MetaScribe : un outil pour la génération de moteurs de réécriture

Fabrice Kordon,
Laboratoire d'Informatique de Paris 6,
4 place Jussieu, 75252 Paris Cedex 05,
E-mail: Fabrice.Kordon@lip6.fr

Résumé : Ce rapport présente les principes de fonctionnement de l'outil ***M**eta**S**cribe*. ***M**eta**S**cribe* est un générateur de moteurs de réécriture permettant de construire facilement des traducteurs d'un format de représentation (formalisme) vers un autre format. ***M**eta**S**cribe* distingue différents aspects d'un moteur de réécriture :

- la description du formalisme en entrée (ses composants) de manière à en dériver des descriptions de modèles (des descriptions à traiter),
- la description des modèles selon un format d'entré prédéfini et polymorphe, déduit de la description du formalisme associé,
- la description des transformation sémantiques (i.e. les règles de production d'arbres d'expressions),
- la description d'un sucre syntaxique à appliquer aux arbres d'expressions.

MetaScribe est utilisé pour construire des générateurs de programmes (à partir de représentations graphiques ou textuelles).

Mots clefs : Génération de programmes, Génie Logiciel

Abstract : This report presents ***M**eta**S**cribe*, a transformation engine generator. ***M**eta**S**cribe* eases the construction of translator (form a format to another) and program generators. It distinguishes discrete aspects :

- the description of input formalism (components of a representation) to deduce the description of models (specification to be processed),
- the description of models using the format deduced from the formalism description,
- the definition of semantic transformations rules (e.g. production of expression trees),
- the definition of a syntax to apply on expression trees.

MetaScribe is currently used to produce program generators.

Key-words : Program Generation, Software Engineering

1. Introduction

MetaScribe est un générateur de moteurs de transformations. Sa réalisation répond à différents besoins :

- supporter le développement de générateurs de programmes (Section 1.1.),
- faciliter la mise en œuvre des standards secondaires dans FrameKit (Section 1.2.).

Dans les deux cas, un mécanisme de description de formats de représentations et la possibilité de réaliser des transformations entre formats d'échanges doivent être mis en œuvre. ***M**eta**S**cribe* propose un gabarit pour la définition des formats en entrée et facilite la production de moteurs de réécriture adaptés aux deux cas de figure sus-mentionnés.

***M**eta**S**cribe* couvre un problème voisin de celui identifié dans la conception en co-développement matériel/logiciel (codesign) de systèmes embarqués pour lesquels des processeurs spéciaux sont construits en petite série (en général à base de FPGA). Le problème est celui de la particularisation à faible coût d'un compilateur en vue de la production d'un code spécifique lié aux composants matériels. Pour cela, on utilise des compilateurs multi-cibles (retargetable compilers) que **[1]** classifie comme suit :

- les compilateurs automatiquement adaptables, ce qui suppose que le compilateur contient la description de tous les jeux d'instructions qu'il sera susceptible d'utiliser;
- les compilateurs adaptable par l'utilisateur, pour lesquels un jeu d'instruction réduit est prédéfini en vue d'être paramétré. L'usager qui souhaite particulariser son compilateur redéfinit alors le codage de ce jeu d'instructions;
- les compilateurs adaptables par un développeur, qui permettent non seulement l'adaptation du jeu d'instruction mais aussi la définition de règles d'optimisation propres au composant matériel cible.

***M**eta**S**cribe* propose, dans son domaine, une réponse aux deux derniers points mais en intégrant également des caractéristiques propres à celle des générateurs d'analyseur lexicaux et syntaxiques comme le couple flex/bison **[11, 4]**.

D'un générateur d'analyseur lexicaux et syntaxiques, il reprend à la fois la description d'un format en entrée et la possibilité de définir des règles qui seront appliquées selon un schéma d'analyse des données fournies en entrée du moteur de réécriture. ***M**eta**S**cribe* est orienté vers l'analyse de représentations graphiques ou hiérarchiques et utilise donc des mécanismes de description du format en entrée différents de ceux utilisés dans un analyseur lexical ou syntaxique. Par contre, l'utilisateur peut parfaitement définir, à l'aide d'un *patron sémantique,* non seulement le schéma d'analyse des entrées mais aussi les constructions sémantiques qui constituent des éléments de du format de sortie indépendamment de leur expression syntaxique.

D'un compilateur multi-cible, il reprend la possibilité de paramétrer une «sortie» en fonction d'un format défini dans un *patron syntaxique*. Il est ainsi possible d'associer aux éléments sémantiques identifiés dans un patron sémantique un sucre syntaxique permettant de les exprimer. Le paramétrage du format en sortie s'apparente à l'hypergénéricité **[3]** d'un générateur de code. L'hypergénéricité du générateur de code est en alors décrite à l'aide de fichiers de configurations. L'outil Objectering **[13]** utilise son propre format de configuration (formalisme en entrée : Classe-relation) et celui de S-CASE **[10]** (formalisme en entrée : UML) est décrit en Tcl.

*M*eta*S*cribe permet de paramétrer le format d'une spécification en entrée et dissocie les aspects sémantiques et syntaxiques de la description d'un moteur de réécriture, ce qui les rend potentiellement réutilisables. Ce point sera détaillé en Section 1.3.

1.1. Application à la génération de code

La génération de code vise à la production de programmes par transformation d'une spécification initiale décrite selon un formalisme exploité par le moteur de transformation. L'approche proposée par *M*eta*S*cribe convient parfaitement à ce problème (Figure 1). A partir d'un modèle source, on peut identifier les règles sémantiques produisant une description en mémoire des programmes générés en fonction de concepts issus d'un paradigme de programmation (i.e. des arbre d'expression de programmes). Ensuite, un patron syntaxique appliqué à cette vision conceptuelle permet de l'exprimer avec la syntaxe d'un langage de programmation.

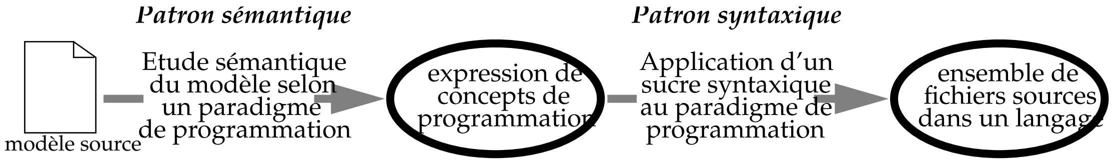

Figure 1 : Utilisation de *M*eta*S*cribe pour la construction d'un générateur de programmes.

1.2. Application à la mise en œuvre de standards secondaires dans FrameKit

FrameKit **[6, 7, 8]** est une plate-forme d'accueil dédié au prototypage rapide d'environnements de Génie Logiciel. Une plate-forme d'accueil doit permettre la manipulation de modèles graphiques et hiérarchiques, l'application de services sur ces modèles et le stockage des résultats obtenus et dispose de fonctions permettant une administration simple d'entités comme les utilisateurs, les services etc.

Pour assurer ces différents objectifs, FrameKit dispose :

- d'une interface utilisateur paramétrable (Macao **[9]**) pour étudier, mettre au point puis utiliser des formalismes graphiques et hiérarchiques,
- de bibliothèques de composants facilitant le développement d'applications dans le contexte de cette plate-forme afin d'offrir aux chercheurs un environnement de développement cohérent,
- de mécanismes d'intégration permettant d'importer des logiciels développés par d'autres,
- de techniques de développement multi-cibles pour assurer une bonne diffusion du «produit». Cette caractéristique doit concerner aussi bien la plate-forme elle-même que les composants offerts aux programmeurs d'outils,
- de mécanismes d'administration et de diffusion assurant une administra-

tion simple et proposant un gabarit pour faciliter la diffusion d'outils.

L'expérience montre qu'il est difficile pour une plate-forme d'accueil d'offrir un standard d'échange générique et adapté à tous les usages. Il est alors intéressant de d'associer à ce format d'échange une bibliothèque de fonctions adaptée à certaines formes de représentations. Un bon exemple de cette stratégie est la mise en place de formalismes[(1)] prédéfinis dans Ptolemy **[12]**, comme le «Synchronous Dataflow Domain» pour lesquels des bibliothèques optimisées er reposant sur les mécanismes de base de l'environnement **[2]**. Ce travail se justifie pour des formats de représentation (formalismes) fortement utilisés dans les domaines d'application couverts par le projet.

