Proposition d'une méthode de conception d'un superviseur de contrôle pour un processus de production distribué

Dés que le but d'un système de production est d'avoir une productivité optimale, l'existence d'un système de supervision devienne nécessaire. Un superviseur pour un système de production vise à assurer le bon déroulement, de l'ensemble des opérations exécutées par les processus de production. L'opération de supervision consiste à réaliser une synchronisation et une vérification de certaines propriétés relatives à l'aspect comportemental du système.

Tant que le rôle du superviseur est d'assurer la bonne synchronisation entre les opérations de production et de vérifier les propriétés relatives à la dynamique du système, nous avons recours à un modèle illustrant l'aspect statique de la partie opérative du système, tant qu'elle est le responsable de l'exécution réelle de l'ensemble des processus de production. Ainsi un autre modèle exprimant l'aspect comportemental, ce modèle doit être inspiré du celui de l'aspect statique. La conception de tel superviseur se base alors sur ces deux modèles.

Dans ce qui suit, nous présentons notre méthode de conception d'un superviseur de contrôle pour un processus de production distribué .Nous commençons par l'étude de transformation de modèles et nous étudions un outil adapté pour cela. Ensuite une démarche pour la modélisation et la vérification des processus de production en vue de leur supervision est présentée, dans laquelle nous expliquons comment la notion de transformation de graphes est utilisée via l'outil de transformation. Nous concluons par un rappel sur les principes de la méthode proposée.

2.1 Transformation de modèles

2.1.1 Pourquoi modéliser ?

Modéliser un système avant sa réalisation permet de mieux comprendre son fonctionnement. C'est également un bon moyen de maîtriser sa complexité et d'assurer sa cohérence.

Depuis longtemps, divers langages et techniques de modélisation sont utilisés lors des phases d'analyse et de conception, entre autres en développement logiciel. Se situant à un niveau

intermédiaire entre l'expression des besoins et le code développé pour répondre à ces besoins. Les modèles permettent une meilleure communication entre un informaticien et un client.

Du point de vue du développeur, un modèle permet de garder tout au long du cycle de vie du logiciel une vision globale sur le système à traiter, en offrant plusieurs niveaux d'abstraction.

2.1.2 Méta-modélisation

Tout comme un programme, un modèle destiné à être traité par une machine ne doit pas permettre d'ambiguïtés d'interprétation, et doit être exprimé en respectant des règles bien définies. Il s'agit là de l'objet de la démarche de méta-modélisation, qui s'attache à définir des formalismes pour les langages de modélisation. Une fois le formalisme défini, on peut garantir que tout modèle conforme à ce formalisme pourra être traité correctement [Meylan, 2006].

Un méta-modèle est un modèle d'un langage de modélisation. Le terme ''méta '' indique un niveau plus élevé, soulignant le fait qu'un méta-modèle décrit un langage de modélisation à un plus haut niveau d'abstraction que le langage de modélisation lui-même.

Un méta-modèle a deux caractéristiques principales. Premièrement, il doit capturer les caractéristiques essentielles du langage de modélisation, ainsi, il devrait être capable de décrire la syntaxe concrète, et la sémantique de ce langage [Clark, 2004].

Le tableau suivant définit quelques concepts qui différencient les niveaux d'abstraction de la méta-modélisation.

Méta-méta-modèle	Langage de spécification des méta-modèles
Méta-modèle	Définition du langage utilisé pour exprimer le modèle
Modèle	Abstraction du système
Système	Information et flux de contrôle d'un domaine

Tab 2.1 Niveaux d'abstraction de la méta-modélisation [Meylan, 2006].

- Au premier niveau, se trouve le système à étudier.
- Au second niveau, se trouve le modèle qui décrit certains aspects du premier niveau que nous voulons l'étudier: il peut s'agir d'un diagramme de classes UML, d'un modèle conceptuel de traitement MERISE, qui représente une vue abstraite des objets modélisés.
- Au troisième niveau se trouve le langage de modélisation ou méta-modèle : par exemple les définitions d'un MCD de MERISE d'un diagramme d'objet UML. C'est ce langage qui doit faire l'objet d'une spécification formelle.

§ Au quatrième niveau se trouve le méta-méta-modèle : il s'agit d'un langage qui doit être assez générique pour définir les différents langages de modélisation existants, et assez précis pour exprimer les règles que chaque langage doit respecter pour pouvoir être traité automatiquement [Meylan, 2006]. Des exemples de méta-méta-modèles : DSL Tools de Microsoft, MOF (Meta Object Facility), KM3 (Kernel MetaMetaModel).

2.1.3 Transformation de modèles

2.1.3.1 Définitions

La notion de transformation de modèles est centrale pour L'ingénierie dirigée par les modèles ou (**MDI** : Model Driven Engineering). Une transformation de modèles prend comme entrée un modèle conformément à un méta-modèle donné et produit comme sortie un autre modèle conformément à un méta-modèle donné.

La transformation de modèles est définie comme étant le processus de convertir un modèle d'un système à un autre modèle [Czarnecki, 2006].

Deux types de transformations sont réalisables :

§ Les transformations endogènes (si le méta-modèle source et cible sont identiques).

§ Les transformations exogènes (si le méta-modèle source et cible sont différents).

Une autre classification indique qu'il y a les transformations de type modèle vers code et celles de type modèle vers modèle.

Une transformation de modèles peut également avoir plusieurs modèles source et plusieurs modèles cibles. Une caractéristique de transformation de modèles est qu'elle est un modèle puisque elle doit être conforme à un méta-modèle donné [Czarnecki, 2006].

Une présentation générale des principaux concepts impliqués dans la transformation de modèles est illustrée dans la Figure suivante :

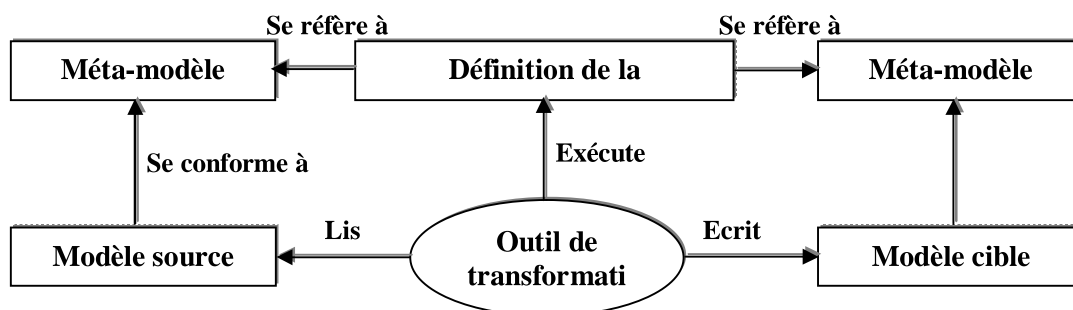

Figure 2.1 Concepts de base de la transformation de modèles [Czarnecki, 2006].

2.1.3.2 Transformations de type modèle vers modèle

La transformation de type modèle vers modèle est une transformation basée sur les modèles dans les deux cotés, parmi les types de cette transformation est les transformations de graphes qui nous intéressent dans notre méthode de conception.

2.1.3.3 Structure d'une transformation de type modèle vers modèle

Une transformation de modèles est principalement caractérisée par la combinaison des éléments suivants :

§ Des règles de transformation, une règle de transformation est composée de deux parties: une partie gauche (**LHS**, Left Hand Side) qui accède au modèle source, et une partie droite (**RHS**, Right Hand Side) qui accède au modèle cible. Ces deux parties sont composées de variables, de modèles (patterns) et d'expressions logiques.

§ Une relation entre le modèle source et le modèle cible. Pour certains types de transformations, la création d'un nouveau modèle cible est nécessaire, pour d'autres, la source et la cible sont le même modèle, ce qui revient en fait à une modification de modèle.

§ Une spécification, certaines approches de transformation fournissent un mécanisme dédié de spécifications, tel que des pré-conditions et des post-conditions exprimées dans un langage. Des spécifications particulières de transformation peuvent représenter une fonction entre les modèles source et cible et peuvent être exécutables. Mais généralement, les spécifications décrivent des relations et elles ne sont pas exécutables.

§ La planification d'application des règles, détermine l'ordre dans lequel sont appliqués les règles, sa forme est implicite, où l'ordre est défini par l'outil de transformation ou explicite où l'ordre d'exécution des règles est contrôlé à l'extérieur de l'outil de transformation.

§ L'organisation des règles, définit comment composer plusieurs règles de transformation. Elle est soit modulaire, c'est la possibilité pour un module d'importer un autre, soit un mécanisme de réutilisation par l'héritage entre règles , soit les règles peuvent être organisées selon une structure dépendante du modèle source ou du modèle cible.

