REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE MOHAMED KHIDER BISKRA

FACULTE DES SCIENCES EXACTES, DES SCIENCES DE LA NATURE ET DE LA VIE

DEPARTEMENT INFORMATIQUE

N° d'Ordre :

N° Série :

Thèse de Doctorat en Sciences en Informatique

Thème

Une Approche de Spécification des Changements des Besoins Basée Transformation de Graphes

Présentée par : **Khaled Khalfaoui**

Date de soutenance: 24 Juin 2014

Composition du jury :

Djedi Noureddine	Professeur, Université de Biskra	Président
Chikhi Salim	Professeur, Université de Constantine 2	Examinateur
Babahenini M^{ed} Chaouki	Maître de Conférences A, Université de Biskra	Examinateur
Amirat Abdelkarim	Maître de Conférences A, Université de Souk Ahras	Examinateur
Foudil Cherif	Maître de Conférences A, Université de Biskra	Rapporteur
Chaoui Allaoua	Professeur, Université Constantine 2	Co-Rapporteur

Résumé

L'ingénierie des lignes de produits logiciels est une discipline récente en génie logiciel. C'est une adaptation du principe des chaînes de production au développement d'applications informatiques. Elle vise principalement à rationaliser le processus de développement des systèmes fortement similaires par une réutilisation logicielle stratégique et planifiée au préalable. L'apport majeur de ce paradigme est l'introduction d'une architecture de référence permettant la gestion de la variabilité. Elle fournit un cadre de développement des composants réutilisables et garantit leur incorporation appropriée. Pour chaque produit, elle est utilisée comme guide d'assemblage et de personnalisation des artéfacts nécessaires selon ses besoins spécifiques. Cette discipline a connu un grand succès en terme de productivité, mais des efforts conséquents liés à la gestion de la variabilité doivent être envisagés.

Dans le cadre de cette thèse, nous nous intéressons particulièrement à l'analyse automatique des modèles FD et FTS. Le diagramme FD est un formalisme largement utilisé pour la spécification structurelle des produits. Il s'agit d'un arbre spécifiant les caractéristiques et leurs dépendances. Le formalisme FTS est utilisé pour la spécification comportementale. C'est un système de transitions paramétré dans lequel ces transitions sont étiquetées avec les caractéristiques d'un diagramme FD en plus d'être marquées avec des actions. Il est instancié différemment pour chaque produit selon les caractéristiques requises.

Nous présentons trois contributions. Les deux premières permettent la génération de tous les produits structurellement valides à partir du diagramme FD. L'une est basée sur l'intégration du Backtracking dans le processus de recherche, alors que l'autre procède par une construction progressive de configurations partielles. La troisième contribution permet une analyse comportementale des produits modélisés par un FTS. La vérification des propriétés est réalisée en se basant sur la logique de réécriture et plus précisément le langage Maude.

Les techniques proposées sont mises en œuvre en s'appuyant sur l'approche transformation de graphes et les grammaires développées sont implémentées en utilisant l'environnement AToM^3.

Mots Clés: Ligne de Produits Logiciels, FD, FTS, Analyse et Vérification, Transformation de Graphes, AToM^3, BackTracking, Logique de Réécriture, Maude.

Abstract

Software product line is a recent discipline in software engineering. It is an adaptation of the principle of production chains in developing computer applications. It mainly intends to rationalize the process of developing highly similar systems by a strategic and planned software reuse beforehand. The major contribution of this paradigm is the introduction of a reference architecture allowing the management of the variability. It provides a framework for developing reusable components and ensures their appropriate incorporation. For each product, it is used as an assembly guide and personalization of necessary artifacts according to its specific needs. This discipline has a great success in terms of productivity, but significant efforts related to the management of variability must be considered.

In the context of this thesis, we are particularly interested in the automatic analysis of FD and FTS models. The FD diagram is a widely used formalism for the structural specification of the products. It is a tree specifying the features and their dependencies. FTS formalism is used for behavioral specification. It is a parameterized transition system wherein these transitions are labelled with the characteristics of an FD diagram besides being marked with actions. It is instantiated differently for each product according to the required characteristics.

We present three contributions. The first two ones allow the generation of all structurally valid products from the FD diagram. One is based on the integration of Backtracking in the research process, while the other proceeds by a progressive construction of partial configurations. The third contribution allows behavioural analysis of products modelled by FTS. The properties verification is performed on the basis on the rewriting logic and more exactly the Maude language.

The proposed techniques are set up based on the graph transformations approach and the developed grammars are implemented using AToM^3 environment.

Keywords: Software product lines, FD, FTS, Verification and Analysis, Graph Transformations, AToM^3, BackTracking, Rewriting Logic, Maude.

ملخص

هندسة سلسلة إنتاج البرمجيات تخصص حديث في مجال هندسة البرمجيات. وهو تكييف لمبدأ سلاسل الإنتاج لأستعمالها في تطوير تطبيقات الحاسوب. إنها تهدف أساسا إلى تبسيط عملية تطوير أنظمة متماثلة بإعادة استخدام برماجيات بطريقة إستراتيجية ومخططة مسبقا. المساهمة الرئيسية لهذا النموذج يندرج في تقديم بنية مرجعية لإدارة جميع التباينات. إنها توفر إطارا لتطوير مكونات قابلة لإعادة الاستعمال وتضمن إدماجها السليم. لكل منتوج، تستخدم كدليل للتجميع والتخصيص للمنتجات المصنعة اللازمة وفقا لاحتياجاته الخاصة. وقد لقي هذا التخصص نجاحا كبيرا من حيث الإنتاجية، لكن إدارة التباين مازال يتطلب جهودا كبيرة.

في إطار هذه الأطروحة، نحن مهتمون بشكل خاص بالتحليل الآلي للنماذج الـ FD وFTS . المخطط FD هو شكلية مستخدمة على نطاق واسع للمواصفات الهيكلية للمنتوجات. إنه على شكل شجرة تحدد الخصائص وتبعاتها. يستخدم النظام FTS لوصف السلوك. هو نظام انتقالات خاص حيث أن هذه الانتقالات مسماة مع خصائص مخطط FD، بالإضافة إلى وصمها بإجراءات. ويتم إنشاء مثيل بشكل مختلف لكل منتوج وفقا للخصائص المطلوبة.

نقدم في هذه الأطروحة ثلاثة مساهمات . الأولتين تسمحان بإنشاء جميع المنتوجات الصحيحة هيكليا من المخطط FD. ويستند الأول على إدماج التراجع في عملية البحث، في حين أن الأخر يعتمد على البناء التدريجي للتكوينات الجزئية. أما المساهمة الثالثة فتسمح بالتحليل السلوكي للمنتوجات المصممة بواسطة المخطط FTS. و يتم إجراء التحقق من الخصائص على أساس منطق إعادة الكتابة وبالضبط اللغة Maude.

يتم تطبيق التقنيات المقترحة استنادا إلى منهجية تحويل الرسوم البيانية، أما قواعد النحو يتم وضعها وتنفيذها باستخدام النظام $AToM^3$.

الكلمات الرئيسية: برامج سلسلة الإنتاج، FD، FTS، التحليل والتحقق، تحويل الرسوم البيانية، $AToM^3$، التراجع، منطق إعادة الكتابة، Maude.

Remerciements

Je tiens à remercier toutes les personnes m'ayant soutenue et encouragé pendant ces années de thèse.

Tout d'abord, un grand merci à mon directeur de thèse Mr Cherif Foudil pour tous ses conseils, sa confiance, ses idées, ses encouragements, ses corrections et ses remarques tout au long de ce travail.

je tiens a adresser mes plus chaleureux remerciements à Mr Allaoua Chaoui pour son engagement dans le co-encadrement de ma thèse. Sa compétence, sa rigueur scientifique et sa clairvoyance m'ont beaucoup appris. Ils ont été et resteront des moteurs de mon travail de chercheur.

Je tiens également à remercier tous les membres de mon jury pour avoir accepté d'examiner ce travail.

J'adresse toute ma gratitude à tous mes amis et à toutes les personnes qui m'ont aidé dans la réalisation de cette thèse. En particulier, je remercie vivement Omar et Tahar qui m'ont accompagné, encouragé et partagé avec moi de bons moments tout au long de ce chemin. Je souhaite remercier également tous mes collègues de travail.

Pour finir j'aimerais remercier toute ma famille pour leur soutien constant. Sidi, cette thèse t'est dédiée.

Table des Matières

Table des Matières

Chapitre 02 : Transformation de Modèles

Chapitre 03 : Génération Automatique des Produits Structurellement Valides: Approche Basée sur la Recherche des Configurations Correctes

Table des Figures

Table des Figures

Introduction

Générale

La réutilisation logicielle est une préoccupation fondamentale et récurrente depuis l'origine du génie logiciel. Son objectif principal est de réduire les coûts de développement logiciel en favorisant la réutilisation d'éléments logiciels préexistants. Malgré la multitude des approches et des outils ayant fait leur apparition sur le marché, les résultats attendus n'étaient pas suffisamment significatifs. En effet, le mécanisme de réutilisation de loin le plus usité est toujours l'inépuisable ''copier coller'' de morceaux de code. Cette réutilisation limitée est surtout opportuniste, c'est-à-dire qu'elle n'a pas été planifiée au préalable. Elle se base principalement sur l'expérience du développeur qui identifie un morceau de code, une classe ou une librairie développé dans un projet précédent afin de résoudre un problème relativement similaire. Dans la majorité des cas, ce type de réutilisation nécessite des adaptations conséquentes et potentiellement dommageables.

Pour remédier à ces inconvénients, la systématisation de la réutilisation constitue une solution très prometteuse. Dans ce contexte, le paradigme lignes de produits logiciels est apparu. Fondamentalement, cette approche propose d'adapter le principe des chaînes de productions au développement d'applications informatiques. Elle vise à rationaliser le processus de développement des systèmes fortement similaires. Ses principes de base ne sont pas nouveaux et ont été abordés dès 1968, respectivement par McIllroy [1] et Parnas [2]. Néanmoins, l'ingénierie des lignes de produits logiciels n'a émergé qu'au début de ce siècle avec l'apparition d'une communauté très active aussi bien dans des projets européens que dans des projets menés par le Software Engineering Institute (SEI) aux Etats-Unis. Une ligne de produits logiciels est définie comme un ensemble de systèmes partageant un ensemble de propriétés communes et satisfaisant des besoins spécifiques pour un domaine particulier. L'idée donc est de ne plus développer un seul logiciel à la fois, mais plutôt toute une famille de produits en systématisant la réutilisation d'éléments communs.

L'apport majeur des lignes de produits est l'introduction d'un mécanisme de gestion de la variabilité. Ce dernier permet de constituer les applications en réutilisant des artéfacts logiciels de manière stratégique pour s'adapter aux besoins requis. Cette approche se caractérise par la mise en place d'un double processus de développement, pour et par la réutilisation. Le but du premier processus, l'ingénierie du domaine, est de spécifier et d'implanter les composants logiciels communs à tous les logiciels de la ligne de produits, tout en identifiant les points de variation. Dans ce contexte, ils sont appelés caractéristiques. Le but du second processus, l'ingénierie applicative, est de produire des applications spécifiques en s'appuyant sur les artefacts mis à disposition par le processus précédent. Dans cet esprit, la construction d'un produit logiciel spécifique est un processus d'assemblage d'artéfacts réutilisables selon les besoins requis.

Cette approche distingue trois catégories de composants: les caractéristiques obligatoires pour tous les produits, les caractéristiques optionnelles qui sont présentes seulement dans certains produits et les caractéristiques alternatives. En plus, certaines caractéristiques peuvent nécessiter la présence d'autres. Ils peuvent également exclure leurs sélections. Ces contraintes impactent naturellement sur la configuration du produit considéré. Il faudra donc s'assurer qu'au moment de la sélection des différents composants, toutes ces dépendances soient respectées et que leur assemblage se fasse de manière correcte. Ceci ne peut évidemment se faire ni manuellement, ni de manière improvisée. C'est un processus qui demande des méthodes et des outils adaptés afin d'éviter toute source d'erreurs. Un obstacle majeur apparaît au niveau du choix des techniques de modélisation capables de gérer la notion de variabilité. Plusieurs formalismes ont été proposés. Ils sont basés principalement sur la définition d'une architecture commune et partagée par l'ensemble de produits logiciels dite architecture de référence. Elle apparaît généralement comme un modèle spécifiant l'ensemble des caractéristiques ainsi que leurs relations. Elle inclut également des points de variation et possède des mécanismes permettant de personnaliser les applications spécifiques afin de satisfaire les exigences demandées.

Dans le cadre de cette thèse, nous avons opté pour les modèles FD et FTS. Le diagramme FD est utilisé pour spécifier les produits d'un point de vue structurel. Il s'agit d'un arbre dont les nœuds représentent les caractéristiques réutilisables, alors que les arcs spécifient leurs dépendances. Chaque produit est défini en donnant sa structure par la sélection des composants nécessaires à son implémentation. Le modèle FTS est utilisé pour décrire l'aspect comportemental. C'est un système de transitions paramétré dans lequel les transitions sont étiquetées avec des caractéristiques en plus d'être marquées par les actions. Il est instancié différemment pour chaque produit en fonction des caractéristiques sélectionnées.

Malgré le grand succès de ces deux formalismes en termes d'expression, ils manquent de moyens de validation. Cette thèse vise à leur associer une plate forme de vérification formelle afin d'assurer les activités d'analyse nécessaires. Nous proposons trois outils automatiques basés sur l'approche transformation de graphes. C'est une technique largement utilisée dans l'ingénierie dirigée par les modèles (IDM). Elle est inspirée principalement des approches classiques de réécriture des termes. Les traitements souhaités sont réalisés par des grammaires de graphes composées de règles de transformation. Pour chaque règle, la partie gauche est destinée à être mise en concordance avec les fragments similaires du modèle traité alors que la partie droite décrit les modifications à réaliser.

Ce manuscrit est structuré selon le plan suivant :

Dans le premier chapitre, nous abordons le paradigme lignes de produits logiciels. Nous commençons par un rappel des concepts fondamentaux. Ensuite, nous présentons l'ingénierie des lignes de produits logiciels en donnant un aperçu sur l'ingénierie du domaine et l'ingénierie d'application. Ce chapitre se termine par un tour d'horizon sur les techniques de modélisation et d'analyse proposées dans la littérature.

Le deuxième chapitre sera consacré à la présentation des concepts clés de l'approche transformations de modèles. Nous introduisons d'abord quelques notions de l'IDM. Puis, nous nous intéressons à la technique transformation de graphes. Enfin, une présentation détaillée de l'outil d'implémentation utilisé sera donnée. Il s'agit de l'environnement AToM^3.

Dans les chapitres, trois et quatre, nous présentons respectivement deux approches différentes permettant la génération de tous les produits structurellement valides à partir du diagramme FD. La première approche est basée sur l'intégration du Backtracking dans le processus de recherche. C'est un algorithme largement utilisé dans la résolution des problèmes d'optimisation. Dans la seconde approche, nous procédons par une technique basée sur la composition des combinaisons partielles. A partir des caractéristiques, l'idée consiste à construire progressivement des configurations partielles valides plus larges jusqu'à l'obtention des produits recherchés.

Dans le dernier chapitre, nous présentons une approche pour l'analyse comportementale des produits. Il s'agit d'un outil de validation basé sur la logique de réécriture. L'idée de base est de générer d'abord une description formelle des modèles FTS en langage Maude. Ensuite, la vérification des propriétés sera effectuée par le biais du Model-Checker intégré.

Enfin, une conclusion générale résume ce manuscrit et présente quelques perspectives des travaux proposés dans cette thèse.

Chapitre 01

Lignes de produits Logiciels

1.1 Introduction

En industrie, la production de masse est née avec l'automatisation des processus favorisant l'assemblage d'éléments interchangeables sur des lignes de production. Prenons comme exemple la fabrication des voitures. Différents modèles sortent des mêmes chaînes de montage en utilisant les mêmes châssis, les mêmes moteurs et les mêmes plans de tests avec quelques différences. Les avantages d'une telle approche sont nombreux et peuvent être transposables au monde des logiciels. Les principales méthodes qui prônent ces principes ont été regroupées sous un même nouveau paradigme appelé ''Ingénierie des lignes de produits logiciels''.

Cette méthodologie vise à rationaliser le processus de développement des systèmes fortement similaires. Elle est basée principalement sur l'introduction d'une architecture de référence commune spécifiant les parties fixes, les parties variables ainsi que les choix possibles au niveau des points de variation. Son adaptation aux besoins spécifiques de chaque produit est assurée par un mécanisme de gestion de la variabilité.

Le but de ce chapitre est d'introduire les concepts clés de cette ingénierie. Un intérêt particulier sera porté sur les techniques de modélisation et d'analyse associées.

1.2 Réutilisation Logicielle

La réutilisation logicielle est une préoccupation fondamentale et récurrente depuis l'origine du génie logiciel [3]. L'idée de base est de construire les nouvelles applications par assemblage, plus ou moins complet, d'artefacts logiciels déjà existants. Dans la littérature, plusieurs stratégies de mises en œuvre ont été proposées. Elles se différencient par le grain et le niveau d'abstraction d'éléments réutilisés. Les plus intéressantes sont :

1.2.1 Réutilisation ad-hoc

C'est la solution la plus ancienne. Ici, il s'agit d'une réutilisation non préparée, mais entièrement décidée et mise en œuvre en fonction des besoins du moment. Elle peut être très avantageuse en ce sens qu'elle ne coûte rien au niveau de la préparation et qu'elle peut rapporter gros. Cette approche est également très risquée. La qualité et l'adéquation des artefacts réutilisés ne sont pas garanties. Elle dépend complètement du talent, et de la chance, du programmeur.

1.2.2 Approche orientée objet

Cette approche est basée sur le concept de classe [4]. La réutilisation repose sur les notions d'héritage et de polymorphisme. L'héritage permet explicitement l'adaptation de types par le mécanisme de surcharge, alors que le polymorphisme offre l'utilisation de mêmes interfaces sur

des objets différents. Malgré ces atouts, la pratique a démontré que la réutilisation des classes existantes lors du développement de nouvelles applications est une tache trop lourde. Le principal inconvénient réside dans fait que l'adaptation doit être réalisée à un niveau architectural trop bas.

1.2.3 Approches orientées composants

Les approches basées composants visent explicitement la réutilisation [5]. Elles perçoivent le développement d'applications logicielles comme un assemblage de composants, et gèrent la maintenance et l'évolution par la personnalisation et le remplacement de composants réutilisables [6]. Typiquement, un composant possède un ensemble d'interfaces définissant les fonctionnalités fournies et requises sous la forme d'un contrat spécifique et visible de l'extérieur. Cette représentation explicite de l'architecture élève le niveau d'abstraction par rapport aux assemblages d'objets. Cependant, plusieurs défis et limites entravent encore la mise en pratique de ces approches :

- Le développement d'une bibliothèque de composants réutilisables est souvent coûteux.
- Les dépendances de fonctionnalités causent un couplage très fort couplage contractuel entre certains composants d'une application. Cela réduit énormément le dynamisme et la flexibilité.

1.2.4 Approches orientée services

De même, une architecture à service regroupe un ensemble de services réutilisables ainsi que des mécanismes d'assemblage [7]. Mais, par opposition aux approches orientées composants, le contrat entre consommateur et fournisseur n'est mis en place que lors de la sélection du service suite à une négociation. Cela permet de :

- Réduire le couplage lié aux dépendances.
- Améliorer le niveau d'abstraction des applications.
- Elever la granularité de la réutilisation.

Toutefois, la composition de services suivant cette démarche est certainement une tâche complexe, source de nombreuses erreurs potentielles. Il s'est avéré qu'il est très difficile de vérifier la conformité d'une composition de services, incluant les conformités syntaxique et sémantique.

1.2.5 Réutilisation de modèles

La réutilisation ne se limite pas au code mais concerne tous les artefacts d'un développement logiciel. Les modèles en particulier sont des éléments de réutilisation très intéressants. Lors d'un développement logiciel, de nombreux modèles sont créés tels que des exigences, des architectures,…etc. C'est dans cette optique que les patterns de conception (design patterns) [8] sont apparus. Un pattern est un ensemble de modèles décrivant une solution à un

problème donné. Malheureusement, cette forme de réutilisation est d'une grande complexité du fait que les modèles constituent une abstraction partielle et très focalisée d'un système. Ils sont ainsi très liés à un contexte bien particulier.

1.2.6 Approche lignes de produits

C'est une approche basée principalement sur une réutilisation systématique et planifiée au préalable. L'idée principale est de viser dès le début le développement de toute une famille de systèmes fortement similaires [9]. Dès lors, le temps de mise sur le marché et les coûts de développement sont réduits pendant que la qualité des produits s'améliore. Dans le cadre de cette thèse, nous sommes intéressés particulièrement par cette approche. Par conséquent, une présentation des principaux concepts est donnée.

1.3 Concepts fondamentaux

L'approche ligne de produits logiciel a été investiguée, sous une forme ou sous une autre, depuis longtemps. Néanmoins, elle n'a émergé comme un nouveau paradigme du génie logiciel qu'à ces dernières années.

Dans la littérature, il y a un consensus sur la définition d'une ligne de produits logiciels :

*Une **ligne de produits logiciels** est un ensemble de systèmes partageant un ensemble de propriétés communes et satisfaisant des besoins spécifiques pour un domaine particulier développés de manière contrôlée à partir d'un ensemble commun d'éléments réutilisables* [10].

En suivant cette démarche, toute une famille de produits est construite en regroupant les connaissances logicielles réutilisables concernant un domaine spécifique.

*Un **domaine** est un secteur de métier ou de technologies ou des connaissances caractérisées par un ensemble de concepts et de terminologies compréhensibles par les utilisateurs de ce secteur* [11].

Dans cette approche, les composants réutilisables ne se limitent pas seulement au code, mais peuvent prendre la forme d'un document, d'un modèle, d'une conception…etc. Ils sont représentés par des caractéristiques (features). La définition donnée par Kang [12] donne une idée plus précise sur ce concept.

*Une **caractéristique** représente tout aspect important et distinctif ou caractéristique visible par les diverses parties prenantes.*

Une ligne de produit est spécifiée par une architecture dite architecture de référence.

*Une **architecture de référence** est la description d'un ensemble de composants et de leurs relations permettant de construire des systèmes logiciels appartenant à une ligne de produits. De*

plus, l'architecture de référence doit contenir les informations nécessaires pour gérer la variabilité au sein des systèmes [13].

L'architecture de référence fournit le cadre de développement des composants réutilisables et garantit leur incorporation appropriée. Pour chaque produit, elle est utilisée comme guide d'assemblage des artéfacts nécessaires. Généralement, elle apparaît comme un modèle spécifiant les parties fixes, les parties variables et les choix à faire afin de personnaliser les applications spécifiques. Cette variabilité est introduite par des points de variation.

*Un **point de variation** identifie un ou plusieurs emplacements auxquels la variation peut se produire* [14].

Les éléments logiciels communs à tous les membres de la ligne de produit sont appelés ***similarités***, alors que les éléments logiciels variant d'un membre de la ligne de produits à un autre sont appelés ***variabilités***. Elles peuvent dépendre de différents facteurs : techniques, commerciaux ou culturels.

En plus, l'architecture de référence contient également toutes les règles et les dépendances caractérisant la solution architecturale du domaine. Principalement, elles sont liées à des considérations techniques. Toutes les architectures des applications spécifiques doivent être conformes à l'architecture de référence de la ligne de produits. En effet, la résolution d'un point de variation peut influencer la résolution d'autres points de variation. Un élément peut exclure ou exiger d'autres éléments.

La construction d'un produit particulier consiste à figer certains choix vis-à-vis de la variabilité définie dans l'architecture de référence. Elle se limite à sélectionner les éléments logiciels spécifiques au produit considéré en respectant les contraintes imposées. On parle ici de dérivation de produit.

*La **dérivation** produit est le processus de construction d'un produit à partir d'une Ligne de produits logiciels* [15].

La mise en place d'une ligne de produits demande un investissement initial important. Développer une ligne de produits est complexe pour de multiples raisons. Tout d'abord, l'identification des points de variabilité sur les différents composants logiciels est très difficile. Cela demande une analyse minutieuse des différentes déclinaisons possibles d'un artéfact et de maîtriser les techniques de modélisation et de codage permettant d'introduire un certain niveau de variabilité. En plus, la phase de maintenance est rendue plus complexe. Il ne s'agit plus de gérer les versions successives d'un même produit mais, maintenant, de gérer de façon conjointe l'évolution de plusieurs produits similaires et de leurs artéfacts réutilisables. Il faut ainsi gérer l'évolution des

éléments du domaine ainsi que les liens de dépendances avec les différents produits déjà construits. Ainsi, au même titre que le développement logiciel classique, le développement d'une ligne de produits logiciels est devenu toute une discipline en génie logiciel. Les travaux sur ce sujet ont donné naissance à une nouvelle ingénierie dite ''Ingénierie des lignes de produits logiciels'' [10]. Elle sera détaillée dans la section suivante.

1.4 Ingénierie des lignes de produits logiciels

La mise en œuvre d'une ligne de produits logiciels regroupe deux processus interdépendants. La figure suivante (Fig.1.1) illustre précisément ces deux processus de développement en donnant certaines de leurs activités.

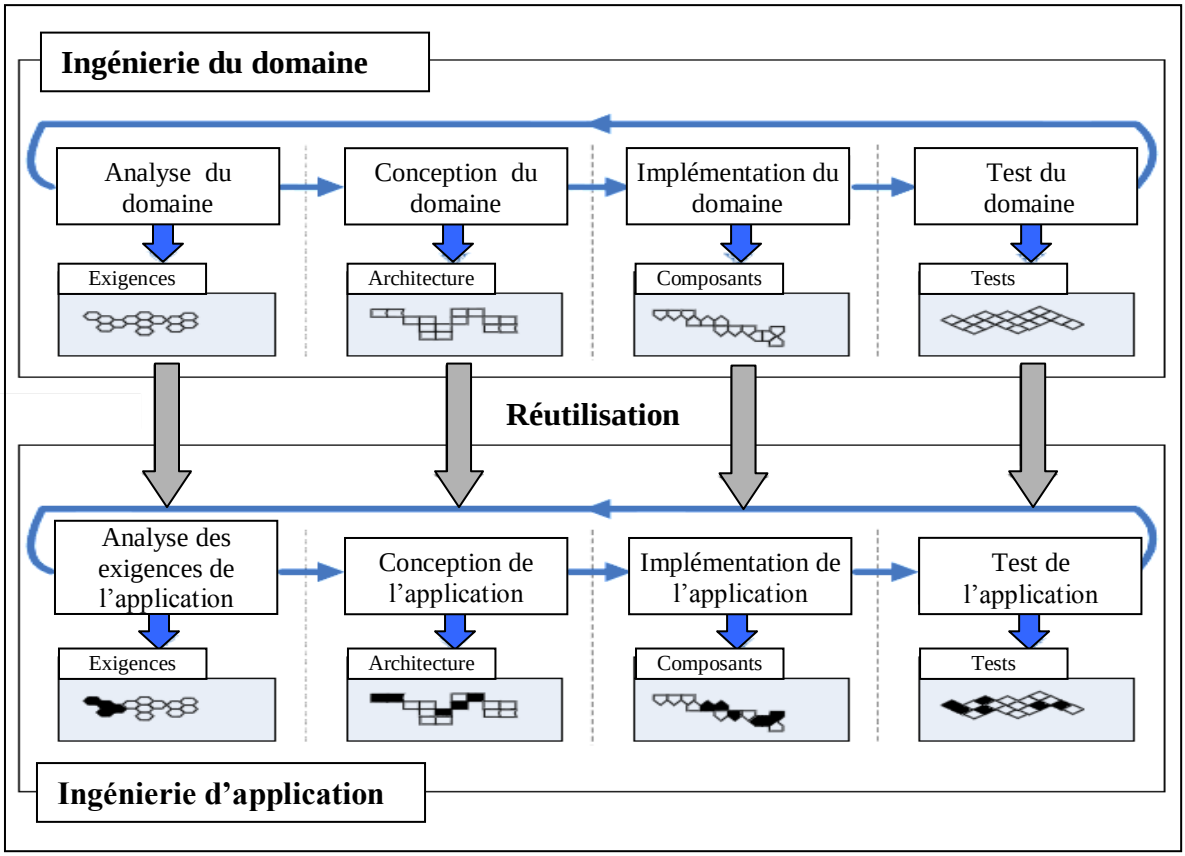


Figure.1.1: Ingénierie des lignes de produits logiciels

- Le premier processus, appelé ingénierie du domaine (Domain Engineering), a pour but le développement des composants réutilisables. C'est un processus de développement pour la réutilisation en ce sens qu'il prépare les éléments qui seront plus tard réutilisés.
- Le second processus, appelé ingénierie des applications (Application Engineering), a pour but de construire des applications spécifiques. C'est un processus de développement par la réutilisation puisqu'il se base sur les composants logiciels réutilisables développés lors de l'ingénierie du domaine.

1.4.1 Ingénierie du domaine

L'ingénierie du domaine consiste à la mise en place de la plate-forme de réutilisation. Cette dernière comprend tous les artéfacts logiciels couvrant toutes les activités d'une production logicielle allant de l'analyse des besoins du domaine jusqu'aux phases de test. Ils sont de natures différentes et doivent tous être testés. Cette plate-forme comprend également des stratégies et des procédures de production systématique des produits individuels.

Dans ce premier niveau, nous distinguons quatre activités: l'analyse, la conception, l'implantation et le test du domaine.

Analyse du domaine: L'analyse du domaine est une activité visant à définir un domaine en formalisant toutes les connaissances initiales. Il s'agit d'une activité de transformation des exigences métier vers des représentations informatiques. Elle doit identifier les fonctionnalités du domaine, les contraintes ainsi que les points communs et variables. Une étape fondamentale de l'analyse du domaine est de déterminer la portée de la ligne de produits [16]. Celle-ci définit explicitement l'ensemble des produits à étudier et implicitement les futurs candidats pourront faire partie de cette ligne de produits.

Conception du domaine: La conception du domaine repose sur les résultats obtenus à l'issue de l'activité précédente. Elle consiste à concevoir les éléments logiciels qui feront partie de la plate-forme de la ligne de produits. A ce niveau, deux types d'éléments particulièrement importants sont à définir :

- Les composants logiciels réutilisables : ils sont généralement configurables.
- L'architecture de référence : Elle définit de façon abstraite les composants structurels des produits et leurs relations. Généralement, les experts du domaine s'appuient sur l'ensemble des modules communs identifiés lors de l'analyse du domaine pour obtenir une première structuration de cette architecture. Ensuite, elle sera affinée successivement de façon à faire apparaître les points de variation.

Implantation du domaine: Dès que la description de l'architecture est créée, les développeurs procèdent à l'implémentation des composants réutilisables.

Test du domaine : Cette activité est responsable de la validation et la vérification des composants prêt à être utilisés. Ils doivent tous être testés et leur fiabilité doit être avérée afin d'assurer un développement de haute qualité.

En résumé, le développement de l'ingénierie des domaines est un processus itératif qui vise à la construction des artefacts et des outils qui seront utilisables durant la phase suivante, à savoir la phase d'ingénierie applicative.

1.4.2 Ingénierie d'application

L'ingénierie d'application consiste à la création et la mise à disposition des logiciels finaux en se basant sur la plate-forme établie lors de la phase d'ingénierie du domaine. Plus précisément, la construction d'un produit spécifique est un processus d'assemblage et de configuration des composants réutilisables en conformité avec l'architecture de référence. Il s'agit d'une instanciation particulière qui répond à ses exigences spécifiques. L'application construite s'inscrit automatiquement dans la portée de la ligne de produits.

Comme dans l'ingénierie du domaine, la mise en œuvre de l'ingénierie des applications est constituée de quatre activités principales :

Analyse des exigences d'applications : La première activité de l'ingénierie des applications est l'analyse des exigences du produit visé. Les exigences spécifiques [17] sont obtenues à partir des interactions avec les différents intervenants tels que les utilisateurs, les manageurs ou les techniciens de maintenance. Cette étape est primordiale. Elle met en évidence les besoins spécifiques de chaque produit et sert de guide pour tous les travaux d'adaptation nécessaires.

La conception d'applications et l'implémentation : A partir du résultat de l'activité de l'analyse des exigences spécifiques, l'architecture du produit visé est dérivée de l'architecture de référence conçue lors de l'ingénierie du domaine. Dans un premier temps, les points de variation définis sont instanciés et configurés afin de créer une architecture applicative. La résolution d'un point de variation peut dépendre des autres points de variation de l'architecture. Pour éviter toute source d'erreurs, les dépendances existantes doivent être considérées. Ensuite, la correspondance avec les artéfacts d'implémentation sera réalisée.

Test de l'application : De nombreux tests ont déjà été spécifiés et exécutés pour les composants réutilisables lors de l'ingénierie d'application. L'objectif de cette phase est de s'assurer que leurs interactions dans les applications spécifiques ne génèrent pas de comportements inattendus.

Il est clair que les deux processus de cette ingénierie doivent parfaitement s'articuler du fait qu'ils sont fortement interdépendants. Leur complexité n'est certainement pas négligeable. Une gestion efficace et correcte de la variabilité est un facteur de réussite déterminant lors de l'utilisation d'un processus de ce type. D'où la nécessité d'une architecture de référence très puissante. Dans la section suivante, on présentera les solutions proposées du point de vue modélisation.

1.5 Modélisation de la variabilité

La variabilité est un concept clé dans le contexte des lignes de produits logiciels. Comme vue précédemment, c'est la base d'un développement par réutilisation d'artéfacts ajustables. Elle est introduite lors de l'ingénierie du domaine pour spécifier les différences entre les produits planifiés. Pour la modéliser, plusieurs formalismes ont été proposés. Ils se distinguent principalement par la nature de la diversité introduite qu'elle soit statique ou dynamique.

Pour l'aspect statique, l'objectif est de modéliser uniquement la structure de l'architecture de référence. Il s'agit de définir l'organisation des différents composants en spécifiant tous les points de variations ainsi que les contraintes associées. Par contre, l'aspect dynamique fait référence au comportement des produits. Il se focalise principalement sur la spécification des changements d'états possibles en prenant en compte les différents points de variation. Dans ce qui suit, on présentera un survol des approches et formalismes proposés.

1.5.1 Feature-Oriented Domain Analysis (FODA)

FODA [12] est le premier formalisme dédié spécifiquement à la modélisation de la variabilité. Plusieurs constructions graphiques et textuelles sont utilisées pour modéliser les composants et ses relations au moyen d'une structure arborescente particulière appelée diagramme de fonctionnalités (FD). Un exemple de ce modèle est présenté à la figure Fig.1.2.

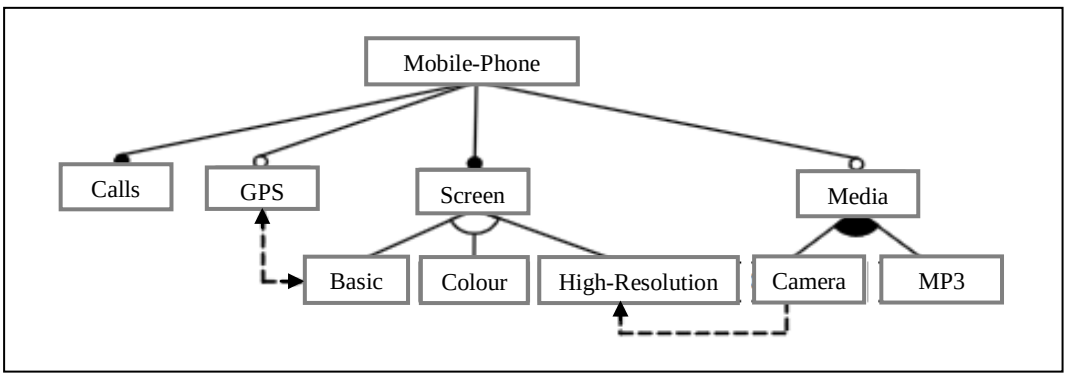


Figure.1.2: Diagramme de caractéristiques

Suite à la publication de FODA, plusieurs extensions de ce formalisme ont été proposées afin de compenser certaines limites des diagrammes FD [18]. Les plus intéressantes sont: FORM [19], FeatuRSEB [20], Cardinality-Based Feature Modeling (CBMF) [21] [22], l'approche de Riebisch [23], Forfamel [24] et RequiLine [25]. Ces variantes sont basées sur des enrichissements spécifiques du diagramme FD de base. Citons à titre d'exemple : les cardinalités entre les fonctionnalités, les attributs associés aux fonctionnalités ou encore de nouveaux types de relations.

1.5.2 Techniques basées sur les diagrammes UML

Plusieurs auteurs se sont intéressés à la modélisation des lignes de produits en UML. Ils s'appuient principalement sur une adaptation des diagrammes UML classiques. La variabilité est spécifiée par des mécanismes d'extension standards tels que les Tagged-values, les stéréotypes et les contraintes [26] [27] [28].

A titre d'exemple, Ziadi dans [28] a proposé d'introduire la variabilité dans les diagrammes de séquence (Fig.1.3). Elle concerne les objets participants ainsi que le comportement lui-même.

- Un objet optionnel est spécifié par le stéréotype ''*OptionalLifeLine* ''. Cela signifie que tous les messages envoyés ou reçus par cet objet sont facultatifs. Ils peuvent donc être supprimés pour certains produits.
- Une interaction optionnelle est définie par le stéréotype ''*OptionalInteraction*''. Cela signifie que tous les messages contenus dans cette interaction peuvent être ignorés dans certains produits.
- La variabilité du comportement est modélisée comme une interaction stéréotypée ''*variation*''. L'ensemble des sous-interactions sont référencées par le stéréotype ''*variant*''.

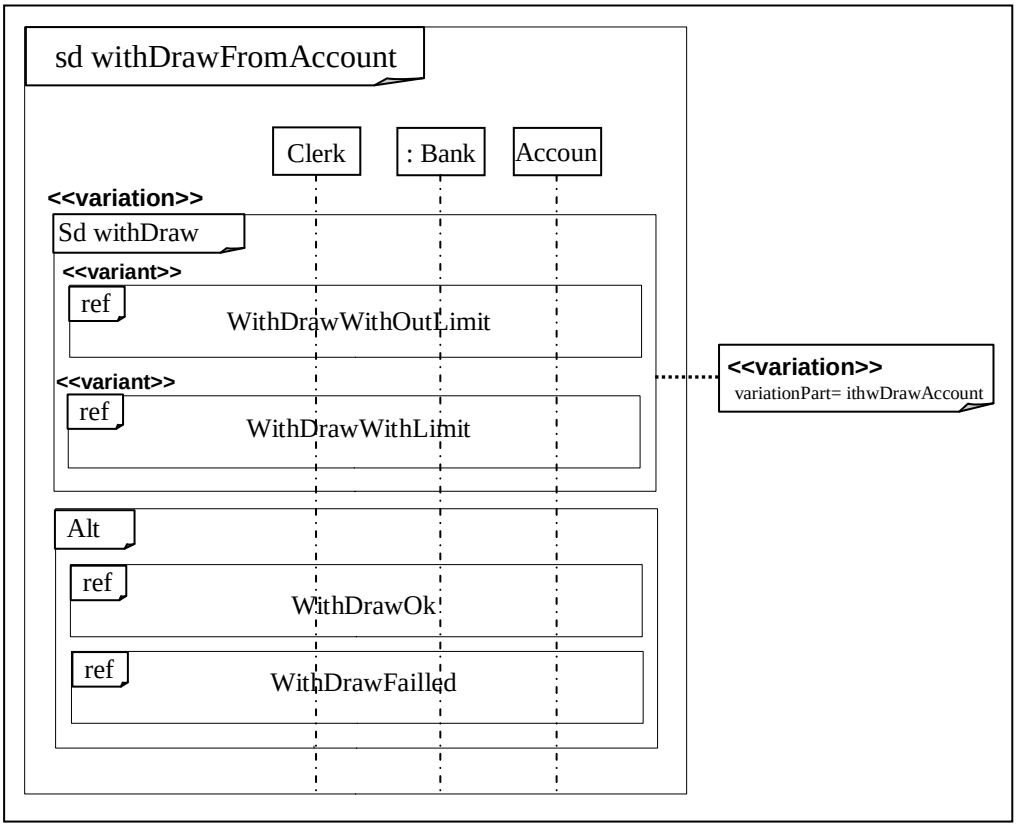


Figure.1.3: Diagramme de séquence étendu [28]

Les extensions d'UML pour modéliser les lignes de produits présentent l'avantage d'utiliser des diagrammes très connus dans le milieu de l'informatique. Toutefois, la mise en œuvre de ces techniques est d'une grande difficulté. En fait, il est habituellement plus naturel de penser à un produit en termes de fonctionnalités plutôt que d'éléments UML.

1.5.3 Techniques basées sur des spécifications algébriques

Dans ce contexte, le formalisme le plus utilisé est le PL-CCS (Product line CCS). C'est une extension de l'algèbre des processus classique CCS proposée par Gruler et al [29]. Chaque produit est considéré comme étant constitué d'un ensemble de processus. La dynamique de toute la famille est spécifiée en introduisant des points de variation au niveau des processus. Pour ce faire, les auteurs utilisent principalement :

- L'operateur classique ‖ : il est utilisé pour exprimer le parallélisme.
- L'operateur classique + : il est utilisé pour exprimer les choix non déterministes.
- Un nouveau operateur ⊕: il est ajouté afin d'introduire la variabilité. Il permet de spécifier les comportements alternatifs.

Les processus sont exprimés en utilisant le langage μ-calculus [30].

1.5.4 Techniques à base des systèmes de transitions

Les formalismes les plus remarquables sont les suivants :

Feature Pétri-Nets (FPN) : Ce modèle a été introduit par Muschevici [31]. C'est un type particulier des réseaux de pétri dans lequel les transitions sont munies des conditions d'application. Il s'agit de formules booléennes logiques définies sur un ensemble de caractéristiques. En plus de la présence des jetons dans les places d'entrée, l'application de chaque transition est conditionnée par la vérification de la formule booléenne associée.

