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].

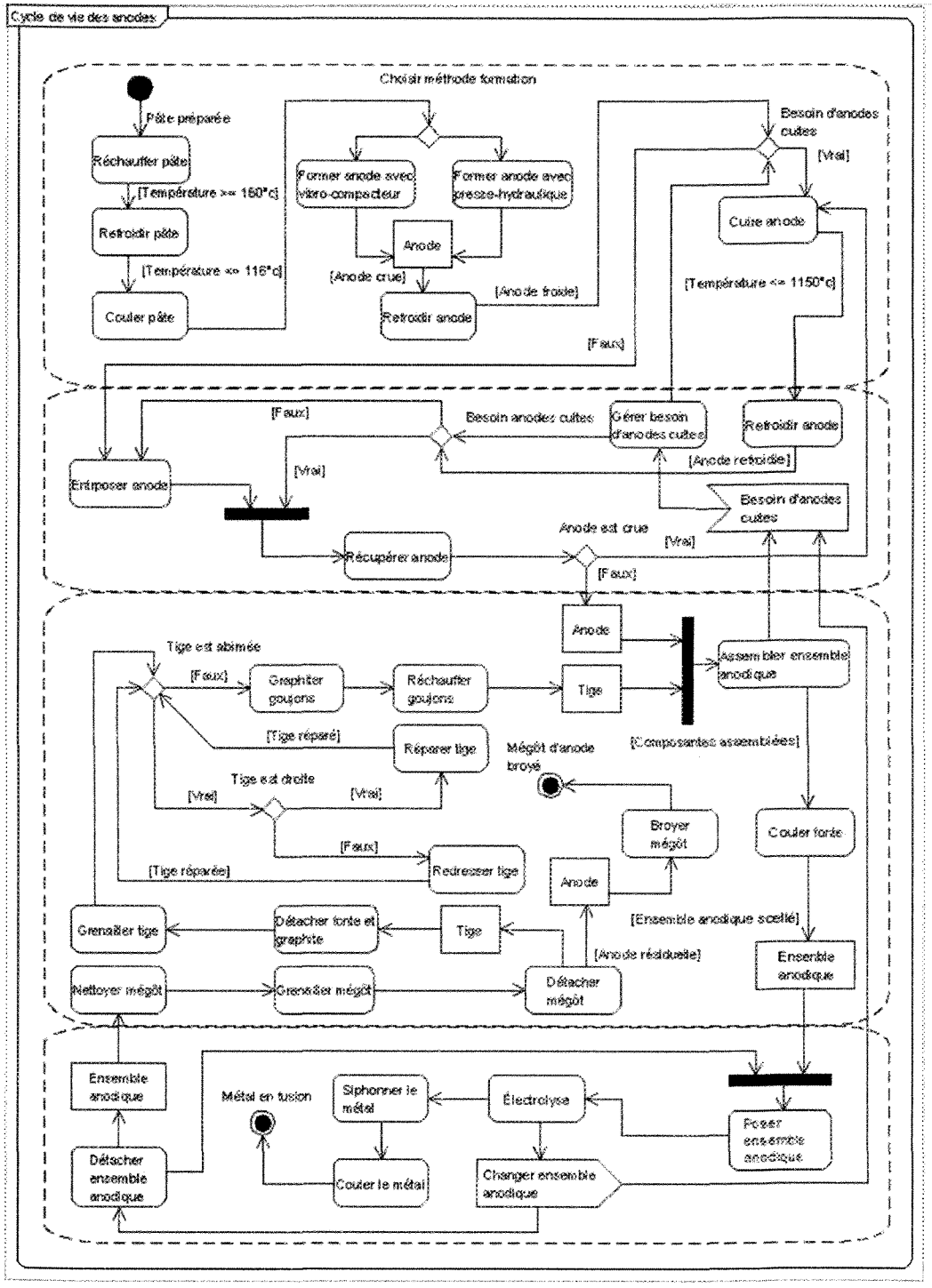

Figure 19 : diagramme des activités du cycle de vie des anodes

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].