§ La traçabilité, permet de garder la trace des liens entre éléments source et cible, d'analyser l'impact d'un changement d'un modèle sur un autre et de synchroniser entre les modèles

§ La direction, il y a deux types, des règles unidirectionnelles, qui s'exécutent dans une seule direction et des règles bidirectionnelles, qui peuvent être exécutées dans les deux sens [Helsen, 2006] [Czarnecki, 2006].

2.1.3.4 Transformation de graphes

Comme les grammaires de Chomsky pour des textes, la transformation de graphes se fait grâce à des règles de transformation [Guerra, 2006]. Ces règles sont appliquées sur un graphe, et si les règles trouvent plusieurs concordances, elles effectuent les changements déterminés en opérant des remplacements, des ajouts ou des suppressions. Donc le processus de transformation s'effectue, en partant d'un graphe de départ, produire un graphe d'arrivée qui pourra être modifié ensuite [Scolas, 2007].

Une règle est constituée de deux parties, le Left Hand Side (**LHS**) et le Right Hand Side (**RHS**). Le LHS est une partie du graphe, destinée à être mise en concordance avec les parties du graphe (appelé host graph) où nous voulons appliquer la règle. Le RHS quant à lui décrit la modification qui sera effectuée, elle substitue la partie identifiée dans le host graph [Scolas, 2007].

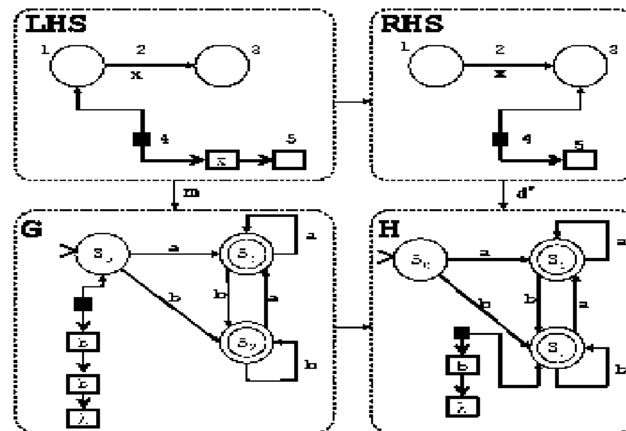

Figure 2.2 Exemple d'application d'une règle sur un graphe [Guerra, 2006].

La Figure 2.2 montre un exemple de dérivation, où une règle indiquant une étape dans la simulation d'un automate d'état finis est appliquée à un host graph ***G***. Le LHS montre deux états de l'automate (marqué en tant que 1 et 3) reliés par une transition (marquée 2). Le noeud 4 a un arc vers l'état actuel et un lien à une séquence des entrées. La valeur du premier élément de la séquence des entrées doit être égale à la valeur de la transition (variable **x**). La règle décrit la consommation de la première entrée et le changement de l'état actuel. Dans la dérivation, la variable **x** dans la règle est instanciée par la valeur **b** et l'état actuel est déplacé de $\mathbf{s_0}$ à $\mathbf{s_2}$ [Guerra, 2006]. Les résultats d'application de règle sont illustrés dans le graphe **H** [Guerra, 2006].

L'ordre dans lequel les règles d'une grammaire de graphes sont appliquées est arbitraire. Cependant, il est possible de spécifier un ordre en équipant les règles d'une priorité ou d'une couche. Dans le cas des priorités, des règles avec des priorités plus élevées sont appliquées d'abord .Dans le cas des couches, Les règles dans la première couche sont appliquées tant que possible. Puis, les couches suivantes sont consécutivement exécutées dans l'ordre croissant.

Parmi les outils permettant les transformations de graphes : **VIATRA**, **ATOM3**, **GreAT**, **UMLX**, **BOTL et AGG.** Nous nous intéressons dans notre méthode par l'outil ATOM3.

2.2 ATOM^3

2.2.1 Présentation

AToM^3 (A Tool for Multi-formalism and Meta-Modelling) est un outil de modélisation multi-paradigmes écrit en Pyton ,développé par le laboratoire MSDL (Modelling, Simulation and Design Lab) à l'université de McGill Montréal, Canada. Cet outil a été conçu en collaboration avec Juan de Lara de l'université de Madrid (UAM), Espagne [ATOM3, 2007].

Les deux principales fonctionnalités d'ATOM^3 sont la méta-modélisation et la transformation de modèles.

Dans AToM^3 les formalismes et les modèles sont décrits comme des graphes. Cet environnement génère un outil de manipulation graphique des modèles décrits dans un formalisme donné, à partir d'une méta-spécification de ce formalisme (généralement décrite dans le formalisme Entité/Relation). Les transformations entre modèles sont ensuite effectuées par réécriture des graphes (utilisant des grammaires de graphes) [De Lara, 2002].

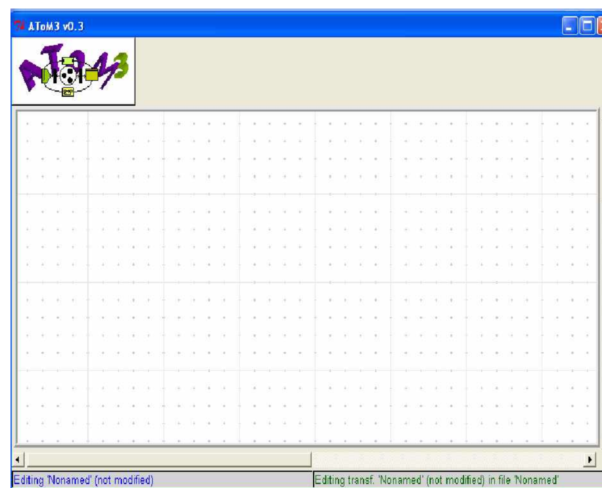

Figure 2.3 Interface d'ATOM^3.

2.2.2 Architecture d'ATOM^3

Le composant principal d'ATOM3 est le noyau (Kernel), qui est responsable du chargement, de la sauvegarde, de la création et de la manipulation des modèles (à tout méta-niveau). Par défaut, les méta-modèles et les méta-méta-modèles sont chargés quand ATOM^3 est lancé. Ces méta-méta-modèles permettent de modéliser les méta-modèles (modélisation des formalismes). Le formalisme Entité-Relation avec des contraintes est utilisé dans le méta-méta-niveau .Les contraintes sont du code en Pyton et le concepteur doit spécifier si ces contraintes sont (pré, post et sur quel événement une contrainte est évaluée).

Dans la modélisation à ce niveau, les entités qui doivent apparaître sur les modèles sont spécifiées ensemble avec leurs attributs, leurs apparences graphiques, leurs contraintes et leurs actions. Chaque relation entre deux entités peut être caractérisée par une contrainte en terme de cardinalités [De Lara, 2002].

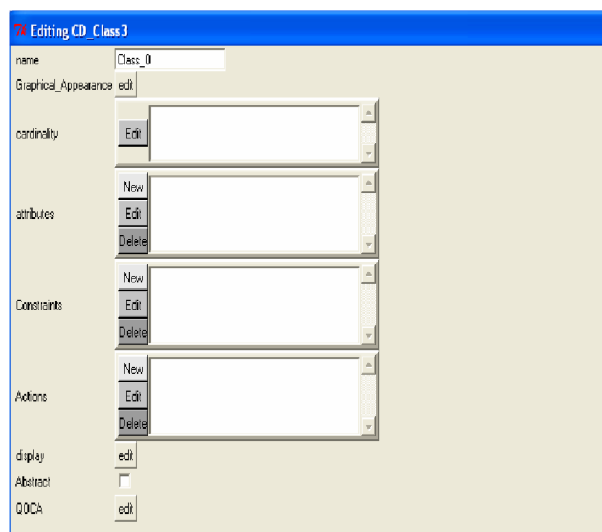

Figure 2.4 Editions des caractéristiques d'une entité.

2.2.2.1 Attributs

En générale, dans ATOM^3, il y a deux types d'attributs :

§ Les attributs réguliers, qui sont utilisés pour identifier les caractéristiques d'une entité.

§ Les attributs générateurs, qui permettent de générer d'autres propriétés [De Lara, 2002].

Chaque attribut a un type et un nom et une valeur initiale, comme il est montré dans la Figure 2.5.

Dans ATOM^3, chaque type a une classe Pyton, qui est responsable à l'édition des valeurs et les tests de leur validité. Il y a deux types de base :

§ Type régulier tel que String, Integer, Float, Boolean…etc.

§ Type générateur qui permet de générer des attributs, des contraintes, des attributs graphiques [De Lara, 2002].