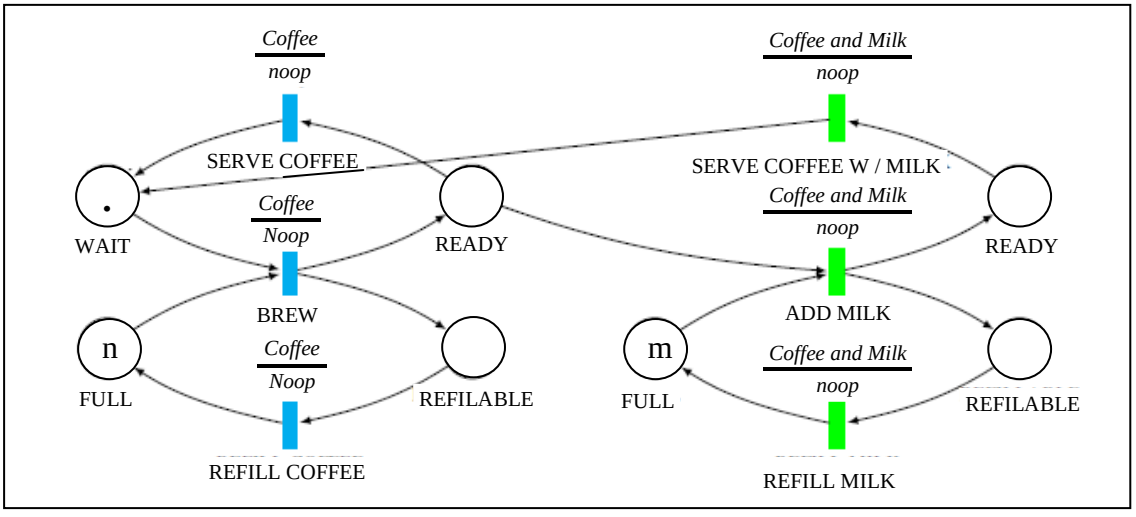


Figure.1.4: Feature Petri-Nets [31]

Modal Transition System (MTS) : Ce formalisme a été proposé par Fantechi et *al.* [32]. C'est une extension des LTS (Labelled Transition Systems) en introduisant des modalités au niveau des transitons. Ces dernières sont regroupées en deux catégories :

- *Obligatoires* : C'est les transitions nécessaires. Elles doivent être présentes dans toutes les implémentations.
- *Facultatives :* C'est les transitions optionnelles. Elles peuvent être présentes dans une implémentation mais pas nécessairement.

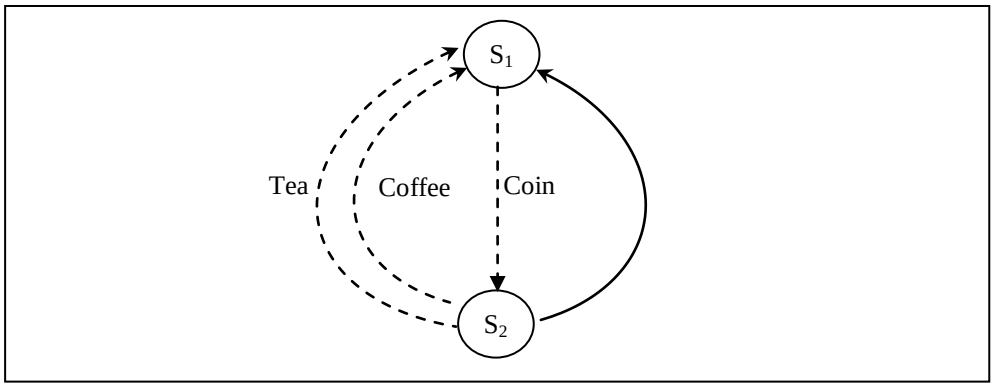


Figure.1.5: Modal Transition System [32]

Ce formalisme est largement utilisé pour le raffinement modal. Il s'agit d'une technique permettant de spécifier un comportement à plusieurs niveaux de précision.

Featured transition system (FTS) : Ce formalisme a été proposé par Classen et *al.* [33] pour une spécification comportementale de toute une famille de produits. La variabilité est introduite en utilisant un système de transition paramétré. En plus d'être étiquetées par des actions ordinaires, les transitions sont marquées par les caractéristiques nécessaires à leurs déclenchements. Pour un produit donné, chaque transition n'est activée que si sa caractéristique associée est sélectionnée.

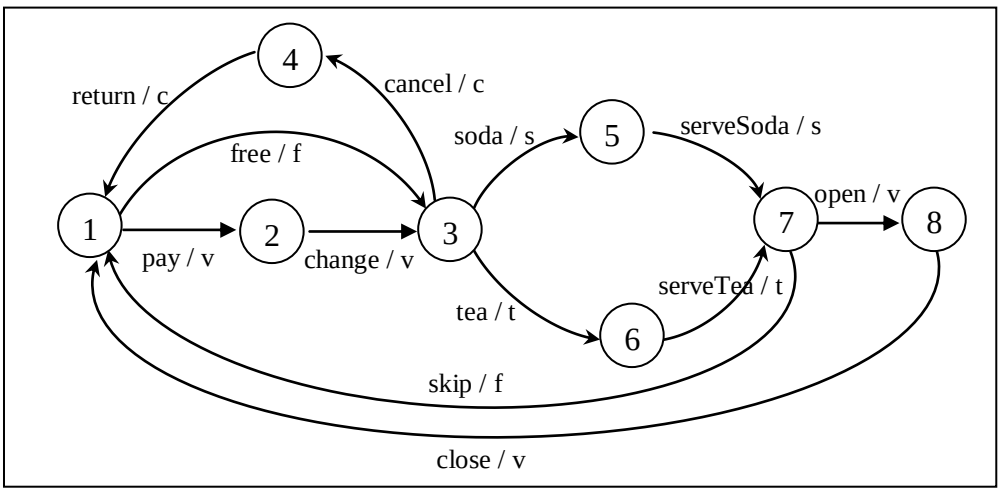


Figure.1.6: Featured Transition System [33]

1.6 Techniques d'analyse et de vérification

Le succès de l'approche ''ligne de produits'' est conditionné par la qualité des futurs produits envisagés. De même que pour le développement d'applications classiques, l'activité d'analyse structurelle ainsi que celle de vérification comportementale sont des tâches primordiales.

1.6.1 Analyse structurelle

Pour l'aspect statique, le diagramme de caractéristiques reste toujours reconnu comme l'une des contributions les plus intéressantes en termes de modélisation. L'analyse de ce modèle avait déjà été identifiée comme une tâche essentielle dans le rapport initial de FODA [12]. Elle est d'une grande importance dans le processus de développement suivant cette démarche. Dans la littérature, une grande variété d'opérations associées, qualifiées de pertinentes, ont été identifiées. Les plus importantes sont :

- Lister l'ensemble des produits respectant les dépendances imposées par le diagramme de caractéristiques.
- Vérifier l'existence d'au moins un produit satisfaisant ces dépendances.
- Vérifier la validité d'un produit donné vis-à-vis ces contraintes.
- Identifier les caractéristiques mortes. Ce sont les composants qui n'apparaissent dans aucun produit. Pour des raisons d'optimisation, il sera intéressant de les supprimer. .
- Identifier les caractéristiques communes qui apparaissent dans tous les produits.

En pratique, un diagramme FD réel peut contenir des centaines de caractéristiques et de contraintes. Ainsi, la tâche d'analyse devient très complexe. Au cours des dernières années, des avancées significatives ont été réalisées et une grande variété de techniques d'analyse a été proposée. Les plus intéressantes sont :

- Travaux basés sur les solveurs CSP : Dans cette optique, les opérations souhaitées lors de l'analyse d'un diagramme FD sont exprimées comme des problèmes de satisfaction de contraintes (CSPs). Un CSP est constitué d'un ensemble de variables avec des domaines associés et d'un ensemble de contraintes. La résolution consiste à trouver les états (valeurs pour variables) dans lesquels toutes les contraintes sont satisfaites. Cette solution a été présentée dans plusieurs articles [34] [35].
- Travaux basés sur les solveurs SAT : Chaque traitement souhaité est exprimé comme une formule propositionnelle. C'est une expression constituée sur un ensemble de variables booléennes (littérales) reliées par des opérateurs logiques. Pour limiter le problème de satisfiabilité, la formule est spécifiée dans la forme CNF en utilisant les équivalences logiques. Les concepts de base de cette solution ont été introduits dans [36] [37].

- Travaux basés sur l'utilisation des diagrammes de décision : Un diagramme de décision binaire (BDD) est une structure de données utilisée pour représenter une fonction booléenne. C'est un graphe dirigé, acyclique composé de deux types de nœuds : des nœuds intermédiaires de décision et des feuilles booléennes. Les chemins de la racine aux feuilles de valeur vraie représentent les affectations de variables pour lesquelles la fonction booléenne est considérée comme étant valide. Par contre, ceux qui terminent par des feuilles de valeur nulle, représentent les affectations de variables pour lesquelles elle est considérée comme étant fausse. Dans le contexte de l'analyse automatisée des FDs, plusieurs travaux basés sur l'usage des BDD ont été proposés [38] .
- Sun et al. [39] proposent d'utiliser le langage de spécification Z afin de fournir une sémantique formelle aux diagrammes de caractéristiques. La vérification automatique des propriétés est réalisée en utilisant le langage Alloy.
- Wang et al. dans [40] ont proposé une approche de modélisation des diagrammes de caractéristiques basée sur les ontologies OWL. Les auteurs ont également développé quelques outils automatiques de vérification.
- Osman et al. [41] proposent une méthode à base de connaissances pour analyser les modèles de caractéristiques. Dans leur contribution, les modèles de caractéristiques sont représentés comme une base de connaissances contenant des prédicats et des règles définis en utilisant la logique du premier ordre.

1.6.2 Vérification comportementale

En ce qui concerne l'aspect comportemental, l'approche de vérification la plus utilisée est le Model-Checking. C'est une démarche formelle constituée de trois étapes (Fig.1.6):

- Modélisation de la ligne de produits: C'est l'étape clé de cette technique. Elle sert à générer une représentation dynamique de tous les produits de façon précise et sans ambiguïté. Généralement, il s'agit d'un automate fini constitué d'un ensemble d'états et de transitions.
- Spécification des propriétés : les propriétés désirées doivent être décrites dans l'une des logiques temporelles. C'est des extensions de la logique propositionnelle classique avec des opérateurs faisant référence aux comportements au fil du temps.
- Analyse et vérification : La vérification d'une propriété nécessite une exploration de tous les états possibles. Si un état rencontré viole la propriété en cours d'examen, le Model-Checker fournit un contre-exemple en indiquant son chemin d'exécution.

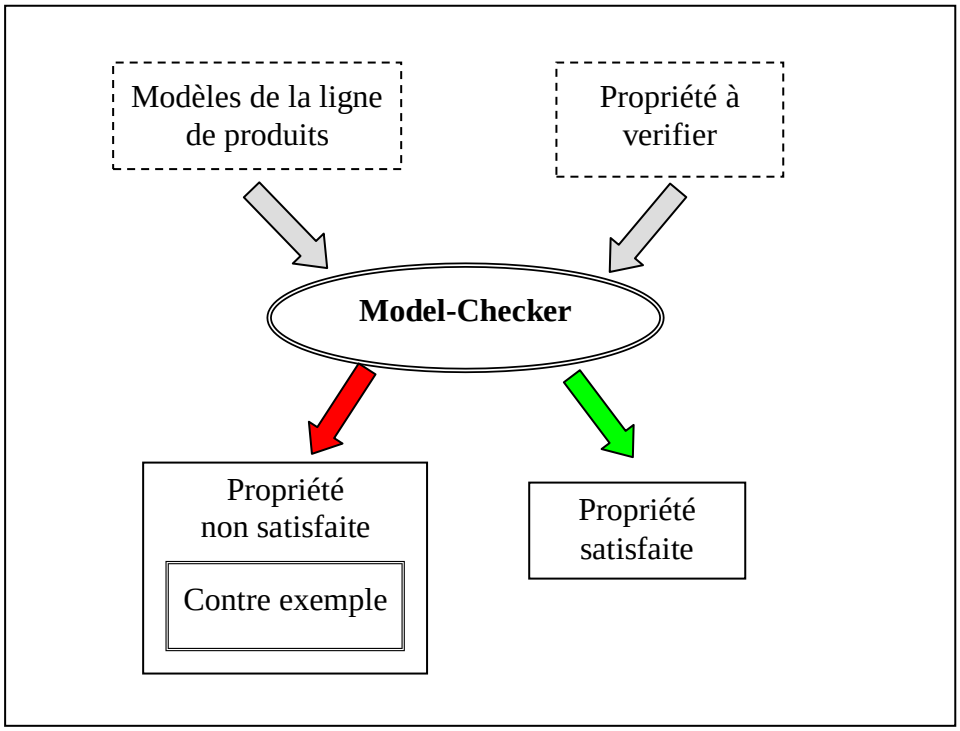


Figure.1.7: Approche du Model-Checking

Cette technique est très adaptée pour la vérification des lignes de produits modélisées par des systèmes de transitions. Dans ce contexte, des avancées significatives ont été réalisées et une grande variété de travaux ont été proposés. Les plus intéressants sont :

- Classen et *al.* [33] [42]: Dans ces deux travaux, les auteurs ont proposé respectivement deux techniques de vérification formelle des modèles FTSs. La première permet une validation des propriétés exprimées en LTL, alors que la seconde est spécifique à l'analyse des propriétés CTL. Le Model-Checking est réalisé par le standard NuSMV.
- Fischbein et al. [43] : les auteurs ont développé un Model-Checker pour le formalisme MTS noté MTSA (Modal Transition System Analyzer). Les modèles sont exprimés en algèbre FFP. Cet outil supporte la vérification des propriétés spécifiées en FLTL.
- Liu et al.[44] : les auteurs ont proposé une Model-Cheker pour les modèles FSMs. Ils ont implémenté une technique d'analyse basée sur la composition de vérifications partielles afin d'éviter des traitements répétitifs.

Pour plus de détails, voir [45], où Benavides a présenté une revue exhaustive de la littérature sur les techniques et les outils les plus importants.

1.7 Discussion autour de l'approche lignes de produits

D'un point de vue technique, la prise en compte de la notion de variabilité tout au long du cycle de vie différencie nettement l'approche des lignes de produits des approches classiques de réutilisation. Cela permet de satisfaire les besoins de personnalisation à tous les niveaux et se traduit par :

- Une augmentation de la productivité.
- Une diminution des coûts de développement et de maintenance des produits finaux.
- Une augmentation de la qualité des produits finaux.
- Enfin, une réduction des risques et des coûts liés à l'élaboration de nouveaux produits.

La figure suivante (Fig.1.7) montre les gains obtenus en termes de couts pour ma mise sur marché.

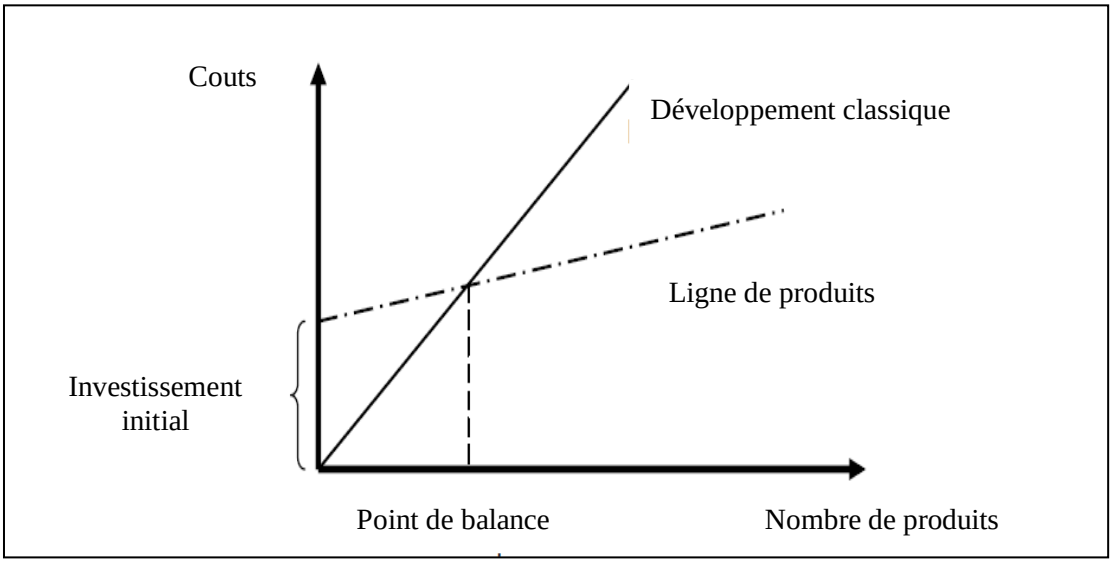


Figure.1.8: Aspect économique de l'ingénierie des lignes de produits [46]

Ces résultats affirment que :

- Les coûts des premiers logiciels développés à partir d'une ligne de produits sont relativement élevés par rapport au développement classique.
- A partir d'un certain nombre de logiciels (point de balance), des bénéfices importants sont consentis.

Il est clair que cette approche est très adaptée pour une production à grande échelle. Récemment, elle a été appliquée avec succès dans plusieurs grands projets européens, y compris ARES, PRAISE, ESAPS et CAFE [47].

Cependant, des efforts conséquents liés à la gestion de la variabilité doivent être envisagés. Ils peuvent être regroupés en trois activités principales :

- L'identification de la variabilité. Ce processus consiste à déterminer :
    - Les caractéristiques et les points de variations.
    - La structuration de ces caractéristiques : ceci est réalisé par l'introduction d'une architecture de référence permettant de spécifier toutes les dépendances et les contraintes nécessaires.
- L'implémentation de la variabilité. Il s'agit de la mise en œuvre d'un mécanisme permettant de retranscrire cette variabilité au niveau de l'architecture ou du code.
- Gestion de l'évolution de la ligne de produits. Cette activité permet d'éviter l'introduction de conflits ou la création d'interactions inattendues en cas d'ajout de nouvelles caractéristiques ou de nouveaux points de variation. Tout changement portant sur un élément logiciel partagé par plusieurs produits doit être contrôlé méticuleusement.

1.8 Conclusion

Dans ce chapitre, nous avons présenté les principes fondamentaux de l'ingénierie des lignes de produits logiciels. C'est une approche de développement basée typiquement sur une réutilisation anticipée et planifiée. Elle se caractérise par la mise en place de deux processus de développement complémentaires :

- L'ingénierie du domaine : elle consiste à la mise en place de la plate-forme de réutilisation.
- L'ingénierie d'application : elle consiste à la création et la mise à disposition des applications spécifiques.

L'architecture de référence joue un rôle fondamental. Elle fournit un cadre de développement des composants réutilisables et garantit leur incorporation appropriée. Pour chaque produit, elle est utilisée comme guide d'assemblage et de personnalisation des artéfacts nécessaires selon ses besoins spécifiques.

Une partie importante du chapitre était consacrée aux techniques de modélisation et d'analyse des lignes de produits. Pour la modélisation, deux des formalismes parmi les plus remarquables sont le diagramme FD et le modèle FTS. Le premier fournit une description structurelle détaillée de tous les produits, alors que le second est dédié à leurs spécifications comportementales. Ils sont qualifiés d'un grand pouvoir expressif doté d'une simplicité attirante due à leurs caractères graphiques. Le reste des travaux proposés s'appuient généralement sur les diagrammes UML. La variabilité est introduite par des mécanismes d'extension standards. Concernant l'analyse, les contributions les plus intéressantes sont celles basées sur la programmation par contraintes. L'idée principale est de formuler textuellement chaque opération souhaitée sous forme d'un problème combinatoire. La résolution est effectuée en utilisant des outils classiques tels que les solveurs CSP et SAT. Ces approches sont très intéressantes en termes d'operateurs proposés, mais elles demandent un effort considérable dû à leurs spécifications textuelles.

En résumé, on a constaté que l'une des solutions les plus prometteuses consiste à enrichir les formalismes graphiques par des outils automatiques d'analyse. Il s'avère que l'ingénierie dirigée par les modèles offre tous les moyens nécessaires. Elle fera l'objet du chapitre suivant.

Chapitre 02

Transformation de Modèles

2.1 Introduction

En génie logiciel, l'approche ligne de produits est une solution très intéressante pour un développement en masse des systèmes fortement similaires. Cependant, les formalismes de modélisation utilisés manquent souvent de fondements formels. Pour assurer la correction structurelle et comportementale des produits, les diagrammes établis nécessitent alors une analyse rigoureuse.

Dans ce contexte, l'ingénierie dirigée par les modèles (IDM), associée aux techniques formelles apparaît comme la solution la plus prometteuse. L'idée principale est d'introduire les techniques de vérification classiques dans le processus d'analyse en se basant sur des transformations de graphes.

Ce chapitre est dédié à la présentation des concepts clés de l'IDM. Dans un premier temps, nous introduisons les principes de la transformation de modèles. Nous présentons également une classification des approches de développement basées sur cette ingénierie. Nous détaillons ensuite l'approche transformation de graphes en donnant un aperçu sur les grammaires et les règles de transformations. Enfin, Nous concluons par une présentation de l'environnement AToM^3.

2.2 Ingénierie Dirigée par les Modèles

Suite à l'approche objet des années 80 et de son principe du ‘ *''tout est objet''*, l'ingénierie du logiciel s'oriente aujourd'hui vers l'ingénierie dirigée par les modèles (IDM) et le principe du ‘ *''tout est modèle''* [48]. Cette démarche reflète l'évolution du processus de développement logiciel de l'utilisation contemplative à l'utilisation productive des modèles. Là où ils étaient utilisés comme des éléments de conception, de discussion ou de documentation, l'idée de l'IDM est de les exploiter par la machine lors du processus de développement. En pratique, cela nécessite une spécification formelle. Dans la terminologie de l'IDM, les programmes permettant de traiter les modèles sont regroupés sous le terme ‘‘transformations de modèles‘‘.

2.2.1 Langage de modélisation

La première notion importante est la notion de modélisation. Quelle que soit la discipline scientifique considérée, un modèle est une abstraction d'un système construite dans un but précis. C'est une abstraction dans la mesure où chaque modèle contient un ensemble restreint d'informations sur un système. Il est construit dans un but précis dans la mesure où les informations qu'il contient sont choisies pour être pertinentes vis-à-vis de l'utilisation qui en sera faite. On dit alors que le modèle *représente* le système.

La notion de modèle dans l'IDM fait explicitement référence à la notion de langage bien défini. En effet, pour qu'un modèle soit productif, il doit pouvoir être manipulé par une machine. Le langage dans lequel ce modèle est exprimé doit donc être clairement défini. De manière naturelle, la définition d'un langage de modélisation prend la forme d'un modèle, appelé méta-modèle. Il représente les concepts du langage de méta-modélisation utilisé et la sémantique qui leur est associée [49]. On dit qu'un modèle est *conforme* à son méta-modèle.

De manière analogue, pour pouvoir interpréter un méta-modèle il faut disposer d'une description du langage dans lequel il est écrit : un méta-modèle pour les méta-modèles. C'est naturellement que l'on désigne ce méta-modèle particulier par le terme de méta-méta-modèle. Les méta-méta-modèles sont auto-descriptifs, c'est-à-dire capables de se définir eux-mêmes. L'organisation de la modélisation de l'OMG est basée sur ces principes. Généralement, elle est décrite sous une forme pyramidale à quatre niveaux.

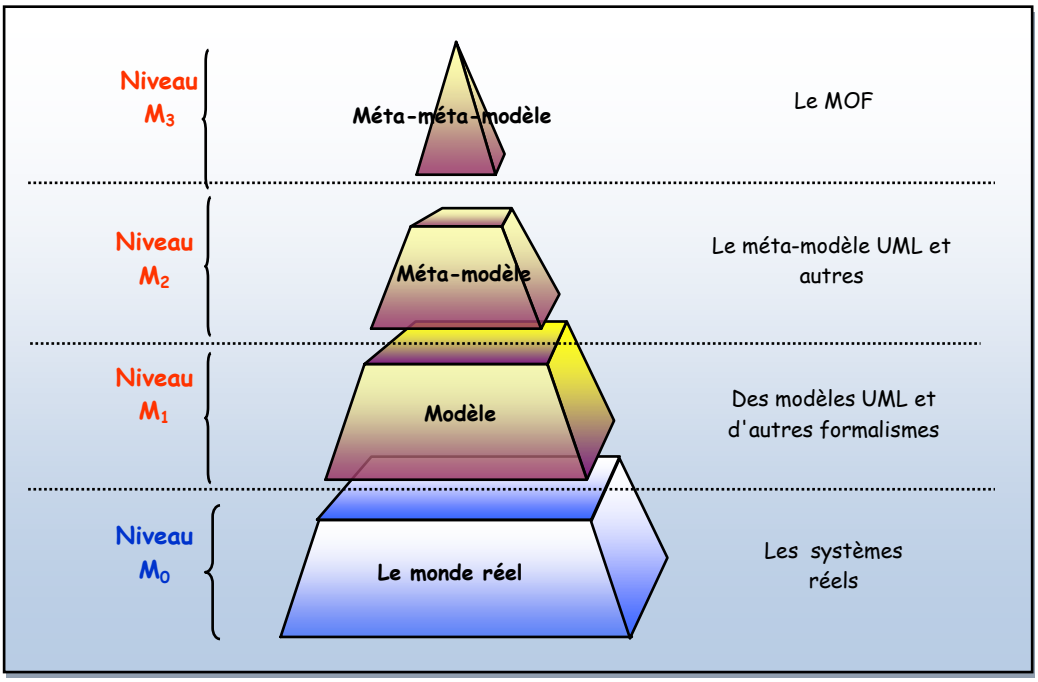


Figure 2.1: Pyramide de modélisation de l'OMG

- Niveau M_0 : Le monde réel.
- Niveau M_1: Les modèles représentant cette réalité.
- Niveau M_2 : Les méta-modèles permettant la définition de ces modèles.
- Niveau M_3 : Le méta-méta-modèle. Il est unique et méta-circulaire. Il est noté MOF [50].

2.2.2 Transformation de modèles

Dans la démarche IDM, le processus de développement des systèmes peut être vu comme un ensemble de transformations de modèles partiellement ordonné. Chaque transformation agit sur un modèle en entrée et produit un modèle en sortie. Pour réaliser cette translation, il faut bien sûr sur établir des règles de transformation. Classiquement, il suffit d'élaborer ces règles à partir des éléments du modèle source pour les transformer en éléments désirés du modèle cible. Cette approche pose un problème évident. La transformation n'est utile que pour un seul modèle. L'IDM et la méta-modélisation permettent d'établir des règles à un niveau d'abstraction plus élevé (niveau M_2 de l'OMG). Les règles sont donc définies pour toutes les instances du méta-modèle source. Pour ce faire, les correspondances entre les concepts des deux formalismes à ce niveau doivent être spécifiées. Elles sont exprimées sous une forme :

- Impérative : une logique impérative est souvent exprimée par un langage de programmation qui fait appel à des APIs pour manipuler directement les modèles. Elle offre un haut niveau de contrôle, ce qui donne une grande flexibilité à l'implémentation. Les langages impératifs sont appropriés pour les transformations nécessitant des sélections ou des itérations.
- Ou bien déclarative : La logique déclarative ne garantit aucun contrôle explicite. Les transformations sont définies par composition de règles décrites par leur pré- et post-conditions. Les pré-conditions définissent des motifs d'intérêt dans les modèles sources, alors que les post-conditions définissent le motif correspondant dans le modèle destination. Les langages déclaratifs sont compacts et les descriptions des transformations sont courtes et concises.

La génération automatique du modèle cible est réalisée par un moteur de transformation ou d'exécution (Fig.2.2).

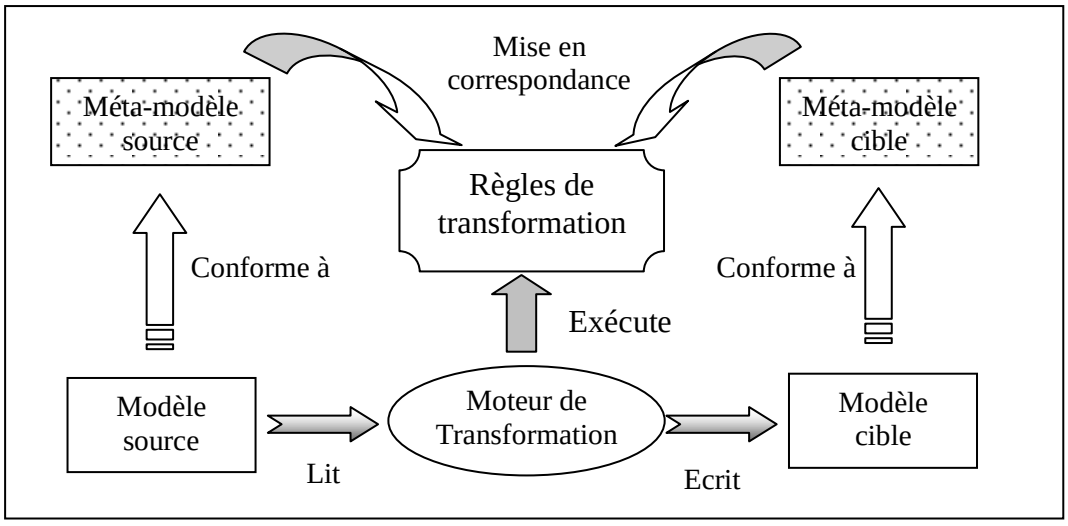


Figure 2.2: Concepts de base de la transformation de modèles

2.2.3 Approches de transformation

Les approches de transformation proposées dans la littérature varient beaucoup, surtout quant aux techniques de leur mise en œuvre. Czarnecki et al. [51] ont publié une classification basée sur plusieurs critères. Dans ce qui suit, donnons un aperçu.

Déjà, au plus haut niveau de la classification, on distingue deux types d'approches :

- Les transformations de modèles vers modèles : Ces transformations sont aujourd'hui moins maîtrisées. Cependant, elles ont beaucoup évolué au cours de ces dernières années, et plus particulièrement depuis l'apparition du MDA [52].
- Les transformations modèles vers code : Ces approches sont relativement matures.

Concernant l'homogénéité des modèles traités, on distingue deux types de transformations:

- Les transformations endogènes : Le modèle source et le modèle cible sont conformes au même méta-modèle.
- Les transformations exogènes : Le modèle source et le modèle cible sont conformes à des méta-modèle différents.

Un autre facteur important à prendre en considération dans la classification des approches de transformation des modèles concerne le niveau d'abstraction. On distingue :

- Les transformations horizontales : une transformation horizontale est une transformation où les modèles sources et cibles sont du même niveau d'abstraction.
- Les transformations verticales : les modèles impliqués sont de différents niveaux d'abstraction. Une transformation qui baisse le niveau d'abstraction est appelée un raffinement. Par contre, une transformation qui élève le niveau d'abstraction est appelée une abstraction.

Enfin, les techniques de transformations peuvent être classées en cinq catégories selon la technique de mise en œuvre et qui sont :

- Approches manipulant directement les modèles : Ces approches sont basées sur l'utilisation d'APIs pour manipuler directement la représentation interne des modèles. Elles sont en général implémentées comme un Framework orienté objet. La spécification des règles de transformation et leur ordonnancement restent à la charge du développeur. L'interface JMI (Java Metadata Interface) est souvent utilisée dans la mise en œuvre de ce type d'approches.

- Approches relationnelles : Pour ces approches, le principe de base consiste à établir une relation entre les éléments des modèles sources et cibles. Ces relations sont spécifiées à l'aide de contraintes. Elles sont purement déclaratives. La programmation logique est particulièrement adaptée pour ce type d'approches et la norme QVTRelational [53] est largement utilisée.

- Approches guidées par la structure : Dans ces approches, la transformation se réalise en deux phases : la première crée la structure hiérarchique du modèle cible et la seconde vient compléter le modèle en définissant les valeurs des attributs et des références. Comme exemples, citons OptimalJ [54].

- Approches basées sur la transformation de graphes : Les modèles et les méta-modèles associées possèdent souvent une représentation graphique. Ainsi, les techniques de réécriture de graphes peuvent être appliquées pour réaliser des transformations de graphes. Les traitements souhaités sont exprimés sous forme d'une grammaire de graphes. Cette catégorie d'approches est mise en ouvre par exemple dans : GReAT [55], AGG [56] et AToM^3 [57].

  Comme les modèles manipulés dans le cadre de cette thèse sont graphiques, nous avons opté pour cette approche. Ce paradigme sera détaillé dans la section 2.3.

- Approches hybrides : C'est des combinaisons des techniques citées précédemment. On peut notamment retrouver des approches utilisant à la fois des règles déclarative et impérative. ATLAS (ATL) [58], Kermeta[59] et ModTransf [60] sont des approches de cette catégorie.

2.2.4 Model-Driven Architecture (MDA)

Le MDA [52] est une démarche de développement à l'initiative de l'OMG (Object Management Group) rendue publique fin 2000. C'est une proposition à la fois d'une architecture et d'une approche de développement. L'idée de base est de séparer les spécifications fonctionnelles d'un système des spécifications de son implémentation. Pour cela, le MDA définit trois niveaux de modèles représentant les niveaux d'abstraction de l'application, le CIM, le PIM et le PSM.

- Le modèle CIM (Computational Independant Model) : C'est le modèle d'exigences. Il décrit les besoins fonctionnels de l'application indépendamment des détails liés à son implémentation. L'indépendance technique de ce modèle lui permet de garder tout son intérêt au cours du temps. Il n'est modifié que si les connaissances ou les besoins métier changent.

- Le modèle PIM (Platform-Independent Model) : C'est le modèle d'analyse et de conception de l'application. Son rôle est de donner une vision structurelle et dynamique de l'application, toujours indépendamment de la conception technique. En effet, la prise en compte très tardive des détails d'implémentation permet de maximiser la séparation des préoccupations entre la logique de l'application et les techniques d'implémentation. Par ailleurs, ce modèle doit contenir toutes les informations nécessaires pour qu'une génération automatique de code soit envisageable.
- Le modèle PSM (Platform-Specific Model) : C'est le modèle de code. Il décrit l'implémentation d'une application sur une plateforme particulière. Ce modèle sert à faciliter la génération de code. En effet, MDA considère que le code d'une application peut être facilement obtenu à partir d'un modèle comportant à la fois la spécification fonctionnelle et les informations de la plate-forme d'exécution.

La mise en œuvre du MDA est entièrement basée sur les modèles et leurs transformations. L'architecture de base est schématisée comme l'indique la figure Fig.2.3.

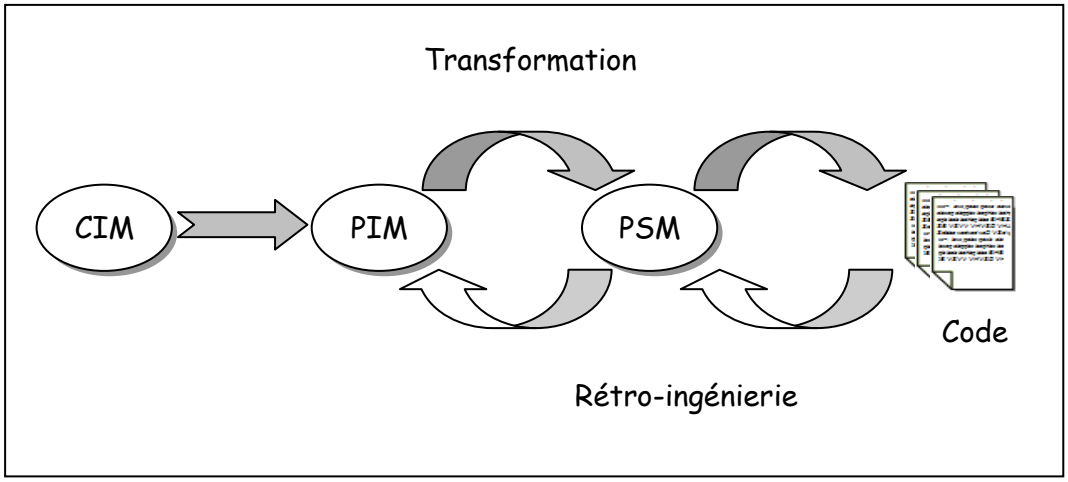


Figure 2.3: Les transformations de modèles dans l'approche MDA

Aujourd'hui, plusieurs outils respectent cette approche. Parmi les plus récents, nous pouvons citer :

- AndroMDA : une plate-forme de génération de code extensible qui transforme des modèles UML en composants qui peuvent êtres déployés sur une plate-forme donnée (JEE, Spring, .NET, etc.)
- OAW : l'acronyme d'OpenArchitectureWare, qui est une plateforme modulaire de génération développée en Java. Elle supporte les modèles EMF, UML2, XML et même des modèles exprimés en JavaBeans.

2.3 Transformation de Graphes

L'approche transformation de graphes est inspirée des approches classiques de réécriture des termes telles que les grammaires de Chomsky utilisées dans la théorie des langages formels. La recherche dans ce domaine remonte aux années 1970. Elle a connu une popularité immense qui n'a cessé de croitre jusqu'à aujourd'hui. Les graphes et leurs transformations ont été étudiés pour être appliqués dans de nombreux domaines de l'informatique. Une présentation détaillée de leurs applications est donnée dans [61].

Dans ce domaine, la manipulation des modèles est effectuée par des traitements locaux. Cela permet aux concepteurs de limiter le champ de concentration. Par conséquent, les grammaires de graphes constituent le moyen le plus adapté pour l'implémentation.

2.3.1 Système de réécriture de graphes

Une grammaire de graphes est un ensemble de règles de transformations. Chaque règle est constituée de deux parties : le *LHS* (Left Hand Side) et le *RHS* (Right Hand Side). La partie gauche est destinée à être mise en concordance avec les parties similaires du graphe global appelé *host-graph*. La partie droite de la règle, le RHS, décrit les modifications à réaliser. Afin de guider le processus de transformation, les règles sont munies de priorités attribuées par l'utilisateur.

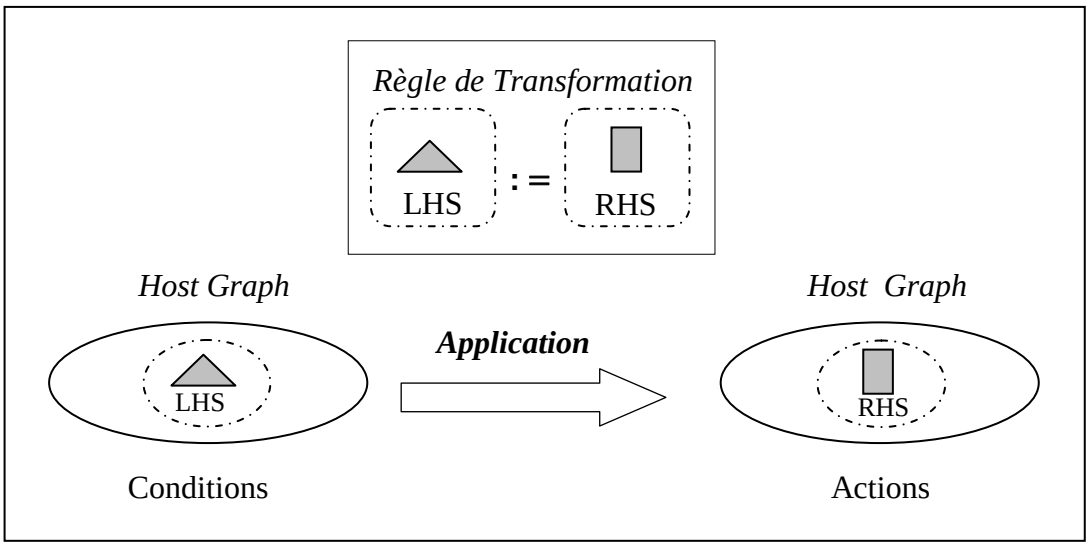


Figure 2.4: Principe de l'application d'une règle de transformation

En plus, chaque règle est dotée de conditions supplémentaires d'activation ainsi que des actions à effectuer [62].

L'exécution d'une grammaire de graphes est un traitement itératif. A chaque itération, le système de réécriture doit :

- Identifier la règle la plus prioritaire, c'est-à-dire la règle pour la quelle une occurrence du LHS est localisée dans le host graph.
- Evaluer les conditions d'application de cette règle.
- Si les conditions sont vérifiées :
    - Retirer l'occurrence repérée du host graph ainsi que les arcs pendillés.
    - Coller le RHS dans le host graph.

Ainsi, les règles sont appliquées sur le host graph dans l'ordre spécifié jusqu'à ce qu'aucune règle ne puisse être applicable.

2.3.2 Domaines d'application

L'approche transformations de graphes est très adaptée pour une large gamme d'applications. Parmi les plus prometteuses, on peut citer sans être exhaustif:

La génération de code :

L'exemple type est l'approche MDA. A partir des modèles initiaux, cette démarche permet la génération automatique d'applications exécutables sur des plateformes différentes. Cela se traduit par une réduction importante des efforts de programmation inhérents à ce type de déploiement. A titre d'exemple, Rhapsody [63] est un outil basé sur l'approche transformation de graphes dédié à l'ingénierie des systèmes embarqués. Il propose un générateur de code spécifique pour chacun des quatre langages de programmation : C, C++, Ada et Java.

La simulation des modèles

Cette activité consiste en une animation de modèles dynamiques immédiatement après leurs conceptions. Il s'agit de les debugger afin de vérifier le fonctionnel ainsi spécifié en leurs donnant vie. L'avantage d'une telle approche est de pouvoir intervenir le plus tôt possible dans le cycle de développement en cas d'erreurs. Ainsi, à l'inverse des solutions s'appuyant sur la génération de code à partir des modèles, la simulation directe de modèles est une solution générique ne mettant en œuvre que des modèles et manipulant ceux-ci par transformation de graphes. C'est exactement le cas avec l'outil smartQVT [64].

La vérification des modèles

La vérification des modèles est une tâche primordiale dans le processus de développement des systèmes. Les solutions les plus intéressantes sont celles basées sur le Model-Checking. Cette technique permet une exploitation globale de l'espace d'états, mais nécessite une spécification des modèles initiaux dans des formalismes formels. A titre d'exemple, les propriétés qualitatives d'un système comme la détection d'interblocage peuvent être analysées à l'aide des réseaux de Pétri colorés, en utilisant l'environnement CPN-AMI [65]. Dans ce domaine, l'approche transformation

de graphes est largement utilisée comme un moyen très adapté pour réaliser la translation nécessaire.

La synchronisation des modèles

Au cours du cycle de développement des systèmes, les modèles établis sont constamment mis à jour pour plusieurs raisons. Ils peuvent être étendus par l'ajout de nouvelles fonctionnalités, comme ils peuvent être corrigés en cas d'erreurs détectées. Dans ce contexte, la répercussion automatique des modifications effectuées sur tous les diagrammes impliqués se pose comme un problème majeur. Grâce à l'introduction des règles bidirectionnelles, l'approche transformation de graphes offre des moyens très efficaces pour répondre à cette difficulté. Plusieurs environnements ont été développés dans ce sens. Citons à titre d'exemple l'outil TGG [66].