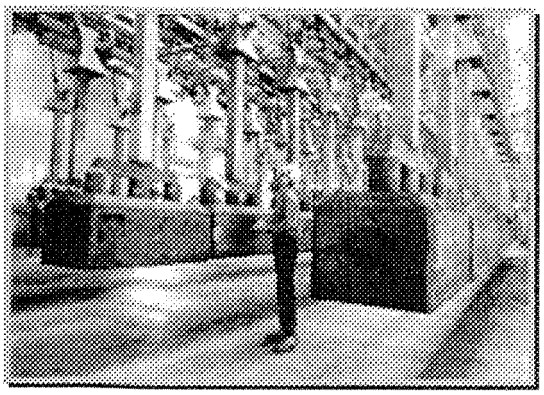

Figure 20 : ensembles anodiques (tige métallique scellée à l'anode cuite)

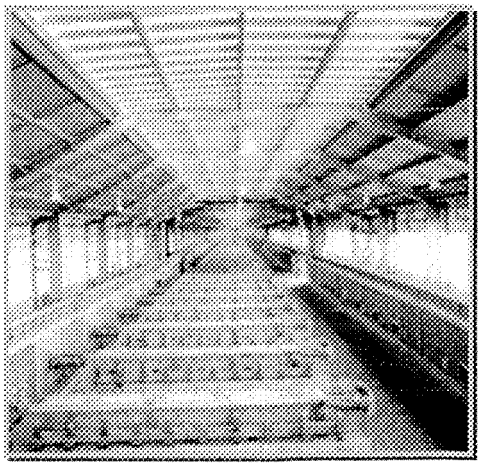

Figure 21 : salle des cuves (cellules de réduction de l'aluminium)

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].

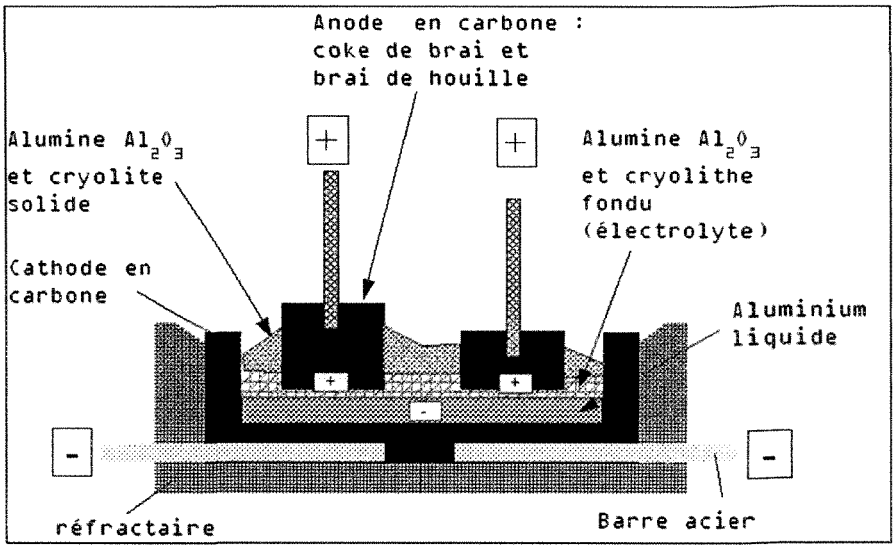

Figure 22 : électrolyse de l'aluminium dans une cellule de réduction

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.