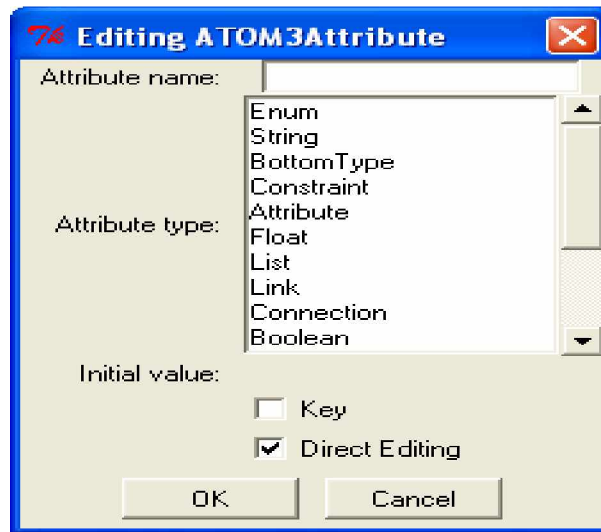

Figure 2.5 Edition des valeurs d'un attribut.

2.2.2.2 Contraintes et actions

Dans ATOM[3] Une contrainte est un code en Pyton ou des expressions en OCL, il y a deux types de contraintes :

§ Des contraintes locales associées à une entité, elle doit contenir que des attributs pour cette entité.

§ Des contraintes globales, ou les informations de toutes les entités du modèle sont utilisées.

La Figure 2.6 montre la structure d'une contrainte. Une contrainte est composée de :

§ Un nom de contrainte.

§ Un événement déclenchant l'évaluation de la contrainte, l'événement peut être sémantique tel que la sauvegarde d'un modèle, création d'une entité ou graphique ou structurel, tel que le déplacement ou la sélection d'une entité.

§ Une spécification quand la contrainte doit être évaluée, avant l'événement (pré-condition) ou après (post-condition).

§ Une zone pour écrire un code Pyton ou en OCL [De Lara, 2002].

Si la pré-condition d'un événement échoue alors ce dernier ne va pas être exécuté, si la post-condition d'un événement échoue ce dernier ne va pas être accompli.

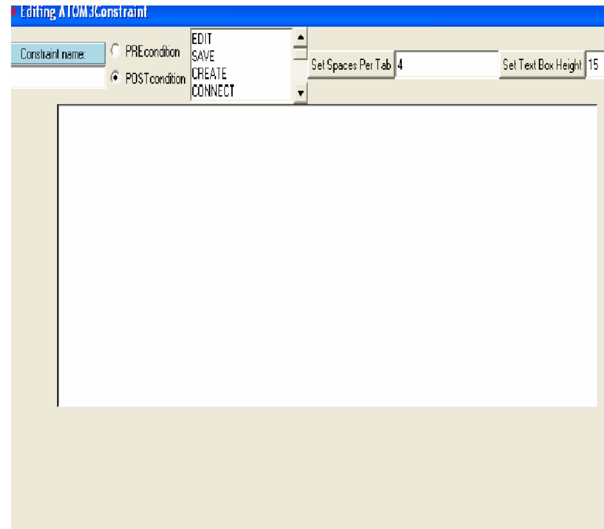

Figure 2.6 Structure d'une contrainte.

Une action est similaire à une contrainte sauf qu'elle a d'autres effets et elle est un code en Pyton seulement.

Une action est composée de :

§ Un nom d'action.

§ Un événement déclenchant l'action, l'événement peut être sémantique tel que la sauvegarde d'un modèle, création d'une entité ou graphique ou structurel tel que le déplacement ou la sélection d'une entité.

§ Une spécification quand le code de l'action doit être exécuté, avant l'événement (pré-condition) ou après (post-condition).

§ Une zone pour écrire un code Pyton [De Lara, 2002].

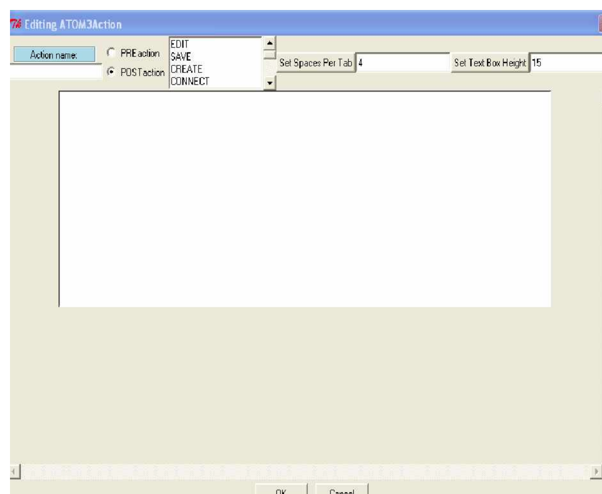

Figure 2.7 Structure d'une action.

2.2.2.3 Transformation de graphes

Dans ATOM[3], une fois un modèle est chargé il peut être transformé à un autre modèle équivalent, exprimé par un autre formalisme ou dans le même formalisme.

Dés que les modèles sont représentés à l'intérieur sous forme de graphes alors les transformations entre modèle s'effectue par des grammaires de graphes.

Une grammaire de graphes est un ensemble de règles, avec une action initiale et une action finale. Elle a un nom et un ensemble de menus permettant sa manipulation, tel que la sauvegarde, l'exécution…etc. comme le montre la Figure suivante :

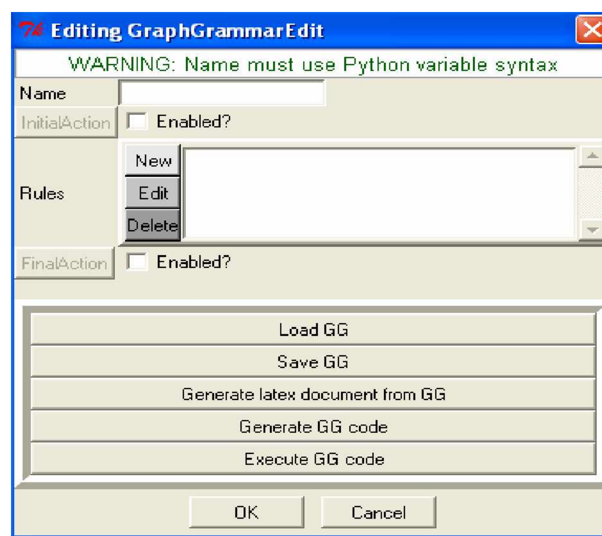

Figure 2.8 Structure d'une grammaire.

Chaque règle est constituée de:

§ Un nom spécifique pour la règle.

§ Une priorité indiquant l'ordre dans lequel la règle est appliquée.

§ Une partie gauche (Left Hand Side : LHS) qui est un graphe.

§ Une partie droite (Right Hand Side : RHS) qui peut être un graphe.

§ Une condition (Un code en Pyton) qui doit être vérifiée avant que la règle soit appliquée.

§ Une action (Un code en Pyton) qui doit être exécutée une fois que la règle est appliquée.

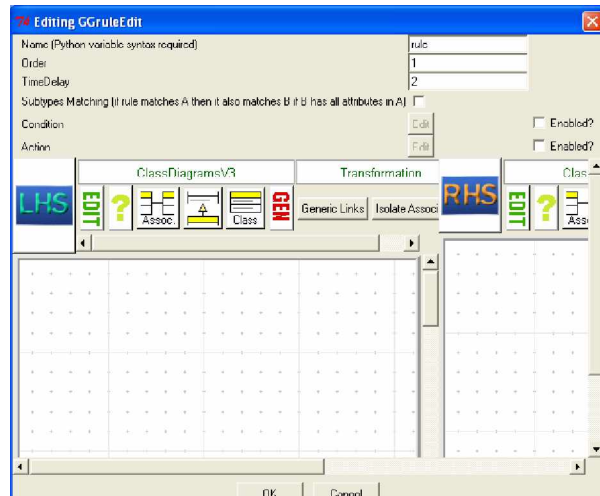

Figure 2.9 Structure d'une règle.

Les deux parties gauche et droite peuvent être des graphes illustrés par des formalismes différents, et pour faire la transformation il faut charger les méta-modèles qui concernent ces formalismes.

L'exécution de règles peut être continue (aucune interaction d'utilisateur) ou étape par étape ou l'utilisateur doit intervenir après chaque exécution d'une règle, ou d'une manière arbitraire [De Lara, 2002].

2.2.2.4 Langage Pyton

Python, est un langage autorisant la programmation impérative et la programmation orientée objet, inspiré du langage C, portable sur la plupart des systèmes d'exploitation (Unix, Mac Os, Windows, ..), placé sous la licence BSD (pour Berkeley Software Distribution) et disponible gratuitement, aussi bien utilisable pour écrire des programmes de quelques lignes que des projets très complexes [Pyton, 2008].