2.3.3 Outils d'implémentation

Il existe plusieurs outils implantant les systèmes de réécriture de graphes. Les plus intéressants sont :

- **Fujaba** [67] : un outil très puissant utilisé principalement pour la génération de code Java à partir de diagrammes UML. Ces dernières années, il est devenu une référence dans plusieurs axes de recherche. Particulièrement, il est très utilisé dans la modélisation et la simulation des systèmes mécaniques et électriques.
- **AGG** [56] : un environnement développé à l'université technique de Berlin. C'est l'un des outils de transformation de graphes les plus cités dans la littérature. Il est implémenté en langage Java. Il est muni d'une interface permettant de spécifier graphiquement les règles de transformation. Il permet des simulations pas à pas ainsi que quelques vérifications élémentaires. Les graphes manipulés sont orientés. Les nœuds et les arcs peuvent être étiquetés par des objets Java.
- **TGG** [66] : un environnement spécifique aux transformations bidirectionnelles. Sa puissance réside dans le fait que les mêmes règles permettant d'effectuer des translations dans les deux sens. Il est très utile pour assurer la synchronisation et le maintien de correspondance entre les deux modèles source et destination.
- **GReAT** [55] : un outil permettant la définition des transformations unidirectionnelles de plusieurs modèles sources vers plusieurs modèles cibles. Il est basé sur une notation graphique pour la spécification des règles de transformations, mais les expressions d'initialisation des attributs et les conditions d'application sont éditées textuellement. Les modèles manipulés doivent être conformes à leurs méta-modèles. Ces derniers peuvent être créés avec l'outil GME.

- **Viatra** [68] : un plugin Eclipse développé en 2005 à l'université de Budapest. Le logiciel utilise le langage VPM pour la modélisation. Pour définir les règles de transformations, l'utilisateur peut utiliser des motifs récursifs et des motifs de négations représentant les conditions d'application négatives. L'ordonnancement dans l'application des règles est basé sur une machine à états abstraite.

- **Eclipse Modeling Galileo** [69] : un environnement spécifique aux transformations de graphes intégré à la plate forme éclipse. Le framework principal est celui utilisé pour la modélisation. Il est appelé Eclipse Modeling Framework (EMF). Les modèles y sont décrits en XML, mais peuvent être spécifiés dans des documents UML/SysML. L'interopérabilité avec d'autres outils basés sur Eclipse est l'un des points forts de ce framework. Un de ces outils appelé Graphical Modeling Framework (GMF) permet le développement d'éditeur graphique de modèles. Les transformations de modèles sont exprimées en langage ATL (Atlas Transforming Language).

- **Progres** [70]**:** un environnement de transformation de graphes à nœuds et arcs étiquetés. La syntaxe de Progres est mi-textuelle, mi-graphique. Chaque opération de manipulation d'un graphe, est composée d'étapes basiques de recherche de motifs et de transformation des sous-graphes correspondants. Bien que basée principalement sur des règles, la programmation avec Progres peut aussi se faire de manière impérative. Les transformations sont atomiques et préservent la cohérence du graphe manipulé. Le programmeur peut faire des choix non-déterministes avec la possibilité d'initier un retour arrière plus tard si nécessaire.

- **AToM^3** [57] : un outil de modélisation multi-paradigme développé par le laboratoire MSDL de l'université McGill Montréal au Canada. AToM^3 est un outil visuel dédié à la transformation de graphes, implémenté en langage Python [71]. Il possède une couche de Méta-Modélisation permettant une spécification syntaxique et graphique des formalismes utilisés. Les manipulations de modèles souhaitées sont définies sous forme de grammaires de graphes.

  Dans le cadre de cette thèse, nous avons opté pour l'utilisation de cet environnement pour l'implémentation des grammaires de graphes proposées. Il sera présenté dans la section suivante.

2.4 AToM^3

Pour une bonne introduction de cet environnement, nous proposons de l'illustrer à travers une application très simple. Il s'agit de transformer des processus décrits dans le formalisme d'automates à états finis (FSA) simplifié en leurs réseaux de Pétri (Rdp) équivalents. Dans ce qui suit, nous présentons toutes les étapes nécessaires en donnant les détails utiles.

2.4.1 La Méta-Modélisation

Le méta-formalisme utilisé est le diagramme Entités-Associations. Un méta-modèle est spécifié par la fourniture de deux syntaxes. D'une part, une syntaxe abstraite et formelle. Elle sert à définir les entités, les relations et toutes les contraintes associées. Les entités et les relations sont définies en donnant tous les attributs nécessaires, alors que les contraintes sont exprimées textuellement en langage OCL [72]. D'autre part, une syntaxe graphique permettant d'ajuster l'apparence graphique selon les notations appropriées. Une fois le méta-modèle d'un formalisme donné est défini, AToM^3 génère un environnement interactif pour manipuler ses entités. Ainsi, la conformité des modèles créés vis-à-vis la syntaxe spécifiée est assurée automatiquement.

Pour notre exemple, les méta-modèles nécessaires sont :

FSM méta-modèle: il est composé de:

- **Entité FSA-State :** Chaque état dispose d'un attribut *Name* pour l'identifier.
- **Relation FSA-Transition :** Chaque transition est dotée de deux attributs : *Input* et *Output.*

Rdp méta-modèle: il est composé de :

- **Entités :**
  - **Rdp-Place**: Chaque place est identifiée en utilisant un attribut *Name*. Le nombre de jetons est spécifié en utilisant un autre attribut *Nbr-Tokens*.
  - **Rdp-Transition** : Chaque transition est dotée de trois attributs : *Name*, *Input* et *Output*.
- **Relations :**
  - **Rdp-OutputArc :** Cette relation est utilisée pour représenter les arcs sortants des places. Elle est dotée d'un attribut *Weight*.
  - **Rdp-InputArc :** Cette relation est utilisée pour représenter les arcs entrants des places. Elle est dotée d'un attribut *Weight*.

Dans AToM^3, ces deux méta-modèles sont spécifiés en se basant sur le formalisme **ClassDiagramsV3_META** intégré dans cet environnement.

A titre d'exemple, la Figure Fig.2.5 montre le méta-modèle des automates à états finis.

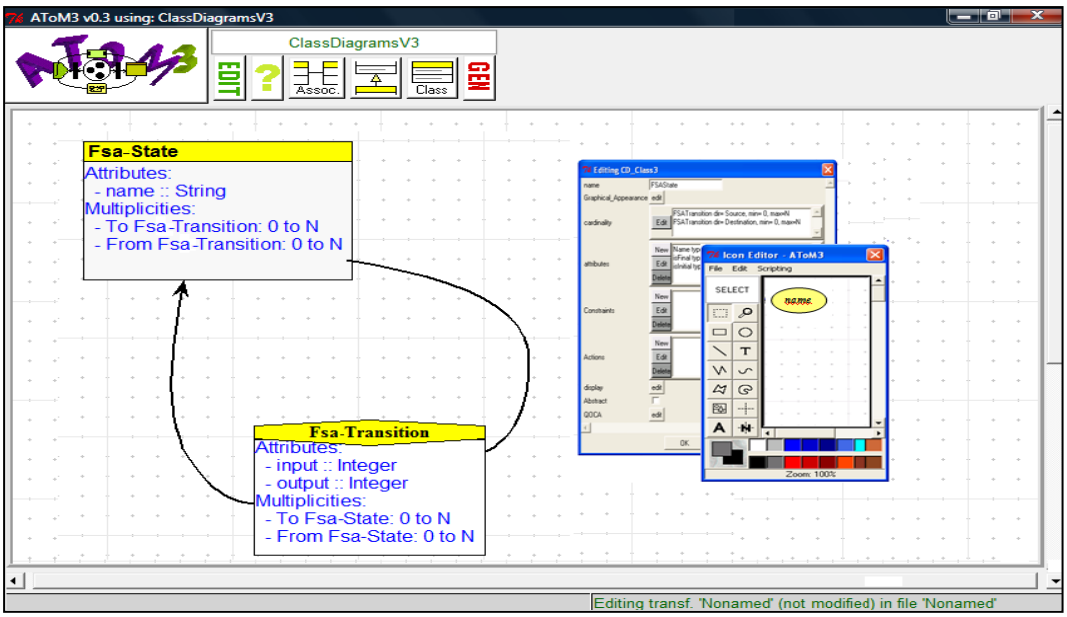


Figure 2.5: Méta-Modèle des automates à états finis

L'apparence graphique des états et des transitions est ajustée selon les notations appropriées en utilisant un utilitaire intégré. La Figure 2.6 présente l'éditeur graphique associé.

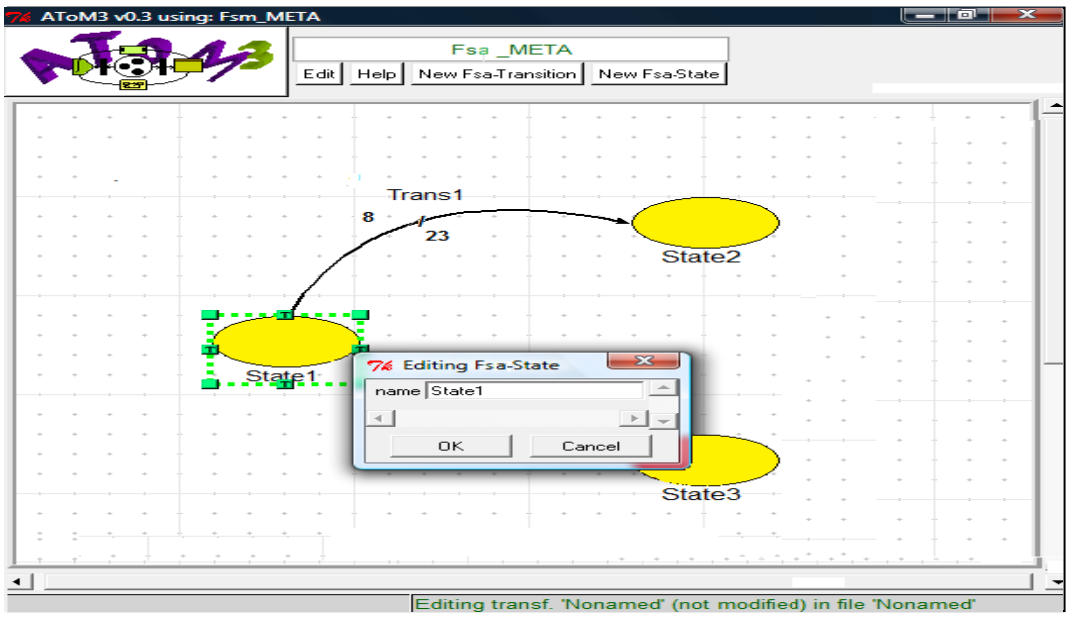


Figure 2.6: Editeur graphique généré pour les automates à états finis

De même pour le formalisme Rdp, l'éditeur graphique est généré en suivant la même procédure.

2.4.2 Les grammaires de graphes

Dans AToM^3, la manipulation des modèles est basée sur la réécriture de graphes. Les traitements souhaités sont exprimés sous forme de grammaires. Chaque grammaire est caractérisée par :

- Une action initiale : Elle sert à spécifier toutes les opérations à exécuter avant l'application des règles.
- Des règles de transformations : Chaque règle est constituée de deux parties, le *LHS* et le *RHS*. En plus, elle est munie d'une priorité et des conditions d'activation ainsi que d'une partie actions spécifiant les traitements à réaliser.
- Une action finale : Elle sert à spécifier toutes les opérations à exécuter après l'application des règles.
- Un ensemble d'attributs auxiliaires : Des fois, pour implémenter un traitement spécifique, les entités manipulées doivent être enrichies par des informations supplémentaires. Elles ne figurent pas au niveau du méta-modèle. Dans AToM^3, elles sont déclarées comme étant des attributs temporaires associés aux entités concernées.
- Un ensemble de variables globales : Les variables globales sont des informations utiles et accessibles au niveau des règles. Elles ne sont associées à aucune entité.

Pour chaque règle, en cas de correspondance du LHS dans le *host graph*, l'évaluation des conditions d'application est basée sur les valeurs des attributs associés aux entités et aux relations. Les valeurs d'attributs nécessaires à cette vérification doivent être ajustées en choisissant l'option (<*Specified*>). Le reste d'attributs sera ignoré. Leurs valeurs sont définies comme étant (<Any>). L'application de cette règle consiste à substituer le LHS par le RHS. Pour les éléments maintenus, les valeurs d'attributs associés peuvent être copiées (<Copied>) ou bien spécifiées (<*Specified*>). Par contre, celles des éléments nouvellement créés doivent être calculées (<*Specified*>). Pour faciliter l'édition du code spécifiant la partie actions, les éléments du LHS et du RHS sont étiquetés (numéros) automatiquement par AToM^3.

Pour notre application, la génération du réseau de pétri commence par la création des places. Pour chaque état du modèle FSA, un nouveau nœud du modèle Rdp est associé. Pour ce faire, nous proposons d'utiliser un attribut temporaire *Translated.* Il est utilisé pour localiser les états déjà traités. Ensuite, on passe à la création des transitions du Rdp. Chaque transition du formalisme FSA est translatée en :

- Une transition Rdp.
- Un arc sortant.
- Et un arc entrant.

Les attributs associés à tous les éléments créés sont ajustés à leurs valeurs appropriées. Enfin, cette transformation se termine la suppression des états du modèle FSA.

Ce traitement est réalisé en utilisant une grammaire de graphes constituée de trois règles(Fig.2.5) et qui sont :

Règle N°1. Create-RdpPlaces : Cette règle est appliquée pour créer les places du modèle Rdp. A ce niveau, on garde les états du modèle FSA afin de préserver leurs relations.

Règle N°2. Create-RdpTransitions : Cette règle est appliquée pour créer les transitions ainsi que les arcs entrant et sortant associés du modèle Rdp.

Règle N°3. Delete-FSA-States : Cette règle est appliquée pour supprimer les états du modèle FSA.

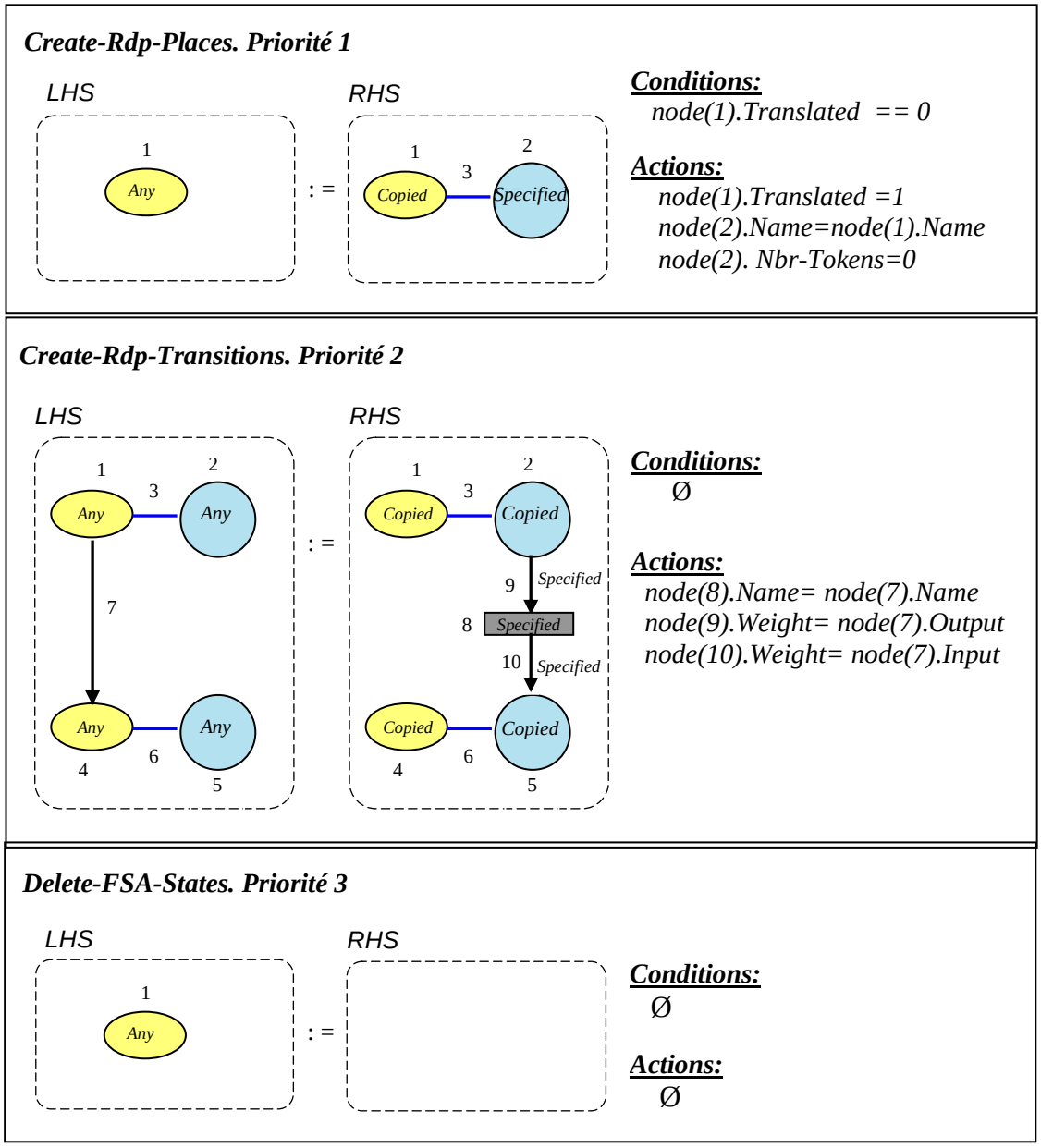


Figure.2.7: Grammaire de graphes de transformation d'un FSA à un Rdp équivalent

Dans AToM[3], les grammaires de graphes sont manipulées en utilisant un autre utilitaire graphique intégré. Il permet d'implantation des règles ainsi que la spécification de l'action initiale et l'action finale. A titre d'exemple, la figure Fig.2.3 présente la définition de la grammaire proposée.

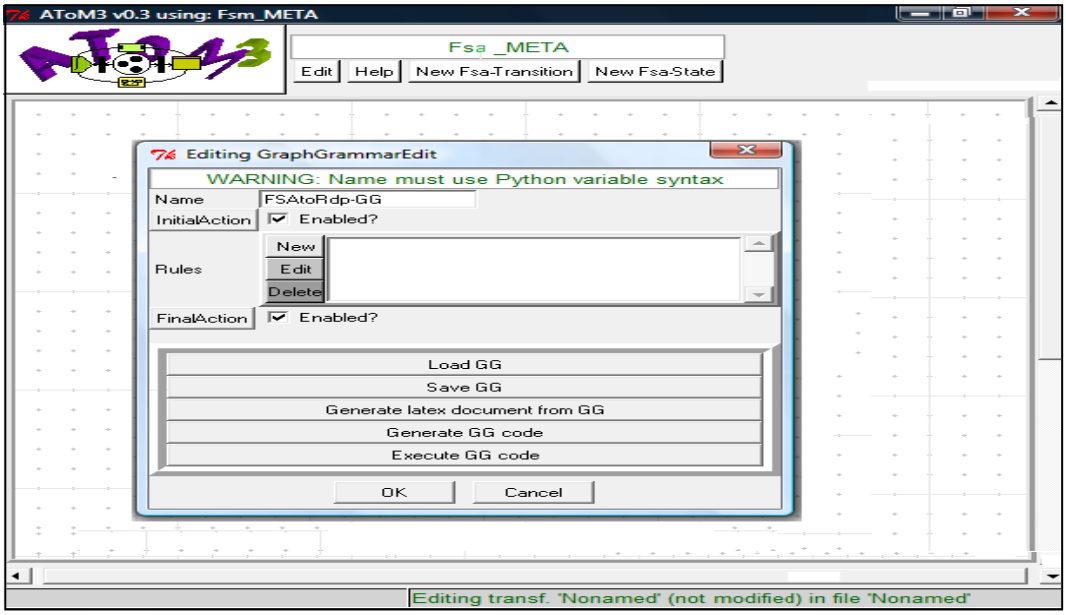


Figure 2.8: Définition des Grammaires de Graphes dans AToM [3]

La figure Fig.2.8 présente la spécification de la première règle.

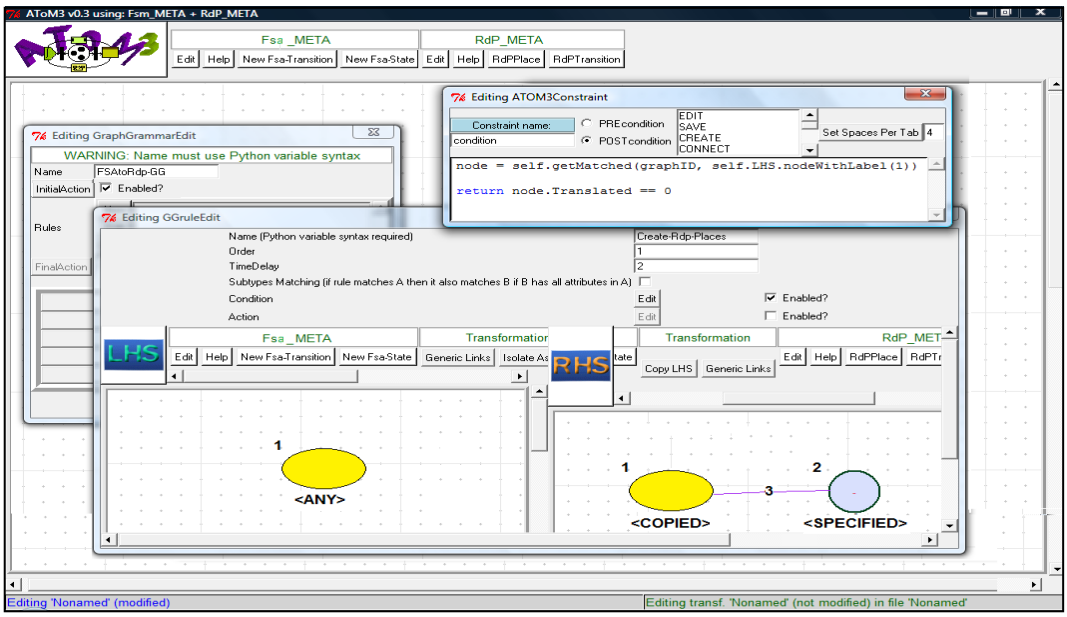


Figure 2.9: Définition d'une règle de transformation dans AToM [3]

2.4.3 Exemple illustratif

Considérons l'exemple de l'automate d'états finis présenté dans la figure suivante (Fig.5.1).

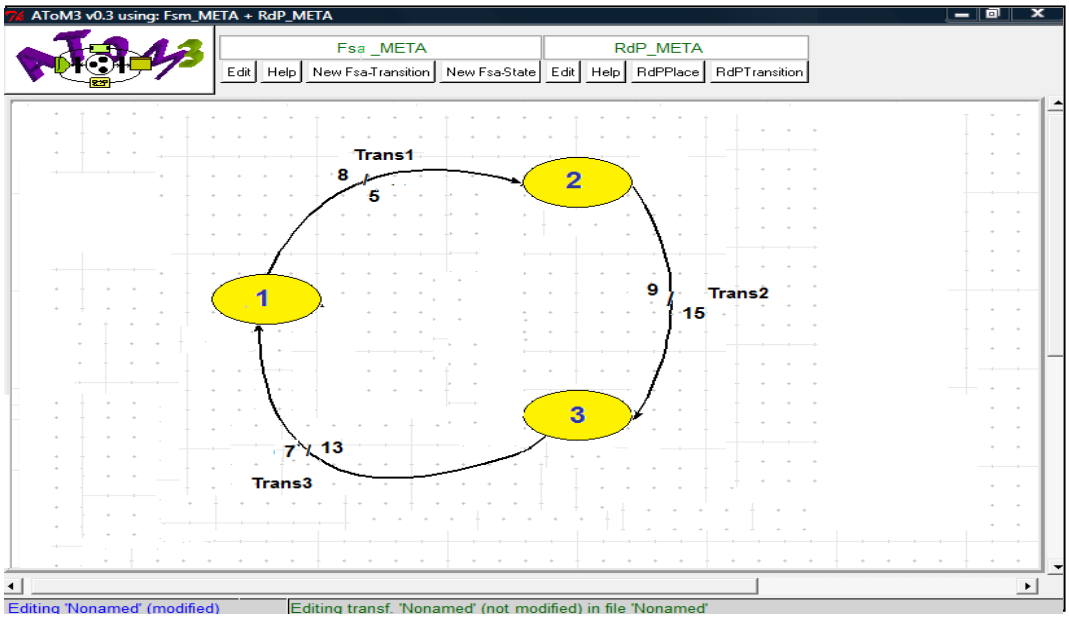


Figure 2.10: Exemple illustratif, Modèle source(FSA)

L'application de la grammaire proposée génère automatiquement le réseau de pétri présenté Figure Fig.3.13.

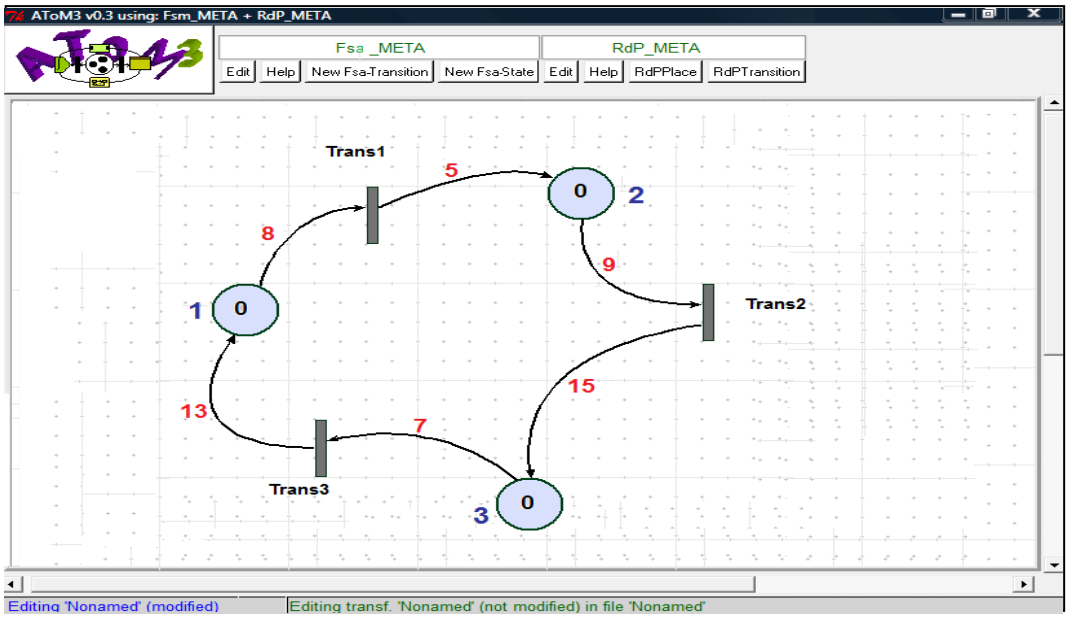


Figure 2.11: Exemple illustratif, Modèle cible (Rdp)

2.5 Conclusion

Dans ce chapitre, nous avons abordé les principaux concepts de l'IDM. C'est une ingénierie mettant à disposition des outils, concepts et langages pour créer et transformer des modèles. La définition des langages de spécification est basée sur la méta-modélisation, alors que la réalisation des traitements souhaités est implantée par des transformations de modèles.

Dans ce contexte, l'approche transformation de graphes joue un rôle fondamental. Elle est inspirée principalement des approches classiques de réécriture des termes. La partie gauche (LHS) et la partie droite (RHS) de chaque règle de transformations est un sous-graphe. A l'exécution, le système de réécriture applique itérativement la règle la plus prioritaire et pour laquelle les conditions d'activations sont vérifiées. A chaque itération, le LHS est remplacé par le RHS dans le graphe courant.

En conclusion, nous avons présenté l'outil AToM^3. C'est un environnement largement utilisé dans ce domaine. Il supporte la méta-modélisation et offre des mécanismes très puissants facilitant l'implémentation des transformations complexes tels que les variables globales et les attributs temporaires.

Les concepts abordés dans ce chapitre constituent un background nécessaire pour la réalisation des travaux de cette thèse présentés dans chapitres suivants.

Chapitre 03

Génération Automatique des Produits Structurellement Valides : Approche Basée sur la Recherche des Configurations Correctes

3.1 Introduction

Comme présenté au premier chapitre, l'analyse des diagrammes FD est une tâche primordiale dans l'ingénierie des lignes de produits logiciels. Une des préoccupations majeures consiste à déterminer l'ensemble des variantes structurellement valides. En pratique, avec un grand nombre de caractéristiques et de contraintes associées, réaliser cette tâche manuellement est quasi impossible. En plus, ces modèles peuvent évoluer au fil du temps pour des fins d'optimisations ou d'extensions. Ainsi, les concepteurs doivent refaire cet immense calcul pour chaque modification. Par conséquent, le développement d'environnements automatiques est d'une grande nécessité.

Pour répondre à ce besoin, nous proposons dans ce chapitre une approche basée sur la technique transformation de graphes. Elle consiste à chercher les configurations vérifiant toutes les dépendances spécifiées dans le diagramme FD [73]. On commence d'abord par présenter la solution naïve traitant toutes les combinaisons possibles des caractéristiques. Ensuite, on propose une version optimisée en introduisant le BackTracking dans le processus de recherche. Enfin, le chapitre se termine par un exemple illustratif et une conclusion.

3.2 Diagramme de caractéristiques

Ce formalisme a été introduit au début des années 1990 par Kang et *al.* [12]. Il permet de modéliser la variabilité d'une ligne de produits au niveau de ses fonctionnalités. Typiquement, il s'agit d'une arborescence dans laquelle :

- La racine représente le concept modélisé.
- Les nœuds spécifient les caractéristiques.
- Les arêtes définissent les relations.

La Figure suivante (Fig.3.1) présente un modèle de caractéristiques simplifié d'une ligne de produits de téléphones mobiles (inspiré de [45]).

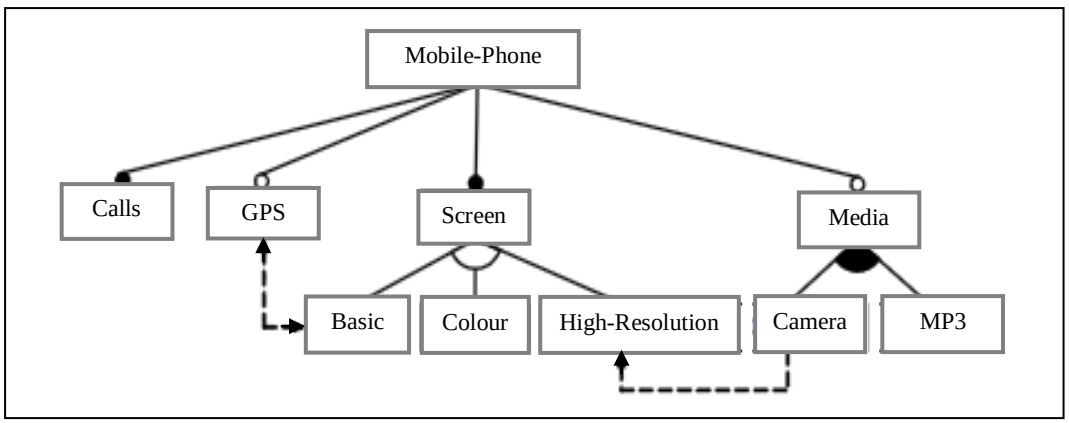


Figure.3.1: Diagramme de caractéristiques

Dans le cadre ce cette thèse, nous nous sommes intéressés uniquement par les diagrammes de caractéristiques de base. Les relations paternelles possibles sont:

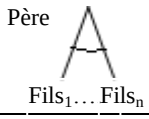
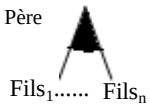
Type	Notation	Sémantique
Obligatoire		Si le père est sélectionné, le fils doit être sélectionné.
Optionnel		Si le père est sélectionné, le fils peut être sélectionné mais pas nécessairement.
Alternative (XOR)		Si le père est sélectionné, exactement un seul fils doit être sélectionné.
OR		Si le père est sélectionné, au moins un fils doit être sélectionné.

Figure 3.2: Table des relations paternelles

En plus, deux types de contraintes sont définis:

Implication		La sélection de la caractéristique source implique la sélection de la destination
Exclusion		Les deux caractéristiques source et destination ne peuvent pas être sélectionnées en même temps.

Figure 3.3: Table des contraintes

Pour plus de clarté sur l'analyse structurelle, prenons le diagramme FD précédent (Fig.3.1) et considérons les deux configurations suivantes:

- P_1: *Mobile-Phone, GPS, Screen, Basic, Media*

  Ce produit est incorrect du fait que:

  - La caractéristique *Calls* est spécifiée comme étant obligatoire, mais elle n'est pas sélectionnée.
  - les caractéristiques *GPS* et *Basic* sont liées par une relation d'exclusion dans le diagramme FD, alors que toutes les deux sont sélectionnées.
  - La caractéristique *Media* doit avoir au moins un fils sélectionné et ce n'est pas le cas pour ce produit.

- P_2: *Mobile-Phone, Calls, GPS, Screen, High-resolution, Media, Camera, MP3*

  Ce produit est valide. Toutes les dépendances spécifiées dans le diagramme FD sont respectées.

3.3 Principe de la recherche des configurations valides

Prenons le cas général : un FD avec n caractéristiques. Par définition, chaque produit est identifié par l'ensemble des caractéristiques sélectionnées. Ainsi, il peut être représenté par un tableau binaire de taille n:

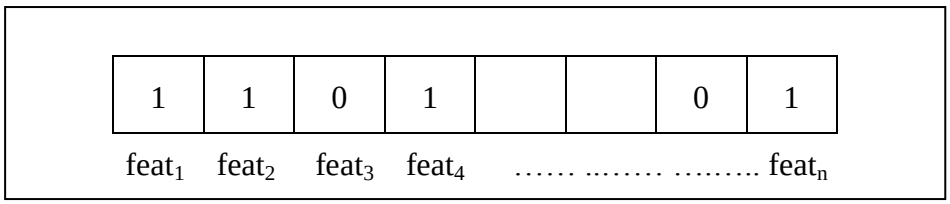


Figure.3.4: Représentation binaire des configurations

Les valeurs possibles sont :

- 1 : si la caractéristique correspondante est sélectionnée.
- 0 : le cas contraire.

Pour trouver tous les produits structurellement valides, on propose de :

- Générer toutes les configurations possibles du tableau binaire.
  - Pour chaque configuration, le tableau binaire est projeté sur le diagramme FD afin de vérifier toutes les dépendances. En cas de validité, le produit correspondant sera édité dans un fichier texte. (Fig.3.5).

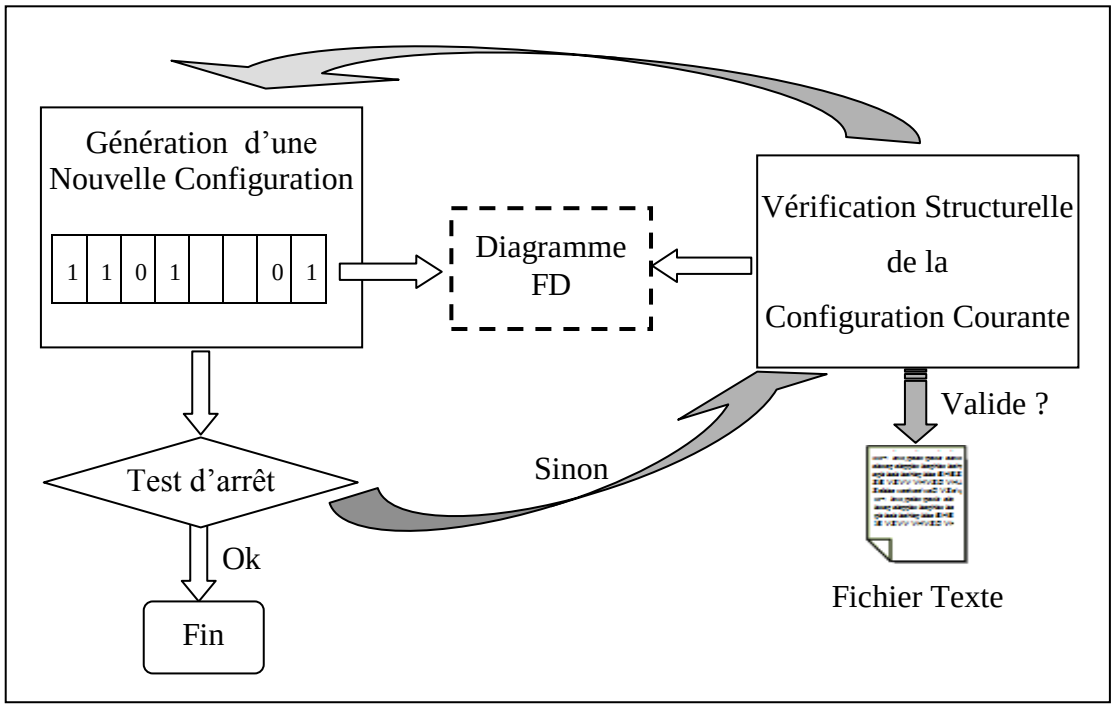


Figure.3.5: Principe de recherche des configurations valides

Pour réaliser ce traitement, nous proposons l'algorithme suivant:

Calculer le nombre de caractéristiques: N.
Conf = 0
Tant que (Conf < 2^N) faire

- Générer la représentation binaire de Conf : Le résultat est transféré dans le tableau binaire afin de pouvoir l'exploiter.

- Projeter ce tableau sur le diagramme D-Tree en sélectionnant les caractéristiques correspondantes.
- Vérifier la validité de cette configuration.
- Si (valide) alors
    - Editer ce produit (caractéristiques sélectionnées) dans un fichier texte.
- Conf = Conf + 1

Figure 3.6. Algorithme de recherche des configurations valides

Afin d'automatiser ce processus, les traitements réalisés doivent être exprimés en terme de règles de transformation. Les grammaires de graphes proposées sont présentées dans la section suivante.

3.4 Automatisation de l'algorithme de recherche

Après une analyse du problème, nous avons constaté qu'il existe plusieurs traitements similaires pour les différents types des relations du modèle FD. Or, dans l'approche transformation de graphes, les règles sont exprimées en donnant leur partie gauche ainsi que leur partie droite et qui sont toutes les deux graphiques. De ce fait, l'utilisation directe du diagramme FD exige la spécification d'une règle pour chaque type de relation. Pour remédier à ce problème, nous préférons d'abord translater le diagramme FD en un modèle homogène noté D-Tree. C'est un arbre décoré dans lequel toutes les relations ont la même apparence graphique. Le type est spécifié en utilisant un attribut supplémentaire. Afin de faciliter les traitements ultérieurs, à ce niveau chaque nœud sera doté d'un attribut indice spécifiant sa position dans le tableau binaire. Ensuite, on procède à la génération des configurations valides en appliquant l'algorithme précédent.

Donc, nous avons deux grammaires de graphes à réaliser (Fig.3.7) :

- 1ère GG : utilisée pour générer le modèle D-Tree à partir du diagramme FD.
- 2ème GG : utilisée pour générer les configurations valides à partir du modèle D-Tree.

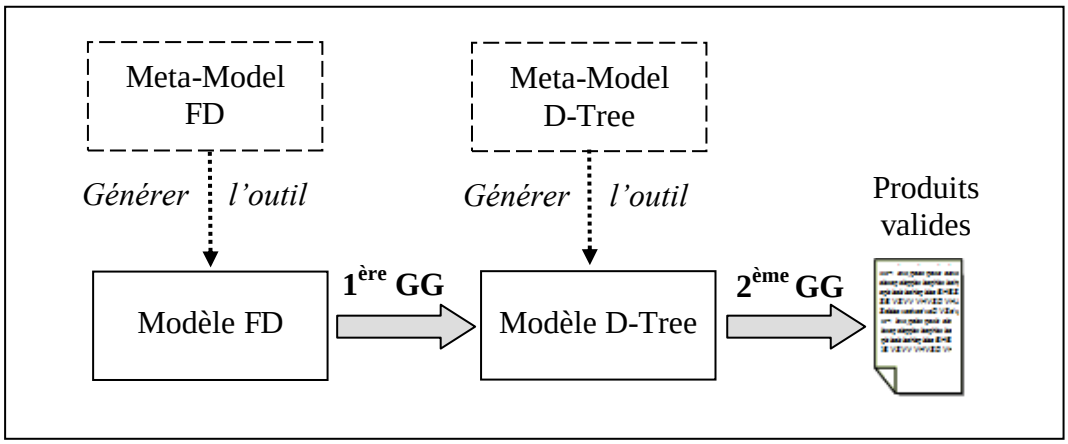


Figure.3.7: Grammaires de graphes proposées pour la recherche des configurations valides

Dans ce qui suit, nous présentons d'abord les Méta-Modèles utilisés. Ensuite, nous introduisons les règles de chaque grammaire.

3.4.1 Méta-Modélisation

3.4.1.1. FD Méta-Modèle: il est composé de:

- **Entité *Feature*:** Chaque caractéristique dispose de deux attributs:
  - *Name* : c'est son identifiant.
  - *isRoot* : un attribut booléen utilisé pour spécifier le nœud racine.
- **Relations *Mandatory, Optional, OR, XOR, Requires*** et ***Excludes*** : Toutes ces relations n'ont pas d'attributs. Leurs apparences graphiques sont ajustées selon les notations appropriées (Fig.3.2 et Fig.3.3).

3.4.1.2. D-Tree Méta-Modèle: il est composé de :

- **Entité *Node***: Cette entité possède trois attributs:
  - *Name* et *isRoot*.
  - *Index:* utilisé pour spécifier sa case dans le tableau binaire.
- **Relation *GenericLink***: Cette association représente tous les liens possibles entre les nœuds dans le modèle D-Tree. Graphiquement, ces liens ont la même apparence. Le type est spécifié en utilisant un attribut noté *RelationType*.

3.4.2 Grammaires de graphes

3.4.2.1. Génération du modèle D-Tree: 1ère GG

Cette translation commence par la création des nœuds du modèle D-Tree. Pour chaque caractéristique du diagramme FD, un nouveau nœud du modèle D-Tree est généré. Les attributs *Name* et *isRoot* sont copiés avec les mêmes valeurs, alors que l'attribut *Index* est ajusté à la valeur

d'un entier *Index-Max*. C'est une variable globale initialisée à zéro et incrémentée pour chaque nœud traité. Il sera utilisé dans la deuxième grammaire. Ensuite, on passe à la création des liens D-Tree. Les relations du modèle FD sont transformées en liens équivalents de type *GenericLink*. L'attribut *RelationType* est spécifié en fonction du type de la relation source. Enfin, ce traitement se termine par la suppression des caractéristiques du modèle FD.

Dans cette grammaire de graphes, nous avons proposé huit règles. Elles sont présentées dans les figures Fig.3.8 et Fig.3.9 classées dans l'ordre ascendant de leurs priorités.