Cette observation nous a conduit à proposer deux niveaux de standardisation dans FrameKit : primaires (c'est à dire propre à la plate-forme) et secondaires (liés à un formalisme). Les standards primaires sont constitués par le langage de description générique d'échanges (CAMI) pour tout ce qui est échange de données et expression des résultats. Les standards secondaires définissent les conventions devant être respectées par tous les outils associés à un formalisme comme :

- Les formats de représentation des modèles. Ainsi, on peut factoriser la vérification des modèles en la confiant à un outil dédié acceptant du CAMI en entrée et produisant le format adapté, directement exploité par les outils qui ignorent alors CAMI;
- Les formats de représentation des résultats, ce qui permet à des outils d'exploiter les résultats produits par d'autres;

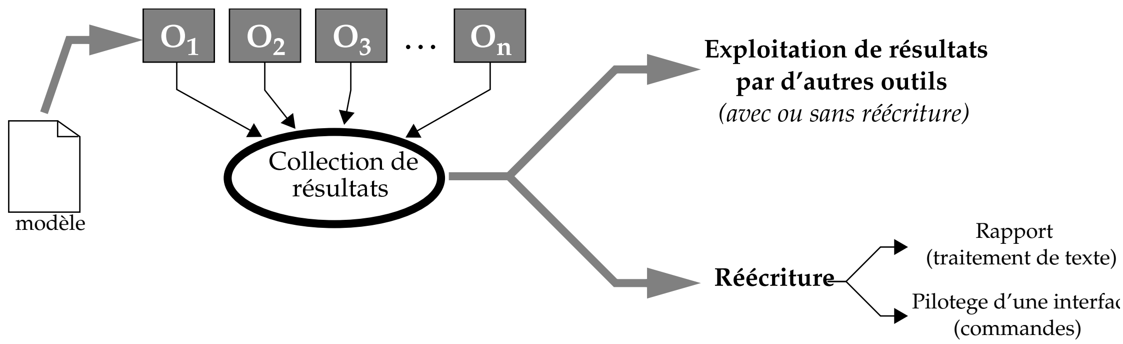

Figure 2 : Utilisation de moteurs de réécriture pour supporter la gestion de standards secondaires.

La mise en œuvre de standards secondaires associés à un formalisme permet d'unifier la représentation des résultats par des outils d'origine distincte. Ainsi, à un formalisme, on pourra associer par application de services une collection de résultats décrits selon un format unique (Figure 2). De tels résultats peuvent alors être exploités par d'autres outils sans que ceux ci se préoccupent de leur origine. Un résultat susceptible d'être calculé par plusieurs outils peut être exploité par d'autres dans que l'on se préoccupe de formats «propriétaires». La

(1) Dans Ptolemy, les formats de représentation sont appelés «domaines».

multiplicité d'outils calculant un même résultat peut être liée à un changement de version, d'architecture ou d'invocations différentes en fonction de critères de performance liées aux caractéristiques du modèle source.

Lorsque les outils n'admettent pas directement en entrée le format choisi pour représenter ce standard secondaire, on peut construire un moteur de réécriture effectuant la traduction de ce format vers le format exigé. Ce moteur sera appliqué avant exploitation des résultats. De même, différents moteurs de réécriture peuvent aboutir à des formats de visualisation différents. Dans la Figure 2, on trouve deux formats : la production d'un rapport (exportation d'un document exploitable par un traitement de texte) ou le pilotage d'une interface utilisateur (par exemple, des directives graphiques de visualisation en langage CAMI).

1.3. Principes de *M*eta*S*cribe

Un moteur de transformation est un outil prenant en entrée un modèle dans un formalisme source, et produisant un autre modèle dans un formalisme cible (Figure 3). Les modèles soumis au moteur de réécriture sont décrits au moyen d'un langage proposé par *M*eta*S*cribe. Les modèles cibles sont produits au format ASCII.

Figure 3 : Fonctionnement d'un moteur de transformation.

Entités composant un moteur de réécriture

La première composante nécessaire à la définition d'un moteur de réécriture est le *formalisme source*. De cette description, on déduit les entités potentiellement présentes dans le modèle à traiter. La description du *formalisme cible* (en sortie), quand à elle, est «diluée» dans la définition des règles de transformation.

Nous avons dissocié les aspects sémantiques et syntaxiques d'une transformation. Le premier concerne la transformation en elle même, le second correspond au sucre syntaxique appliqué pour représenter le format de sortie. Pour cela, nous introduisons deux types de patrons :

- Un *patron sémantique* est associé au formalisme source. Il définit les règles de transformations sémantiques applicables entre le formalisme en entrée et le formalisme en sortie.
- Un *patron syntaxique* est associé à un patron sémantique. Il décrit les expressions syntaxiques applicables pour la forme sémantique définie dans le patron sémantique correspondant; c'est à dire la structure de ce qui est produit dans le formalisme cible.

*M*eta*S*cribe génère un moteur de transformation à partir du triplet <*formalisme source, patron sémantique, patron syntaxique*> le caractérisant. Ce mécanisme permet de réutiliser les différents composants : à un formalisme source, on peut associer plusieurs patrons sémantiques correspondant à des transformations dans des formalismes cibles de classes différentes. De même, à un couple <*formalisme, patron sémantique*>, on peut associer plusieurs patrons syntaxiques, chacun correspondant à un habillage syntaxique différent des éléments produits par le patron sémantique.

La Figure 4 illustre les possibilités de réutilisation des différents composants décrivant un moteur de transformation. Nous y considérons la génération de code à partir d'un réseau de Petri pour un langage procédural classique ou un langage objet. On construit deux types de patrons sémantiques : le premier produit des expressions adaptées aux langages procéduraux tandis que le second génère des expressions adaptées aux langages objets. Dans chacun des cas, on peut appliquer des patrons syntaxiques différents selon que l'on souhaite générer du C, de l'Ada83, du C++ ou du Java.

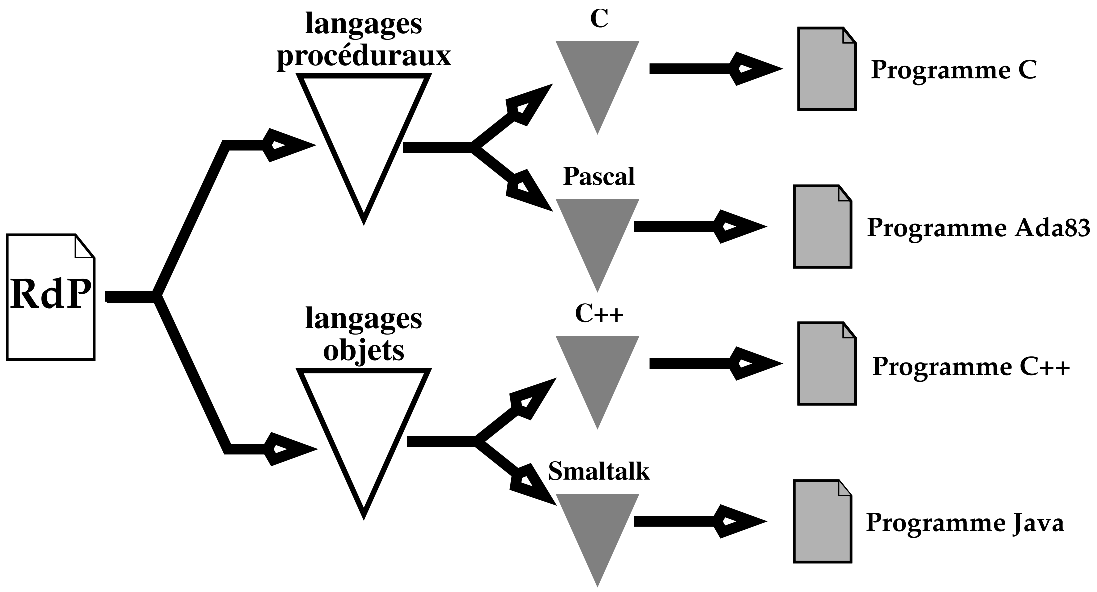

Figure 4 : Exemple de transformations d'un réseau de Petri dans plusieurs formalismes cibles appartenant à des classes différentes. Si tous les patrons sont définis, on peut obtenir ainsi quatre moteurs de transformations à partir d'un réseau de Petri.