La syntaxe de Python est très simple et combinée à des types de données évolués (listes, dictionnaires,...), conduit à des programmes à la fois très compacts et très lisibles. A fonctionnalités égales, un programme Python est souvent de 3 à 5 fois plus court qu'un programme C ou C++ (ou même Java) équivalent, ce qui représente en général un temps de développement de 5 à 10 fois plus court et une facilité de maintenance largement accrue.

Python est orienté-objet. Il supporte l'héritage multiple et la surcharge des opérateurs. Dans son modèle objets, et en reprenant la terminologie de C++, toutes les méthodes sont virtuelles. Il intègre, un système d'exceptions, qui permettent de simplifier considérablement la gestion des erreurs.

Un programme écrit en Python est aussi bien interprète que compilé. Les programmes sources seront écrits dans des fichiers dont le nom aura l'extension ''**.py**'', et seront exécutés par la commande **Python nom_prograamme.py**, lancé dans un terminal [Pyton, 2008].

Notre utilisation de ce langage est limitée à certaines instructions, mais la plupart des autres instructions sont propres à ATOM[3], mais sur la base de ce langage.

2.3 Modélisation de processus de production distribué

Un processus de production distribué est un ensemble de relations entre des événements et des activités dans le système de production .Une description d'un processus de production est utilisée pour décrire comment le système de production fonctionne , dans le sens de contrôler son fonctionnement .

La distribution signifie que le processus de production est composé d'un ensemble de composants, chacun a son propre contrôle et capable d'effectuer certaines activités.

Comme il est mentionné dans le chapitre précèdent, pour concevoir un superviseur de contrôle pour un système de production, il faut disposer d'un modèle modélisant la partie physique, qui est capable d'exécuter l'ensemble des processus. Ces modèles doivent présenter la structure et le comportement du système et permettent l'analyse de ses propriétés. Donc la modélisation de cette partie est une étape initiative pour notre méthode de conception.

2.3.1 Choix des moyens de modélisation

Les systèmes de production sont considérés comme des systèmes a événements discrets. Les formalismes les plus connus pour la représentation des DES, sont les automates d'états finis, les réseaux de Petri, la logique temporelle…etc. En effet ces techniques souffrent des problèmes d'explosion d'états puisque les DES ont des structures complexes. Il est alors nécessaire de faire recours aux approches modulaires.

Une approche modulaire consiste à identifier les composants d'un système de production, représenter chacun séparément, dans le but de faciliter la modélisation des systèmes complexes et les rendre plus compréhensibles [Bordbar, 2000].

UML étant le langage standard de modélisation orientée objets .Il offre une gamme d'outils étendue, permettant de représenter sous forme graphique divers aspects d'un système à traiter. Il prend le concepteur durant le cycle de vie de conception, commençant par la description fournie par des utilisateurs ou des experts vers le logiciel final.

Notre méthode de conception est basée alors sur le langage UML, pour la modélisation de l'aspect statique du processus de production distribué.

Pour offrir une assistance réelle dans les niveaux de Modélisation, il est nécessaire qu'un modèle puisse être traité automatiquement. Ce traitement est effectué par l'outil ATOM[3].

Cet outil permet une phase de méta-modélisation .La méta-modélisation correspond à une phase d'abstraction de haut niveau. Elle permet de mettre en place les concepts fondamentaux qui seront utilisés au niveau utilisateur dans la phase de modélisation du processus.

2.3.2 Démarche de modélisation d'un processus de production distribué

Comprendre un processus de production revient à présenter les deux concepts caractérisant ce processus :

§ Son pôle-ontologique : ce que le processus est, décrivant sa structure.

§ Son pôle fonctionnel : ce que le processus fait, décrivant ses activités.

Nous considérons dans ce cas deux niveaux de modélisation :

Ø **Niveau structurel**

Dans ce niveau, on considère deux aspects distincts :

§ Un aspect décrivant d'une manière textuelle de quoi consiste le processus et de quoi il est constitué ; et comment se fait l'enchaînement des activités du processus. Nous utilisons pour cela le diagramme de cas d'utilisation d'UML.

§ Un aspect décrivant d'une manière graphique la structure du processus de production en modélisant les composants du processus et leurs relations. Nous utilisons pour cela le diagramme de classes d'UML.

Ø **Niveau Comportemental**

Dans le niveau comportemental, nous considérons l'enchaînement d'activités du processus de production, pour le comportement normal et avec l'évitement du comportement anormal. Ce niveau doit permettre la vérification des propriétés comportementales du processus de production. Pour cela, nous utilisons le formalisme réseau de Petri.

2.3.2.1 Modélisation structurelle

a. Diagramme de cas d'utilisation

Le diagramme des cas d'utilisation pour un processus de production, est une description des objectifs du processus et comment les activités constituant ce processus sont planifiées.

L'étude du diagramme des cas d'utilisation permet au concepteur de reconnaître les individus d'un système qui sont des objets dans la terminologie UML.

Le diagramme des cas d'utilisation se présente sous forme textuelle contenant :

§ Des membres réalisant des opérations, qui sont présentés par des noms, se sont des acteurs du diagramme de cas d'utilisation.

§ Des préconditions qui décrivent dans quel état doit être le processus de production avant qu'un événement puisse être déclenché.

§ Des scénarios qui sont décrits sous la forme d'échanges d'évènements entre les acteurs. On distingue le scénario normal, qui se déroule quand il n'y a pas d'erreur, et le scénarii d'exception qui décrit les cas d'erreurs.

§ Des postconditions qui décrivent l'état du système à l'issue des différents scénarios.

Avec ces points, le diagramme de cas d'utilisation va permettre au concepteur de déterminer la stratégie de synchronisation qui doit la suivre dans la conception du superviseur de contrôle.

Le scénario de fonctionnement du processus de production présenté dans le diagramme de cas d'utilisation correspond à la spécification H de la théorie de supervision de Ramadge &Wonham définie dans le chapitre précédent.

b. Diagramme de classes

Le but d'un diagramme de classes est d'identifier les classes dans un modèle. Comme nous avons mentionné dans le chapitre précédent, les classes dans UML sont représentées graphiquement par des rectangles contenant trois parties. La première est le nom de la classe, la deuxième est une liste d'attributs et la dernière est un ensemble de méthodes. Les classes sont connectées par des liens de type association ou de généralisation.

Pour notre méthode de conception du superviseur de contrôle, le diagramme de classes pour un processus de production distribué permet de modéliser la structure statique de ce processus, il est composé de classes illustrant les acteurs du diagramme de cas d'utilisation. Pour modéliser la synchronisation entre les différents acteurs du processus, nous avons proposé que :

Ø Chaque classe du diagramme doit avoir quatre variables booléennes indiquant, l'état de l'acteur dans le processus de production à un instant précis. Ces attributs sont :

§ **dp:** (décision point) un point de décision sur la prochaine activité à réaliser.

§ **wait** :(Attendre) arrêter l'activité en cours en attendant qu'une autre activité se termine.

- § **out:** (sortir) l'activité en cours est terminée, en sortant de la zone de travail d'un processus de production.
- § **work:** Une activité d'un processus de production est en cours d'exécution.

ØChaque classe doit avoir cinq méthodes qui permettent de changer l'état un acteur. Ces méthodes sont :

- § **new ():**Une nouvelle activité vient d'être commencé.
- § **ab ():** (abort) stopper une activité d'un processus de production en attendant l'arrivée d'un événement.
- § **start ():** reprendre une activité stoppée par la méthode **ab ().**
- § **go ():** pas d'attente, exécuter une activité après une décision de la commencer.
- § **exit ():** sortir de la zone de travail, une activité en cours est terminée normalement.

Une classe dans le diagramme de classe pour un processus de production est illustrée dans la Figure suivante :

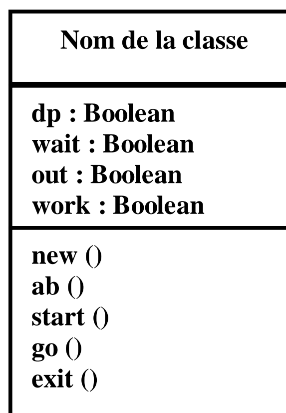

Figure 2.10 Structure d'une classe d'un processus de production distribué.

Dans notre travail, les codes de ces méthodes ne nous nous intéressent pas puisque dans la politique de supervision, ce qui est important c'est la synchronisation entre l'exécution de ces méthodes et pas comment chaque méthode est implémentée.

Les relations entre les classes sont des associations indiquant depuis le diagramme de cas d'utilisation s'il y a une synchronisation entre deux acteurs ou non.

Le diagramme de classes pour un processus de production distribué correspond au modèle Plant de la théorie de supervision de Ramadge &Wonham.

c. Modélisation automatique avec ATOM[3]

Pour automatiser la modélisation de la structure du processus de production et plus précisément, avoir un moyen de modélisation du diagramme de classes pour ce processus, nous avons utilisé l'outil ATOM[3].