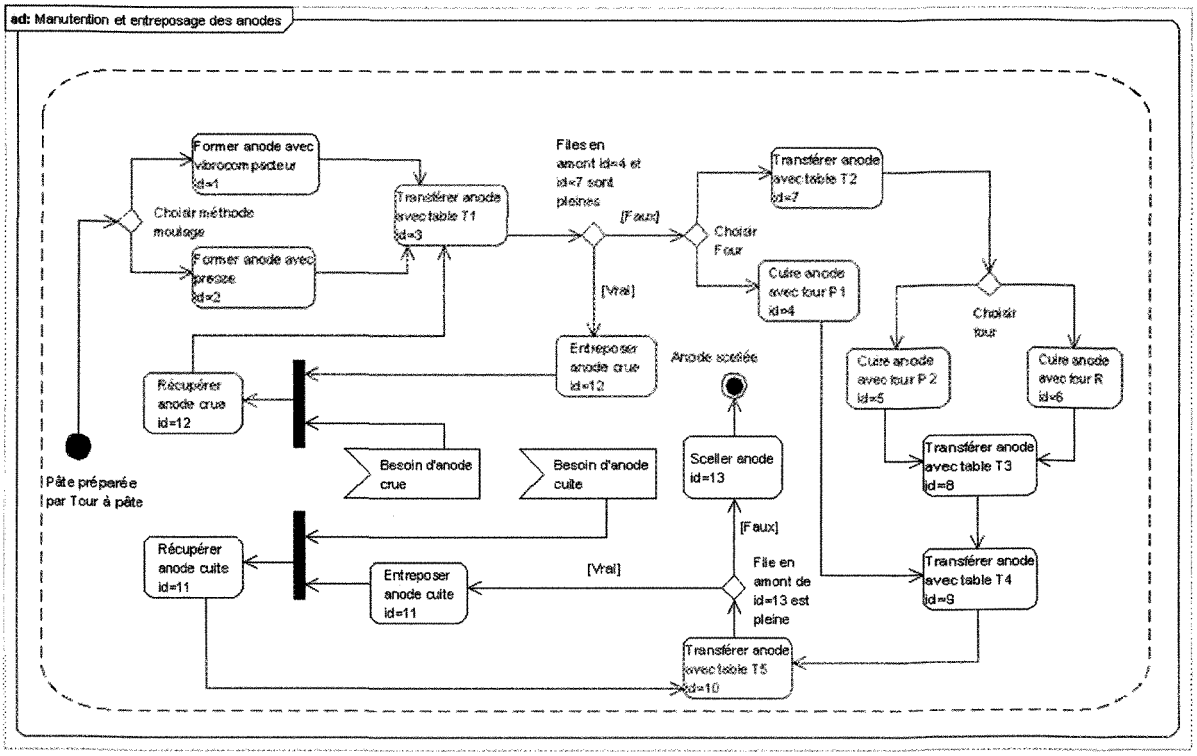

Figure 23 : formation, manutention et entreposage des anodes

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*.

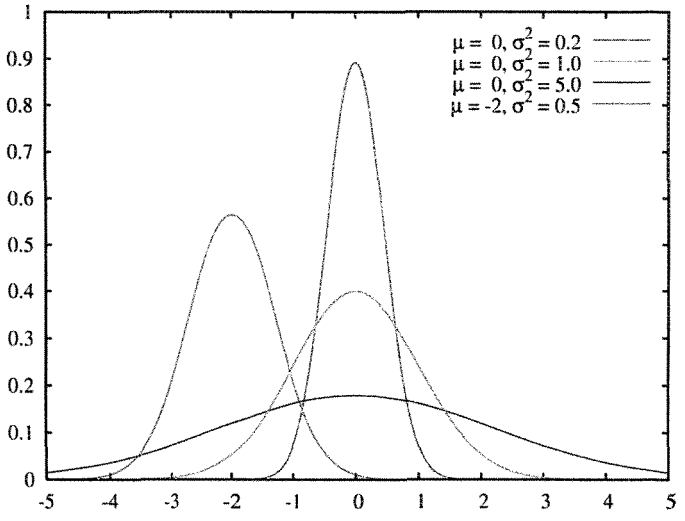

Figure 24 : lois normales de moyenne μ et de variance σ^2

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].

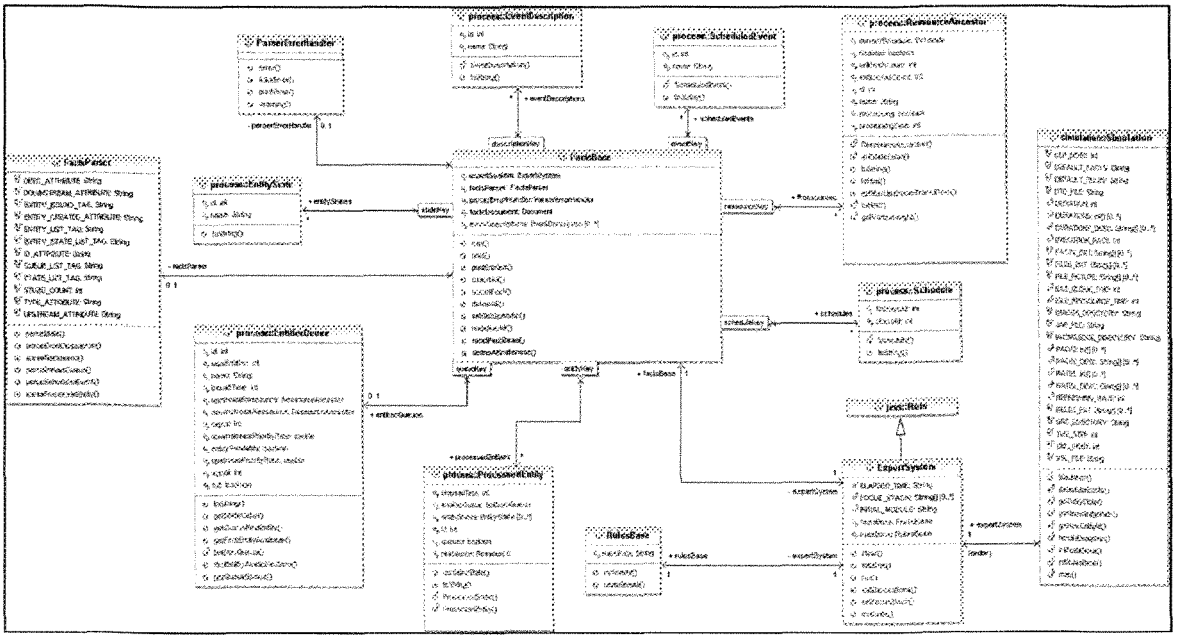

Figure 25 : classes du paquetage connaissance

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*).

