sur le train, pour être placées dans les alvéoles des chambres de cuisson. Le processus d'alimentation et de récupération est ensuite interrompu, en attendant le prochain départ de train. Les tableaux de l'Annexe 2 décrivent l'horaire des navettes de train. Les fours doivent être continuellement alimentés en anodes crues, afin de répondre aux demandes du

scellement (*id=13*). On considère que les fours manquent d'anodes *crues*, lorsque le convoyeur en amont du *four P1* (*id=4*) est à moins de 75% de sa capacité ou encore, lorsque celui en amont de la *table T2* (*id=7*) est à moins de 79%. Également, on statue que les fours produisent peu d'anodes *cuites* si le convoyeur en amont de la table T5 (*id=10*) n'est pas plein ou bien si les deux convoyeurs en amont de la *table T3* (*id=8*) sont à moins de 50% de leur capacité. Notons que le signal de *récupération* des anodes *crues*, dont il est question un peu plus bas, est tributaire de cette règle empirique.

Procédure d'alimentation du scellement

Dès que l'atelier de *scellement* (*id=13*) est fonctionnel, il faut s'assurer que son convoyeur en amont soit toujours à 100% de sa capacité, pour ne jamais manquer d'anodes cuites. Afin de faire varier le nombre d'anodes de ce convoyeur, le *scellement* (*id=13*) peut recourir aux signaux d'*entreposage* et de *récupération* des anodes cuites (*id=11*).

Procédure d'entreposage et de récupération des grues

Puisque l'*entrepôt* agit comme un réservoir d'anodes *cuites* ou *crues*, les *grues G1* et *G2* accomplissent l'*entreposage* et la *récupération* des anodes en fonction de l'état des *fours* (*id=4*, *id=5*, *id=6*) et du *scellement* (*id=13*). Lorsque le *scellement* est activé et que les *fours* produisent peu d'anodes *cuites*, alors le signal de *récupération* des anodes *cuites* est émis, pour l'une des deux *grues*. Par contre si le *scellement* est désactivé et que son convoyeur en amont est plein, alors le signal d'*entreposage* des anodes *cuites* est émis. Également, lorsque les *fours* reçoivent peu d'anodes *crues*, alors on génère le signal de *récupération* des anodes *crues*, afin de pouvoir les alimenter. Finalement s'il y a suffisamment d'anodes *crues* sur le convoyeur adjacent à l'*entrepôt*, alors on émet le signal d'*entreposage* des anodes *crues*.

Contrairement aux autres ressources, les grues *G1* et *G2* manipulent plusieurs anodes en même temps. Elles ont une *capacité* de d'*entreposage* et de *récupération* de 12 anodes à la fois. Pour l'*entreposage*, les *grues* prélèvent donc les anodes d'un convoyeur adjacent, seulement s'il contient *au moins 12 anodes* prêtes à être entreposées. De la même manière, la grue *G1* n'entame

la récupération que s'il est possible de placer *au moins 12 anodes* sur un convoyeur adjacent. Toutefois, parce que la grue *G2* est plus éloignée dans l'*entrepôt*, elle commence à récupérer ses 12 anodes, seulement s'il est possible de placer *au moins 24 anodes* sur le convoyeur adjacent. La *récupération* dépend donc d'un autre paramètre que la *capacité* de la grue, soit l'*espace* nécessaire (en anodes) sur le convoyeur. En résumé, les règles opérationnelles pour l'*entreposage* et la *récupération* sont décrites ci-dessous. Notons qu'elles sont les mêmes quelque soit l'état de l'anode, c'est-à-dire *crue* ou *cuite*.

Entreposage $\Leftrightarrow N \geq E_i$

- Où i : indice de la grue, $i \subseteq [1..2]$
- N : nombre d'anodes actuellement en transit sur un convoyeur *c*
- E_i : la capacité d'*entreposage* (c'est-à-dire de *déchargement* de *c*) avec la grue i

Récupération $\Leftrightarrow M - N \geq R_i$

- M : capacité maximale d'un convoyeur *c*
- N : nombre d'anodes actuellement en transit sur un convoyeur *c*
- R_i : espace nécessaire pour la *récupération* (c'est-à-dire de *chargement* de *c*) avec la grue i

Procédure de routage des autres ressources

Comme nous venons de le mentionner, les grues de l'*entrepôt* (*id=10*, *id=11*) doivent aiguiller les anodes, en fonction des besoins particuliers. Cependant, il existe d'autres règles de routage, dites génériques, pour l'ensemble des ressources des étapes de *formation, manutention et entreposage des anodes*. La première de ces règles implique qu'une ressource ne peut aiguiller des anodes vers un convoyeur qui a atteint sa pleine capacité. De plus, lorsqu'une ressource est liée à plusieurs convoyeurs en aval, celui-ci cherche à distribuer les anodes également entre eux. Pour ce faire, une estampille temporelle (traduction de l'Anglais « timestamp » selon le [Dictionnaire terminologique]) est affectée lorsqu'un convoyeur reçoit une anode d'une ressource en amont. Les anodes traitées par une ressource sont donc toujours aiguillées vers le convoyeur qui a la plus petite estampille. À l'inverse, un poste va prélever une anode d'un convoyeur en amont que s'il a plus d'une anode en transit. Toutefois si ce poste est lié à plus d'un convoyeur en amont, il récupère aussi l'anode du convoyeur qui a la plus petite estampille temporelle, c'est-à-dire celui sur lequel les anodes s'accumule depuis le plus longtemps. En effet, l'estampille des convoyeurs en amont d'un poste est affectée dès qu'une anode est récupérée sur l'un d'entre eux.

Procédure de gestion des arrêts planifiés des ressources

Comme nous avons mentionné, le fonctionnement des postes du *cycle de vie des anodes* est en partie régi par des *arrêts planifiés*. Ceux-ci sont dus à des événements prédéterminés qui impliquent une interruption des opérations, comme par exemple : l'horaire de travail des employés et les périodes d'entretien ou de vérification des équipements. Le Tableau 2 représente la liste des *arrêts planifiés* au cours d'une semaine d'opération. Comme l'horaire de l'atelier de scellement est plus complexe, il est décrit à l'Annexe 2.

Nom du poste		Plage horaire des arrêts de la semaine	Arrêté durant toute la fin de semaine
Entrepôt	Grue n° 1	Mardi 8h00 à 16h00	Non
	Grue n° 2	Vendredi 8h00 à 16h00	Non
2 fours *P*	Four n° 1	Aucun	Oui
	Four n° 2	Aucun	Oui
Four *R*		Aucun	Non
Moule		Lundi 8h00 à 16h00	Non
Presse		Mercredi 8h00 à 16h00	Non
Scellement		Voir Annexe 2	Oui
Table de transfert 2030		Mardi 8h00 à 12h00	Non
Tour à pâte et basin		Mardi 8h00 à 16h00	Non

Tableau 2 : plage horaire des *arrêts planifiés*

3.3.3 Gestion des événements probabilistes

À la section 1.5, nous avons stipulé qu'un modèle déterministe n'intègre aucune notion de probabilité [Hoover et Perry, 1990]. Or, dans bien des cas, pour atteindre un niveau de réalisme raisonnable, il est recommandé de modéliser les événements probabilistes du système comme des variables aléatoires [Law et Kelton, 2000]. Dans un premier temps, nous allons discuter des événements du système qui sont sujets à un comportement probabiliste. Ensuite, nous allons aborder quelques façons d'intégrer ces variables au modèle. Finalement, nous mentionnons l'approche de modélisation qui a été retenue.

3.3.3.1 Variables aléatoires

Le *cycle de vie des anodes* est un réseau de files (convoyeur), sur lesquelles transitent les anodes. Les files desservent des ressources (postes). Les postes peuvent être soumis à des

activités non-planifiées dont la durée est variable. Citons par exemple, des *arrêts non-planifiés* occasionnés par des bris d'équipement ou des accidents de travail. Dans ce contexte, les principales variables aléatoires du modèle sont les suivantes :

- T : le temps (durée) du *Transit* d'une anode dans une file.
- S : le temps (durée) de *Service* d'une ou plusieurs anodes(s) à un poste.
- C : le temps de *Commencement* d'une *activité non-planifiée* à un poste.
- A : le temps (durée) d'une *Activité non-planifiée* à un poste.

3.3.3.2 Distributions de probabilités

Une première option consiste à rechercher les distributions (ou lois) de probabilité et leurs paramètres, afin d'obtenir des valeurs plausibles, pour chaque variable aléatoire. Or, nous ne connaissons pas le type de distribution des variables aléatoires ni leur moyenne ni même leur variance. Il existe un théorème statistique, appelé théorème central limite qui nous permet de résoudre en partie ce problème [Baillargeon, 1990]. Pour ce faire, il faut dans un premier temps calculer la moyenne d'un échantillon de taille n, prélevé à partir d'une population qui suit une distribution quelconque de moyenne μ et de variance σ^2. Puis, on répète cette expérience, afin d'obtenir d'autres moyennes d'échantillons, aussi appelées *moyennes échantillonnales*.

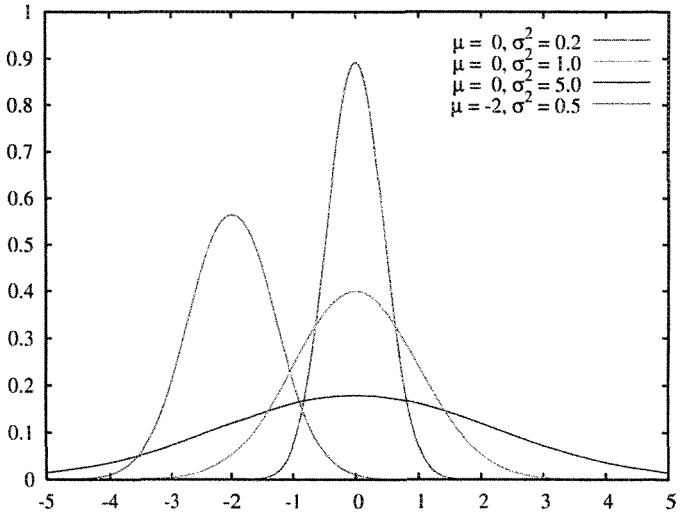

Figure 24 : lois normales de moyenne μ et de variance σ^2

Or, le théorème central limite stipule que la *distribution de la moyenne échantillonnale*, c'est-à-dire la distribution de toutes les moyennes d'échantillons prélevés, a une forme qui tend vers une distribution normale (voir Figure 24), à mesure que la taille de l'échantillon (n) augmente [Christensen, 1986]. D'ailleurs, les statisticiens ont observés que la *distribution de la moyenne échantillonnale* est presque normale, à partir de n = 30 [Kakmier, 1982]. La moyenne et la variance de la *distribution de la moyenne échantillonnale* ont les caractéristiques suivantes :

Pour sa moyenne $\mu_{\bar{x}} = \mu$ (lemme 1)

Pour sa variance $\sigma^2_{\bar{x}} = \frac{\sigma^2}{n}$ (lemme 2)

x : une variable aléatoire

$\bar{x}$: La moyenne échantillonnale pour la variable aléatoire

$\mu_{\bar{x}}$: La moyenne de la distribution de la moyenne échantillonnale

$\sigma^2_{\bar{x}}$: la variance de la distribution de la moyenne échantillonnale

μ : La moyenne de la population sur laquelle on observe x

σ^2 : la variance de la population sur laquelle on observe x

Une approche possible consiste donc à prélever un échantillon d'au moins 30 observations, pour chacune des variables aléatoires (T, S, C, A) et à calculer une *moyenne échantillonnale*. Ensuite, on répète cette expérience avec au moins 29 autres échantillons, pour obtenir d'autres moyennes. On peut ainsi calculer la moyenne et la variance de la *distribution de la moyenne échantillonnale*, ainsi que la moyenne μ et la variance σ^2 de la population, en vertu des lemmes 1 et 2. Quant aux distributions des variables aléatoires (T, S, C, A), on peut tenter de la faire correspondre avec une loi de probabilité théorique, par exemple une loi gamma, normale, log normale, U ou Weibull [Hoover et Perry, 1990] [Law et Kelton, 2000].

Il existe plusieurs méthodes qui permettent d'établir une correspondance avec une distribution de probabilité théorique. Celles-ci cherchent à établir une corrélation entre des lois de probabilité connues et un histogramme des fréquences qui représente la distribution des données [Chung, 2004] [Law et Kelton, 2000]. Cependant, les techniques d'estimation des lois de probabilité dépassent le cadre du présent mémoire. C'est pourquoi plusieurs spécialistes en modélisation utilisent par défaut la distribution normale. De même, il n'est pas rare d'utiliser un logiciel pour établir la correspondance entre les données du système et une loi de probabilité [Chung, 2004].

Mentionnons la bibliothèque Java SSJ (Simulation Stochastique en Java), du Département d'Informatique et de Recherche Opérationnelle (DIRO) de l'Université de Montréal [SSJ].

Une autre approche possible consiste à utiliser distribution empirique, autrement dit expérimentale, qui est construite à partir des données observées [Bratley et al, 1987] [Law et Kelton, 2000]. Une dernière méthode, s'apparentant à celle de la distribution empirique, fait appel à un algorithme d'estimation avec noyau de densité. NDLR : traduit de l'Anglais « Kernel Density Estimation » (KDE) [Hörmman et Bayar, 2000]. On prélève d'abord un échantillon de n valeurs notées X_1, X_2, ..., X_n. Puis, on applique l'algorithme KDE, afin d'obtenir une valeur aléatoire Y comme suit :

1. On choisit une valeur appelée paramètre de lissage noté p (voir la formule ci-dessous).
2. On génère ensuite un nombre aléatoire noté $i \subseteq [1..n]$.
3. On génère une valeur aléatoire B issue d'une distribution de bruit.
4. On retourne la valeur aléatoire $Y = X_i + pB$

La fonction de densité de la distribution de B (c'est-à-dire la fonction qui permet de représenter la courbe de la distribution de bruit) [Density Function] est appelée noyau et est dénotée par $o(x)$. La fonction $o(x)$ doit être symétrique autour de l'origine. Pour déterminer le paramètre de lissage p, on utilise la formule suivante :

$p = \alpha(o)\ 1.364\ min(s, E/1.34)\ n^{-1/5}$ où l'on a

1. $\alpha(o) = 0.776$ si $o(x)$ est une fonction gaussienne (c'est-à-dire en forme de cloche telle une loi normale) [Gaussian Function] ou bien $\alpha(o) = 1.351$ si $o(x)$ est une fonction rectangulaire.
2. s l'écart type de l'échantillon.
3. E l'écart interquartile [Interquartile Range], c'est-à-dire l'écart entre le premier et troisième quartile de l'échantillon X_1, X_2, ..., X_n.

Selon [Hörmman et Bayar, 2000] la plupart du temps, l'algorithme KDE garantit que la fonction de densité de la distribution empirique approxime bien celle de la distribution réelle.

Toutefois, l'algorithme fait en sorte que la variance de la distribution empirique est plus grande que celle de l'échantillon. Ceci peut être désavantageux pour une simulation sensible aux changements de variance. Pour contrer cet effet, il existe une autre version de l'algorithme. Son acronyme est KDEVC pour « Kernel Density Estimation Variance Corrected » [Hörmman et Bayar, 2000]. Son objet dépasse cependant le cadre du présent mémoire.

3.3.3.3 Pourcentage d'efficacité

La seconde option possible consiste à recourir à un pourcentage d'efficacité des objets du système. On calcule d'abord la performance de chaque convoyeur et de chaque poste, sous des conditions optimales. Puis, on multiplie ce niveau de performance, par un pourcentage d'efficacité moyen, noté ϵ. Celui-ci peut être obtenu en se basant sur les précédents opérationnels du système.

Par exemple, à partir de relevés hebdomadaires, on peut remarquer qu'un vibrocompacteur est en moyenne arrêté dans 1.25% du temps. Son pourcentage d'efficacité ϵ est donc de 98.75%. Ceci implique que sur une semaine de d'opération de 40 heures, il faut simuler 30 minutes d'arrêt. De même, en se basant sur les précédents d'opération, on peut déterminer la répartition des arrêts durant une semaine.

À première vue, cette approche semble intégrer une certaine notion de probabilité. Ainsi, elle permet d'estimer le moment d'occurrence le plus vraisemblable, pour chaque *activité non-planifiée*, en se basant sur les précédents du système. Cependant, nous considérons la méthode du pourcentage d'efficacité comme déterministe, puisque les valeurs des variables aléatoires (T, S, C, A) sont toujours fixes. D'ailleurs, on ne doit plus parler de variable aléatoire, au sens statistique du terme [Kakmier, 1982], puisque la notion de hasard n'intervient plus.

Mentionnons que l'approche du pourcentage d'efficacité permet de prendre en compte l'effet global du hasard sur le système, sans les désavantages d'une approche purement probabiliste. En effet, la notion de probabilité fait en sorte que l'on simule des expériences aléatoires. Du même coup, il devient beaucoup plus difficile d'interpréter les résultats produits par le système, puisqu'on

ne peut pas nécessairement savoir quelle part est attribuable à sa nature probabiliste ou à son fonctionnement [Banks et al, 1998].

3.3.3.4 Moyenne échantillonnale

Cette approche s'apparente à la méthode du pourcentage d'efficacité. En fait, elle consiste à remplacer les valeurs des distributions de probabilité des variables aléatoires étudiées (T, S, C, A), par leurs moyennes échantillonnales respectives. Elle est utilisée lorsqu'on manque d'information pour estimer la forme de la loi de probabilité. Mentionnons que la méthode de la moyenne échantillonnale est déterministe, pour les mêmes raisons qui ont été évoquées plus haut, dans le cas du pourcentage d'efficacité.

Bien que fort simple, cette approche n'est pas dénuée de risque [Law et Kelton, 2000]. Par exemple, considérons une machine qui traite des entités quelconques et que son temps de service (S) est en moyenne de 0.99 minute. Supposons que le temps d'arrivé (A) entre deux entités soit en moyenne d'une minute. Alors, dans cette situation, les entités vont arriver à des intervalles de 1 minute, 2 minutes, ... et aucune d'entre elle ne sera jamais retardée. Or, cette situation est fort improbable, dès qu'on considère que l'écart type σ de A ou de S est moindrement important. Par conséquent, la moyenne échantillonnale doit être utilisée avec parcimonie, particulièrement si l'on constate qu'elle entraîne une perte de réalisme du modèle.

3.3.3.5 Approche minimaliste

La dernière option considérée, dite *approche minimaliste*, consiste à ignorer toutes les variables aléatoires du modèle. En fait, il s'agit presque d'une extension de l'expérience de simulation dite optimum, dont il est question à la section 4.4.2 et 4.4.3. Comme nous allons voir, celle-ci équivaut à négliger les *arrêts non-planifiés*, ce qui fait en sorte que tous les postes sont fiables à 100 %. L'approche minimaliste n'est pas souhaitable, parce qu'elle manque de flexibilité et qu'elle donne rarement un portait fidèle du système.

3.3.3.6 Approche retenue

Chacune des approches énumérées ont leurs avantages et leurs inconvénients. Toutefois, ce qui importe d'abord, c'est de choisir une méthode qui confère au modèle suffisamment de réalisme, sans complexifier sa conception et ajourner l'échéancier.

Un modèle probabiliste semble idéal d'un point de vue pragmatique. Par contre, il nécessite l'échantillonnage de presque tous les objets du modèle, afin d'obtenir des indicateurs tels la moyenne ou la variance de la *distribution de la moyenne échantillonnale*. Il implique aussi un effort d'analyse supplémentaire, pour déterminer la loi de probabilité de chaque variable (T, S, C, A) [Law et Kelton, 2000]. Finalement, comme nous avons dénoté à section 1.3, lorsque des résultats sont en fonction de variables aléatoires, il est difficile de savoir si les caractéristiques observées sont imputables aux interactions du modèle ou à sa nature probabiliste [Banks et al, 1998]. D'ailleurs, nous avons constaté que certains décideurs éprouvent une certaine méfiance, lorsqu'ils doivent prendre des décisions basées sur des informations non-reproductibles. De même, lors d'une session de travail qui doit permettre de répondre aux interrogations de décideurs, il est plus facile de simuler une expérience déterministe, en faisant varier différents paramètres et d'expliquer l'impact sur les mesures de performances.

À l'inverse, l'approche de modélisation axée sur la moyenne échantillonnale, nous apparaît beaucoup plus simple. En effet, elle permet d'éviter l'étape d'estimation des distributions de probabilité et l'intégration d'une librairie statistique. Toutefois, la méthode de la moyenne échantillonnale n'est pas sans danger, comme on l'a mentionné à la section 3.3.3.4. Malgré ces désavantages, nous avons choisi d'utiliser en partie l'approche de la moyenne échantillonnale. Notamment parce que la reproductibilité des résultats était souhaitée par les spécialistes du système, afin de mieux appréhender les interactions du modèle. Les valeurs des variables S (« durée de *Service* d'une ressource ») et T (« durée du *Transit* d'une file ») correspondent donc à leur moyenne échantillonnale. En ce qui concerne les valeurs des variables C (« temps de *Commencement* d'une activité *non-planifiée* ») et A (« durée d'une *Activité non-planifiée* »), nous

les avons obtenues à partir d'un ensemble d'observations. Elles ont été inscrites dans le fichier d'initialisation du modèle, pour fin de configuration. Puisque les résultats sont reproductibles à chaque exécution et que la notion de probabilité n'intervient pas, le modèle est donc considéré comme déterministe.

3.3.4 Gestion des événements discrets

À la section 1.5, nous avons mentionné qu'un modèle dynamique est considéré comme discret si ses variables d'état changent à des instants précis. Inversement, un modèle dynamique est dit continu, lorsque ses variables sont modifiées de façon ininterrompue à travers le temps [Hoover et Perry, 1990]. Notre modèle est discret, puisque les événements qu'il génère surviennent à des moments distincts. D'ailleurs, il peut se produire un intervalle de temps assez important entre deux événements. Ainsi, la notion de continuité n'est pas envisageable. Cette partie du mémoire vise à présenter la stratégie d'intégration des événements discrets. Plus précisément, nous allons discuter de la gestion du temps et de l'approche de conception du contrôleur du modèle.

3.3.4.1 Gestion du temps du modèle à événements discrets

La gestion du temps est nécessaire pour d'une part, affecter une valeur à l'horloge de la simulation et d'autre part, définir le mécanisme qui régit son avancement. Comme nous avons vu à la section 1.6.2, il existe deux stratégies d'avancement : *incrément fixe* et *événement suivant* [Law et Kelton, 2000]. Dans le cadre de notre modèle, la stratégie retenue est la méthode avec *incrément fixe*. Nous avons donc divisé la simulation en pas de temps (Δt) de 5 secondes. À chaque pas, des vérifications sont faites, pour statuer si des événements se produisent.

La méthode avec *incrément fixe* met en évidence l'évolution de la simulation. De plus, bien que le temps entre les événements ne soit pas nécessairement régulier, nous avons quand même opté pour cette méthode, en raison de sa simplicité. En effet, une approche avec *incrément fixe* permet de traiter facilement les événements *conditionnels*, autrement dit de type *C* (pour « *Cooperative* » ou « *Conditional* »). D'ailleurs, notre modèle incorpore plusieurs de ces

événements. Or, il s'avère plus difficile de mettre en œuvre des événements *conditionnels*, avec une méthode de type *événement suivant*. En contrepartie, les événements planifiables peuvent être adaptés à l'un ou l'autre des mécanismes d'avancement du temps [Pidd, 1998].

Afin d'illustrer le point précédent, considérons l'événement qui initie l'activité de récupération des anodes de l'entrepôt. Cette activité est notée *récupérer anode cuite* (*id=11*) sur le diagramme de la Figure 23. Normalement, l'événement se produit lorsque l'atelier de scellement (*id=12*) nécessite des anodes cuites. Cet événement est quant à lui déclenché lorsque les fours produisent peu d'anodes. Ceci peut entre autre se produire, en raison d'un *arrêt planifié* ou *non-planifié* d'un poste en amont. Par ailleurs, la grue de l'entrepôt ne peut entamer la récupération que lorsque le convoyeur adjacent a suffisamment de place. Comme on peut le constater, l'activité *récupérer anode cuite* peut difficilement être planifiée à l'avance, car elle dépend d'un amalgame d'événements circonstanciels.

Le mécanisme d'avancement avec *incrément fixe* peut avoir un impact sur la performance du modèle, puisqu'à chaque pas de temps, il faut vérifier si un événement risque de se produire [Hoover et Perry, 1990]. Par contre, comme nous allons le voir un peu plus loin, sous certaines conditions, il est possible d'éviter une altération drastique des performances, en optimisant l'algorithme de vérification.

3.3.4.2 Approche de conception du contrôleur

Tel que mentionné à la section 1.6.3, il existe quatre approches de conception du contrôleur d'un modèle à événements discrets : *planification d'événements*, *recherche d'activités*, *méthode basée sur des processus* et *méthode en trois phases* [Balci, 1988] [Pidd, 1998]. Ces approches ont un impact au niveau de la mise en œuvre du contrôleur. L'approche retenue s'apparente à la *recherche d'activités*. Nous allons donc brièvement rappeler les fondements de ces approches et discuter des raisons qui nous ont amené à notre choix.

La *planification d'événements* fait en sorte que l'on maintient une *liste d'événements* futurs. Chaque élément de la liste référence une *routine événementielle* appropriée [Banks et al, 1998].

Cette méthode s'agence naturellement avec un mécanisme d'avancement du temps de type *événement suivant*. Cependant, comme mentionné à la section 3.3.4.1, certains événements de notre modèle sont difficiles à planifier. C'est pourquoi l'approche *planification d'événements* n'a pas été retenue.

La seconde approche, appelée *recherche d'activités*, considère que tous les événements sont conditionnels à des prémisses. Si une condition est satisfaite, l'activité qui lui est associée est exécutée [Balci, 1988]. Nous avons opté pour cette méthode, car elle permettait de représenter facilement les deux types d'événements : *planifiables* (type *B*) et *conditionnels* (type *C*). Les événements *planifiables* sont adjoints à une *liste d'événements*. De plus, ils sont traités comme des événements *conditionnels,* puisqu'on les apparie avec une prémisse. Par exemple, lorsqu'un événement *e* est planifié au temps *t*, on crée une condition qui ressemble à ce qui suit :

```
si
      e se produit à t et
      temps de l'horloge est t
alors
     exécuter e
```

Le fonctionnement de cette méthode s'apparente à celui d'un *système expert à base de règles*, puisque le déclenchement d'une activité est conditionnel à la satisfaction des prémisses [Banks et al, 1998]. Comme on l'a mentionné, la *recherche d'activités* est moins performante que d'autres méthodes, parce qu'il faut vérifier toutes les prémisses d'événements, afin de déterminer celles qui sont satisfaites [Pidd, 1998]. Or, les *systèmes de productions*, basés sur l'algorithme de Rete [Forgy, 1982], permettent d'adresser une telle problématique, en limitant le nombre d'assortiments entre les prémisses de règles et les informations factuelles [Giarratano et Riley, 1998]. L'approche *recherche d'activités* est donc plus efficace, lorsqu'on lui adjoint un *système de production*, pour définir les événements déclenchés [Jeong, 2000].

La *méthode basée sur des processus* implique que le contrôleur fait progresser chaque entité à travers le processus de sa classe. Cette progression peut être interrompue par des délais *conditionnels* ou *inconditionnels*, aussi qualifiés de *planifiables* [Pidd, 1998]. Quant à la *méthode en*

trois phases, rappelons qu'elle consiste à d'abord rechercher le prochain événement à travers une liste d'*événements planifiables* et à l'ajouter à la *liste des entités échues,* lorsqu'il se produit au temps actuel de simulation. La seconde étape implique l'exécution des événements de la liste des *entités échues*. La dernière étape nécessite la vérification des événements conditionnels et leur exécution, lorsque leurs prémisses sont satisfaites [Banks et al, 1998].

Ces deux dernières méthodes semblent intéressantes, car elles différencient clairement les deux types d'événements : *conditionnels* et *planifiables* (*inconditionnels*). De plus, comme on l'a mentionné à la section 1.6.3.4, la *méthode en trois phases* permet de diminuer le risque d'interblocage. Par contre, ces méthodes obligent de maintenir deux listes d'événements. La mécanique de leur contrôleur est donc plus délicate à mettre en œuvre que celle de la *recherche d'activités*.

3.4 Conception et mise en œuvre du modèle

Comme notre modèle a été mis en œuvre sous la forme d'un programme informatique, nous allons maintenant présenter la phase de conception, aussi appelée architecture. L'architecture réfère aux contraintes qui s'appliquent à tout un système [McConnell, 2004]. Par exemple, on peut penser au paradigme de programmation (structuré ou orienté objet), aux structures de données, aux algorithmes, aux classes, aux modules, aux paquetages, à la hiérarchie, aux relations, aux langages, aux entrées/sorites, à la performance, à la réutilisabilité, à l'organisation des sous-systèmes et aux outils de conception [McConnell, 1996].

Nous allons d'abord présenter le diagramme de classes du modèle. Ensuite, nous discutons aussi de l'intégration d'un système expert au sein du modèle. Plus particulièrement, nous traitons de la représentation de l'expertise à l'aide de règles, du mécanisme d'inférence, du rôle d'un système expert élémentaire et de la coquille de système expert Jess. L'Annexe 3 décrit également d'autres considérations conceptuelles qui sont intervenues. Il est notamment question du choix du langage Java et des outils que nous avons employés.

3.4.1 Modèle UML

Un modèle UML est « une abstraction sémantiquement fermée d'un système », s'appuyant sur le langage UML. NDLR : traduit de l'Anglais [Priestley, 2000]. Nous nous intéressons aux modèles, dans un contexte de prise de décision appliquée à un système précis, soit : *la manutention et l'entreposage des anodes*. C'est pourquoi nous avons mis en œuvre le modèle UML, sous la forme d'un programme Java, pour pouvoir effectuer des simulations qui soutiennent le processus décisionnel.

Cette section présente les diagrammes de classe du modèle UML [Fowler, 2004]. Les classes ont été regroupées en paquetages ou paquets (de l'Anglais « packages »). Un paquetage est défini comme suit : « En programmation, conteneur qui, au moyen d'un mot-clé commun, regroupe des classes partageant des relations logiques. » [Dictionnaire terminologique] Les principaux paquetages du modèle sont : connaissance (*knowledge*), interface usager (gui), procédé (*process*) et simulation (*simulation*).

3.4.1.1 Paquetage connaissance

Une instance unique d'*ExpertSystem* (voir le paquetage *knowledge* à la Figure 25) est créée et manipulée à partir de *Simulation* (paquetage *simulation*). La classe *ExpertSystem* hérite de la classe *Rete* (paquetage *jess*). *ExpertSystem* est donc une extension de la coquille de système expert Jess. La classe *ExpertSystem* crée et manipule une instance unique de *RulesBase* (la base de connaissances) et de *FactsBase* (mémoire de travail). L'instance de *FactsBase* référence les classes du procédé (paquetage *process*). Ces classes sont : *EntitiesQueue*, *EntityState*, *EventDescription*, *ProcessedEntity*, *RessourceAncestor*, *Schedule* et *ScheduledEvent*. La classe *FactsBase* fait appel aux méthodes de *FactsParser* pour lire le fichier inital des faits et créer toutes les instances de classes du paquetage *process*. Puisque chacune de ces instances sont des faits, la méthode *assertAll* (classe *FactsBase*) les affirme dans la mémoire de travail de Jess. La méthode *defineAll* (classe *RulesBase*) lit le fichier texte des règles et les définit dans la base de

connaissances de Jess. La méthode *run* (classe *ExpertSystem*) déclenche le moteur d'inférence. Celui-ci est exécuté jusqu'à ce qu'il n'y ait plus d'activation [Jess 7.0 Manual].

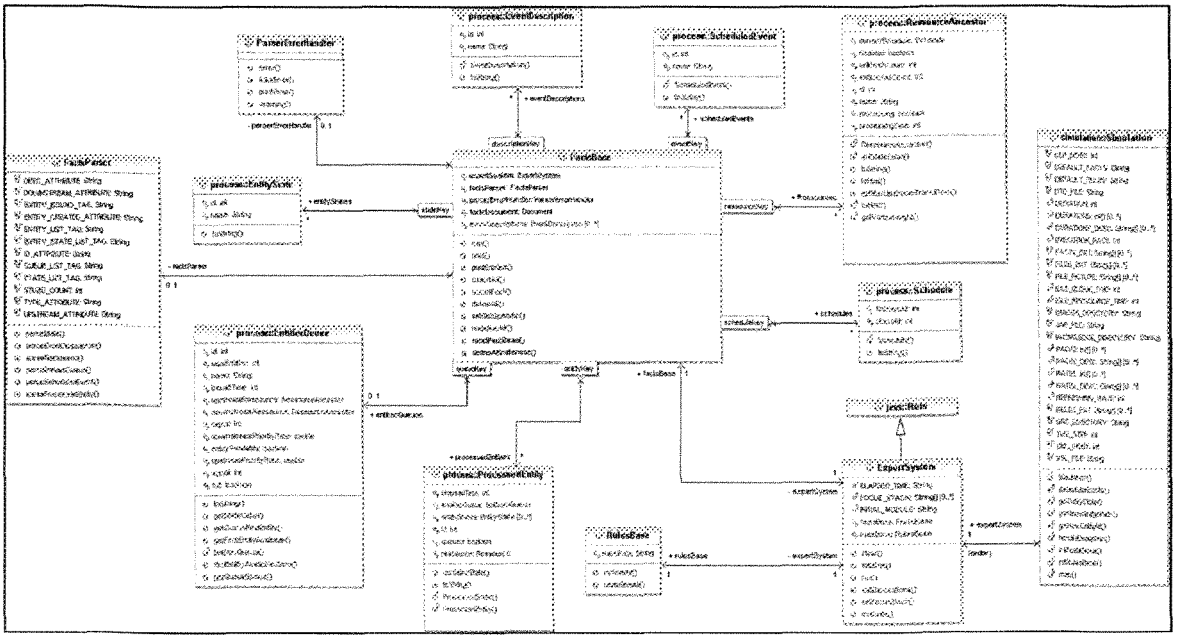

Figure 25 : classes du paquetage connaissance

3.4.1.2 Paquetage interface graphique

Comme nous allons voir à la section 4.2, l'interface graphique du modèle est un graphe dirigé [Cormen et al., 1994]. Les deux principales classes du paquetage sont *GuiGraphEdge* (les arêtes du graphe) et *GuiGraphVertex* (les sommets du graphe). Une instance de *GuiGraphVertex* (une arête) est associée à une instance d'*EntitiesQueue* (paquetage *process*), c'est-à-dire un convoyeur. Une instance de *GuiGraphVertex* (un sommet) est adjointe à une instance de *RessourceAncestor* (paquetage *process*), c'est-à-dire une ressource. Lorsque *FactsParser* (paquetage *knowledge* : voir la section 3.4.1.1) crée les instances d'*EntitiesQueue* et de *RessourceAncestor*, il crée aussi celles de *GuiGraphEdge* et de *GuiGraphVertex* qui leur sont associées. Chacune de ces instances sont ajoutées à la liste des arêtes (c'est-à-dire *guiGraphEdges* : une propriété publique/statique de *GuiGraphEdge*) et à celle des sommets (c'est-à-dire *guiGraphVertexes* : une propriété publique/statique de *GuiGraphVertex*). Avant que la

simulation ne débute, la classe *Simulator* (paquetage *simulation*) lit chacune des listes pour insérer les arêtes (*GuiGraphEdge*) et les sommets (*GuiGraphVertex*) au sein du graphe de l'interface. À chaque fois que celle-ci doit être mise à jour, *Simulator* parcourt la liste des arêtes et celle des sommets pour faire apparaître les changements d'états des convoyeurs (*EntitiesQueue*) et des ressources (*RessourceAncestor*).