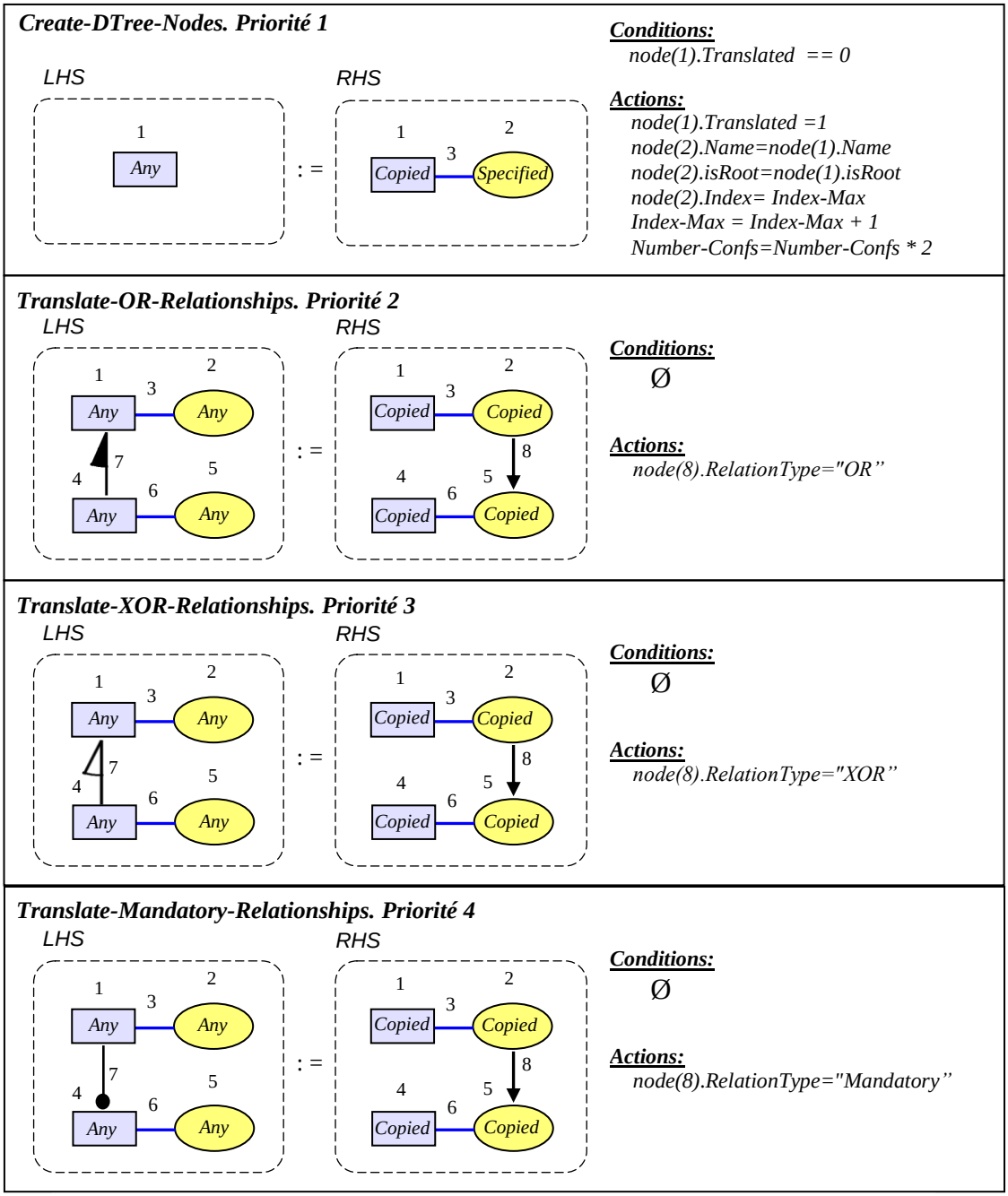


Figure.3.8: 1ère GG, les règles N°1-4

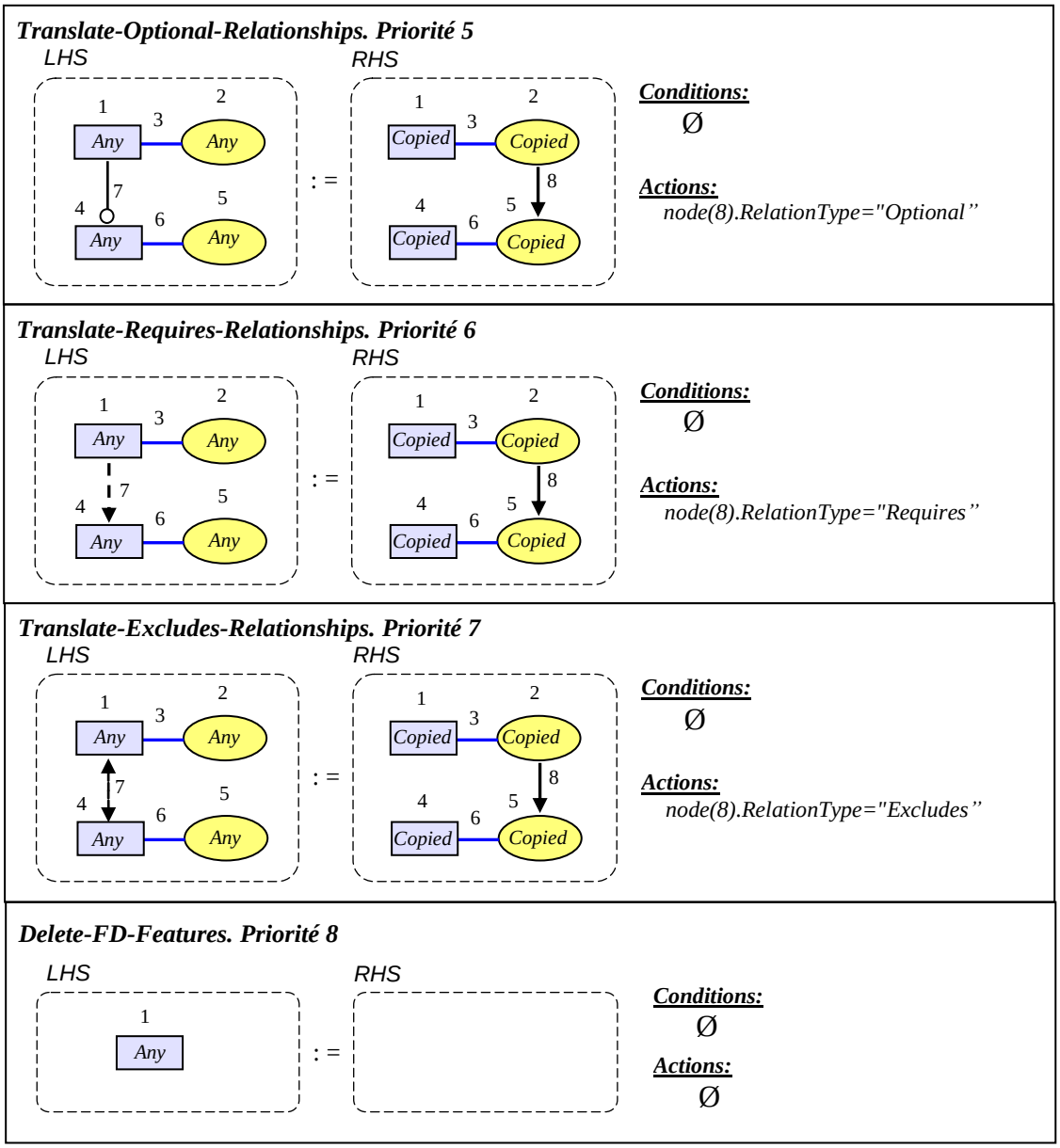


Figure.3.9: 1ère GG, les règles N°5-8

- *Number-Confs* : une variable globale utilisée pour compter le nombre de configurations possibles. Elle est initialisée à 1.

3.4.2.2. Génération des produits valides: 2ème GG

Une fois le modèle D-Tree généré, on passe à la recherche de configurations correctes. La grammaire de graphes proposée est constituée de deux parties :

- Les règles de la première partie sont utilisées principalement pour générer toutes les configurations possibles.
- Les règles de la deuxième partie sont utilisées uniquement pour vérifier la configuration courante. En cas de validité, le produit correspondant sera édité dans le fichier texte.

Partie$_1$: Il s'agit d'un processus répétitif de 2^n - 1 itérations. Ce traitement est réalisé par trois règles (Fig.3.10) en se basant sur les variables globales suivantes :

- *Conf* : un entier spécifiant la valeur entière de la configuration courante. Il est initialisé à 0.
- *Current-Product*: un tableau binaire.
- *Continue*: un booléen utilisé pour contrôler la répétition.
- *Current-Index* : un entier utilisé pour parcourir le tableau *Current-Product*.
- *NewConfiguration-Generated* : un booléen utilisé pour déclencher la génération d'une nouvelle configuration.
- *isStillValid*: un booléen. A ce niveau, il est utilisé pour désactiver les règles de la deuxième partie.

Pour la manipulation des nœuds du modèle D-Tree, les attributs temporaires suivants sont ajoutés :

- *isSelected:* un booléen utilisé pour spécifier les caractéristiques sélectionnées.
- *isReinitialized* : un booléen utilisée pour identifier les nœuds à réinitialiser.

Generating-NewConfiguration. Priorité 25

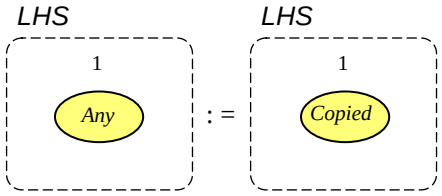
Conditions:

```
NewConfiguration-Generated  == 0
```

Actions:

```
Conf  =  Conf  + 1
CurrentProduct = [ ]
Cpt = 0
Nbr= Conf
While ( Nbr !=  0 ) :
        CurrentProduct.append ( Nbr % 2 )
        Nbr = Nbr // 2
        Cpt = Cpt + 1
Reste = Cpt-Features - Cpt
If  ( reste  >  0 ) :
         For i in range ( Reste ):
                  CurrentProduct.append ( 0 )
CurrentProduct.reverse()
Continue = False
 isStillValid = False
Current-Index = 0
NewConfiguration-Generated   = 1
```

Mapping-NewConfiguration. Priorité 26



Conditions:

```
node(1).Index ==  Current-Index     and
node(1).isReinitialized == 1
```

Actions:

```
If ( CurrentProduct[Current-Indice] == 1):
        node(1).isSelected  == True
Else:
        node(1).isSelected  == False
node(1). isReinitialized = 0
Current-Index = Current-Index + 1
```

Test-Stop. Priorité 27

Conditions:

```
Conf <= ( Number-Confs – 1 )
```

Actions:

```
Continue = True
isStillValid = True
NewConfiguration-Generated  = 0
```

Figure 3.10: 2ème GG, les règles N°25-27

Règle N°25. Generating-NewConfiguration: Cette règle est appliquée pour générer une nouvelle configuration. On commence par l'incrémentation de la variable globale *Conf*. Ensuite, on calcule le tableau *Current-Product* contenant sa représentation binaire. Enfin, toutes les variables globales sont réinitialisées.

Règle N°26. Mapping-NewConfiguration: Cette règle sert à projeter cette nouvelle configuration sur le modèle D-Tree. Pour chaque nœud, l'attribut *isSelected* est ajusté selon la valeur correspondante dans le tableau binaire.

Règle N°27. Test-Stop : Cette règle est appliquée pour contrôler la répétition. A chaque itération, un test d'arrêt est effectué de la manière suivante :

- Si *Conf* < *Number-Confs* : On déclenche la vérification de cette nouvelle configuration. Dans ce cas, la première règle de la deuxième partie sera activée.
- Sinon: Tout le processus s'arrête. Aucune règle ne sera applicable.

Partie$_2$: Ici, le but est d'analyser uniquement la configuration courante [74].

Dans un premier temps, on s'intéresse uniquement aux relations parentales. On a constaté que pour une configuration incorrecte, il sera mieux de détecter une anomalie le plutôt possible. Afin d'optimiser le processus de vérification, nous proposons d'explorer le modèle D-Tree de haut en bas, en commençant par la racine jusqu'aux feuilles (Fig.3.11). Pour chaque nœud, sa relation avec ses fils est examinée en se basant sur le nombre de ceux qui sont sélectionnés. Deux cas de figure se présentent :

- Présence d'anomalie : la configuration en cours est considérée comme étant non valide et le processus de vérification s'arrête à ce niveau. Cela permet d'éviter des vérifications inutiles.
- Sinon, tous les fils appartenant sélectionnés seront traités à leur tour comme des pères.

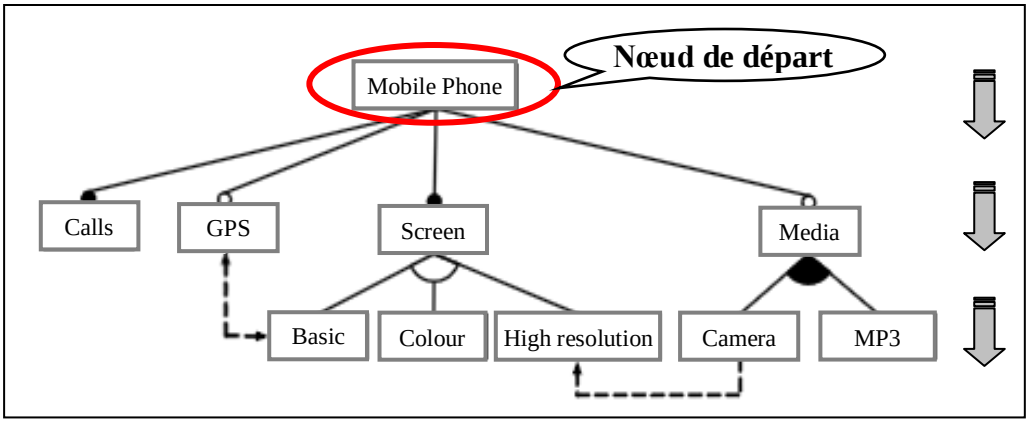


Figure 3.11: Processus de vérification des relations paternelles

Cette démarche donne de très bons résultats. Cependant, les descendants d'un nœud non sélectionné ne sont jamais examinés. Par conséquent, en cas d'anomalies aucune violation ne sera

signalée. Pour éviter ce problème, la présence d'un seul fils sélectionné dans sa descendance est considérée comme étant une erreur.

Ensuite, on passe à la vérification des contraintes d'implication et d'exclusion. A la fin, en cas où le produit en question reste toujours valide, il sera édité dans le fichier texte.

Cette validation est effectuée via treize règles. Les attributs temporaires utilisés sont:

- Pour chaque nœud:
  - *Current-Parent:* utilisé pour identifier le parent en cours de traitement.
  - *ToTreat-AsParent*: utilisé pour identifier les fils qui doivent être traités comme parents.
  - *isTreated-AsParent*: utilisé pour identifier les nœuds déjà traités en tant que parents.
  - *isVisited-AsChild*: utilisé pour identifier les nœuds fils déjà visités durant le traitement de leur parent.
  - *Count-SelectedChilds*: permet de spécifier le nombre de fils sélectionnés.
  - *isVisitedAsChild_CountSelectedChilds* : utilisé pour identifier les fils déjà visités lors du calcul des nombres de fils sélectionnés.
- Pour chaque association :
  - *ValidityChecked:* permet le parcours de toutes les relations lors du processus de vérification.
  - *AbsParent-PresChild_Checked*: permet de parcourir les relations paternelles afin de vérifier la présence de fils sélectionnés d'un père non sélectionné.
  - *isReinitialized* : utilisé pour la réinitialisation des attributs temporaires.

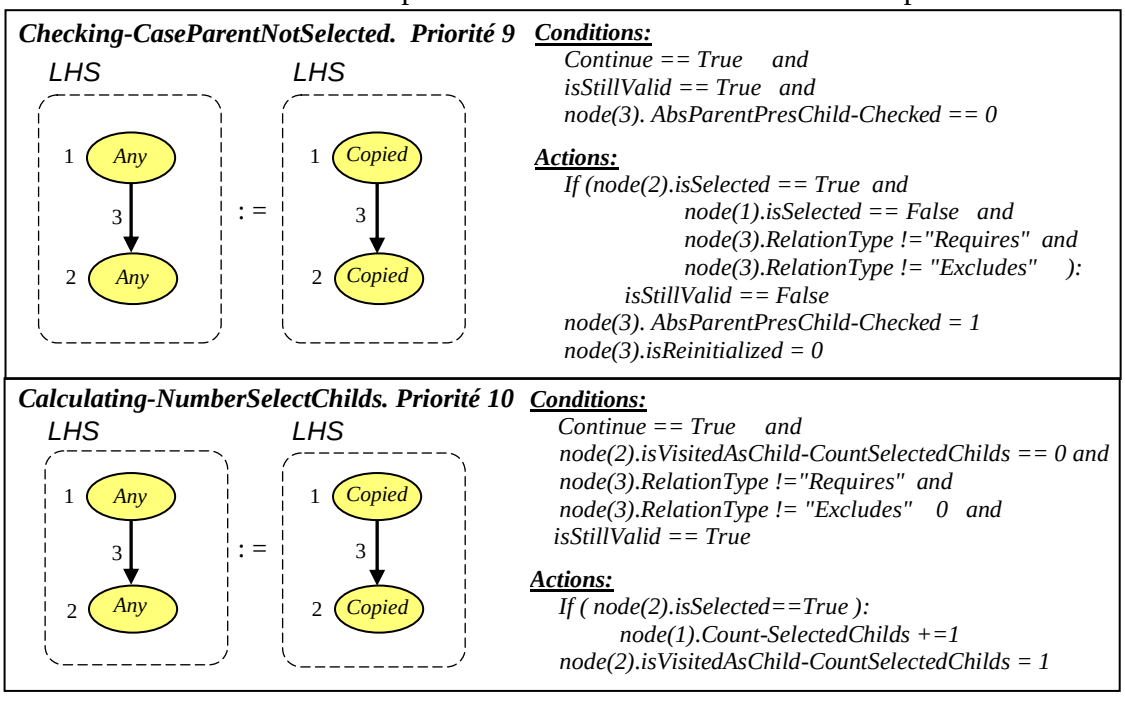
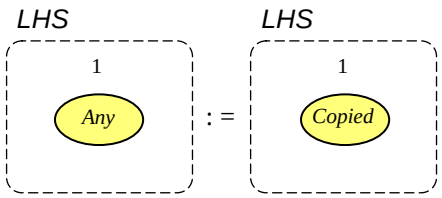


Figure 3.12 : 2ème GG, les règles N°9-10

StartChecking-PrentalRelationships. Priorité 11



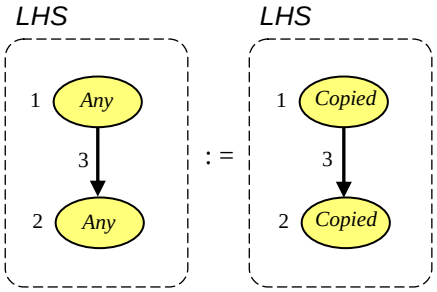
Conditions:

Continue == True and
node(1).isRoot == True and
node(1).ToTreat-AsParent == 0 and
node(1). isTreated-AsParent == 0 and
isStillValid == True

Actions:

If (node(1).isSelected == False):
isStillValid = False
node(1).ToTreat-AsParent =1

Checking-OR-Relationships-Validity. Priorité 12



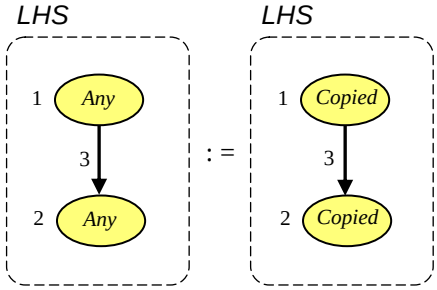
Conditions:

Continue == True and
node(1).Current-Parent = 1 and
node(3).RelationType = = "OR" and
node(3).ValidityChecked == 0
isStillValid == True

Actions:

If (node(1).Count-SelectedChilds == 0):
isStillValid = False
node(3).ValidityChecked = 1

OR-Relationships-Processing. Priorité 13



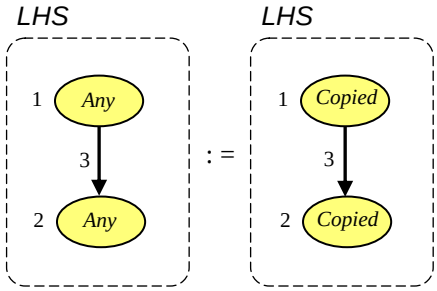
Conditions:

Continue == True and
node(1).Current-Parent = 1 and
node(3).RelationType = = "OR" and
node(2).isVisited-AsChild == 0 and
isStillValid == True

Actions:

If (node(2).isSelected == True):
node(2).ToTreat-AsParent =1
node(2).isVisited-AsChild = 1

Checking-XOR-Relationships-Validity. Priorité 14



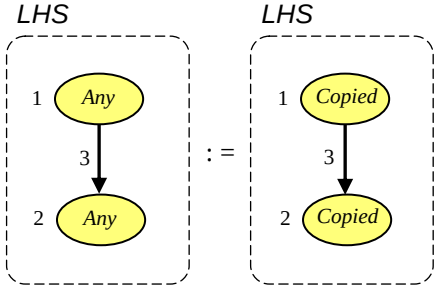
Conditions:

Continue == True and
node(1).Current-Parent = 1 and
node(3).RelationType = = "XOR" and
node(3).ValidityChecked == 0
isStillValid == True

Actions:

If (node(1).Count-SelectedChilds != 1):
isStillValid = False
node(3).ValidityChecked = 1

XOR-Relationships-Processing. Priorité 15



Conditions:

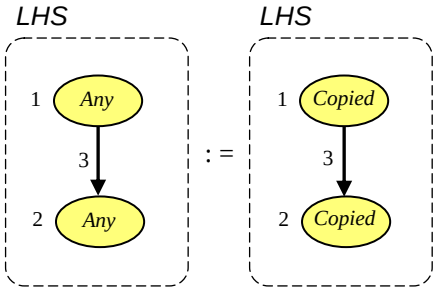
Continue == True and
node(1).Current-Parent = 1 and
node(3).RelationType = = "XOR" and
node(2).isVisited-AsChild == 0 and
isStillValid == True

Actions:

If (node(2).isSelected == True):
node(2).ToTreat-AsParent =1
node(2).isVisited-AsChild = 1

Figure 3.13: 2ème GG, les règles N°11-15

Checking-Mandatory-Relationships-Validity & Processing. Priorité 16



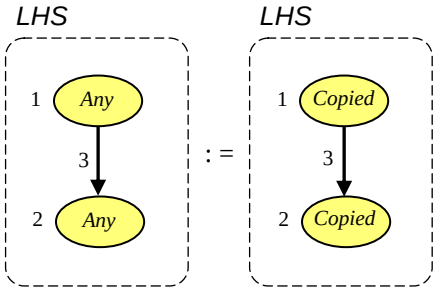
Conditions:

Continue == True and
node(1).Current-Parent = 1 and
node(3).RelationType = = "Mandatory" and
node(3).ValidityChecked == 0 and
isStillValid == True

Actions:

If (node(2).isSelected == False):
isStillValid = False
Else
node(2).ToTreat-AsParent = 1
node(3).ValidityChecked = 1

Optional-Relationships- Processing. Priorité 17



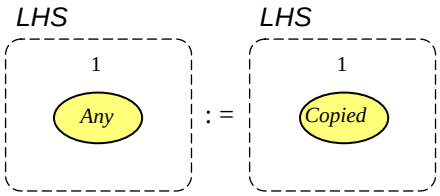
Conditions:

Continue == True and
node(1).Current-Parent = 1 and
node(3).RelationType = = "Optional" and
node(2).isVisited-AsChild == 0 and
isStillValid == True

Actions:

If (node(2).isSelected == True):
node(2).ToTreat-AsParent = 1
node(2).isVisited-AsChild = 1

SetCurrentParent-AsTreated. Priorité 18



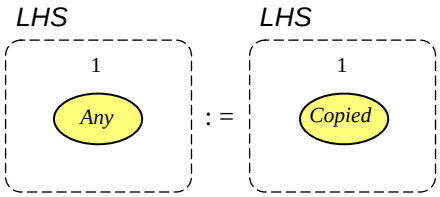
Conditions:

Continue == True and
node(1). Current-Parent == 1 and
node(1). isTreated-AsParent == 0 and
isStillValid == True

Actions:

node(1).Current-Parent = 0
node(1).isTreated-AsParent = 1

Select-Parent-NotTreated. Priorité 19



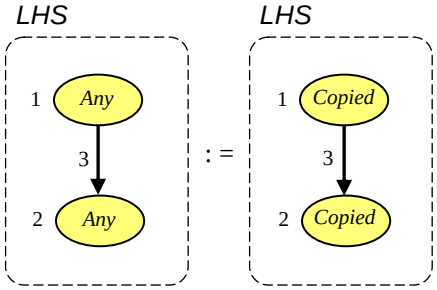
Conditions:

Continue == True and
node(1).ToTreat-AsParent == 1 and
node(1).isTreated-AsParent == 0 and
node(1).Current-Parent == 0 and
isStillValid == True

Actions:

node(1).Current-Parent = 1

Checking-Exludes-Relationships. Priorité 20



Conditions:

Continue == True and
node(3).RelationType== "Excludes" and
node(3) ValidityChecked == 0 and
isStillValid == True

Actions:

If(node(1).isSelected==True && node(2)is.Selected== True):
isStillValid == False
node(3). ValidityChecked = 1

Figure 3.14: 2ème GG, les règles N°16-20

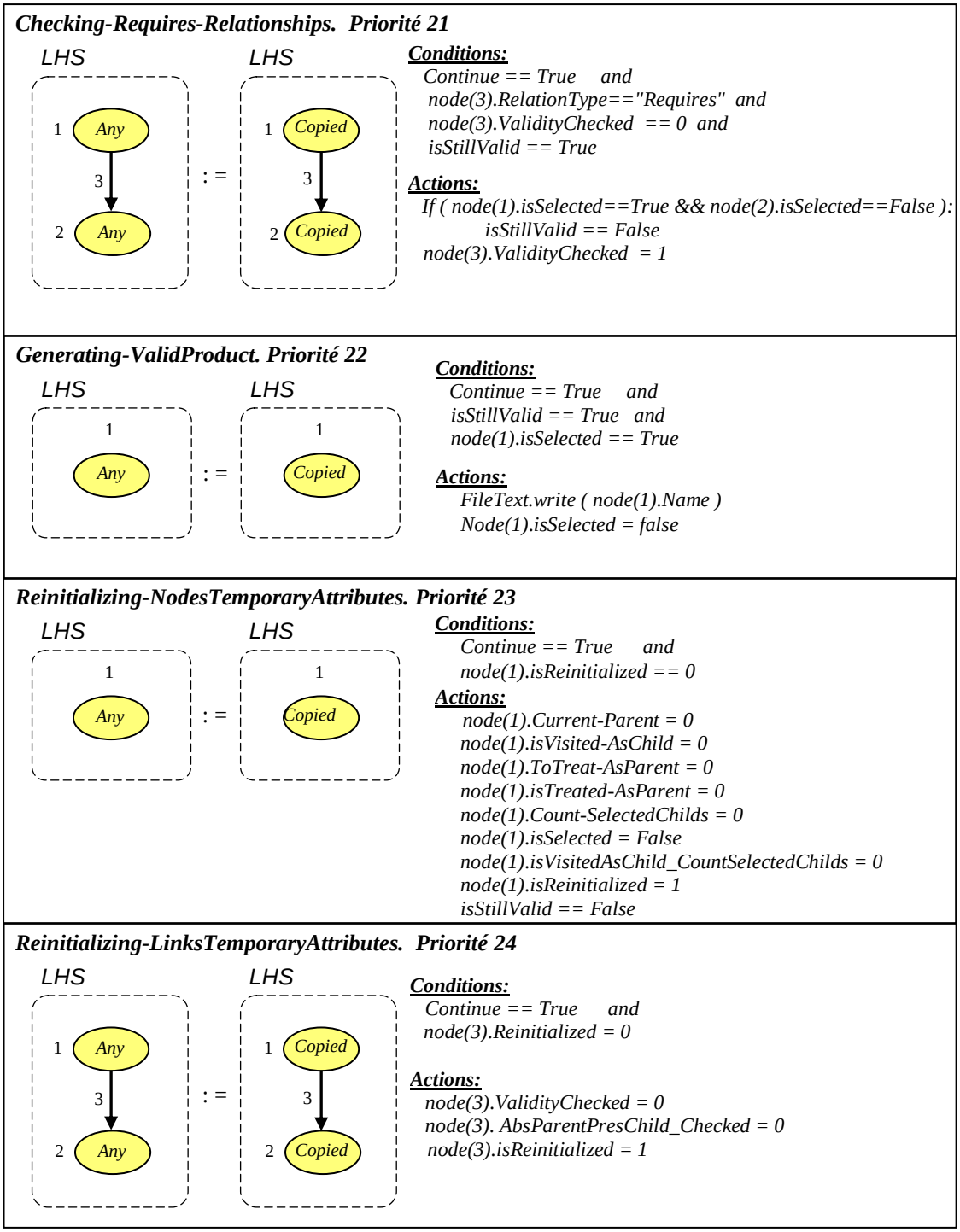


Figure 3.15: 2ème GG, les règles N°21-24

***Règle N°.9 Checking-CaseParentNotSelected*:** Cette règle est appliquée pour s'assurer qu'il n'existe pas de fils sélectionné d'un père non sélectionné.

***Règle N°10. Calculating-NumberSelectedChilds* :** Cette règle sert à calculer le nombre de fils sélectionnés (*Count-SelectedChilds*) pour chaque nœud du modèle D-Tree.

Règle N°11. StartChecking-PrentalRelationships : Cette règle sert à déclencher la vérification des relations paternelles. La racine est sélectionnée comme étant le premier père à traiter.

Règles N°12, N°14 et N°16 : sont utilisées pour la validation des relations OR, XOR et Mandatory respectivement. Pour les relations de type Optional, aucune vérification n'est nécessaire.

Règles N°13, N°15 et N°17 : sont utilisées pour préparer le traitement des fils sélectionnés (prochainement) en tant que parents. Pour les relations de type Mandatory, ce traitement est réalisé au moment de la validation du fait qu'il s'agit d'un seul fils.

Règle N°18. SetCurrentParent-AsTreated**:** Une fois que tous les traitements nécessaires ont été réalisés pour le père courant, cette règle est appliquée pour le marquer comme étant déjà traité.

Règle N°19. Select-Parent-NotTreated**:** Cette règle est appliquée pour localiser un nœud candidat à être traité comme un père. Il sera marqué comme le père courant.

Règles N°20 et N°21 : sont utilisées pour valider les contraintes d'implication et d'exclusion.

Règle N°22. Generating-ValidProduct**:** Dans le cas d'une configuration valide, cette règle est appliquée pour générer le produit correspondant. Tous les nœuds sélectionnés sont édités dans le fichier texte.

Règle N°23. Reinitializing-NodesTemporaryAttributes**:** Une fois le traitement de la configuration courante est terminé, on procède à la préparation de l'itération suivante. Par conséquent, toutes les variables temporaires doivent être réinitialisées. Cette règle est appliquée pour la remise à zéro des attributs utilisés au niveau des nœuds.

Règle N°24. Reinitializing-LinksTemporaryAttributes**:** De même, cette règle est utilisée pour réinitialiser tous les attributs utilisés au niveau des relations.

En analysant la structure des configurations, on a remarqué que le rejet d'une affectation partielle contenant quelques caractéristiques entraine automatiquement celui de toutes les configurations auxquelles elle fait partie. Afin d'optimiser la solution présentée précédemment, on s'est aperçu qu'il sera mieux d'intégrer un mécanisme permettant de réduire l'espace de recherche des produits valides selon cette perspective.

Vue la nature de notre problème, on a opté pour l'algorithme BackTracking[75]. C'est une technique largement utilisée pour la résolution des problèmes de satisfaction de contraintes (CSP). La grammaire de graphe proposée sera présentée dans la section suivante.

3.5 Version optimisée

3.5.1 Le Bachtracking

Le Backtracking est une stratégie de recherche utilisée principalement dans les solveurs de contraintes. Le fonctionnement de base est de choisir une seule variable à la fois, et d'envisager pour elle une valeur de son domaine, en s'assurant que la nouvelle valeur choisie est compatible avec l'affectation partielle courante. Si l'affectation actuellement choisie viole certaines contraintes une autre valeur qui est disponible sera prise. En cas où toutes les variables sont affectées, le problème est résolu. Si, à n'importe quelle étape, aucune valeur ne peut être affectée à une variable sans violer une contrainte, l'affectation de la dernière variable (juste avant l'´echec) sera révisée, et une autre valeur, lorsqu'elle est disponible, sera assignée à cette variable. Cela procède de suite jusqu'à ce qu'une solution soit trouvée ou que tous les tests des combinaisons de valeurs aient échoué.

L'algorithme suivant donne toutes les affectations possibles sous forme d'une procédure récursive.

```
Fonction Backtracking ( A,V )
Début
    Si V = ∅ alors
        Imprimer A   ( Retourner Vrai)
    Sinon
        Choisir x ∈V
        D ← D_x
        Consistance ←  Faux
        Tant que ( D =/= Ø et non (Consistance)) Faire
            Choisir v dans D
            D ← D \ { v }
            Si  existe pas c ∈C telle que c viole A union {x ← v } Alors
                Consistence ←  Backtracking ( A union {x ← v } ,V \ { x } )
            Fsi
        Fin TantQue
        Retourner ( Consistance)
    Fsi
Fin
```

Figure 3.16: Algorithme BackTracking

Tel que :

- *A : l'affectation courante. Elle est initialisée avec Ø.*
- *V* : L'ensemble des variables non instanciées. Il est initialisé avec *X*.

3.5.2 Adaptation de l'algorithme de recherche

Comme l'approche transformation de graphes ainsi que les outils associés ne supportent pas la récursivité, nous devons élaborer une version itérative de l'algorithme précédent (Fig.3.16). Il s'agit d'une adaptation basée sur l'utilisation du tableau binaire noté cette fois-ci *Partial-Assignment*. En effet, considérons le cas général :

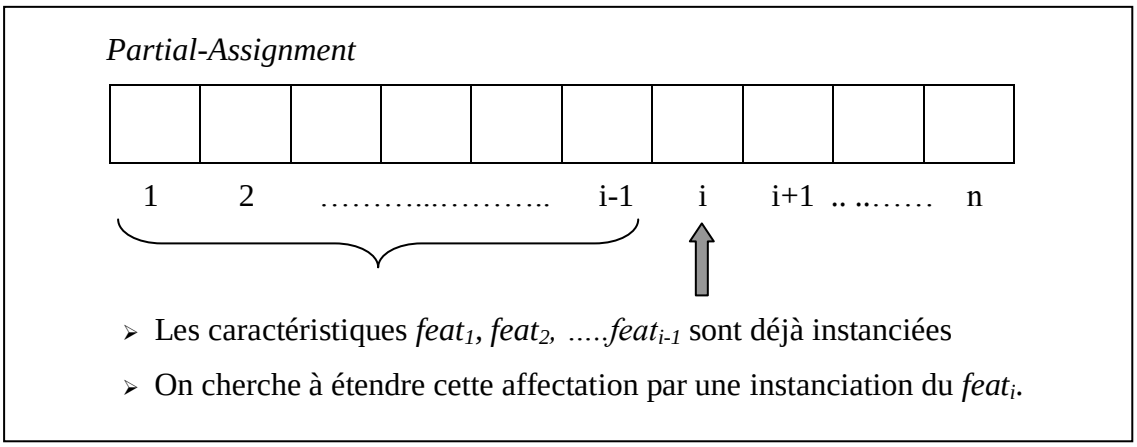


Figure 3.17: Manipulation du tableau binaire

A ce niveau, on cherche à étendre de façon consistante l'affectation partielle contenant les caractéristiques $feat_1$, $feat_2$, ...$feat_{i-1}$ par une instanciation correcte de la caractéristique $feat_i$. Pour cela, on commence d'abord par tester la valeur 0. On évalue alors la consistance de cette nouvelle affectation en vérifiant qu'aucune contrainte liant $feat_i$ à une caractéristique déjà instanciée n'est violée. En cas d'anomalie, on testera $feat_i$ avec la valeur 1. Si la nouvelle affectation est consistante, on procédera de la même manière avec la caractéristique $feat_{i+1}$. Par contre, si cette nouvelle affectation est à son tour inconsistante, il faux faire un retour arrière à la caractéristique précédente $feat_{i-1}$ et ainsi de suite. La recherche se poursuit jusqu'à ce que :

- Toutes les variables aient été instanciées de façon consistante (i > n). Dans ce cas, le produit correspondant à cette affectation sera édité dans le fichier texte. Afin d'entamer la recherche de la solution suivante, on doit d'abord localiser la dernière caractéristique évaluée uniquement avec la valeur 0. Ensuite, on recommence le même traitement en lui affectant la valeur 1.
- Ou bien, le retour arrière se termine en effectuant un débordement à gauche du tableau (i < 1) . Dans ce cas, le reste des possibilités ne peut donner aucune affectation globale consistante. Donc, tout le processus s'arrête.

Pour implémenter cette solution, on propose l'algorithme suivant :

```
Pour i = 1 à n faire
    Partial-Assignment [ i ] ← -1
FinPour

i ← 1 ;  Forward ← Vrai ;
Répéter
        Tant que ( i >= 1 et  i <= n ) Faire
            Si (Avancer ) alors
                    Si ( Partial-Assignment [ i ] = -1 ) alors
                        Partial-Assignment [ i ] ← 0
                        Si ( valide (Partial-Assignment , i) ) alors
                            i ← i +1 ;
                        FSi
                    Sinon
                        Partial-Assignment [ i ] ← 1
                        Si ( valide (Partial-Assignment , i) ) alors
                            i ← i +1
                        Sinon
                            Forward ← Faux
                        FSi
                    FSi
            Sinon
                    Si (Partial-Assignment [ i ] = 0) alors
                        Forward ← Vrai
                    Sinon
                        Partial-Assignment [ i ] ← -1
                        i ← i -1
                    FSi
                FSi
        Fin Tant que
          Si ( i > n ) alors
                Editer cette affectation dans le fichier texte.
                i ← n
                Tant que ( (Partial-Assignment [ i ] = 1 ) and ( i > 0) ) Faire
                        Partial-Assignment [ i ] ← -1
                        i ← i – 1
                Fin Tant que
                Si ( i != -1 ) alors
                        Forward ← Faux ;
                FSi
          FSi
Jusqu'à ( i < 1 )
```

Figure 3.18: Adaptation de l'algorithme BackTracking

Tel que :

- Les éléments du tableau binaire sont initialisés à -1. De même, dans un retour arrière, ceux qui sont parcourus (valeur égale à 1) sont réinitialisés par cette valeur.
- *Forward* : un booléen utilisé pour spécifier le sens du déplacement dans le tableau binaire.
- *Valide(Partial-Assignment ,i): une fonction* booléenne utilisée pour la vérification de l'affectations partielle contenant les caractéristiques $feat_1$, $feat_2$, $.....feat_{i-1}$.

3.5.3 Grammaire de graphe basée sur le BackTracking: 3ème GG

La grammaire de graphes proposée est constituée de deux parties. Les règles de la première partie implémentent l'algorithme principal. A chaque nouvelle instanciation d'une variable, celles de la deuxième partie sont invoquées afin de vérifier la validité de la l'affectation partielle obtenue. C'est une implémentation de la fonction *Valide( )*.

Partie$_1$: Les règles de cette partie (Fig.3.19 et Fig.3.20) sont utilisées principalement pour la manipulation des affectations partielles. Nous proposons d'utiliser les variables globales suivantes :

- *Partial-Assign* : le tableau binaire.
- *Index-Max*: la taille du tableau binaire. Elle est déjà calculée par la première grammaire 1ere GG.
- *isStillValid*: un booléen utilisé pour récupérer les résultats de la vérification des affectations partielles.
- *Index-FeatInstanced*: un entier utilisé pour repérer la caractéristique en cours de traitement dans le tableau binaire. Il est initialisé à 0.
- *Forward* : un booléen utilisé pour spécifier le sens du déplacement dans le tableau binaire.
- *Start-PartialVerification* : un booléen utilisé pour déclencher la vérification de la nouvelle affectation partielle générée.
- *Continue*: un booléen utilisé afin de désactiver les règles en cas de débordement du tableau binaire.

TestValidity- NewFeatAssignedAsNotSelected. Priorité 21

Conditions:	***Actions:***
Continue == True and *Forward == True and* *Partial-Assign [Index_FeatInstanced] == -1*	*Partial-Assign [Index-FeatInstanced] = 0* *isStillValid = True* *Start-PartialVerification = True*

InterpretationVerifResult- CasePartialAssigValid- NewFeatAssignedAsNotSelected. Priorité 22

Conditions:	***Actions:***
Continue == True and *Forward == True and* *Partial-Assign [Index-FeatInstanced] == 0 and* *isStillValid == True*	*Index-FeatInstanced = Index-FeatInstanced + 1* *Continue = False*

Figure 3.19: 3ème GG, les règles N°21-22

TestValidity- NewFeatAssignedAsSelected. Priorité 23

Conditions:
Continue == True and
Forward == True and
Partial-Assign [Index_FeatInstanced] == 0

Actions:
Partial_Assign [Index-FeatInstanced] = 1
isStillValid = True
Start-PartialVerification = True

InterpretationVerifResult- NewFeatAssignedAsSelected. Priorité 24

Conditions:
Continue == True and
Forward == True and
Partial-Assign [Index_FeatInstanced] ==1

Actions:
If (isStillValid == True) :
Index-FeatInstanced = + 1
Else:
Forward == False
Continue == False

Backtracking. Priorité 25

Conditions:
Continue == True and
Forward == False

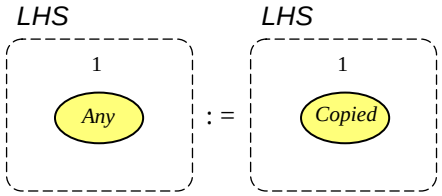
Actions:
If (Partial-Assign[Index-FeatInstanced] == 0) :
Forward == True
Else:
Partial-Assign [Index-FeatInstanced] = -1
Index-FeatInstanced = Index-FeatInstanced – 1
Continue = False

Verify-Debordment. Priorité 26

Conditions:
Index-FeatInstanced >= 0 and
Index-FeatInstanced <= Index_Max

Actions:
Continue = True

Generating-ValidProduct. Priorité 27



Conditions:
Index-FeatInstanced > Index-Max and
Partial-Assign [node(1).Index] == 1
node(1).isEditedTextFile == 0

Actions:
FileText.write (node(1).Name)
node(1).isEditedTextFile = 1

Searching-NewProduct. Priorité 28

Conditions:
Index-FeatInstanced > Index_Max

Actions:
ind =Index_Max
while ((Partial-Assign[ind] == 1) and (ind >= 0)) :
Partial-Assign [ind] = -1
ind = ind – 1

Index-FeatInstanced = ind
Continue = False

if (ind != -1) :
Index-FeatInstanced = ind
Continue = True
Forward = 1
isStillValid = False

Figure 3.20: 3ème GG, les règles N°23-28

Pour simplifier l'explication de ces règles, notons $feat_i$ la caractéristique en cours de traitement. C'est l'élément d'indice *Indice_FeatInstancié* dans le tableau binaire.