Les entités caractérisant une transformation (la description du formalisme source, le patron sémantique et le patron syntaxique) sont définies de manière modulaire : ainsi, deux patrons sémantiques, syntaxiques, ou deux formalismes peuvent partager des définitions communes. Dans l'exemple donné précédemment, il est évident que le patron syntaxique associé au langage C++ empruntera beaucoup au patron syntaxique du C. De même, le patron sémantique pour les langages objets peut s'appuyer, pour ce qui concerne des constructions «classiques», sur des éléments appartenant au patron sémantique des langages procéduraux.

Structure d'un moteur de transformation généré avec MetaScribe

La structure d'un moteur de transformation généré avec ***M**eta**S**cribe* est donnée en Figure 5. La transformation se déroule en trois temps :

- décodage du modèle et construction d'une représentation en mémoire,
- application sur ce modèle du patron sémantique (PSm dans la Figure) qui produit des arbres d'expression en mémoire (les arbres sémantiques),
- application du patron syntaxique (PSy dans la figure) sur les arbres sémantiques pour y appliquer le sucre syntaxique correspondant.

Les modèles fournis au moteur de transformation doivent être décrits selon un format précis indépendant du formalisme. L'interprétation des entités d'un modèle se fait à partir de la description du formalisme.

Les patrons sémantiques sont composés de règles qui décrivent les mécanismes de transformation devant aboutir à la production d'une représentation sémantique dans le formalisme cible. Cette représentation s'exprime sous la forme d'arbres d'expressions (arbres sémantiques). Les nœuds de ces arbres sont étiqueté par des chaînes de caractères et/ou par des constructeurs ayant une signification dans le formalisme cible.

Figure 5 : Les étapes du traitement dans un moteur de réécriture obtenu grâce à ***M**eta**S**cribe*.

Pour appliquer des règles syntaxiques aux arbres produits par application des règles d'un patron sémantique, on associe un ensemble de règles syntaxiques aux constructeurs définis dans le patron sémantique, ces derniers constituant le lien entre le patron sémantique et le patron syntaxique.

Dans la section suivante, nous allons décrire les principes de la mise en œuvre de ***M**eta**S**cribe*. Nous présentons à partir de la Section 2. les mécanismes mis en œuvre pour décrire les différentes entités dans ***M**eta**S**cribe* et proposons en Section 6. un exemple d'utilisation.

1.4. Mise en œuvre de *MetaScribe*

Quatre langages ont été définis pour permettre la description des différentes entités nécessaires à la caractérisation d'un moteur de transformation (Figure 6) :

- MSF (***M**eta**S**cribe* **F**ormalism) pour déclarer les entités d'un formalisme,
- MSM (***M**eta**S**cribe* **M**odel) pour décrire un modèle en fonction des entités déclarées dans la description au format MSF du formalisme associé,
- MSSM (***M**eta**S**cribe* **SeM**antic pattern) pour expliciter les règles de production d'une description sémantique dans la représentation cible,

- MSST (***M**eta**S**cribe* **S**yn**T**actic pattern) pour définir le sucre syntaxique applicable aux expressions produites par application d'un patron sémantique.

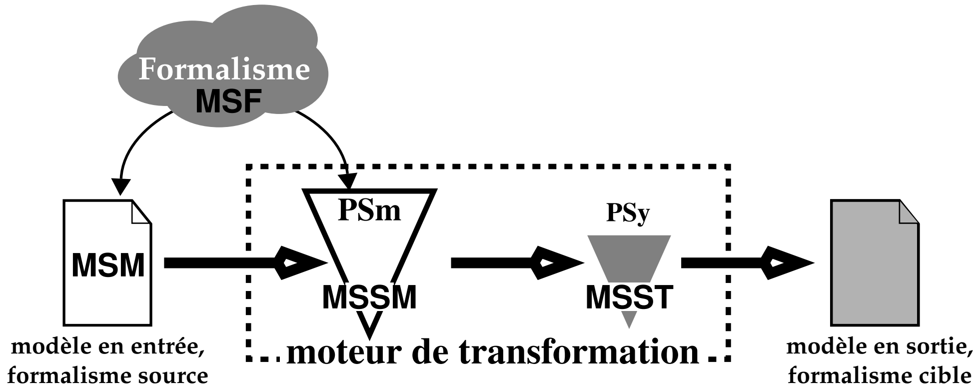

Figure 6 : Les entités manipulées par ***M**eta**S**cribe* et leur langages de description.

***M**eta**S**cribe* est un générateur de code produisant des programmes Ada à partir des règles contenues dans les patrons sémantiques et syntaxiques. Les vérifications liées au fort typage de ce langage sont utilisées pour garantir la sécurité du fonctionnement du moteur de transformation en assurant le contrôle dynamique des entités manipulées pendant l'exécution(2). Le moteur de transformation ainsi obtenu produit des fichiers ASCII et s'intègre tel quel dans FrameKit (respect des hypothèses de fonctionnement et utilisation des interfaces programmatiques). Ce moteur est dédié à la transformation de modèles (au format MSM) conformes au formalisme source (décrit en MSF).

Pour représenter les formalismes et les modèles, il était impossible d'utiliser tels quels les mécanismes offerts par Macao qui sont trop rudimentaires. Aux notions de nœuds et de connecteurs auxquels sont associés des attributs, il faut ajouter un typage de ces attributs, ce qui permet à la fois de vérifier leur valeur mais ouvre des possibilités importantes quand à leur manipulation (ce qui est important pour définir les règles sémantiques).

1.4.1. Principes du langage MSF

MSF s'apparente à TIM dans l'environnement Tycho **[5]** et est tout à fait adapté à la description de formalismes graphiques et hiérarchiques. Les formalismes que nous avons retenus sont potentiellement graphiques et hiérarchiques (comme les réseaux de Petri ou H-COSTAM). La description d'un formalisme textuel n'est qu'un cas particulier facile à élaborer.

MSF permet de déclarer des composants élémentaires :

- les *attributs globaux* sont des informations associées au modèle;
- les *nœuds* sont des objets typés auxquels sont potentiellement associés des informations définies au moyen d'attributs. Chaque instance de nœud

(2) Le générateur de code de ***M**eta**S**cribe* est ainsi déchargé de cette tâche, ce qui ne serait pas toujours le cas pour d'autres langages de programmation.

dans un modèle sera caractérisée par un identificateur (unique) et les valeurs associées à ses attributs;

- les *liens* sont des objets typés auxquels sont potentiellement associés des attributs. Chaque instance de lien dans un modèle relie entre eux N nœuds (N ≥ 2); elle sera caractérisée par un identificateur (unique) et les valeurs associées à ses attributs;
- les *constructions sources prédéfinies* correspondent aux constructions syntaxiques licites que l'on peut trouver dans les attributs correspondant à des arbres d'expression (qui seront définis plus loin).

Les nœuds ne peuvent être reliés qu'au moyen d'un lien. Pour accroître les possibilités de contrôle dynamique, on exprime des règles de connectivité des nœuds permettant de préciser :

- Le nombre maximum de liens d'un type donné auquel il peut être relié,
- la direction des connexions acceptées. Les choix possibles sont : en entrée, en sortie ou sans direction. Dans les deux premiers cas, le lien est orienté et ne peut relier plus de deux nœuds. Dans le dernier cas, il n'est pas orienté et aucune limite n'est donnée aux nombre de nœuds qu'il relie.

Les attributs, qu'ils soient globaux, associés à un nœud ou à un lien, sont typés comme suit :

- *integer* : l'attribut contient une valeur entière,
- *string* : l'attribut contient une chaîne de caractère,
- *text* : l'attribut contient une suite de chaînes de caractères,
- *expression* : l'attribut contient un arbre d'expression(3),
- *boolean* : l'attribut contient une valeur booléenne(4).