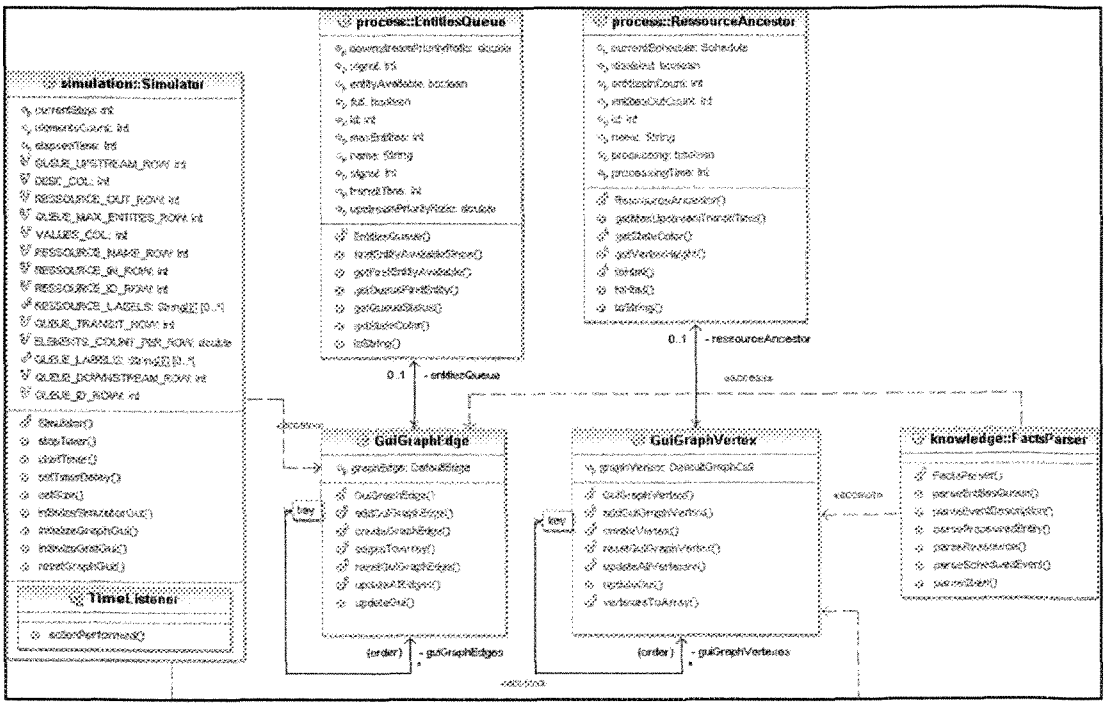

Figure 26 : classes du paquetage interface graphique

3.4.1.3 Paquetage procédé

La classe abstraite *RessourceAncestor* représente les ressources du système. Il existe deux catégories de ressources. D'abord, il y a les ressources standards qui traitent les anodes une à une (classe *Ressource*), comme les fours (classe *Furnace*), ainsi que les tables de transferts et l'atelier de scellement (classe *Station*). La classe *Ressource* est donc associée (voir l'association *processedEntity* [0..1]) à *ProcessedEntity*, à savoir la classe des entités traitées (anodes). Ensuite, il y a les ressources qui traitent plusieurs anodes à la fois, c'est le cas des grues (classe *Crane*).

Comme nous l'avons mentionné, elles ont une capacité d'entreposage et de récupération de 12 anodes. Chaque grue (classe *Crane*) réfère à l'entrepôt (classe *Warehouse*). Le nombre d'anodes cuites et crues de l'entrepôt change selon les signaux traités par les grues.

La classe *ScheduledEvent* symbolise les événements du procédé, mentionnons : les arrêts planifiés, les arrêts non-planifiés, les arrivées de train, etc. Chaque instance de classe *Schedule* est associée à une instance de *ScheduledEvent* (événement) et de *RessourceAncestor* (ressource). Ainsi, on peut connaître la planification de chaque ressource (*RessourceAncestor*), c'est-à-dire le moment où débute et se termine un événement (*ScheduledEvent*).

La classe *EntitiesQueue* représente une file (convoyeur) d'entités traitées. Elle est donc associée à *ProcessedEntity* (voir association *processedEntities* [0..*]). Chaque file (*EntitiesQueue*) est liée à deux ressources (*RessourceAncestor*), l'une en amont (association *downstreamRessource* [1]) et l'autre en aval (association *upstreamRessource* [1]). Chaque ressource (*RessourceAncestor*) a zéro ou plusieurs files en amont (association *downstreamQueue* [0..*]) et en aval (association *upstreamQueue* [0..*]).

La classe *ProcessedEntity* représente les entités traitées. Il y a trois types d'entités : les anodes (*Anode*), les ensembles anodiques (*AnodicSet*) et les tiges (*Rod*). Un ensemble anodique (*AnodicSet*) est composé d'une anode (*Anode*) et d'une tige (*Rod*). Une entité traitée (*ProcessedEntity*) a trois possibilités : être sur un convoyeur (association *entitiesQueue* [1] avec *EntitiesQueue*), être traitée par une ressource standard (association *ressource* [1] avec *Ressource*) ou encore être traitée par une grue (*Crane*). Dans ce dernier cas, il n'y a pas d'association entre les deux classes, puisque les entités (*ProcessedEntity*) sont prélevées du convoyeur (*EntitiesQueue*) que lorsque le temps de récupération de la grue (*Crane*) est atteint.

Finalement, la classe *EntityState* symbolise l'ensemble des états que peuvent avoir une entité traitée (voir l'association *entityStates* [0..*] entre *ProcessedEntity* et *EntityState*). Parmi les états d'entités mentionnons : *chaude*, *compactée*, *crue*, *cuite*, *droite*, *froide*, *pressée*, *recourbée*, etc. L'association *outputStates* [0..*], avec *RessourceAncestor*, représente l'état qu'ont les entités

sorties d'une ressource (*RessourceAncestor*). Par exemple, les anodes issues d'un four sont *cuites* et *chaudes*.

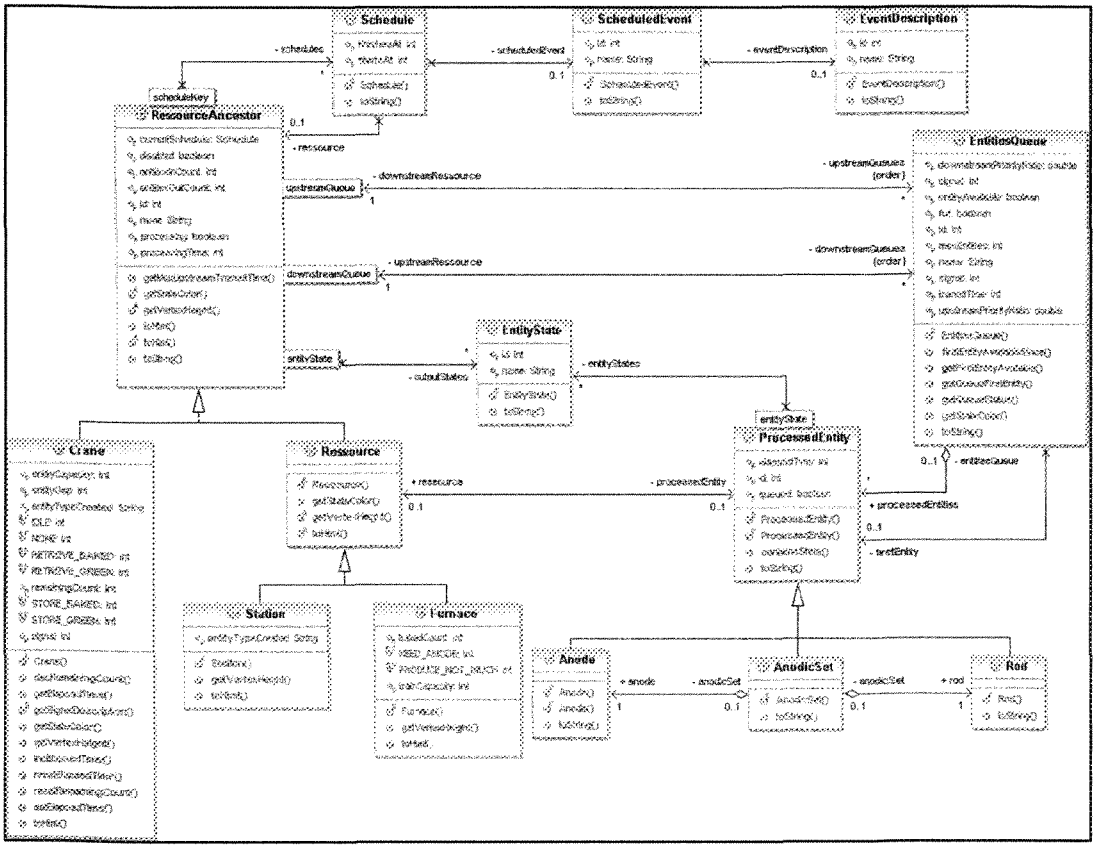

Figure 27 : classes du paquetage procédé

3.4.1.4 Paquetage simulation

La classe *Simulation* est le point d'entré du programme (méthode *main*). Elle permet entre autre, de créer et d'afficher la fenêtre principale (*MainFrame*). L'usager spécifie les paramètres initiaux de la simulation à partir de cette fenêtre, mentionnons : les chemins des fichiers de règles et de faits, la durée de la simulation, le rythme d'exécution, ainsi que le taux de rafraîchissement de l'interface. L'usager clique sur le bouton *Initialiser* de *MainFrame*. Les fonctions publiques/statiques *initFactsBase* et *initRulesBase* de *Simulation* sont alors appelées. Ceci a pour effet de créer les objets *ExpertSystem, FactsBase* et *RulesBase* (paquetage *knowledge*).

L'usager clique sur le bouton *Simuler* de la classe *MainFrame*. La fenêtre de dialogue *Simulator* est alors créée et affichée. Celle-ci correspond à l'interface graphique de la simulation. *Simulator* possède une minuterie dont l'écouteur est déclenché à un intervalle de temps fixe. Cet intervalle correspond au rythme de simulation. Lors de l'appel de l'écouteur, le moteur d'inférence est lancé (méthode *run* d'*ExpertSystem*) et l'interface est mise à jour au besoin (méthode *updateGui* de *Simulator*).

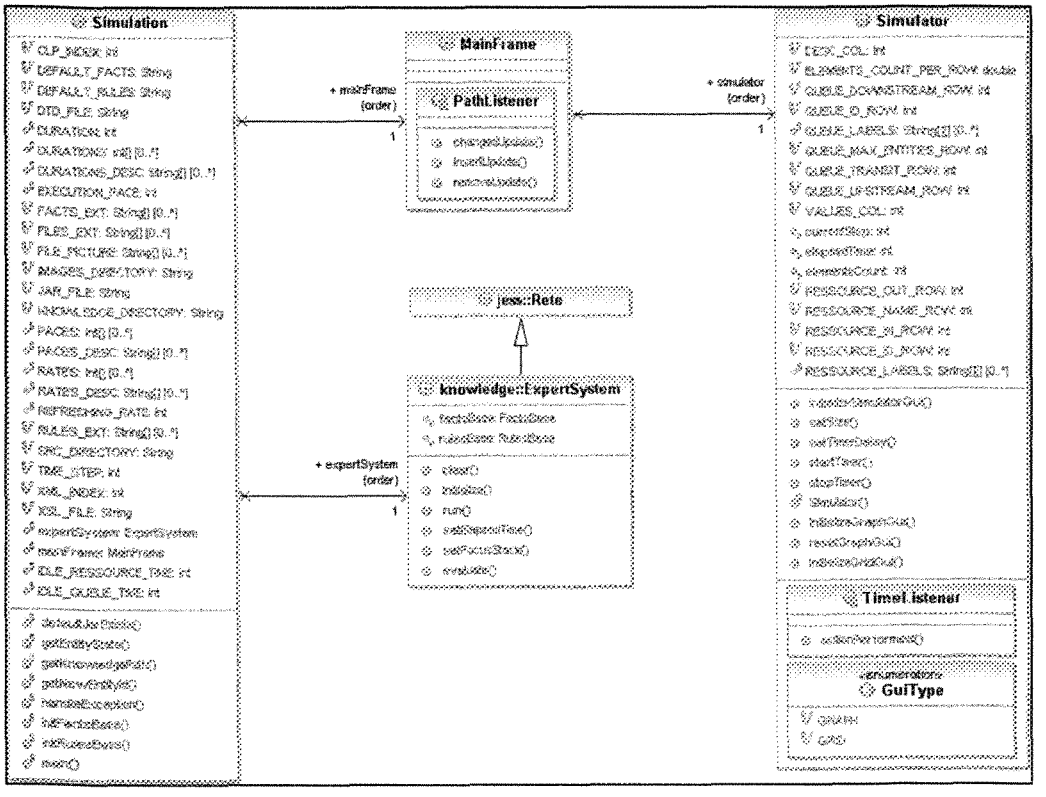

Figure 28 : classes du paquetage simulation

3.4.2 Intégration d'un système expert

Tel qu'il a été mentionné aux sections 3.3.4.1, les activités du système étudié sont difficilement planifiables à l'avance, car elles dépendent d'un ensemble d'événements conditionnels interdépendants. C'est pourquoi nous avons opté pour une gestion de l'horloge de type *incrément fixe* [Law et Kelton, 2000] et une approche de conception avec *recherche d'activités*, à chaque pas de temps Δt. Conséquemment, compte tenu de la nature conditionnelle de plusieurs événements

du système et d'une perte de performance du modèle, occasionnée par la *recherche d'activités* [Pidd, 1998], l'intégration d'un système expert devient une avenue intéressante. En effet, un système de production augmente la performance, grâce à l'algorithme de Rete [Forgy, 1982]. Celui-ci limite le nombre d'appariement et de vérification, durant de la phase *recherche d'activités* [Jeong, 2000]. De plus, un système expert facilite la mise en œuvre des événements conditionnels, à partir des prémisses des règles de production.

Normalement, un système à base de règles est envisageable lorsque certaines conditions sont rencontrées. Ci-dessous se trouve quelques-unes des conditions recommandées par [Rudolph, 2003] :

1. Si la logique du domaine implique plusieurs conditions de branchement ou de prise de décision, alors un système à base de règles peut être considéré. Dans notre cas, cette ligne de conduite est respectée, puisqu'un grand nombre d'événements du modèle dépendent d'une suite de conditions (voir Annexe 4).
2. S'il est possible d'écrire des règles, à partir des conditions de branchement et qu'elles comportent au moins trois prémisses, alors un système à base de règles peut être envisagé. On préconise aussi son usage, lorsqu'il y a au moins trois niveaux d'imbrication de règles. Une bonne façon de vérifier le nombre de niveaux d'imbrication est le *réseau d'inférence* du domaine [Gonzalez et Dankel, 1993], tel que décrit à la section 2.5.2. Dans le cadre de notre modèle, près de la moitié des règles ont plus de deux prémisses (voir Annexe 4) et il y a six niveaux d'imbrication (comme nous allons le voir à la section 3.4.4.6).
3. Lorsque les règles doivent être flexibles et qu'elles risquent de changer à travers le temps, un système de production est recommandé. Les règles de notre modèle demeurent plus ou moins statiques, une fois que celui-ci a été conçu. Cependant, comme au départ certaines d'entre elles étaient inconnues, un système de production est idéal, pour encoder et modifier les règles, à mesure qu'elles sont découvertes.

4. Si le programme doit être maintenu, une fois qu'il a été livré, alors un système de production est souhaitable. Comme on l'a vu à la section 1.4, le modèle conceptuel est basé sur des hypothèses [Law et Kelton, 2000]. Le système à base de règle est donc utile, puisqu'une fois le modèle livré, certaines hypothèses peuvent être invalides, ce qui implique d'apporter des changements à sa logique.

Maintenant que les avantages et les conditions d'intégration d'un système de production ont été introduits, nous allons discuter de la représentation de la connaissance, à partir de règles et du mécanisme d'inférence sélectionné.

3.4.2.1 Représentation de la connaissance à partir de règles

Tel que nous avons mentionné à la section 2.5.1, les principaux formalismes de représentation de la connaissance sont : la *logique prédicative*, les *réseaux sémantiques*, les *cadres* et les *règles de production* [Gonzalez et Dankel, 1993]. Notre choix s'est arrêté sur la dernière forme de représentation. Cette section énonce les raisons qui justifient leur usage, dans un contexte de modélisation et de simulation d'un système.

Le système étudié s'appuie sur des procédures opérationnelles. Celles-ci ont été décrites à la section 3.3.2. Ces procédures constituent un ensemble de règles empiriques (de l'Anglais « rule of thumb ») reconnues par les spécialistes des étapes de *formation, manutention et entreposage des anodes*. Une règle empirique est définie comme suit :

> Une méthode, une procédure ou une analyse, basée sur l'expérience et la connaissance, qui a pour but de fournir une solution approximative et généralement valide, d'une problématique donnée. NDLR traduction de l'Anglais « rule of thumb » [Dictionnaire terminologique]

Ces méthodes empiriques sont soient appliquées par les opérateurs du système ou programmées au sein d'automates. Par exemple, on peut penser aux signaux qui gèrent les grues gerbeuses de l'entrepôt [Poirier, 1997]. Or, des études en intelligence artificielle, effectuées durant les années cinquante et soixante, ont démontrées que la capacité de résolution des humains pouvait être exprimée par des règles de production de type *si ... alors* [Newell et Simon, 1972].

Nous nous sommes donc intéressés à ce formalisme, afin de représenter les règles empiriques, issues de l'expérience et de la connaissance des spécialistes du système.

Les règles de production équivalent à des unités de connaissance modulaires en interrelation [Giarratano et Riley, 1998]. La modularité promeut la flexibilité du modèle, puisqu'il devient facile de changer sa logique [Pidd, 1998], par l'ajout, la modification ou la suppression de règles. De plus, comme elles agissent en tant qu'unités autonomes, les impacts de ces changements sur les autres règles sont minimisés [Rudolph, 2003]. La modularité est d'autant plus souhaitable, car en temps normal, pour s'assurer de la crédibilité du modèle, les hypothèses de base sont graduellement raffinées et validées [Banks et al, 1998] [Law et Kelton, 2000].

Puisque les règles sont reliées entre elles [Giarratano et Riley, 1998], il devient possible de modéliser les interactions entre les ressources ou les entités du système. Ces interrelations entre les règles permettent de connaître le raisonnement qui mène à la solution du système expert, en suivant leur séquence de déclenchement [Gonzalez et Dankel, 1993]. Ce raisonnement peut ensuite servir à produire des rapports explicatifs, afin de supporter un gestionnaire, dans un contexte décisionnel.

Finalement, afin d'illustrer la force des règles de production, considérons l'approche de modélisation la plus souvent utilisée, soit celle avec *planification d'événements* (voir 1.6.3.1). L'une de ses principales difficultés consiste à définir les conséquences de chaque routine événementielle [Pidd, 1998]. Pour ce faire, on met en œuvre le modèle, comme un algorithme séquentiel, afin de décider quels sont les actions intégrées aux routines événementielles et leur séquence d'exécution. Le système à base de règles simplifie cette tâche. En effet, celui-ci permet d'omettre certains détails de la mise en œuvre du modèle, puisque le moteur d'inférence spécifie la séquence de déclenchement des règles. Ce type de programme, où un système en cours d'exécution planifie et contrôle le flot de décision, est dit *déclaratif* [Friedman-Hill, 2003]. À ce titre, la section suivante présente le type de raisonnement du moteur d'inférence, ainsi que les raisons de ce choix.

3.4.2.2 Moteur d'inférence avec *chaînage avant*

Nous avons opté pour un type de raisonnement avec *chaînage avant*. Ce choix nous est apparu le plus logique, puisque comme nous avons mentionné à la section 2.7.1, il convient aux problèmes de simulation ou de planification. Un raisonnement avec *chaînage arrière* eut été souhaitable pour un problème de diagnostic [Giarratano et Riley, 1998]. D'ailleurs, avec le *chaînage arrière*, l'utilisateur interagit parfois avec la base de connaissances, en saisissant les faits manquants. Or, ce n'est pas ce que nous voulons.

Le *chaînage avant* est recommandé lorsque les faits sont limités et que la plupart d'entre eux contribuent au processus de résolution. Cette méthode d'inférence convient également si la problématique comporte plusieurs solutions. À l'inverse, le *chaînage arrière* est souhaitable lorsque les faits initiaux sont beaucoup plus nombreux que les conclusions [Gonzalez et Dankel, 1993]. Le *chaînage avant* constitue donc un choix judicieux, car les faits à modéliser se limitent aux variables d'états, aux ressources et aux entités du système. Également, le nombre de solutions, c'est-à-dire l'ensemble des états terminaux du système, est quasi illimité, en considérant que chaque variable d'état peut prendre plusieurs valeurs.

Rappelons qu'une méthode d'inférence avec *chaînage avant* facilite la recherche en largeur (de l'Anglais « breadth search ») [Shaffer, 1997] [Cormen et al., 1994]. Idéalement, le réseau d'inférence du problème devrait donc être large et peu profond [Giarratano et Riley, 1998]. Ce critère est rencontré, puisque comme nous allons le voir à la section 3.4.4.6, le réseau d'inférence du système a un maximum de six niveaux, ce qui est peu profond. De plus, même si certains niveaux du réseau, tel le premier, ont seulement quelques règles, d'autres, tel le cinquième, en ont une dizaine. Ceci implique plusieurs conditions de branchement et un réseau d'inférence assez large.

3.4.3 Intégration d'un système expert élémentaire

Dans le cadre de la mise en œuvre de notre modèle, nous nous sommes questionnés sur la pertinence d'intégrer un système expert complet. Ainsi, l'ajout d'une coquille de développement,

comme Jess, nous est d'abord apparu un peu trop complexe, par rapport aux avantages qu'elle pouvait procurer, puisque d'une part, les possibilités d'intégration sont multiples et d'autre part, les fonctionnalités de son API sont forts étendues [Jess 7.0 Manual]. Nous avons donc décidé de concevoir un premier prototype de notre modèle, en intégrant un système expert élémentaire. Les sections qui suivent le décrivent brièvement.

3.4.3.1 Base de connaissances

Les règles de production sont encodées au sein du code source Java. Ces règles sont en fait une suite de conditions séquentielles, de type **`if`** `<prémisse> {...}`. Mentionnons que ce système expert élémentaire ne constitue certes pas une coquille de développement comme Jess. En effet, puisque les règles sont statiques, la base de connaissances est dédiée à un domaine particulier [Giarratano et Riley, 1998], à savoir la *formation, la manutention et l'entreposage des anodes*.

3.4.3.2 Mémoire de travail

En ce qui concerne la mémoire de travail, elle est formée de structures de données, comme des tables de hachages ou des files [Cormen et al., 1994]. Les éléments de ces structures sont des faits. Chaque type de fait est explicitement associé à une structure de donnée. Un fait symbolise une information qui est référencée par des règles de production en Java, par exemple : l'état d'une ressource ou d'une file. Les valeurs de chacun des faits sont extraites à partir d'attributs de classes du modèle. Mentionnons les classes symbolisant les anodes, les ressources, les files, etc. Conséquemment, il importe d'effectuer la synchronisation entre les faits de la mémoire de travail et les propriétés des instances de classes.

3.4.3.3 Mécanisme d'inférence

La mise en œuvre du modèle débute par la lecture du nombre de pas de temps, c'est-à-dire la durée de la simulation (voir l'étape 1 de l'algorithme ci-dessous). Les faits, issus des propriétés d'instances de classes, sont ensuite ajoutés aux structures de données de la base de

connaissances (2.1). Puis, on effectue une boucle, dont le nombre d'itération est une valeur arbitraire (2.2). Une itération constitue un cycle d'inférence [Friedman-Hill, 2003]. Durant un cycle, on tente d'assortir les règles Java, avec les faits contenus dans les structures de données (2.2.1). S'il est possible de les associer, les règles sont activées et des faits additionnels sont déduits (2.2.2). Ils viennent s'ajouter à ceux déjà contenus dans les structures de données de la mémoire de travail (2.2.3). Finalement, on traite chacune des ressources, afin de voir l'impact des faits déduits sur l'ensemble du système (2.3). Ce processus est répété, jusqu'à ce que le temps de simulation soit entièrement écoulé. L'algorithme qui vient d'être décrit est le suivant :

```
1. Lire le nombre de pas de temps (n)
2. Itérer de 1 à n pas de temps :
   2.1.  Ajouter les faits issus d'instances de classes dans les
       structures de données
   2.2.  Tant que la condition d'arrêt n'est pas satisfaite faire :
      2.2.1.   Assortir les règles encodées avec les faits des structures
          de données
      2.2.2.   Exécuter les règles actives pour déduire de nouveaux faits
      2.2.3.   Ajouter les faits déduits dans les structures de données
   2.3.  Traiter chacune des ressources pour voir l'impact des faits
       déduits sur le procédé
   2.4.  Modifier l'interface usager
```

Le pseudo-code précédent s'apparente quelque peu à la solution naïve de l'algorithme avec *chaînage avant*, tel que décrit à la section 2.7.1. Toutefois, un constat s'impose, à savoir que l'ordre de déclenchement des règles actives, voir l'étape 2.2.2 du pseudo-code, demeure toujours le même, puisqu'elles sont exécutées séquentiellement, à chaque cycle d'inférence. Ceci fait en sorte que nous ne pouvons pas exploiter les avantages d'un programme déclaratif, comme nous avons mentionné à la section 3.4.2.1 [Friedman-Hill, 2003]. D'autre part, puisque les règles sont encodées au sein même du modèle, celui-ci risque d'être beaucoup moins flexible et plus difficile à maintenir [Rudolph, 2003].

3.4.4 Intégration de Jess

Suite aux vérifications qui ont été faites avec le premier prototype, nous avons décidé de mettre en œuvre un second prototype. Celui-ci intègre la coquille de développement de système

expert Jess, acronyme de : « Java Expert System Shell » [Jess]. Comme nous avons mentionné à la section 2.3, une coquille de développement (traduction de l'Anglais « Expert System Shell ») est un outil qui facilite la conception d'un système expert. En fait, une coquille de développement possède toutes les caractéristiques d'un système expert traditionnel, à l'exception de sa base de connaissances qui doit être fournie par l'usager [Giarratano et Riley, 1998]. Jess a été conçu en 1995, par le Dr. Ernest Friedman-Hill, du Sandia National Labs à Livermore en Californie. Au moment de l'écriture du mémoire, la dernière version stable est Jess 7.0. Il est possible de se procurer une licence académique gratuite de la coquille.

3.4.4.1 Stratégies d'intégration possibles

Jess est une librairie formée d'un ensemble de paquetages Java. Cette librairie fournit une interface de programmation (API), afin d'appeler les fonctionnalités du système expert. La coquille peut donc être facilement intégrée à n'importe quel type d'application, par exemple : une application Java légère, riche ou un simple applet [Friedman-Hill, 2003]. En utilisant Jess, on peut concevoir des programmes capables de « raisonnement », grâce à la connaissance déclarative [Turban et Aronson, 2000], représentée par des règles de production [Jess]. Le moteur d'inférence de Jess est basé sur l'algorithme de Rete [Forgy, 1982]. Il a été question de celui-ci à la section 2.8. Les faits et les règles de la coquille sont généralement encodés dans la syntaxe Jess. Cette syntaxe s'apparente à celle de la coquille CLIPS [Riley et al., 2005] qui est elle-même basée sur le langage LISP [Anderson et al, 1987].

L'intégration avec Java peut se faire dans les deux sens. Ainsi, un programme Java peut appeler l'ensemble de l'API de la libraire Jess. À l'inverse, toutes les possibilités de Java sont accessibles, à partir d'un programme Jess [Friedman-Hill, 2003]. Voici quelques possibilités d'interaction, entre la coquille et Java [Jess 7.0 Manual] :

- Il est possible de créer des objets de l'API Java ® 2 [Java 2 SE v.1.4.2 API], à partir de la syntaxe Jess et de référencer leurs méthodes, pour entre autre, lire ou manipuler leurs propriétés.

- Comme nous avons mentionné à la section 2.6.2, la mémoire de travail de la coquille est formée d'un ensemble de faits *non-ordonnés* ou *ordonnés*, dont les valeurs sont des types élémentaires [Jess 7.0 Manual]. Cependant, Jess peut aussi connecter des objets JavaBeans (voir la section « Composant JavaBeans » de l'Annexe 3) à sa mémoire de travail. De cette façon, un Bean est perçu comme un fait *non-ordonné*, dont les propriétés agissent comme des *fentes* (« Slots »). Celles-ci peuvent ainsi être appariées aux prémisses de règles. Un fait associé à un Bean est appelé un « shadow fact » [Friedman-Hill, 2003]. Ce concept est décrit de façon plus détaillée à la section 3.4.4.5.
- Il est aussi possible d'ajouter de nouvelles commandes au langage Jess. Pour ce faire, il suffit de définir des fonctions Java, de les regrouper et de les charger au sein de la coquille Jess. L'ajout de commandes peut s'avérer nécessaire, puisque dans certains cas, le code source en Jess devient trop difficile à lire, comparativement à Java. De plus, des fonctionnalités, telles la manipulation de tableaux multidimensionnels, sont parfois impossibles à mettre en œuvre avec Jess [Friedman-Hill, 2003].

L'architecture d'un programme qui interagit avec Jess dépend du niveau d'intégration entre la librairie et le code source Java. Voici les principaux niveaux d'intégration [Friedman-Hill, 2003] [Jess 7.0 Manual] :

1. Un programme simplement en Jess, sans code source Java, équivaut à une application de type ligne de commande, sans interface graphique. L'usager se borne à entrer du texte, pour répondre aux questions du système expert.
2. Simplement en Jess mais le programme accède à l'API Java ® 2, pour effectuer des tâches telles la création d'objets, l'appel de leurs méthodes et la manipulation de leurs propriétés.
3. Principalement du code Jess, avec quelques nouvelles commandes misent en œuvre en Java. Ces commandes sont appelées directement à partir de Jess.

4. Environ la moitié du code source en Jess, avec une part substantielle de code Java, pour concevoir de nouvelles commandes et accéder à l'API Java ® 2. Jess fournit le point d'entrée du programme.
5. La même chose que le point précédent, excepté que le point d'entrée du programme, se situe au niveau du code source Java.
6. Presque exclusivement du code source Java qui charge le code Jess au sein de la coquille, durant l'exécution du programme.
7. Uniquement du code source Java qui manipule Jess, à l'aide de son API Java.

Pour notre part, notre stratégie d'intégration implique de programmer les règles de production en Jess dans un fichier texte. Le programme Java charge ce fichier de règles au sein de la base de connaissances, avant que la simulation ne débute. Cette stratégie d'intégration de Jess, s'apparente à celle du point 6. Nous allons maintenant décrire ses principales caractéristiques.

3.4.4.2 Mécanisme d'inférence

À la section 3.4.2.2, nous avons vu que le mécanisme d'inférence choisi est celui avec *chaînage avant*. Il s'agit du mécanisme par défaut de la coquille de développement Jess [Friedman-Hill, 2003]. Sa mise en œuvre s'appuie sur l'algorithme de Rete [Forgy, 1982], dont il a été question au point 2.8.2. Jess ne nécessite aucune particularité pour spécifier qu'une règle de production est avec *chaînage avant*. En fait, comme nous avons mentionné à la section 2.5.3, il suffit d'utiliser l'instruction **`defrule`**, avec un identificateur, une prémisse (LHS) et une conséquence (RHS), pour qu'une nouvelle règle soit créée [Jess 7.0 Manual]. L'Annexe 4 du mémoire contient les règles avec *chaînage avant* du modèle.

3.4.4.3 Stratégie de résolution de conflits

Comme nous avons mentionné à la à la section 2.9, il importe de définir une *stratégie de résolution de conflits*, pour déterminer l'ordonnancement des règles de l'*agenda des activations*. À ce titre, Jess comporte deux stratégies de résolution, la première est dite *en profondeur* (« Depth »)

ou FIFO et la seconde *en largeur* (« Breadth ») ou LIFO [Jess 7.0 Manual]. Nous avons eu recours à la stratégie par défaut, soit celle *en profondeur*. Cette dernière convient généralement pour la plupart des problèmes [Friedman-Hill, 2003]. Par ailleurs, nous n'avons pas éprouvé les complications qui lui sont imputables et dont il a été question à la section 2.9.2.

3.4.4.4 Optimisation des règles

Comme on l'a vu, l'algorithme de Rete, sur lequel s'appuie Jess [Jess 7.0 Manual], permet de conserver les règles assorties en mémoire, afin de limiter le processus d'appariement d'un cycle d'inférence à l'autre. Or, l'ordonnancement des conditions (motifs) d'une règle peut avoir un impact sur le nombre d'*associations partielles* maintenues en mémoire (voir la section 2.8.2). Il importe de vérifier que chaque règle ne génère pas un nombre excessif d'*associations partielles*, afin d'optimiser la performance et l'utilisation de la mémoire [Giarratano et Riley, 1998]. Pour illustrer l'impact de l'ordonnancement des motifs, considérons les faits suivants :

```
F1 : (multiple-item a c e g)
F2 : (item a)
F3 : (item b)
F4 : (item c)
F5 : (item d)
F6 : (item e)
F7 : (item f)
F8 : (item g)
```

Considérons maintenant la règle `regle-1` :

```
(defrule regle-1 "Règle numéro 1"
      (multiple-item ?x ?y ?z ?w)   ; 1ère condition
      (item ?x)                     ; 2ème condition
      (item ?y)                     ; 3ème condition
      (item ?z)                     ; 4ème condition
      (item ?w)                     ; 5ème condition
=>
      (assert (item-trouve ?x ?y ?z ?w)))
```

Ses *associations de conditions* correspondent à :

```
1ère condition : F1
2ème condition : F2, F3, F4, F5, F6, F7, F8
3ème condition : F2, F3, F4, F5, F6, F7, F8
4ème condition : F2, F3, F4, F5, F6, F7, F8
5ème condition : F2, F3, F4, F5, F6, F7, F8
```

Quant aux *associations partielles*, considérant les variables qui lient les conditions, ce sont :

```
1ère condition : {F1}
2ème condition : {F1, F2}
3ème condition : {F1, F2, F4}
4ème condition : {F1, F2, F4, F6}
5ème condition : {F1, F2, F4, F6, F8}
```

Maintenant, modifions la règle `règle-1`, de façon à placer la première prémisse en dernier, nous obtenons ainsi la règle `règle-2`, c'est-à-dire :

```
(defrule regle-2 "Règle numéro 2"
      (item ?x)                      ; 1ère condition
      (item ?y)                      ; 2ème condition
      (item ?z)                      ; 3ème condition
      (item ?w)                      ; 4ème condition
      (multiple-item ?x ?y ?z ?w)    ; 5ème condition
=>
      (assert (item-trouve ?x ?y ?z ?w)))
```

Le nombre d'*associations de condition* est le même que pour `regle-1` :

```
1ère condition : F2, F3, F4, F5, F6, F7, F8
2ème condition : F2, F3, F4, F5, F6, F7, F8
3ème condition : F2, F3, F4, F5, F6, F7, F8
4ème condition : F2, F3, F4, F5, F6, F7, F8
5ème condition : F1
```

Par contre, le nombre d'*associations partielles*, pour la première condition est le suivant :

```
{F2}, {F3}, {F4}, {F5}, {F6}, {F7}, {F8}
```

Quant à la deuxième condition, elle possède 42 *associations partielles* :

```
{F2, F3}, {F2, F4}, {F2, F5}, {F2, F6}, {F2, F7}, {F2, F8}
{F3, F2}, {F3, F4}, {F3, F5}, {F3, F6}, {F3, F7}, {F3, F8}
{F4, F2}, {F4, F3}, {F4, F5}, {F4, F6}, {F4, F7}, {F4, F8}
{F5, F2}, {F5, F3}, {F5, F4}, {F5, F6}, {F5, F7}, {F5, F8}
{F6, F2}, {F6, F3}, {F6, F4}, {F6, F4}, {F6, F7}, {F6, F8}
{F7, F2}, {F7, F3}, {F7, F4}, {F7, F5}, {F7, F6}, {F7, F8}
{F8, F2}, {F8, F3}, {F8, F4}, {F8, F5}, {F8, F6}, {F8, F7}
```

Dans cet exemple particulier, le nombre d'*associations partielles* croît, à mesure que le nombre de condition à traiter augmente. En fait, il y a autant d'*associations partielles* qu'il y a d'arrangements de **c** conditions parmi **n** faits appariés aux conditions. Dans cet exemple précis, le nombre d'*associations partielles* correspond donc à :

$$A_n^c = \frac{n!}{(n-c)!} \quad \text{où } c < n$$

A : le nombre d'*associations partielles*
n : le nombre de faits à associés aux conditions
c : le nombre de conditions pour la vérification des *associations partielles*

Puisque pour les deux premières conditions (**c** = 2), nous avons sept faits associés (**n** = 7), le nombre d'*associations partielles* est :

$$A_7^2 = \frac{7!}{(7-2)!} = \frac{7 \times 6 \times 5!}{5!} = 42$$

Idéalement, chaque règle devrait donc faire l'objet d'une analyse avant d'être rédigée, afin de vérifier leur impact sur les performances. Voici quelques principes généraux selon [Giarratano et Riley, 1998] que nous avons considérés, lors de la mise en œuvre des règles de production du modèle :

- On doit ajouter les conditions qui sont les plus spécifiques au début de la règle, afin de limiter le nombre d'*associations partielles*. Les conditions les plus spécifiques sont celles qui ont le moins de faits associés et le plus grand nombre de variables qui contraignent les autres conditions. La condition `(multiple-item ?x ?y ?z ?w)` des règles `regle-1` et `regle-2` constitue un bon exemple.
- Les conditions qui sont appariés à des faits qui sont fréquemment ajoutés ou supprimés doivent être placées à la fin de la règle. Ceci cause un nombre de changement moindre au niveau des *associations partielles* de la règle.
- À l'inverse, les conditions qui sont appariées à seulement quelques faits doivent être placées au début de la règle. Une fois encore, ceci permet de limiter la modification d'*associations partielles*.