La modélisation avec ATOM[3] correspond à la phase d'utilisation des fonctions génériques ou objets mis en place à une étape de méta-modélisation pour développer un modèle représentant un système .Donc la méta-modélisation nous permet de définir de manière abstraite les différents concepts communs aux éléments d'un modèle.

La modélisation automatique du diagramme de classes avec ATOM[3], doit passer par les étapes suivantes:

1. Chargement d'un méta-modèle pour les diagrammes de classes, qui se trouve dans les formalismes principaux d'ATOM[3].
2. Concevoir le méta-modèle du processus de production selon la structure définie dans la Figure 2.10. Ce méta- modèle est composé d'une méta-classe présentant un composant du processus de production qui possède un nom ,les attributs et les méthodes mentionnées auparavant et une association avec la même méta-classe, avec une cardinalités égale à 0-N, comme le montre la Figure suivante :

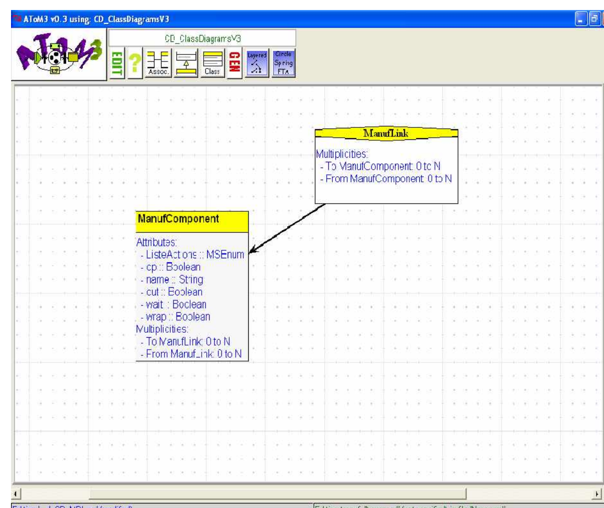

Figure 2.11 Méta-modèle pour le diagramme de classes.

3. A partir de ce méta-modèle, ATOM[3] génère un outil de modélisation de diagramme de classes pour un processus de production distribué. Cet outil offre un ensemble de boutons de manipulation de ce diagramme. Cet outil est illustré dans la Figure suivante :

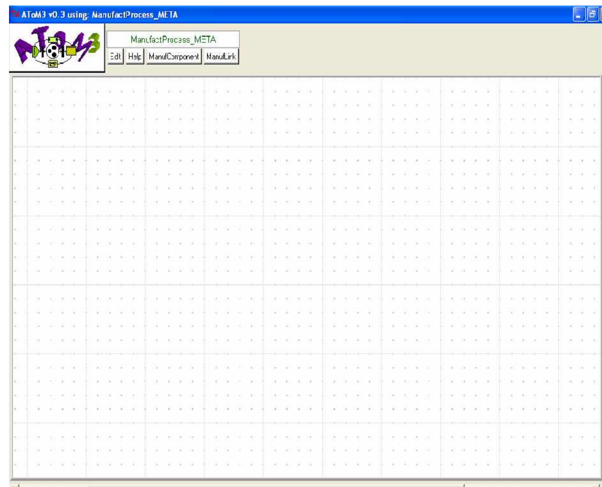

Figure 2.12 Outil de modélisation généré par ATOM^3.

2.4 Vérification du processus de production distribué

La vérification du processus de production est une étape très importante dans la conception du superviseur de contrôle, car elle s'intéresse à vérifier les propriétés comportementales qui concernent l'aspect dynamique du processus, ainsi que la définition de la bonne démarche de fonctionnement que doit respecter le processus pour éviter certains problèmes.

Pour cela, cette étape nécessite un modèle illustrant la dynamique du processus de production d'une part, et d'autre part il doit permettre une vérification efficace des propriétés comportementales de ce processus.

2.4.1 Choix des moyens de vérification

Tant que les systèmes de productions sont des systèmes a événements discrets .La plupart du temps, leur comportement est déterminé par des événements discrets qui apparaissent dans ces systèmes. Ce qui engendre un comportement complexe, nécessitant un formalisme très efficace pour modéliser le fonctionnement du système.

Pour ce type de systèmes est utilisé plusieurs formalismes tel que les statecharts d'UML, les automates d'états finis, le modèle de checking…etc. Mais ces formalismes n'ont pas satisfait la vérification de toutes les propriétés tel que, le non déterminisme, le parallélisme, le blocage…etc [Uzam, 1998] et n'ont pas la capacité suffisante pour modéliser la dynamique de ces systèmes.

Les réseaux de Petri comme un outil graphique et formel, sont de plus en plus employés dans la modélisation, l'analyse, la conception et la commande des systèmes à événement discrets [Ramadge, 1987]. Leur théorie permet d'analyser les propriétés de base des DEDS telle que la synchronisation, la concurrence, les conflits, le partage de ressources, les blocages [Bordbar,

2000]. Ainsi la simplicité de leur aspect graphique leur permet de simplifier la modélisation des systèmes complexes. Ainsi nous pouvons également évaluer l'exécution du système modéliser par un réseau de Petri. Un autre point c'est que l'outil ATOM[3] supporte ce formalisme, et permet sa méta-modélisation.

Notre méthode de conception se base donc sur le formalisme réseau de Petri pour la modélisation de l'aspect dynamique du processus de production et la vérification de ses propriétés comportementale et la génération de la bonne stratégie de fonctionnement de ce processus.

Pour, bénéficier de l'automatisation de cette étape, nous avons introduit un autre outil de vérification basé sur les réseaux de Petri qui est l'outil INA.

2.4.2 INA

INA (Integrated Net Analyzer) est un programme qui permet d'analyser des modèles réseau de Petri, il est développé par le Professeur Peter H. Starke. Avec son menu interactif, on peut éditer, réduire, exécuter et analyser un modèle réseau de Petri [Roch, 1999], comme le montre la Figure suivante :

Figure 2.13 Interface de l'outil INA.

Pour l'analyse d'un modèle, cet outil doit accepter en entrée un fichier texte qui porte l'extension ''**.pnt**''.Ce fichier est une description de la structure du modèle à analyser, il doit suivre la syntaxe suivante :

§ Chaque ligne est utilisée pour décrire une place du réseau a analyser.

§ Chaque ligne commence par un numéro de place suivi par un espace ensuite le nombre de marquage dans la place, si cette place a des transitions d'entrée et de sortie alors ils sont présentées comme des listes séparées par des virgules.

Pi nb_jeton_Pi [T_Pré],[T_Post]
Pj nb_jeton_Pj [T_Pré],[T_Post]
.
.

Figure 2.14 Structure du fichier d'entrée pour l'outil INA.

L'analyse s'effectue en tapant le caractère ''**A**'', ensuite le nom du fichier texte à analyser, l'outil va présenter des choix sur le type d'analyse que l'utilisateur veut faire (Figure 2.15) et ensuite, il affiche les résultats demandés concernant les propriétés du modèle analysé.

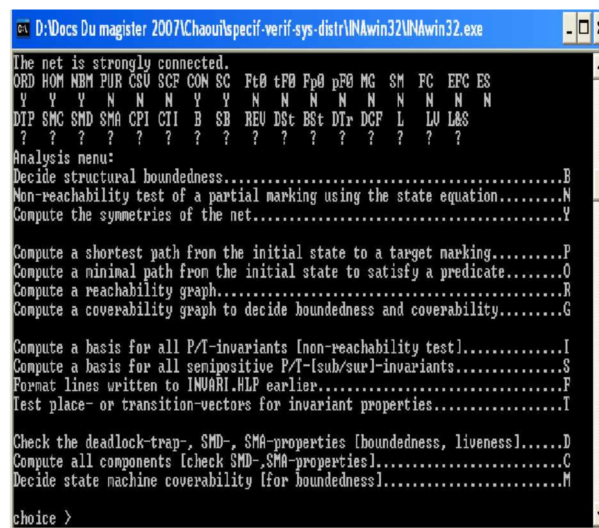

Figure 2.15 Différents choix d'analyse d'un modèle.

2.4.3 Démarche de vérification d'un processus de production distribué

La vérification du processus de production, repose sur un modèle réseau de Petri, modélisant la dynamique de ce processus. Ce modèle doit être inspiré du modèle illustrant la structure du processus, c'est-à-dire le diagramme de classes. Pour cela il faut trouver une équivalence entre le diagramme de classes et les réseaux de Petri.

Dans la modélisation avec les réseaux de Petri, chaque place représente l'état dans le quel se trouve l'élément modélisé et chaque transition représente la réalisation d'une opération par cet élément.