Noeuds, liens, attributs et constructions sources prédéfinies constituent le lien entre un formalisme et le patron sémantique qui lui est associé.

1.4.2. Principes du langage MSM

Le langage MSM permet de décrire un modèle. C'est un mode de représentation neutre par rapport au formalisme et le format de description fourni en entrée d'un moteur de transformation produit par ***M**eta**S**cribe*.

Ce langage décrit un modèle par rapport à son formalisme en identifiant les instances de chaque classe d'objet et leurs relations. Si une entité non déclarée dans le formalisme apparaît, le moteur de transformation trace l'erreur et stoppe son exécution. La description MSM n'est pas directement manipulée par ***M**eta**S**cribe*; il est exploité par les moteurs de transformation générés.

1.4.3. Principes du langage MSSM

Le langage MSSM permet de décrire un patron sémantique composé de trois types d'entités : les *constructeurs prédéfinis*, les *règles sémantiques* et les *arbres*

(3) Cet arbre correspond typiquement à une expression compilée.

(4) Ce booléen est utilisable dans les expressions conditionnelles du langage MSSM.

sémantiques statiques. Les constructeurs prédéfinis identifient les concepts manipulés dans le formalisme cible (pour un langage de programmation, il s'agit des instructions potentielles). Les règles sémantiques constituent le noyau des mécanismes de transformation tandis que les arbres sémantiques statiques sont des macro paramétrées définissant des constantes.

Une règle sémantique manipule les entités définies dans le formalisme source, c'est à dire :

- accès aux nœuds et liens d'un modèle,
- accès aux attributs, qu'il soient globaux ou associés aux nœuds ou liens du modèle.

Une règle sémantique construit un arbre sémantique selon un format choisi, qui constitue une norme pour le patron sémantique considéré. Pour cela, il est possible :

- d'utiliser les constructeurs prédéfinis du patron sémantique. Ces constructeurs caractérisent des concepts propres au formalisme cible. Il sont référencés dans les nœuds des arbres sémantiques et servent de lien avec le patron syntaxique associé;
- d'invoquer d'autres règles sémantiques;
- de référencer des arbres statiques,
- d'appliquer une règle syntaxique à un arbre sémantique pour produire un texte dans un fichier.

Règles sémantiques et arbres statiques sont paramétrables. Les paramètres sont typés et peuvent être :

- une chaîne de caractères quelconque,
- un entier quelconque,
- un arbre syntaxique au format décrit dans le langage MSM,
- un arbre sémantique au format de ceux produits par une règle sémantique ou un arbre statique,
- un constructeur prédéfini du patron sémantique,
- une référence à une entité du modèle (nœud ou lien).

Les règles sémantiques sont typées par le type de valeur qu'elles rendent qui peuvent être :

- des chaînes de caractères,
- des entiers,
- des arbres sémantiques,
- des constructeurs prédéfinis,
- aucune valeur.

Il existe deux types de règles sémantiques : les *règles principales* qui seront toutes appliquées par le moteur de réécriture, et les *règles secondaires* qui correspondent à des «sous-programmes» invoqués depuis une autre règle. Un patron sémantique doit contenir au moins une règle principale. Par définition, les règles principales ne rendent aucune valeur.

Le corps d'une règle sémantique manipule les entités d'un modèle en vue de produire les arbres décrivant le résultat de la transformation. Ces arbres sont d'arité quelconque. Leurs nœuds contiennent *au moins une* des trois informations suivantes : un constructeur déclaré dans le patron sémantique, une chaîne de caractères ou un entier.

1.4.4. Principes du langage MSST

Le langage MSST permet de décrire un patron syntaxique. Un patron syntaxique est composé de règles s'appliquant aux arbres sémantique issus de l'application d'un patron sémantique sur un modèle. Le lien entre un patron sémantique et un patron syntaxique est effectué à l'aide des constructeurs prédéfinis déclarés dans le patron sémantique : à chacun d'eux doit correspondre une règle d'écriture dans le patron syntaxique qui lui est associé.

Il existe deux types de règles syntaxiques :

- Les *règles externes* : il y en a autant que de constructeurs prédéfinis déclarées dans le patron sémantique associé(5);
- Les *règles internes* : elles ne sont pas associées à des catégories syntaxiques du patron sémantique associé et ne peuvent qu'être invoquées par d'autres règles.

Une règle externe est en général «frontal» de plusieurs règles internes. Cela permet de générer des constructions compliquées ne pouvant être produites par la seule règle externe.

On applique toujours un patron syntaxique à un arbre sémantique et on range toujours le résultat dans un fichier ascii. Aussi, ces deux paramètres (l'arbre et le fichier) ne sont jamais déclarés explicitement dans une règle. On les manipule comme suit :

- les lectures concernent un arbre sémantique,
- les écritures s'appliquent sur un fichier.

1.5. Principes communs des grammaires des langage dans *M*eta*S*cribe

Bien que les langages MS caractérisent des entités assez différentes, quelques points communs de construction en facilitent l'utilisation :

- Ce sont des langages fortement déclaratifs; cela simplifie les vérifications effectuées dynamiquement lors de la définition des entités composant un formalisme, un modèle, un patron sémantique ou syntaxique;
- Une définition est divisée en plusieurs fichiers : un fichier principal dans lequel on trouve au minimum une partie déclarative et un ensemble de fichiers annexes contenant les définitions des entités composant un formalisme, un modèle, un patron sémantique ou syntaxique;

(5) Il n'est pas exclu que certaines de ces règles soient vide, par exemple pour ignorer un constructeur prédéfini.

Par ailleurs, les constructions syntaxiques suivantes sont contenues dans la BNF de tous les langages MS :

<table>
<tr><td>char_alpha(6)</td><td>::=</td><td>A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | _</td></tr>
<tr><td>char_num</td><td>::=</td><td>0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9</td></tr>
<tr><td>char</td><td>::=</td><td>char_alpha | char_num</td></tr>
<tr><td>identifier(7)</td><td>::=</td><td>alpha_char{char}</td></tr>
<tr><td>ident_list</td><td>::=</td><td>identifier |
identifier, ident_list</td></tr>
<tr><td>one_comment_line</td><td>::=</td><td>//{char}(8)</td></tr>
<tr><td>valeur</td><td>::=</td><td>valeur_positive |
valeur_negative</td></tr>
<tr><td>valeur_positive</td><td>:=</td><td>char_alpha{char_alpha}</td></tr>
<tr><td>valeur_negative</td><td>::=</td><td>-valeur_positive</td></tr>
<tr><td>chaine_qcq</td><td>::=</td><td>'{char}'</td></tr>
<tr><td>crochet_ouvrant(9)</td><td>::=</td><td>[</td></tr>
<tr><td>crochet_fermant</td><td>::=</td><td>]</td></tr>
</table>

Les références à ces lexèmes sont indiquées en italique dans la BNF des langages MS.

Remarques

Pour toutes les grammaires définies dans le présent document :

- Les caractères espace, tabulation et retour chariot sont considérés comme des séparateurs sauf dans les chaînes de caractères;
- les mots clefs et les délimiteurs des langages sont indiqués comme ceci : **MOT_CLEF**.

2. Description des formalismes (langage MSF)

Le langage MSF permet de déclarer les éléments que l'on peut trouver dans un formalisme, à savoir :

- des classes de nœuds ou de liens,
- des règles de connexion entre classes (nœuds et liens),
- des attributs associées, soit à une classe (nœud ou lien), soit à un modèle (attributs globaux). Les attributs sont typés.

2.1. BNF d'un fichier MSF principal

Le fichier principal contient la déclaration de toutes les entités composant le formalisme (nœuds et liens) et des informations globales (nom du formalisme, attributs du modèle, constructeurs acceptés sur les arbres d'expression…).

(6) Majuscules et minuscules sont équivalentes.
(7) Un identificateur ne peut pas être un mot clef du langage.
(8) Un commentaire se termine à la fin de la ligne à laquelle il a commencé.
(9) Pour ne pas interférer avec la notation des BNF ([expr] correspond à une expression facultative).