Règle N°21. TestValidity-NewFeatureAssignedAsNotSelected: Cette règle est appliquée afin d'étendre l'affectation partielle courante en affectant la valeur 0 à la caractéristique $feat_i$. Une fois l'instanciation réalisée, le processus de validation est relancé pour vérifier la consistance de la nouvelle affectation obtenue.

RègleN°22. InterpretationVerifResult-CasePartialAssigValid-NewFeatAssignedAsNotSelected : Cette règle est appliquée si l'affectation générée précédemment ($feat_i = 0$) est valide. Dans ce cas, on passe au traitement de la caractéristique suivante.

Règle N°23. TestValidity-NewFeatureAssignedAsNotSelected: Cette règle est appliquée afin d'étendre l'affectation partielle courante en affectant la valeur 1 à la caractéristique $feat_i$. De même, le processus de validation est relancé pour vérifier la consistance de la nouvelle affectation.

Règle N°24. InterpretationVerificationResult-NewFeatureAssignedAsSelected: Cette règle est appliquée afin d'interpréter le résultat de validation de l'affectation précédente ($feat_i$ =1). Deux cas de figures sont possibles :

- Si cette affectation est valide, on passe au traitement de la caractéristique suivante.
- Sinon, on fait appel à un retour arrière (*Règle N°25)*. Il n'y à plus de valeurs à tester pour cette caractéristique.

Règle N°25. Backtracking : Cette règle est appliquée pour traiter le retour arrière.

- Dans le cas ou la caractéristique $feat_i$ n'est évaluée qu'avec la valeur 0, on active la *règle N°23* afin de tester la valeur 1.
- Sinon, un autre retour arrière sera effectué.

Règle N°26. Verify-Debordment: Cette règle est appliquée après chaque déplacement dans le tableau binaire afin de vérifier la valeur du nouveau indice.

- S'il n'y a pas de débordement, la règle planifiée sera activée.
- Sinon, deux cas de figures sont possibles :
  - Débordement à droite (i > n) : une affectation globale consistante est trouvée. Dans ce cas, la règle permettant de générer le produit correspondant dans le fichier texte sera déclenchée (*Règle N°27)*.
  - Débordement à gauche (i < 1) : tout le processus s'arrête et aucune règle ne sera applicable.

Règle N°27. Generating-ValidProduct: Cette règle est appliquée après un débordement à droite. Elle sert à générer le produit correspondant dans le fichier texte. L'édition des caractéristiques sélectionnées est basée sur l'utilisation d'un attribut temporaire associé aux nœuds du modèle D-Tree noté *isEditedTextFile*.

Règle N°28. Searching-NewProduct: Cette règle est appliquée juste après la règle précédente (Règle N°27) afin de relancer la recherche d'une autre affectation globale. Elle localise d'abord la dernière caractéristique évaluée uniquement avec la valeur 0 en parcourant le tableau binaire de droite à gauche. Ensuite, elle déclenche son test avec la valeur 1 en activant la *Règle N°23*.

Partie$_2$: Les règles de cette partie sont activées à chaque instanciation d'une caractéristique et plus précisément elles sont déclenchées par les *Règles N°21 et N°23*. Ici, le traitement se limite uniquement aux caractéristiques appartenant à l'affectation partielle. Pour ce faire, nous avons adapté les règles proposées dans l'approche naïve. La vérification de chaque dépendance du diagramme *D-Tree* est conditionnée par l'instanciation des caractéristiques source et destination dans le tableau binaire.

On procède en trois étapes :

- Projection du tableau binaire sur le modèle *D-Tree*.
- Validation des relations paternelles et des contraintes concernées.
- Réinitialisation des variables globales et des attributs temporaires en vue de préparer le traitement de la prochaine affectation partielle.

Le résultat de cette vérification dépend de la valeur finale de la variable globale *Is_StillValid*. Il sera utilisé par les *Règles N°22 et N°24*.

Les règles proposées sont présentées dans les figures Fig.3.21, Fig.3.22 et Fig.3.23 (voir ci-dessous). Elles sont basées sur l'utilisation des attributs temporaires suivants:

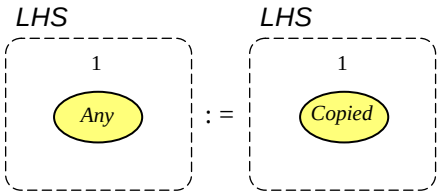
- Pour chaque nœud du modèle *D-Tree*:
  - *Count-SelectedChilds:* permet de spécifier le nombre de fils sélectionnés.
  - *isVisitedAsChild-CountSelectedChilds :* utilisé pour identifier les fils déjà visités lors du calcul des nombres de fils sélectionnés.
  - *isReinitialized* : utilisé pour la réinitialisation des attributs temporaires.
- Pour chaque association :
  - *ValidityChecked :* permet le parcours de toutes les relations lors du processus de vérification.
  - *AbsParentPresChild-Checked :* permet de parcourir les relations paternelles afin de vérifier la présence de fils sélectionnés d'un père non sélectionné.
  - *isReinitialized* : utilisé pour la réinitialisation des attributs temporaires.

Les variables globales ajoutées sont :

- *Current-Index* : un entier utilisé pour parcourir le tableau binaire.

- *GlobalVariablesReinitialized*: un booléen utilisé pour la réinitialisation des variables globales.
- *Root-Checked* : un booléen utilisé pour vérifier la présence de la racine.

Mapping-NewPartialAssignment. Priorité 9



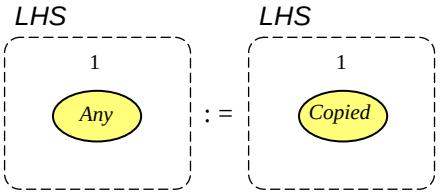
Conditions:

Start-PartialVerification = True and
CurrentIndex <= Index-FeatInstanced and
node(1).Index == CurrentIndex and
node(1).isReinitialized == 1 and

Actions:

If (Partial-Assign [CurrentIndex] == 1):
node(1).isSelected == True
Else:
node(1).isSelected == False
node(1). isReinitialized = 0
CurrentIndex = CurrentIndex + 1
GlobalVariablesReinitialized = 0

Checking-Root-Selection. Priorité 10



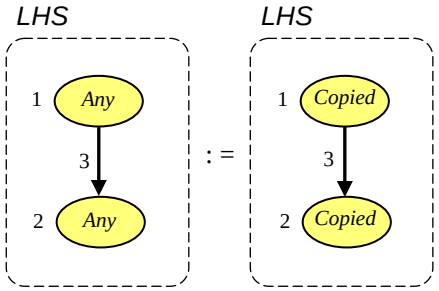
Conditions:

Start-PartialVerification = True and
node(1).isRoot == True and
RootNode.Index <= Index-FeatInstanced
isStillValid == True
Root-Checked == 0

Actions:

If (node(2).isSelected == False):
isStillValid == False
Root_Checked = 1

Checking-CaseParentNotSelected. Priorité 11



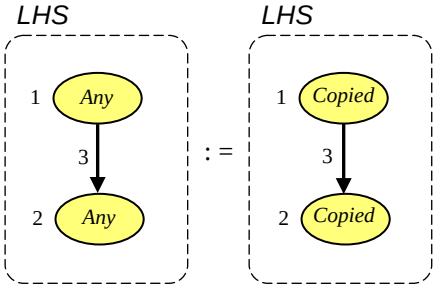
Conditions:

Start-PartialVerification = True and
node(1).Index <= Index-FeatInstanced and
node(2).Index <= Index-FeatInstanced and
isStillValid == True and
node(3). AbsParent-PresChild-Checked == 0

Actions:

If (node(2).isSelected == True and
node(1).isSelected == False and
node(3).RelationType !="Require" and
node(3).RelationType != "Exclude"):
isStillValid == False
node(3). AbsParent-PresChild-Checked = 1
node(3).isReinitialized = 0

Calculating-NumberSelectedChilds. Priorité 12



Conditions:

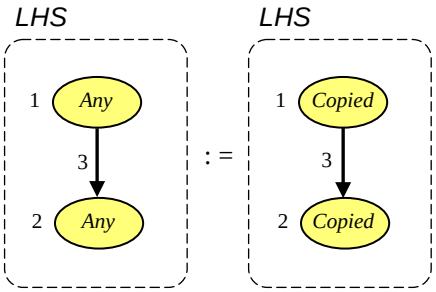
Start_PartialVerification = True and
isStillValid == True and
node(2).isVisitedAsChild-CountSelectedChilds == 0 and
node(3).RelationType !="Require" and
node(3).RelationType != "Exclude" and
node(1).Index <= Index-FeatInstanced and
node(2).Index <=Index-FeatInstanced

Actions:

If (node(2).isSelected==True):
node(1).Count-SelectedChilds +=1
node(2).isVisitedAsChild-CountSelectedChilds = 1

Figure 3.21: 3ème GG, les Règles N°9-12

Checking-OR-Relationships-Validity. Priorité 13



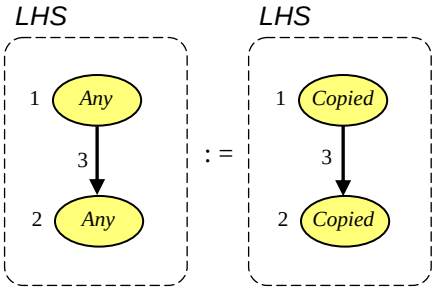
Conditions:

Start_PartialVerification = True and
node(1).Index <= Index-FeatInstanced and
node(3).RelationType = = "OR" and
node(3).ValidityChecked == 0
isStillValid == True

Actions:

```
If (node(1).isSelected == True and
                Index-FeatInstanced == Index_Max and
                node(1).Count-SelectedChilds == 0):
        isStillValid = False
    node(3).ValidityChecked = 1
```

Checking-XOR-Relationships-Validity. Priorité 14



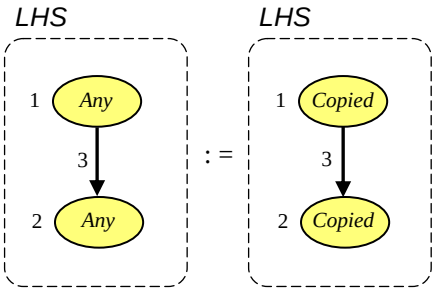
Conditions:

Start-PartialVerification = True and
node(1).Index <= Index-FeatInstanced and
node(3).RelationType = = "XOR" and
node(3).ValidityChecked == 0
isStillValid == True

Actions:

```
If (node(1).isSelected == True and
                Index-FeatInstanced == Index_Max ):
        If (node(1).Count-SelectedChilds != 1):
            isStillValid = False
    Else:
        If (node(1).Count-SelectedChilds > 1):
            isStillValid = False
    node(3).ValidityChecked = 1
```

Checking-Mandatory-Relationships-Validity & Processing. Priorité 15



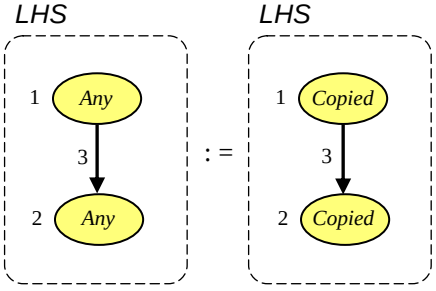
Conditions:

Start-PartialVerification = True and
node(1).Index <= Index-FeatInstanced and
node(2).Index <= Index-FeatInstanced and
node(3).RelationType = = "Mandatory" and
node(3).ValidityChecked == 0 and
isStillValid == True

Actions:

```
If (node(1).isSelected == True and
                node(2).isSelected == False ):
        isStillValid = False
    node(3).ValidityChecked = 1
```

Checking-Exlude-Relationships. Priorité 16



Conditions:

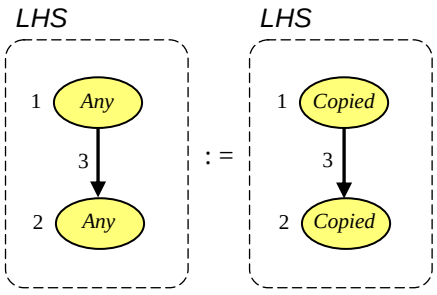
Start_PartialVerification = True and
node(1).Index <= Index-FeatInstanced and
node(2).Index <= Index-FeatInstanced and
node(3).RelationType== "Exclude" and
node(3) ValidityChecked == 0 and
isStillValid == True

Actions:

```
If( node(1).isSelected==True && node(2).isSelected== True ):
        isStillValid == False
    node(3). ValidityChecked = 1
```

Figure 3.22: 3ème GG, les Règles N°13-16

Checking- Require-Relationships. Priorité 17



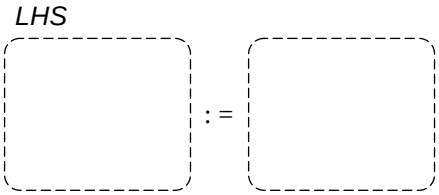
Conditions:

Start-PartialVerification = True and
node(1).Index <= Index-FeatInstanced and
node(2).Index <= Index-FeatInstanced and
node(3).RelationType== "Require" and
node(3) ValidityChecked == 0 and
isStillValid == True

Actions:

If(node(1).isSelected==True && node(2)is.Selected== False):
isStillValid == False
node(3). ValidityChecked = 1

Reinitialization-GlobalVariables. Priorité 18



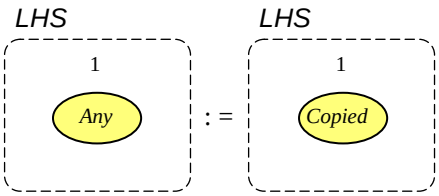
Conditions:

GlobalVariablesReinitialized == 0

Actions:

CurrentIndex = 0
Root-Checked = 0
Start-PartialVerification = False
GlobalVariablesReinitialized = 1

Reinitializing-NodesTemporaryAttributes. Priorité 19



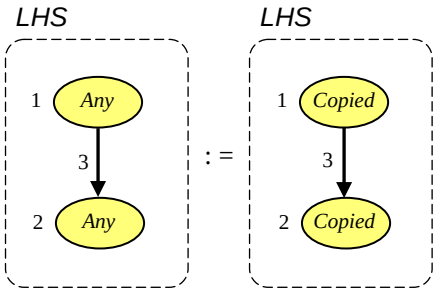
Conditions:

node(1).isReinitialized = 0

Actions:

node(1).Count_SelectedChchilds = 0
node(1).isSelected = False
node(1).isVisitedAsChild-CountSelectedChilds = 0
node(1)is.EditedTextFile = 0
node(1).isReinitialized = 1

Reinitializing-LinksTemporaryAttributes. Priorité 20



Conditions:

node(3).isReinitialized = 0

Actions:

node(3).Validity_Checked = 0
node(3). AbsParent-PresChild-Checked = 0
node(3).isReinitialized = 1

Figure 3.23: 3ème GG, les Règles N°17-20

3.6 Exemple d'application

Pour illustrer cette approche, considérons l'exemple *Mobile-Phone* présenté au début de ce chapitre (Fig.3.1). Une fois les Méta-Modèles des diagrammes *FD* et *D-Tree* spécifiés, AToM^3 génère automatiquement un éditeur graphique permettant de manipuler les entités des deux formalismes. Il permet également de spécifier les grammaires de graphes proposées.

- On commence d'abord par la création du diagramme *FD* tel que présenté dans la figure Fig.3.24.

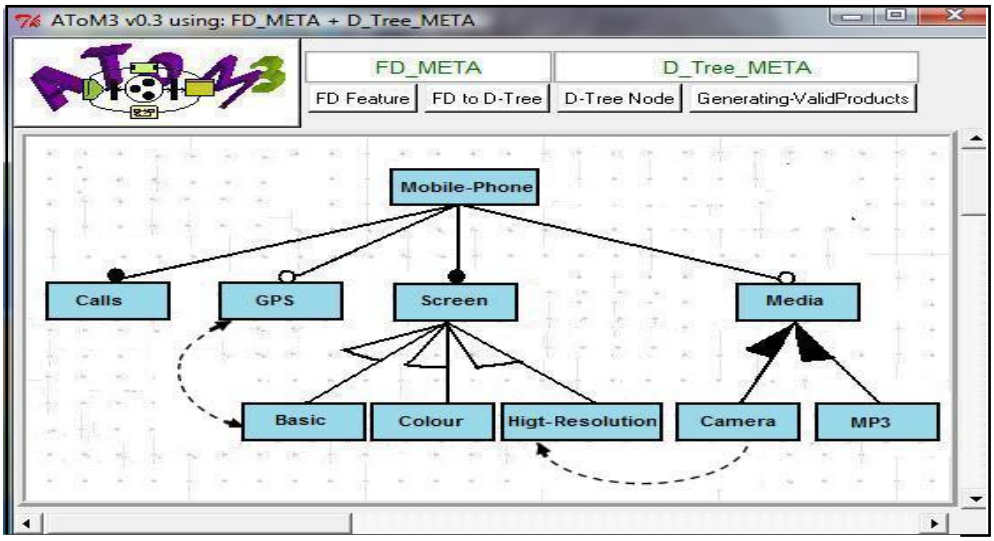


Figure 3.24: Diagramme FD initial

- En exécutant les règles de la première grammaire (1èreGG), le diagramme *FD* est translaté en un modèle *D-Tree* équivalent (Fig.3.25).

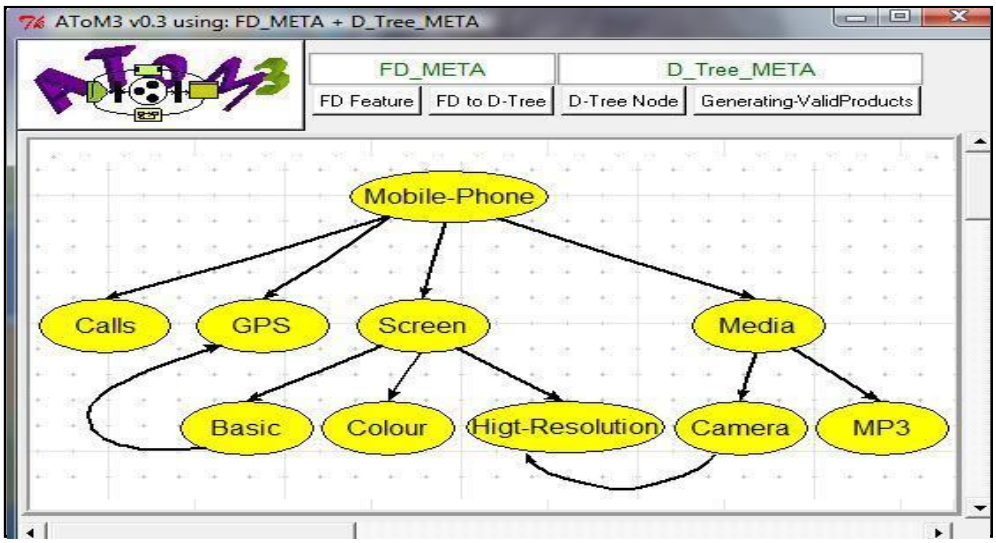


Figure 3.25: Modèle D-Tree généré

- Ensuite, on passe à la recherche des produits structurellement valides. Les deux versions proposées, simple et optimisée, donnent le même résultat.
  - Le fichier texte généré en exécutant la 2ème GG ou bien la 3ème GG est présenté dans la figure suivante :

Valid Products:
- Mobile-Phone, Calls, Screen, Basic
- Mobile-Phone, Calls, Screen, Basic, Media, MP3
- Mobile-Phone, Calls, Screen, Colour
- Mobile-Phone, Calls, Screen, Colour, Media, MP3
- Mobile-Phone, Calls, Screen, Colour, GPS
- Mobile-Phone, Calls, Screen, Colour, GPS, Media, MP3
- Mobile-Phone, Calls, Screen, Higt-Resolution
- Mobile-Phone, Calls, Screen, Higt-Resolution, Media, MP3
- Mobile-Phone, Calls, Screen, Higt-Resolution, Media, Camera
- Mobile-Phone, Calls, Screen, Higt-Resolution, Media, Camera, MP3
- Mobile-Phone, Calls, Screen, Higt-Resolution, GPS
- Mobile-Phone, Calls, Screen, Higt-Resolution, GPS, Media, MP3
- Mobile-Phone, Calls, Screen, Higt-Resolution, GPS, Media, Camera
- Mobile-Phone, Calls, Screen, Higt-Resolution, GPS, Media, Camera, MP3

Figure 3.26: Produits structurellement valides

Pour cet exemple, on a trouvé 14 produits différents. C'est le même résultat présenté dans [45].

3.7 Conclusion

Dans ce chapitre, nous avons présenté un outil automatique permettant la génération de tous les produits structurellement valides d'un diagramme de caractéristiques. La technique proposée consiste en une recherche des configurations vérifiant toutes les dépendances en utilisant l'approche transformations de graphes. Nous avons proposé deux approches différentes.

La première approche est basée sur une recherche exhaustive. L'idée de base est de calculer toutes les combinaisons possibles d'un tableau binaire spécifiant les caractéristiques. Pour chaque combinaison, le produit correspond est projeté sur le diagramme FD pour le vérifier. En cas de validité, il est édité dans un fichier texte.

La deuxième approche est une solution optimisée. Afin de réduire l'espace des configurations possibles, nous avons opté pour l'intégration du Backtracking dans le processus de recherche. Comme la technique transformation de graphes ainsi que les outils associés ne supportent pas la récursivité, une version itérative adaptée de cet algorithme est proposée.

Pour ces deux approches, les Méta-Modèles utilisés ainsi que les grammaires de graphes associées sont présentés.

Chapitre 04

Génération Automatique des Produits Structurellement Valides : Approche Basée sur la Composition de Configurations Partielles

4.1 Introduction

Dans le chapitre précédent, nous avons proposé une approche permettant de générer tous les produits valides à partir du diagramme FD. Elle est basée sur la recherche des variantes qui satisfont les dépendances en testant toutes les configurations possibles. Malgré l'optimisation apportée par l'intégration de l'algorithme BackTraking, le temps de calcul demeure relativement considérable pour des FDs de grande taille.

Pour remédier à ce problème, on a constaté que la meilleure solution sera d'éviter carrément l'exploration de l'espace de recherche. Par conséquent, dans ce chapitre, nous proposons une technique basée le calcul (plutôt que des tests) des produits valides. A partir des caractéristiques, l'idée consiste à construire progressivement des configurations partielles valides plus larges jusqu'à l'obtention des produits recherchés [76].

Dans la suite, nous présentons d'abord le principe de calcul ainsi que la technique utilisée pour la construction des configurations partielles. Le traitement détaillé de tous les cas de figures possibles sera donné. Ensuite, nous exposons la grammaire de graphe proposée. Enfin, ce chapitre se termine par un exemple illustratif et une conclusion.

4.2 Principe de calcul des produits valides

Dans un premier temps, on s'intéresse uniquement aux relations paternelles. Pour déterminer l'ensemble de produits valides à ce stade, on procède par une exploration ascendante du diagramme FD (Fig.4.1).

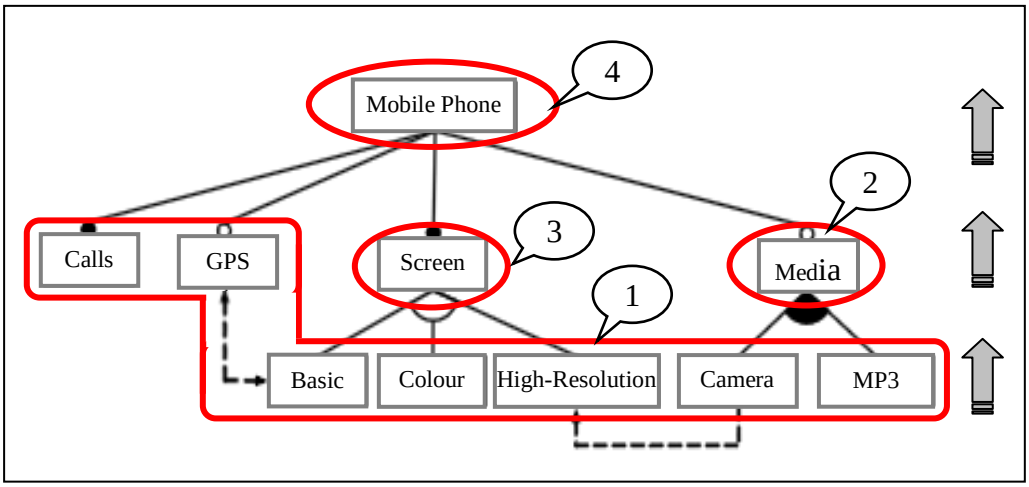


Figure.4.1: Traitement des relations paternelles

Le traitement commence par les feuilles. Comme ce type de nœuds n'a pas fils, les configurations partielles possibles se limitent uniquement aux caractéristiques associées. Elles constituent les

configurations les plus élémentaires. Une fois que toutes les feuilles ont été visitées, on passe au traitement des nœuds intermédiaires jusqu'à la racine. Il s'agit d'un traitement itératif, où à chaque itération :

- On sélectionne un nœud pour lequel tous les fils sont déjà traités.
- On construit l'ensemble des configurations partielles possibles à son niveau par une composition des configurations partielles déjà calculées au niveau de ses fils. C'est une combinaison de listes qui dépend principalement du type de la relation paternelle. La technique proposée sera détaillée dans la section suivante.

En suivant cette démarche, le résultat final de cette exploration sera calculé au niveau de la racine. Ensuite, on passe au traitement des contraintes. Parmi les produits obtenus, ceux qui violent les dépendances Require et Exclude seront supprimés. Pour le reste des configurations, toutes les dépendances du diagramme FD sont respectées. En plus, elles sont globales dans le sens où toutes les caractéristiques sont prises en considération. Elles constituent donc tous les produits structurellement valides.

4.3 Composition des configurations partielles

Initialement vide, l'ensemble des configurations partielles possibles d'un nœud père est construit en parcourant tous les fils un par un. Considérons le cas général (Fig.4.2) :

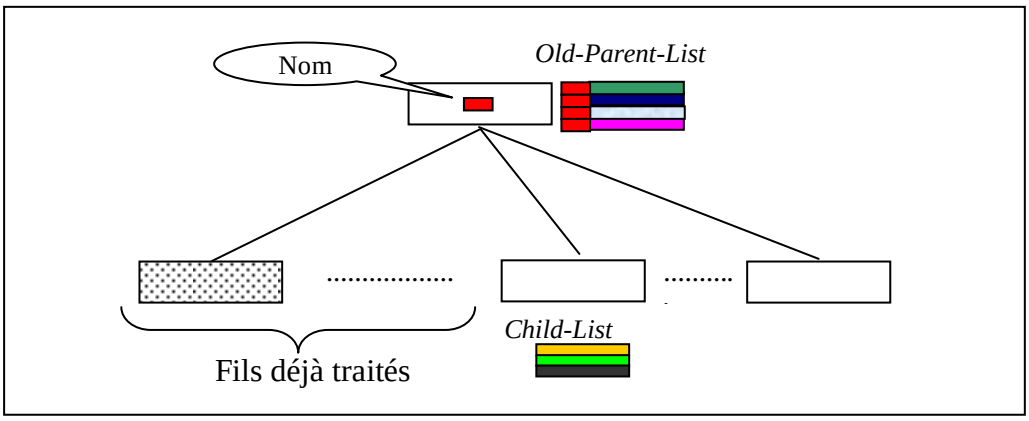


Figure.4.2: Formalisation du problème de la composition

Tel que :

- *Old-Parent-List* : l'ensemble des configurations partielles possibles du père avant le traitement.
- *Child-List* : l'ensemble des configurations partielles possibles du fils en cours de traitement.

Soit *New-Parent-List* l'ensemble des configurations partielles possibles du nœud père après le traitement. En fait, c'est la mise à jour de *Old-Parent-List*. Il est calculé par une combinaison des éléments des deux listes *Old-Parent-List* et *Child-List* en prenant en compte les deux cas de figure suivants :

- Le fils en cours de traitement est supposé comme étant sélectionné (Cas_1).
- Ou bien le contraire. Il est supposé absent (Cas_2).

Ainsi, l'ensemble *New-Parent-List* sera composé de deux parties. Le traitement à réaliser dépend principalement de la sémantique de l'opérateur paternel. Dans ce qi suit, on présentera la solution proposée pour chaque relation.

4.3.1 Traitement des relations de type ''XOR''

Par définition, dans une relation XOR :

''La sélection du père exige la sélection d'un et un seul fils''

Donc, les deux cas sont possibles :

- Cas_1 : Les configurations partielles possibles au niveau du père sont toutes les configurations partielles du fils (*Child-List*) concaténées à la caractéristique du père. Les autres fils doivent être absents.
- Cas_2 : Dans ce cas, les configurations partielles possibles au niveau du père restent celles de l'ancienne liste *Old-Parent-List.*

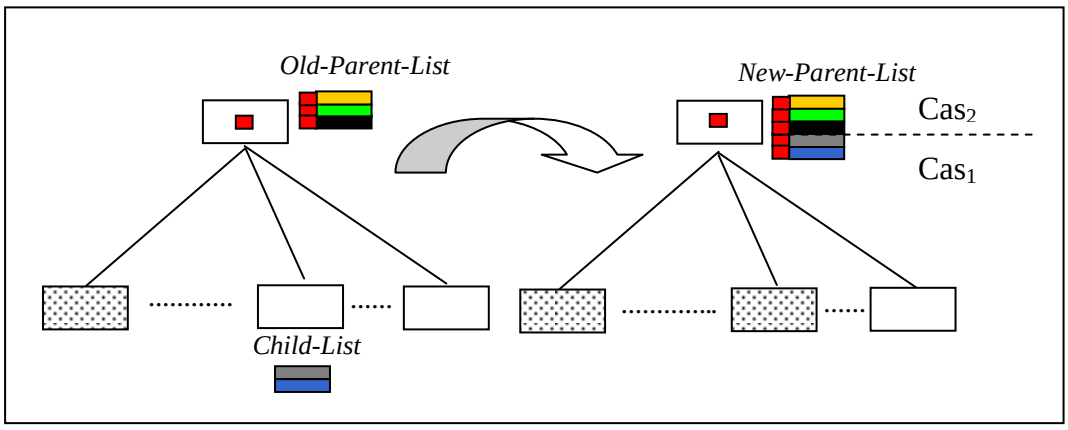


Figure.4.3: Traitement des relations de type ''XOR''

4.3.2 Traitement des relations de type ''OR''

Par définition, dans une relation OR :

''La sélection du père exige la sélection d'au moins un de ses fils''

Donc, les deux cas sont possibles. Le traitement du premier fils visité est différent de celui des autres.

- Traitement du premier fils: Pour ce premier nœud, la présence est obligatoire. Donc, on ne considère que le premier cas (Cas_1). La nouvelle liste des configurations possibles au niveau du père (*New-Parent-List*) sera constituée de toutes les configurations possibles du fils (*Child-List*) concaténées à la caractéristique du père.

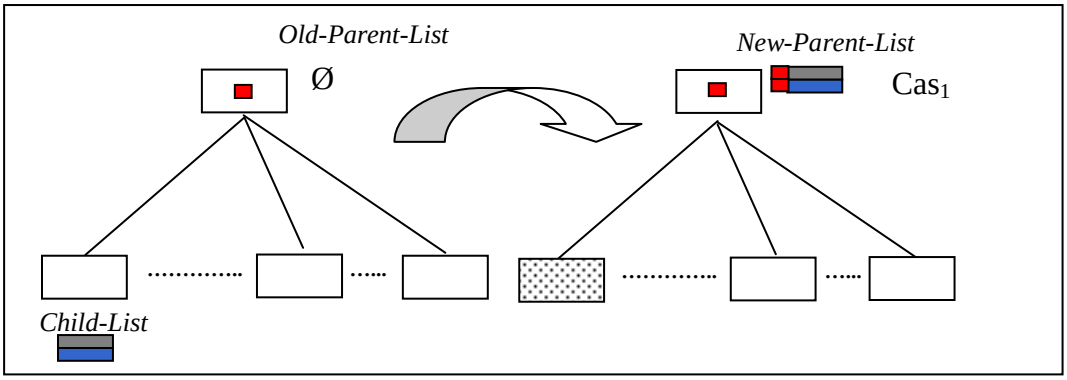


Figure.4.4: Traitement des relations de type ''OR'', premier fils

- Traitement des autres fils: Pour les autres fils, les deux cas sont possibles :
  - Cas_1 : Si le fils en cours de traitement est considéré comme le seul fils sélectionné, les configurations partielles possibles seront toutes les configurations partielles du fils (*Child-List*) concaténées à la caractéristique du père. Dans le cas contraire, les possibilités déjà calculées lors du traitement des fils précédents devront être prise en considération. Donc, on ajoute les combinaisons possibles de la concaténation de chaque élément de *Old-List-Parent* avec chaque élément de *Child-List*.
  - Cas_2 : Les configurations partielles possibles au niveau du père restent celles de l'ancienne liste *Old-Parent-List.*

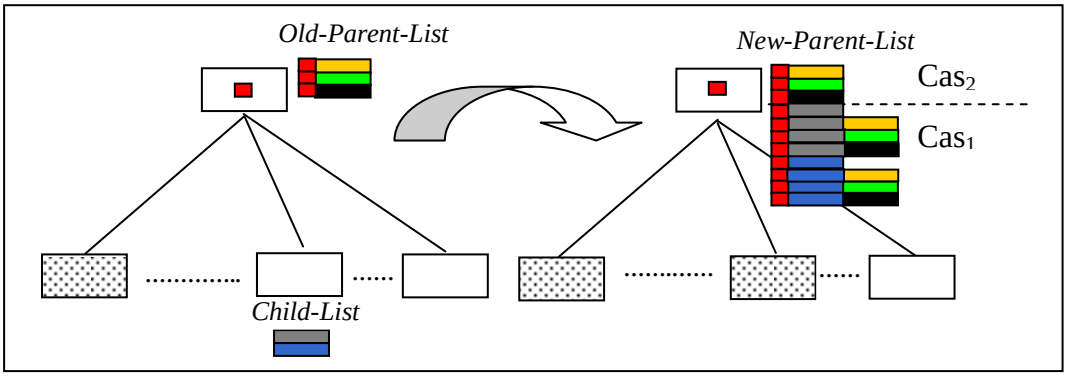


Figure.4.5: Traitement des relations de type ''OR'', les autres fils

Pour le cas des fils obligatoires et facultatifs, on propose de commencer d'abord par le traitement des fils obligatoires, puis on passe à ceux qui sont optionnels.

4.3.3 Traitement des relations de type ‘’Mandatory’’

Par définition, dans une relation Mandatory:

‘’La sélection du père exige celle du fils’’

Pour cette relation, un seul cas valable (Cas_1). La présence du fils traité est obligatoire. Le traitement du premier fils visité est différent de celui des autres.

- Traitement du premier fils: La liste *Old-Parent-List* est initialement vide. Comme la présence du père nécessite la sélection du premier fils, la nouvelle liste *New-List-Parent* doit contenir toutes les configurations partielles du fils (*Child-List)* concaténées avec la caractéristique du père.

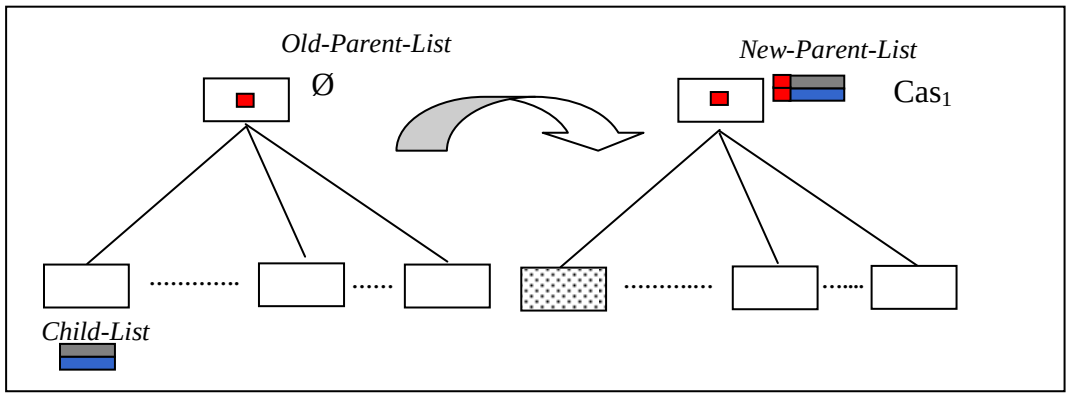


Figure.4.6: Traitement des relations de type ‘’Mandatory’’, le premier fils

- Traitement des autres fils: Même pour le reste des fils, la présence est obligatoire. Ainsi, la nouvelle liste du père *New-List-Parent* est construite par la concaténation de chaque élément de *Old-List-Parent* avec un élément de *Child-List.*

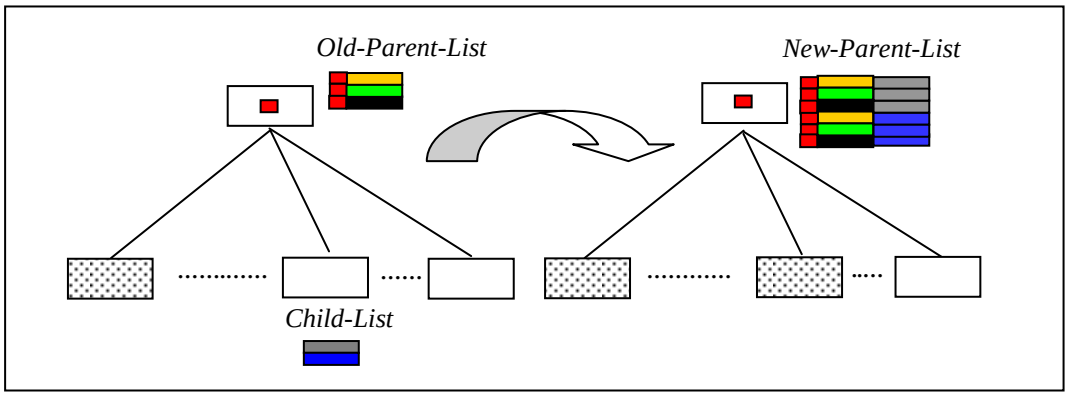


Figure.4.7: Traitement des relations de type ‘’Mandatory’’, les autres fils

4.3.4 Traitement des relations de type ''Optional''

Par définition, dans une relation Optional:

''La sélection du père permet la sélection du fils mais pas nécessairement''

Pour chaque fils traité même le premier, les deux cas de figure sont possibles :

- Cas_1 : Les configurations partielles possibles au niveau du père sont toutes les combinaisons obtenues par la concaténation chaque élément de la liste *Old-Parent-List* avec chaque élément de liste *Child-List*.
- Cas_2 : Les configurations partielles possibles au niveau du père restent celles de l'ancienne liste *Old-Parent-List.*

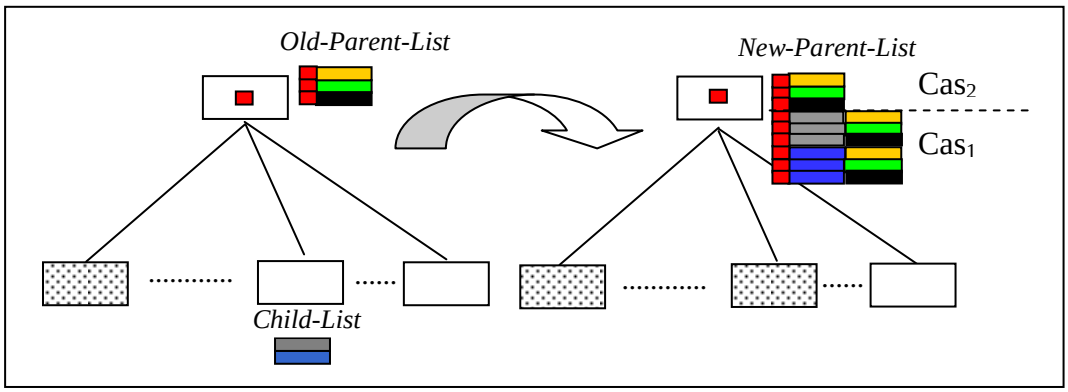


Figure.4.8: Traitement des relations de type ''Optional''

4.4 Grammaire de graphe proposée: 4ème GG

De même que pour le chapitre précédent et afin de réduire le nombre de règles utilisées, nous proposons d'abord translater le diagramme FD en un modèle homogène noté D-Tree. C'est un arbre décoré dans lequel toutes les relations ont la même apparence graphique. Le type est spécifié par un attribut supplémentaire noté *RelationType*. Les méta-modèles ainsi que la grammaire de graphe (1ère GG) utilisés sont les mêmes.

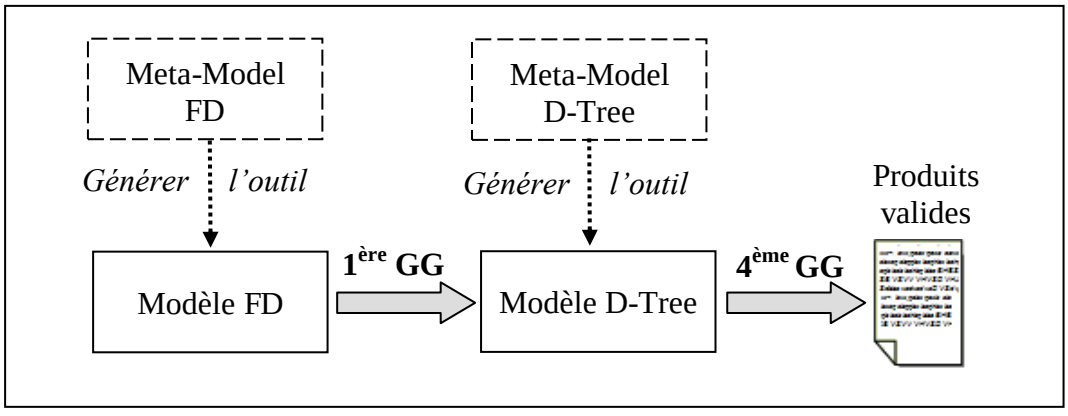


Figure.4.9: Grammaires de graphes proposées pour la construction des configurations valides

Dans la suite, on se limitera uniquement à la présentation de la grammaire permettant de générer les produits valides (4ème GG). Nous procédons en trois étapes :

4.4.1 Construction des produits vérifiant les relations paternelles

Cette première étape sert à la construction des produits vérifiant les relations paternelles. L'exploration ascendante du diagramme FD ainsi que la construction des configurations partielles proposées nécessite une décoration des nœuds par les attributs auxiliaires suivants :

- *Treated:* un booléen utilisé pour spécifier les nœuds déjà traités.
- *Count-Childs :* un entier utilisé pour spécifier le nombre de fils de chaque nœud.
- *Count-TreatedChilds:* un entier utilisé pour compter le nombre de fils traités.
- *Current:* un booléen utilisé pour identifier le nœud père en cours de traitement.
- *Visited-asChild:* un booléen utilisé pour spécifier les fils déjà visités lors du traitement du père courant.
- *Count-VisitedChilds*: un entier utilisé pour compter le nombre de fils visités du père courant.
- *Set-ValidPartialConfs* : liste utilisée pour spécifier l'ensemble des configurations partielles valides au niveau de chaque nœud.