Dans notre travail, et pour faire la correspondance entre le diagramme de classes et les réseaux de Petri, nous avons proposé que :

§ Chaque classe du diagramme de classes est représentée par un réseau de Petri formé de cinq transitions et quatre places, tel que, chaque attribut de la classe est une place et chaque méthode est une transition, comme le montre la Figure suivante :

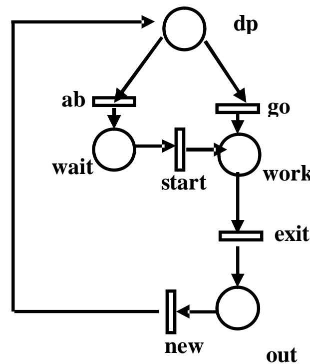

Figure 2.16 Un RdP correspondant à une classe du diagramme de classes.

Après un état de prise de décision **dp**, un composant d'un processus de production peut effectué un **ab** s'il n y a pas les conditions suffisantes pour exécuter une activité, ou **go** dans le cas contraire. Après un **ab()**, ce composant passe à l'état **wait** et s'il y a l'apparition d'un événement attendu dans le système le composant reprend son travail avec un **start()** et passe à l'état **work**. Après un **go()**, le processus passe à l'état **work**. Une fois une activité est exécutée, le composant fait un **exit ()** et passe a l'état **out**. Le cycle de production se répète en effectuant **new().**

Les associations entre les classes sont représentées par un ensemble de places de synchronisation suivant la stratégie de supervision utilisée.

La vérification du processus de production est réalisée suivant un ensemble d'étapes :

§ Génération d'un outil de modélisation avec les réseaux de Petri.

§ Définition d'une grammaire de graphes permettant de faire une transformation du diagramme de classes vers les réseaux de Petri.

§ Vérification automatique des propriétés comportementales du modèle réseau de Petri généré par la grammaire de graphes.

§ Génération automatique d'un graphe des états désirables (GDS), présentant le scénario de bon fonctionnement du processus de production.

2.4.3.1 Génération d'un outil de modélisation avec les réseaux de Petri

Pour générer un outil de modélisation avec les réseaux de Petri, nous avons utilisé ATOM³. Et comme nous avons mentionné que la modélisation avec cet outil nécessite une étape de méta-modélisation, la génération de cet outil de modélisation, doit passer par les étapes suivantes:

1. Chargement d'un méta-modèle pour les diagrammes de classes, qui se trouve dans les formalismes principaux d'ATOM³.
2. Concevoir le méta-modèle du réseau de Petri selon la structure définie dans la Figure 2.17. Ce méta- modèle est composé de quatre méta-classes :
   - § Une méta-classe pour les places du réseau de Petri, chaque place a deux attributs : **name** de type string et **Tokens** de type Integer.
   - § Une méta-classe pour les transitions, chaque transition a un attribut : **name** de type string.
   - § Deux associations, présentant les arcs, une pour les arcs entre les places et les transitions et l'autre pour les arcs entre les transitions et les places. Ces associations ont des cardinalités 0-N.

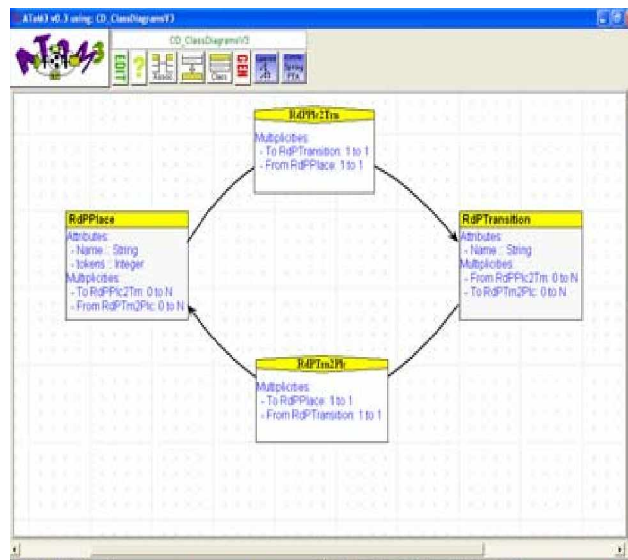

Figure 2.17 Méta-modèle pour les réseaux de Petri.

2. A partir de ce méta-modèle, ATOM³ génère un outil de modélisation pour le formalisme réseau de Petri. Cet outil est illustré dans la Figure suivante :

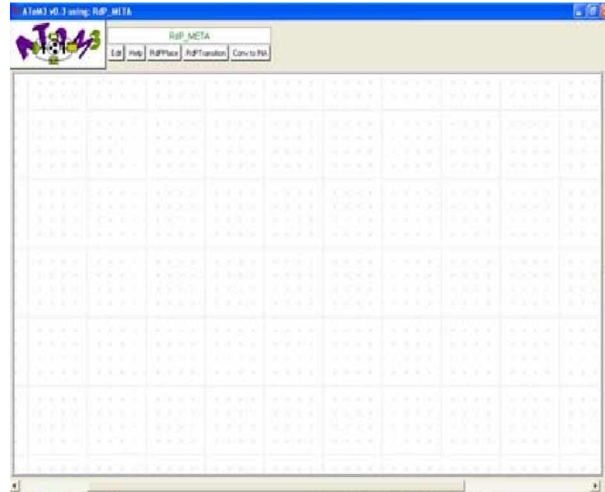

Figure 2.18 Outil de modélisation pour les RdPs généré par ATOM[3].

2.4.3.2 Définition d'une grammaire de graphes

Une grammaire de graphes est une grammaire constituée d'un ensemble de règles, permettant de transformer deux formalismes de même nature ou de nature différente. Chaque règle est composée de deux parties, la partie gauche (LHS) et la partie droite (RHS). Chaque partie peut être un sous graphe des formalismes considérés dans la transformation.

Dans notre travail, la partie gauche de chaque règle est un sous graphe du diagramme de classes pour le processus de production, et la partie droite peut être un sous graphe du réseau de Petri.

Cette grammaire est définie en utilisant l'outil ATOM[3]. Pour la définir et comme notre but est de transformer un diagramme de classes vers les réseaux de Petri, il faut :

- Charger les deux méta-modèles définis précédemment qui sont le diagramme de classes et les réseaux de Petri.
- Définir les règles de la grammaire, et générer son fichier exécutable.

a. Règles de la grammaire de graphes

Notre grammaire de transformation du diagramme de classes pour un processus de production vers les réseaux de Petri est composée de trois parties:

1. **Action initiale**

La partie action initiale de la grammaire est une partie d'initialisation, où nous avons initialisé l'ensemble des variables globales utilisées le code Pyton dans la grammaire.

2. **Action finale**

La partie action finale de la grammaire est une partie**,** où la libération de l'espace mémoire occupé par l'ensemble des variables globales a eu lieu.

3. Ensemble de règles

Notre grammaire est composée de dix règles. Chaque règle est caractérisée par un nom et une priorité d'exécution. Nous citons ici quelques règles :

∅ **Règle 1 :*ClasstoRdpRule (priority 1)***

Cette règle est appliquée pour transformer chaque classe du diagramme de classes vers un RdP formé de quatre places et cinq transitions et les associer des noms. La nomination est faite selon le nom de la classe correspondante de la partie gauche de la règle. Les noms sont sous la forme suivante : **nom de la classe_nom de place /transition.**

Cette règle a la priorité égale à un, c à d c'est la première règle appliquée dans la grammaire.

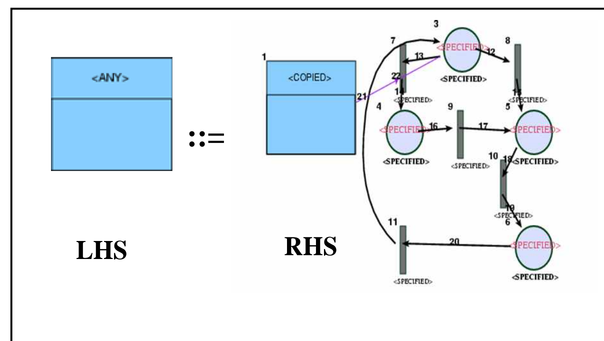

Figure 2.19 Partie gauche et droite de la règle une.