Structure générale du fichier

MSF_main_file	::=	**MAIN_FILE** formalism_name nodes_and_links global_attrib_list possible_tree_operators file_list

Attribution du nom du formalisme

formalism_name	::=	**FORMALISM (***chaine_qcq***);**

Déclaration des entités (noeuds et liens)

On y définit la liste des classes (nœuds et liens) que l'on peut trouver dans le formalisme.

nodes_and_links	::=	**ENTITY_LIST** decl_entity_list**;**
decl_entity_list	::=	decl_entity {**,** decl_entity}
decl_entity	::=	*identifier* **: NODE** | *identifier* **: LINK**

Définition des attributs globaux du formalisme

On y définit la liste des attributs globaux du formalisme. Ces attributs caractérisent des informations concernant l'ensemble du formalisme.

global_attrib_list	::=	**GLOBAL_ATTRIBUTES** attribute_list; **END;**
attribute_list	::=	**NONE** | one_attribute {one_attribute}
one_attribute	::=	**ATTRIBUTE** type_attribute **:** *identifier***;**
type_attribute	::=	**INTEGER** | **STRING** | **TEXT** | **BOOLEAN** | **EXPRESSION**

Définition des opérateurs prédéfinis que l'on peut trouver dans un arbre syntaxique

On y définit la liste des «constructions prédéfinies» que l'on trouvera dans les nœuds des attributs de type «expression».

possible_tree_operators	::=	**CONSTRUCTION_LIST (**Possible_operator_arg**);**
Possible_operator_arg	::=	**NONE** | *ident_list*

Références aux fichiers annexes

On y définit la liste des fichiers annexes décrivant les entités déclarées pour le formalisme[(10)].

file_list	::=	**FILE_LIST** one_file_name{**,** one_file_name}**;**
one_file_name	::=	**FILE (***chaine_qcq***)**

(10) Un même fichier annexe peut parfaitement être référencé dans la définition de deux formalismes différents.

2.2. BNF d'un fichier MSF annexe

Un fichier annexe contient au moins la définition d'une entité du formalisme (nœuds ou liens). La définition d'une entité ne peut pas être répartie sur plusieurs fichiers annexes.

Structure générale d'un fichier annexe

```
MSF_annex_file        ::= ANNEX_FILE
                           one_entity
                          { one_entity}
one_entity            ::= one_link |
                          one_node
```

Définition d'un lien

Un lien possède un ensemble fini d'attributs. Le nom du lien que l'on définit doit être répété à la fin de la définition.

```
one_link              ::= LINK (identifier) IS
                            ATTRIBUTE_LIST
                              attribute_list(11)
                            END;
                          END identifier;
```

Définition d'un nœud

Un nœud possède un ensemble fini d'attributs. On indique les liens auquel il peut être connecté. Le nom du nœud que l'on définit doit être répété à la fin de la définition.

```
one_node              ::= NODE (identifier) IS
                            ATTRIBUTE_LIST
                              attribute_list(12)
                            END;
                            CONNECTABILITY_LIST
                              connectability_list
                            END;
                          END identifier;
connectability_list   ::= NONE |
                          one_connection {one_connection}
one_connection        ::= WITH identifier
                            DIRECTION dir_type,
                            MAXIMUM max_type;
dir_type              ::= IN |
                          OUT |
                          NON_ORIENTED
max_type              ::= NONE |
                          valeur_positive
```

On ne peut définir de règle de connectabilité avec un lien qui n'a pas été déclaré. Si des liens sont destiné à des connexions multiples sans direction (un lien relie éventuellement plus de deux nœuds), le lien est de type **NON_ORIENTED**.

(11) voir la BNF page 13.
(12) voir la BNF page 13.

Si le maximum de connexion est **NONE**, cela signifie que le nœuds peut posséder un nombre arbitrairement élevé de liens du type considéré pour la direction considérée. Si une valeur est spécifiée, elle doit être ***strictement*** positive.

Exemple 1 : utilisation des directions «NON_ORIENTED» dans la définition des règles de connectabilité.

Considérons la définition du formalisme décrivant un réseau de machines connectées par un bus. Considérons le lien «bus» définit comme suit :

```
link (BUS) is
   attribute_list
      none
   end;
end BUS;
```

Nous pouvons définir un nœud «machine» comme suit :

```
node (MACHINE) is
   attribute_list
      // les attributs
      none
   end;
   connectability_list
      with BUS direction NON_ORIENTED, maximum 1;
   end;
end MACHINE;
```

Il est ainsi possible de relier un nombre arbitraire(13) de machine sur un seul bus. Chaque machine ne peut être connectée qu'à un seul bus de donnée (i.e. appartenir à un seul réseau).

3. Description des modèles (langage MSM)

Le langage MSM permet de décrire un modèle d'un formalisme donné (préalablement défini en MSF). A partir d'une telle description, ***M**eta**S**cribe* assure :

- la construction d'une représentation en mémoire manipulable au moyen d'une API;
- la vérification de la conformité au formalisme, pendant cette construction.

La description d'un modèle est constitué :

- de l'instanciation des entités (nœuds ou liens) définis dans le formalisme,
- de l'association d'une valeur aux attributs (qu'ils soient globaux ou liés à une instance de nœud ou de lien),
- de la connexion des nœuds entre eux par l'intermédiaire de liens.

3.1. BNF d'un fichier MSM principal

Un fichier principal MSM n'est pas uniquement dédié à la déclaration des entités présentes dans le modèle et peut contenir la définition des entités qui le composent (nœuds, liens). Ainsi, il est possible de produire la description d'un modèle dans un unique fichier. Le partitionnement en fichier principal et fichiers annexes est respecté par soucis d'unicité par rapport aux autres langages MS.

(13) L'analyseur syntaxique du langage MSF exige qu'au moins deux nœuds soient connectés de cette manière.

Structure générale du fichier

main_model_description	::=	**MAIN_FILE** global_information_def model_body[14]

Partie déclarative

La partie déclarative précise le nom du formalisme associé au modèle ainsi que la valeur des attributs globaux du modèle.

global_information_def	::=	**FORMALISM (***chaine_qcq***);** **WHERE (**atr_affectation**);**
atr_affectation	:=	**NONE** | one_atr_affectation {**,** one_atr_affectation}
one_atr_affectation	::=	**ATTRIBUTE** *identifier* **=>** expression

Les expressions associées aux attributs sont forcément statiques (valeurs immédiates).

expression	::=	*valeur* | *chaine_qcq* | text_expr | expression_expr | bool_expr
text_expr	::=	**(***chaine_qcq* {**,** *chaine_qcq*}**)**
bool_expr	::=	**TRUE** | **FALSE**

Les arbres d'expression sont d'arité variable (pour plus de facilité dans l'écriture). Les nœuds non terminaux contiennent forcément une construction prédéfinie dans le formalisme suivit d'une liste de fils (au moins un).

Les feuilles (nœuds terminaux) peuvent contenir une expression du type suivant :

- chaîne de caractères,
- entier,
- construction prédéfinie déclaré dans le formalisme (voir page 13).

expression_expr	::=	**SY_EMPTY** | **SY_LEAF (***chaine_qcq***)** | **SY_LEAF (***valeur***)** | **SY_LEAF (***identifier***)** | **SY_NODE (***identifier* **:** expression_expr_list**)**
expression_expr_list	::=	expression_expr {**,** expression_expr}

3.2. BNF d'un fichier MSM annexe

MSM permet de partitionner le fichier de description en un ensemble de fichiers annexes avec une «profondeur» arbitrairement élevée : un fichier annexe peut lui-même référencer un autre fichier MSM annexe[15]. Cela peut s'avérer utile dans le cas de modèles hiérarchiques.

---

[14] La BNF est donnée dans la description d'un fichier annexe, section 3.2., page 16.

[15] *ATTENTION* : aucune détection des cycles dans les références de fichiers MSM annexe n'est pour l'instant effectuée.