3.4.4.5 Les « shadows facts »

Un « shadow fact » est un fait non-ordonné (voir section 2.6.2), dont les fentes correspondent aux propriétés d'un objet JavaBeans. » NDLR traduit de l'Anglais [Friedman-Hill, 2003]. Tel que stipulé à l'Annexe 3, les objets JavaBeans, appelés aussi Beans, sont des composants Java [JavaBeans]. Les « shadow facts » servent donc à établir la connexion entre la

mémoire de travail et les Beans d'un programme Java qui appelle Jess. En fait, il est possible d'insérer n'importe quel objet Java au sein de la mémoire de travail [Jess 7.0 Manual]. Les « shadow facts » sont nommés ainsi, parce qu'ils agissent comme une image, autrement dit l'ombre (de l'Anglais « shadow ») des Beans [Friedman-Hill, 2003].

Modèle de fait

Tout comme les faits non-ordonnés, dont il a été question à la section 2.6.2, les « shadow facts » utilisent une instruction pour définir les *modèles de faits* (de l'Anglais « templates »). Toutefois, les *fentes* (« slots ») de leur *modèle de faits* sont explicitement associées aux propriétés JavaBeans d'une classe. La façon la plus simple de définir un modèle consiste à faire appel à la fonction **`defclass`** [Jess 7.0 Manual] :

```
(defclass <Nom modèle> <Nom classe Java> [<Nom modèle ancêtre>])
```

Par exemple, pour définir le *modèle de fait ProcessedEntityTmpl*, associé à la classe *ProcessedEntity* du paquetage Java *process*, on utilise la fonction **`defclass`** avec les arguments suivants :

```
(defclass "ProcessedEntityTmpl" "process.ProcessedEntity")
```

De plus, pour définir le modèle *AnodeTmpl*, associé à la classe *Anode* du paquetage *process*, qui hérite du modèle *ProcessedEntityTmpl*, on utilise :

```
(defclass "AnodeTmpl" "process.Anode" "ProcessedEntityTmpl")
```

Création d'un « shadow fact »

Une fois qu'un *modèle de fait* a été créé, il est possible d'y associer un objet Java et de l'ajouter à la mémoire de travail. Le « shadow fact » correspondant à cet objet est alors automatiquement affirmé. Pour ce faire, on utilise la fonction **`definstance`** [Jess 7.0 Manual] :

```
(definstance <Nom modèle> <Objet Java> [static | dynamic])
```

Le troisième argument de la fonction (`[static | dynamic]`) est optionnel. Il est décrit un peu plus loin. L'appel ci-dessous montre comment créer un nouvel objet *Anode* et son « shadow fact » :

```
(bind ?anAnode (new process.Anode))     ; Création de l'objet Java et
                                        ; affectation à la variable
                                        ; ?anAnode.
(definstance AnodeTmpl ?anAnode)        ; Création du "shadow fact".
```

Ces fonctions peuvent aussi être appelées à partir de l'API Java de Jess. Celui-ci dispose de la classe `Rete` qui constitue le cœur de la coquille de développement. Une instance de `Rete` exécute le réseau de Rete et coordonne plusieurs autres activités, dont la création de *modèles de faits* (**defclass**) et de « shadow facts » (**definstance**) [Jess 7.0 Manual]. Voici la reproduction de l'exemple précédent, à partir de l'API Java de Jess :

```
// Objet représentant le réseau de Rete.
Rete reteObj = new Rete();

// Définition des modèles de faits.
reteObj.defclass("ProcessedEntityTmpl", "process.ProcessedEntity", null);
reteObj.defclass("AnodeTmpl", "process.Anode", "ProcessedEntityTmpl")

// Crée les instances de classes avec des propriétés JavaBeans.
ProcessedEntity processedEntityObj = new process.ProcessedEntity();
Anode anodeObj = new process.Anode();

// Crée les "shadow facts" associés aux instances.
reteObj.definstance("ProcessedEntityTmpl", processedEntityObj, true);
reteObj.definstance("AnodeTmpl", anodeObj, true);
```

Mise en œuvre d'un Bean

Pour que la correspondance entre les fentes des « shadow facts » et les propriétés d'un Bean puisse se faire, il importe que celles-ci soient définies de façon standard. Une propriété JavaBeans (notée `<Nom de propriété>`) est associée à deux méthodes, dont les noms sont formatés ainsi [JavaBeans Simple Properties] :

1. `get<Nom de la propriété>()` permet de lire la valeur de la propriété.
2. `set<Nom de propriété>(<Type de la propriété> <variable>)` permet de modifier la propriété.

L'exemple suivant montre la classe `Anode` qui comporte une propriété JavaBeans appelée `State` :

```
public class Anode {
     private int state = 0;
```

```
        public void setState(int newState) {
              state = newState;
        }
        public int getState() {
              return state;
        }
}
```

L'API JavaBeans dispose de la classe `Introspector`, pour examiner un Bean et déterminer ses propriétés qui suivent la convention `get` / `set` [JavaBeans Concept]. Jess utilise cette même classe, afin de générer les *modèles de faits* (« template »), avec les fentes appropriées. Un « shadow fact » créé à partir de ce modèle agit donc comme un adaptateur du Bean. Les fentes du « shadow fact » contiennent automatiquement les valeurs des propriétés du Bean [Friedman-Hill, 2003].

Type de « shadow fact »

Un « shadows fact » peut être *statique* ou *dynamique*. Pour spécifier le type de « shadow fact », il suffit d'affecter le troisième argument de la méthode **`definstance`**, dont il a été question plus haut. Les valeurs des fentes d'un « shadow fact » *statique* ne sont pas modifiées si les propriétés du Bean correspondant changent. Toutefois, elles sont rafraîchies lorsque la méthode **`reset`** est appelée [Jess 7.0 Manual].

Pour qu'un « shadow fact » suive continuellement les changements de valeurs des propriétés du Bean associé, son type doit être *dynamique*. Toutefois, il faut que Jess soit notifié des changements du Bean. Pour ce faire, la propriété du Bean, adjointe à la *fente* d'un « shadow fact », déclenche un événement Java, lorsque sa valeur change. Cet événement est capté par un écouteur. C'est ce qu'on appelle une propriété liée [JavaBeans Bound Properties]. Ce type de propriété est mise en œuvre par les objets JavaBean de notre modèle. L'exemple ci-dessous montre comment définir la propriété liée `State` de la classe `Anode` :

```
public class Anode {
      private int state = 0;

      public void setState(int newState) {
            // On préserve l'ancienne valeur de la propriété.
```

```
            int = oldState = state;
            // On effectue le changement de valeur de la propriété.
            state = newState;

            // On déclenche l'événement de l'écouteur, pour que Jess
            // soit notifié du changement de valeur de la propriété.
            changeSupport.firePropertyChange(
                  "state", new Integer(oldSate), new Integer(newState));
      }
      public int getState() {
            return state;
      }
      // On crée l'écouteur, pour plus de détails voir
      // [JavaBeans Bound Properties].
      private PropertyChangeSupport changeSupport =
            new PropertyChangeSupport(this);
      ...
}
```

Avantages des « shadows fact »

Les « shadow facts » rendent la synchronisation transparente et évitent toute forme de duplication de l'information, puisque les objets JavaBeans sont des faits du point de vue de Jess [Friedman-Hill, 2003]. À l'inverse, cette duplication est inévitable, lorsqu'on considère le système expert élémentaire de la section 3.4.3, car il faut synchroniser continuellement les faits et les objets Java. Les « shadows facts » permettent aux règles de production de référencer les attributs et les méthodes d'un Bean, au lieu de recourir à des structures de données intermédiaires, comme on a vu à la section 3.4.3.2.

Finalement, les « shadow facts » facilitent la mise en œuvre d'une relation de généralisation [Fowler, 2004] entre les faits. Pour ce faire, il suffit d'affecter le nom du modèle ancêtre, lors de l'appel de la fonction **`defclass`** [Jess 7.0 Manual]. Ceci peut s'avérer très pratique, pour rendre une règle plus ou moins spécialisée. Par exemple, dans le cas de notre problème, nous avons trois sous-classes (`Anode`, `Rod` et `AnodicSet`) qui sont une généralisation de `ProcessedEntity` (voir le diagramme UML des classes de la Figure 11). Il devenait donc possible d'écrire des règles adressant tous les faits de ces trois classes spécialisées, grâce à `ProcessedEntity`, ou encore seulement les faits d'une classe en particulier (`Anode`, `Rod` ou `AnodicSet`).

3.4.4.6 Module

Un programme typique, s'appuyant sur un système expert, comporte généralement plusieurs dizaines de règles de production. Des programmes encore plus élaborés peuvent comprendre jusqu'à des centaines de règles. Notre modèle intègre, quant à lui, une vingtaine de règles de production (voir Annexe 4). Or, même dans ce cas, sa mise en œuvre apparaît complexe, puisque plusieurs règles peuvent interagir entre elles et qu'il est difficile de gérer toutes ces interactions.

Jess résout ce problème, en divisant les faits et les règles en regroupements distincts : les modules. Un module permet d'organiser les règles en unités logiques qui peuvent être référencées individuellement. D'autre part, un module sert de mécanisme de contrôle, puisque ses règles sont déclenchées uniquement lorsqu'il devient le *module cible* (de l'Anglais « *focus module* ») [Friedman-Hill, 2003].

Définition d'un module, *module courant* et résolution de nom

Dès qu'un module est défini, il devient le *module courant* (de l'Anglais « active module »), à ne pas confondre avec le *module cible* ou « focus module », dont il est question un peu plus loin. Par défaut, Jess définit toujours un module appelé `MAIN`. Initialement, celui-ci constitue le *module courant*. Si aucun module n'est spécifié durant la définition d'une règle ou d'un *modèle de fait*, alors leur module devient le *module courant* [Jess 7.0 Manual]. Par exemple, la commande pour définir le module `STATUS` est :

```
(defmodule STATUS)
```

Pour spécifier que le module d'un *modèle de fait* (`anode`) ou d'une règle (`check-status`) est `STATUS`, il suffit de faire précéder leur définition par « `STATUS::` » :

```
(deftemplate STATUS::anode (slot status))

(defrule STATUS::check-status
      (STATUS::anode (status ?s))
=>
...)
```

Toutefois, puisque le module `STATUS` vient tout juste d'être défini, à l'aide de la commande **defmodule** et qu'il s'agit du *module courant*, nous aurions pu omettre l'identificateur « `STATUS::` »

des instructions **deftemplate** et **defrule**. Mentionnons que lorsque Jess compile une règle, il recherche toujours un *modèle de fait* dans l'ordre suivant [Friedman-Hill, 2003] :

1. Si la condition spécifie explicitement un module, alors le *modèle de fait* est recherché dans ce module. C'est le cas du précédent exemple, puisque la condition « `(STATUS::anode (status ?s)` » de la règle `check-status` spécifie le module `STATUS`.
2. Si la condition ne spécifie aucun module, alors le *modèle de fait* est recherché dans le module de la règle. Par exemple si la condition `(STATUS::anode (status ?s)` de la règle « `STATUS::check-status` » n'avait pas spécifié de module et avait été « `(anode (status ?s))` », alors Jess aurait recherché le *modèle de fait* `anode` dans le module `STATUS`, car il s'agit du module de la règle.
3. Si le *modèle de fait* n'est pas trouvé dans celui de la règle, alors il est recherché dans le module `MAIN`. Naturellement si la recherche n'est toujours pas fructueuse, alors la règle ne sera jamais activée.

***Module cible* et ordre de déclenchement**

Il est aussi possible de définir le *module cible* (« focus module »), pour contrôler l'ordre de déclenchement des règles. Bien que plusieurs règles puissent être *activées* en même temps, seulement celles du *module cible* vont être *déclenchées* [Friedman-Hill, 2003]. Par exemple, la règle suivante ne sera jamais déclenchée, même si elle n'a aucun motif :

```
(defmodule STATUS)

(defrule STATUS::my-rule
=>
(printout t "La règle est déclenchée." crlf))
```

Par contre si la commande **focus** [Jess 7.0 Manual] est appelée avec comme paramètre `STATUS`, alors `STATUS` devient le *module cible* et la règle `my-rule` est déclenchée :

```
(focus STATUS)
```

Notons qu'il est possible d'appeler **focus** et de changer le *module cible*, dans la partie droite d'une règle (RHS). Mentionnons aussi que Jess maintient la *pile des cibles* (« focus stack ») qui

comporte tous les modules. La commande `focus` sert à placer le nouveau *module cible* sur la *pile des cibles*. Ainsi, lorsqu'il n'y a plus aucune règle *activée* dans le *module cible*, celui-ci est retiré du sommet de la *pile des cibles* et le module suivant devient le *module cible.*

Avantages des modules

La notion de module est fort utile, puisque bien que notre problème n'ait pas de solution connue d'avance, il est parfois nécessaire de définir un certain ordre de raisonnement. Ainsi, avant d'effectuer la transition des états des objets, entre autre lorsqu'une ressource entame une activité, il faut vérifier sont état courant : actif ou inactif. C'est pourquoi notre modèle incorpore deux modules formés de règles qui se chargent de la vérification des états (`SIGNAL` et `STATUS`), ainsi que trois modules responsables de la transition des états (`PROCESS`, `PROCESS-GREEN` et `PROCESS-BAKED`). Globalement, notre modèle comporte les modules suivants : `MAIN`, `FINALIZE`, `PROCESS`, `PROCESS-GREEN`, `PROCESS-BAKED`, `SIGNAL`, `STATUS`. Il y a donc au moins six niveaux d'imbrication de règles, puisqu'il n'y aucune règle définie dans le module `MAIN`. Les modules sont placés au sein de la *pile des modules cibles* dans l'ordre énuméré ci-dessus. Le module `STATUS` est donc appelé en premier et le module `MAIN` en dernier. Tous les *modèles de faits* sont associés au module `MAIN` (voir l'Annexe 4).

3.4.4.7 Point d'entrée du programme Jess

Nous allons maintenant décrire l'association entre le modèle Java et la coquille Jess. Une fois que l'usager a déterminé les paramètres de simulation ; le nombre de pas de temps, la base de connaissances et la mémoire de travail de Jess sont initialisées. Durant le premier pas de temps, le moteur d'inférence démarre. Puis, celui-ci effectue une ou plusieurs itérations, pour d'une part, vérifier les règles qui sont activées et d'autre part les exécuter. L'ordre d'exécution est défini par les modules et la stratégie d'activation. Les itérations se poursuivent, jusqu'à ce qu'il n'y ait plus aucune règle active [Friedman-Hill, 2003]. Ensuite, le simulateur incrémente le pas de temps et met à jour l'interface utilisateur décrivant l'évolution du procédé. Puis, le mécanisme d'inférence est à

nouveau lancé pour le second pas de temps. Ce cycle se répète, jusqu'à ce que tout le temps spécifié par l'utilisateur se soit entièrement écoulé. Ci-dessous, se trouve le pseudo-code simplifié illustrant l'exécution de l'algorithme que nous venons de décrire :

```
1. Initialiser la base de connaissances et la mémoire de travail
2. Lire le nombre de pas de temps (n)
3. Itérer de 1 à n pas de temps :
   3.1.  Tant qu'il y a des règles actives :
      3.1.1.   Déterminer les règles actives
      3.1.2.   Exécuter les règles actives
      3.1.3.   Modifier la mémoire de travail (base de faits)
   3.2.  Modifier l'interface usager
```

Ce pseudo-code ressemble à l'algorithme naïf de la section 2.8.1. Néanmoins, comme nous l'avons mentionné à la section 2.8.2.4, en recourant à l'algorithme de Rete, dans le cas moyen, on peut s'attendre à une amélioration de la performance, lorsque peu de faits sont modifiés. Or, l'état des entités du modèle ne risque pas de changer à chaque pas de temps. D'ailleurs, pour certaines itérations, aucun changement ne survient.

Nous terminons ainsi ce chapitre, puisque nous avons vu en détail la problématique étudiée et l'architecture du modèle qui doit permettre de la résoudre. Nous allons maintenant montrer au cours du chapitre suivant, comment nous avons validé le modèle, quels ont été les principales expériences effectuées et quelles sont nos conclusions, à la lumière de l'analyse des résultats obtenus.

CHAPITRE 4

VALIDATION, EXPÉRIMENTATION ET ANALYSE DES RÉSULTATS

4 Validation, expérimentation et analyse des résultats

4.1 Limitation de la validité

Comme nous avons vu à la section 1.4, le processus de modélisation incorpore une étape de validation qui précède l'expérimentation à partir de scénarios. Cette étape permet de s'assurer que le modèle est une représentation raisonnable du système actuel [Law et Kelton, 2000]. Règle générale, le modèle ne peut représenter exactement la réalité, en raison d'actions posées par le spécialiste en modélisation. Ces actions peuvent être imputables à des hypothèses qui ont été émises, à des simplifications, des oublis, voire même à certaines limitations [Chung, 2004]. Sans entrer dans le détail de toutes les causes qui limitent la validité de notre modèle, nous allons en énumérer quelques-unes.

4.1.1 Hypothèses de modélisation

Des hypothèses de modélisation sont souvent posées, parce que le spécialiste manque de connaissance par rapport au fonctionnement du système. Ceci est d'autant plus vrai, lorsque le système est inexistant ou ne peut pas être observé [Pidd, 1998]. De plus, il n'est pas rare que le spécialiste soit confronté à un manque de données qui décrivent le système. Dans ce cas, plutôt que d'attendre que celles-ci soient disponibles, le spécialiste décide de poursuive la conception du modèle et de poser une hypothèse [Chung, 2004].

Par exemple, dans le cas de notre modèle, il n'était pas possible de répéter plusieurs fois l'expérience d'échantillonnage qui a été décrite à la section 3.3.3.2. Nous avons donc décidé de poser une hypothèse, en ce qui concerne l'assignation des valeurs de certaines variables aléatoires du modèle (voir la section 3.3.3.1). Nous avons assumé que les distributions de probabilités de deux variables (« temps de service » et « temps de transit ») pouvaient être remplacées par leur moyenne échantillonnale (voir la discussion de la section 3.3.3.6).

4.1.2 Simplifications du modèle

Le spécialiste doit parfois simplifier le modèle d'un système, pour entre autre, respecter l'échéancier du projet [Banks et al, 1998]. Également, il peut arriver que certains détails du fonctionnement interne du système soient jugés trop complexes ou non significatifs. Une simplification commune consiste à modéliser plusieurs sous-procédés du système en un seul. Par exemple si un système est basé sur plusieurs variables aléatoires, nous pourrions le simplifier, en collectant les données qu'une seule fois. Ainsi, nous modéliserions qu'une variable au lieu de plusieurs. Le spécialiste peut aussi simplifier le modèle s'il décide délibérément qu'une partie du système n'a pas besoin d'être modélisée. Ceci est notamment envisagé lorsque la partie en question a peu ou pas d'impact sur le reste du modèle [Chung, 2004].

Dans le cadre de notre modèle, nous nous sommes livrés à certaines simplifications. Par exemple, celui-ci ne comporte pas la ressource correspondant à la tour à pâte, puisqu'elle a été modélisée en conjonction avec le vibrocompacteur et la presse (voir la section 3.3.1). Également, nous n'avons pas représenté la progression détaillée des anodes au sein des fours via les trains. Nous nous sommes bornés à représenter les fours comme de simples ressources du système.

4.1.3 Omissions au sein du modèle

Dès que le spécialiste est confronté à un système complexe, il est fort probable qu'il oublie d'intégrer certains éléments au modèle. Dans une telle situation, l'impact sur les mesures de performances [Law et Kelton, 2000] risque d'être significatif et il sera impossible de valider le modèle. Il n'existe aucune solution définitive à ce problème. Une liste des leçons apprises durant chaque projet peut être souhaitable [Chung, 2004]. Pour notre part, nous avons maintenu tout au long du projet, une liste des tâches à effectuer. Celle-ci nous garantissait à tout le moins, que nous n'omettrions aucune de ces tâches et pouvait à la rigueur, nous aider à en identifier d'autres.

4.1.4 Limitations durant la modélisation

Il arrive qu'il soit très difficile de modéliser un système en raison de certaines limitations. L'impact de chacune d'entre elles peut varier de façon significative sur la validité du modèle [Chung, 2004]. On peut penser aux limitations inhérentes au spécialiste, occasionnées entre autre, par un manque de formation. D'autres limitations peuvent aussi apparaître si un VIMS (voir la section 1.8) est utilisé et qu'il ne dispose pas de toutes les fonctionnalités voulues [Pidd, 1998].

Comme nous n'avons pas recouru à des VIMS, nous n'avons pas eu à faire face à ce type de problème. D'ailleurs, comme nous l'avons vu à la section 1.8.4, les modèles conçus à partir de VIMS sont beaucoup moins flexibles que ceux conçus à partir d'une librairie de simulation ou d'un langage de troisième génération, tel Java [Law et Kelton, 2000]. Toutefois, le temps de conception risque d'être accru si cette dernière solution est empruntée.

Finalement, il est possible qu'on ne puisse pas valider adéquatement le système en raison d'un problème durant la collecte des données [Chung, 2004]. Par exemple, il se peut qu'on ait besoin de plusieurs mois pour collecter suffisamment de données, pour effectuer une analyse statistique robuste. Dans ce cas, le spécialiste n'a d'autre choix que de poursuivre l'étude sans ces données. Lorsqu'il est confronté à un manque de données pour décrire une partie d'un système, le spécialiste est souvent amené à regrouper plusieurs sous-processus. Cette limitation est similaire à une simplification du modèle et entraîne les mêmes désavantages.

4.2 Validation en surface

La validation en surface (traduction de l'Anglais « face validity ») est faite en compagnie de l'expert du système modélisé. On considère que le modèle est valide en surface si les résultats de la simulation semblent corroborer avec le comportement du système [Law et Kelton, 2000]. La validation en surface est fondamentale, puisqu'elle permet de faire la promotion du modèle auprès de l'expert, de répondre aux interrogations du spécialiste et de réduire les demandes d'ajustements du modèle [Chung, 2004].

Trop souvent, la validation en surface survient à la fin de la conception du modèle, afin de faire des mises aux points avant la validation statistique [Law et Kelton, 2000] [Pidd, 1998]. Cependant, elle doit être plutôt considérée comme un processus continu. Ainsi, entre la première rencontre avec l'expert et le rapport final, le spécialiste doit planifier des rencontres additionnelles, pour discuter de la validité en surface du modèle.

Afin de mener à bien le processus de validation en surface, le spécialiste s'assure que le modèle ait une certaine ressemblance visuelle avec le système. Ceci ne signifie pas que l'interface et les animations du modèle doivent être la réplication exacte du système actuel. De toute manière, les représentations à l'échelle fournissent rarement assez de détails visuels pour que l'expert puisse appréhender correctement le système [Chung, 2004].

Pour notre part, nous avons choisi d'utiliser un graphe dirigé [Cormen et al., 1994] pour représenter les étapes de *formation, manutention et entreposage des anodes* [Totten et Mackenzie, 2003]. Chaque nœud du graphe symbolise l'une des principales ressources du système. Un nœud affiche l'identificateur unique et le nom de la ressource, ainsi que deux mesures de performances qui changent progressivement, soit le nombre d'entités entrées et sorties de la ressource. Puisque les experts du domaine veulent souvent savoir comment le modèle traite les événements insolites [Chung, 2004], les ressources changent de couleur au gré d'états particuliers :

- Rouge : la ressource est non-opérationnelle en raison d'un arrêt *planifié* ou *non-planifié*.
- Orange : les grues ont cette couleur si leur temps de service est supérieur au temps attendu. Les autres ressources sont ainsi colorées, lorsque leur temps d'inactivité (temps en attente) dépasse le temps de transit maximal des files en aval. Ceci permet d'identifier les blocages au niveau des files en aval.
- Vert : la ressource est en traitement. Lorsqu'une ressource, à l'exception d'une grue, a cet état, alors nous avons [Nombre d'entités entrées] = [Nombre d'entités sorties] +1.
- Vert foncé : la ressource est inactive (en attente). Lorsqu'une ressource, à l'exception d'une grue, a cet état, alors nous avons [Nombre d'entités entrées] = [Nombre d'entités sorties].

- Blanc : la ressource est dans un état indéfini. Il s'agit d'une situation anormale qui doit être vérifiée par le spécialiste.

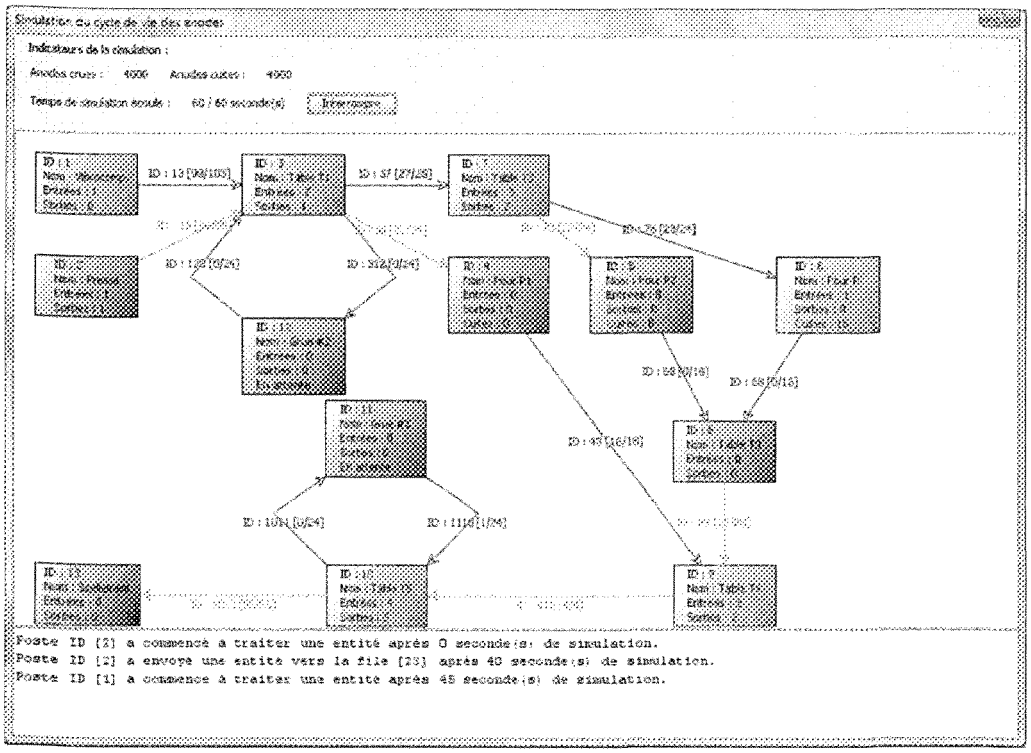

Figure 29 : représentation des étapes de *formation, manutention et entreposage des anodes*

Quant aux files, c'est-à-dire les convoyeurs du système, elles sont symbolisées par les arêtes du graphe. Chacune d'elles affichent un identificateur unique, le nombre d'entités en transit et leur capacité. Les files changent également de couleur en fonction d'états particuliers :

- Rouge foncé : la file est non-opérationnelle, en raison d'un événement *planifié* ou *non-planifié*. Ainsi, on ne peut pas ajouter ou enlever d'entité à la file. Néanmoins, celles qui sont déjà en cours de transit continues à avancer.
- Rouge : le transit des entités dans la file est interrompu, à cause d'un événement *planifié* ou *non-planifié*. Toutefois, on peut encore ajouter ou enlever une entité à la file.

- Orange : l'entité à la tête de la file est disponible, depuis plus de temps qu'il n'en faut pour faire transiter toutes les entités de la file, lorsqu'elle est à sa pleine capacité. Cette règle empirique nous permet d'identifier les blocages au niveau de la ressource en aval.
- Vert : l'entité à la tête de la file a terminé son transit et est donc disponible pour être traitée par la ressource aval.
- Vert foncé : l'entité à la tête de la file n'a pas terminé son transit ou bien la file est vide.

Notons également que l'interface comporte deux indicateurs qui affichent le nombre d'anodes crues et cuites qui sont actuellement dans l'entrepôt. Cette mesure de performance varie selon l'opération qu'est en train d'effectuer la grue gerbeuse.

4.3 Validation statistique

Afin de valider le modèle nous avons simulé le cas de référence. Celui-ci correspond à un scénario de production standard, d'une durée de sept jours, du lundi 00:00:00 au dimanche 23:59:59. Les données du cas de référence nous ont été fournies par les spécialistes du système. Ainsi, nous avons pu les comparer avec les résultats générés par la simulation de ce cas. Le comparatif a été effectué pour certaines ressources du système. Bien entendu, il est possible de s'intéresser à de nombreuses mesures de performances des ressources. Toutefois, les données fournies par les spécialistes nous permettaient d'établir une corrélation avec seulement un certain nombre de mesures, pour des raisons techniques et de confidentialité. Nous avons donc uniquement effectué notre validation à partir du nombre d'anodes traitées par le *vibrocompacteur*, la *presse*, les *fours* (*P1*, *P2*, *R*) et le *scellement*. Le Tableau 3 montre les données du système durant une semaine de production (ligne [1]), ainsi que les résultats produits par le modèle (ligne [2]). La dernière ligne donne le pourcentage d'écart entre le modèle et le système selon l'équation suivante :

$$\%\text{écart} = \frac{|\,[\text{Système}] - [\text{Modèle}]\,|}{[\text{Système}]} \times 100$$

	Vibro-comp. / presse	*Four P1* (cycles de 36 heures)	*Four P2* (cycles de 36 heures)	*Four R* (cycles de 36 heures)	*Four R* (cycles de 32 heures)	*Scellement*
[1] Système	7400	2268	2268	2613	2690	6652
[2] Modèle	7310	2268	2214	2625	2930	6697
$\% = \frac{\lvert [1] - [2] \rvert}{[1]} \times 100$	1,22%	0%	2,38%	0,46%	8,92%	0,68%

Tableau 3 : comparaison du système et du modèle pour le cas de référence

Notons que le pourcentage d'écart est toujours inférieur à 9%, ce qui est tout à fait acceptable dans les circonstances. En fait, outre le résultat obtenu avec un cycle de cuisson de 32 heures pour le *four R*, les données impliquent des pourcentages d'écart tout à fait négligeables.

4.3.1.1 Validation de l'entrepôt

Afin de valider les données générées par le modèle, nous avons également soumis l'évolution des anodes entreposées du lundi 00:00:00 au dimanche 23:59:59, aux opérateurs du *cycle de vie des anodes*. Le comparatif avec l'historique de production de l'usine, s'est avéré positif. Les figures ci-dessous montrent l'évolution de l'*entrepôt* selon le modèle. Notons, encore une fois, que pour des raisons de confidentialité, nous ne pouvons pas présenter l'évolution du système réel.