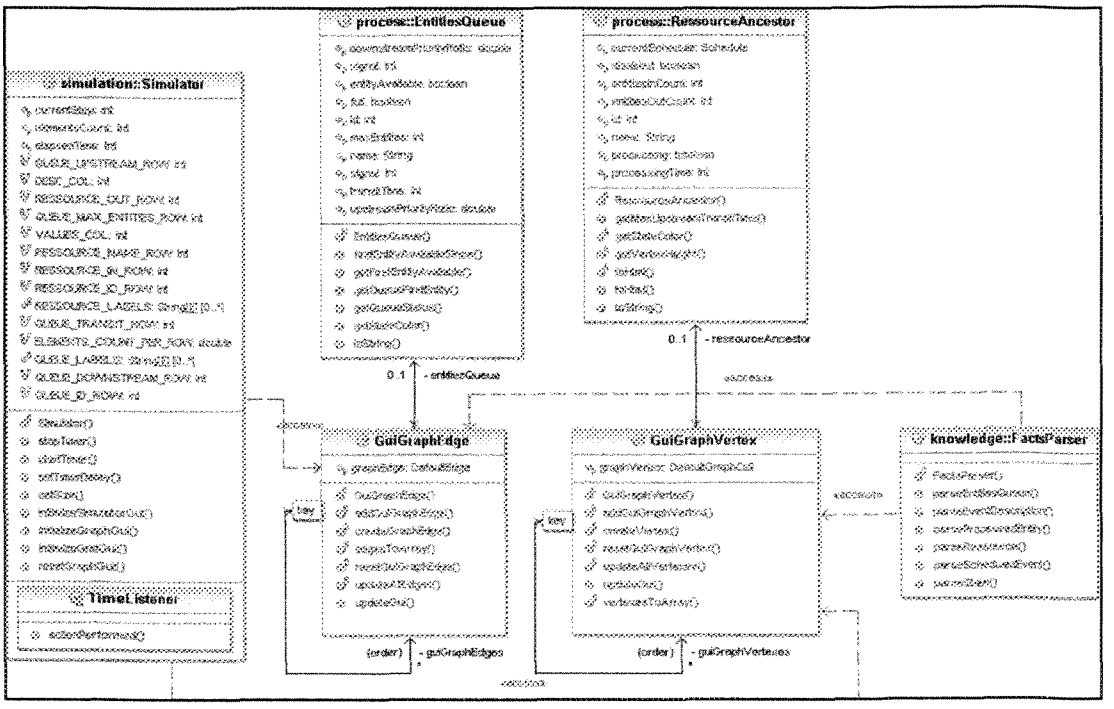

Figure 26 : classes du paquetage interface graphique

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*.