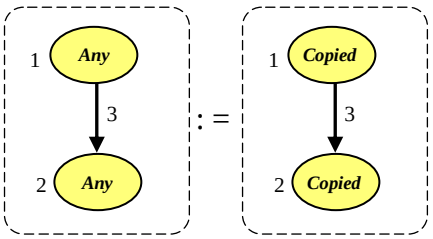
Afin de calculer le nombre de fils (*Count-Childs)* pour chaque nœud, nous proposons de parcourir toutes les relations paternelle en utilisant un attribut temporaire noté *isVisited-CountChilds*.

Pour réaliser cette étape, nous proposons les règles suivantes :

- *La règle N°1* est utilisée pour calculer le nombre de fils de chaque nœud.
- *La règle N°2* est utilisée pour le traitement des feuilles.
- *Les règles N°3, N°4, N°5 et N°6* sont utilisées pour la construction des configurations partielles. Dans cet ordre, elles permettent respectivement le traitement des relations OR, XOR, Mandatory et Optional. A chaque itération, la règle activée (conditions d'application vérifiées) localise un fils qui n'a pas été visité et met à jour l'attribut *Set-ValidPartialConfs* de son père (nœud courant).
- *Règle N°7. SetCurrentParent-AsTraited:* une fois tous les fils du nœud courant visités, cette règle permet de le spécifier comme étant déjà traité.
- *Règle N°8. Select-Parent-NotTraited:* Cette règle est appliquée pour spécifier le prochain père à traiter. Une fois localisé, il sera marqué comme le père courant.

Count-Childs. Priorité 1

LHS RHS

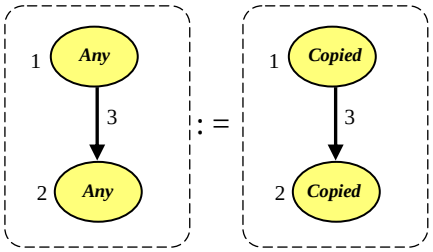


Conditions:
node(3). isVisited-CountChilds = = 0 and
node(3). Relation-Type != "Include" and
node(3). Relation-Type != "Exclude"

Actions:
node(3). isVisited-CountChilds =1
node(1).Count-Childs +=1

Leaves-Processing. Priorité 2

LHS RHS

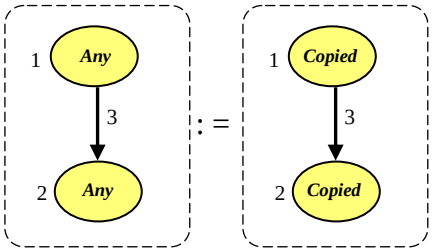


Conditions:
node(2).Treated = = 0 and
node(2). Count-Childs = = 0 and
node(3). Relation-Type != "Include" and
node(3). Relation-Type != "Exclude"

Actions:
node(2).Treated =1
node(1).Count-TreatedChilds +=1
Leaf-List = [ ]
Leaf-List .append (node (2).name)
node(2).Set-ValidPartialConfs = Leaf-List

OR-Relationships-Processing. Priorité 3

LHS RHS

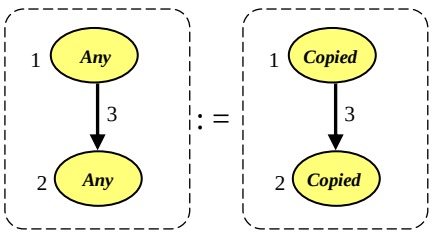


Conditions:
node(1).Treated == 0 and
node(1).Current == 1 and
node(2).Visited-asChild == 0
node(3).Relation-Type = = "OR"

Actions:
```
node(2).Visited-asChild = 1
New-Parent-List =[ ]
Child-List = node(2).Set-ValidPartialConfs
Old- Parent-List = node(1).Set-ValidPartialConfs
If ( Len( Old- Parent-List ) = = 0):
   New- Parent-List .append (node(1).name)
   For child-variant in Child-List:
      NewVariant= node(1).name + ", "+ child-variant
      New- Parent-List .append(NewVariant)
Else
   For parent-variant in Old- Parent-List:
      New- Parent-List .append (parent-variant )
      For child-variant in Child-List:
         NewVariant= parent-variant + "," + child-variant
         New-Parent-List .append(NewVariant))
node(1).Count-VisitedChilds +=1
If (node(1).Count-VisitedChilds== node(1).Count-Childs):
   Old- Parent-List = New-List-Parent
   New- Parent-List =[ ]
   indice = 1
   For parent-variant in Old- Parent-List :
      If indice != 1:
         New- Parent-List .append (parent-variant )
      indice = indice + 1
node(1).Set-ValidPartialConfs = New- Parent-List
```

XOR-Relationships-Processing. Priorité 4

LHS RHS



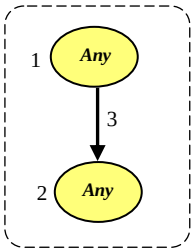
Conditions:
node(1).Treated == 0 and node(1).Current == 1 and node(2).Visited-asChild == 0 and node(3).Relation-Type == "XOR"

Actions:
```
node(2).Visited-asChild = 1
New- Parent-List =[ ]
Child-List = node(2).Set-ValidPartialConfs
Old- Parent-List = node(1).Set-ValidPartialConfs

For parent-variant in Old- Parent-List:
      New- Parent-List.append (parent-variant )

For child-variant in Child-List:
      NewVariant= node(1).name+", "+ child-variant
      New- Parent-List.append(NewVariant)

node(1).Set-ValidPartialConfs = New-Parent-List
```

Figure.4.10: 4ème GG, les Règles N°1-4

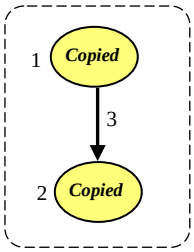
Mandatory-Relationships-Processing. Priorité 5

LHS



: =

RHS



Actions:

node(2).Visited-asChild = 1
New-Parent-List- =[ ]
Child-List = node(2).Set- ValidPartialConfs
Old- Parent-List = node(1).Set- ValidPartialConfs

If (Len(Old- Parent-List) = = 0):
For child-variant in Child-List:
NewVariant=node(1).name+ ", "+ child-variant
New- Parent-List .append(NewVariant)
Else
For parent-variant in Old- Parent-List:
For child-variant in Child-List:
NewVariant= parent-variant +", " +child-variant
New- Parent-List . append(NewVariant))

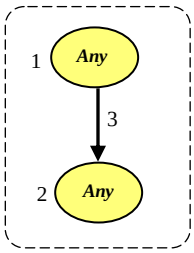
node(1).Set-ValidPartialConfs = New-Parent-List

Conditions:

node(1).Treated == 0 and
node(1).Current == 1 and
node(2).Visited-asChild == 0
node(3).Relation-Type = = "Mandatory"

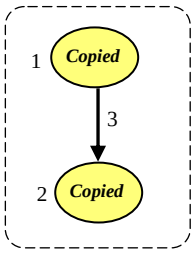
Optional-Relationships-Processing. Priorité 6

LHS



: =

RHS



Actions:

node(2).Visited-asChild = 1
New- Parent-List =[ ]
Child-List = node(2).Set- ValidPartialConfs
Old- Parent-List = node(1).Set- ValidPartialConfs

For parent-variant in Old- Parent-List:
NewVariant= parent-variant
New- Parent-List .append(NewVariant))
For child-variant in Child-List:
NewVariant= parent-variant + ", "+ child-variant
New- Parent-List .append(NewVariant))

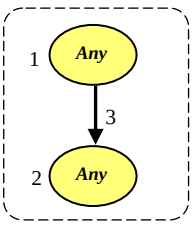
node(1).Set-ValidPartialConfs = New- Parent-List

Conditions:

node(1).Treated == 0 and
node(1).Current == 1 and
node(2).Visited-asChild == 0
node(3).Relation-Type = = "Optional"

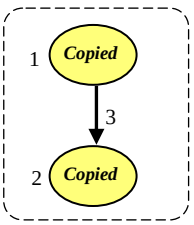
SetCurrentParent-AsTraited. Priorité 7

LHS



: =

RHS



Conditions:

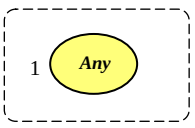
node(2).Treated == 0 and node(2).Current == 1
node(3). Relation-Type != "Include" and
node(3). Relation-Type != "Exclude"

Actions:

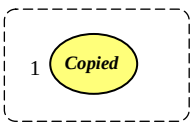
node(2). Traited = 1
node(2).Current = 0
node(1). Count-TreatedChilds +=1

Select-Parent-NotTraited. Priorité 8

LHS



RHS



Conditions:

node(1).Treated == 0 and node(1).Current == 0 and
node(1). Count-TreatedChilds == node(1).Count-Childs

Actions:

node(1).Treated = 0
node(1).Current = 1

Figure.4.11: 4ème GG, les Règles N°5-8

4.4.2 Suppression des produits violant les contraintes d'implication et d'exclusion

La deuxième étape consiste à traiter les contraintes Require et Exlude. Elle agit directement sur la racine en supprimant les configurations non valides. Afin de parcourir tous les liens de ces types, on utilise un attribut temporaire noté *is-Treated.*

Les deux règles proposées sont les suivantes :

- *Règle N°9. Require-Relationships-Processing :* Cette règle est utilisée pour traiter les contraintes de type Require. A chaque itération, elle localise d'abord une relation de ce genre dans le modèle FD. Ensuite, elle met à jour l'attribut *Set-ValidPartialConfs* de la racine en supprimant les configurations incorrectes.
- *Règle N°10. Require-Relationships-Processing :* Cette règle procède avec le même principe, mais elle est utilisée pour le traitement des contraintes de type Exclude.

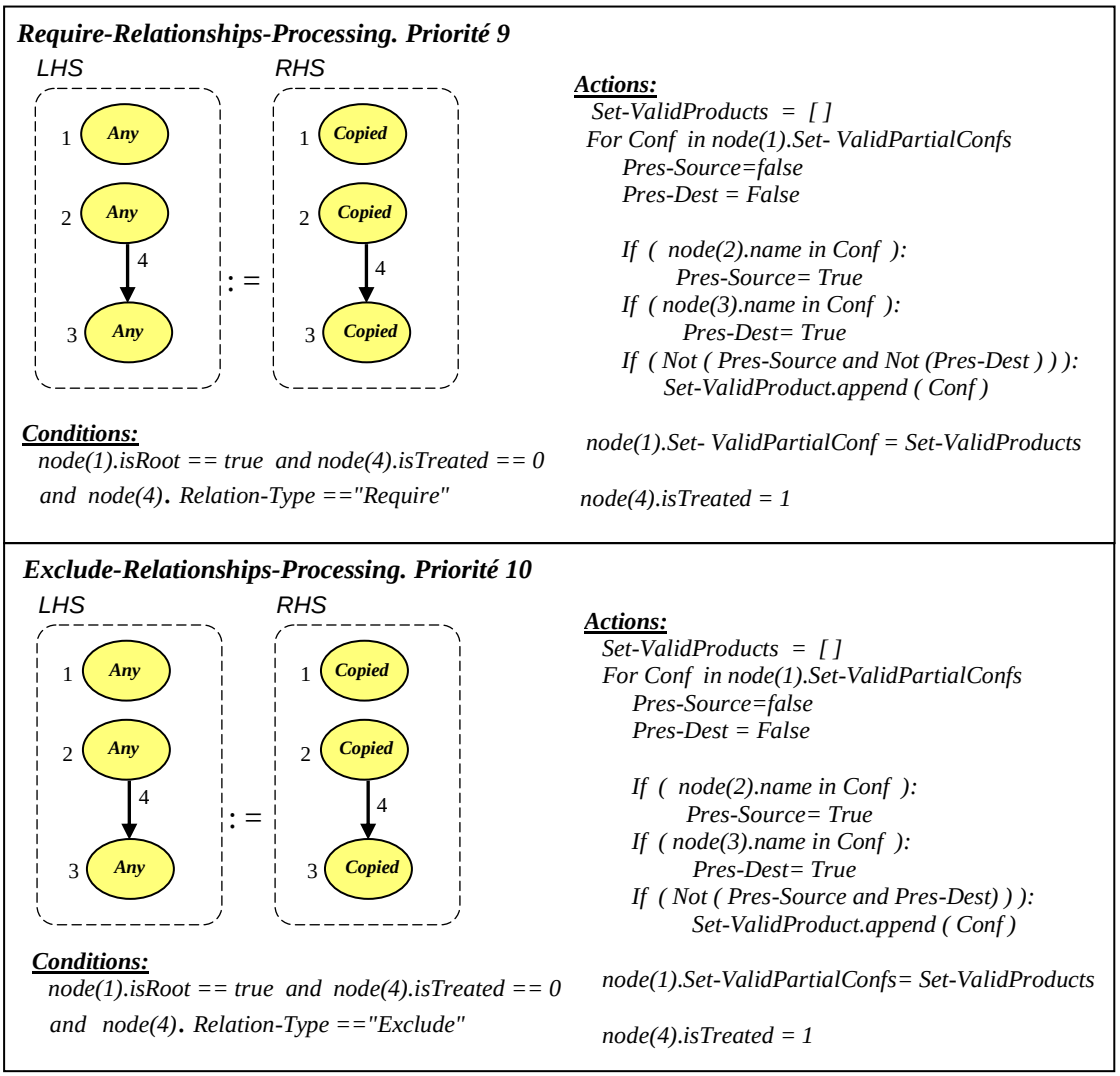


Figure.4.12: 4ème GG, les Règles N°9-10

4.4.3 Génération du fichier texte

Cette étape sert à éditer le fichier texte en parcourant l'attribut *Set-ValidPartialConfs* de la racine. Pour éviter une répétition infinie, une variable globale notée *TextFile-Generated* est utilisée.

Ce traitement est réalisé par la règle suivante :

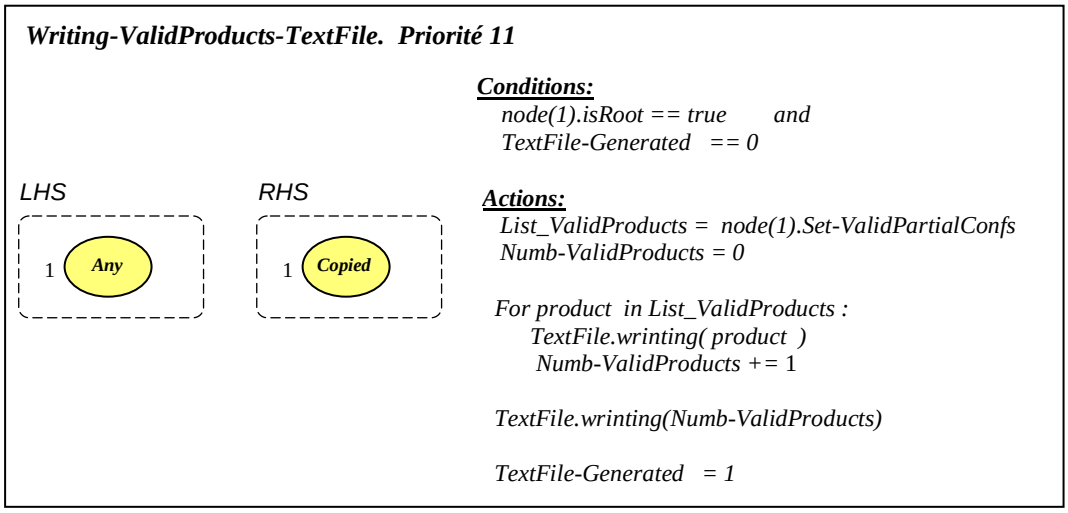


Figure.4.13: 4ème GG, les Règles N°11

4.5 Exemple illustratif

Comme modèle source, on utilise le même exemple présenté dans le chapitre précédent. On commence donc par créer le diagramme FD en utilisant l'environnement visuel (Fig.4.14).

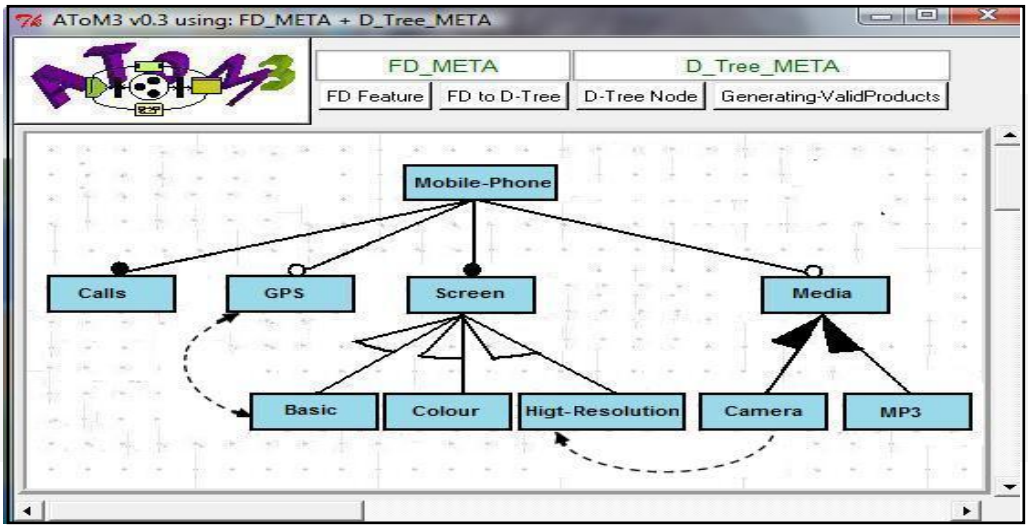


Figure.4.14: Diagramme FD initial

En exécutant les règles de la première grammaire (1ère GG), le diagramme *FD* est translaté en un modèle *D-Tree* équivalent (Fig.4.15).

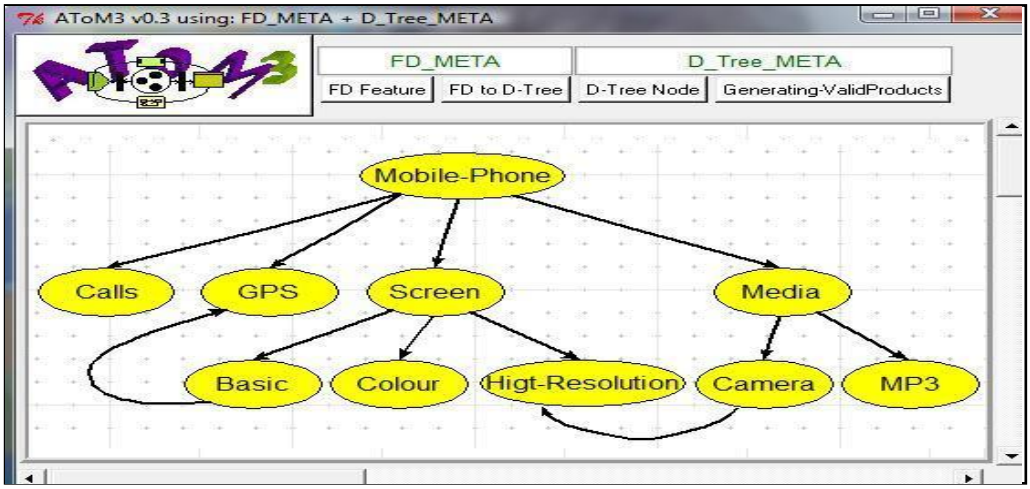


Figure 5.15: Modéle D-Tree géneré

L'exécution de la 4ème grammaire de graphes proposée génère le fichier texte contenant les produits structurellement valides. Pour une bonne illustration de cette approche, on présentera dans suite quelques résultats intermédiaires.

Etape$_1$:

- **Traitement des feuilles:** Pour chaque feuille, la seule configuration partielle possible est la caractéristique elle-même. A titre d'exemple, la Fig.4.16 montre le résultat du traitement du nœud caméra. L'attribut *Set-ValidPartialConfs* ne contient qu'un seul élément " Caméra".

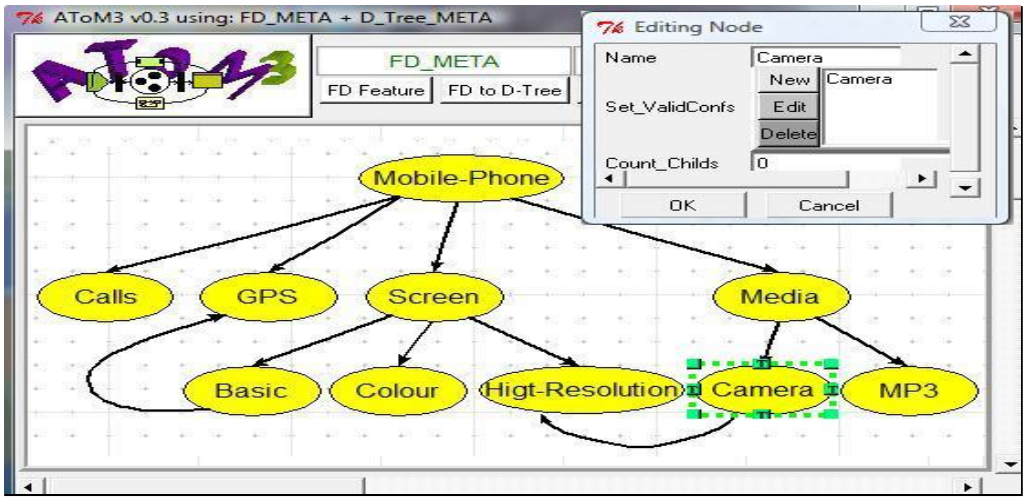


Figure.4.16: Résultat du traitement du nœud ''camera''

- **Traitement des nœuds intermédiaires :** Pour chaque nœud, l'ensemble des configurations partielles possibles est construit en se basant sur les résultats déjà obtenus au niveau des fils et du type de la relation.

  La figure Fig.4.17 montre le résultat du traitement du nœud "Média". Pour ce cas, c'est un traitement d'une dépendance de type OR.

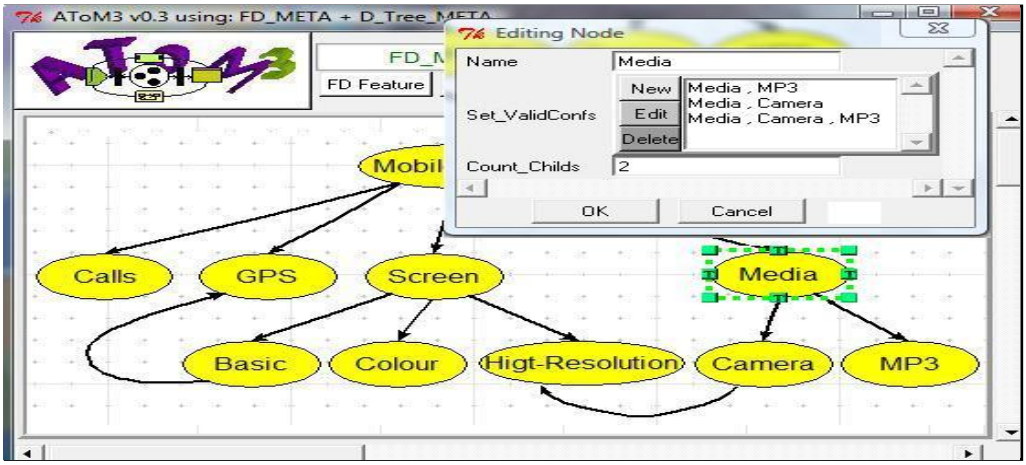


Figure.4.17: Résultat du traitement du nœud ''Media''

L'attribut *Set-ValidPartialConfs* est constitué de trois éléments :

- *Media, Camera*
- *Media, MP3*
- *Media, Camera, MP3*

Pour le nœud "Screen", il s'agit d'une relation XOR. Les configurations partielles possibles calculées sont :

- *Screen, Basic*
- *Screen, Colour*
- *Screen, High-Resolution*

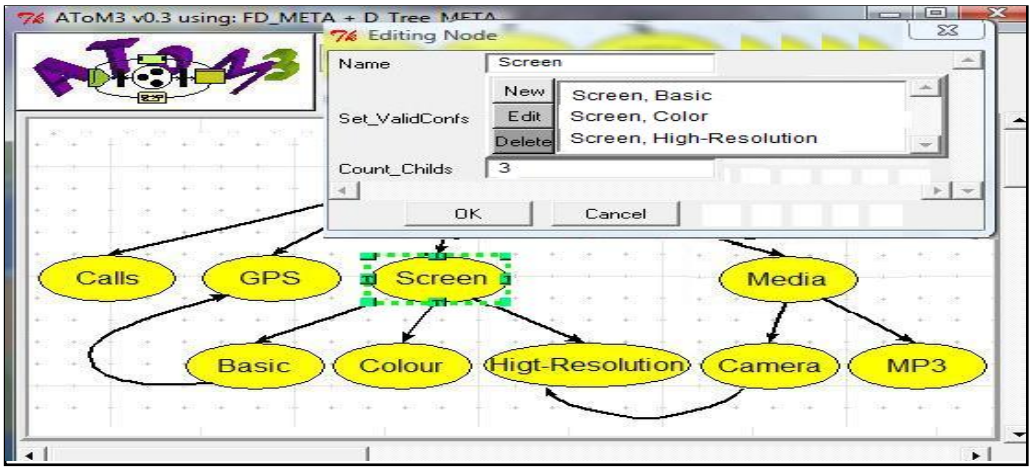


Figure.4.18: Résultat du traitement du nœud ''Screen''

- **Traitement de la racine:** Pour ce nœud, on a deux fils obligatoires et deux autres facultatifs. Les configurations obtenues au niveau de l'attribut *Set-ValidPartialConfs* sont les suivantes :

  - *Mobile-Phone, Calls, Screen, Basic*
  - *Mobile-Phone, Calls, Screen, Basic,Media,MP3*
  - *Mobile-Phone, Calls, Screen, Basic,Media,Camera*
  - *Mobile-Phone, Calls, Screen, Basic,Media,Camera,MP3*
  - *Mobile-Phone, Calls, Screen, Basic, GPS*
  - *Mobile-Phone, Calls, Screen, Basic, GPS , Media, MP3*
  - *Mobile-Phone, Calls, Screen, Basic, GPS , Media, Camera*
  - *Mobile-Phone, Calls, Screen, Basic, GPS , Media, Camera, MP3*
  - *Mobile-Phone, Calls, Screen, Colour*
  - *Mobile-Phone, Calls, Screen, Colour, Media, MP3*
  - *Mobile-Phone, Calls, Screen, Colour, Media, Camera*
  - *Mobile-Phone, Calls, Screen, Colour, Media, Camera, MP3*
  - *Mobile-Phone, Calls, Screen, Colour, GPS*
  - *Mobile-Phone, Calls, Screen, Colour, GPS, Media , MP3*
  - *Mobile-Phone, Calls, Screen, Colour, GPS, Media , Camera*
  - *Mobile-Phone, Calls, Screen, Colour, GPS, Media , Camera, MP3*
  - *Mobile-Phone, Calls, Screen, Higt-Resolution*
  - *Mobile-Phone, Calls, Screen, Higt-Resolution, Media, MP3*
  - *Mobile-Phone, Calls, Screen, Higt-Resolution, Media, Camera*
  - *Mobile-Phone, Calls, Screen, Higt-Resolution, Media, Camera , MP3*
  - *Mobile-Phone, Calls, Screen, Higt-Resolution, GPS*
  - *Mobile-Phone, Calls, Screen, Higt-Resolution, GPS, Media, MP3*
  - *Mobile-Phone, Calls, Screen, Higt-Resolution, GPS, Media, Camera*
  - *Mobile-Phone, Calls, Screen, Higt-Resolution, GPS, Media, Camera, MP3*

  La figure Fig.4.19 donne le résultat obtenu. C'est le dernier nœud traité dans cette étape. Ces configurations représentent donc tous les produits corrects vis-à-vis les relations paternelles spécifiées dans le diagramme FD.

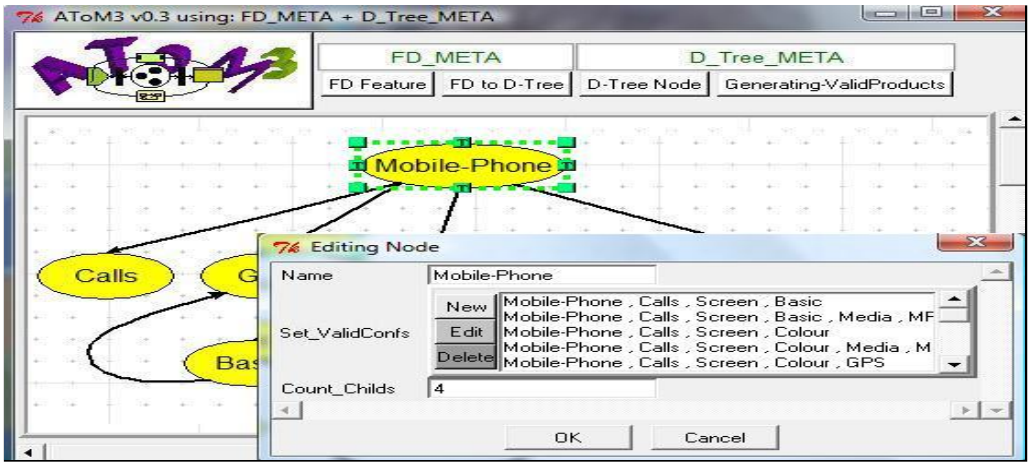


Figure.4.19: Résultat du traitement de la racine

Etape 2: Maintenant, on passe au traitement des contraintes Require et Exclude.

- **Traitement des relations Exclure:** On ne dispose que d'un seul lien de ce type.

  Basic ◄ – – ► *GPS.*

  Les produits:

  - *Mobile-Phone, Calls, Screen, Basic, GPS*
  - *Mobile-Phone, Calls, Screen, Basic, GPS , Media, MP3*

  violent cette exclusion. Les fonctionnalités Basic et GPS sont présents en même temps. Donc, ces deux produits sont supprimés.

- **Traitement des relations Require:** De même, on ne dispose que d'une seule contrainte de ce type :

  Camera – – → *Higt-Resolution*

  Les produits :

  - *Mobile-Phone, Calls, Screen, Basic, Media , Camera*
  - *Mobile-Phone, Calls, Screen, Basic, Media , Camera , MP3*
  - *Mobile-Phone, Calls, Screen, Basic, GPS , Media, Camera*
  - *Mobile-Phone, Calls, Screen, Basic, GPS , Media, Camera, MP3*
  - *Mobile-Phone, Calls, Screen, Colour, Media, Camera*
  - *Mobile-Phone, Calls, Screen, Colour, Media, Camera, MP3*
  - *Mobile-Phone, Calls, Screen, Colour, GPS, Media , Camera*
  - *Mobile-Phone, Calls, Screen, Colour, GPS, Media , Camera, MP3*

  violent cette inclusion. La caractéristique *Camera* est présente, alors que *Higt-Resolution* est absente. Ces produits sont alors supprimés.

Au total, 10 configurations supprimées dans cette étape et n'en reste que 14. La figure Fig.4.19 montre le nœud racine avec le contenu de l'attribut *Set-ValidPartialConfs.*

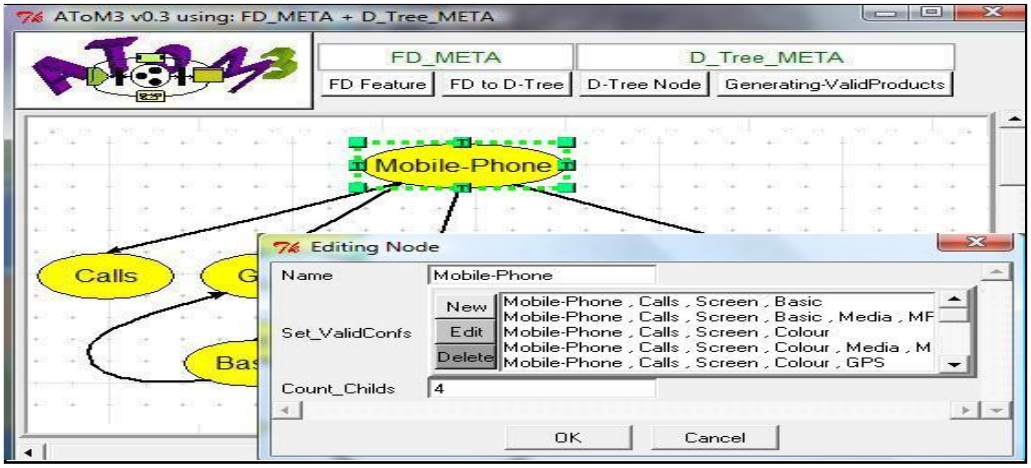


Figure.4.20: Résultat du traitement des contraintes d'implication et d'exclusion

Pour ces configurations, toutes les dépendances du diagramme FD sont respectées. C'est donc l'ensemble des produits structurellement valides pour cet exemple.

Etape 3: Finalement, les produits trouvés sont générés dans le fichier texte (Fig.4.20).

- *Mobile-Phone, Calls, Screen, Basic*
- *Mobile-Phone, Calls, Screen, Basic, Media, MP3*
- *Mobile-Phone, Calls, Screen, Colour*
- *Mobile-Phone, Calls, Screen, Colour, Media, MP3*
- *Mobile-Phone, Calls, Screen, Colour, GPS*
- *Mobile-Phone, Calls, Screen, Colour, GPS, Media, MP3*
- *Mobile-Phone, Calls, Screen, Higt-Resolution*
- *Mobile-Phone, Calls, Screen, Higt-Resolution, Media, MP3*
- *Mobile-Phone, Calls, Screen, Higt-Resolution, Media, Camera*
- *Mobile-Phone, Calls, Screen, Higt-Resolution, Media, Camera, MP3*
- *Mobile-Phone, Calls, Screen, Higt-Resolution, GPS*
- *Mobile-Phone, Calls, Screen, Higt-Resolution, GPS, Media, MP3*
- *Mobile-Phone, Calls, Screen, Higt-Resolution, GPS, Media, Camera*
- *Mobile-Phone, Calls, Screen, Higt-Resolution, GPS, Media, Camera, MP3*

Figure.4.21: Fichier texte généré

C'est le même résultat obtenu dans le chapitre précédent.

4.6 Conclusion

Dans ce chapitre, nous avons proposé une approche permettant le calcul des produits structurellement valides à partir du diagramme FD. Elle est basée principalement sur la composition de configurations partielles.

La technique proposée est basée sur une exploration ascendante du diagramme FD. Ce choix est justifié par la structure arborescente de ce modèle. Pour chaque nœud visité, l'ensemble des configurations partielles possibles est construit par une composition des configurations partielles déjà calculées au niveau de ses fils. Ce traitement dépend principalement de la nature de l'operateur paternel. Ainsi, des combinaisons de caractéristiques vont s'élargir jusqu'à l'obtention de configurations globales et valides d'un point de vue structurel. Ensuite, parmi les produits obtenus, ceux qui violent les contraintes Require et Exclude doivent être supprimés.

Afin d'automatiser tous ces traitements, une grammaire de graphes est développée. La grande difficulté rencontrée réside dans le processus de construction des configurations partielles. Pour chaque relation paternelle, une règle spécifique est définie suivant sa sémantique. Les parties actions sont basées sur des traitements d'ensembles relativement complexes.

Enfin, pour des fins de comparaison, on a utilisé l'exemple illustratif du chapitre précédent. Le fichier texte généré est le même.

Jusqu'ici, les travaux présentés concernent uniquement l'aspect structurel des produits. Dans le prochain chapitre, nous proposons une approche d'analyse comportementale.

Chapitre 05

Vérification Comportementales des Produits : Approche Basée sur la Logique de Réécriture

5.1 Introduction

Dans ce chapitre, nous présentons une nouvelle approche automatique permettant une analyse comportementale dans le cadre des lignes de produits [77]. Les produits sont modélisés en utilisant le formalisme Featured Transition System (FTS) que nous avons jugé comme complet et très expressif. D'une part, le diagramme FD fournit la structure interne des produits en termes de caractéristiques et des dépendances associées. D'autre part, le système de transitions donne une description comportementale détaillée de toute la famille.

Pour l'analyse, nous avons opté pour la logique de réécriture et plus précisément le langage Maude. C'est un outil largement utilisé pour la vérification des systèmes dynamiques. Ce choix est motivé par son caractère formel et la disponibilité d'un Model-Checker automatique.

Ce chapitre commence par un rappel sur le formalisme FTS ainsi qu'un aperçu sur le langage Maude. Ensuite, nous procédons d'abord à la formalisation du système de transitions en termes de règles de réécriture. Une fois l'idée de la translation expliquée, la grammaire de graphe correspondante sera détaillée. Finalement, un exemple illustratif sera présenté en donnant toutes les étapes franchies de la modélisation jusqu'à la vérification des propriétés.

5.2 Featured Transition System (FTS)

FTS [33], est un formalisme conçu pour décrire le comportement combiné de tous les produits d'une famille de produits en utilisant un système de transitions spécifique. En plus des actions ordinaires, les transitions sont étiquetées avec les caractéristiques d'un diagramme FD. Pour bien illustrer ce formalisme, considérons l'exemple présenté dans les figures suivantes (Fig.5.1, Fig5.2). Il s'agit d'un distributeur automatique avec plusieurs variantes. Ils partagent des éléments en commun et se différencient les uns des autres par leurs modes de fonctionnement.

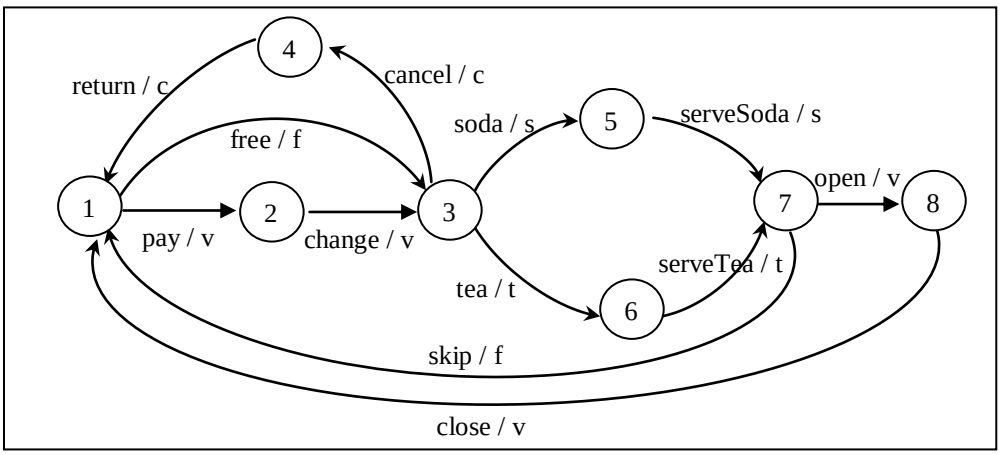


Figure.5.1: Featured Transition System

Ce modèle peut être instancié différemment selon les caractéristiques sélectionnées.

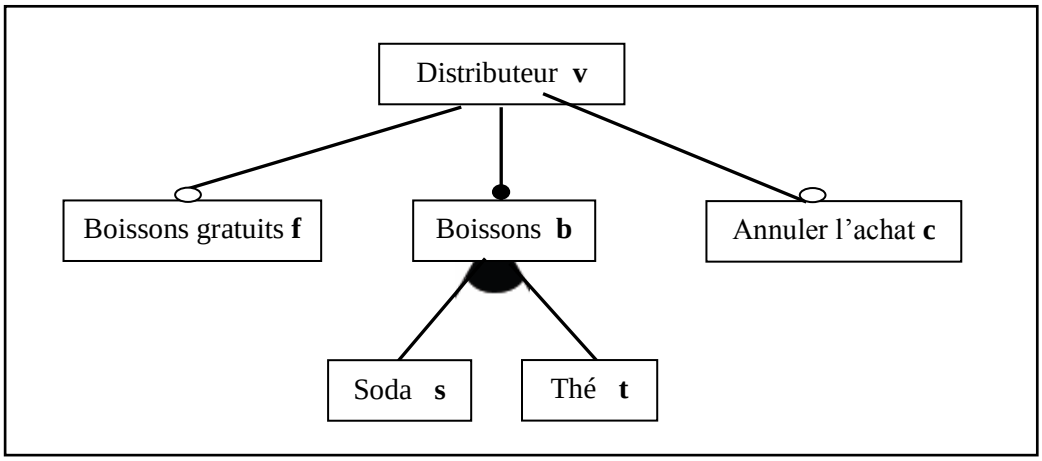


Figure.5.2: Diagramme de caractéristiques Associé

Pour éviter les conflits entre les transitions concurrentes, une relation de priorité particulière a été définie. Une transition $S \rightarrow S_1$ étiquetée par la caractéristique $feat_1$ est prioritaire par rapport à $S \rightarrow S_2$ étiquetée par la caractéristique $feat_2$, si et seulement si $feat_1$ est un descendant de $feat_2$ dans le diagramme FD.

Le comportement individuel de chaque produit est donné par une projection de ses caractéristiques sur le système de transitions. C'est une transformation syntaxique qui consiste à :

- Enlever toutes les transitions associées aux caractéristiques qui ne figurent pas dans ce produit.
- Enlever les transitions les moins prioritaires en cas de conflit. La présence d'une caractéristique substitue celles de ses ascendants.

Le résultat de ce traitement est un système de transitions ordinaire. A titre d'exemple, la figure Fig.5.3 montre les modèles obtenus pour les produits suivants :

- P_1 {v, b, s}: C'est la version de base. Il s'agit d'un distributeur payant ne servant que du soda.
- P_2 {v, b, s, t} : C'est une version améliorée du premier produit. Le distributeur est équipé d'une nouvelle fonctionnalité permettant de servir du thé. Le client a la possibilité de choisir entre soda et thé, mais pas les deux à la fois.
- P_3 {v, b, s, c}: C'est une autre variante du premier produit pour laquelle le client peut annuler l'achat. La fonctionnalité ''cancel'' a été intégrée.
- P_4 {v, b, s, f} : C'est un distributeur de soda gratuit.