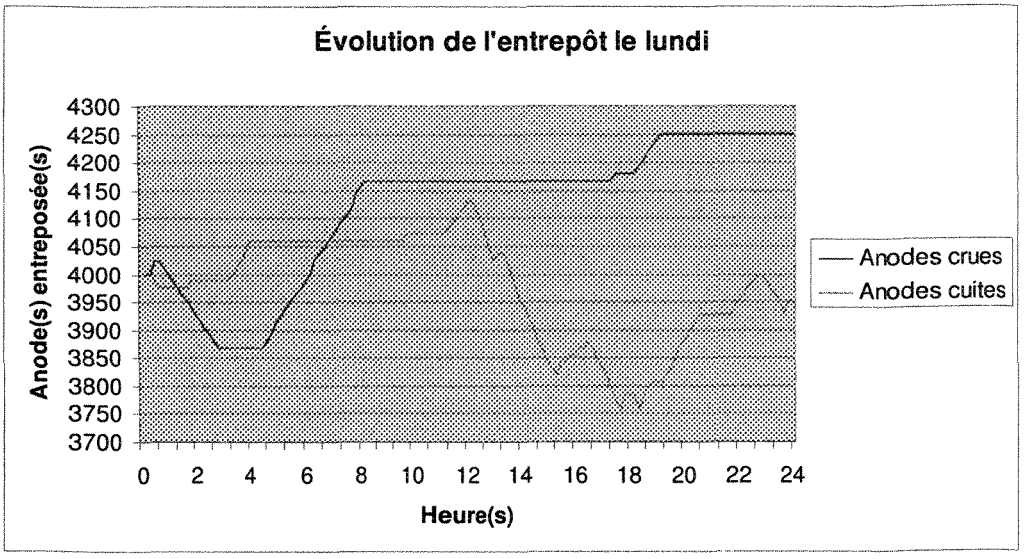

Figure 30 : évolution de l'*entrepôt* le lundi

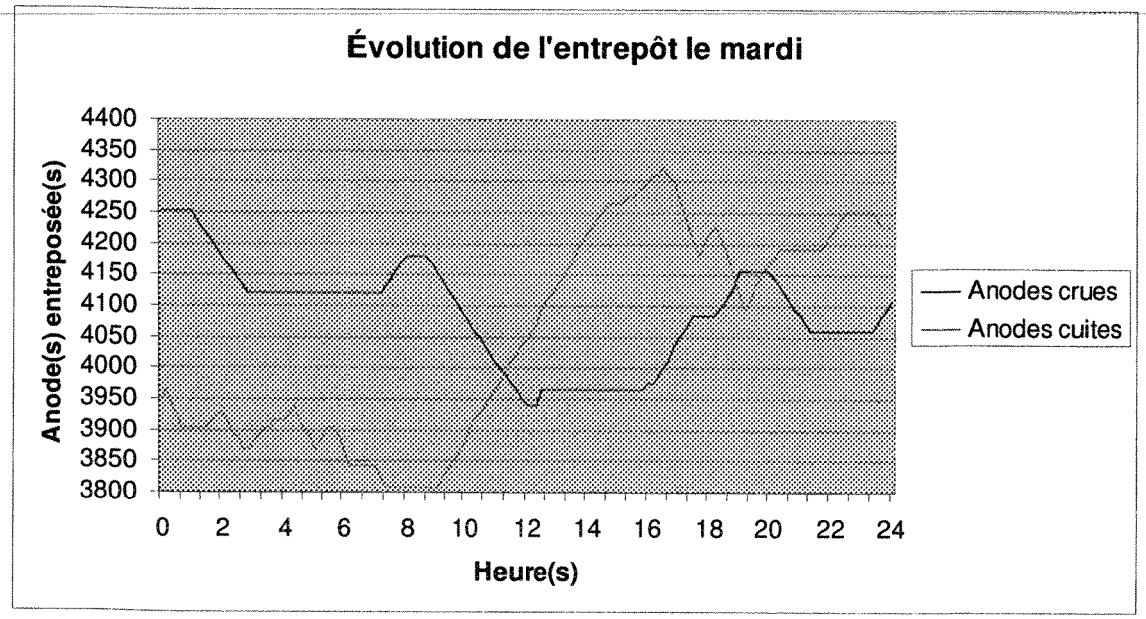

Figure 31 : évolution de l'*entrepôt* le mardi

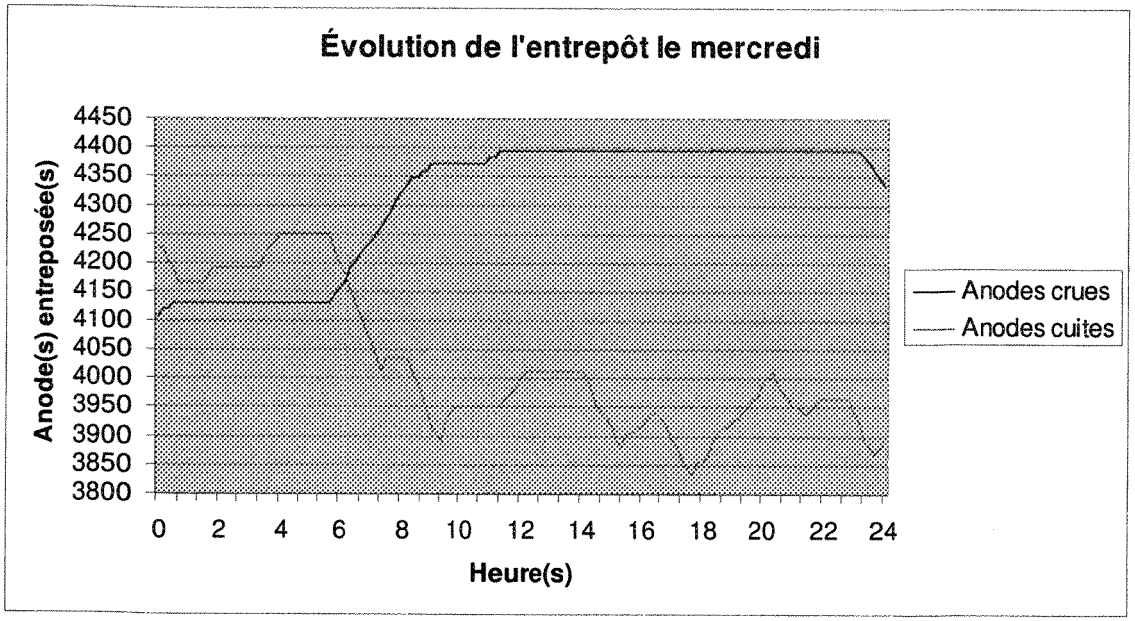

Figure 32 : évolution de l'*entrepôt* le mercredi

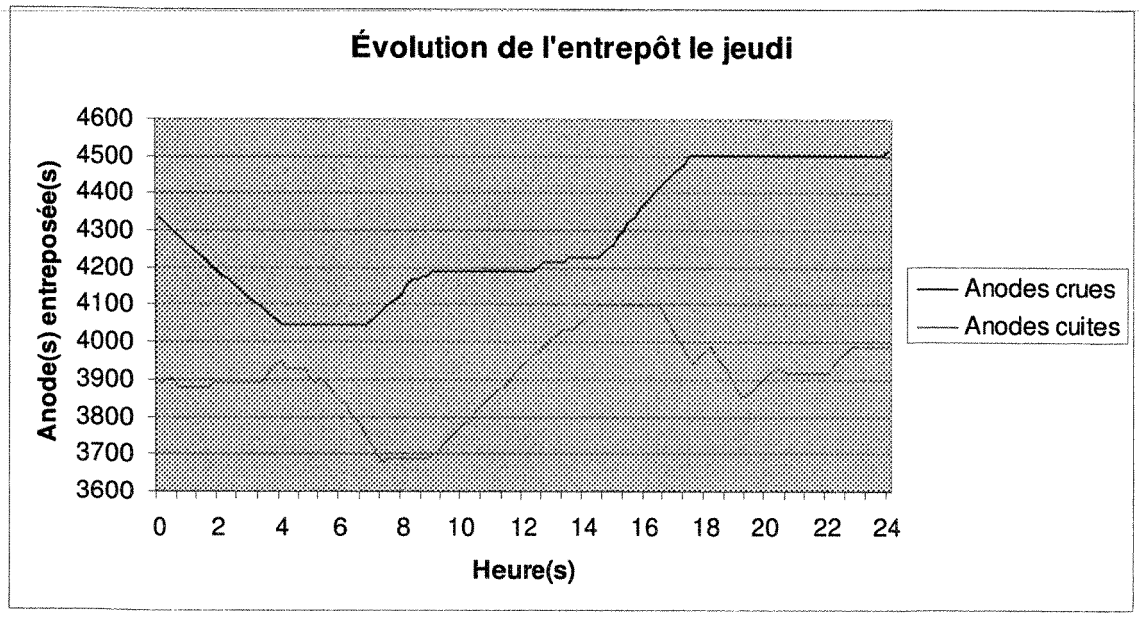

Figure 33 : évolution de l'*entrepôt* le jeudi

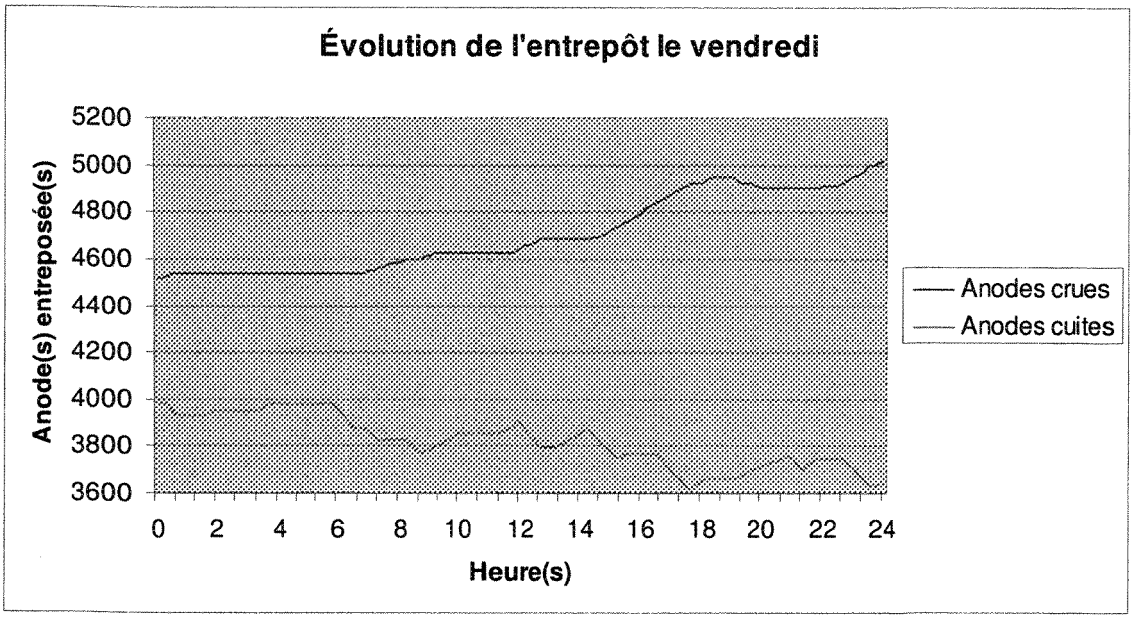

Figure 34 : évolution de l'*entrepôt* le vendredi

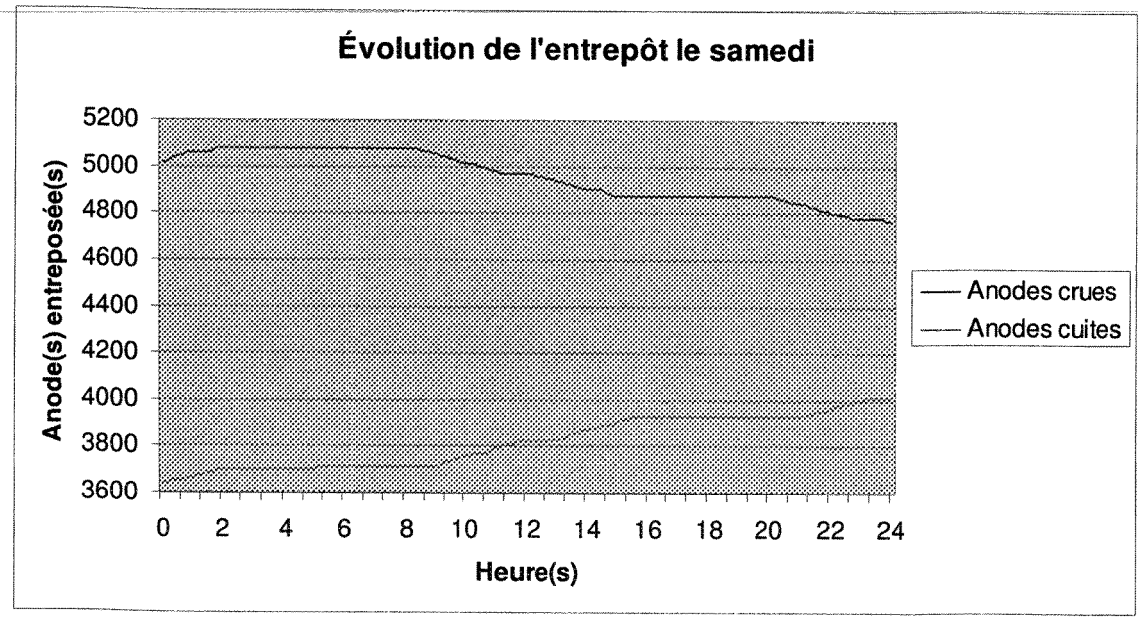

Figure 35 : évolution de l'*entrepôt* le samedi

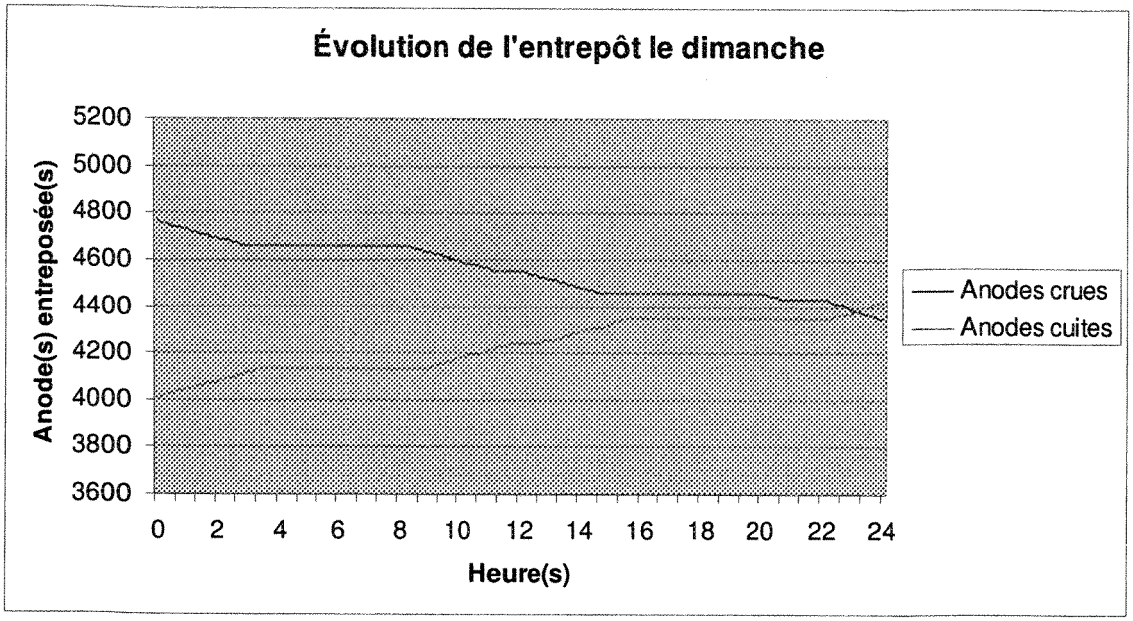

Figure 36 : évolution de l'*entrepôt* le dimanche

4.3.1.2 Validation des fours

Toujours pour des fins de validation, nous avons également soumis aux opérateurs le nombre d'anodes cuites par les *fours P1*, *P2* et *R*, du lundi 00:00:00 au dimanche 23:59:59. Les résultats des figures suivantes montrent la production des *fours*.

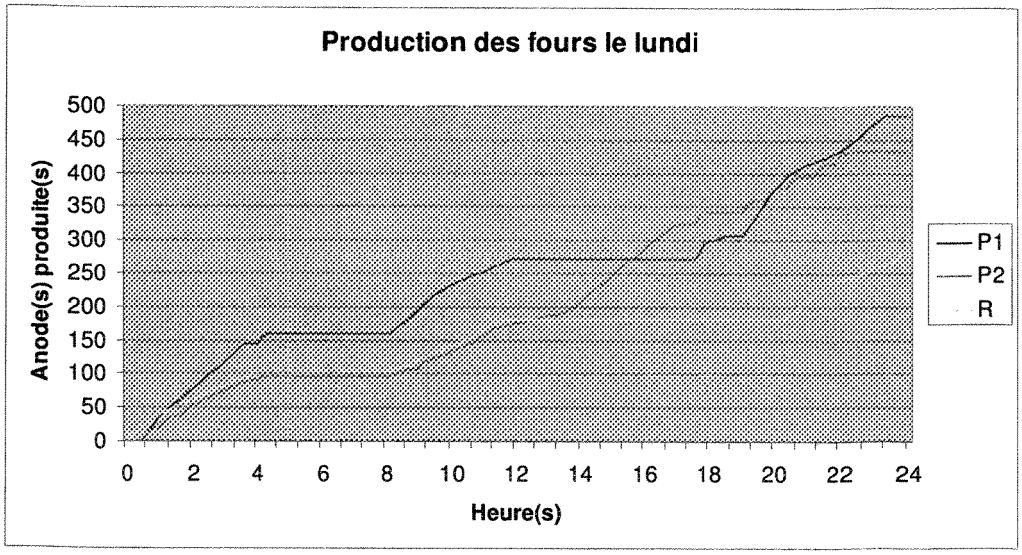

Figure 37 : production des *fours* le lundi

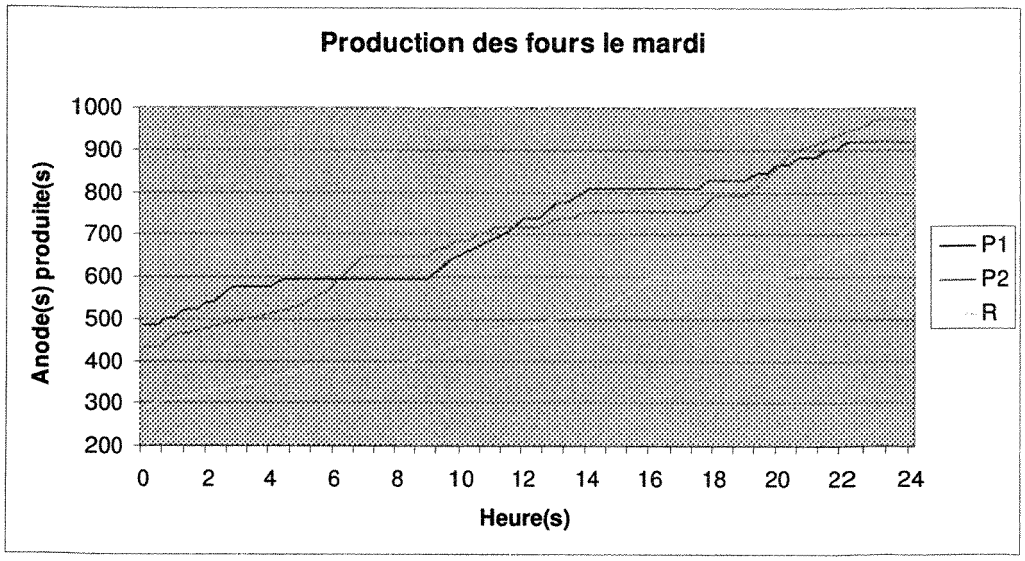

Figure 38 : production des *fours* le mardi

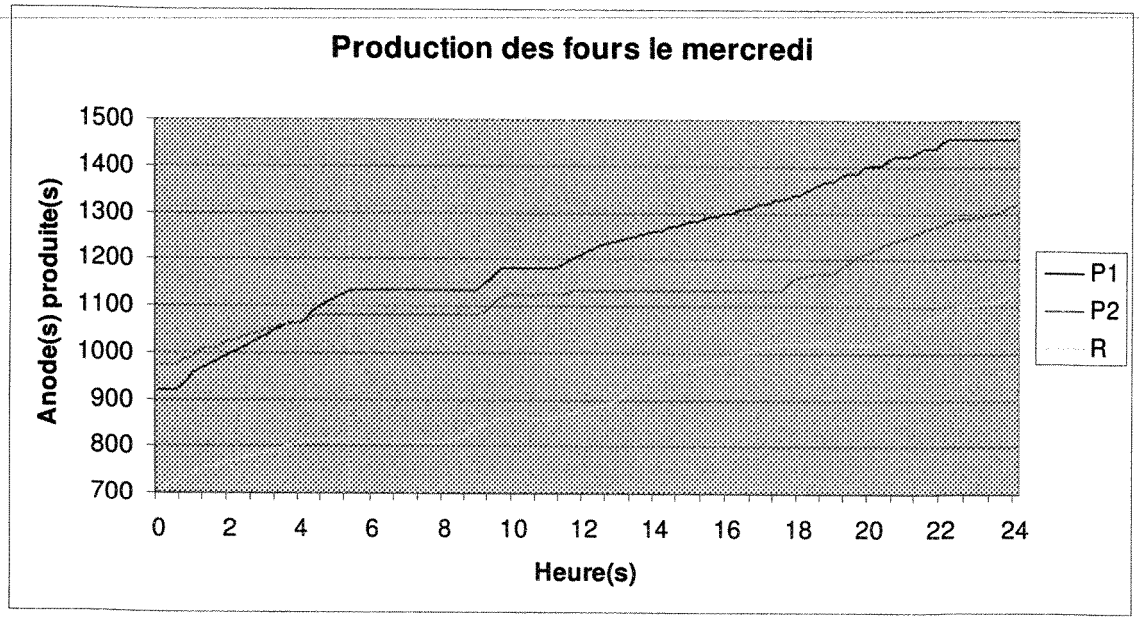

Figure 39 : production des *fours* le mercredi

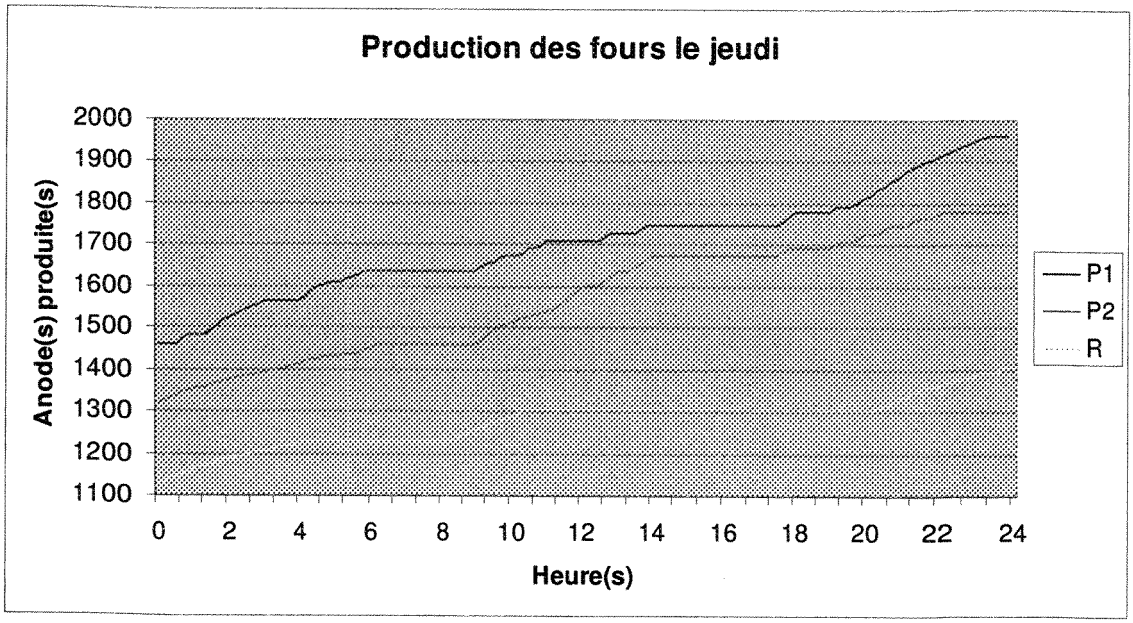

Figure 40 : production des *fours* le jeudi

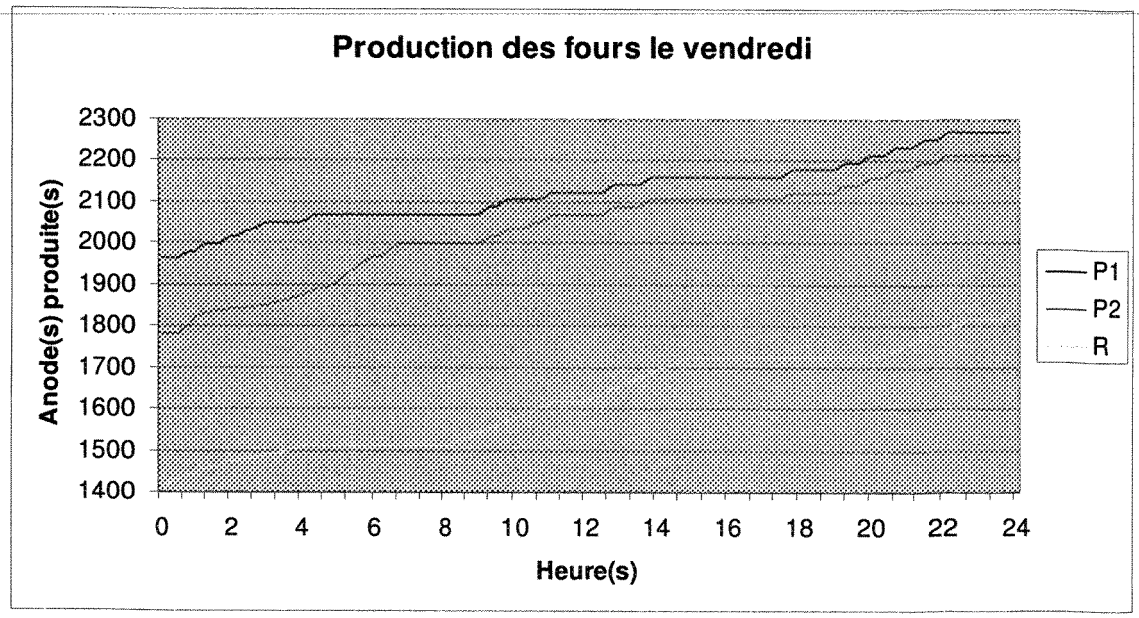

Figure 41 : production des *fours* le vendredi

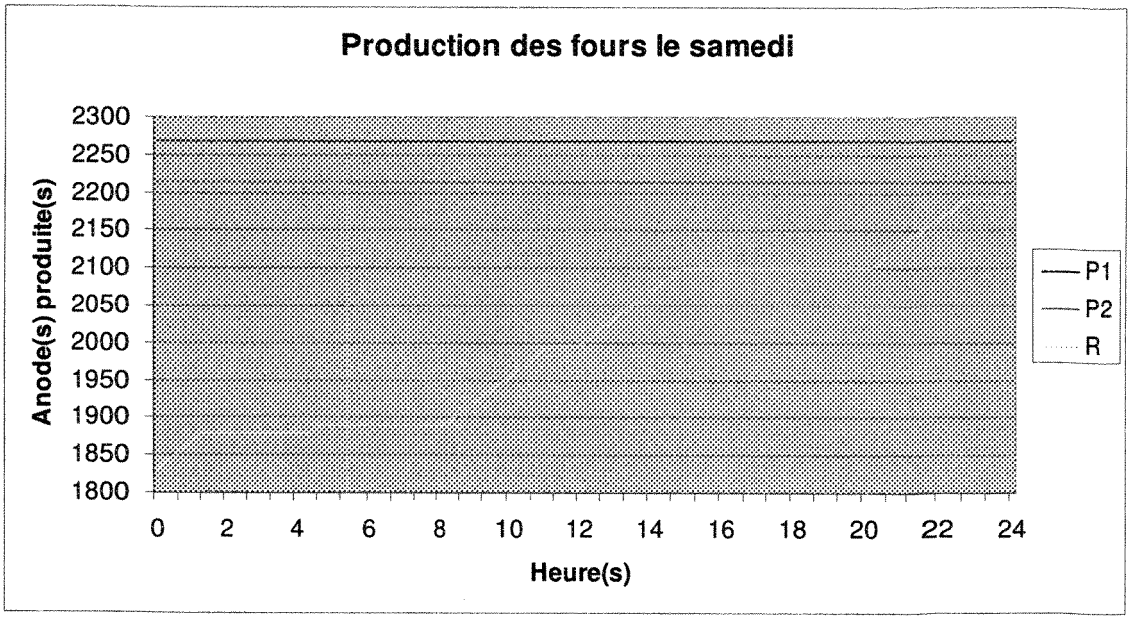

Figure 42 : production des *fours* le samedi

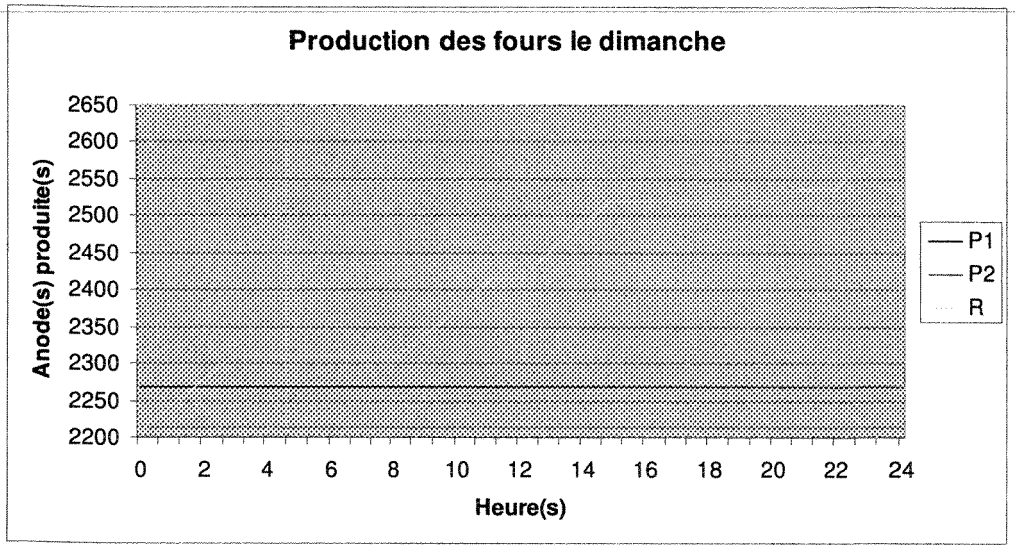

Figure 43 : production des *fours* le dimanche

4.3.1.3 Validation de la tour à pâte et du scellement

Finalement, nous avons aussi vérifié les statistiques de production pour le tour à pâte (c'est-à-dire le *vibrocompacteur/presse*) et le *scellement*, en comparant avec l'historique du système. Là encore, les opérateurs ont observé une corrélation entre le système et le modèle. Les figures ci-dessous illustrent le nombre d'anodes, durant une semaine de production.

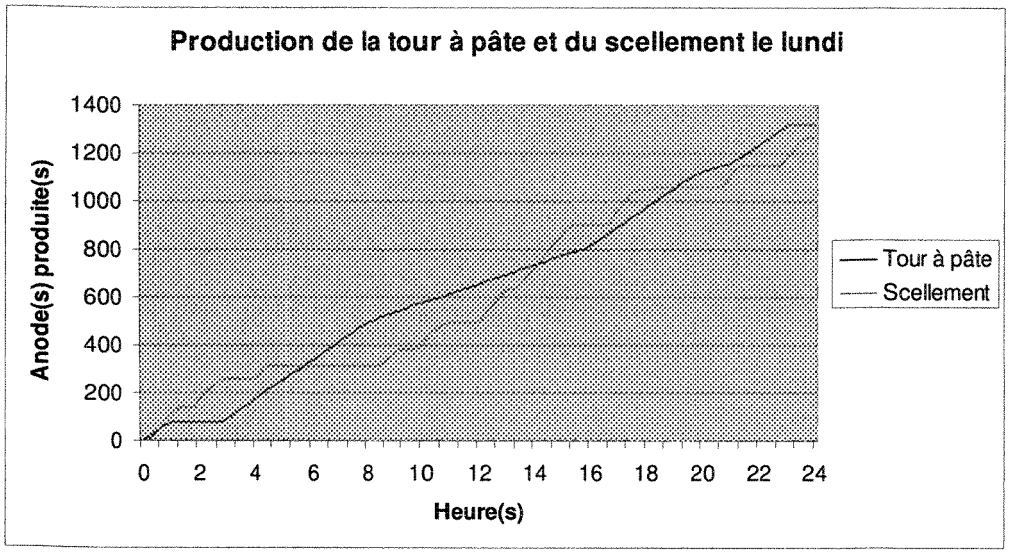

Figure 44 : production de la *tour à pâte* et du *scellement* le lundi

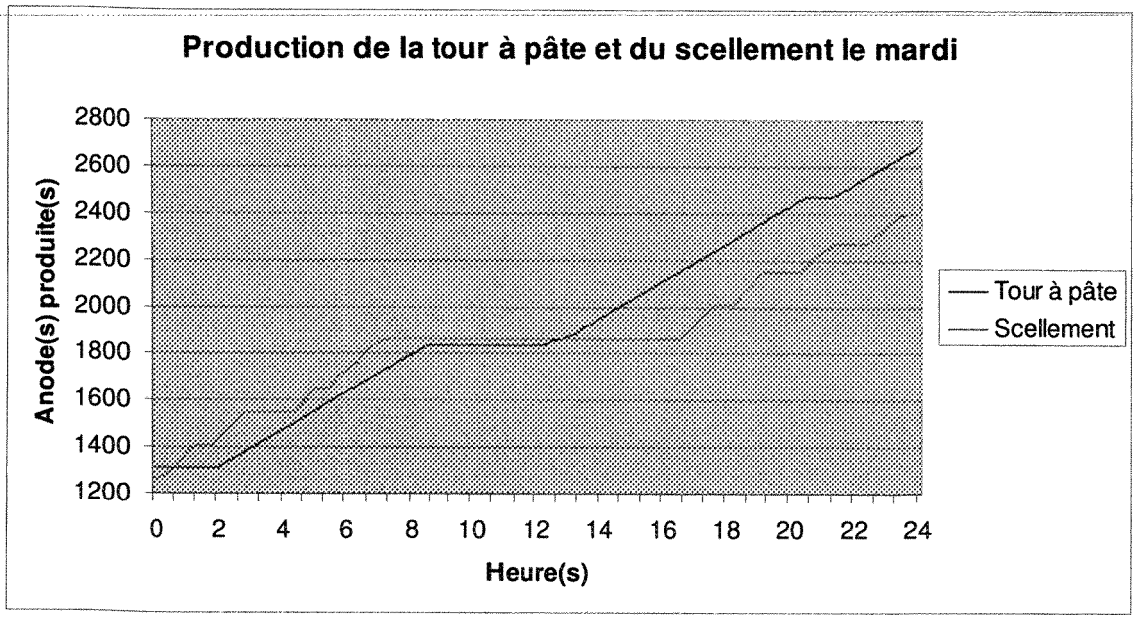

Figure 45 : production de la *tour à pâte* et du *scellement* le mardi

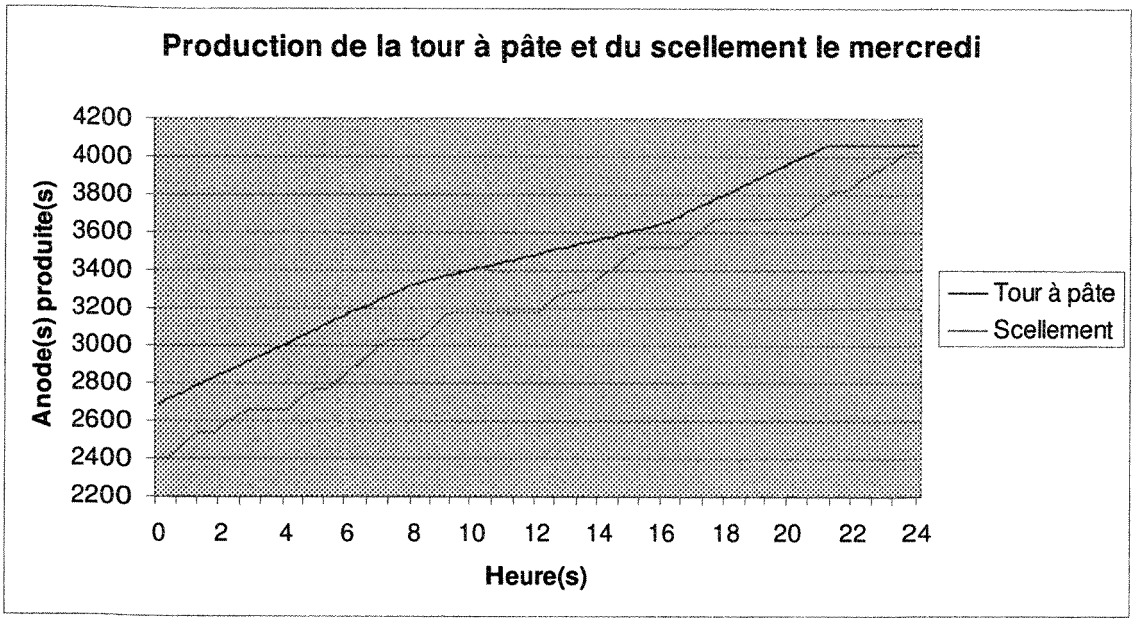

Figure 46 : production de la *tour à pâte* et du *scellement* le mercredi

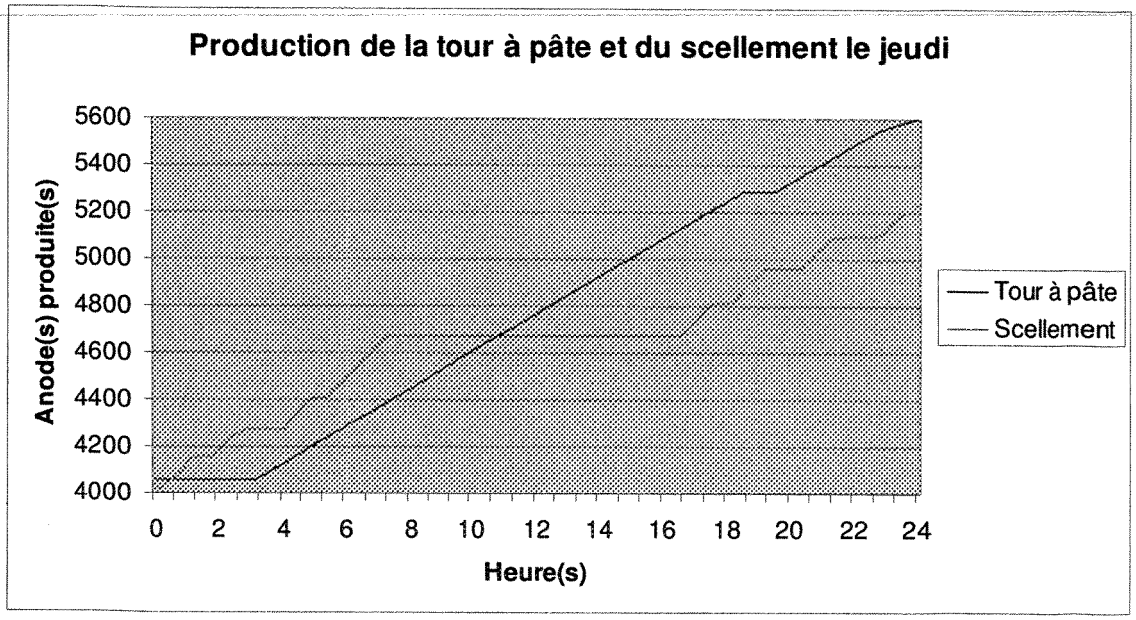

Figure 47 : production de la *tour à pâte* et du *scellement* le jeudi

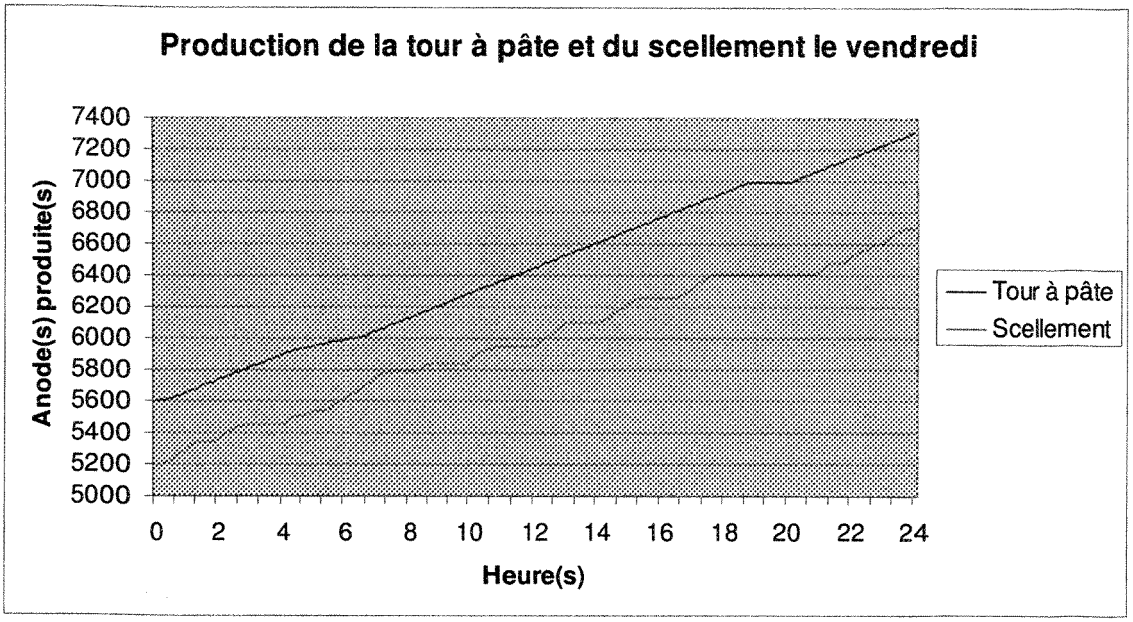

Figure 48 : production de la *tour à pâte* et du *scellement* le vendredi

Notons qu'à partir du samedi 00:00:00 jusqu'au dimanche 23:59:59, tel que stipulé à l'Annexe 1, la *tour à pâte* et l'atelier de *scellement* cessent de fonctionner. Le nombre d'anodes

produites par le *vibrocompacteur* et la *presse* (*tour à pâte*) est de 7310 à ce moment. Quant au nombre d'anodes scellées il est de 6696.