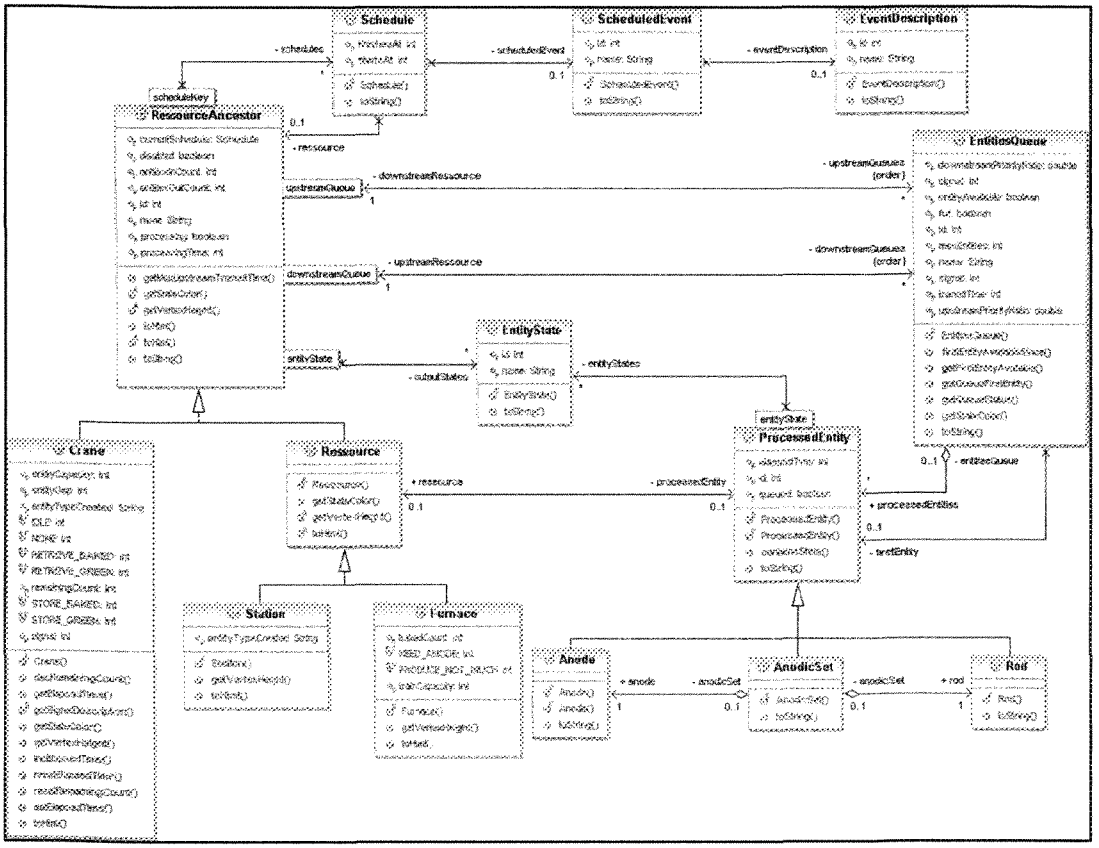

Figure 27 : classes du paquetage procédé

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*).

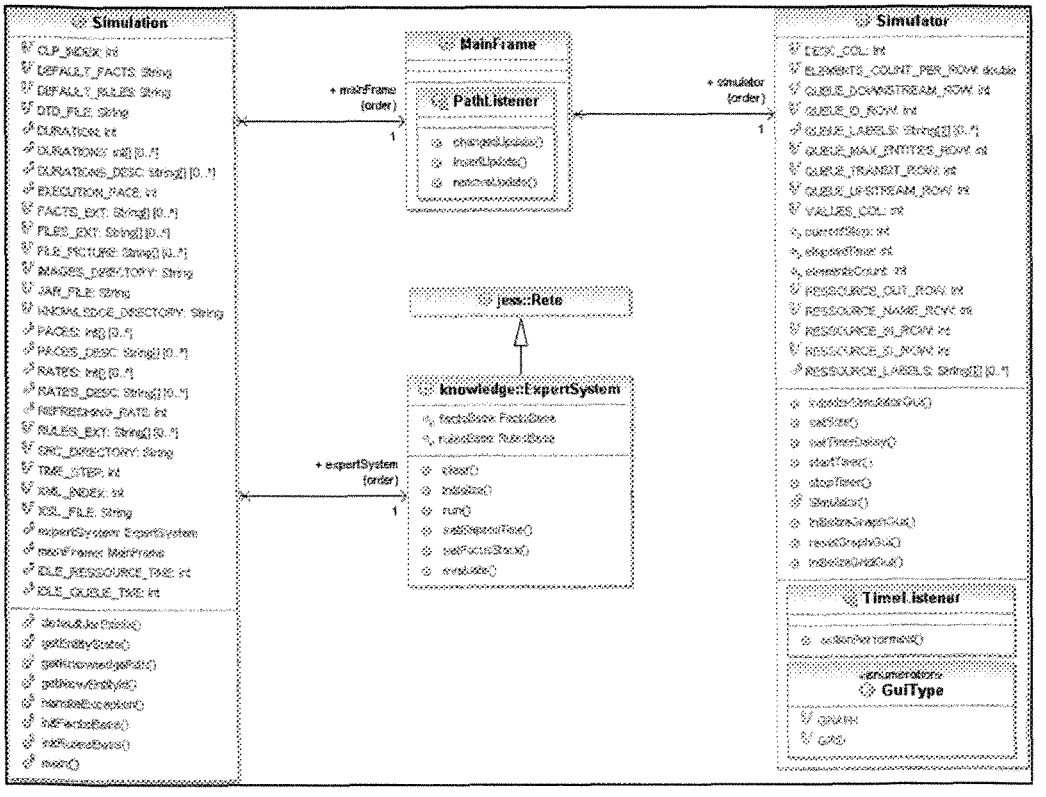

Figure 28 : classes du paquetage simulation

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.