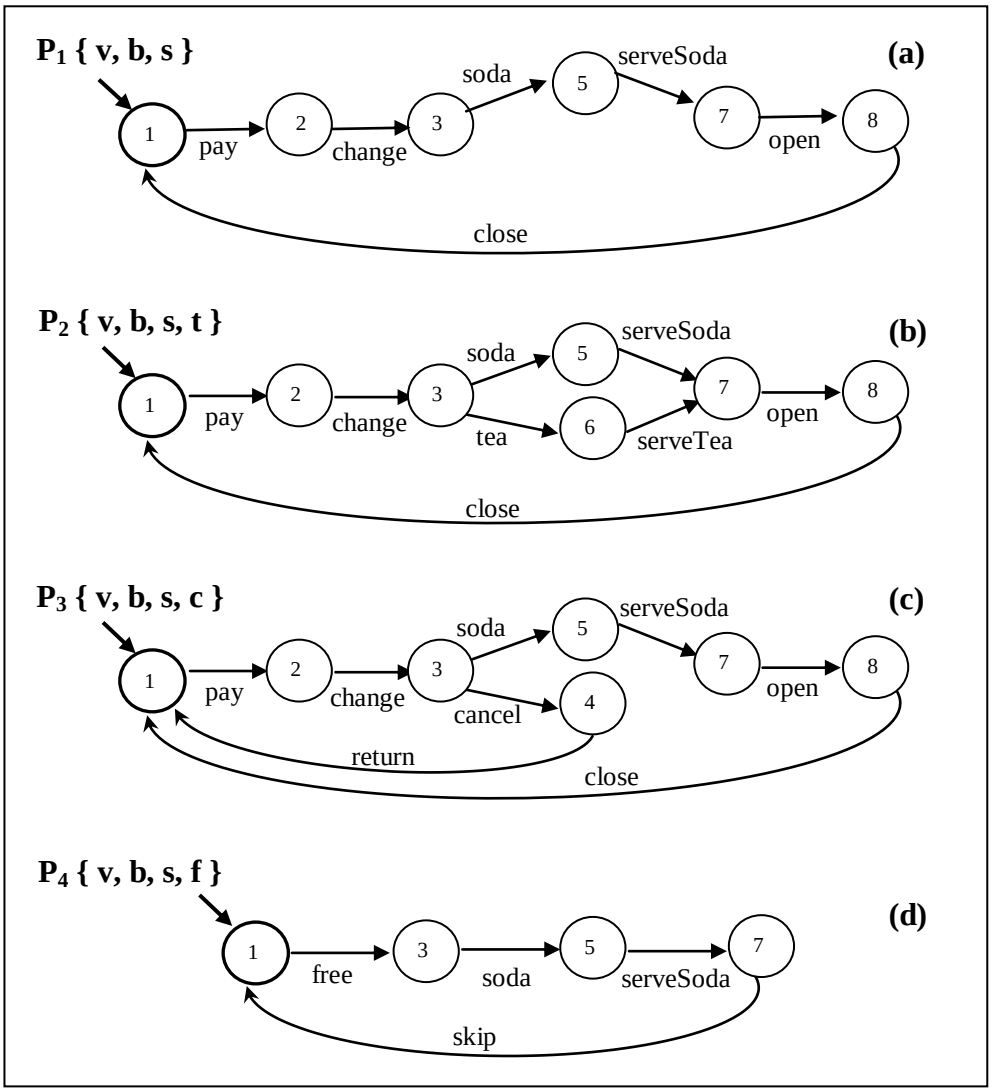


Figure 5.3 : Variantes du distributeur automatique

5.3 Langage Maude

La logique de réécriture est un outil de calcul formel proposé par Meseguer [78]. Elle est largement utilisée pour la simulation et la vérification des systèmes complexes. Pour la spécification du comportement, ce formalisme s'appuie sur logique équationnelle élargie par des règles de réécriture conditionnelles. Chaque règle correspond à une action élémentaire pouvant se produire en concurrence avec d'autres.

Maude [79] est un langage de programmation déclaratif basé principalement sur la théorie de réécriture. Trois types de modules sont définis : fonctionnels, systèmes et orienté-objet. Dans le cadre de cette thèse, nous faisons appel uniquement à des modules fonctionnels et systèmes. Dans ce qui suit, une brève description est présentée.

5.3.1 Les modules fonctionnels

Les modules fonctionnels sont utilisés pour la définition des types de données utilisés ainsi que les opérations qui leur sont associées. Ils sont spécifiés algébriquement par des équations orientées de gauche à droite. Le calcul dans un module fonctionnel est accompli en utilisant les équations comme des règles de réduction jusqu'à ce qu'une forme canonique soit trouvée. le résultat final est unique quelque soit l'ordre d'application de ces équations.

D'un point de vue programmation, un module fonctionnel est un programme équationnel dans le quel l'utilisateur doit définir tous les types utilisés ainsi que les fonctions nécessaires à leur implantation.

5.3.2 Le module système

Un module système décrit la "théorie de réécriture". Ce module augmente les modules fonctionnels par l'introduction des règles de réécriture. Une règle de réécriture conditionnelle est spécifiée en Maude avec la syntaxe :

crl [*Nom_Regle* **] :** *état_source* => *état_destination* **if** *conditions* **.**

Sémantiquement, une règle spécifie une transition concurrente et locale qui peut se produire si la partie gauche de la règle correspond à un fragment de l'état du système. Le déclenchement de cette transition est conditionné par la validation des conditions d'activation.

D'un point de vue programmation, un module système est un programme déclaratif définissant le comportement dynamique du système modélisé.

5.3.3 Simulation et Vérification des propriétés

L'environnement Maude est très flexible pour la simulation. L'interpréteur exécute des programmes de réécriture par une application "arbitraire" des règles de réécriture de gauche à droite sur un état initial donné jusqu'à ce qu'aucune règle ne soit applicable. L'utilisateur peut même limiter le nombre de réécritures en donnant un seuil maximal.

En plus, cet outil est doté d'un Model-Checker [80] très performant (Fig.5.4). Pour analyser le comportement d'un système, deux étapes primordiales sont nécessaires :

- **Une spécification système :** elle engendre :
  - La théorie de réécriture.
  - Un état initial.
- **Une spécification des propriétés :** la propriété à vérifier doit être décrite sous forme d'une formule en logique temporelle linéaire (LTL).

Le Modèle-Checker intégré vérifie automatiquement la validité de la formule LTL en parcourant tous les états accessibles depuis l'état initial spécifié. Si la formule n'est pas valide, un contre-exemple sera généré.

Doté d'une sémantique saine et complète, ce travail montre que le langage Maude est un outil très adapté pour la simulation et l'analyse des modèles FTS [77].

5.4 Spécification des modèles FTS en langage Maude

Dans un premier temps, nous présentons d'abord la spécification proposée des transitions FTS en langage Maude. Pour ce faire, considérons le cas général (Fig.5.1). Soit une transition $Trans_i$ sortante d'un état source $State_i$ vers un état destination $State_j$, la caractéristique associée est notée f_i.

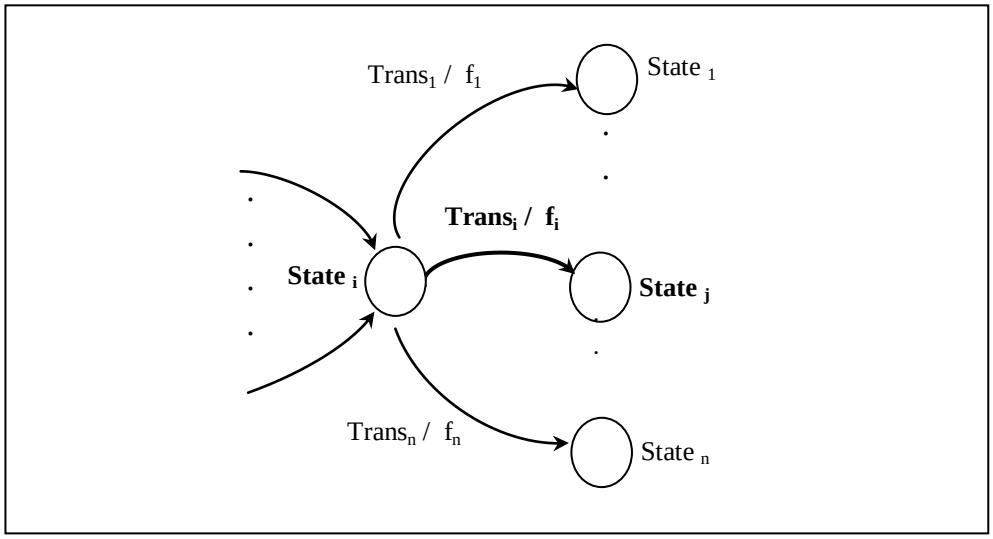


Figure 5.4 : Transition FTS

Soient :

- *SelectFeats* : l'ensemble des caractéristiques sélectionnées.
- *Set_DescFeats_*f_i : l'ensemble des descendants de la caractéristique f_i dans le diagramme FD.

Selon la sémantique du formalisme FTS, la transition $Trans_i$ n'est activée que si les deux conditions suivantes sont vérifiées:

- La caractéristique f_i figure parmi les caractéristiques sélectionnées. Elle fait donc partie de l'ensemble *SelectFeats*.
- Toutes les caractéristiques, qui sont à la fois requises dans les transitions concurrentes et descendantes de f_i dans le modèle FD, sont absentes de *SelectFeats*.

En Maude, cette transition est spécifiée par la règle réécriture suivante :

Crl <***Trans$_i$***> *:* < ***State$_i$***; *SelectFeats; faux* > -> < ***State$_j$*** *; SelectFeats;* ***Flag*** >

if Contains ($\boldsymbol{f_i}$ *, SelectFeats) And*

Not (Contains ($\boldsymbol{f_k}$ *, SelectFeats) and Contains (* $\boldsymbol{f_k}$ *,* ***Set_DescFeats_f_i*** *)) .*

Pour chaque f_k d'une transition concurrente

- *Flag :* une valeur booléenne indiquant si $State_j$ est un état final ou non.
- *Contains:* fonction booléenne définie dans le module *Feature_FunctMod.* Elle permet de vérifier l'appartenance d'un élément à un ensemble de caractéristiques.

Dans le cas d'absence de transitions concurrentes, $Trans_i$ est la seule transition sortante de l'état source. Donc, il suffit de vérifier sa présence dans l'ensemble *SelectFeats.* Ainsi, la règle de réécriture associée sera réduite à la forme suivante :

Crl <***Trans$_i$***> *:* < ***State$_i$***; *SelectFeats; faux* > -> < ***State$_j$*** *; SelectFeats;* ***Drapeau*** >

if Contains ($\boldsymbol{f_i}$ *, SelectFeats)*

En se basant sur cette formulation, la spécification complète d'un modèle FTS en Maude est donnée par les modules suivants :

- *Feature_FunctMod* : C'est un module fonctionnel utilisé pour la manipulation des caractéristiques du diagramme FD. Il contient la déclaration de deux types *Feature* et *ListFeats*, ainsi que la définition des opérations associées.
- *FTS_FunctMod* : C'est un module fonctionnel utilisé pour la manipulation des états du formalisme FTS. Les états classiques sont représentés comme des constantes d'un nouveau type appelé *TsState*. Pour spécifier un état FTS, nous définissons l'opération "<_;_;_>". Le premier paramètre est un état classique. Le deuxième est l'ensemble des caractéristiques sélectionnées. Le dernier est un booléen indiquant s'il s'agit d'un état final ou non .
- *FTS_SysMod* : C'est le module système. Il est constitué des règles de réécriture associées aux transitions du modèle FTS.

Dans la section suivante, nous présentons une approche automatique permettant la génération de ces trois modules en se basant sur des transformations de graphes.

5.5 Génération automatique des modules fonctionnels et système

Dans cette section, nous présentons un outil automatique permettant la génération de la spécification Maude des modèles FTS. Nous commençons d'abord par une présentation des Meta-Modèles utilisés. Ensuite, une description détaillée des grammaires de graphes proposées sera donnée.

5.5.1 Méta-Modélisation

5.5.1.1 FD Méta-Modèle

Dans un modèle FTS, le diagramme FD associé est réduit en un arbre simple. Il n'est utile que pour définir les liens de parenté entre les caractéristiques. Donc, toutes les relations auront la même apparence graphique. Nous proposons le Méta-Modèle suivant :

- ***Une entité "Feature"*:** utilisée pour spécifier les caractéristiques. Chaque caractéristique est identifiée par un attribut *Name.*
- ***Une relation "HasChild"*:** utilisée pour représenter tous les liens du diagramme FD. L'entité de destination est considérée comme un fils de l'entité source. Aucun attribut n'est nécessaire.

5.5.1.2 TS Méta-Modèle

Un modèle TS se compose d'états et de transitions. Ainsi, nous proposons un méta-modèle constitué de:

- ***Une entité "TS-State"***: utilisée pour définir les états. Chaque état possède trois attributs. Son identifiant (*Name*) et deux booléens indiquant respectivement s'il s'agit d'un un état initial (*Initial_state*) ou d'un état final (*Final_State*).
- ***Une relation "TS-Transition"***: utilisée pour spécifier les transitions. Chaque transition possède deux attributs. Le premier est son identifiant (*Name*) et le second spécifie la caractéristique requise (*Required_Feat*).

Maintenant, on procède à la présentation des grammaires de graphes proposées.

5.5.2 Grammaires de graphes proposées

Afin de faciliter la génération des règles de réécriture, nous avons opté pour un calcul préalable de toutes les informations nécessaires. Pour cela, nous proposons d'utiliser trois grammaires de graphes complémentaires (voir Fig.5.5).

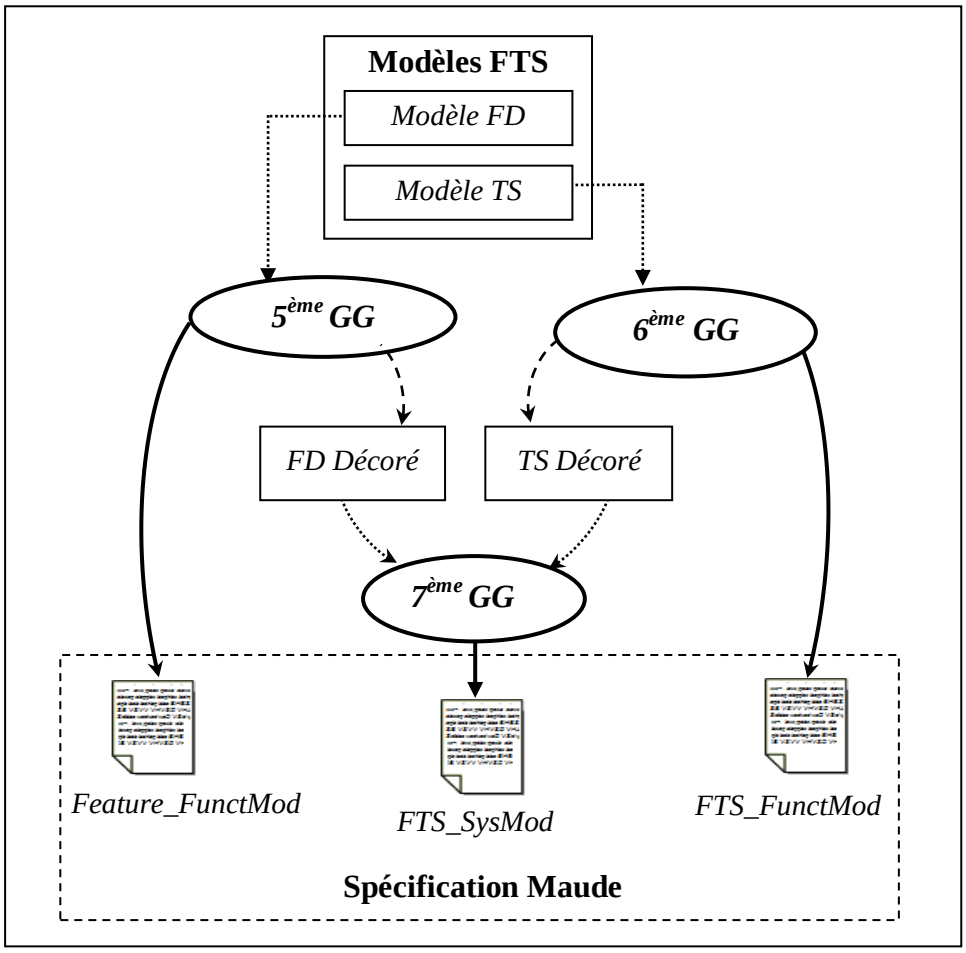


Figure 5.5 : Génération de la Spécification Maude

- La première grammaire génère le module fonctionnel *Feature_FunctMod* et produit un FD décoré. C'est le diagramme FD enrichi par un attribut temporaire au niveau des nœuds noté *Set_DescFeats*. Il est utilisé pour spécifier tous les descendants de chaque caractéristique.
- La deuxième grammaire génère le module fonctionnel *FTS_FunctMod* et produit un TS décoré. C'est le diagramme TS enrichi par un attribut temporaire au niveau des transitions noté *Set_ReqFeatCTs*. Il est utilisé pour spécifier toutes les caractéristiques requises dans ses transitions concurrentes.
- Enfin, la dernière grammaire génère le module de système *FTS-SysMod* contenant les règles de réécriture.

L'application de ces trois grammaires de graphes conduit à la génération d'une spécification textuelle en langage Maude. Il s'agit donc d'une transformation de type modèles vers texte.

5.5.2.1. Génération du module fonctionnel *Feature-FunctMod et du modèle FD-Décoré*: 5ème GG

Cette grammaire agit uniquement sur le diagramme FD. En analysant la structure du module fonctionnel *Feature_FunctMod* (Fig 5.6), on remarque qu'il est constitué de trois parties :

- Partie$_1$: Elle sert à spécifier l'entête du ficher ainsi que les sortes utilisées. C'est du code invariant situé tout au début de ce module. Il sera édité directement par l'action initiale de la grammaire.
- Partie$_2$: Elle sert à spécifier les caractéristiques du diagramme FD. C'est du code qui change selon le modèle traité. Il sera généré lors du calcul de l'attribut *Set_DescFeats.*
- Partie$_3$: Elle sert à déclarer les opérations ainsi que les équations associées. De même, c'est du code invariant mais situé à la fin de ce module. Donc, il sera édité directement par l'action finale de la grammaire.

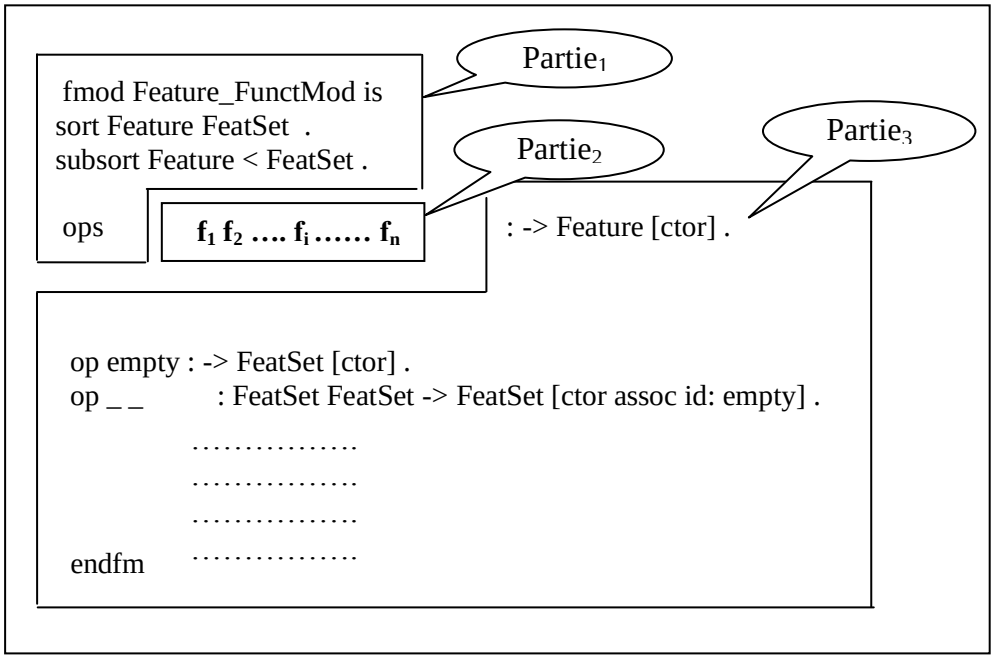


Figure 5.6 : Le module fonctionnel Feature_FunctMod

Le calcul de l'attribut *Set_DescFeats* nécessite un parcours de toutes les caractéristiques. Pour chaque nœud, cet ensemble est constitué de ses descendants. De ce fait, on a opté pour une exploration ascendante du diagramme FD en utilisant un attribut temporaire noté *Treated*. Il s'agit d'un traitement itératif, où à chaque itération :

- Le système de réécriture localise un nœud pour lequel tous les fils sont déjà traités.
- Le système de réécriture met à jour l'attribut *Set_DescFeats* du père en lui ajoutant tous les descendants de ses fils ainsi que sa caractéristique (*Name*).

La génération de la deuxième partie ($Partie_2$) du module fonctionnel *Feature_FunctMod* est réalisée en même temps que le traitement des caractéristiques du modèle FD. Pour chaque nœud visité, l'attribut *Name* est édité dans le fichier texte.

Le traitement commence donc par les feuilles. Ce type de nœuds est repéré par l'absence de fils. Ensuite, on passe au traitement des nœuds intermédiaires jusqu'à la racine. Ici, la sélection de chaque nœud est conditionnée par le traitement auparavant de tous ses fils. Pour cela, nous proposons d'ajouter un compteur de fils traités noté *Count_TreatedChilds.*

La grammaire proposée est constituée de :

- Une action initiale dont le rôle est la création du fichier texte ainsi que l'édition de la première partie du module fonctionnel *Feature_FunctMod*.
- Deux règles de transformations (Fig. 4.7).
- Une action finale dont le rôle est d'éditer la troisième partie du module fonctionnel *Feature_FunctMod* dans le fichier texte avant de le fermer.

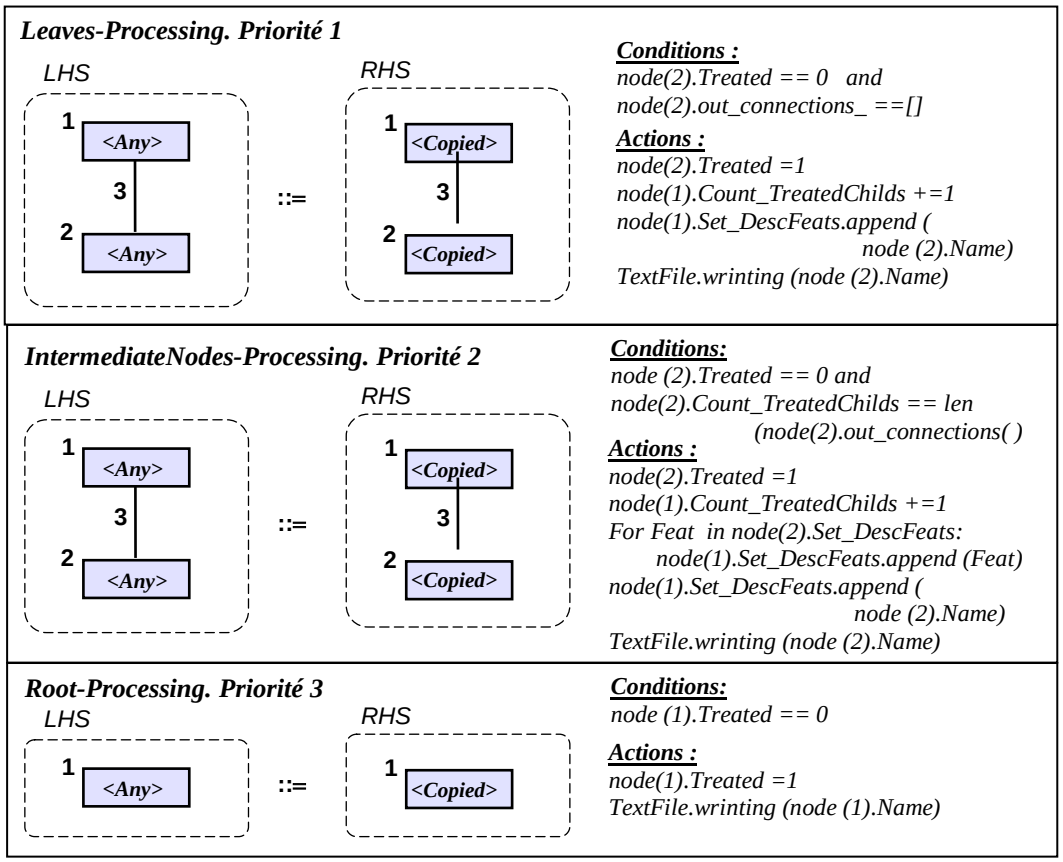


Figure 5.7 : 5ème GG, les règles N°1-3

Règle N°1. Leaves-Processing : Cette règle est appliquée pour le traitement des feuilles.

Règle N°2. IntermediateNodes-Processing **:** Cette règle est appliquée pour le traitement des nœuds intermédiaires.

Règle N°3. Root-Processing **:** Cette règle est appliquée pour le traitement de la racine.

5.5.2.2. Génération du module fonctionnel *FTSFunctMod et du modèle FTS-Décoré*: 6ème GG

Cette grammaire agit uniquement sur le diagramme TS. De même, le module fonctionnel *FTS_FunctMod* est constitué de trois parties (voir Fig. 5.8).

- $Partie_1$: Elle sert à spécifier l'entête du ficher et à déclarer les sortes utilisées. Elle sera éditée directement par l'action initiale de la grammaire.
- $Partie_2$: Elle sert à spécifier les états classiques du modèle TS. C'est du code qui change selon le modèle traité. Il sera généré lors du calcul de l'attribut *Set_ReqFeatCTs.*
- $Partie_3$: Elle sert à déclarer l'opération utilisée pour spécifier les états FTS et sera générée dans l'action finale de la grammaire.

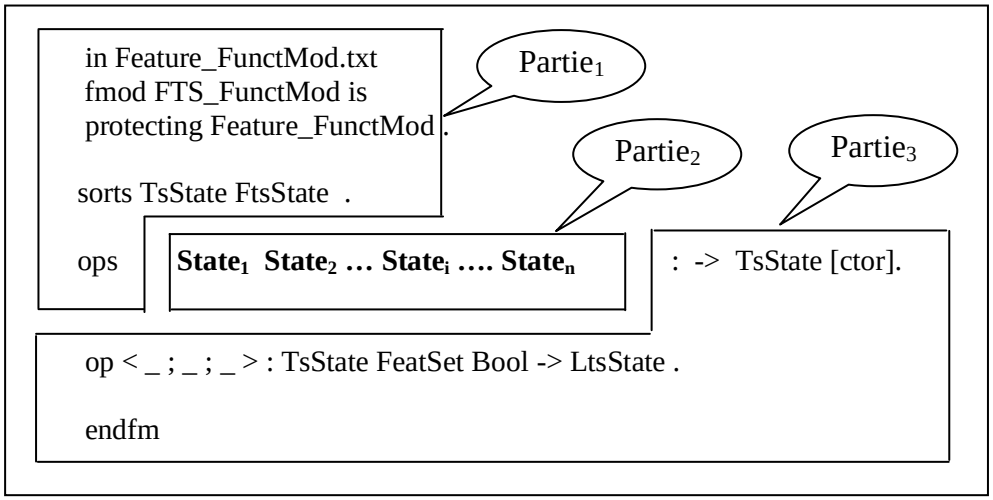


Figure 5.8: Le module fonctionnel FTS_FunctMod

Cette grammaire de graphes est basée sur un parcours des états du modèle TS. Pour chaque nœud visité, l'attribut *Name* est d'abord édité dans le fichier texte afin de générer la deuxième partie du module fonctionnel *FTS_FunctMod.* Ensuite, les transitions sortantes sont traitées à tour de rôle. Pour chaque transition, l'attribut *Set_ReqFeatCTs* est calculé en visitant toutes ses transitions concurrentes. Afin de réaliser ce traitement, les attributs temporaires utilisés sont les suivants :

- Pour les états *: Current* et *Visited*. Le premier est utilisé pour identifier l'état en cours de traitement, alors que le second sert à spécifier les états déjà visités.
- Pour les transitons : *Current, Treated* et *FeatureInserted*. *Current* est utilisé pour localiser la transition en cours de traitement. *Treated* est utilisé pour spécifier les transitions déjà traitées. *FeatureInserted* est utilisé pour savoir si la caractéristique requise dans cette transition a été préalablement ajoutée à l'ensemble *Set_ReqFeatCTs* de la transition courante.

La grammaire proposée est constituée de :

- Une action initiale dont le rôle est la création du fichier texte ainsi que l'édition de la première partie du module fonctionnel *FTS_FunctMod*.
- Sept règles de transformations (Fig. 5.9 et Fig.5.10).
- Une action finale dont le rôle est d'éditer la troisième partie du module fonctionnel *FTS_FunctMod* dans le fichier texte avant de le fermer.

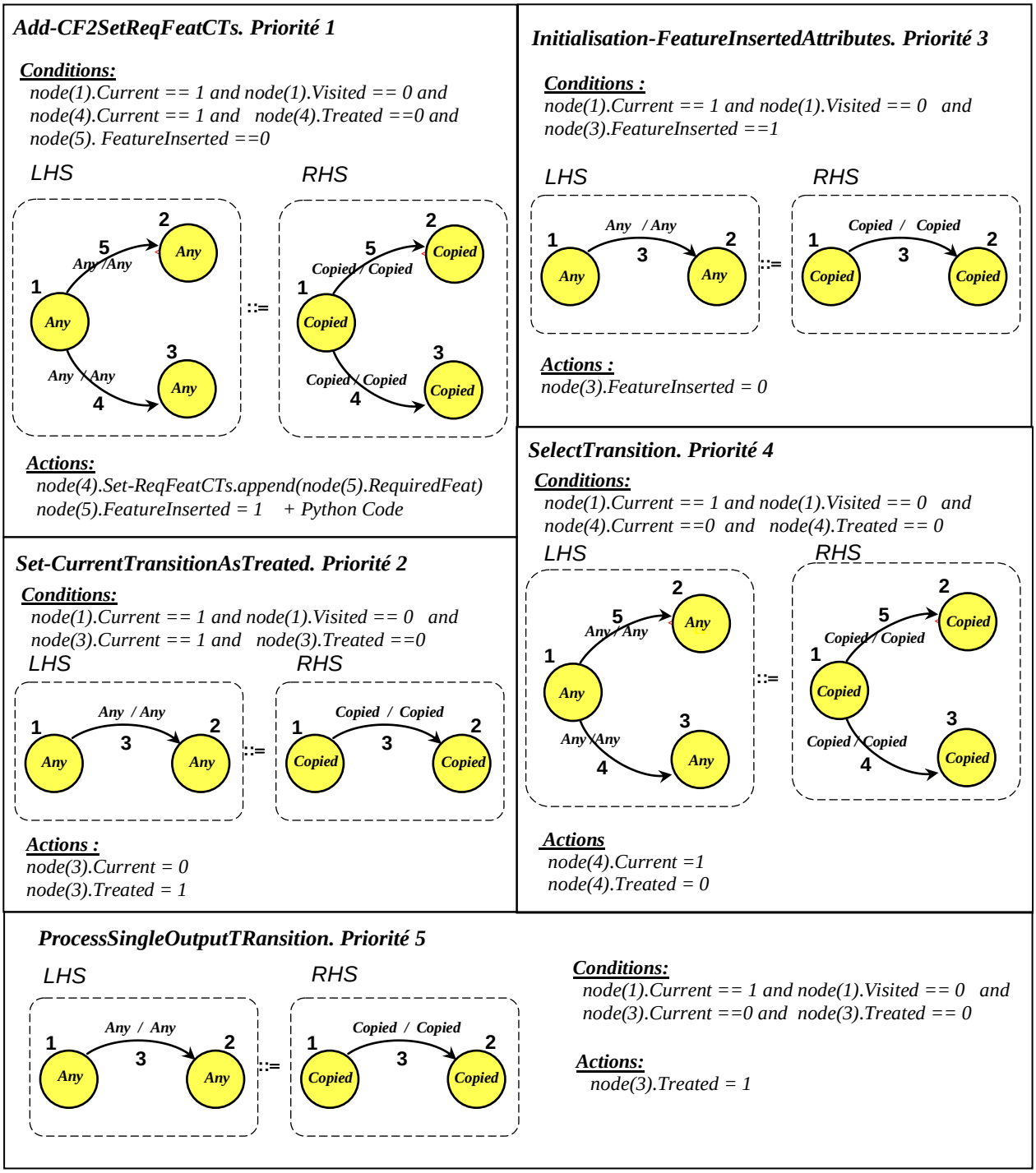


Figure 5.9: 6ème GG, les règles N°1-5

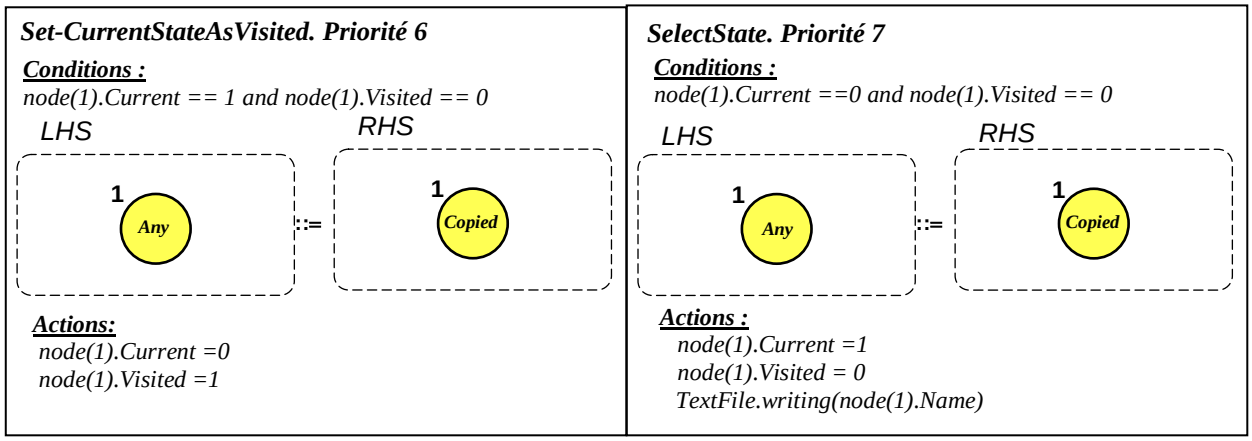


Figure 5.10: 6ème GG, les règles N°6-7

***Règle N°1. Add-CF2SetReqFeatCTs* :** Cette règle est appliquée pour localiser une transition concurrente afin de mettre à jour l'attribut *Set_ReqFeatCTs* de la transition courante.

***Règle N°2. Set-CurrentTransitionAsTreated* :** une fois que toutes les transitions concurrentes sont visitées, cette règle marque la transition courante comme étant déjà traitée.

***Règle N°3. Initialisation-FeatureInsertedAttributes* :** Cette règle est appliquée pour réinitialiser l'attribut *FeatureInserted* de toutes les transitions sortantes de l'état courant. Cela permet le traitement de la prochaine transition.

***Règle N°4. SelectTransition* :** Cette règle est appliquée pour localiser la prochaine transition sortante à traiter. Une fois identifiée, elle sera marquée comme la transition courante afin de procéder à son traitement par les règles *N°1*, *N°2* et *N°3*.

***Règle N°5. ProcessSingleOutputTransition* :** Cette règle traite le cas où la transition en cours de traitement est la seule transition sortante de l'état courant. L'attribut *Set_ReqFeatCTs* reste vide.

***Règle N°6. Set-CurrentStateAsVisited* :** une fois que toutes les transitions sortantes de l'état courant sont traitées, cette règle marque cet état comme étant déjà visité.

***Règle N°7. SelectState* :** Cette règle est appliquée pour localiser un nouveau état à visiter. Une fois identifié, il sera marqué comme étant l'état courant. En même temps, l'attribut *Name* est inséré dans le fichier texte afin de produire la deuxième partie du module fonctionnel *FTS_FunctMod*.

5.5.2.3. Génération du module système *SystemMod*: 7ème GG

Cette grammaire agit sur les deux modèles *FD-Décoré* et *TS-Décoré* afin de générer le module système *FTS_SysMod*. Selon la figure Fig.5.11, ce fichier est composé de deux parties. La première partie est standard. Elle est utilisée pour spécifier l'entête du ficher. Elle sera donc éditée directement par l'action initiale. Par contre, la seconde partie est constituée des règles de réécritures spécifiant les transitions FTS en langage Maude.

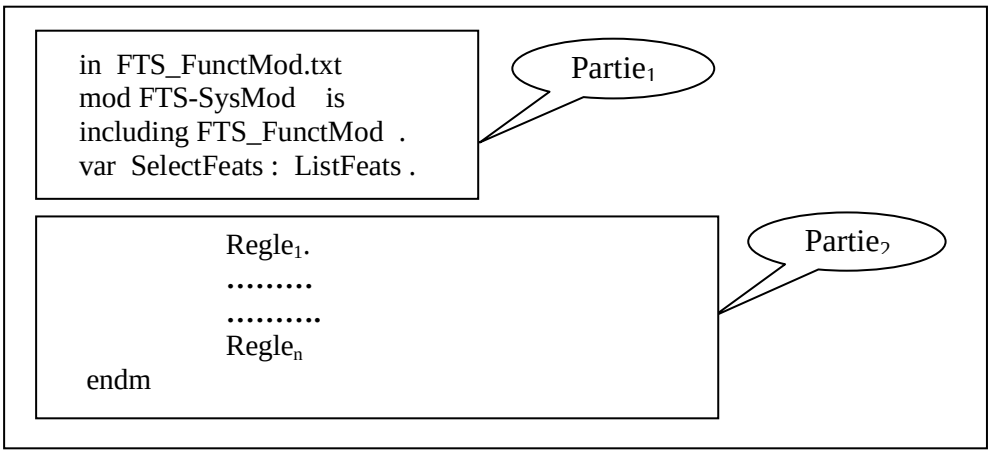


Figure 5.11: Le module système FTS_SysMod

Pour générer la deuxième partie d'une manière automatique, nous proposons un traitement des transitons du modèle FTS à tour de rôle. Ce parcours est réalisé en utilisant deux attributs temporaires *Current et Treated.* Le premier permet de spécifier la transition en cours de traitement, alors que le second est utilisé pour repérer les transitions déjà traitées. Pour chaque transition visitée, la règle de réécriture correspondante sera éditée selon la formalisation présentée dans la section 5.4. Elle est constituée de deux segments (Fig.5.12):

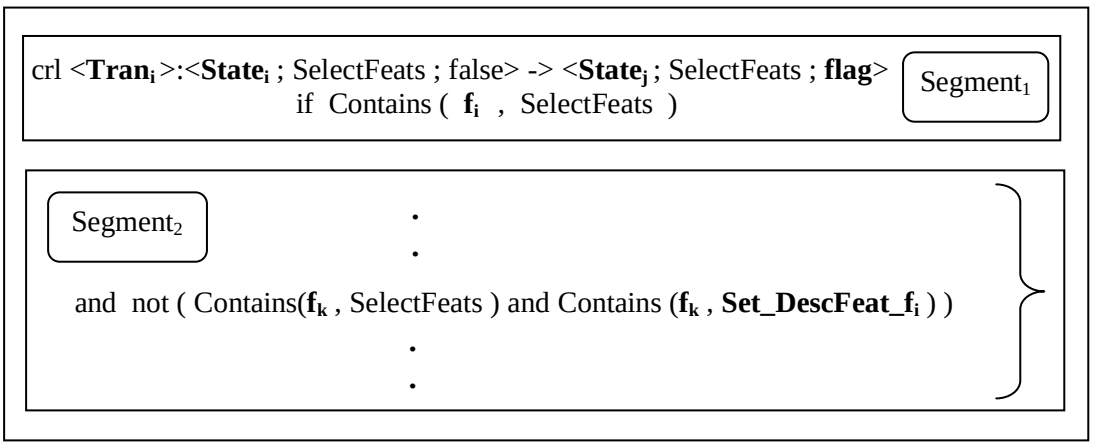


Figure 5.12: Structure d'une transition FTS en Maude

- Segment$_1$: utilisé pour spécifier l'état source, l'état destination ainsi que le code vérifiant la présence de la caractéristique requise. Il est généré de la même manière pour toutes les transitions.
- Segment$_2$: utilisé pour spécifier les conditions d'activation en tenant compte de la priorité. Ce code dépend de la transition en cours de traitement. Il est généré en parcourant les éléments f_k de l'attribut *Set_ReqFeatCTs*. En ce qui concerne les éléments de l'ensemble *Set_DescFeats_*f_i représentant les descendants de f_i , il suffit de localiser le nœud correspondant du diagramme FD. C'est bien le contenu de l'attribut *Set_DescFeats* qui sera édité. En cas d'absence des transitions concurrentes, ce segment sera ignoré.

Pour chaque transition, l'édition de ces deux segments est réalisée séparément en utilisant un autre attribut temporaire noté *Step*. La grammaire de graphe proposée est la suivante.