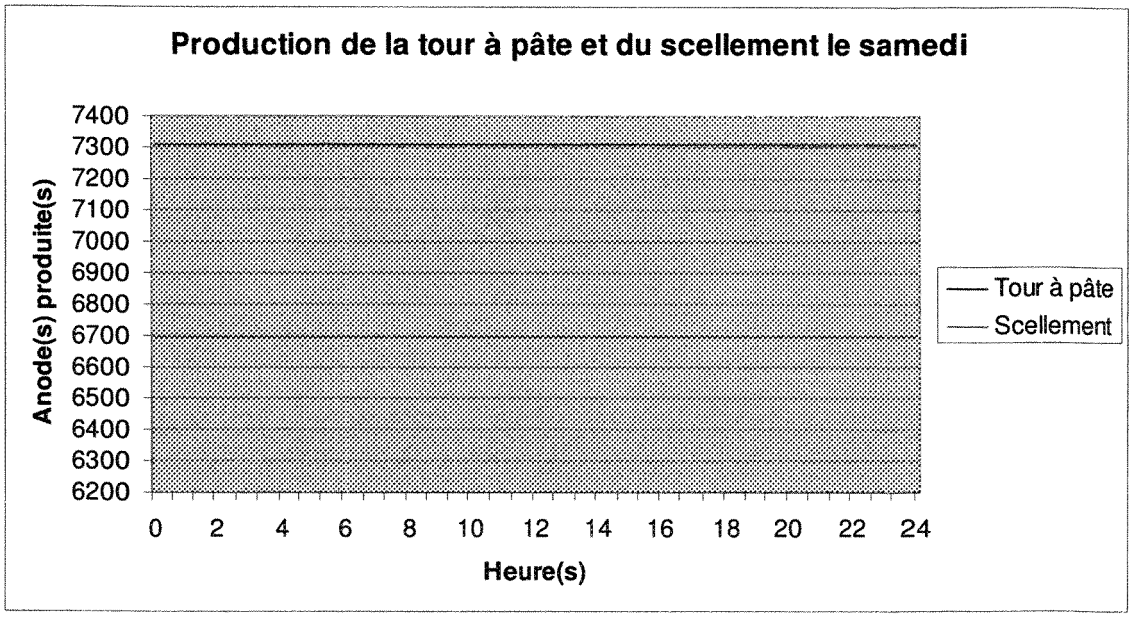

Figure 49 : production de la *tour à pâte* et du *scellement* le samedi

En somme, les comparaisons statistiques des résultats du modèle, avec les mesures du système, ainsi que la validation en surface réalisée en compagnie des spécialistes du procédé, nous permettent de conclure que le modèle représente suffisamment bien la réalité. Par conséquent, il devient possible de l'utiliser comme un outil d'aide à la décision.

4.4 Expérimentation du modèle

Une fois que le modèle du système actuel a été validé, le spécialiste doit concevoir des expériences qui serviront à simuler des scénarios de production alternatifs. Leur choix dépend des objectifs énoncés lors de la formulation du problème. En ce qui nous concerne, cette étape du processus de modélisation a été introduite à la section 3.2. Toutefois, avant de présenter les expériences que nous avons choisies pour la présente étude, nous allons effectuer un petit rappel terminologique et discuter des différents types d'expérimentation.

4.4.1 Facteur et niveau d'une expérience

Lorsque l'on conçoit une expérience, on appelle facteurs « les différentes variables qui sont supposées influencer les résultats produits par le système » NDLR traduit de l'Anglais [Chung, 2004]. Les facteurs peuvent être contrôlés par le spécialiste, de manière à faire varier leurs valeurs, à la fois pour le système actuel et pour le modèle. Par exemple, la vitesse de tables de transfert, la capacité de convoyeurs, ou encore l'horaire d'un poste, sont trois facteurs qui peuvent influés sur les résultats du système que nous modélisons.

De même, pour concevoir des scénarios expérimentaux, il faut pouvoir assigner différentes valeurs aux facteurs du système. Les valeurs qui permettent d'obtenir ces variations sont appelés niveaux [Pidd, 1998]. Par exemple, en augmentant la vitesse de transit d'un convoyeur, sa capacité ou en assignant un nouvel horaire à un poste, on obtient un niveau alternatif, pour chacun des facteurs de l'exemple précédent. En pratique, il existe différentes possibilités d'expérimentations [Law et Kelton, 2000]. La section suivante en présente quelques-unes.

4.4.2 Type d'expérimentation

Deux niveaux pour un facteur

Le type d'expérience le plus simple est celui avec deux niveaux d'alternative pour un même facteur. En temps normal, lorsque le système modélisé est existant, la représentation de la situation actuelle constitue le premier niveau d'alternative. Le second niveau est donc défini par le spécialiste [Chung, 2004]. C'est ce type d'expérience que nous avons choisi pour quatre scénarios d'expérimentation dits extrêmes. Chacun d'eux considère l'état de la ressource comme facteur d'expérimentation. Le premier niveau consiste à assigner l'état de la ressource selon le fonctionnement actuel du système. Quant au second niveau, il implique de désactiver la ressource pendant toute la simulation. Il s'agit d'une configuration hypothétique du système. Nous reviendrons sur ces quatre scénarios à la section 4.4.3. Bien entendu, il est aussi possible d'avoir plus de deux niveaux d'alternative pour un facteur. Toutefois, nous n'avons pas eu recours à ce type d'expérimentation.

Plusieurs niveaux et plusieurs facteurs

Une autre possibilité est de concevoir une expérience avec deux facteurs [Chung, 2004]. Ce type d'expérimentation, plus complexe, consiste à faire varier les deux facteurs à différents niveaux. Dans ce cas, lorsque nous avons un premier facteur A et un second B, le nombre d'alternatives qui sont générées par l'expérience correspond à :

[Nombre d'alternatives] = [Nombre de niveaux facteur A] × [Nombre de niveaux facteur B]

Une expérience avec deux facteurs est assez facile à gérer, tant que le nombre de niveaux n'est pas trop grand. Pour notre part, nous avons deux expériences qui comportent deux facteurs, soient celle du cas optimum et celle des tables de transfert. Ces expériences sont aussi décrites à la section 4.4.3.

Plusieurs niveaux et plusieurs facteurs

Le prochain type d'expérience est celui avec de multiples facteurs d'expérimentations et de multiples niveaux [Chung, 2004]. Le nombre d'alternatives peut alors rapidement exploser, ce qui entraîne des expériences forts complexes. Par exemple, en assumant que nous avons le même nombre de niveaux pour chaque facteur, le nombre d'alternatives croît exponentiellement, soit :

$$[\text{Nombre d'alternatives}] = [\text{Nombre de niveaux des facteurs}]^{[\text{Nombre de facteurs}]}$$

Pour réduire le nombre d'alternatives, le spécialiste n'a d'autre choix que de diminuer le nombre de niveaux ou de facteurs. Pour notre part, la complexité inhérente à ce type d'expérience a fait en sorte que nous ne l'avons pas considéré. Une variante fréquemment utilisée de cette méthode consiste à réduire le nombre de niveaux de chaque facteur à deux. Du même coup, on diminue le nombre d'alternatives. C'est ce qu'on appelle la conception d'expérience factorielle 2^k [Law et Kelton, 2000], où k est le nombre de facteurs. Ce type d'expérience est notamment utilisé pour connaître les effets d'un facteur sur un autre. Un arbre binaire est souvent employé pour recenser l'ensemble des alternatives [Chung, 2004].

4.4.3 Scénarios d'expérimentation

Dans le cadre de notre expérimentation, il est possible de considérer une grande variété de scénarios alternatifs, notamment en raison des possibilités de configuration du modèle. Ainsi, les sept expériences que nous avons réalisées génèrent un total de vingt alternatives. Certaines de ces alternatives sont communes à plusieurs expériences, mentionnons celle de la situation initiale, aussi appelée le cas de référence. En fait, les expériences qui ont été réalisées sont les suivantes :

Expérimentation des cycles de cuisson

L'expérience est basée sur un seul facteur, soit la durée des cycles de cuisson du *four R*. Le premier niveau d'expérimentation est la situation actuelle. À ce moment, les cycles de cuisson sont de 36 heures pour les trois fours. Quant au second niveau, il consiste à assigner des cycles de cuisson de 32 heures au *four R*. Pour ces deux alternatives, la répartition des accidents, c'est-à-dire les *arrêts non-planifiés*, est conforme avec la distribution de probabilité qui leur est associée.

Quatre expérimentations de situation extrême

Chacune de ces quatre expériences requiert un seul facteur. Celui-ci correspond à l'état des quatre ressources suivantes : le *vibrocompacteur / la presse* (celles-ci sont considérés comme une même ressource puisqu'elles sont synchronisées l'une après l'autre), l'*entrepôt*, le four *P1* et l'atelier de *scellement*. Le premier niveau de chaque expérience est la situation actuelle. Celle-ci fait en sorte que les ressources sont désactivées selon les arrêts *planifiés* et *non-planifiés*. Le niveau alternatif consiste à désactiver la ressource pour toute une semaine. Cette dernière alternative permet de connaître l'importance relative de la ressource, puisque leur désactivation permet d'étudier leur impact sur l'ensemble du système.

Expérimentation de la situation optimale

Pour ce cas, nous faisons appel à deux facteurs d'expérimentation. Le premier facteur est l'horaire des arrêts *planifiées* et *non-planifiées*. Le premier niveau du facteur consiste à considérer l'horaire. Quant au second niveau, il consiste à l'ignorer. Cette alternative fait en sorte que les ressources sont fiables à 100%, d'où l'appellation « situation optimale ». Le second facteur

d'expérimentation est le nombre de cycles de cuisson du four *R*. Les deux niveaux du facteur sont des cycles de 36 et 32 heures. L'expérimentation de la situation optimale permet de connaître la capacité de production maximale et ce, pour chacune des ressources. Le nombre d'alternatives est donc le suivant :

[Nombre niveaux pour l'horaire] × [Nombre niveaux des cycles de cuisson] = 2 × 2 = 4

Expérimentation des tables de transfert

Nous utilisons aussi deux facteurs pour cette expérience. Le premier est la vitesse des tables de transfert. Il y a trois niveaux pour le transfert : lent, normal et rapide. Le second facteur d'expérimentation est le nombre de cycles de cuisson du four *R*. Comme nous l'avons mentionné, il y a deux cycles, soit : 36 et 32 heures. Le nombre d'alternatives est donc :

[Nombre niveaux pour le transfert] × [Nombre niveaux des cycles de cuisson] = 3 × 2 = 6

4.5 Analyse des résultats

Au cours de cette section, nous présentons les résultats des sept expériences que nous venons de décrire. Celles-ci consistent à simuler un essai de sept jours. Durant chaque essai, nous avons noté plusieurs mesures de performances du système. Plus particulièrement, le nombre d'anodes entrées et sorties de certaines ressources constituent deux mesures de performance qui ont fait l'objet de notre attention. En fait, les ressources en question sont : le *vibrocompacteur* / la *presse*, l'*entrepôt*, le four *P1*, le four *P2*, le four *R* et l'atelier de *scellement*. Rappelons que l'*entrepôt* récupère et entrepose douze anodes à la fois et que ces deux opérations ne sont pas forcément consécutives (voir la section 3.3.1). Le nombre d'anodes entrées et sorties ne sont donc pas nécessairement les mêmes. C'est pourquoi chaque tableau comporte les deux colonnes suivantes : « *Entrepôt (entrées)* » et « *Entrepôt sorties* ». Par contre, les autres ressources traitent les anodes une à la fois. Le nombre d'anodes entrées et sorties par ces ressources est donc ou bien égal ou bien différent à une anode prêt.

4.5.1 Expérimentation des cycles de cuisson

Comme nous l'avons mentionné, cette expérience consiste à faire varier la durée des cycles de cuisson du *four R*. Le tableau ci-dessous illustre l'impact de cette variation sur le nombre d'anodes entrées et sorties. Les mesures de performances ont été obtenues au terme de sept jours de simulation. Le tableau contient les deux alternatives possibles : le cas de référence, avec des cycles de cuisson de 36 heures, ainsi qu'un cas hypothétique, avec des cycles de 32 heures. Étonnamment, des cycles de cuisson plus courts entraînement une diminution du nombre d'anodes scellées. C'est essentiellement parce qu'il y a plus d'anodes en circulation et donc, plus d'encombrement en amont de l'atelier de *scellement*.

	Vibro-comp. / presse	*Entrepôt (entrées)*	*Entrepôt (sorties)*	*Four P1*	*Four P2*	*Four R*	*Scellement*
[1] Cycles 36 heures	7310	6036	5268	2268	2214	2625	6697
[2] Cycles 32 heures	7310	5952	5148	2268	2214	2955	6656
Variation : Δ = [2] - [1]	0	-84	-120	0	0	330	-41

Tableau 4 : variation des anodes entrées / sorties selon les cycles de cuisson

4.5.2 Quatre expérimentations de situation extrême

Les tableaux ci-dessous contiennent les résultats des quatre expérimentations dites extrêmes. Comme nous avons mentionné, la première alternative de chacune des expériences est le cas de référence. Celui-ci correspond à la seconde ligne de chaque tableau. La deuxième alternative consiste à désactiver une ressource pour la durée de la simulation, c'est-à-dire sept jours. Les valeurs des mesures de performances sont celles obtenues au terme de la simulation.

Le Tableau 5 montre l'impact sur le nombre d'anodes entrées et sorties, lorsque l'on désactive le *vibrocompacteur* et la *presse*. Puisque aucune anode n'est produite, les grues entreposent moins, d'où l'écart notable entre les deux alternatives de la colonne « *Entrepôt (entrées)* ». Par contre, les grues de l'*entrepôt* doivent récupérer d'avantage pour compenser cette absence de production, tel que le stipule la colonne « *Entrepôt (sorties)* ».

	Vibro-comp. / presse	*Entrepôt (entrées)*	*Entrepôt (sorties)*	*Four P1*	*Four P2*	*Four R*	*Scellement*
[1] Cas de référence	7310	6036	5268	2268	2214	2625	6697
[2] Désactivation *Vibro-comp. / presse*	0	3900	10308	2268	2214	2214	6566
Variation : Δ = [2] - [1]	-7310	-2136	5040	0	0	-411	-131

Tableau 5 : variation des anodes entrées / sorties lors de la désactivation du *vibrocompacteur*

Le Tableau 6 montre l'impact sur le nombre d'anodes entrées et sorties, lorsque l'on désactive l'*entrepôt*. Cette alternative crée des blocages en aval du *vibrocompacteur* et des *fours*, parce que les grues n'entreposent plus. Ceci explique pourquoi moins d'anodes sont sorties de certaines ressources. Également, parce les grues ne récupèrent plus, l'atelier de *scellement* n'est pas suffisamment approvisionné, d'où la variation de la colonne « *Scellement* »

	Vibro-comp. / presse	*Entrepôt (entrées)*	*Entrepôt (sorties)*	*Four P1*	*Four P2*	*Four R*	*Scellement*
[1] Cas de référence	7310	6036	5268	2268	2214	2625	6697
[2] Désactivation de l'*entrepôt*	5482	0	0	2268	1692	1486	5399
Variation : Δ = [2] - [1]	-1828	-6036	-5268	0	-522	-1139	-1298

Tableau 6 : variation des anodes entrées / sorties lors de la désactivation de l'*entrepôt*

Le ci-dessous montre l'impact sur le nombre d'anodes entrées et sorties, lorsque l'on désactive le *four P1*. Puisque autant d'anodes sont produites par le *vibrocompacteur* et que le four *P1* n'en traite plus, les grues doivent entreposer d'avantage, d'où l'écart positif de la colonne « *Entrepôt (entrées)* ». De même, pour répondre aux demandes de l'atelier de *scellement*, les grues doivent compenser en récupérant encore plus d'anodes. Ceci explique la variation sous la colonne « *Entrepôt (sorties)* ».

	Vibro-comp. / presse	*Entrepôt (entrées)*	*Entrepôt (sorties)*	*Four P1*	*Four P2*	*Four R*	*Scellement*
[1] Cas de référence	7310	6036	5268	2268	2214	2625	6697
[2] Désactivation du *four P1*	7315	7476	7104	0	2214	2625	7093
Variation : Δ = [2] - [1]	5	1440	1836	-2268	0	0	396

Tableau 7 : variation des anodes entrées / sorties lors de la désactivation du four *P1*

Le tableau ci-dessous montre l'impact sur le nombre d'anodes entrées et sorties, lorsque l'on désactive le *scellement*. Les anodes que l'atelier de *scellement* traitait doivent maintenant être entreposées, d'où l'écart sous la colonne « *Entrepôt (entrées)* ». Également, les grues n'ont plus à récupérer autant d'anodes pour répondre aux demandes du *scellement*. Ceci explique la variation négative de la colonne « *Entrepôt (sorties)* ».

	Vibro-comp. / presse	*Entrepôt (entrées)*	*Entrepôt (sorties)*	*Four P1*	*Four P2*	*Four R*	*Scellement*
[1] Cas de référence	7310	6036	5268	2268	2214	2625	6697
[2] Désactivation du *scellement*	7312	9240	1776	2268	2214	2625	0
Variation : Δ = [2] - [1]	2	3384	-3492	0	0	0	-6697

Tableau 8 : variation des anodes entrées / sorties lors de la désactivation du *scellement*

4.5.3 Expérimentation de la situation optimale

Le Tableau 9 présente une expérience avec deux facteurs, soit : l'horaire des arrêts et la durée des cycles de cuisson. Les valeurs affichées sont le nombre d'anodes entrées et sorties après sept jours de simulation. Pour la première alternative, nous considérons les arrêts *planifiés* et *non-planifiés* (premier facteur), ainsi que des cycles de cuisson de 36 heures (deuxième facteur). Ceci équivaut à simuler le cas de référence. La seconde alternative est aussi effectuée avec des cycles de 36 heures. Toutefois, elle est dite optimale, puisqu'elle ignore les arrêts. Étant donné que toutes les ressources sont fonctionnelles à 100%, le nombre d'anodes produites et scellées

s'accroît, comme le montre les valeurs de la ligne « Variation : Δ = [2] - [1] ». L'impact sur les fours est mineur, car le nombre d'anodes qu'ils traitent est assujetti à l'horaire des trains (voir l'Annexe 1). La troisième alternative considère les arrêts et simule des cycles de cuisson de 32 heures pour le four *R*. Finalement, la dernière alternative est aussi basée sur des cycles de 32 heures. Cependant, elle ne considère pas les arrêts. Que ce soit avec des cycles de 32 ou 36 heures, l'absence des arrêts nous amène au même constat quant à la variation du nombre d'anodes. Mentionnons aussi que même en ignorant les arrêts, une diminution des cycles de cuisson a un impact assez faible, lorsque l'on compare les deuxième et quatrième alternatives.

	Vibro-comp. / presse	*Entrepôt (entrées)*	*Entrepôt (sorties)*	*Four P1*	*Four P2*	*Four R*	*Scellement*
Cycles de cuisson de 36 heures pour le four *R*							
[1] Avec arrêts	7310	6036	5268	2268	2214	2625	6697
[2] Sans arrêt	8672	6300	5564	2304	2214	2670	8193
Variation : Δ = [2] - [1]	1362	264	296	36	0	45	1496
Cycles de cuisson de 32 heures pour le four *R*							
[3] Avec arrêts	7310	5952	5148	2268	2214	2955	6656
[4] Sans arrêt	8672	6252	5484	2304	2214	3030	8067
Variation : Δ = [4] - [3]	1362	300	336	36	0	75	1411

Tableau 9 : variation des anodes entrées / sorties lorsque la situation est optimale

4.5.4 Expérimentation des tables de transfert

Le Tableau 10 présente une expérience avec deux facteurs, soit : la vitesse des tables de transfert et le nombre de cycles de cuisson du four *R*. Comme pour les autres expérimentations, les deux mesures de performance qui nous ont d'avantage intéressées sont : le nombre d'anodes entrées et sorties. Les valeurs affichées sont celles après sept jours de simulation. La première alternative est effectuée avec une table de transfert normale (premier facteur) et des cycles de cuisson de 36 heures (deuxième facteur). Cette alternative est le cas de référence. Les deux alternatives suivantes sont aussi réalisées avec des cycles de 36 heures mais cette fois-ci, la

vitesse de transfert de leurs tables passe respectivement à lente et à rapide. La vitesse semble être directement proportionnelle avec le nombre d'anodes produites et scellées. Cette conclusion demeure vraie, même lorsque l'on diminue la durée des cycles de cuisson à 32 heures pour les trois dernières alternatives. Nous avons également effectué une petite analyse de sensibilité, avec trois tables situées entre les *fours* et le *scellement* (voir la Figure 23). Ainsi, lorsque la table T5, la T4 ou la T3 est mise en vitesse rapide, nous obtenons 6778, 6675 et 6675 anodes scellées respectivement. Il semble donc que la vitesse de la table T5 joue un rôle prédominant.

	Vibro-comp. / presse	*Entrepôt (entrées)*	*Entrepôt (sorties)*	*Four P1*	*Four P2*	*Four R*	*Scellement*
Cycles de cuisson de 36 heures pour le four *R*							
[1] Normale	7310	6036	5268	2268	2214	2625	6697
[2] Lente	7309	5664	4800	2268	2214	2625	6609
[3] Rapide	7328	6504	5736	2268	2214	2625	6717
Cycles de cuisson de 32 heures pour le four *R*							
[4] Normale	7310	5952	5148	2268	2214	2955	6656
[5] Lente	7298	5616	4728	2268	2214	2955	6558
[6] Rapide	7340	6180	5436	2268	2214	2955	6750

Tableau 10 : variation des anodes entrées / sorties en fonction du taux de transfert

4.6 Conclusions

À la lumière des expérimentations que nous avons effectuées, plusieurs conclusions peuvent être émises. Ces conclusions font l'objet de recommandations qui permettent de répondre aux principales interrogations des spécialistes du système.

D'abord, on observe une baisse minime du nombre d'anodes scellées avec des cycles de cuisson plus courts (Tableau 4). Puisque la durée des cycles du four *R* n'a pas un impact significatif, une modification de ce paramètre n'est pas recommandée.

Deuxièmement, comme on l'a vu à la section 4.5.2, outre l'arrêt du *scellement* (Tableau 8), c'est l'arrêt des grues de l'*entrepôt* qui entraîne la plus forte diminution d'anodes scellées (Tableau 6). Il importe donc de minimiser les arrêts planifiés et non-planifiés de ces deux ressources autant que possible.

L'arrêt du four *P1* provoque un accroissement du nombre d'anodes scellées et une chute drastique du nombre d'anodes cuites entreposées (Tableau 7). L'arrêt d'un *four* n'est donc pas souhaitable, à moins que l'on ait suffisamment d'anodes cuites entreposées.

Comme l'a montré l'expérimentation de la situation optimale à la section Expérimentation de la situation optimale 4.5.3, la diminution du nombre d'arrêts entraîne une augmentation significative du nombre d'anodes scellées (Tableau 9). Il importe donc de réduire leur nombre autant que possible.

Finalement, comme vous pouvez le constater au Tableau 10, une augmentation de la vitesse des tables de transfert est souhaitable, pour accroître le nombre d'anodes scellées et ce, peu importe la durée des cycles de cuisson du four *R*.

CONCLUSION

Conclusion

Objectif de la recherche

Notre travail de recherche consistait à exposer une méthode de conception d'outil d'aide à la décision appliquée à un procédé industriel. L'objectif principal du mémoire était de montrer comment utiliser cette méthode, pour concevoir un modèle, c'est-à dire un outil d'aide à la décision, qui intègre des connaissances de diverses natures, afin de résoudre des problématiques non-structurées et ce, dans un contexte d'amélioration d'un système [Turban et Aronson, 2000]. En plus de méthodes de modélisation traditionnelles [Banks et al, 1998] [Hoover et Perry, 1990] [Law et Kelton, 2000], nous avons eu recours à des notions d'intelligence artificielle, notamment aux systèmes experts [Giarratano et Riley, 1998], afin de représenter les différentes formes de connaissances appliquées par les opérateurs et les automates du procédé étudié.

À propos de la modélisation et de la prise de décision

Nous avons d'abord effectué une revue de la littérature, afin de mettre en contexte le sujet. Nous avons précisé le lien entre les notions de prise de décision et de modèle [Simon, 1977] [Pidd, 1998]. Nous avons discuté des différentes manières d'étudier un système [Turban et Aronson, 2000], telle la conception de modèles symboliques, aussi appelée modélisation [Hoover et Perry, 1990]. Nous avons mentionné que les modèles permettent d'effectuer des simulations, afin de soutenir la prise de décision [Pidd, 1998]. Nous avons également abordé les avantages et les désavantages de la modélisation. Il a été question des étapes du processus de modélisation [Banks et al, 1998] [Law et Kelton, 2000]. Nous avons aussi discuté des différents types de modèles [Hoover et Perry, 1990] [Law et Kelton, 2000].

Ensuite, nous avons mentionné que nous nous intéressons plus particulièrement aux modèles à événements discrets [Law et Kelton, 2000]. Nous avons discuté de la terminologie propre à ces modèles (entité, ressource, système, attribut, classe, file, activité, événement, etc.), de leur gestion de l'avancement du temps (incrément fixe et événement suivant), de quatre approches de conception de leur contrôleur (planification d'événements, recherche d'activités, méthode basée

sur des processus, méthode en trois phases) et de leur architecture [Balci, 1988] [Banks et al, 1998] [Pidd, 1998].

Nous avons également abordé la notion de modèle orienté objet, aussi appelé OOS (« Object Oriented Simulation ») [Banks et al, 1998]. Nous avons discuté des caractéristiques de ces modèles et de leur organisation sous la forme de cadre ou de cadriciel. Nous avons aussi introduit la notion de composants logiciels [Szyperski, 1999], leurs différents formalismes et leur usage en modélisation [Pidd et al, 1999]. Nous avons présenté DEVS (« Discrete Event System Specification ») [Zeigler et Sarjoughian, 2005], une spécification qui permet de formaliser des systèmes à événements discrets à partir de composants. Nous avons énoncé les avantages de DEVS et quelques-unes de ses applications.

Finalement, nous avons abordé les systèmes de modélisation interactive aussi appelés VIMS (« Visual Interactive Modeling System ») [Pidd, 1998]. Nous avons décrit leur fonctionnement, leurs caractéristiques et nous avons présenté un VIMS nommé Arena. Nous avons également comparé l'approche de modélisation classique [Law et Kelton, 2000], dont il a été question plus haut, avec les VIMS.

Système expert et prise de décision

La deuxième partie du travail de recherche traitait plus particulièrement de la notion de coquille de développement de système expert [Friedman-Hill, 2003], car notre approche de modélisation est en partie basée sur celle-ci. Nous avons introduit la notion d'intelligence artificielle (IA) et mentionné que les systèmes experts constituent un de leur domaine d'application. Nous avons brièvement décrit le principe et les composantes d'un système expert, mentionnons : la base de connaissances, les faits et le moteur d'inférence. Nous avons montré qu'il est possible d'utiliser les règles de production d'un système expert [Giarratano et Riley, 1998], pour modéliser et traiter la connaissance des spécialistes d'un procédé.

La suite du second chapitre avait pour but de décrire plus en détail les systèmes experts. Ainsi, nous avons introduit les coquilles de développement et l'architecture élémentaire d'un

système expert [Friedman-Hill, 2003] [Giarratano et Riley, 1998]. Il a aussi été question de la base de connaissances d'un système expert et de divers concepts connexes tels : les formalismes de représentation de la connaissance (logique prédicative, réseau sémantique, cadre, règle de production), le réseau d'inférence [Gonzalez et Dankel, 1993], ainsi que l'acquisition et la formalisation des règles de production [Friedman-Hill, 2003].

Nous avons ensuite présenté la notion de mémoire de travail ou base de faits. Nous avons discuté de l'organisation des faits de deux coquilles de développement de systèmes experts (CLIPS et Jess) et des différents types de faits [Giarratano, 2002] [Jess 7.0 Manual]. Nous avons aussi présenté le moteur d'inférence d'un système expert et deux catégories de raisonnement : chaînage avant et arrière [Gonzalez et Dankel, 1993].

L'assortisseur de conditions entre les règles et les faits [Friedman-Hill, 2003] a fait l'objet d'une bonne partie de la discussion suivante. En effet, notre approche de modélisation est fondée sur cette notion. Nous avons donc présenté un algorithme naïf qui permet de mettre en œuvre l'assortiment des conditions. Ensuite, nous avons discuté de l'algorithme de Rete [Forgy, 1982] et expliqué en quoi celui-ci constitue une amélioration de la solution naïve. Nous avons exposé les concepts d'associations de conditions et d'associations partielles qui permettent de construire un réseau de conditions et de branchements [Friedman-Hill, 2003] [Giarratano et Riley, 1998]. Nous avons terminé cette section en analysant la complexité de l'algorithme de Rete.

Nous avons aussi présenté l'agenda des activations. Celui-ci constitue une liste de règles issues du processus de résolution des conflits entre les activations [Giarratano, 2002]. Deux stratégies de résolution des conflits sont d'ailleurs exposées, soient celles en profondeur et en largeur [Jess 7.0 Manual]. Nous concluons le second chapitre en décrivant le rôle du moteur d'exécution, de l'interface utilisateur et de la composante d'acquisition d'un système expert [Friedman-Hill, 2003].

Application à un cas réel de l'approche de modélisation avec système expert

Afin de vérifier la viabilité et la validité de notre approche de modélisation, nous avons décidé de l'appliquer à un cas réel. Ainsi, nous avons conçu puis mis en œuvre un modèle qui permet de simuler le cycle de vie des anodes d'une aluminerie [Beghein et al., 2003] [Totten et Mackenzie, 2003]. Au cours du troisième chapitre, nous avons d'abord présenté les quatre étapes de ce procédé : la formation des anodes, la manutention et l'entreposage, le scellement d'anodes, ainsi que la production du métal. [Beghein et al., 2003] [Poirier, 1997]. Puis, nous avons formulé le problème et les objectifs qui nous ont amené à étudier ce système. Nous avons discuté des mesures de performances qui nous permettent d'évaluer l'efficacité des scénarios simulés.

Dans un deuxième temps, il a aussi été question de l'ensemble des informations que nous avons collectées auprès des spécialistes du système, principalement en ce qui a trait à la configuration des ressources du système et aux procédures d'opération : l'alimentation et la récupération des fours, l'alimentation du scellement, l'entreposage et la récupération des grues, les règles de routage génériques et la gestion des arrêts planifiés.

Ensuite, puisqu'il est souhaitable de modéliser les événements probabilistes du système en tant que variables aléatoires [Baillargeon, 1990] [Banks et al, 1998], nous avons étudié quelques approches d'intégration de ces événements au sein du modèle. Nous avons d'abord discuté des variables du système qui sont sujet à un comportement probabiliste. Ensuite, les méthodes qui nous permettent de modéliser ces variables ont été décrites : les distributions de probabilité [Law et Kelton, 2000], le pourcentage d'efficacité, la moyenne échantillonnale et l'approche minimaliste. Finalement, nous avons comparé ces méthodes entre elles, pour justifier celle qui a été retenue pour le modèle.

Nous avons présenté le mécanisme de gestion des événements discrets du modèle. Nous avons d'abord mentionné que la stratégie d'avancement du temps avec incrément fixe [Law et Kelton, 2000], nous apparaissait la plus appropriée, car elle permet de traiter facilement les événements conditionnels [Pidd, 1998]. Nous avons aussi fondé le contrôleur sur une approche

avec recherche d'activité [Balci, 1988], parce qu'elle permettait de représenter les deux types d'événements du modèle : planifiables et conditionnels. Nous avons noté que le fonctionnement de l'approche avec recherche d'activité s'apparente à celui d'un système à base de règles de production [Banks et al, 1998]. Cette approche s'avère cependant moins efficace que d'autres, mentionnons notamment la planification d'événements [Pidd, 1998]. Pour améliorer sa performance, nous lui avons donc adjoint un système de production basé sur l'algorithme de Rete [Forgy, 1982].

La quatrième partie du chapitre présentait la phase de conception et de mise en œuvre du modèle. Nous avons montré que notre approche de modélisation permet d'adresser les contraintes et les pratiques établies de conception d'un système d'information [McConnell, 1996] [McConnell, 2004]. Dans un premier temps, nous avons présenté le diagramme UML des classes du modèle [Fowler, 2004]. Nous avons ensuite brièvement rappelé qu'un système à base de règles est une avenue intéressante pour concevoir notre modèle [Jeong, 2000]. Nous avons mentionné quatre conditions générales pour recourir à de tels systèmes [Rudolph, 2003]. Nous avons montré que ces conditions étaient rencontrées pour notre modèle. Nous avons aussi démontré que les règles de production permettent de représenter les méthodes empiriques du système modélisé. Nous avons affirmé que ces règles rendent le modèle plus flexible et plus facile à maintenir. Du même coup, il devient possible de raffiner graduellement les hypothèses de modélisation [Law et Kelton, 2000]. Nous avons aussi vu qu'un mécanisme d'inférence avec chaînage avant était souhaitable pour notre problème, parce que le nombre de faits est limité et que la plupart contribuent au processus de résolution [Gonzalez et Dankel, 1993].

Au cours de la dernière partie du chapitre, nous avons présenté les deux prototypes qui ont été conçus. Le premier s'appuie sur un système expert élémentaire intégré au modèle. Nous avons brièvement décrit la mise en œuvre de la base de connaissances, de la mémoire de travail et du moteur d'inférence [Giarratano et Riley, 1998]. Nous avons conclu que ce prototype était peu flexible et difficile à maintenir. Le second modèle, qui intègre la coquille de développement Jess

[Jess], a ensuite été présenté. Nous avons discuté du mécanisme d'inférence avec Jess (« chaînage avant »), de la stratégie de résolution de conflits (« en profondeur ») [Friedman-Hill, 2003] et de quelques approches d'optimisation des règles, notamment en limitant le nombre d'associations partielles et leur modification [Giarratano et Riley, 1998]. Il a aussi été question des « shadows facts ». Ceux-ci permettent d'établir une connexion entre les faits de la mémoire de travail et les Beans d'un programme Java qui référence Jess [Jess 7.0 Manual]. Nous avons aussi présenté la notion de module, leur rôle, leurs avantages, ainsi que deux modules particuliers : le module courant et le module cible [Friedman-Hill, 2003]. Nous avons complété en présentant le point d'entrée du programme Jess, à savoir l'association entre le modèle Java et la coquille.

Validation de l'approche de modélisation et prise de décision

Au cours du dernier chapitre du mémoire, nous avons démontré que notre approche permettait de concevoir des modèles valides qui peuvent être utilisés dans un contexte de prise de décision. Nous avons dans un premier temps rappelé en quoi consiste l'étape de validation [Law et Kelton, 2000]. Ensuite, nous avons énuméré quelques aspects qui peuvent avoir un impact sur la validité de notre modèle [Chung, 2004], mentionnons : les hypothèses de modélisation, les simplifications, les omissions ainsi que les limitations propres au spécialiste, aux outils de modélisation et aux données du système.

Dans un deuxième temps, nous avons discuté de la notion de validation en surface [Chung, 2004] [Law et Kelton, 2000] et de la démarche entreprise pour l'accomplir dans le cadre de notre modèle. Nous avons ensuite abordé la validation statistique du modèle. Pour ce faire, nous avons simulé le cas de référence et l'avons comparé avec les données fournies par les spécialistes du système. Les mesures de performances qui ont été utilisées à des fins de validations sont : le nombre d'anodes produites par le vibrocompacteur, la presse et les fours, ainsi que le nombre d'anodes scellées. Nous avons calculé l'écart entre les valeurs hebdomadaires du modèle et du système. Cet écart s'est avéré raisonnable. De plus, l'évolution quotidienne des mesures de performance du modèle a été validée par les spécialistes du système.

La quatrième partie du chapitre visait à concevoir des expériences qui permettent de simuler des scénarios de production alternatifs. Nous avons d'abord défini quelques termes propres à l'expérimentation, à savoir la notion de facteur et de niveau d'expérience, ainsi que les différents types d'expérimentation [Chung, 2004]. Puis, nous avons décrit les quatre expériences que nous avons conçues, afin de répondre aux objectifs de l'étude. Nous avons présenté les résultats obtenus et analysé l'impact des différents facteurs sur le système. Nous terminons en énumérant les principales recommandations émises à la lumière de l'analyse des résultats.