Structure générale du fichier

annex_model_description	:=	**ANNEX_FILE** model_body
model_body	::=	body_or_file_ref {body_or_file_ref}

Déclaration d'une entité du modèle ou référence à un fichier annexe

Lors de la définition d'une entité du modèle (nœud ou lien), il faut :

- impérativement lui attribuer un identificateur ***unique***(16) sous la forme d'une chaîne de caractères;
- éventuellement préciser la valeur des attributs.

body_or_file_ref	::=	**FILE (***chaine_qcq***);** |
		NODE *chaine_qcq* **IS** *identifier*
		    WHERE (attr_affectation**);** |
		LINK *chaine_qcq* **IS** *identifier*
		    WHERE (attr_affectation**)**
		    RELATE relation_definition**;** |
		one_comment_line

Dans le cas ou l'entité est ***un lien***, il faut en outre préciser les nœuds qu'il relie. Il existe deux façon de procéder selon le type de lien auquel on a affaire :

i. liens directionnel : le lien est une connexion point à point entre deux nœuds pour laquelle une direction est définie,
ii. liens sans direction : le lien relie deux nœuds ou plus sans qu'un sens soit défini (relation non ordonnée).

Les relations sont testées conformément aux règles de connectabilité définies dans le formalisme (voir "Définition d'un nœud", page 14). Deux nœuds reliés selon (i)peuvent accepter le lien en entrée (nœud de départ) et en sortie (nœud d'arrivée) ou en entrée et en sortie (pour les deux nœuds). Deux nœuds reliés selon (ii) ne peuvent accepter le lien qu'en mode «entrée et sortie».

relation_definition	::=	entity_ref **TO** entity_ref | **TOGETHER** entity_ref**,** entity_ref **{,** entity_ref}

Il existe deux manières de référencer une unité : soit via son nom (une chaîne de caractères), soit par son nom et son type. La seconde technique permet d'accélérer la construction d'une représentation en mémoire(17).

entity_ref	::=	*identifier* **:** *chaine_qcq* | *chaine_qcq*

4. Description des patrons sémantiques (langage MSSM)

Le résultat d'un patron sémantique est un ensemble d'arbres d'expressions auxquels seront ensuite appliqués un patron syntaxique. Règles sémantique et arbres statiques sont paramétrables. Les paramètres sont typés par :

(16) Le format de cette chaîne de caractères est libre mais des conventions seront précisées dans le cas où l'on souhaite intégrer le moteur de réécriture produit dans l'environnement FrameKit.

(17) Statistiquement, si le formalisme possède N classes de nœuds et L classes de liens, le facteur d'accélération sera de l'ordre de N+L.

- une chaîne de caractères,
- un entier,
- un arbre syntaxique au format décrit dans le langage MSM,
- un arbre sémantique au format de ceux produits par une règle sémantique ou un arbre statique,
- un constructeur prédéfini identifié dans la partie déclarative du patron sémantique,
- une référence à une entité du modèle (nœud ou connecteur).

Ces paramètres sont référencés dans le corps de la règle sémantique ou de l'arbre statique.

Les règles sémantiques

Il existe deux types de règles sémantiques : les règles principales qui sont toutes appliquées par le moteur de réécriture, et les règles secondaires qui correspondent à des «sous-programmes» invoqués depuis une autre règle (secondaire ou une principale). Les règles sémantiques sont typées par le type de valeur qu'elles rendent :

- des chaînes de caractères,
- des entiers,
- des arbres sémantiques,
- des constructeurs prédéfinis,
- aucune valeur[18].

Le corps d'une règle sémantique permet de manipuler les entités d'un formalisme ainsi que les attributs qui leur sont associés. L'utilisation de variables locales est une facilité offerte pour la construction de résultats intermédiaires.

Les arbres statiques

Les arbres statiques sont des macros permettant de définir des constantes de type arbre sémantique.

Structure d'un arbre produit par l'application d'un patron sémantique

Les arbres produits par l'application d'un patron sémantique sont d'arité quelconque. Leurs nœuds contiennent *au moins une* des trois informations suivantes : un constructeur déclaré dans le patron sémantique, une chaîne de caractères ou un entier.

4.1. BNF d'un fichier MSSM principal

Un fichier MSSM principal est dédié aux déclarations nécessaires à la construction d'un patron sémantique.

[18] C'est en particulier le cas des règles principales qui, par définition, ne rendent pas de valeur.

Structure générale du fichier

le paramètre situé après la commande **MAIN_FILE** permet de nommer le patron sémantique et de désigner le formalisme qu'il accepte en entrée.

main_sem_pattern_def	::=	**MAIN_FILE (***identifier* **FOR** *chaine_qcq***)** global_declarations annex_file_list**;**

Les déclarations globales concernent :

- les règles sémantiques,
- les règles sémantiques principales,
- les arbres sémantiques,
- les constructeurs prédéfinis,
- les variables globales, accessibles depuis n'importe quelle règle ou arbre sémantique.

Pour les règles sémantiques et les arbres sémantiques, il faut indiquer un profil (paramètres, valeur de retour si besoin est). Les règles sémantiques principales doivent être de type **VOID** (pas de valeur de retour).

global_declaration	::=	**SEMANTIC_RULE_LIST IS** sm_rule_list **END;** **SEMANTIC_TREE_LIST IS** sm_tree_list **END;** **CONSTRUCTRUCTOR_LIST IS** *ident_list***;** **GLOBAL_VARIABLE** global_var_list **END;**
sm_rule_list	::=	one_sm_rule_decl {**,** one_sm_rule_decl}
sm_tree_list	::=	one_sm_stree_decl {**,** one_sm_stree_decl}
global_var_list	::=	**NONE** | *ident_list*

Tous les fichiers de annexes d'un patron sémantique doivent être référencés dans le fichier principal.

annex_file_list	::=	**INCLUDE_FILE (***chaine_qcq***);** {**INCLUDE_FILE (***chaine_qcq***);**}

Déclaration d'un arbre sémantique statique

Un arbre sémantique statique peut avoir un nombre arbitrairement élevé de paramètres et rend toujours un arbre sémantique.

one_sm_stree_decl	::=	*identifier* **(**formal_param_decl**)**
formal_param_decl	::=	**NONE** | one_formal_param_decl {**,** one_formal_param_decl}
one_formal_param_decl	::=	*identifier* **:** possible_param_types
possible_param_types	::=	**STRING** | **INTEGER** | **SYNTACTIC_TREE** | **SEMANTIC_TREE** | **SEMANTIC_CONSTRUCTOR** | **MODEL_ENTITY**

La signification des types de paramètres est la suivante :

- ***string*** : chaîne de caractères,
- ***integer*** : entier quelconque,
- ***syntactic_tree*** : arbres syntaxiques de description d'un attribut de type expression au sens défini page 16,
- ***semantic_tree*** : référence à un arbre sémantique,
- ***semantic_constructor*** : un constructeur prédéfini,
- ***model_entity*** : une référence à une entité du modèle (nœud ou lien) dont on souhaite exploiter des informations.

Déclaration d'une règle sémantique

Une règle sémantique peut avoir un nombre arbitrairement élevé de paramètres et rend une valeur du type précisé dans le profil. La déclaration des paramètres est identique à celle d'un arbre sémantique statique (voir page 19).

one_sm_rule_decl	::=	[**MAIN**[19]]*identifier* **(**formal_param_decl**)** **RETURN** possible_rv_types
possible_rv_types	::=	**STRING** | **INTEGER** | **SEMANTIC_TREE** | **SEMANTIC_CONSTRUCTOR** | **MODEL_ENTITY** | **VOID**

La signification des types de valeur de retour est la suivante :

- ***string*** : chaîne de caractères,
- ***integer*** : entier quelconque,
- ***semantic_tree*** : référence à un arbre sémantique,
- ***semantic_constructor*** : un constructeur prédéfini,
- ***model_entity*** : une référence sur une entité du modèle (un nœud ou un lien),
- ***void*** : pas de valeur de retour. la règle peut alors être une règle principale ou bien être invoquée «comme une procédure».

4.2. BNF d'un fichier MSSM annexe

Les fichiers annexes en MSSM ne peuvent pas référencer d'autres fichiers annexes. Ils contiennent forcément des corps de règles sémantiques ou d'arbres sémantiques statiques. Aucune déclaration ne peut y être faite.

Structure générale du fichier

main_sem_pattern_def	::=	**ANNEX_FILE** list_of_sem_entities
list_of_sem_entities	::=	one_entity {one_entity}
one_entity	::=	semantic_rule_body | semantic_tree_body

(19) Cette option permet de définir ainsi des règle sémantiques principales. Leur valeur de retour est alors forcément de type «void» et elle ne doivent pas comporter de paramètres.