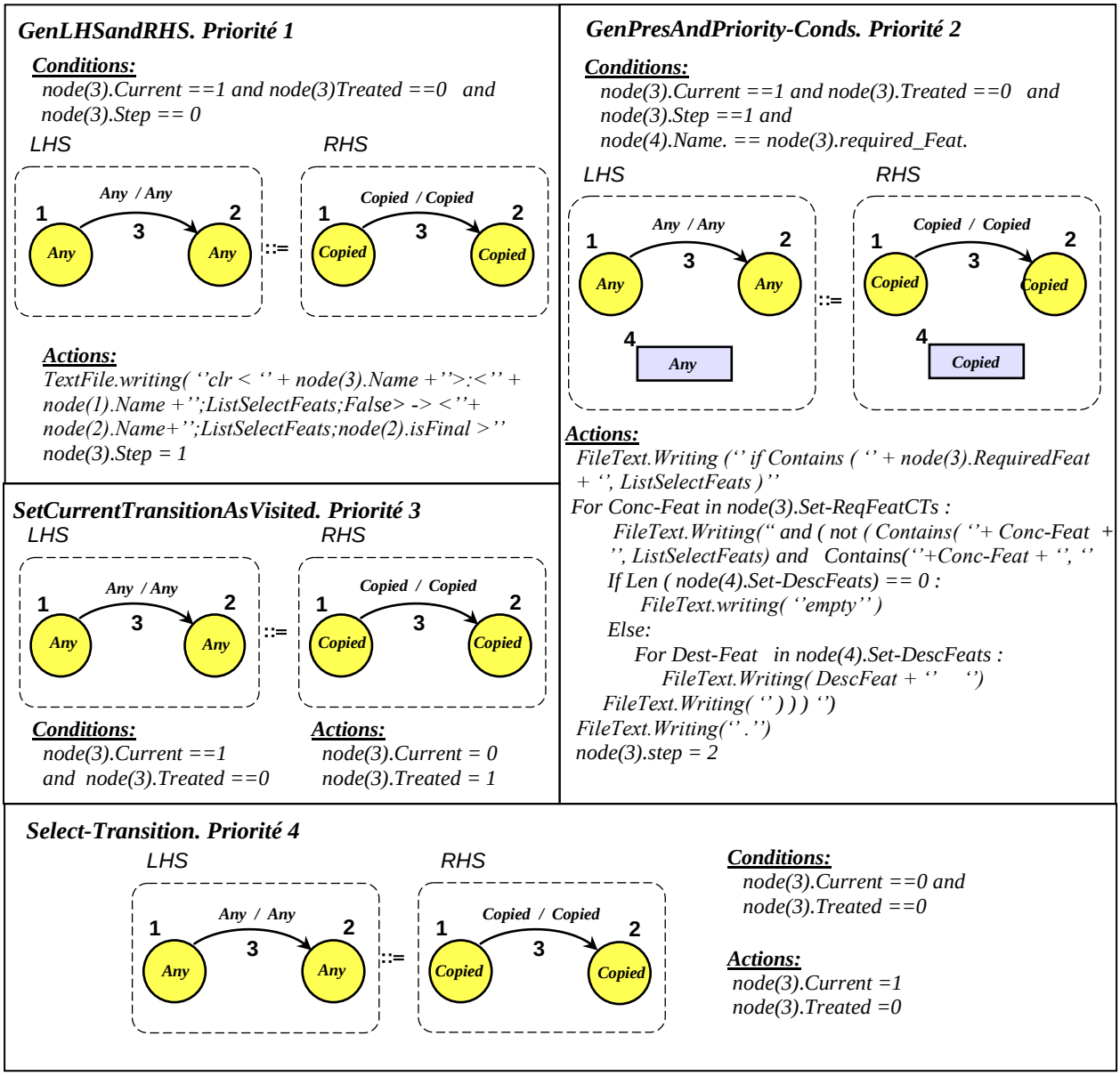


Figure 5.13: 7ème GG, les règles N°1-4

Règle N°1. Gen_LHS&&RHS : Cette règle est appliquée pour générer le premier segment ($Segment_1$) de la règle de réécriture dans le fichier texte. Cette partie du code contient le nom de la transition courante, l'état source, l'état de destination et l'indicateur booléen. Ce dernier est ajusté à la valeur de l'attribut *Final_State* de l'état $State_j$.

Règle N°2. Gen_ActivationConds : Cette règle est appliquée pour générer le deuxième segment de la règle ($Segment_2$). Elle n'est activée qu'en présence de transitions concurrentes. Elle procède comme expliqué précédemment.

Règle N°3. SetCurrentTransitionAsVisited : Une fois sa spécification Maude est générée, cette règle marque la transition courante comme étant déjà traitée.

***Règle N°4. Select-Transition* :** Cette règle est appliquée pour localiser la nouvelle transition à traiter. Une fois identifiée, elle sera marquée comme étant la transition courante.

5.6 Exemple illustratif

Pour illustrer cette approche, considérons l'exemple du distributeur automatique présenté au début de ce chapitre (Fig.5.1).

5.6.1 Création des modèles sources

Une fois les méta-modèles des formalismes *FD* et *TS* définis, AToM^3 génère automatiquement un éditeur graphique permettant de manipuler les entités de ces deux formalismes. Il permet également de spécifier les trois grammaires de graphes proposées.

- Comme modèles sources, on commence d'abord par la création du diagramme *FD* tel que présenté dans la figure Fig.5.14.

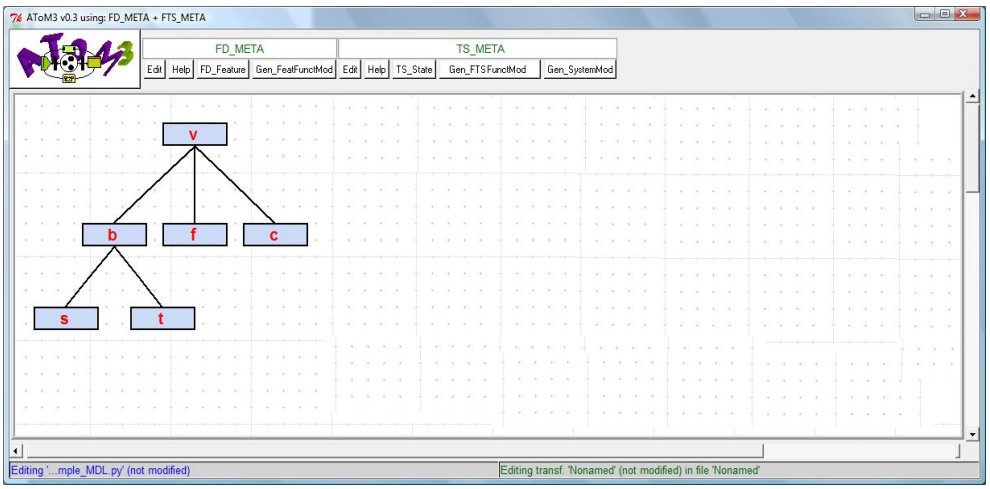


Figure 5.14: Création du diagramme FD

- Ensuite, on procède à la création du modèle TS comme présenté dans la figure Fig.5.15.

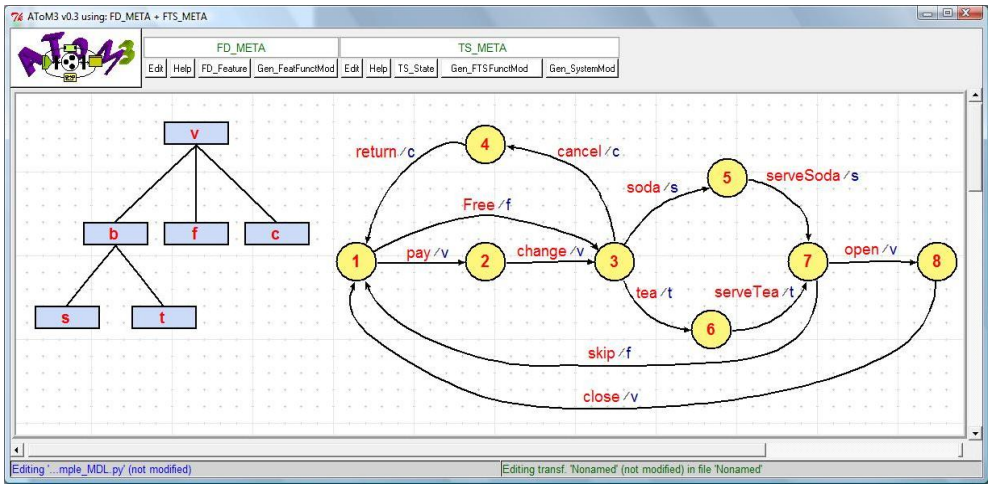


Figure 5.15: Création du modèle TS

5.6.2 Génération de la spécification Maude

La spécification Maude est générée automatiquement en appliquant les trois grammaires de graphes.

- La première grammaire *Gen_FeatFunctMod* s'applique sur le diagramme FD et génère :
  - Le module fonctionnel *Feature_FunctMod* (Fig.5.16).

```
fmod Feature_FunctMod is
sort Feature FeatSet  .
subsort Feature < FeatSet .
ops  v  b  s  c  t  f  : -> Feature [ctor] .

op empty : -> FeatSet [ctor] .
op __          : FeatSet FeatSet -> FeatSet [ctor assoc id: empty] .
op isEmpty : FeatSet -> Bool .
op size        : FeatSet -> Nat .
op Contains : Feature  FeatSet -> Bool .

vars E E' : Feature  .
vars S S' : FeatSet .
eq isEmpty(empty) = true .
eq isEmpty(E S) = false .
eq size(empty) = 0 .
eq size(E S) = 1 + size(S) .
eq Contains(E, empty) = false .
eq Contains(E, E' S) =  E == E' or Contains(E,S) .
endfm
```

Figure 5.16: Le module fonctionnel Feature_FunctMod généré

  - Le diagramme FD-Décoré (Fig.5.17). Pour chaque nœud l'ensemble de ses descendants est calculé.

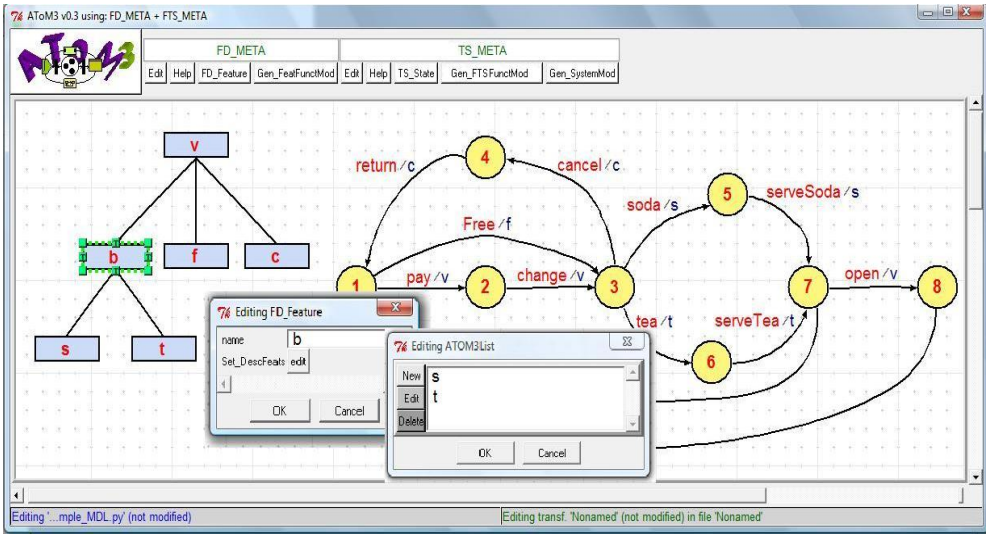


Figure 5.17: Le modèle FD décoré

A titre d'exemple, cette figure montre le contenu de l'attribut *Set_OfDesc* du nœud b. Il s'agit bien des caractéristiques s et t.

- La deuxième grammaire *Gen_FTSFunctMod* agit sur le modèle TS et produit :
  - Le module fonctionnel *FTS_FunctMod* (Fig.5.18).

```
in Feature_FunctMod.txt
fmod FTS_FunctMod is
protecting Feature_FunctMod .

sorts TsState FtsState .
ops   State1 State2 State3 State4
                    State5 State6 State7 State8 : - > TsState [ctor].

op < _ ; _ ; _ > : TsState FeatSet Bool -> FtsState .

endfm
```

Figure 5.18: Le module fonctionnel FTS_FunctMod généré

  - Le modèle TS-Décoré (Fig.5.19). Pour chaque transition, l'ensemble des caractéristiques requises pour le franchissement des transitions concurrentes est calculé.

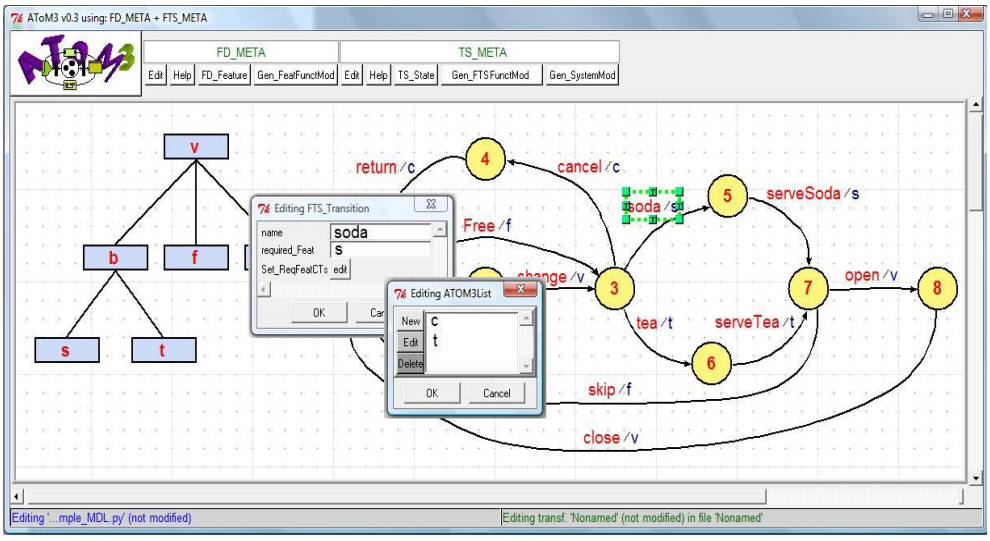


Figure 5.19: Le modèle TS décoré

A titre d'exemple, cette figure montre le contenu de l'attribut *Set_ReqFeatCTs* de la transition Soda. Il s'agit bien des caractéristiques c et t.

- Enfin, la dernière grammaire *Gen_SystemMod* s'applique sur les modèles FD-Décoré et TS-Décoré. Elle génère le module système *FTS_SysMod* contenant les règles de réécriture (Fig5.20).

```
in FTS_FunctMod.txt
mod FTS_SysMod    is
including FTS_FunctMod  .

var  SelectFeats :  FeatSet .

crl [pay]  : < State1; SelectFeats ; False > => < State2 ; SelectFeats; False >
                    if  Contains ( v , SelectFeats)  and
                            (  not  ( Contains ( f , SelectFeats )  and Contains ( f , c f s t b ) ) ) .

crl [change]:< State2; SelectFeats ; False > => < State3; SelectFeats; False >
                    if  Contains ( v , SelectFeats )  .

crl [free] : < State1; SelectFeats ; False > => < State3; SelectFeats ; False>
                    if  Contains ( f , SelectFeats )  and
                             ( not  ( Contains ( v , SelectFeats )  and Contains ( v , empty ) ) ) .

crl [return]:< State4; SelectFeats ; False > => <State1; SelectFeats ; True >
                    if  Contains ( c , SelectFeats )  .

crl [cancel]:< State3; SelectFeats; False > => <State4 ; SelectFeats; False >
                    if  Contains ( c , SelectFeats)  and
                             ( not ( Contains ( s , SelectFeats )  and  Contains (s , empty)  ) )  and
                             ( not ( Contains ( t , SelectFeats)   and  Contains ( t , empty)  ) ) .

crl [soda]:< State3; SelectFeats ; False > => <State5 ; SelectFeats; False >
                    if  Contains ( s , SelectFeats ) and
                             ( not  ( Contains ( t , SelectFeats )  and  Contains (t , empty )  ) ) and
                             ( not  ( Contains ( c , SelectFeats)  and  Contains ( c , empty)  ) ).

crl [tea] : < State3; SelectFeats ; False > =>  < State6; SelectFeats ; False >
                    if  Contains ( t , SelectFeats )  and
                             ( not  ( Contains ( c , SelectFeats )  and  Contains (c , empty)  ) ) and
                             ( not  ( Contains ( s , SelectFeats)   and  Contains ( s , empty)  ) ) .

crl [serveSoda]:<State5; SelectFeats ; False> => <State7; SelectFeats; False>
                    if  Contains ( s , SelectFeats )  .

crl [serveTea]:<State6; SelectFeats ; False > => < State7; SelectFeats; False>
                    if  Contains ( t , SelectFeats )  .

crl[open]:< State7; SelectFeats ; False > =>  < State8 ; SelectFeats; False >
                    if  Contains ( v , SelectFeats )  and
                             ( not  ( Contains ( f , SelectFeats )  and  Contains ( f ,  c f s t b ) ) ) .

crl [skip]:< State7; SelectFeats ; False >  =>  < State1 ; SelectFeats ; True>
                    if  Contains ( f , SelectFeats )  and
                             ( not  ( Contains ( v , SelectFeats )  and Contains ( v , empty ) ) ).

crl [close]: < state8; SelectFeats; False>  =>  < state1 ; SelectFeats; True >
                        if  Contains ( v , SelectFeats )  .


endm
```

Figure 5.20: Le module système FTS_SysMod généré

Après la génération de cette spécification formelle, on procède maintenant à l'analyse et la vérification des propriétés comportementales. Pour s'assurer que la sémantique du modèle FTS a été préservée, on commence d'abord par une simulation du code.

5.6.3 Simulation

Considérons le produit {*v, b ,s, f* }. C'est un distributeur gratuit de Soda. Pour simuler son comportement, l'état initial fourni à l'interpréteur Maude est le suivant :

$$< State_1; v\ b\ s\ f\ ;\ False >$$

La Fig.4.21 montre les résultats de l'évaluation de cette expression avec tous les états intermédiaires.

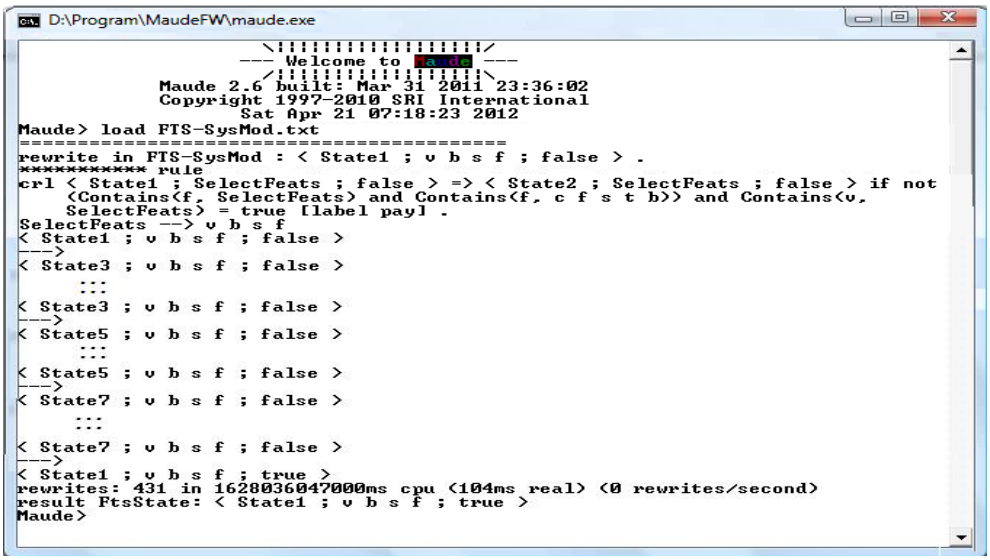


Figure 5.21: Résultats de la simulation

Interprétation : Cette simulation montre que la relation de priorité entre les transitions concurrentes est respectée.

5.6.4 Vérification des propriétés

A titre d'exemple, considérons la propriété suivante :

''Depuis l'état initial, le système finit toujours dans un état final''

Cette propriété est exprimée en LTL de la manière suivante:

[ ] (Initial (State1) | -> Final (State1))

Tel que *Initial* et *Final* sont deux prédicats utilisés pour spécifier respectivement l'état initial et l'état final. Ils sont définis manuellement dans un autre module noté *FTS_PredicatesMod* (Fig.5.22).

```
in model-checker
in FTS_SysMod.txt

mod FTS_PredicatesMod is

protecting FTS_SysMod   .
including SATISFACTION .

subsort FtsState <  State .
op Initial     : TsState -> Prop .
op Final       : TsState -> Prop .

var State  : TsState .
var SelectFeats    : FeatSet .
var flag : Bool .

ceq  < State ; SelectFeats ; flag >  |= Initial (State) = true
                                        if ( flag == false and State == State1) .

ceq  < State ; SelectFeats ; flag >  |= Final  (State) = true
                                        if ( flag == true  and State == State1) .
endm
```

Figure 5.22: Spécification des prédicats

Maintenant, on procède à la vérification de cette propriété. A titre d'exemple, considérons deux variantes suivantes du distributeur automatique:

- P_1 : { *v* , *b* , *d* }
- P_2 : { *v* , *b* , *f* }

L'analyse automatique de ces deux produits est effectuée en faisant appel au Model-Checker par le biais du module suivant :

```
in FTS_PredicatesMod.txt
mod FTS_Check is

protecting FTS_PredicatesMod .
including MODEL-CHECKER .
including LTL-SIMPLIFIER .

ops FTS_Init1   FTS_Init2  : ->  FtsState  .
eq   FTS_Init1    =   < State1 ;  v  b  s  ;   false >  .
eq   FTS_Init2    =   < State1 ;  v  b  f   ;  false >  .

endm

red ModelCheck ( FTS_Init1 , [ ] ( Initial (State1) |-> Final (State1)) ) .
red ModelCheck ( FTS_Init2 , [ ] ( Initial (State1) |-> Final (State1)) ) .
```

Figure 5.23: Spécification de la propriété LTL

Les résultats obtenus sont :

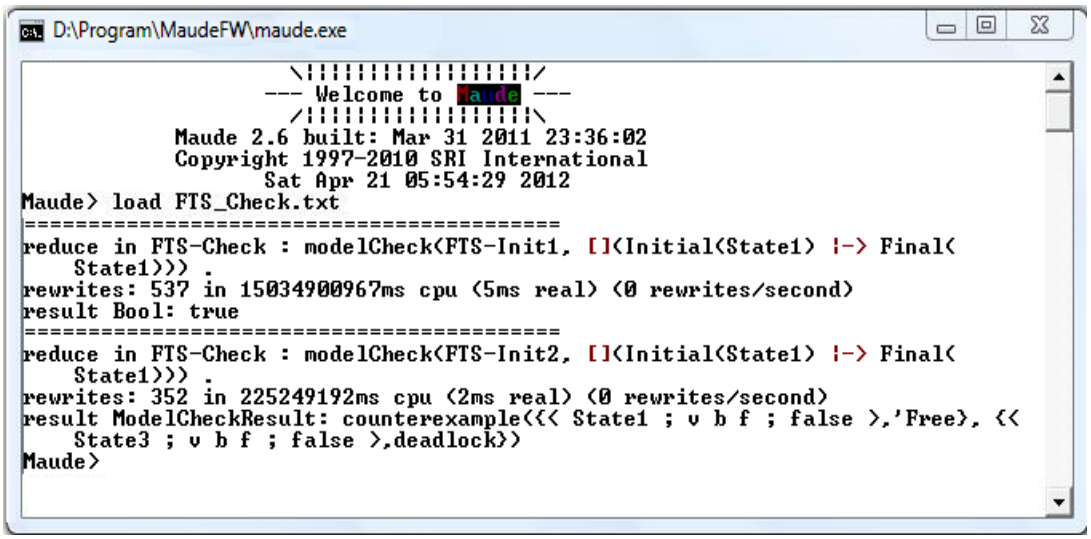


Figure 5.24: Résultats de la vérification

Cette figure montre que :

- La propriété est vérifiée avec succès pour le produit P_1.
- La propriété n'est pas valide pour le produit P_2. Pour affirmer ce résultat, un contre-exemple est affiché.

5.7 Conclusion

Dans ce chapitre, nous avons présenté un support outillé de simulation et d'analyse des lignes de produits. Pour la modélisation, nous nous sommes appuyés sur le formalisme FTS. L'approche proposée est basée sur la logique de réécriture. Elle constituée de deux étapes :

- Etape$_1$: Génération de la spécification Maude.

  Cette étape permet l'édition de trois modules. Les deux premiers sont fonctionnels. Ils sont utilisés pour la spécification algébrique des caractéristiques et des états. Le dernier est le module système. Il est constitué de règles de réécriture conditionnelles spécifiant le comportement souhaité des produits. La priorité entre les transitions concurrentes du modèle FTS est exprimée dans les conditions d'activation. Pour automatiser cette étape, trois grammaires de graphes complémentaires ont étaient proposées. La spécification Maude obtenue procure une description formelle offrant une base solide pour le processus de vérification.

- Etape$_2$: Vérification et analyse.

  Cette étape sert à l'analyse comportementale des produits. L'utilisateur doit d'abord:

  - Spécifier le produit à vérifier en donnant un état FTS initial.
  - Décrire manuellement le comportement souhaité par le biais d'une formule LTL.

  Ensuite, il fait appel au Model-Checker intégré dans l'environnement Maude afin d'évaluer le produit en question. Dans le cas ou la formule n'est pas valide, un contre-exemple est affiché.

A la fin du chapitre, un exemple illustratif détaillé est présenté. Les résultats obtenus sont très significatifs.

Conclusion

Générale

Dans cette thèse, nous nous sommes intéressés particulièrement au paradigme ‘’Lignes de Produits Logiciels’’. Ce dernier a connu un très grand succès dans le monde du génie logiciel. Il est basé principalement sur une réutilisation systématique et planifiée au préalable. L'idée principale consiste à viser dès le début le développement de toute une famille de systèmes fortement similaires. Dès lors, le temps de mise sur le marché et les coûts de développement sont réduits pendant que la qualité des produits s'améliore. Cependant, le constat fait a montré que le développement d'applications suivant cette démarche se heurte à des problèmes inhérents à la vérification des produits finaux. C'est dans cette optique que se situent les travaux réalisés dans cette thèse où nous avons proposé trois outils d'analyse des diagrammes FD et FTS basés sur la technique ‘’Transformations de Graphes’’.

Le premier outil permet la recherche de tous les produits structurellement valides d'un diagramme FD. Nous avons développé deux versions différentes. La première version consiste en une recherche exhaustive de toutes les combinaisons de caractéristiques vérifiant les dépendances [73]. L'idée de base est de générer itérativement les configurations candidates en utilisant un tableau binaire spécifiant les caractéristiques. Pour chaque configuration possible, le produit correspondant est projeté sur le diagramme FD pour le valider. La vérification des dépendances est basée principalement sur une exploration descendante de ce diagramme [74]. La deuxième version est fondée sur l'intégration de l'algorithme Backtracking dans le processus de recherche. Au fait, la détection des combinaisons partielles incorrectes élimine le traitement de toutes les configurations auxquelles elles appartiennent. Cela a permis une réduction importante au niveau des vérifications réalisées.

Pour des raisons d'optimisation, nous avons proposé un deuxième outil permettant d'éviter carrément l'exploration de l'espace de recherche. Il est basé principalement sur le calcul des produits valides par une composition de configurations partielles. A partir des caractéristiques élémentaires, l'idée consiste à construire progressivement des configurations partielles valides plus larges jusqu'à l'obtention des produits recherchés [76]. La technique proposée procède par un traitement à deux étapes. La première étape sert à construire les produits vérifiant les relations paternelles par une exploration descendante du diagramme FD. Il s'agit d'un traitement itératif dans lequel, à chaque itération :

- On sélectionne un nœud pour lequel tous les fils sont déjà traités.
- On construit l'ensemble des configurations partielles possibles à son niveau par une composition des configurations partielles déjà calculées au niveau de ses fils.

La deuxième étape consiste à l'élimination des produits qui violent les contraintes d'exclusion et d'implication.

Enfin, le dernier outil permet une analyse comportementale des modèles FTS en se basant sur logique de réécriture comme formalisme formel de vérification [77]. L'approche proposée est constituée de deux étapes. La première étape sert à la génération d'une spécification Maude composée d'un module système et de deux modules fonctionnels associés. Les transitions du modèle FTS sont translatées en règles de réécriture classiques tout en spécifiant la priorité au niveau des conditions d'activation. La deuxième étape sert à l'analyse comportementale de la spécification générée. Pour vérifier la dynamique d'un produit donné, l'utilisateur doit :

- Spécifier ce produit en donnant un état FTS initial.
- Décrire manuellement une propriété par le biais d'une formule LTL.
- Faire appel au Model-Checker intégré à l'environnement Maude afin de vérifier cette propriété.

Nous considérons que l'intégration de la technique de transformation de graphes est un nouveau moyen d'investigation dans le domaine des lignes de produits. Ce choix est motivé par les raisons suivantes:

- Les grammaires de graphes constituent la solution la plus appropriée pour l'exploration des diagrammes FD et FTS.
- Comme la plupart des traitements proposés sont basés sur des calculs locaux, il a été constaté que les règles de graphes offrent un support très adapté.
- Les grammaires proposées sont mises en œuvre directement comme des outils entièrement automatiques.
- Les grammaires de graphes proposées sont extensibles.

Bien que les objectifs fixés dans cette thèse soient atteints, nous envisageons d'étendre les travaux proposés suivant deux perspectives intéressantes. La première consiste à analyser les diagrammes FD étendus, particulièrement ceux dotés de cardinalités. Cela nécessite le traitement de nouvelles dépendances relativement complexes. Pour la deuxième perspective, nous planifions d'utiliser d'autres formalismes formels pour la vérification. Il nous semble que ceux basés sur l'algèbre de processus sont très convenables.

Bibliographie

Bibliographie

[1] D. McIlroy, "*Mass-Produced Software Components*", In Proceedings of the 1st International Conference on Software Engineering, Germany, pp. 88-98, 1968.

[2] D. Parnas, "*On the Design and Development of Program Families*", IEEE Transactions on Software Engineering, vol. SE-2(1), pp. 1-9, 1976.

[3] P. Naur and B. Randell, eds. "*Software Engineering: Report on a Conference sponsored by the NATO Science Committee*", Scientific Affairs Division NATO, Belgium, 1969.

[4] B. Cox, A. Novabilsky, "*Object-Oriented Programming: An Evolutionary Approach*", Addison Wesley, 1986.

[5] C. Szypersky, "*Component Software: Beyond Object-Oriented Programming*", Addison Wesley and ACM Press, 1998.

[6] I. Crnkovic, "*Component-Based Software Engineering - New Challenges in Software Development*", Journal of Computing and Information Technology, vol. 11(3), pp. 151–162, 2003.

[7] M.P. Papazoglou, "*Service -Oriented Computing: Concepts, Characteristics and Directions*", in Proceedings of the 4th International Conference on Web Information Systems Engineering, Italy, pp.3-12, 2003.

[8] E. Gamma, R. Hem, R. Jahnson and J. Vissides, "*Design Patterns: Elements of Reusable Objec t- Oriented Software*", Addison-Wesley Professional, 1994.

[9] J.D. McGregor, L.M. Northrop, S. Jarrad and K. Pohl , "*Initiating Software Product Lines*", IEEE Software, vol.19(4), pp.24-27, 2002.

[10] P. Clements and L. Northrop, "*Software Product Lines: Practices and Patterns*", SEI Series in Software Engineering, Addison-Wesley, 2001.

[11] R. L. Glass, "*Frequently forgotten fundamental facts about software engineering*", Journal of IEEE Software, IEEE, vol. 18(3), pp. 112 -111, 2001.

[12] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "*Feature-Oriented Domain Analysis (FODA) Feasibility Study*", Tech. Rep. CMU/SEI-90- TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[13] J. Bosch, "*Design & use of Software Architectures: Adopting and Evolving a Product-Line Approach*", Addison-Wesley, 2000.

[14] J. van Gurp, J. Bosch and M. Svahnberg, "*On the Notion of Variability in Software Product Line*", in Proceedings of the Working IEEE/IFIP Conference on Software Architecture, The Netherlands, pp.45-54, 2001.

[15] G. Perrouin, J. Klein, N. Guelfi, and J.M. Jézéquel, "*Reconciling automation and flexibility in product derivation*", In Proceedings of the 12th International Software Product Line Conference, IEEE Computer Society, pp. 339–348, 2008.

[16] P. Clements, "*On the Importance of Product Line Scope*", in Proceedings of the 4th International Workshop, Software Product-Family Engineering, Spain, pp.70-78, 2001

[17] K. Pohl, G. Bockle and F. van der Linden, "*Software Product Line Engineering: Foundations, Principles and Techniques*", Springer, 2005.

[18] P. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps, "*Generic Semantics of Feature Diagrams*", Journal of Computer Networks, vol.51 (2), pp. 456-479, 2007.

[19] K. Kang, S. Kim, J. Lee, K. Kim, E. Shin and M. Huh, "*FORM: a Feature-Oriented Reuse Method with Domain-Specific Reference Architectures*", Annals of Software Engineering, vol.5(1), pp. 143–168, 1998.

[20] M. Griss, J. Favaro, and M. d'Alessandro, "*Integrating Feature Modeling with the RSEB*", In Proceedings of the 5th International Conference on Software Reuse, Canada, IEEE, pp. 76-85, 1998.

[21] K. Czarnecki, S. Helsen, and U. Eisenecker, "*Staged Configuration using Feature Models*", Proceedings of the 3rd International Conference on Software Product Lines, USA, LNCS, Springer, vol. 3154, pp. 266-283, 2004.

[22] K. Czarnecki, S. Helsen, and U. Eisenecker, "*Formalizing Cardinality-Based Feature Models and their Specialization*", Software Process: Improvement and Practice, vol. 10(1), pp. 7-29, 2005.

[23] M. Riebisch, D. Streitferdt, and I. Pashov. "*Modeling Variability for Object-Oriented Product Lines*". ECOOP 2003 Workshop Reader, LNCS, Springer, vol. 3013, pp. 165-178, 2004.

[24] T. Asikainen, T. Mannisto, and T. Soininen. "*A Unified Conceptual Foundation for Feature Modelling*". In Proceedings of the 10th International Software Product Line Conference, IEEE, pp. 31-40, 2006.

[25] T. von der Massen and H. Lichter. "*RequiLine: A Requirements Engineering Tool for Software Product Lines*", Software Product-Family Engineering, LNCS, Springer, pp. 168-180, 2004.

[26] C. Atkinson, J. Bayer, and D. Muthig, "*Component-Based Product Line Development: the Kobra Approach*", In Proceedings of the 1st Conference on Software product lines, Kluwer Academic Publishers, pp. 289–309, 2000.

[27] H. Gomaa, "*Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*", Addison Wesley, USA, 2004.

[28] T. Ziadi, L. Hélouet and J-M. Jézéquel, "*Towards a UML profile for software product lines*", In Proceedings of the 5th International Workshop on Product Familly Engineering (PFE-5), LNCS, Springer, vol. 3014, pp. 129–139, 2003.

[29] A. Gruler, M. Leucker, K. Scheidemann, "*Calculating and Modeling Common Parts of Software Product Lines*", Proceedings of the 12th Software Product Line Conference, IEEE Press, pp. 203 – 212, 2008.

[30] D. Kozen: "*Results on the propositional mu-calculus*", Theoretical Computer Science, vol. 27(3), pp. 333–354, 1983.

[31] R. Muschevici, D. Clarke and J. Proenca, "*Feature Petri Nets*", Proceedings of the 14th International Software Product Line Conference, UK, vol. 2, pp. 99–106, 2010.

[32] A. Fantechi and S. Gnesi, "*Formal modeling for product families engineering*", Proceedings of the 12th International Software Product Line Conference, IEEE Computer Society Press, pp. 193–202, 2008.

[33] A. Classen, P. Heymans, P. Y. Schobbens, A. Legay and J. F. Raskin, "*Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines*", Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol.1, pp. 335–344, Cape Town, 2010.

[34] D. Benavides, P. Trinidad and A. Ruiz-Cortes, "*Automated Reasoning on Feature Models*", LNCS, Springer, vol. 3520, pp. 491–503, 2005.

[35] D. Benavides, S. Segura, P. Trinidad and A. Ruiz-Cortes, "*Using Java CSP Solvers in the Automated Analyses of Feature Models*", LNCS, Springer, vol. 4143, pp 399-408, 2006.

[36] E. Bagheri, T.D. Noia, D. Gasevic and A. Ragone, "*Formalizing Interactive Staged Feature Model Configuration*", Journal of Software: Evolution and Process, vol. 24(4), pp. 375–400, 2012.

[37] D. Batory, "*Feature models, grammars, and propositional formulas*", in Proceedings of the 9th international conference on Software Product Lines, LNCS, Springer, vol. 3714(20), pp. 7-20, 2005.

[38] M. Mendonca, A.Wasowski, K. Czarnecki and D. Cowan, "*Efficient Compilation Techniques for Large Scale Feature Models*", Proceedings of the 7th International Conference on Generative Programming and Component Engineering, pp. 13–22, 2008.

[39] J. Sun, H. Zhang, Y.F. Li and H. Wang, "*Formal Semantics and Verification for Feature Modeling*", In Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, pp. 303-312, 2005.

[40] H. Wang, Y.Li, J. Sun, H. Zhang and J. Pan, "*Verifying Feature Models using OWL*", Journal Web Semantics: Science Services and Agents on the World Wide Web, vol. 5(2), pp. 117-129, 2007.

[41] A. Osman, S. Phon-Amnuaisuk and C.K. Ho, "*Using First Order Logic to Validate Feature Model*", In Proceedings of the 3rd International Workshop on Variability Modeling of Software-intensive Systems, Spain, pp. 169–172, 2009.

[42] A. Classen, P. Heymans, P. Y. Schobbens, and A. Legay, "*Symbolic Model Checking of Software Product Lines*", in Proceedings of the 33rd International Conference on Software Engineering, ACM, pp. 321-330, 2011.

[43] N. D'Ippolito, D. Fischbein, M. Chechik, and S. Uchitel ,"*MTSA: The Modal Transition System Analyser*", in Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 475-476, 2008.

[44] J. Liu, S. Basu and R. Lutz, "*Compositional Model Checking of Software Product Lines using Variation Point Obligations*", Journal of Automated Software Engineering, vol. 18(1), pp. 39-76, 2011.

[45] D. Benavides, S. Segura, and A. Ruiz-Cortés, "*Automated Analysis of Feature Models 20 Years Later: A Literature Review*", Journal of Information Systems, vol. 35(6), pp. 615–636, 2010.

[46] K. Pohl, G. Bockle, and F. van der Linden, "*Software Product Line Engineering: Foundations, Principles and Techniques*", Springer, 2005.

[47] F. van der Linden, K. Schmid and E. Rommes, "*Software Product Lines in Action: The Best Industrial Practice in Product Line engineering*", Springer, 2007.

[48] J. Bézivin. "*In Search of a Basic Principle for Model Driven Engineering*", The European Journal for the Informatics Professional, vol. 2, pp. 21-24, 2004.

[49] C. Atkinson and T. Kuhne, "*Model-Driven Development: A Metamodeling Foundation*", Journal IEEE Software, vol. 20(5), pp. 36–41, 2003.

[50] OMG: MetaObject Facility (MOF), version 2.0, Online: http://www.omg.org/mof/.

[51] K. Czarnecki and S. Helsen, "*Classification of Model Transformation Approaches*", In Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, USA, 2003.

[52] OMG: Model Driven Architecture (MDA), Online: http://www.omg.org/mda/.

[53] OMG: Meta Object Facility (MOF) 2.0, Query/View/Transformation (QVT Specification), Online: http://www.omg.org/spec/QVT/1.0/.

[54] "*OptimalJ*", Online: http://www.compuware.com/products/optimalj.

[55] D. Balasubramanian, A. Narayanan, C. vanBuskirk and G. Karsai, "*The Graph Rewriting and Transformation Language: GReAT*", In Proceedings of the 3rd International Workshop on Graph Based Tools, Brazil, 2006.

[56] "*AGG*", Online: http://tfs.cs.tu-berlin.de/agg/.

[57] J. De Lara and H. Vangheluwe, "*AToM 3: A Tool for Multi-Formalism Modelling and Meta-Modelling*", LNCS, Springer, vol. 2306, pp. 174-188, 2002.

[58] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev and P. Valduriez, "*ATL: a QVT Like Transformation Language*", In Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, USA, pp. 719– 720, 2006.

[59] P.A. Muller, F. Fleurey and J.M. Jezequel, "*Weaving Executability into Object-Oriented Meta-Languages*", LNCS, Springer, vol. 3713, pp. 264–278, 2005.

[60] C. Dumoulin, "*ModTransf : A Model to Model Transformation Engine*", 2004, http://www.lifl.fr/west/modtransf.

[61] H. Ehrig, G. Taentzer, "*Graphical Representation and Graph Transformation*", ACM Computing Surveys, vol. 31(3), 1999.

[62] G. Rozenberg, "*Handbook of graph grammars and computing by graph transformation*", World Scientific, 1999.

[63] "Rhapsody", Online: http://www.ilogix.com.

[64] M. Belaunde, and G. Dupé, "*SmartQVT*", 2006, http://smartqvt.elibel.tm.fr.

[65] "*CPN-AMI*", Online: http: //move.lip6.fr/software/cpnami/.

[66] A. Konigs, "*Model Transformations with Triple Graph Grammars*", in Model Transformations in Practice Workshop at MoDELS, Jamaica, 2005.

[67] S. Burmester, H. Giese, J. Niere, M. Tichy, J.P. Wadsack, R. Wagner, L.Wendehals, and A. Zündorf, "*Tool Integration at the Meta-Model Level within the Fujaba Tool Suite*", International Journal on Software Tools for Technology Transfer, vol. 6(3), pp. 203-218, 2004.

[68] G. Csertan, G. Huszerl, I. Majzik, Z. Pap, D. Pataricza, A.and Varro. "*Viatra -Visual Automated Transformations for Formal Verification and Validation of UML Models*", In Proceedings of the 17th IEEE International Conference on Automated Software Engineering, UK, pp. 267-270, 2002.

[69] "*Eclipse Modeling Galileo*", Online: http://www.eclipse.org/downloads/packages/release/galileo/.

[70] A. Schürr, A. J. Winter and A. Zündorf, "*The PROGRES Approach: Language and Environment* ", World Scientific Publishing , vol. 2, pp. 487-550, 1999.

[71] "Python", Online: http://www.python.org.

[72] OMG: Object Constraint Language (OCL), Online: http://www.omg.org/technology/documents/formal/ocl.htm.

[73] K. Khalfaoui, A. Chaoui, C. Foudil and E. Kerkouche, "*Automatic Generation of SPL Structurally Valid Products Using Graph Transformations Approach"*, Studies in Computational Intelligence, Springer, vol. 488, pp. 347-356, 2013.

[74] K. Khalfaoui, A. Chaoui, C. Foudil and E. Kerkouche, "*Structural Validation of Software Product Line Variants: a Graph Transformation Based Approach*", International Journal of Software Engineering & Applications, vol.4(2), pp. 19-31, 2013.

[75] J. R. Bitner and E. M. Reingold, "*Backtrack Programming Techniques*", Communications of the ACM, vol. 18(11), pp. 651–656, 1975.

[76] K. Khalfaoui, A. Chaoui, C. Foudil and E. Kerkouche, "*Automatic Generation of SPL Structurally Valid Products : an Aapproach Based on Progressive Composition of Partial Configurations*", (to be submitted).

[77] K. Khalfaoui, A. Chaoui, C. Foudil and E. Kerkouche, "*Formal Specification of Software Product Lines: A Graph Transformation Based Approach*", Journal of Software, vol. 7(11), pp. 2518-2532, 2012.

[78] J. Meseguer, "*Conditional Rewriting Logic as a Unified Model of Concurrency*", Theoretical Computer Science, vol. 96(1), pp. 73-155, 1992.

[79] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada, "*Maude: Specification and Programming in Rewriting Logic*", Internal Report, SRI International, 1999.

[80] S. Eker, J. Meseguer and A. Sridharanarayanan, "*The Maude LTL Model Checker*", Electronic Notes in Theoretical Computer Science, Elsevier, vol. 71, 2002.