Discussion

Nous concluons le travail de recherche en affirmant que nous avons démontré que notre approche permet de modéliser un procédé industriel, dont le fonctionnement repose sur des connaissances de natures diverses. Nous avons aussi démontré que le modèle issu de cette approche peut être utilisé comme un outil d'aide à la décision. L'objectif principal du travail de recherche a donc été atteint. Notre démonstration s'est effectuée en deux temps. Nous avons d'abord vu que les règles de production, de la coquille de développement de système expert Jess, permettent de représenter la connaissance appliquée par les opérateurs et les automates du procédé. Nous avons ensuite montré que le modèle peut être utilisé comme outil d'aide à la décision, puisqu'il représente suffisamment bien le système réel. Pour ce faire, nous avons effectué la validation en surface, en compagnie des spécialistes du système, ainsi que la validation statistique du modèle. Nous avons ainsi pu établir une corrélation entre celui-ci et le système réel.

Toutefois, on peut se questionner sur la viabilité de notre méthode de modélisation, lorsqu'elle est appliquée à d'autres systèmes que le cycle de vie des anodes d'une aluminerie [Totten et Mackenzie, 2003]. Comme on l'a mentionné à la section 1.3, la modélisation est à prime abord un processus généralement coûteux en temps et en ressources [Turban et Aronson, 2000]. De plus, tel qu'il a été dit à la section 2.2, les systèmes de production, tout comme n'importe quel domaine de l'IA, impliquent des spécialistes et une expertise particulière qui risque de complexifier

la conception du modèle [Gonzalez et Dankel, 1993]. D'où la nécessité de se questionner sur la pertinence d'intégrer un système de production.

Nous croyons à ce titre que les modèles qui doivent être livrés rapidement et dont l'usage est unique ne conviennent pas à notre approche. Par contre, lorsqu'il faut concevoir un modèle qui nécessite des ajustements continuels, qu'il est utilisé sur une base régulière et que l'échéancier le permet, notre méthode devrait être envisagée. La présence de connaissance heuristique appliquée au système modélisé [Friedman-Hill, 2003] [Gonzalez et Dankel, 1993] constitue également un critère fondamental. Comme nous l'avons vu au chapitre deux et trois, le savoir-faire, appliqué par les opérateurs et les automates du système, a pu être capté en grand partie grâce à notre approche de modélisation. Du même coup, nous avons pu obtenir un niveau de précision tout à fait raisonnable, comme nous avons vu au chapitre quatre. Notons que les critères énoncés par [Rudolph, 2003], dont il a été question à la section 3.4.2, devraient aussi être considérés, afin de valider la convenance à opter pour notre méthode de modélisation.

Mentionnons que ce travail s'est effectué dans le cadre des recherches de Sylvain Boivin, professeur au Département d'informatique et de mathématique de l'Université du Québec à Chicoutimi. Ainsi, un des objectifs secondaires de l'approche proposée était d'assister M. Boivin dans le cadre des projets de modélisation qu'il a à réaliser. Le cas étudié au cours des chapitres trois et quatre constitue d'ailleurs un des projets de M. Boivin.

D'un point de vue plus personnel, ce travail s'est avéré une expérience tout à fait enrichissante. J'ai ainsi eu l'occasion de m'initier à la recherche appliquée au monde de la modélisation, simulation et aux systèmes experts. De plus, je retire une grande satisfaction d'avoir réussi à accomplir mon projet de maîtrise, malgré les nombreuses embuches et mon emploi qui ne me permettaient pas toujours de rédiger à la vitesse que j'aurais souhaité. Bien que je n'applique pas à tous les jours dans mon travail les connaissances que j'ai acquises, je pense que ce mémoire m'a permis d'aiguiser mon sens de l'analyse et de la rigueur. De plus, comme le disait feu mon grand-père à mon propre père, je crois que « le travail d'un homme n'est jamais perdu. »

BIBLIOGRAPHIE

Bibliographie

[Anderson et al, 1987] John R. Anderson, Albert T. Corbett, Brian J. Reiser, *Essential Lisp*, Addison-Wesley, 1987

[Arena1] Rockwell Automation, *Arena 9.0*, http://www.arenasimulation.com/

[Arena2] Wikipedia: The Free Encyclopedia, *Rockwell Arena*, http://en.wikipedia.org/wiki/Rockwell_Arena

[Arena3] Rockwell Automation, *Arena Basic Edition: Technical Data*, http://www.arenasimulation.com/pdf/ARENAB-TD001B-EN-P.pdf

[Baillargeon, 1990] Gérard Baillargeon, Méthodes statistiques de l'ingénieur Volume 1, 2ième édition, Les éditions SMG, 1990

[Banks et al, 1998] Jerry Banks et al, Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice, Wiley-Interscience, 1998

[Balci, 1988] Osman Balci, *The implementation of four conceptual frameworks for simulation modeling in high-level languages*, ACM Press New York, 1988

[Batory et O'Maley, 1992] Don Batory et Sean O'Maley, The design and implementation of hierarchical software systems with reusable components, ACM Transactions on Software Engineering. And Methodology, 1992

[Beghein et al., 2003] Philippe Beghein, Roy Cahill, Wayne Hale, Warren Haupin, Richard Jeltsch, Halvor Kvande, Andrew Macleod, Vinko Potocnick, Alton Tabereaux, Gary Tarcy, Mark Taylor, Barry Welch, Nick Zienkowicz, *TMS Industrial Aluminum Electrolysis, Theory & Practice of Primary Aluminum Production*, Quebec, QC Canada, 2003

[Bratley et al, 1987] Paul Bratley, Bennet L. Fox, Linus E. Schrage, *A Guide to Simulation Second Edition*, Springer-Verlag, 1987

[Buchanan et Shortliffe, 1984] Bruce G. Buchanan, Edward Hance Shortliffe, *Rule-based expert systems: the MYCIN experiments of the Stanford heuristic programming project*, Addison-Wesley, 1984

[Cadriciel] Wikipedia : projet d'encyclopédie libre et gratuite, *Cadre d'application (Cadriciel)*, http://fr.wikipedia.org/wiki/Cadriciel

[Champeaux et al, 1993] Dennis de Champeaux, Douglas Lea et Penelope Faure, *Object-Oriented System Development*, Addison-Wesley, 1993

[Chapman, 1998] Davis Chapman, *Visual C++ 6, Formation en 21 jours*, Simon & Schuster Macmillan, 1998

[Chase, 2001] Nicholas Chase, *XML et Java,* Campus Press, 2001

[Christensen, 1986] Howard B. Christensen, *La statistique : démarche pédagogique programmée*, traduit par François Gagné et Robert Proulx, Gaëtan Morin éditeur, 1986

[Chung, 2004] *Simulation Modeling Handbook, A Practical Approach*, Christopher A. Chung, CRC Press, 2004

[CLIPS] NASA's Artificial Intelligence Section: Chris Culbert, Brian Donnell, Frank Lopez, Chris Ortiz, Bebe Ly, Gary Riley, Robert Savely et al., *C Language Integrated Production System (CLIPS) v 6.2.3*, http://www.ghg.net/clips/CLIPS.html

[Clocksin et Mellish, 1987] William F. Clocksin, Christopher.S. Mellish, *Programming in Prolog (Third revision and extended edition)*, Springer-Verlag, 1987

[Code source libre] Wikipedia : projet d'encyclopédie libre et gratuite, *Open source : code source libre*, http://fr.wikipedia.org/wiki/Open_Source

[COM] Microsoft Corporation, *COM: Component Object Model Technologies*, http://www.microsoft.com/com/, 2006

[CORBA] Object Management Group (OMG), CORBA: Common Object Resquest Broker Architecture, *CORBA Basics*, http://www.omg.org/gettingstarted/corbafaq.htm, 2006

[Cormen et al., 1994] Thomas H. Cormen, Charles Leiserson, Ronald Rivest, *Introduction à l'algorithmique,* Dunod, 1994

[Density Function] Wikipedia: The Free Encyclopedia, *Probability Density Function*, http://en.wikipedia.org/wiki/Density_function

[Dictionnaire terminologique] Le grand dictionnaire terminologique de L'Office québécois de la langue française, http://w3.granddictionnaire.com/

[Donaldson et Siegel, 2000] Scott E. Donaldson et Stanley G. Siegel, *Successful Software Development 2nd Edition*, Prentice Hall PTR, 2000

[Dréo et Siarry, 2003] Johann Dréo et Patrick Siarry, Métaheuristiques pour l'optimisation difficile, Paris : Eyrolles, 2003

[Durkin, 1994] John Durkin, *Expert Systems, Design and Development*, Prentice-Hall, 1994

[Eclipse] Eclipse Foundation, *Eclipse: an open development platform*, http://www.eclipse.org/

[Eclipse JDT] Eclipse Foundation, *Eclipse Java Development Tools (JDT) Subproject*, http://www.eclipse.org/jdt/

[Eclipse software] Wikipedia: The Free Encyclopedia, *Eclipse (software)*, http://en.wikipedia.org/wiki/Eclipse_(computing)

[eUML2] Soyatec, *eUML2 v.2.3.1*, http://www.soyatec.com/main.php

[EDI] Wikipedia : projet d'encyclopédie libre et gratuite, *Environnement de développement intégré (EDI)*, http://fr.wikipedia.org/wiki/Environnement_de_développement_intégré

[Elmasri et Navathe, 1994] Ramez Elmasri, Shamkant B. Navathe, *Fundamentals of Database Systems, Second Edition*, Benjamin/Cummings Publishing, 1994

[Extreme Programming] Wikipedia : projet d'encyclopédie libre et gratuite, *Extreme Programming*, http://fr.wikipedia.org/wiki/Extreme_programming

[Feigenbaum, 1982] Edward A. Feigenbaum, *Knowledge Engineering in the 1980s*, Department of Computer Science, Stanford University, Sanford, CA, 1982

[First-order logic] Wikipedia: The Free Encyclopedia, *First-order logic*, http://en.wikipedia.org/wiki/First-order_logic

[Fischwick, 1992] Paul A. Fischwick, *An Integrated Approach to System Modeling Using a Synthesis of Artificial Intelligence, Software Engineering and Simulation Methodologies*, ACM Transactions on Modeling and Computer Simulation, Vol. 2, No. 4, p. 307-330, 1992

[Fowler, 2004] Martin Fowler, *UML Distilled, A Brief Guide to the Standard Object Modeling Language (Third Edition)*, Addison-Wesley, 2004

[Forgy, 1982] Charles L. Forgy, Rete : *A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem*, Artificial Intelligence, Vol. 19, p. 17-37, 1982

[Friedman-Hill, 2003] Ernest J. Friedman-Hill, *Jess in Action: Rule-Based Systems in Java*, Manning Publications Co., 2003

[Fukunari et al, 1998] Miki Fukunari, Yu-liang Chi et Philip M. Wolfe, *JavaBean-Based Simulation with a Decision Making Bean*, Proceedings of the 1998 Winter Simulation Conference, p.1699-1702, 1998

[Gardarin, 2001] Georges Gardarin, *Bases de données, objet et relationnel, troisième édition*, Eyrolles, 2001

[Gaussian Function] Wikipedia: The Free Encyclopedia, *Gaussian function*, http://en.wikipedia.org/wiki/Gaussian_function

[Gilbert et McCarthy, 2001] Object-Oriented Design in Java, Stephen Gilbert et Bill McCarty, SAMS, 1998

[Giarratano, 2002] Joseph C. Giarratano, *CLIPS v 6.2.0 User's Guide*, http://www.ghg.net/clips/download/documentation/, 2002

[Giarratano et Riley, 1998] Joseph C. Giarratano, Gary Riley, *Expert Systems, Principles and Programming, Third Edition*, PWS Publishing Company, 1998

[Gonzalez et Dankel, 1993] Avelino J. Gonzalez, Douglas D. Dankel, *The Engineering of Knowledge-Based Systems Theory and Practise*, Prentice Hall, 1993

[Griebel et al., 1998] Michael Griebel, Thomas Dornseifer; Tilman Neunhoeffer, *Numerical Simulation in Fluid Dynamics: A Pratical Introduction*, Philadelphia: Society for Industrial and Applied Mathematics, 1998

[Highsmith, 2002] Jim Highsmith, *What Is Agile Software Development?*, CROSS-TALK : The Journal of Defense Software Engineering, October 2002 Issue, p. 4-9, 2002

[Homer, 1999] Alex Homer, *XML IE5 Programmer's Reference*, Wrox Press Ltd., 1999

[Hoover et Perry, 1990] Stewart Hoover, Ronald F. Perry, *Simulation A Problem-Solving Approach*, Addisson Wesley, 1990

[Hörmman et Bayar, 2000] Wolfgang Hörmman et Onur Bayar, *Modelling Distributions from Data and its Influence on Simulation*, Department of Applied Statistics and Data Processing, Vienna University of Economics and Business Administration, January 2000

[Horstmann et Cornell v.1, 2001] Cay S. Horstmann et Gary Cornell, *Au Cœur de Java : Notions Fondamentales, Volume 1*, Campus Press, 2001

[Horstmann et Cornell v.2, 2001] Cay S. Horstmann et Gary Cornell, *Au Cœur de Java : Fonctions Avancées, Volume 2*, Campus Press, 2001

[Hu, 1989] David Hu, *C/C++ for Expert Systems: Unleashes the power of artificial intelligence*, MIS Press, 1989

[IDE] Wikipedia: The Free Encyclopedia, *Integrated development environment (IDE)*, http://en.wikipedia.org/wiki/Integrated_development_environment

[IDE Comparison] Wikipedia: The Free Encyclopedia, *Comparison of integrated development environments*, http://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments

[Interquartile Range] Wikipedia: The Free Encyclopedia, *Interquartile range*, http://en.wikipedia.org/wiki/Interquartile_range

[Java Programming] Wikipedia: The Free Encyclopedia, *Java (programming language)*, http://en.wikipedia.org/wiki/Java_programming_language

[Java SE] Sun Microsystems, *Java® Platform, Standard Edition,* http://java.sun.com/javase/

[Java 2 SE JAXP v.1.1.1] Sun Microsystems, *Java® 2 Platform, Standard Edition, API for XML Parsing v.1.1.1*, http://java.sun.com/xml/jaxp/dist/1.1/docs/api/index.html

[Java 2 SE JAXP v.1.1.1 tutorial] Sun Microsystems, *Java® 2 Platform, Standard Edition, v.1.4.2, The Java API for Xml Processing (JAXP) Tutorial,* http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/index.html

[Java 2 SE v.1.4.2 API] Sun Microsystems, *Java® 2 Platform, Standard Edition, v.1.4.2 API Specification*, http://java.sun.com/j2se/1.4.2/docs/api/

[Java 2 SE v.1.4.2 Serialization] Sun Microsystems, *Java® 2 Platform, Standard Edition, v.1.4.2 Object Serialization Specification*, http://java.sun.com/j2se/1.4.2/docs/guide/serialization/spec/serialTOC.html

[JavaBeans] Sun MicroSystems, *Desktop Java JavaBeans*, http://java.sun.com/products/javabeans/

[JavaBeans Concept] Sun MicroSystems, *The Java™ Tutorials, Lesson: JavaBeans Concepts*, http://java.sun.com/docs/books/tutorial/javabeans/whatis/

[JavaBeans Bound Properties] Sun MicroSystems, *The Java™ Tutorials, Lesson: Bound Properties*, http://java.sun.com/docs/books/tutorial/javabeans/properties/bound.html

[JavaBeans Simple Properties] Sun MicroSystems, *The Java™ Tutorials, Lesson: Simple Properties*, http://java.sun.com/docs/books/tutorial/javabeans/properties/properties.html

[JavaBeans Trail] Sun MicroSystems, *The Java™ Tutorials, Trail: JavaBeans™*, http://java.sun.com/ docs/ books/ tutorial/ javabeans/

[Jeong, 2000] Ki-Young Jeong, *Conceptual frame for development of optimized simulation-based scheduling systems*, Expert Systems with Applications 18, p. 299–306, 2000

[Jess] Sandia National Laboratories: Ernest J. Friedman-Hill, *Java Expert System Shell (Jess)* v. 7.0, http://herzberg.ca.sandia.gov/jess/

[Jess 7.0 Manual] Sandia National Laboratories: Ernest J. Friedman-Hill, *Jess v. 7.0 Documentation Manual*, http://herzberg.ca.sandia.gov/jess/docs/70/

[JFC] Wikipedia: The Free Encyclopedia, *Java Foundation Class*, http://en.wikipedia.org/wiki/Java_Foundation_Classes

[JFC and Swing] Sun MicroSystems, *The Java™ Tutorials, Trail: About the JFC and Swing*, http://java.sun.com/docs/books/tutorial/uiswing/start/about.html

[JFC / Swing concurrency] Sun Microsystems, *The Java™ Tutorials, Trail: Creating a GUI with JFC/Swing, Lesson: Concurrency in Swing*, http://java.sun.com/docs/books/tutorial/uiswing/concurrency/index.html

[JGraph] Gaudenz Alder and JGraph Ltd, *JGraph: The Java Open Source Graph Drawing Component*, http://www.jgraph.com/jgraph.html

[Kakmier, 1982] Leonard J. Kakmier, *Statistiques de la gestion, Théorie et problèmes*, Série Schaum, Chenelière McGraw-Hill, 1982

[Kasputis et Ng, 2000] Stephen Kasputis et Henry C. Ng, *Model Composability: Formulating a Research Thrust: Composable Simulations*, Proceedings of the 2000 Winter Simulation Conference, p.1577-1584, 2000

[Kilgore, 2000] Richard A. Kilgore, ThreadTec Inc., *Silk: Java and Object-Oriented Simulation*, Proceedings of the 2000 Winter Simulation Conference, J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, Editors, 2000

[Krishnamoorthy et Rajeev, 1996] C.S. Krishnamoorthy et S. Rajeev, *Artificial Intelligence and Expert Systems for Engineers*, CRC Press LLC, 1996

[Law et Kelton, 2000] Averil M. Law et W. David Kelton, *Simulation Modeling and Analysis* (Third Edition), McGraw-Hill, 2000

[Lindsay, 1980] Robert K. Lindsay, *Applications of Artificial Intelligence for Organic Chemistry: the Dendral Project*, McGraw-Hill, 1980

[List of UML tools] Wikipedia: The Free Encyclopedia, *List of UML tools*, http://en.wikipedia.org/wiki/List_of_UML_tools

[L'Écuyer et al., 2002] Pierre L'Écuyer, Lakhdar Meliani et Jean Vaucher, Département d'Informatique et de Recherche Opérationnelle (DIRO), *SSJ : A Framework for Stochastic Simulation in Java*, Proceedings of the 2002 Winter Simulation Conference, E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, Editors, 2002

[Logiciel libre] Wikipedia : projet d'encyclopédie libre et gratuite, *Logiciel libre*, http://fr.wikipedia.org/wiki/Free_software

[Luger et Stubblefield, 1998] George F. Luger et William A. Stubblefield, *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, Addison Wesley Longman, 1998

[MacKeown, 1997] Kevin MacKeown, *Stochastic Simulation in Physics*, Springer-Verlag 1997

[McCarty et Cassidy-Dorion, 1999] Bill McCarty, Luke Cassidy-Dorion, *Java Distributed Objects*, SAMS, 1999

[McAllister et Stanat, 1977] Donald F. Stanat, David F. McAllister, *Discrete mathematics in computer science*, Prentice-Hall Englewood, 1977

[McConnell, 1996] Steve McConnell, *Rapid Development: Taming Wild Software Schedules*, Microsoft Press, 1996

[McConnell, 2004] Steve McConnell, *Code Complete: A practical handbook of software construction (Second Edition)*, Microsoft Press, 2004

[Model-driven Architecture] Wikipedia: The Free Encyclopedia, *MDA: Model-driven architecture*, http://en.wikipedia.org/wiki/Model-driven_architecture

[Méthode agile] Wikipedia : projet d'encyclopédie libre et gratuite, *Méthode agile*, http://fr.wikipedia.org/wiki/Méthode_agile

[Meyer, 2000] Bertrand Meyer, *Object-Oriented Software Construction (Second Edition)*, Prentice Hall, 2000

[Minsky, 1974] Marvin Minsky, *A Framework for Representing Knowledge*, MIT-AI Laboratory Memo 306, http://web.media.mit.edu/~minsky/papers/Frames/frames.html, June, 1974

[Müller, 1996] Peter Müller, *Introduction to Object-Oriented Programming Using C++*, Globewide Network Academy (GNA), 1996

[Newell et Simon, 1972] Allen Newell, Herbert A. Simon, *Human Problem Solving*, Prentice-Hall, 1972

[Page et al., 2000] Michel Page, Jérôme Gensel, Cécile Capponi, Christophe Bruley, Philippe Genoud et Danielle Ziébelin, *Représentation de connaissances au moyen de classes et d'associations : le système AROM*, 6èmes journées "Langages et Modèles à Objets 2000", janvier 2000

[Pidd, 1998] Michael Pidd, *Computer Simulation in Management Science (Fourth Edition)*, John Wiley & Sons, 1998

[Pidd et al, 1999] Michael Pidd, Noelia Oses, Roger J. Brooks, *Component-Based Simulation on the Web?*, Proceedings of the 1999 Winter Simulation Conference, P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, Editors, 1999

[Plateforme client riche] Wikipedia : projet d'encyclopédie libre et gratuite, *Plateforme client riche*, http://fr.wikipedia.org/wiki/Plateforme_client_riche

[Plugiciel] Wikipedia : projet d'encyclopédie libre et gratuite, *Plugin : plugiciel*, http://fr.wikipedia.org/wiki/Plugiciel

[Poirier, 1997] Mario Poirier, *Étude sur l'entreposage et la manutention des anodes (étude CEG.6493)*, Cégertec Experts-Conseils, 1997

[Processus stochastique] Wikipedia : projet d'encyclopédie libre et gratuite, *Procesus stochastique*, http://fr.wikipedia.org/wiki/Processus_stochastique

[Praehofer et al, 1999] Herbert Praehofer, Johannes Sametinger, Alois Stritzinger, *Discrete-Event Simulation Using the JavaBeans Component Model*, Proceedings of 1999 International Conference On Web-Based Modelling & Simulation, January p.17-20, 1999

[Priestley, 2000] Mark Priestley, *Practical Object-Oriented Design with UML*, McGraw-Hill, 2000

[Quillian, 1968] M. R Quillian, *Semantic Memory, Semantic Information Processing*, 216-270, The MIT press, Cambridge, MA, 1968

[Riley et al., 2005] Gary Riley, Chris Culbert, Brian Donnell, Frank Lopez, *CLIPS v 6.2.3 Advanced and Basic Programming Guides*, http://www.ghg.net/clips/download/documentation/, 2005

[Round-trip] Wikipedia: The Free Encyclopedia, *Round-trip engineering*, http://en.wikipedia.org/wiki/Round-trip_engineering

[Rudolph, 2003] George Rudolph, *An Essay: Some Guidelines for Deciding whether to Use a Rules Engine*, http://herzberg.ca.sandia.gov/jess/guidelines.shtml, 2003

[Russell et Norvig, 2003] Stuart J. Russell et Peter Norvig, *Artificial Intelligence: A Modern Approach, Second Edition,* Pearson Education Inc., 2003

[Schaffer, 1997] Clifford A. Schaffer, *A Pratical Introduction to Data Structures and Algorithm Analysis*, Prentice Hall, 1997

[Simon, 1977] Herbert Alexander Simon, *The New Science of Management Decision*, Englewood Cliffs, NJ : Prentice Hall, 1977

[SSJ] Pierre L'Écuyer et Richard Simard, Simulation Stochastique en Java, Département d'Informatique et de Recherche Opérationnelle (DIRO) de l'Université de Montréal, http://www.iro.umontreal.ca/~simardr/ssj/

[Software Component] Wikipedia: The Free Encyclopedia, *Software Componentry*, http://en.wikipedia.org/wiki/Software_component

[Szyperski, 1999] Clemens Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1999

[Tabsh et Dupuis 1, 1997] Imad Tabsh, Marc Dupuis, *Simulation of the Dynamic Response of Aluminum Reduction Cells*, TMS Light Metal, 1997

[Tabsh et Dupuis 2, 1997] Imad Tabsh, Marc Dupuis, *Simulation of the Dynamic Response of Aluminum Reduction Cells*, TMS Light Metal, 1997

[Teghem, 2003] Jacques Teghem, *Programmation linéaire*, Paris : Bruxelles : Ellipses, 2003

[Thread] Wikipedia: The Free Encyclopedia, *Thread (computer science)*, http://en.wikipedia.org/wiki/Thread_(computer_science)

[Tokuba et Batory, 2001] Lance Tokuba et Don Batory, *Evolving Object-Oriented Designs with Refactorings*, Automated Software Engineering, Vol. 8, p. 89–120, 2001

[Totten et Mackenzie, 2003] George E. Totten, D. Scott MacKenzie, *Handbook of aluminum*, Marcel Dekker, 2003

[Turban et Aronson, 2000] Efraim Turban et Jay E. Aronson, *Decision Support Systems and Intelligent systems (sixth edition)*, Pretice Hall, 2000

[UML Tool] Wikipedia: The Free Encyclopedia, *UML Tool*, http://en.wikipedia.org/ wiki/UML_tool

[Ündey et al, 2003] Cenk Ündey, Eric Tatara et Ali Çinar, *Real-time batch process supervision by integrated knowledge-based systems and multivariate statistical methods*, Engineering Applications of Artificial Intelligence, Vol. 16, p. 555-566, 2003

[VBA] Microsoft Visual Basic for Applications (VBA), http://msdn.microsoft.com/vba/

[Vincent, 2003] Cyril Vincent, *XML et XSL : les feuilles de style XML*, Éditions ENI, 2003

[Wampler, 2001] Bruce E. Wampler, *The Essence of Object Oriented Programming with Java and UML*, Addison-Wesley, 2001

[World-Aluminium.org] World-Aluminium.org, Home of the International Aluminum Institute, http://www.world-aluminium.org

[XMI] Wikipedia: The Free Encyclopedia, XML Metadata Interchange (XMI), http://en.wikipedia.org/wiki/XML_Metadata_Interchange

[XML v.1.1] World Wide Web Consortium (W3C), *Extensible Markup Language (XML) v.1.1*, http://www.w3.org/XML/

[XSL v.1.1] World Wide Web Consortium (W3C), *Extensible Stylesheet Language (XSL)*, http://www.w3.org/Style/XSL/

[Young et Tag, 2001] Young Ik Cho et Tag Gon Kim, *DEVS Framework for Component-based Modeling/Simulation of Discrete Event Systems*, Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology (KAIST), 2001

[Zeigler, 1995] Bernard P. Zeigler, *Object Oriented Simulation with Hierarchical, Modular Models: Selected Chapters Updated for DEVS-C++,* Copyright by Author, http://www.acims.arizona.edu/PUBLICATIONS/HTML/objsim.html, 1995

[Zeigler et al, 1999] Bernard P. Zeigler, Doohwan Kim, Stephen J. Buckley, *Distributed Supply Chain Simulation in a DEVS/CORBA Execution Environment*, Proceedings of the 1999 Winter Simulation Conference, P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, editors, 1999

[Zeigler et Sarjoughian, 2005] Bernard P. Zeigler et Hessam S. Sarjoughian, *Introduction to DEVS Modeling and Simulation with JAVA: Developing Component-Based Simulation Models*, Arizona Center for Integrative Modeling and Simulation, Electrical & Computer Engineering, http://www.acims.arizona.edu/education/, 2005

[Zobrist et Leonard, 1997] George W. Zobrist, *Object-Oriented Simulation, Reusability, Adaptability, Maintainability*, IEEE Press, 1997

[Zumdahl, 1988] Steven S. Zumdahl, *Chimie des solutions*, Centre éducatif et culturel Inc., 1988

ANNEXE 1

GLOSSAIRE DU CYCLE DE VIE DES ANODES DE CARBONE D'UNE ALUMINERIE

Annexe 1 : Glossaire du cycle de vie des anodes de carbone d'une aluminerie

1. Activité (activity) : Il s'agit d'une opération ou d'une procédure qui contribue à faire évoluer l'état des anodes. Chaque activité est effectuée par un poste du cycle de vie des anodes.
2. Alumine (alumina) : C'est un oxyde d'aluminium (Al_2O_3) qu'on obtient à partir d'un minerai appelé bauxite. Dans le cadre du procédé Hall-Héroult, menant à la production de l'aluminium, l'alumine est dissoute dans un électrolyte.
3. Aluminium (aluminium) : Un des éléments du tableau périodique les plus abondants de la terre (Al). Il fait parti du groupe des métaux. À l'état naturel, l'aluminium est fréquemment combiné avec d'autres éléments, puisqu'il est très réactif. C'est le cas de l'alumine (Al_2O_3).
4. Aluminerie (aluminium smelter) : C'est une usine qui produit de l'aluminium selon le procédé d'Hall-Héroult. Une aluminerie est généralement formée d'environ 300 cuves qui produisent jusqu'à 125 000 tonnes de métal par année. Les installations les plus récentes peuvent atteindre 350 000 à 400 000 tonnes annuellement.
5. Anode (anode) : Électrode chargée positivement. Généralement, une oxydation se produit à sa surface. Dans le cas du procédé de production de l'aluminium, l'anode est un bloc de carbone cuit (voir anode cuite). Elle est suspendue à l'aide d'une tige métallique et sa partie inférieure baigne dans le bain de la cellule électrolytique.
6. Anode auto-cuite ou Söderberg (self-baking or Söderberg anode) : C'est un bloc constitué à partir de la pâte d'anode. La pâte prend sa forme dans un caisson d'acier qui est placé en contact avec l'électrolyte de la cellule de réduction. La chaleur de l'électrolyte et le courant électrique se chargent de cuire l'anode. Ce type d'anode est moins utilisé, en raison des émissions nocives produites par l'auto-cuisson.
7. Anode crue (green anode) : Il s'agit d'un bloc constitué à partir de la pâte d'anode. Il est formé par vibrocompaction ou par pression de la pâte. Ensuite, l'anode crue est cuite dans un four.
8. Anode cuite ou précuite (baked or prebaked anode) : C'est une anode crue qui a été durcie par calcination dans un four, à environ 1100°C. L'anode cuite est suspendue par une tige

métallique et placée en contact avec l'électrolyte de la cellule de réduction. Ce type d'anode produit moins d'émissions nocives que les anodes Söderberg, puisque les matières organiques volatiles sont brûlées dans le four.

9. Assemblage (assembly) : Activité de l'étape de scellement qui consiste à insérer les goujons de la tige métallique, à l'intérieur de l'anode cuite, pour former un ensemble anodique.
10. Bain (bath) : Contenu de la cuve qui est principalement formé d'une solution électrolytique de cryolite fondue (Na_2AlF_6) et d'alumine dissoute (Al_2O_3), d'aluminium en fusion (Al), de divers ions issus de l'électrolyse et de dioxyde de carbone (CO_2). L'anode de carbone trempe à la surface du bain où la solution d'alumine et de cryolite se maintient. La température du bain doit toujours avoisiner les 970 °C pour assurer la conductivité. L'aluminium se dépose au fond de la cuve et est siphonné. On ajoute continuellement de l'alumine dans le bain, afin que la réduction du métal se poursuive et pour éviter l'apparition d'un effet anodique.
11. Bauxite (bauxite) : C'est une variété de minerai constitué principalement d'hydroxyde d'aluminium. La bauxite sert à produire de l'alumine à partir du procédé Bayer.
12. Billette d'extrusion (billet d'extrusion) : Objet obtenu par la coulée de l'aluminium à travers un cylindre. La billette peut être passée à travers une filière profilée, afin d'obtenir des fils ou des pièces métalliques de diverses apparences.
13. Carbone (carbon) : Élément du tableau périodique très abondant dans la nature (C), sous forme de minerai ou de matière organique. Il fait parti du groupe des non-métaux. Dans le cadre du procédé Hall-Héroult, le carbone des anodes est impliqué dans la réduction de l'aluminium.
14. Cathode (cathode) : Électrode chargée négativement. Généralement, une réaction de réduction se produit à sa surface. Dans le cas du procédé de production de l'aluminium, la cathode correspond à l'aluminium qui se précipite au fond de la cuve.

15. Cellule de réduction (reduction cell) : Il s'agit d'un synonyme de cuve. Elle doit son nom du fait qu'elle est le cadre de la demi-réaction de réduction. Elle permet donc la formation d'aluminium et sa précipitation au fond de la cellule.
16. Coke de pétrole (petroleum coke) : Sous-produit du raffinage du pétrole obtenu par craquage thermique. C'est une substance solide, encore plus dure que le charbon, qui est composée principalement de carbone.
17. Coulée de l'aluminium ou du métal (aluminium or metal casting) : Activité de l'étape de production du métal. Elle consiste à récupérer l'aluminium au fond des cellules et à le verser dans des moules ou des filières profilées, afin qu'il se solidifie en lingots ou en billettes.
18. Coulée de la fonte de fer (iron casting) : Activité de l'étape de scellement qui consiste à verser de la fonte de fer, entre les goujons de la tige et l'anode cuite, afin de les sceller ensembles.
19. Coulée de la pâte d'anode (paste anode casting) : Activité de l'étape de production d'anodes qui consiste à verser la pâte d'anode dans des moules, afin d'obtenir des anodes crues. Celles-ci sont formées soit par compression hydraulique ou par vibrocompaction mécanique.
20. Craquage thermique (thermal cracking) : Procédé qui intervient durant la distillation du pétrole. Il consiste à chauffer sous pression et à haute température, les portions les plus lourdes du pétrole brut. Ainsi, on accroît le rendement de l'essence produite.
21. Cryolite (cryolite) : Fluorure d'aluminium qu'on retrouve naturellement sous la forme d'un sel (Na_2AlF_6). Au cours du procédé Hall-Héroult, la cryolite est dissoute au sein du bain. La solution d'alumine et de cryolite est un excellent électrolyte.
22. Cuisson des anodes (anodes baking) : Activité de l'étape de production des anodes. Elle consiste à chauffer les anodes crues dans un four, pour obtenir des anodes cuites. La cuisson permet de débarrasser les anodes de leurs impuretés, d'améliorer leur solidité et leur conductivité.

23. Cuve (cell) : C'est le réservoir du bain électrolytique. Il est généralement fait d'un caisson d'acier dont le fond est constitué de blocs cathodiques carbonés. On l'appelle aussi cellule de réduction.
24. Cycle de vie des anodes (anodes life cycle) : Séquence d'activités qui mènent à la transformation des anodes. Le cycle débute à la tour à pâte, où la mixture formant les anodes crues est préparée. Il se termine une fois que les anodes ont été transformées en mégots. Chacune des activités du procédé peuvent être regroupées en étapes, notons : la production d'anodes, la manutention et l'entreposage d'anodes, le scellement d'anodes et la production de métal.
25. Détachement de l'ensemble anodique (anodic set removal) : C'est une activité de l'étape de production du métal. Elle consiste à détacher l'ensemble anodique qui est suspendu au-dessus de la cuve. L'ensemble anodique est détaché, une fois que son anode de carbone est en grande partie consumée, par la réaction de réduction de l'aluminium.
26. Détachement de la fonte et du graphite (cast iron and graphite removal) : C'est une activité de l'étape de scellement. Elle permet de retirer la fonte de fer et la couche de graphite qui sont incrustées sur les goujons de la tige. Cette opération est généralement faite à l'aide d'une presse et de couteaux mécaniques, une fois que le mégot a été enlevé.
27. Détachement du mégot (butt removal) : Activité de l'étape de scellement qui consiste à disjoindre le mégot de l'ensemble anodique, pour que les goujons de la tige soient dégagés. Cette opération est généralement effectuée à l'aide d'une presse, une fois que le mégot a été nettoyé et grenaillé.
28. Effet anodique (anodic effect) : Il s'agit d'une augmentation du voltage dans la cellule de réduction. Elle est due soit à un manque d'alumine dissoute dans l'électrolyte, ou à une température trop basse dans le bain. Pour éviter cet effet, il est capital de dissoudre continuellement de l'alumine et de maintenir la température de la cellule à près de 970°C. Notons que l'effet anodique occasionne une diminution de performance de la réaction de

réduction de l'aluminium. Ceci entraîne la formation de gaz fluorés, principalement du CF_4, fort dommageable pour la couche d'ozone.