Corps d'un arbre sémantique statique

Un arbre sémantique statique rend toujours une valeur de type ***semantic_tree*** (au sens donné page 20). La construction de cet arbre peut-être paramétrée comme pour une macro-fonction.

semantic_tree_body	::=	**SEMANTIC_TREE** one_sm_stree_decl[20] **IS** body_of_sm_tree **END;**
body_of_sm_tree	::=	**SEMANTIC_NODE (**node_content**:**sm_tree_expr_list**)** | **SEMANTIC_LEAF (**node_content**)**
sm_tree_expr_list	::=	sm_tree_expr {**,** sm_tree_expr} | body_of_sm_tree [**,**sm_tree_expr_list]

La catégorie syntaxique ***node_content*** est définie page 28.

Corps d'une règle sémantique

Le corps d'une règle sémantique implémente un traitement portant sur les données du formalisme, des paramètres de la règle ou des variables locales.

semantic_rule_body	::=	**SEMANTIC_RULE** one_sm_rule_decl **IS**
		[{variable_declaration}]
		BEGIN
		one_smr_instruction
		{ one_smr_instruction}
		END;

Il est possible de déclarer un nombre de variables arbitrairement élevé mais seuls certains types sont autorisés :

- ***string*** : une chaîne de caractères;
- ***integer*** : un entier;
- ***semantic_tree*** : un arbre sémantique (en cours de construction);
- ***model_entity*** : une référence à une entité du modèle (nœud ou lien),
- ***syntactic_tree*** : un arbre d'expression (issu d'un attribut de type expression).

Certains des types disponibles pour les paramètres (text, syntactic_tree...) ne le sont plus pour les variables. Cela se justifie car il s'agit de structures liées au modèle source (qu'il n'est par conséquent pas raisonnable de manipuler en écriture).

variable_declaration	::=	*identifier* **:** possible_variable_type**;**
possible_variable_type	::=	**STRING** | **INTEGER** | **SEMANTIC_TREE** | **MODEL_ENTITY** | **SYNTACTIC_TREE**

Les instructions de base que l'on peut trouver dans une règle sémantique sont les suivantes :

- instruction vide;

[20] Voir la BNF de description du profil d'un arbre sémantique statique page 19

- sortie de la règle (en transmettant si besoin est, une valeur de retour correspondant au type de la règle);[21]
- tests et séquences de tests. Pour cela, un certain nombre d'opérateurs booléens sont disponibles. Il permettent de :
  - réaliser les fonction booléennes classiques (comparaisons, not, or, and);
  - tester l'existence d'attributs dans le modèle (i.e. ont-il été définis). Ces attributs peuvent aussi bien être globaux qu'associés à une entité (nœud ou lien) du modèle;
  - tester l'existence d'instances d'une entité (nœud ou lien) dans un type donné.
- boucle énumérative utile pour le parcours en largeur d'arbres ou de chaînes de caractères contenues dans un attribut de type text. L'indice de la boucle est toujours considéré comme un entier compris prenant ses valeurs dans un intervalle [i1 .. i2]. Il est possible de spécifier une valeur i2 < i1. Dans ce cas, on considèrera que l'intervalle est vide et aucune itération ne sera effectuée;
- création dynamique d'un attribut du modèle (global ou lié à une entité). Seuls des attributs de type entier, chaîne de caractères ou booléen peuvent être créés de la sorte. Cela permet d'associer au modèle des informations temporaires que l'on a calculé et que l'on souhaitera utiliser plus tard (dans une autre règle par exemple);
- affichage d'un message;
- affichage d'un message d'erreur;
- application d'un patron syntaxique à une expression dans un fichier donné (directive `GENERATE`) : la première expression rend forcément un arbre sémantique, la seconde une chaîne de caractères identifiant le fichier cible dans lequel le source sera généré.

```
one_smr_instruction    ::= NOP; |
                           RETURN [non_bool_expr]; |
                           IF bool_expr THEN
                             sm_instruction_list
                           {ELSIF bool_expr THEN
                             sm_instruction_list}
                           [ELSE
                             sm_instruction_list]
                           END IF; |
                           FOR identifier IN integer_expr .. integer_expr DO
                             sm_instruction_list
                           END FOR; |
                           SET_GLOBAL_TEMP_ATTRIBUTE (identifier, atr_val); |
                           SET_TEMP_ATTRIBUTE (identifier, model_entity_expr,
                           atr_val); |
                           sm_rule_reference; (22) |
                           identifier := non_bool_expr; |
```

[21] Toute règle doit comporter au moins une instruction de ce type.
[22] Voir définition page 28.

		MESSAGE (string_expr**) ;** |
		ERROR (string_expr**) ;** |
		GENERATE non_bool_expr **IN** non_bool_expr**;**
atr_val	::=	non_bool_expr |
		bool_expr

Outre les opérateurs booléen habituel (comparaisons, opérations booléennes), on notera la présence d'opérateurs permettant :

- de savoir si un attribut a été défini (i.e. si un accès à sa valeur ne provoquera pas une erreur);
- de savoir s'il existe des instances de classes d'entités dans le modèle (un opérateur pour les nœuds, un opérateur pour les liens);
- de récupérer la valeur d'un attribut du modèle de type booléen;
- de savoir si le nœud d'un arbre d'expression (attributs de type expression) contient une valeur entière, une valeur chaîne de caractère ou une valeur «construction prédéfinie»).

bool_expr	::=	non_bool_expr comparison_operator non_bool_expr |
		NOT (bool_expr**)** |
		(bool_expr**) OR (**bool_expr**)** |
		(bool_expr**) AND (**bool_expr**)** |
		IS_ATR_DEFINED (attribute_reference**)** |
		IS_NODE_INSTANCIATED (*identifier***)** |
		IS_LINK_INSTANCIATED (*identifier***)** |
		$ATRV_BOOL (attribute_reference**)** |
		IS_INTEGER (sy_tree_expr**)** |
		IS_TRSING (sy_tree_expr**)** |
		IS_CONSTRUCTION (sy_tree_expr**)**
comparison_operator	::=	**>** | **<** | **>=** | **<=** | **/=** | **=**

Expressions élémentaires pour la définition des corps de règles ou d'arbres sémantiques

Nous regroupons ici tout ce qui a trait aux expressions que l'on peut rencontrer, tant dans la définition d'une règle sémantique, que dans celle d'un arbre sémantique statique. Ces expressions manipulent différents types de données :

- des constructeurs prédéfinis (*sem_constructor_expr*),
- des constructions prédéfinies du formalisme (*prd_constr_expression*),
- des chaînes de caractères (*string_expr*),
- des entiers (*integer_expr*),
- des arbres syntaxiques associés aux expressions (*sy_tree_expr*),
- des suites de chaînes de caractères (*text_expr*),
- des références à des attributs du modèle, qu'ils soient globaux ou associés à une entité (*attribute_reference*),
- des références sur des nœuds ou liens du modèle (*model_entity_reference*),
- des arbres sémantiques (*sm_tree_expr*).