Pour l'application de cette règle, il faut vérifier que la classe de la partie gauche à transformer n'est pas traitée auparavant. Pour cela nous avons définis une condition pour cette règle :

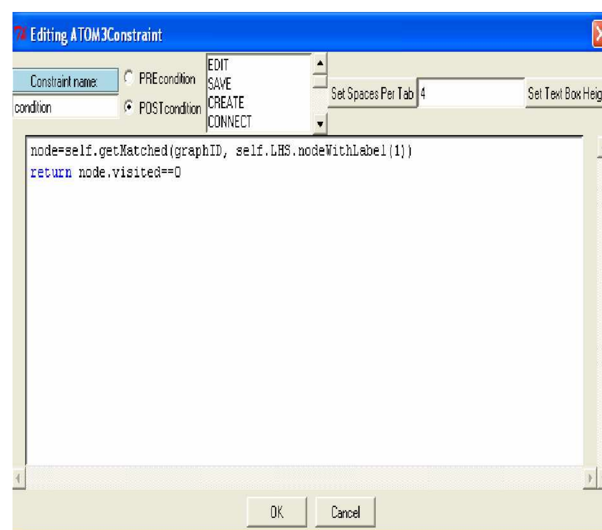

Figure 2.20 Condition d'application de la règle une.

Après l'application de la règle une action est exécutée permettant de marquer que l'objet classe de la partie gauche est traité.

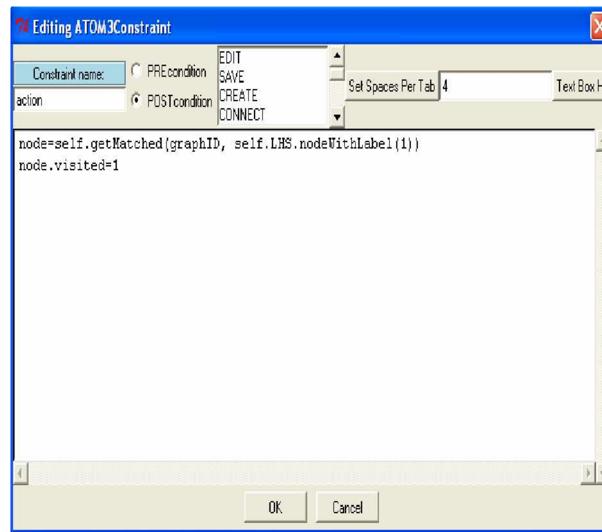

Figure 2.21 Action de la règle une.

∅ **Règle 2:** ***Identify Association Rule (priority 2)***

Cette règle est appliquée pour identifier s'il y a une association entre deux classes, et supprime cette association du diagramme de classes. Ainsi, elle sauvegarde les noms des deux classes membres de l'association dans des variables globales. Elle a la priorité égale à deux.

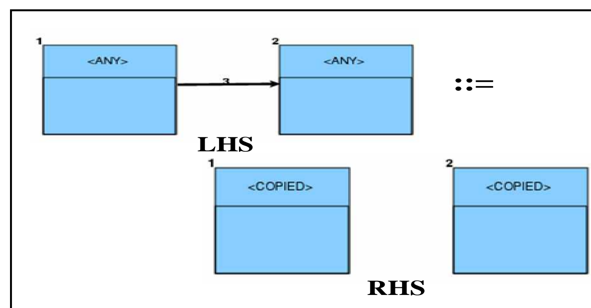

Figure 2.22 Parties gauche et droite de la deuxième règle.

La condition d'application de cette règle, est qu'il faut vérifier que les deux classes membres de l'association ne sont pas traitées auparavant. Cette condition est définie dans la contrainte de la règle, comme le montre la Figure 2.23.

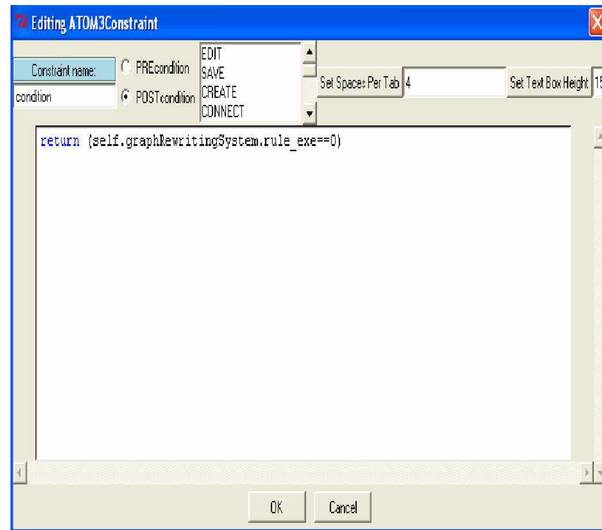

Figure 2.23 Condition d'application de la deuxième règle.

L'action correspondante à cette règle permet de :

§ Sauvegarder les noms des deux classes membres de l'association dans des variables globales.

§ Marquer que ces deux classes sont traitées.

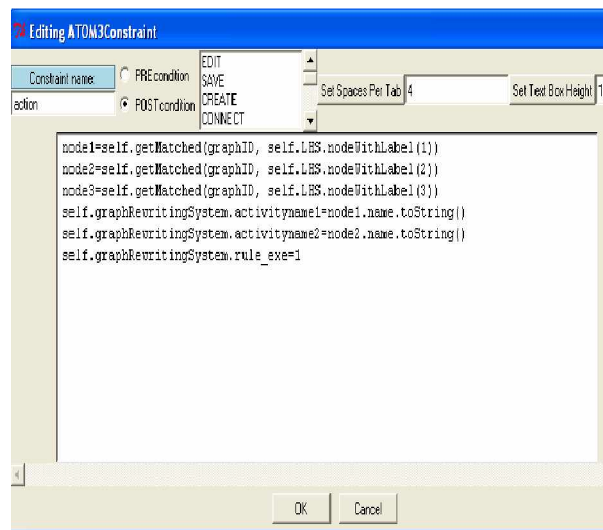

Figure 2.24 Action de la deuxième règle.

∅ **Règle 4:** ***AddPlacesSp1Rule (priority 4)***

Cette règle est une règle de synchronisation entre les composants d'un processus de production. Elle est appliquée pour chercher dans le RdP généré par les règles précédentes, des transitions par ses noms et d'ajouter une place nommée **SP1** comme une place d'entrée pour certaines transitions et une place de sortie pour d'autres. Elle a la priorité égale à quatre.

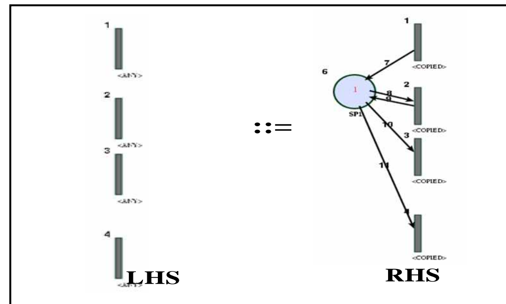

Figure 2.25 Parties gauche et droite de la quatrième règle.

Pour appliquer cette règle, il faut vérifier qu'elle n'est pas appliquée auparavant et que les noms des transitions cherchées correspondent bien à des noms spécifiques.

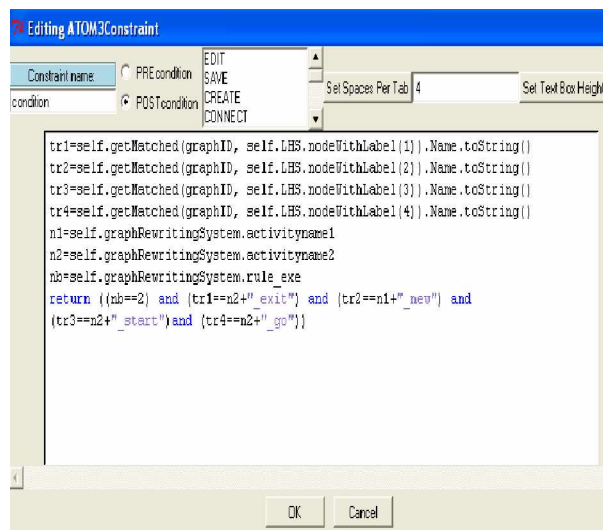

Figure 2.26 Condition d'application de la quatrième règle.

Une fois que la règle est appliquée, son action permet de marquer qu'elle est exécutée.

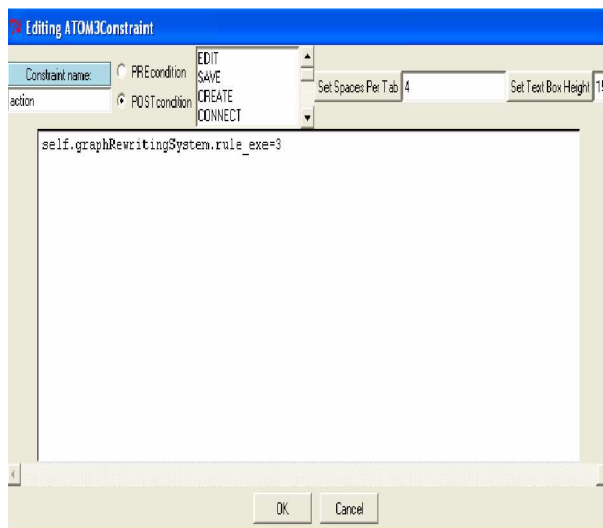

Figure 2.27 Action de la quatrième règle.