29. Électrode (electrode) : Conducteur par lequel le courant arrive ou sort, dans un électrolyte ou un tube de gaz. Pour plus de détails, voyez les descriptions de la cathode et de l'anode qui sont toutes deux des électrodes.
30. Électrolyse (electrolys) : Procédé qui consiste à faire passer un courant électrique dans une cellule, afin d'obtenir une réaction chimique qui ne peut pas avoir lieu spontanément. La réduction de l'aluminium est effectuée à partir des ions d'une solution de cryolite et d'alumine. Or, même en ayant tous les éléments en présences, aucune réaction d'oxydoréduction n'est spontanément enclenchée. Pour initier le déplacement d'électrons, on doit faire passer un courant électrique de l'anode (positive), vers la cathode (négative). Ce courant est à très faible voltage mais à haute intensité (300 kiloampères). En plus, il faut s'assurer de la conductivité de la solution électrolyte, en la chauffant à près de 970°C.
31. Électrolyte (electrolyte) : Substance qui est capable de conduire le courant électrique, grâce au mouvement de ses ions dissociés. La plupart du temps, l'électrolyte est obtenu par la dissolution ou la fusion d'un sel, d'un acide ou d'une base, dans un solvant polaire. Une solution formée de sel d'alumine (Al_2O_3) et de cryolite fondue (Na_3AlF_6) constitue un électrolyte efficace. En effectuant l'électrolyse de cette solution, on peut réduire l'aluminium.
32. Ensemble anodique (anodic set) : Assemblage issu de l'étape de scellement des anodes. Il est formé de quatre composantes : l'anode cuite, la tige métallique, ses goujons et la fonte de fer qui lie l'anode aux goujons.
33. Entreposage des anodes (anodes warehousing) : Activité de l'étape de manutention et d'entreposage qui permet d'entasser les anodes cuites, dans un endroit sec, avant de les expédier à l'étape de scellement. Les anodes crues peuvent aussi être entreposées, puis récupérées, selon les besoins des fours. Les anodes crues et cuites sont généralement

empilées dans l'entrepôt par des grues gerbeuses. Ces grues peuvent manipuler plusieurs anodes à la fois.

34. Étape (stage) : C'est un regroupement d'activités du cycle de vie des anodes. Une étape contribue à décrire un secteur général du procédé : production des anodes, manutention et entreposage des anodes, scellement des anodes, ainsi que production du métal.
35. Fonte de fer (cast iron) : Fer en fusion qu'on coule à l'intérieur de la cavité située entre chaque goujon d'une tige et l'anode cuite.
36. Four à cuisson d'anode (anode baking furnace) : Poste de l'étape de production des anodes. Il est associé à l'activité de cuisson des anodes. Un four est formé de chambres de cuissons qui sont divisées en deux rangées parallèles. Chaque chambre est fractionnée en alvéoles, par des cloisons réfractaires. Les anodes crues sont d'abord empilées dans les alvéoles. Des gaz chauds circulent à travers les cloisons qui sont munies d'orifices. Ces gaz sont le résultat de la combustion de carburant fossile. Les anodes sont cuites, sous l'action de la chaleur générée par la combustion.
37. Formation des anodes crues (green anodes forming) : Consiste à former l'anode crue, par compression ou vibrocompaction de la pâte, une fois qu'elle est suffisamment refroidie.
38. Formation de l'anode avec presse (anode formation by pressing) : Activité qui est associée à l'étape de production des anodes. Elle permet de donner à la pâte une forme rectangulaire, grâce à une presse hydraulique mécanique. On obtient ainsi une anode crue.
39. Formation de l'anode avec vibrocompacteur (anode formation by vibrocompaction) : Activité qui est associée à l'étape de production des anodes. Elle permet de donner à la pâte une forme rectangulaire, grâce à un vibrocompacteur mécanique. On obtient ainsi une anode crue.
40. Goujons (stub) : Composantes de l'ensemble anodique, constituant les embouts de la tige métallique. Une tige comporte généralement de deux à six goujons.
41. Graphite (graphite) : Variété naturelle de carbone cristallisé, servant à enduire les goujons de la tige et à recouvrir la paroi des cuves.

42. Graphitage des goujons (stubs graphitization ou stubs graphite coating) : Activité de l'étape de scellement qui consiste à enduire de graphite les goujons de la tige.
43. Grenaillage de la tige (rod shortblasting) : Il s'agit d'une activité de l'étape de scellement, qui permet de nettoyer la tige, en projetant des grenailles. Elle survient une fois que le mégot et la fonte ont été détachés.
44. Grenaillage du mégot (butt shortblasting) : C'est une activité de l'étape de scellement durant laquelle, on nettoie le mégot de l'ensemble anodique, en projetant des grenailles. Ceci permet d'enlever l'excédant du bain.
45. Ion (ion) : Atome ou molécule qui a perdu sa neutralité électrique, par acquisition ou perte d'un ou de plusieurs électrons.
46. Ligne de cuves (pot line) : Un ensemble de cuves disposées de manière adjacente. On peut retrouver plusieurs lignes au sein d'une même salle de cuves.
47. Manutention et entreposage des anodes (anodes handling and warehousing) : Il s'agit d'une étape qui permet de manipuler et d'empiler les anodes cuites ou crues, avant de les acheminer au scellement. Les activités associées à cette étape sont : le refroidissement des anodes cuites et l'entreposage des anodes cuites ou crues.
48. Mégot (butt) : Anode résiduelle attachée aux goujons de la tige. Le mégot est obtenu une fois que l'anode cuite a été consumée durant l'électrolyse. Ce résidu est fréquemment réutilisé par la tour à pâte, afin de produire de nouvelles anodes.
49. Nettoyage du mégot (butt cleaning) : Il s'agit d'une activité de l'étape de scellement. Elle consiste à faire tremper le mégot de l'ensemble anodique dans des bassins, afin d'abaisser sa température et enlever le surplus du bain qui le recouvre.
50. Oxydation (oxidation) : Il s'agit d'une demi-réaction chimique de l'oxydoréduction. Pour plus de détails voyez la description de l'oxydoréduction.
51. Oxydoréduction (oxidoreduction) : Réaction chimique caractérisée par le transfert d'électrons entre un agent dit réducteur et un autre dit oxydant. Comme son nom le laisse croire,

l'oxydoréduction est divisée en deux demi-réactions. La première est nommée oxydation. Elle entraîne la perte d'électrons d'un des réactifs (agent réducteur). La seconde est la réduction. Au cours de celle-ci, un des réactifs (agent oxydant) a un gain d'électrons. La réduction de l'aluminium est une réaction qui ne peut survenir que par électrolyse.

52. Pâte d'anode (paste anode) : Mélange homogène contenu dans la tour à pâte. Il est composé de coke de pétrole, de goudron liquide et de résidus de mégots broyés. Pour donner de la consistance à la pâte, on doit la réchauffer à près de 160°C.
53. Poste (station) : Un poste est une ressource qui effectue une activité, afin de modifier l'état des anodes.
54. Préparation de la pâte d'anode (paste anode preparation) : C'est une activité de l'étape de production des anodes. Elle consiste à broyer les mégots provenant du scellement et à les mélanger avec du coke de pétrole, ainsi que du goudron, dans la tour à pâte.
55. Procédé Bayer : Processus mécanique et chimique qui permet de raffiner la bauxite pour obtenir de l'alumine. Il a été baptisé en l'honneur de son inventeur l'Autrichien Karl Bayer. On lessive d'abord la bauxite sous haute pression, avec de l'hydroxyde de sodium (NaOH). Ainsi, on obtient de l'hydroxyde d'aluminium ($Al(OH)_3$) qui est dissout dans de l'eau. L'hydroxyde d'aluminium de la solution est ensuite précipité, pour le séparer des impuretés. L'hydroxyde d'aluminium pur est finalement calciné, pour produire de l'alumine (Al_2O_3).
56. Procédé Hall-Héroult (Hall-Héroult process) : Processus à la base de la réduction de l'aluminium. Il est baptisé en l'honneur de ses inventeurs, l'Américain Charles Hall et le Français Paul Héroult, qui firent leur découverte séparément en 1886. Il consiste à dissoudre continuellement de l'alumine dans un bain électrolytique contenant de la cryolite fondue. On élève d'abord la température de la cellule à 970°C, pour que la dissolution survienne et accroître la conductibilité du mélange. Une réaction de réduction est initiée par électrolyse, c'est-à-dire en faisant passer un courant électrique, entre une anode de carbone et une cathode, située au-bas de la cuve. Les ions en solution réagissent ensembles et l'aluminium est

réduit au fond du bain, faisant ainsi office de cathode. L'anode est graduellement consumée, puisqu'elle intervient dans le processus de réduction. La réaction globale du procédé est la suivante :

$$2Al_2O_3 + 3C \rightarrow 4Al + 3CO_2$$

Au terme de la réaction, le métal est siphonné et recueilli dans un creuset. Parfois, l'aluminium est mélangé à un alliage. Puis, il est nettoyé et coulé dans un moule.

57. Production du métal (metal production) : Étape du cycle de vie des anodes qui permet d'obtenir de l'aluminium selon le procédé Hall-Héroult. Elle est constituée des activités suivantes : suspension de l'ensemble anodique, électrolyse, détachement de l'ensemble anodique, siphonage du métal et coulée du métal.
58. Production des anodes (anodes production) : Étape du cycle de vie des anodes qui couvre les activités suivantes : préparation de la pâte d'anode, réchauffage de la pâte, refroidissement de la pâte, coulée de la pâte, formation de l'anode avec un vibrocompacteur, formation de l'anode avec une presse et cuisson de l'anode.
59. Réchauffage de la pâte d'anode (paste anode heating) : Activité de l'étape de production des anodes. Elle consiste à élever la température de la mixture de la tour à pâte, jusqu'à près de 160 °C. On accroît ainsi son homogénéité et sa consistance.
60. Réchauffage des goujons (stubs heating) : Activité de l'étape de scellement des anodes qui consiste à élever la température des goujons, juste avant de procéder à l'assemblage de la tige et de l'anode cuite.
61. Redressement de la tige (rod straightening) : C'est une activité de l'étape de scellement des anodes. Elle consiste à réaligner à l'aide d'une presse hydraulique une tige recourbée.
62. Réduction (reduction) : Il s'agit d'une demi-réaction chimique de l'oxydoréduction. C'est grâce à la réduction que l'aluminium se précipite au fond de la cuve, lorsque les ions du bain interagissent avec le carbone de l'anode.

63. Refroidissement des anodes cuites (baked anode cooling) : Activité de l'étape de manutention et d'entreposage qui consiste à abaisser la température de l'anode cuite, en la faisant tremper dans un bassin rempli d'eau. Une fois que l'anode est refroidie, on peut l'entreposer.
64. Refroidissement de la pâte d'anode (paste anode cooling) : Activité de l'étape de production des anodes. Elle permet d'abaisser la température de la pâte d'anode à environ 116°C. Par la suite, il est possible de la couler et de la mouler avec une presse ou un vibrocompacteur.
65. Réparation de la tige (rod mending) : Activité de l'étape de scellement des anodes qui permet de solidifier une tige. Généralement, elle consiste à scier les goujons fragilisés et à souder de nouveaux goujons à la tige.
66. Salle des cuves (pot room) : C'est l'endroit où l'on retrouve entre autre, une ou plusieurs lignes de cellules de réduction.
67. Salle du scellement (rodding room) : C'est l'endroit où l'étape de scellement des anodes a lieu.
68. Scellement (rodding) : Étape qui permet de fabriquer et traiter les ensembles anodiques. Durant le scellement, les goujons de la tige sont liés à l'anode cuite, grâce à de la fonte de fer. Cette étape peut être décomposée en plusieurs activités : nettoyage du mégot, grenaillage du mégot, détachement du mégot, détachement de la fonte, grenaillage de la tige, grenaillage de la tige, réparation de la tige, redressement de la tige, graphitage des goujons, réchauffage des goujons, assemblage de l'ensemble anodique et coulée de fonte.
69. Siphonage de l'aluminium (aluminium tapping) : Activité de l'étape de production du métal qui consiste à aspirer l'aluminium réduit au fond de la cuve.
70. Suspension de l'ensemble anodique (anodic set hanging) : Activité de l'étape de production du métal. Elle permet de suspendre l'ensemble anodique au-dessus de la cellule de réduction, pour que l'anode de carbone baigne dans le bain.
71. Tige (rod) : Composante métallique qui est rattachée aux goujons. La tige permet de suspendre l'anode, pour qu'elle puisse tremper dans le bain de la cellule de réduction. Elle fait donc partie

de l'ensemble anodique. Les tiges doivent être conductibles, pour que le courant électrique circule jusqu'au bain, via l'anode de carbone. Elles sont faites d'aluminium ou de cuivre.

72. Tour à pâte (paste anode mixer) : C'est un réservoir qui intervient lors de l'étape de production des anodes. Il contient une mixture hétérogène, qui une fois chauffée, forme la pâte d'anode.

ANNEXE 2

PHASE DE MANUTENTION ET D'ENTREPOSAGE DES ANODES

Annexe 2 : Phase de manutention et d'entreposage des anodes

Nom du four	Jour	Heure d'arrivée d'un train à l'entrée du four
P1	Lundi	00h30, 00h45, 01h00, 01h15, 01h30, 01h45, 02h00, 02h15, 04h00, 04h15, 04h30, 04h45, 09h05, 09h20, 09h40, 17h30, 17h45, 19h00, 19h15, 19h30, 19h45, 20h00, 20h15, 20h30, 20h45, 21h00, 21h15
	Mardi	00h30, 01h00, 01h40, 02h10, 02h30, 04h05, 09h00, 09h15, 09h30, 09h45, 10h00, 10h15, 10h30, 10h45, 12h30, 12h45, 13h20, 13h45, 17h30, 19h00, 19h40, 20h20, 21h10, 21h50
	Mercredi	00h30, 00h45, 01h00, 01h15, 01h30, 01h45, 02h00, 02h15, 04h00, 04h15, 04h30, 04h45, 09h00, 09h15, 09h30, 09h45, 10h00, 10h15, 10h30, 10h45, 12h30, 12h45, 13h20, 13h45, 17h30, 19h00, 19h40, 20h20, 21h10, 21h50
	Jeudi	00h30, 00h45, 01h00, 01h15, 01h30, 01h45, 02h00, 02h15, 04h00, 04h15, 04h30, 04h45, 09h00, 09h30, 10h15, 10h45, 12h30, 13h30, 17h30, 17h45, 19h00, 19h15, 19h30, 19h45, 20h00, 20h15, 20h30, 20h45, 21h00, 21h15
	Vendredi	00h30, 01h00, 01h40, 02h10, 02h30, 04h05, 09h00, 09h30, 10h45, 12h30, 13h30, 17h30, 19h00, 19h40, 20h20, 21h10, 21h50

Tableau 11 : horaire des navettes du four *P1*

Nom du four	Jour	Heure d'arrivée d'un train à l'entrée du four
P2	Lundi	00h30, 01h00, 01h40, 02h10, 02h30, 04h05, 09h00, 09h15, 09h30, 09h45, 10h00, 10h15, 10h30, 10h45, 12h30, 12h45, 13h20, 13h45, 17h30, 19h00, 19h40, 20h20, 21h10, 21h50
	Mardi	00h30, 00h45, 01h00, 01h15, 01h30, 01h45, 02h00, 02h15, 04h00, 04h15, 04h30, 04h45, 09h00, 09h30, 10h15, 10h45, 12h30, 13h30, 17h30, 17h45, 19h00, 19h15, 19h30, 19h45, 20h00, 20h15, 20h30, 20h45, 21h00, 21h15
	Mercredi	00h30, 01h00, 01h40, 02h10, 02h30, 04h05, 09h05, 09h20, 09h40, 17h30, 17h45, 19h00, 19h15, 19h30, 19h45, 20h00, 20h15, 20h30, 20h45, 21h00, 21h15
	Jeudi	00h30, 01h00, 01h40, 02h10, 02h30, 04h05, 09h00, 09h15, 09h30, 09h45, 10h00, 10h15, 10h30, 10h45, 12h30, 12h45, 13h20, 13h45, 17h30, 19h00, 19h40, 20h20, 21h10, 21h50
	Vendredi	00h30, 00h45, 01h00, 01h15, 01h30, 01h45, 02h00, 02h15, 04h00, 04h15, 04h30, 04h45, 09h00, 09h30, 10h15, 10h45, 12h30, 13h30, 17h30, 19h00, 19h40, 20h20, 21h10, 21h50

Tableau 12 : horaire des navettes du four *P2*

Nom du four	Jour	Heure d'arrivée d'un train à l'entrée du four
R cycle 36 heures	Lundi	00h00, 00h25, 00h50, 01h15, 01h40, 02h05, 02h30, 08h15, 08h40, 09h05, 09h30, 09h55, 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30
	Mardi	00h00, 00h25, 00h50, 01h15, 01h40, 02h05, 02h30, 08h15, 08h40, 09h05, 09h30, 09h55, 10h20, 10h45, 12h00, 12h25, 12h50, 13h15, 13h40, 14h05, 14h30, 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30

	Mercredi	00h00, 00h25, 00h50, 01h15, 01h40, 02h05, 02h30, 08h15, 08h40, 09h05, 09h30, 09h55, 10h20, 10h45, 12h00, 12h25, 12h50, 13h15, 13h40, 14h05, 14h30, 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30
	Jeudi	00h00, 00h25, 00h50, 01h15, 01h40, 02h05, 02h30, 08h15, 08h40, 09h05, 09h30, 09h55, 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30
	Vendredi	00h00, 00h25, 00h50, 01h15, 01h40, 02h05, 02h30, 08h15, 08h40, 09h05, 09h30, 09h55, 10h20, 10h45, 12h00, 12h25, 12h50, 13h15, 13h40, 14h05, 14h30, 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30
	Samedi	00h00, 00h25, 00h50, 01h15, 01h40, 02h05, 02h30, 08h15, 08h40, 09h05, 09h30, 09h55, 10h20, 10h45, 12h00, 12h25, 12h50, 13h15, 13h40, 14h05, 14h30, 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30
	Dimanche	00h00, 00h25, 00h50, 01h15, 01h40, 02h05, 02h30, 08h15, 08h40, 09h05, 09h30, 09h55, 10h20, 10h45, 12h00, 12h25, 12h50, 13h15, 13h40, 14h05, 14h30, 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30

Tableau 13 : horaire des navettes du four *R* avec un cycle de 36 heures

Nom du four	Jour	Heure d'arrivée d'un train à l'entrée du four
R cycle 32 heures	Lundi	00h00, 00h25, 00h50, 01h15, 01h40, 02h05, 02h30, 02h55, 03h20, 08h15, 08h40, 09h05, 09h30, 09h55, 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30
	Mardi	00h00, 00h25, 00h50, 01h15, 01h40, 02h05, 02h30, 02h55, 03h20, 08h15, 08h40, 09h05, 09h30, 09h55, 10h20, 10h45, 12h00, 12h25, 12h50, 13h15, 13h40, 14h05, 14h30, 14h55, 15h20, 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30
	Mercredi	00h00, 00h25, 00h50, 01h15, 01h40, 02h05, 02h30, 02h55, 03h20, 08h15, 08h40, 09h05, 09h30, 09h55, 10h20, 10h45, 12h00, 12h25, 12h50, 13h15, 13h40, 14h05, 14h30, 14h55, 15h20, 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30
	Jeudi	00h00, 00h25, 00h50, 01h15, 01h40, 02h05, 02h30, 02h55, 03h20, 08h15, 08h40, 09h05, 09h30, 09h55, 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30
	Vendredi	00h00, 00h25, 00h50, 01h15, 01h40, 02h05, 02h30, 02h55, 03h20, 08h15, 08h40, 09h05, 09h30, 09h55, 10h20, 10h45, 12h00, 12h25, 12h50, 13h15, 13h40, 14h05, 14h30, 14h55, 15h20, 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30
	Samedi	00h00, 00h25, 00h50, 01h15, 01h40, 02h05, 02h30, 02h55, 03h20, 08h15, 08h40, 09h05, 09h30, 09h55, 10h20, 10h45, 12h00, 12h25, 12h50, 13h15, 13h40, 14h05, 14h30, 14h55, 15h20, 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30
	Dimanche	00h00, 00h25, 00h50, 01h15, 01h40, 02h05, 02h30, 02h55, 03h20, 08h15, 08h40, 09h05, 09h30, 09h55, 10h20, 10h45, 12h00, 12h25, 12h50, 13h15, 13h40, 14h05, 14h30, 14h55, 15h20 , 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30

Tableau 14 : horaire des navettes du four *R* avec un cycle de 32 heures

Nom du four	Jour	Heure d'arrivée d'un train à l'entrée du four
R cycle 40 heures	Lundi	00h00, 00h25, 00h50, 01h15, 01h40, 08h15, 08h40, 09h05, 9h30, 09h55, 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30
	Mardi	00h00, 00h25, 00h50, 01h15, 01h40, 08h15, 08h40, 09h05, 09h30, 09h55, 10h20, 10h45, 12h00, 12h25, 12h50, 13h15, 13h40, 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30
	Mercredi	00h00, 00h25, 00h50, 01h15, 01h40, 08h15, 08h40, 09h05, 09h30, 09h55, 10h20, 10h45, 12h00, 12h25, 12h50, 13h15, 13h40, 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30
	Jeudi	00h00, 00h25, 00h50, 01h15, 01h40, 08h15, 08h40, 09h05, 09h30, 09h55, 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30
	Vendredi	00h00, 00h25, 00h50, 01h15, 01h40, 08h15, 08h40, 09h05, 09h30, 09h55, 10h20, 10h45, 12h00, 12h25, 12h50, 13h15, 13h40, 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30
	Samedi	00h00, 00h25, 00h50, 01h15, 01h40, 08h15, 08h40, 09h05, 09h30, 09h55, 10h20, 10h45, 12h00, 12h25, 12h50, 13h15, 13h40, 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30
	Dimanche	00h00, 00h25, 00h50, 01h15, 01h40, 08h15, 08h40, 09h05, 09h30, 09h55, 10h20, 10h45, 12h00, 12h25, 12h50, 13h15, 13h40, 20h15, 20h40, 21h05, 21h30, 21h55, 22h20, 23h30

Tableau 15 : horaire des navettes du four *R* avec un cycle de 40 heures

Jours	Plage horaire des arrêts planifiés du scellement
Lundi	00h00 à 00h15, 01h15 à 01h50, 02h50 à 04h00, 05h00 à 05h25, 07h15 à 08h15, 09h15 à 09h50, 10h50 à 12h00, 13h00 à 13h25, 15h15 à 16h35, 17h35 à 18h10, 19h10 à 20h20, 21h20 à 21h45, 23h35 à 23h59
Mardi	00h00 à 00h15, 01h15 à 01h50, 02h50 à 04h00, 05h00 à 05h25, 07h15 à 16h35, 17h35 à 18h10, 19h10 à 20h20, 21h20 à 21h45, 23h35 à 23h59
Mercredi	00h00 à 00h15, 01h15 à 01h50, 02h50 à 04h00, 05h00 à 05h25, 07h15 à 08h15, 09h15 à 09h50, 10h50 à 12h00, 13h00 à 13h25, 15h15 à 16h35, 17h35 à 18h10, 19h10 à 20h20, 21h20 à 21h45, 23h35 à 23h59
Jeudi	00h00 à 00h15, 01h15 à 01h50, 02h50 à 04h00, 05h00 à 05h25, 07h15 à 16h35, 17h35 à 18h10, 19h10 à 20h20, 21h20 à 21h45, 23h35 à 23h59
Vendredi	00h00 à 00h15, 01h15 à 01h50, 02h50 à 04h00, 05h00 à 05h25, 07h15 à 08h15, 09h15 à 09h50, 10h50 à 12h00, 13h00 à 13h25, 15h15 à 16h35, 17h35 à 18h10, 19h10 à 20h20, 21h20 à 21h45, 23h35 à 23h59
Samedi	00h00 à 23h59
Dimanche	00h00 à 23h59

Tableau 16 : plage horaire des arrêts planifiés de l'atelier de scellement

ANNEXE 3

CONCEPTION DE MODÈLES À L'AIDE DU LANGAGE JAVA

Annexe 3 : Conception de modèles à l'aide du langage Java

Dans le cadre de la conception de notre modèle, le choix du langage de programmation s'est naturellement porté vers Java, le langage orienté objet développé par Sun Microsystems [Java SE]. Le but de cette annexe n'est pas de discuter de façon détaillée des forces et les faiblesses de la plateforme Java. Toutefois, nous allons montrer que cette dernière possède les caractéristiques souhaitables, lorsqu'il vient le temps de mettre en œuvre un modèle sous la forme d'un programme informatique.

Indépendance de la plateforme

L'une des particularités de Java est l'indépendance des plateformes [Horstmann et Cornell v.1, 2001]. Ainsi, un même programme peut être exécuté sur diverses machines et systèmes d'exploitation. Cette caractéristique est possible grâce au compilateur Java qui traduit le code source en pseudo-code binaire (de l'Anglais « bytecode »). Le « bytecode » est une suite d'instructions qui sont propres à la plateforme Java. Il est interprété et exécuté par la machine virtuelle Java (JVM de l'Anglais « Java Virtual Machine »). Celle-ci est spécifique au système d'exploitation hôte [Java Programming]. L'indépendance de la plateforme nous garantit donc que notre modèle peut être exécuté sur n'importe quel ordinateur, en autant qu'une machine virtuelle Java soit disponible pour le système d'exploitation concerné.

Récupération de la mémoire

Une seconde caractéristique intéressante est la récupération automatique de la mémoire (de l'Anglais « Garbage Collection ») [Horstmann et Cornell v.1, 2001]. Cette particularité implique que le développeur n'a pas besoin de gérer la mémoire de son programme. En effet, la plupart des langages modernes font en sorte que le programmeur alloue et libère manuellement la mémoire, lors de la création et de la suppression des objets. Or si la libération est omise ou si elle échoue, une fuite de mémoire (de l'Anglais « memory leak ») se produit. En Java, ce type de problème ne survient pas, puisqu'il existe un récupérateur (de l'Anglais « garbage collector ») qui se charge de

libérer la mémoire des objets inutilisés [Java Programming]. Cette caractéristique s'avère fort pratique, car elle minimise le temps de construction et de longs efforts de débogage.

Environnement de développement intégré

Un environnement de développement intégré (EDI, traduction de l'Anglais IDE pour « Integrated Development Environment ») est un logiciel qui assiste les développeurs, lors de la mise en œuvre d'un programme informatique. Un EDI peut intégrer un éditeur de code source, un compilateur ou un interpréteur, des outils de conception automatisés, un débogueur et un système de gestion des versions du code source (de l'Anglais VCS pour « Version Control System ») [EDI].

Il existe un éventail d'EDIs qui sont disponibles sur le marché [IDE Comparison]. L'EDI que nous avons considéré est Eclipse [Eclipse], parce qu'il est distribué gratuitement, que son architecture est relativement ouverte et qu'il permet d'intégrer des plugiciels (traduction de l'Anglais « plugin »). Un plugiciel est « un programme qui interagit avec un logiciel principal, pour lui apporter de nouvelles fonctionnalités » [Plugiciel].

Eclipse est plus qu'un simple EDI. En fait, il s'agit d'avantage d'un cadriciel qui permet de concevoir des applications de type client riche [Eclipse software]. Un client riche (de l'Anglais « fat client ») est un logiciel destiné à être installé localement sur une machine, par opposition au client léger (de l'Anglais « thin client »), tel une application web qui est affichée dans un navigateur internet (client), à partir des traitements effectuées sur un serveur [Plateforme client riche].

Eclipse a été utilisé pour concevoir un EDI Java, dans le cadre du projet « Java Development Tools » (JDT) [Eclipse JDT]. Le JDT est livré avec Eclipse. Il incorpore les principales fonctionnalités d'un EDI, citons un compilateur Java, un débogueur « pas à pas » et le client d'un système de gestion des versions. En plus, le JDT permet d'effectuer des techniques de refactorisation et d'analyse du code source [Eclipse software]. La refactorisation consiste à retravailler le code source, pour améliorer sa lisibilité et simplifier sa maintenance [Tokuba et Batory, 2001] [Wampler, 2001].

Certaines fonctionnalités d'Eclipse que nous avons énumérées, telles la refactorisation ou le débogage « pas à pas », nous ont permises de minimiser l'effort et le temps de mise en œuvre de notre modèle. Elles s'inscrivent dans le cadre de méthodes de génie logiciel, dites *agiles* [Méthode agile], citons entre autre la *programmation extrême*. Ces méthodes promeuvent l'adaptabilité d'un programme, face à des besoins qui sont en constante évolution [Extreme Programming].

Composant JavaBeans

Comme nous avons mentionné à la section 1.7.3, les composants logiciels sont fréquemment utilisés avec des OOS (« Object Oriented Simulation ») [Banks et al, 1998] [Praehofer et al, 1999]. À ce titre, la plateforme Java dispose de la spécification JavaBeans [JavaBeans] [Horstmann et Cornell v.1, 2001] qui permet de concevoir des composants modulables. Selon Sun Microsystems, JavaBeans est :

> Un modèle de composant portable et indépendant de la plateforme qui est écrit dans le langage de programmation Java. L'architecture JavaBeans est le fruit de la collaboration de l'industrie et permet aux développeurs de concevoir des composants réutilisables, dans le langage Java. Les composants JavaBeans sont appelés des Beans. Les Beans sont dynamiques, au sens où ils peuvent être modifiés et configurés. NDLR : traduit de l'Anglais [JavaBeans Trail].

Les Beans sont souvent intégrés à un EDI. Ceux-ci possèdent des fenêtres de propriétés qui permettent de configurer les Beans et de préserver leurs états, grâce à des manipulations visuelles. Il est aussi possible de sélectionner le Bean d'un EDI, de le placer dans une fenêtre, de modifier son apparence et son comportement, de définir ses interactions et de l'associer à un autre Bean ou une application Java. Voici quelques caractéristiques inhérentes aux Beans [JavaBeans Concept] :

- Une application Java ou un EDI peut connaître les caractéristiques d'un Bean, telles ses propriétés, ses méthodes et ses événements, grâce à un mécanisme appelé *introspection*.
- Les propriétés définissent l'apparence et le comportement des Beans. Elles peuvent être modifiées par une application Java. Par exemple, lorsqu'un Bean est ajouté à un EDI, celui-ci

effectue l'*introspection* du Bean, pour connaître ses propriétés et les exposer à des fins de manipulation.

- Les Beans font appels à des événements pour communiquer entre eux. Un Bean reçoit les *événements* (Bean *écouteur*) *émis par un autre* Bean (Bean *source*). Un EDI peut examiner quel événement le Bean déclenche (émet) et quel événement il traite (reçoit).
- La *persistance* permet de sauvegarder et restaurer l'état d'un Bean. Pour ce faire, JavaBeans utilise un processus appelé *sérialisation* des objets (de l'Anglais « Java Object Serialization ») [Java 2 SE v.1.4.2 Serialization].

Certaines caractéristiques des Beans nous ont été forts utiles, dans le cadre de la conception de notre modèle. Plus particulièrement, lors de l'intégration du modèle à la coquille de développement de système expert Jess [Jess]. Ce concept a été abordé à la section 3.4.4.5.

Outil UML

Un outil UML est une « application qui supporte les notations et la sémantique du langage UML (« Unified Modeling Language » voir section 1.6.1) de l'OMG (« Object Management Group »), c'est-à-dire un langage de modélisation générique et orienté objet, qui est utilisé en ingénierie logicielle. » NDLR traduit de l'Anglais [UML Tool]. Les outils UML permettent de concevoir des modèles UML, voir 3.4.1. Généralement les outils UML possèdent l'une ou plusieurs des caractéristiques suivantes :

- Conception de diagrammes UML : Un outil UML permet de créer et d'éditer des diagrammes qui suivent la notation graphique du langage UML [UML Tool]. Il existe plusieurs types de diagrammes UML, citons notamment : les diagrammes de classes, dont il a été question à la section 3.4.1 , les diagrammes d'état, les diagrammes d'activité ou les cas d'utilisation (de l'Anglais « use case ») [Fowler, 2004].
- Génération du code source : Cette caractéristique fait en sorte que « l'outil UML produit des parties ou l'ensemble du code source d'un programme, à partir de diagrammes UML créés par l'utilisateur. » NDLR : traduit de l'Anglais [UML Tool].

- Rétro-ingénierie : Elle implique que « l'outil UML utilise le code source d'un programme, pour reconstruire les données du modèle et les diagrammes UML correspondants » NDLR : traduit de l'Anglais [UML Tool]. La rétro-ingénierie s'avère donc le complément de la génération du code source. La combinaison de ces deux méthodes est ce qu'on appelle le « round-trip engineering » [Round-trip].
- Échange du modèle et des diagrammes : Cette fonctionnalité permet d'échanger les diagrammes entre outils UML [UML Tool]. À ce titre, l'OMG a introduit le standard XMI (« XML Metadata Interchange ») [XMI] qui permet d'échanger les métadonnées du modèle, autrement dit les informations qui le décrive, à l'aide du format de fichier XML (« Extensible Markup Language ») [XML v.1.1].

Il existe un éventail d'outils UML [List of UML tools]. Celui que nous avons utilisé est eUML2 de Soyatec [eUML2]. eUML2 est en fait un plugiciel de l'EDI Eclipse JDT [Eclipse JDT]. Il supporte donc le langage Java. L'édition gratuite d'eUML2 permet d'effectuer la conception de diagrammes UML, la génération du code source, la rétro-ingénierie, ainsi que l'échange du modèle UML dans le format XMI.

Nous avons principalement utilisé eUML2 pour créer les diagrammes de classes de notre modèle. Ainsi, il est possible de faire évoluer ces diagrammes de façon itérative, tout au long de la conception et de la mise en œuvre du modèle. Les fonctions de rétro-ingénierie d'eUML2 ne sont pas toujours efficaces. Ceci est particulièrement vrai, lorsque l'outil reconstitue les relations interclasses, dont la cardinalité est multiple : 1..* [Fowler, 2004]. Par contre, la génération du code source, à partir des diagrammes de classes, est relativement fonctionnelle.

Le fait que l'environnement de modélisation (eUML2) soit intégré à l'environnement de développement (Eclipse JDT) constitue également un avantage. Ceci nous permet d'exploiter les forces d'un IDE, tout en maintenant la cohésion, entre la représentation UML du modèle et son implémentation en Java. Cette cohésion est possible, grâce aux notions de génération du code

source et de rétro-ingénierie misent en œuvre par eUML2. Ainsi, les changements apportés au code source, se répercutent automatiquement sur la représentation UML du modèle et vice-versa.

Mentionnons que les outils UML s'inscrivent dans une nouvelle tendance du monde de la conception logicielle, à savoir la séparation entre la représentation d'un système, telle un diagramme UML et son implémentation, par exemple un programme en Java. Idéalement, il deviendrait donc possible de concevoir un système, à partir de son modèle, sans avoir besoin de programmer une ligne de code source [UML Tool]. Cette tendance a été largement soutenue par l'OMG, notamment en proposant une nouvelle approche de conception logicielle, dite approche MDA (« Model-driven Architecture »). Celle-ci a pour but : « de séparer la conception, l'architecture et les aspects technologiques, afin qu'ils puissent être modifiés indépendamment » NDLR : traduit de l'Anglais [Model-driven Architecture]. Cependant, les outils UML ne couvrent qu'une partie de l'approche MDA, en introduisant une certaine séparation entre le modèle UML et son implémentation [UML Tool].

Interface graphique

Une interface graphique animée permet entre autre, de déboguer le modèle, de faciliter le processus de validation, en plus de mettre en évidence l'évolution des éléments clés qui contribuent à l'amélioration du système [Law et Kelton, 2000]. Nous avons donc choisi une animation *concurrente* (voir la section 1.8.2) où l'interface change à mesure que la simulation avance.

La JFC, acronyme de « Java Foundation Class », est un cadriciel qui permet de concevoir des interfaces graphiques et des animations basées sur la plate-forme Java [JFC]. La librairie Swing de la JFC intègre d'ailleurs des composants graphiques, tels des boutons, des boîtes de textes ou des panneaux, qui peuvent être insérés au sein de la fenêtre d'un programme Java [JFC and Swing].

Puisque les étapes de *formation, manutention et entreposage des anodes* sont constituées d'un ensemble de ressources (postes) reliées par des files (convoyeurs), un graphe dirigé [Cormen

et al., 1994] semble être la structure visuelle idéale pour représenter les objets du système. C'est pourquoi nous avons eu recours à un composant Swing appelé JGraph [JGraph]. Celui-ci permet de visualiser un graphe dirigé. Chaque sommet (ressource) du graphe contient certaines mesures de performances. Chaque arête (file) affiche le nombre d'entités en cours de transit. Les principaux indicateurs sont mis à jour progressivement durant la simulation.

Puisque les composants Swing de l'interface graphique doivent afficher de façon continue les indicateurs du système, l'exécution du système expert et de la simulation s'effectue en arrière-plan, au sein du « background thread » [JFC / Swing concurrency]. Un « thread », appellation anglaise de processus léger ou fil d'exécution, « est une façon de diviser l'exécution d'un programme en plusieurs tâches pseudo-simultanées » NDLR : traduit de l'Anglais [Thread]. En Java, le « background thread » permet d'effectuer des tâches qui consomment beaucoup de temps d'exécution [JFC / Swing concurrency], comme la logique du modèle [Pidd, 1998] ou le moteur d'inférence du système expert [Friedman-Hill, 2003]. Ainsi, il devient possible d'avoir une interface qui répond aux événements initiés par l'utilisateur, tels les clics sur le bouton pour interrompre la simulation ou sur celui qui ferme la fenêtre de simulation.

Cadriciel et libraire de simulation d'événements discrets

Un autre avantage de Java est la possibilité d'intégrer le modèle, avec des cadriciels ou des librairies de simulation d'événements discrets [Law et Kelton, 2000]. Ces cadriciels ou librairies sont en fait un ensemble de paquetages Java. Ceux-ci peuvent être référencés par un programme qui met en œuvre le modèle conceptuel d'un système.

Il existe un vaste éventail de cadriciels et de librairies de simulation discrètes en Java qui sont issus du milieu académique. Certaines d'entre elles sont spécialisées pour des applications précises, telles des systèmes qui s'appuient sur des files d'attentes ou encore, des réseaux d'ordinateurs [L'Écuyer et al., 2002]. Pour notre part, nous avons principalement étudié un cadriciel appelé SSJ (Simulation Stochastique en Java ou « Stochastic Simulation in Java »). Il a été conçu par une équipe du Département d'Informatique et de Recherche Opérationnelle (DIRO) de

l'Université de Montréal [SSJ]. SSJ s'apparente à un cadriciel commercial bien connu, nommé Silk. Celui-ci a été développé par ThreadTec Inc. [Kilgore, 2000].

Les classes des paquetages de SSJ comportent des outils qui permettent de générer des nombre aléatoires, pour plusieurs types de distributions de probabilités, de rassembler des données statistiques, de gérer une horloge de simulation et une liste d'événements futurs, de synchroniser les interactions entre des processus concurrents, etc. [L'Écuyer et al., 2002]. Le cadriciel supporte des méthodes de simulation de type planification d'événements, interaction de processus [Pidd, 1998] et la modélisation de phénomènes continus [Law et Kelton, 2000]. Rappelons que ces derniers impliquent que les variables du modèle évoluent selon des équations différentielles.

ANNEXE 4

BASE DE CONNAISSANCES FORMALISÉE AVEC DES RÈGLES DU LANGAGE JESS

Annexe 4 : Base de connaissances formalisée avec des règles du langage Jess

```
; Importe pour définir les constantes.
(import process.Crane)
(import process.Furnace)
; Importe pour appeler des méthodes.
(import process.Anode)
(import process.ProcessedEntity)
; Importe pour imprimer un message dans un fichier.
(import utility.MessagePrintWriter)
; Requière les classes java du modèle.
(require* MAIN::Clock "knowledge/initialize.clp")
(require* MAIN::AncestorRessource "knowledge/initialize.clp")
(require* MAIN::Ressource "knowledge/initialize.clp")
(require* MAIN::Station "knowledge/initialize.clp")
(require* MAIN::Furnace "knowledge/initialize.clp")
(require* MAIN::Crane "knowledge/initialize.clp")
(require* MAIN::ScheduledEvent "knowledge/initialize.clp")
(require* MAIN::Schedule "knowledge/initialize.clp")
(require* MAIN::ProcessedEntity "knowledge/initialize.clp")
(require* MAIN::Anode "knowledge/initialize.clp")
(require* MAIN::EntitiesQueue "knowledge/initialize.clp")
(require* MAIN::EventDescription "knowledge/initialize.clp")
(require* MAIN::EntityState "knowledge/initialize.clp")
(require* MAIN::Rod "knowledge/initialize.clp")
(require* MAIN::Warehouse "knowledge/initialize.clp")
(require* MAIN::FurnacesStatus "knowledge/initialize.clp")
(require* MAIN::WarehouseStatus "knowledge/initialize.clp")
(require* MAIN::AnodicSet "knowledge/initialize.clp")

; Constantes des grues.
(defglobal ?*idle* = (Crane.IDLE))
(defglobal ?*retrieve-baked* = (Crane.RETRIEVE_BAKED))
(defglobal ?*store-baked* = (Crane.STORE_BAKED))
(defglobal ?*store-green* = (Crane.STORE_GREEN))
(defglobal ?*retrieve-green* = (Crane.RETRIEVE_GREEN))
; Constantes des fours.
(defglobal ?*produce-not-much* = (Furnace.PRODUCE_NOT_MUCH))
(defglobal ?*need-entity* = (Furnace.NEED_ANODE))

(deffunction printOut ($?message)
    ;; Imprime le message avec les routeurs.
    (foreach ?token ?message
        ;; Imprime une partie du message avec le routeur par défaut.
        (printout t ?token)
        ;; Imprime une partie du message avec le routeur du fichier.
        (printout WFILE ?token)
        ;; Imprime une partie du message avec le routeur du JTextArea.
        (printout WTEXT ?token)
        )
    )

(deffunction printFacts (?module)
```

```
        "Imprime les faits du module concerné."
        (if (neq ?module nil) then
            (bind ?previousModule (get-current-module))
            (set-current-module ?module)
            (facts)
            (set-current-module ?previousModule)
            else
            (printOut (create$ "Error : nil argument in printFacts." crlf))
            )
        )

    (deffunction printObjectMessage (?object ?message)
        (if (neq ?object nil) then
            ;; Initialise le type d'objet et son nom.
            (bind ?objectType "")
            (bind ?name "")
            ;; Si c'est une resssource.
            (if (instanceof ?object process.RessourceAncestor) then
                ;; On obtient le nom de la ressource.
                (bind ?name (str-cat ", nom [" (call ?object getName) "]"))
                ;; On recherche de quel type est la ressource.
                (if (instanceof ?object process.Crane) then
                    (bind ?objectType "Grue")
                    else
                    (if (instanceof ?object process.Furnace) then
                        (bind ?objectType "Four")
                        else
                        (if (instanceof ?object process.Station) then
                            (bind ?objectType "Poste")
                            else
                            (if (instanceof ?object process.Ressource) then
                                (bind ?objectType "Ressource")))))
                else ;; Sinon ce n'est pas une ressource.
                ;; Si l'objet est une entité.
                (if (instanceof ?object process.ProcessedEntity) then
                    (bind ?objectType "Entité")))
            ;; Imprime le message.
            (printOut (create$ "****" crlf
                    ?objectType " ID [" (call ?object getId) "]"
                    ?name crlf
                    ?message crlf
                    "après " (fetch elapsedTime) " seconde(s) de simulation."
                    crlf "****" crlf))
            else ;; L'objet est null.
            (printOut (create$ "****" crlf
                    "Error : nil ?object in printObjectMessage."
                    crlf "****" crlf)))
        )

    (defmodule STATUS)

    (defrule STATUS::disable-ressource-ancestor
        "Désactive la ressource dont l'arrêt débute."
```

```
        (MAIN::Clock
            (elapsedTime ?elapsedTime))
        (MAIN::Schedule (startsAt ?elapsedTime)
            (scheduledEvent ?event)
            (ressource ?ressource)
            (OBJECT ?schedule))
        (MAIN::ScheduledEvent
            (id 1)
            (OBJECT ?event))
        (MAIN::RessourceAncestor
            (disabled FALSE)
            (currentSchedule nil)
            (OBJECT ?ressource))
        =>
        ;;(printFacts MAIN)
        (call ?ressource setDisabled TRUE)
        (call ?ressource setCurrentSchedule ?schedule)
        ;;(printObjectMessage ?ressource "a été *DÉSACTIVÉ*,")
        ;;(printFacts MAIN)
        )

    (defrule STATUS::enable-ressource-ancestor
        "Active la ressource dont l'arrêt se termine."
        (MAIN::Clock
            (elapsedTime ?elapsedTime))
        (MAIN::Schedule
            (finishesAt ?elapsedTime)
            (scheduledEvent ?event)
            (ressource ?ressource)
            (OBJECT ?schedule))
        (MAIN::ScheduledEvent
            (id 1)
            (OBJECT ?Event))
        (MAIN::RessourceAncestor
            (disabled TRUE)
            (currentSchedule ?schedule)
            (OBJECT ?ressource))
        =>
        ;;(printFacts MAIN)
        (call ?ressource setDisabled FALSE)
        (call ?ressource setCurrentSchedule nil)
        ;;(printObjectMessage ?ressource "a été *ACTIVÉ*,")
        ;;(printFacts MAIN)
        )

    (defmodule SIGNAL)

    (defrule SIGNAL::furnaces-produce-not-much-1
        "Si la file 910 n'est pas pleine,
        alors les fours produisent peu."
        (not (MAIN::FurnacesStatus
                (signal ?*produce-not-much*)))
        (MAIN::EntitiesQueue
```

```
        (id 910)
        (full FALSE))
    =>
    (assert (MAIN::FurnacesStatus (signal ?*produce-not-much*)))
    )

(defrule SIGNAL::furnaces-produce-not-much-2
    "Si les files 89 et 49 sont pleines à moins
    de 50%, alors les fours produisent peu."
    (not (MAIN::FurnacesStatus
            (signal ?*produce-not-much*)))
    (MAIN::EntitiesQueue (id 89)
        (downstreamPriorityRatio ?ratio&:(< ?ratio 0.5)))
    (MAIN::EntitiesQueue (id 49)
        (downstreamPriorityRatio ?ratio&:(< ?ratio 0.5)))
    =>
    (assert (MAIN::FurnacesStatus (signal ?*produce-not-much*)))
    )


(defrule SIGNAL::furnaces-need-entity
    "Si la file 34 est pleine à moins de 75% ou bien
    si la file 37 est pleine à moins de 79%, alors
    les fours ont besoins d'anodes."
    (not (MAIN::FurnacesStatus
            (signal ?*need-entity*)))
    (or (MAIN::EntitiesQueue (id 34)
            (downstreamPriorityRatio ?ratio&:(< ?ratio 0.75)))
        (MAIN::EntitiesQueue (id 37)
            (downstreamPriorityRatio ?ratio&:(< ?ratio 0.79))))
    =>
    (assert (MAIN::FurnacesStatus (signal ?*need-entity*)))
    )

(defrule SIGNAL::warehouse-retrieves-baked
    "Si les fours produisent peu et que le scellement est
    activé, alors on récupère des anodes cuites."
    (not (MAIN::WarehouseStatus
            (signal ?*retrieve-baked*)))
    (MAIN::FurnacesStatus
        (signal ?*produce-not-much*))
    (MAIN::Station
        (id 13)
        (disabled FALSE))
    =>
    (assert (MAIN::WarehouseStatus (signal ?*retrieve-baked*)))
    )

(defrule SIGNAL::warehouse-stores-baked
    "Si le scellement est désactivé et que la file 1013
    (en amont du scellement) est pleine, alors on entrepose
    des anodes cuites."
    (not (MAIN::WarehouseStatus
```

```
            (signal ?*store-baked*)))
    (MAIN::Station
        (id 13)
        (disabled TRUE))
    (MAIN::EntitiesQueue
        (id 1013)
        (full TRUE))
    =>
    (assert (MAIN::WarehouseStatus (signal ?*store-baked*)))
    )

(defrule SIGNAL::warehouse-stores-green
    "S'il y a une anode disponible sur la file 312
     (anodes crues) en amont de l'entrepôt,
    alors on entrepose des anodes crues."
    (not (MAIN::WarehouseStatus
            (signal ?*store-green*)))
    (MAIN::EntitiesQueue
        (id 312)
        (entityAvailable TRUE))
    =>
    (assert (MAIN::WarehouseStatus (signal ?*store-green*)))
    )

(defrule SIGNAL::warehouse-retrieves-green-1
    "Si les fours ont besoins d'anodes et
    que le poste 3 est désactivé, alors on
    récupère des anodes crues."
    (not (MAIN::WarehouseStatus
            (signal ?*retrieve-green*)))
    (MAIN::FurnacesStatus
        (signal ?*need-entity*))
    (MAIN::Station
        (id 3)
        (disabled TRUE))
    =>
    (assert (MAIN::WarehouseStatus (signal ?*retrieve-green*)))
    )

(defrule SIGNAL::warehouse-retrieves-green-2
    "Si les fours ont besoins d'anodes,
    que les files 13 et 23 n'ont pas
    d'entité disponible, alors on récupère
    des anodes crues."
    (not (MAIN::WarehouseStatus
            (signal ?*retrieve-green*)))
    (MAIN::FurnacesStatus
        (signal ?*need-entity*))
    (MAIN::EntitiesQueue
        (id 13)
        (entityAvailable FALSE))
    (MAIN::EntitiesQueue
        (id 23)
```

```
        (entityAvailable FALSE))
    =>
    (assert (MAIN::WarehouseStatus (signal ?*retrieve-green*)))
    )

(defmodule PROCESS-BAKED)

(defrule PROCESS-BAKED::retrieve-baked
    "S'il y a un signal pour récupérer des anodes cuites,
    qu'aucune grue ne récupère ou n'entrepose déjà des anodes cuites,
    qu'il y a une grue disponible avec un plus petit id (indice)
    possible (car la #1 s'occupe des anodes cuites en priorité)
    et que la file en aval (1110) a suffisamment de place,
    alors on récupère des anodes cuites."
    ?signalFact <-
    (MAIN::WarehouseStatus
        (signal ?*retrieve-baked*))
    (not (MAIN::Crane
            (disabled FALSE)
            (processing TRUE)
            (signal ?*retrieve-baked*|?*store-baked*)))
    (MAIN::Crane
        (id ?id1)
        (disabled FALSE)
        (processing FALSE)
        (entityGap ?gap)
        (signal ?*idle*)
        (OBJECT ?crane))
    (not (MAIN::Crane
            (id ?id2&:(< ?id2 ?id1))
            (disabled FALSE)
            (processing FALSE)))
    (MAIN::EntitiesQueue
        (id 1110)
        (maxEntities ?max)
        (entitiesCount ?count&:(>= (- ?max ?count) ?gap)))
    =>
    ; On a répondu au signal donc on le rétracte.
    (retract ?signalFact)
    (call ?crane setProcessing TRUE)
    (call ?crane setSignal ?*retrieve-baked*)
    (call ?crane resetElapsedTime)
    (call ?crane resetRemainingCount)
    )

(defrule PROCESS-BAKED::store-baked
    "S'il y a un signal pour entreposer des anodes cuites,
    qu'il n'y a aucun signal pour récupérer des anodes cuites,
    qu'aucune grue ne récupère ou n'entrepose déjà des anodes cuites
    et qu'il y a une grue disponible avec le plus petit indice
    possible (car la #1 s'occupe des anodes cuites en priorité)
    et qu'il y a suffisamment d'anodes sur la file amont (1011),
    alors on entrepose des anodes cuites."
```

```
    ?signalFact <-
    (MAIN::WarehouseStatus
        (signal ?*store-baked*))
    (not (MAIN::WarehouseStatus
            (signal ?*retrieve-baked*)))
    (not (MAIN::Crane
            (disabled FALSE)
            (processing TRUE)
            (signal ?*retrieve-baked*|?*store-baked*)))
    (MAIN::Crane
        (id ?id1)
        (disabled FALSE)
        (processing FALSE)
        (entityCapacity ?capacity)
        (signal ?*idle*)
        (OBJECT ?crane))
    (not (MAIN::Crane
            (id ?id2&:(< ?id2 ?id1))
            (disabled FALSE)
            (processing FALSE)))
    (MAIN::EntitiesQueue
        (id 1011)
        (entitiesCount ?count&:(>= ?count ?capacity)))
    =>
    ;;(printFacts MAIN)
    ; On a répondu au signal donc on le rétracte.
    (retract ?signalFact)
    (call ?crane setProcessing TRUE)
    (call ?crane setSignal ?*store-baked*)
    (call ?crane resetElapsedTime)
    (call ?crane resetRemainingCount)
    ;;(printFacts MAIN)
    )

(defmodule PROCESS-GREEN)

(defrule PROCESS-GREEN::store-green
    "S'il y a un signal pour entreposer des anodes crues,
    qu'aucune grue ne récupère ou n'entrepose déjà des anodes crues,
    qu'il y a une grue disponible avec le plus grand indice
    possible (car la #2 s'occupe des anodes crues en priorité)
    et qu'il y a suffisamment d'anodes sur la file amont (312),
    alors on entrepose des anodes crues."
    ?signalFact <-
    (MAIN::WarehouseStatus
        (signal ?*store-green*))
    (not (MAIN::Crane
            (disabled FALSE)
            (processing TRUE)
            (signal ?*retrieve-green*|?*store-green*)))
    (MAIN::Crane
        (id ?id1)
        (disabled FALSE)
```

```
        (processing FALSE)
        (entityCapacity ?capacity)
        (signal ?*idle*)
        (OBJECT ?crane))
    (not (MAIN::Crane
            (id ?id2&:(> ?id2 ?id1))
            (disabled FALSE)
            (processing FALSE)))
    (MAIN::EntitiesQueue
        (id 312)
        (maxEntities ?max)
        (entitiesCount ?count&:
            (>= ?count ?capacity)))
    =>
    ;;(printFacts MAIN)
    ; On a répondu au signal donc on le rétracte.
    (retract ?signalFact)
    (call ?crane setProcessing TRUE)
    (call ?crane setSignal ?*store-green*)
    (call ?crane resetElapsedTime)
    (call ?crane resetRemainingCount)
    ;;(printFacts MAIN)
    )

(defrule PROCESS-GREEN::retrieve-green
    "S'il y a un signal pour récupérer des anodes crues,
    qu'il n'y a aucun signal pour entreposer des anodes crues,
    qu'aucune grue ne récupère ou n'entrepose déjà des anodes crues,
    qu'il y a une grue disponible avec le plus grand indice
    possible (car la #2 s'occupe des anodes crues en priorité)
    et que la file en aval (123) a suffisamment de place,
    alors on récupère des anodes crues."
    ?signalFact <-
    (MAIN::WarehouseStatus
        (signal ?*retrieve-green*))
    (not (MAIN::WarehouseStatus
            (signal ?*store-green*)))
    (not (MAIN::Crane
            (disabled FALSE)
            (processing TRUE)
            (signal ?*retrieve-green*|?*store-green*)))
    (MAIN::Crane
        (id ?id1)
        (disabled FALSE)
        (processing FALSE)
        (entityGap ?gap)
        (signal ?*idle*)
        (OBJECT ?crane))
    (not (MAIN::Crane
            (id ?id2&:(> ?id2 ?id1))
            (disabled FALSE)
            (processing FALSE)))
    (MAIN::EntitiesQueue
```

```
        (id 123)
        (maxEntities ?max)
        (entitiesCount ?count&:
            (>= (- ?max ?count) ?gap)))
    =>
    ; On a répondu au signal donc on le rétracte.
    (retract ?signalFact)
    (call ?crane setProcessing TRUE)
    (call ?crane setSignal ?*retrieve-green*)
    (call ?crane resetElapsedTime)
    (call ?crane resetRemainingCount)
    )

(defmodule PROCESS)

(deffunction getStrId (?object)
    (if (eq ?object nil) then
        (return "NIL")
        else
        (return (call ?object getId))
        )
    )

(deffunction getNewAnode(?ressource ?outputStates ?queue)

    ;; Constructeur d'anode avec clé affectée par le simulateur :
    ;; Anode(boolean queued, int elapsedTime, Map states,
    ;;             Ressource ressource, EntitiesQueue queue, AnodicSet
aAnodicSet)
    (bind ?queued (neq ?queue nil))
    (bind ?anode (new Anode ?queued 0 ?outputStates ?ressource ?queue
nil))
    (call ProcessedEntity addProcessedEntity (?anode getId) ?anode)
    (definstance MAIN::Anode ?anode dynamic)
    (return ?anode)
    )

(deffunction startProcessingEntity (?queue ?ressource ?entity)
    "Débute le traitement d'une entité en l'assignant à un poste."
    (call ?entity resetElapsedTime)
    (call ?entity setQueued FALSE)
    (call ?entity setRessource ?ressource)
    (call ?entity setEntitiesQueue nil)

    (call ?ressource setProcessing TRUE)
    (call ?ressource setProcessedEntity ?entity)

    (bind ?message (str-cat "a *COMMENCÉ À TRAITER* une entité "
            "provenant de la file ["
            (getStrId ?queue) "],"))
    (printObjectMessage ?ressource ?message)
    )
```

```
(deffunction stopProcessingEntity (?queue ?ressource ?entity)
    "Termine le traitement d'une entité en l'assignant à une file."
    (call ?entity resetElapsedTime)
    (call ?entity setQueued (neq ?queue nil))
    (call ?entity setRessource nil)
    (call ?entity setEntitiesQueue ?queue)

    (call ?ressource setProcessing FALSE)
    (call ?ressource setProcessedEntity nil)

    (bind ?message (str-cat "a *ARRÊTÉ DE TRAITER* une entité "
            "qui a été ajoutée "
            "à la file [" (getStrId ?queue) "],"))
    (printObjectMessage ?ressource ?message)
    )

(defrule PROCESS::furnace-train-arrival
    "Un train d'anodes arrive au four."
    (MAIN::Clock
        (elapsedTime ?elapsedTime))
    ?scheduleFact <-
    (MAIN::Schedule
        (startsAt ?startsAt&:(>= ?elapsedTime ?startsAt))
        (scheduledEvent ?event)
        (ressource ?furnace)
        (OBJECT ?schedule))
    (MAIN::Furnace
        (disabled FALSE)
        (OBJECT ?furnace))
    (MAIN::ScheduledEvent
        (id 3)
        (OBJECT ?event))
    =>
    ;;(printFacts MAIN)
    (call ?furnace incBakedCount)
    (retract ?scheduleFact)
    (bind ?message (str-cat "a reçu un train."))
    (printObjectMessage ?furnace ?message)
    ;;(printFacts MAIN)
    )

(defrule PROCESS::first-station-starts-processing-1
    "Si plus d'un poste à l'entrée du processus est en attente,
     on choisit de commencer le traitement avec celui qui a le
     moins d'anodes dans sa file en aval (i.e. celle qui a le
     ratio le plus petit)."
    (MAIN::Station
        (id ?id1)
        (disabled FALSE)
        (processing FALSE)
        (entityTypeCreated "Anode")
        (downstreamQueues nil)
        (outputStates ?outputStates)
```

```
        (OBJECT ?station1))
    (MAIN::Station
        (id ?id2&~?id1)
        (disabled FALSE)
        (processing FALSE)
        (entityTypeCreated "Anode")
        (downstreamQueues nil)
        (OBJECT ?station2))
    (MAIN::EntitiesQueue
        (full FALSE)
        (downstreamPriorityRatio ?ratio1)
        (downstreamRessource ?station1))
    (not (MAIN::EntitiesQueue
            (full FALSE)
            (downstreamPriorityRatio ?ratio2&:(< ?ratio2 ?ratio1))
            (downstreamRessource ?station2)))
    =>
    ;;(printFacts MAIN)
    (bind ?anode (getNewAnode ?station1 ?outputStates nil))
    (startProcessingEntity nil ?station1 ?anode)
    (call ?station1 incEntitiesInCount)
    ;;(printFacts MAIN)
    )

(defrule PROCESS::first-station-starts-processing-2
    "Si un poste à l'entrée du processus est en attente et
     qu'il y a : ou bien un autre poste déscativé ou bien
     aucun autre en traitement, alors on commence le
     traitement d'une entité avec ce poste."
    (MAIN::Station
        (disabled FALSE)
        (processing FALSE)
        (entityTypeCreated "Anode")
        (downstreamQueues nil)
        (outputStates ?outputStates)
        (OBJECT ?station))
    (or (MAIN::Station
            (disabled TRUE)
            (entityTypeCreated "Anode")
            (downstreamQueues nil))
        (not (MAIN::Station
                (processing TRUE)
                (entityTypeCreated "Anode")
                (downstreamQueues nil))))
    =>
    ;;(printFacts MAIN)
    (bind ?anode (getNewAnode ?station ?outputStates nil))
    (startProcessingEntity nil ?station ?anode)
    (call ?station incEntitiesInCount)
    ;;(printFacts MAIN)
    )
```

```
(defrule PROCESS::furnace-starts-processing
    "Si le four est activé, n'est pas en traitement
    et qu'au moins une entité a été débarquée d'un train,
    alors on prélève une entité sur sa file en amont
    (n.b. un four n'a qu'une file en amont)."
    (MAIN::Furnace
        (disabled FALSE)
        (processing FALSE)
        (downstreamQueues ~nil)
        (bakedCount ?count&:(> ?count 0))
        (OBJECT ?furnace))
    (MAIN::EntitiesQueue
        (entityAvailable TRUE)
        (firstEntity ?entity)
        (upstreamRessource ?furnace)
        (OBJECT ?queue))
    =>
    ;;(printFacts MAIN)
    (startProcessingEntity ?queue ?furnace ?entity)
    (call ?queue removeProcessedEntity ?entity)
    ;;(printFacts MAIN)
    )

(defrule PROCESS::station-starts-processing-1
    "Si le poste est prêt à être mis en traitement et
    qu'il n'y a pas d'autre file disponible avec un
    plus grand ratio (i.e. pas d'autre file plus pleine),
    alors on choisit de traiter une anode provenant
    de cette file."
    (MAIN::Station
        (disabled FALSE)
        (processing FALSE)
        (downstreamQueues ~nil)
        (OBJECT ?station))
    (MAIN::EntitiesQueue
        (entityAvailable TRUE)
        (firstEntity ?entity)
        (upstreamPriorityRatio ?ratio1&:(> ?ratio1 0))
        (upstreamRessource ?station)
        (OBJECT ?queue1))
    (not
        (MAIN::EntitiesQueue
            (entityAvailable TRUE)
            (upstreamPriorityRatio ?ratio2&:(> ?ratio2 ?ratio1))
            (upstreamRessource ?station)
            (OBJECT ?queue2)))
    =>
    ;;(printFacts MAIN)
    (startProcessingEntity ?queue1 ?station ?entity)
    (call ?queue1 removeProcessedEntity ?entity)
    (call ?station checkNextDownstreamQueue ?queue1)
    ;;(printFacts MAIN)
    )
```

```
(defrule PROCESS::station-starts-processing-2
    "Si le poste est prêt à être mis en traitement,
    que toutes ses files en amont sont disponibles
    (i.e. aucune file avec (entityAvailable FALSE)) et
    pleines (i.e. aucune file avec (full FALSE)), alors
    on choisit de traiter une anode provenant de la
    prochaine file (nextDownstreamQueue)."
    (MAIN::Station
        (disabled FALSE)
        (processing FALSE)
        (downstreamQueues ~nil)
        (nextDownstreamQueue ?queue)
        (OBJECT ?station))
    (MAIN::EntitiesQueue
        (entityAvailable TRUE)
        (full TRUE)
        (firstEntity ?entity)
        (upstreamRessource ?station)
        (OBJECT ?queue))
    (not (MAIN::EntitiesQueue
            (entityAvailable FALSE)
            (upstreamRessource ?station)))
    (not (MAIN::EntitiesQueue
            (full FALSE)
            (upstreamRessource ?station)))
    =>
    ;;(printFacts MAIN)
    (startProcessingEntity ?queue ?station ?entity)
    (call ?queue removeProcessedEntity ?entity)
    (call ?station checkNextDownstreamQueue ?queue)
    ;;(printFacts MAIN)
    )

(defrule PROCESS::furnace-stops-processing
    "Si le four est activé, qu'il est en traitement
    sur une entité depuis suffisamment longtemps et
    que la file en aval n'est pas pleine,
    alors on envoie l'entité sur sa file en aval
    (n.b. un four n'a qu'une file en aval)."
    (MAIN::ProcessedEntity
        (queued FALSE)
        (ressource ?furnace)
        (elapsedTime ?elapsedTime)
        (OBJECT ?entity))
    (MAIN::Furnace
        (disabled FALSE)
        (processing TRUE)
        (upstreamQueues ~nil)
        (processingTime ?processingTime&:(>= ?elapsedTime
?processingTime))
        (OBJECT ?furnace))
    (MAIN::EntitiesQueue
```

```
        (full FALSE)
        (downstreamRessource ?furnace)
        (OBJECT ?queue))
    =>
    ;;(printFacts MAIN)
    (call ?queue addProcessedEntity ?entity)
    (stopProcessingEntity ?queue ?furnace ?entity)
    (call ?furnace decBakedCount)
    ;;(printFacts MAIN)
    )

(defrule PROCESS::station-stops-processing
    "Arrête le poste en traitement et envoie l'entité vers
    la file en aval la moins pleine (ratio le plus petit)."
    (MAIN::ProcessedEntity
        (queued FALSE)
        (ressource ?station)
        (elapsedTime ?elapsedTime)
        (OBJECT ?entity))
    (MAIN::Station
        (disabled FALSE)
        (processing TRUE)
        (upstreamQueues ~nil)
        (processingTime ?processingTime
            &:(>= ?elapsedTime ?processingTime))
        (OBJECT ?station))
    (MAIN::EntitiesQueue
        (full FALSE)
        (downstreamRessource ?station)
        (downstreamPriorityRatio ?ratio1)
        (OBJECT ?queue1))
    (not
        (MAIN::EntitiesQueue
            (full FALSE)
            (downstreamRessource ?station)
            (downstreamPriorityRatio ?ratio2&:(< ?ratio2 ?ratio1))
            (OBJECT ?queue2)))
    =>
    ;;(printFacts MAIN)
    (call ?queue1 addProcessedEntity ?entity)
    (stopProcessingEntity ?queue1 ?station ?entity)
    ;;(printFacts MAIN)
    )

(defrule PROCESS::last-station-stops-processing
    "Arrête un poste sans ligne en aval, en cours de
    traitement sur une entité."
    ?factEntity <-
    (MAIN::ProcessedEntity
        (queued FALSE)
        (ressource ?station)
        (elapsedTime ?elapsedTime)
        (OBJECT ?entity))
```

```
    (MAIN::Station
        (disabled FALSE)
        (processing TRUE)
        (upstreamQueues nil)
        (processingTime ?processingTime
            &:(>= ?elapsedTime ?processingTime))
        (OBJECT ?station))
    =>
    ;;(printFacts MAIN)
    (call ?station incEntitiesOutCount)
    (stopProcessingEntity nil ?station ?entity)
    (retract ?factEntity)
    ;;(printFacts MAIN)
    )

(defrule PROCESS::crane-retrieves-entity
    "Si la grue est active, qu'elle est en train de récupérer
    des anodes cuites ou crues, qu'elle effectue le traitement depuis
    assez longtemps, qu'il reste des anodes à récupérer et qu'il y a
    assez de place sur la file associée à la récupération, alors on
    récupère une anode."
    (MAIN::Crane
        (disabled FALSE)
        (processing TRUE)
        (signal ?signal&?*retrieve-baked*|?*retrieve-green*)
        (elapsedTime ?elapsedTime)
        (processingTime ?processingTime
            &:(>= ?elapsedTime ?processingTime))
        (remainingCount ?remainingCount
            &:(> ?remainingCount 0))
        (outputStates ?outputStates)
        (warehouse ?warehouse&~nil)
        (OBJECT ?crane))
    (MAIN::EntitiesQueue
        (signal ?signal)
        (maxEntities ?maxEntities)
        (entitiesCount ?entitiesCount
            &:(>= (- ?maxEntities ?entitiesCount) ?remainingCount))
        (OBJECT ?queue))
    =>
    (bind ?anode (getNewAnode nil ?outputStates ?queue))
    (?queue addProcessedEntity ?anode)
    (?crane decRemainingCount)
    (?warehouse handleSignal ?signal)
    (bind ?message (str-cat "a traitée le signal ["
            (call Crane getSignalDescription ?signal) "] "
            "*EN RÉCUPÉRANT* une anode, "
            "à partir de la file ID [" (?queue getId) "]"))
    (printObjectMessage ?crane ?message)
    )

(defrule PROCESS::crane-stores-entity
    "Si la grue est active, qu'elle est en train d'entreposer
```

```
    des anodes cuites ou crues, qu'elle effectue le traitement depuis
    assez longtemps, qu'il reste des anodes à entreposer et qu'il y a
    assez de place sur la file associée à l'entreposage, alors on
    entrepose une anode."
    (MAIN::Crane
        (disabled FALSE)
        (processing TRUE)
        (signal ?signal&?*store-baked*|?*store-green*)
        (elapsedTime ?elapsedTime)
        (processingTime ?processingTime
            &:(>= ?elapsedTime ?processingTime))
        (remainingCount ?remainingCount
            &:(> ?remainingCount 0))
        (outputStates ?outputStates)
        (warehouse ?warehouse&~nil)
        (OBJECT ?crane))
    (MAIN::EntitiesQueue
        (signal ?signal)
        (entitiesCount ?entitiesCount
            &:(>= ?entitiesCount ?remainingCount))
        (firstEntity ?entity)
        (OBJECT ?queue))
    ?entityFact <-
    (MAIN::ProcessedEntity
        (OBJECT ?entity))
    =>
    ;(printFacts MAIN)
    (retract ?entityFact)
    (?queue removeProcessedEntity ?entity)
    (?crane decRemainingCount)
    (?warehouse handleSignal ?signal)
    (bind ?message
        (str-cat "a traitée le signal ["
            (call Crane getSignalDescription ?signal) "] "
            "*EN ENTREPOSANT* une entité, "
            "à partir de la file ID [" (?queue getId) "]"))
    (printObjectMessage ?crane ?message)
    ;(printFacts MAIN)
    )

(defrule PROCESS::crane-stops-processing
    "Si la grue est active, qu'elle est en traitement depuis
    assez longtemps et que les anodes *ont toutes* été
    entreposées ou récupérées, alors on arrête le traitement."
    (MAIN::Crane
        (disabled FALSE)
        (processing TRUE)
        (signal ?signal&~?*idle*)
        (elapsedTime ?elapsedTime)
        (processingTime ?processingTime
            &:(>= ?elapsedTime ?processingTime))
        (entityCapacity ?capacity)
        (remainingCount ?remainingCount
```

```
            &:(<= ?remainingCount 0))
        (OBJECT ?crane))
    =>
    (?crane resetElapsedTime)
    (?crane resetRemainingCount)
    (?crane setProcessing FALSE)
    (?crane setSignal ?*idle*)
    (bind ?message (str-cat "a cessée de traiter le signal ["
            (call Crane getSignalDescription ?signal) "]"))
    (printObjectMessage ?crane ?message)
    )

(defmodule FINALIZE)

(defrule FINALIZE::increment-crane-elapsed-time
    "Incrémente le temps d'opération écoulé pour cette grue."
    (MAIN::Clock
        (elapsedTime ?elapsedTime))
    (MAIN::Crane
        (processing TRUE)
        (disabled FALSE)
        (signal ~?*idle*)
        (OBJECT ?crane))
    =>
    ;;(printFacts MAIN)
    (call ?crane incElapsedTime)
    (bind ?message (str-cat "a été *INCRÉMENTÉE* d'un pas de temps"))
    (printObjectMessage ?crane ?message)
    ;;(printFacts MAIN)
    )

(defrule FINALIZE::increment-queued-entity-elapsed-time
    "Incrémente le temps passé dans la file pour une entité."
    (MAIN::Clock
        (elapsedTime ?elapsedTime))
    (MAIN::ProcessedEntity
        (queued TRUE)
        (entitiesQueue ?queue)
        (OBJECT ?entity))
    (MAIN::EntitiesQueue
        (id ?id)
        (OBJECT ?queue))
    =>
    (call ?entity incElapsedTime)
    (call ?entity checkAvailability)
    (bind ?message (str-cat "a été *INCRÉMENTÉE* d'un pas de temps "
            "lors du transit dans la file ID [" ?id "],"))
    ;;(printObjectMessage ?entity ?message)
    ;;(printFacts MAIN)
    )

(defrule FINALIZE::increment-entity-elapsed-time
    "Incrémente le temps de traitement d'une entité pour une
```

```
        ressource donnée."
    (MAIN::Clock
        (elapsedTime ?elapsedTime))
    (MAIN::ProcessedEntity
        (queued FALSE)
        (ressource ?ressource)
        (OBJECT ?entity))
    (MAIN::Ressource
        (name ?name)
        (processing TRUE)
        (disabled FALSE)
        (OBJECT ?ressource))
    =>
    (call ?entity incElapsedTime)
    (bind ?message (str-cat "a été *INCRÉMENTÉE* d'un pas de temps "
            "du traitement de la ressource [" ?name "],"))
    ;;(printObjectMessage ?entity ?message)
    ;;(printFacts MAIN)
    )

(defrule FINALIZE::retract-warehouse-status
    "On rétracte l'état de l'entrepôt."
    ?statusFact <-
    (MAIN::WarehouseStatus)
    =>
    (retract ?statusFact)
    )

(defrule FINALIZE::retract-furnaces-status
    "On rétracte l'état des fours."
    ?statusFact <-
    (MAIN::FurnacesStatus)
    =>
    (retract ?statusFact)
    )
```