∅ **Règle 10:** ***ComponentDeleteRule (priority 10)***

Cette règle est appliquée pour chercher des classes non connectées c'est-à-dire sans associations, pour les supprimer. La partie droite de la règle est vide. Elle a la priorité égale à dix, c'est la dernière règle qui permet de supprimer toutes les classes restantes du diagramme de classes après la fin d'exécution des autres règles. Son action et condition sont vides.

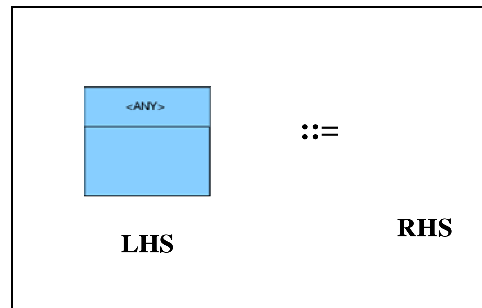

Figure 2.28 Parties gauche et droite de la dixième règle.

2.4.3.3 Vérification automatique des propriétés comportementales du processus de production

Cette étape consiste à vérifier certaines propriétés comportementales du modèle réseau de Petri généré par la grammaire de graphes. Ces propriétés sont relatives à l'aspect dynamique du processus de production et concernent son bon fonctionnement. Elles consistent en : la bornitude, la vivacité et l'absence de transitions bloquantes.

Pour obtenir une vérification automatique après la génération du modèle réseau de Petri, nous avons intégré l'outil INA dans notre travail de telle sorte que l'utilisateur aura la possibilité de faire cette vérification a partir de l'outil ATOM^3.

Cette possibilité est effectuée en appuyant sur un bouton **Convert To INA**, défini dans l'outil de modélisation avec les réseaux de Petri comme le montre la Figure 2.29. Un fichier texte d'extension ''**.pnt**''sera généré, il contient une description du modèle réseau de Petri généré. Ensuite ce fichier sera une entrée à l'outil INA, qui effectue l'analyse du modèle.

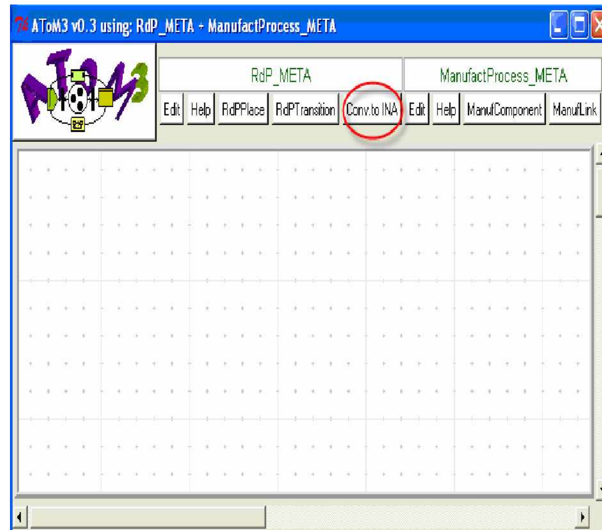

Figure 2.29 Analyse automatique du modèle RdP.

3.4.3.4 Génération automatique d'un graphe des états désirables

Le graphe des états désirables (GDS :Graph of Desirable States) est un graphe qui énumère tous les états désirables et leurs relations. Les états désirables sont des marquages et les relations entre ces marquages sont des transitions franchissables, depuis un marquage et amenant à un autre.

Le GDS utilise les informations du diagramme des cas d'utilisation dans le domaine de RdP, c'est-à-dire les phrases de ce diagramme sont traduites à un ensemble de règles en terme de places et de transitions. Le mot "désirable" reflète le fait que le graphe englobe tout ce que nous attendons du système et n'importe quel comportement indésirable est inhibé.

Le GDS est équivalent à un graphe de marquages, donc pour obtenir ce graphe de manière automatique, nous avons profité de l'outil INA qui calcule ce graphe a partir du fichier texte présentant le RdP généré. Le graphe résultat de cet outil est sous forme d'un fichier texte en terme de places et de transitions, comme le montre la Figure suivante :

Etat i
Liste des places : Pi ,Pj ,…………….PN
Les jetons : toki, tokj,…………tokN
Lest transitions franchissables Ti=>Sj
Tk=>SN
.
.
Etat j
Liste des places : Pi ,Pj ,…………….PN
Les jetons : toki, tokj,…………tokN
Lest transitions franchissables Ti=>Sj
Tk=>SN
.
.

Figure 2.30 Structure du fichier du graphe de marquages.

2.5 Conclusion

La méthode de conception d'un superviseur de contrôle pour un processus de production distribué, que nous avons proposé permet de disposer d'un outil de modélisation et de vérification de ce processus.

Cette méthode est basée sur deux modèles, un pour la structure du processus de production, c'est le diagramme de classes et l'autre pour modéliser la dynamique de ce processus, c'est le formalismes réseaux de Petri .En empruntant une technique de transformation de graphes offerte par l'outil ATOM^3 sous forme de grammaire de graphes, nous avons la possibilité d'effectuer une transformation du diagramme de classes vers son équivalent réseau de Petri , ainsi d'utiliser des outils existants de vérification des propriétés adaptés pour le modèle réseau de Petri.

La méthode de conception proposée est validée par des exemples de processus de production déjà traités d'une façon manuelle. Pour voir son application, ainsi pour bien comprendre les étapes présentées dans ce chapitre, nous allons étudier deux exemples de processus de production , chacun avec deux variantes dans le chapitre suivant.

Proposition d'une méthode de conception d'un superviseur de contrôle pour un processus de production distribué 51
2.1 Transformation de modèles 51
2.1.1 Pourquoi modéliser 51
2.1.2 Méta-modélisation 52
2.1.3 Transformation de modèles 53
2.1.3.1 Définitions 53
Figure 2.1 Concepts de base de la transformation de modèles [Czarnecki, 2006]. 53
2.1.3.2 Transformations de type modèle vers modèle 54
2.1.3.3 Structure d'une transformation de type modèle vers modèle 54
2.1.3.4 Transformation de graphes 55
Figure 2.2 Exemple d'application d'une règle sur un graphe [Guerra, 2006]. 55
2.2 ATOM^3 56
2.2.1 Présentation 56
Figure 2.3 Interface d'ATOM^3 56
2.2.2 Architecture d'ATOM^3 57
Figure 2.4 Editions des caractéristiques d'une entité. 57
2.2.2.1 Attributs 57
Figure 2.5 Edition des valeurs d'un attribut 58
2.2.2.2 Contraintes et actions 58
Figure 2.6 Structure d'une contrainte. 59
Figure 2.7 Structure d'une action. 59
2.2.2.3 Transformation de graphes 60
Figure 2.8 Structure d'une grammaire. 60
Figure 2.9 Structure d'une règle. 61
2.2.2.4 Langage Pyton 61
2.3 Modélisation de processus de production distribué 62
2.3.1 Choix des moyens de modélisation 62
2.3.2 Démarche de modélisation d'un processus de production distribué 63
2.3.2.1 Modélisation structurelle 63
Figure 2.11 Méta-modèle pour le diagramme de classes. 66
Figure 2.12 Outil de modélisation généré par ATOM^3. 67
2.4 Vérification du processus de production distribué 67
2.4.1 Choix des moyens de vérification 67
2.4.2 INA 68
Figure 2.13 Interface de l'outil INA. 68
Figure 2.14 Structure du fichier d'entrée de l'outil INA. 69
Figure 2.15 Différents choix d'analyse d'un modèle. 69
2.4.3 Démarche de vérification d'un processus de production distribué 69
Figure 2.16 Un RdP correspondant à une classe du diagramme de classes. 70
2.4.3.1 Génération d'un outil de modélisation avec les réseaux de Petri 71
Figure 2.17 Méta-modèle pour le réseaux de Petri. 71
Figure 2.18 Outil de modélisation pour les RdPs généré par ATOM^3. 72
2.4.3.2 Définition d'une grammaire de graphes 72
Figure 2.19 Partie gauche et droite de la règle une. 73
Figure 2.21 Action de la règle une. 74
Figure 2.22 Parties gauche et droite de la deuxième règle. 74
Figure 2.23 Condition d'application de la deuxième règle. 75
Figure 2.24 Action de la deuxième règle. 75
Figure 2.25 Parties gauche et droite de la quatrième règle. 76

Figure 2.26 Condition d'application de la quatrième règle. 76
Figure 2.28 Parties gauche et droite de la dixième règle. 77
2.4.3.3 Vérification automatique des propriétés comportementales du processus de production 77
Figure 2.29 Analyse automatique du modèle RdP. 78
2.5 Conclusion.. 79