Les constructeurs prédéfinis sont référencés sous la forme d'une valeur immédiate ou via une variable de type ***semantic_constructor***.

sem_constructor_expr ::= *identifier* |
$SMC (*identifier***)**

Les constructions prédéfinies d'un formalisme (que l'on trouve dans les nœuds des arbres associés aux attributs de type expression) peuvent être référencés de manière immédiate ou extrait du contenu du nœud courant d'un arbre syntaxique.

prd_constr_expression ::= %*identifier* |
$CONST_SY_NODE (sy_tree_expr**)** [23]

Les chaînes de caractères peuvent être référencées sous la forme d'une valeur immédiate ou calculées comme suit :

- extraction d'une ligne d'un attribut de type ***text***;
- transformation d'un entier en la chaîne de caractère correspondante;
- concaténation de deux sous-expressions de type chaîne de caractères;
- transformation en majuscule d'une expression de type chaîne de caractères;
- transformation en minuscules d'une expression de type chaîne de caractères;
- transformation en une chaîne de caractères du nom de la classe d'une entité (lien ou nœud);
- transformation en une chaîne de caractères de l'identificateur (forcément unique) d'une entité du modèle (lien ou nœud);
- extraction d'une étiquette d'un nœud d'un arbre syntaxique de type ***expression*** (forcément de type chaîne de caractère)[24];
- référence à une variable de type ***string***;
- à partir de la valeur d'un attribut de type ***string***.

string_expr ::= *chaine_qcq* |
TEXT_LINE (text_expr, integer_expr**)** |
TO_STRING (integer_expr**)** |
{string_expr **&** string_expr**}** |
UPPER (string_expr**)** |
LOWER (string_expr**)** |
ENTITY_CLASS (non_bool_expr**)** |
ENTITY_IDENTIFIER (non_bool_expr**)** |
$STR_SY_NODE (sy_tree_expr**)** |
$STR (*identifier***)** |
$ATRV_STR (attribute_reference[25]**)**

Les entiers peuvent être référencés sous la forme d'une valeur immédiate ou calculés comme suit :

- résultat d'une opération arrithmétique entre deux expressions entières;
- nombre de fils d'un nœud de type ***expression***;

(23) Il y aura une erreur lors de l'exécution de cette instruction si la racine de l'arbre syntaxique ne contient pas une valeur de type construction prédéfinie.

(24) Il y aura une erreur lors de l'exécution de cette instruction si la racine de l'arbre syntaxique ne contient pas une valeur de type chaine de caractères.

(25) Voir la syntaxe d'une référence à un attribut page 26.

- taille en nombre de lignes d'une suite de chaîne de caractères (le type ***text***);
- nombre d'instances d'un nœud d'une classe donnée (si son identificateur est indiqué) ou dans le modèle (toutes les classes sont alors considérées);
- nombre d'instances d'un lien d'une classe donnée (si son identificateur est indiqué) ou dans le modèle (toutes les classes sont alors considérées);
- nombre total de voisin dans une classe donnée;
- nombre de voisins «non orientés ayant une classe donnée pour une entité d'un modèle (nœud ou lien);
- nombre de voisins «en sortie» ayant une classe donnée pour une entité d'un modèle (nœud ou lien);
- nombre de voisins «en entrée» ayant une classe donnée pour une entité d'un modèle (nœud ou lien);
- nombre de voisins toutes classes confondues d'une entité;
- nombre de voisins toutes classes confondues «non orientés» d'une entité;
- nombre de voisins toutes classes confondues «en sortie» d'une entité;
- nombre de voisins toutes classes confondues «en entrée» d'une entité;
- la longueur d'une chaîne de caractères;
- extraction d'une étiquette d'un nœud d'un arbre syntaxique de type ***expression*** (forcément de type entier)(26);
- référence à une variable de type ***integer***;
- extraction de la valeur d'un attribut de type ***integer***.

integer_expr ::= *valeur* |
{integer_expr math_operator integer_expr**}** |
NUMBER_OF_SY_SON (sy_tree_expr**)** |
TEXT_SIZE (text_expr**)** |
NB_NODE_INSTANCE [(*identifier***)]** |
NB_LINK_INSTANCE [(*identifier***)]** |
NB_NEIGHBOUR_HAVING_CLASS (model_entity_expr**,** *identifier***)** |
NB_NEIGHBOUR_NON_ORIENTED_HAVING_CLASS (model_entity_expr**,** *identifier***)** |
NB_NEIGHBOUR_OUT_HAVING_CLASS (model_entity_expr**,** *identifier***)** |
NB_NEIGHBOUR_IN_HAVING_CLASS (model_entity_expr**,** *identifier***)** |
NB_NEIGHBOUR (model_entity_expr**)** |
NB_NEIGHBOUR_NON_ORIENTED (model_entity_expr**)** |
NB_NEIGHBOUR_OUT (model_entity_expr**)** |
NB_NEIGHBOUR_IN (model_entity_expr**)** |
STRLEN (string_expr**)** |
$INT_SY_NODE (sy_tree_expr**)** |
$INT (*identifier***)** |
$ATRV_INT (attribute_reference**)**

math_operator ::= **+** | **-** | **/** | *****

(26) Il y aura une erreur lors de l'exécution de cette instruction si la racine de l'arbre syntaxique ne contient pas une valeur de type entier.

Les arbres d'expression syntaxique (les ***expressions***) peuvent être référencés comme suit :

- la constante immédiate signifiant «arbre d'expression vide»;
- l'arbre obtenu en supprimant les N premiers fils (Figure 7);
- l'arbre correspondant au $N^{ième}$ fils;
- référence à une variable de type ***expression***;
- à partir de la valeur d'un attribut de type ***expression***.

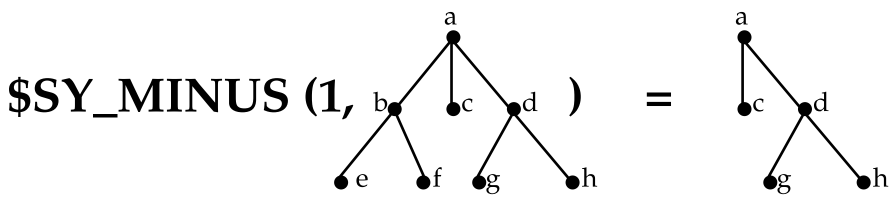

Figure 7 : Effet de l'opérateur "**`$SY_MINUS`**"sur un exemple.

Les opérateurs **`$SY_MINUS`** et **`$SY_SON`** ont été définis pour parcourir récursivement des arbres sémantiques.

sy_tree_expr ::= **`SY_EMPTY`** |
`$SY_MINUS (`integer_expr**`,`** sy_tree_expr**`)`** |
`$SY_SON (`integer_expr**`,`** sy_tree_expr**`)`** |
`$SYT (`*identifier***`)`** |
`$ATRV_SYT (`attribute_reference**`)`**

Les suites de chaînes de caractères (type ***text***) peuvent être référencés comme suit :

- la constante immédiate signifiant «texte vide»;
- référence à une variable de type ***text***;
- à partir de la valeur d'un attribut de type ***text***.

text_expr ::= **`EMPTY_TEXT`** |
`$TXT (`*identifier***`)`** |
`$ATRV_TXT (`attribute_reference**`)`**

L'extraction de la valeur associée à un attribut global du formalisme se fait par référence directe (nom de l'attribut uniquement). Dans les autres cas, il faut référencer l'attribut par rapport à une entité du modèle (nœud ou lien).

attribute_reference ::= **`ATTRIBUTE`** *identifier* |
`ATTRIBUTE` *identifier* **`FROM`** model_entity_expr

L'accès aux entités d'un modèle (nœud ou lien) se fait de la manière suivante :

- par référence au moyen de deux techniques d'identification : par nommage (l'identificateur unique) ou par le couple <*identificateur, classe*>[27];
- par position, relativement à une classe[28] (si son identificateur est précisé) ou de manière absolue (par rapport on nombre total de nœuds ou de liens);
- par rapport à une entité dont on possède la référence. On dispose des opé-

[27] La seconde technique est plus rapide à l'exécution.

[28] Le nombre total d'instance dans la classe est indiquée par les instructions **`NB_NODE_INSTANCE`** ou **`NB_LINK_INSTANCE`**.

rateurs pour extraire :
- le $N^{ième}$ voisin sans relation de précédence dans une classe donnée,
- le $N^{ième}$ voisin «non orienté» dans une classe données,
- le $N^{ième}$ voisin «en sortie» dans une classe données,
- le $N^{ième}$ voisin «en entrée» dans une classe données,
- le $N^{ième}$ voisin sans relation de précédence,
- le $N^{ième}$ voisin «non orienté»,
- le $N^{ième}$ voisin «en sortie»,
- le $N^{ième}$ voisin «en entrée».

• par référence à une variable ou un paramètre de type ***model_entity*** (voir page 20).

model_entity_expr ::= **GET_REF_MODL_ENT** **(**[*identifier*:]*chaine_qcq*) |
GET_NODE_REFERENCE ([*identifier*], integer_expr) |
GET_LINK_REFERENCE ([*identifier*], integer_expr) |
GET_NEIGHBOUR_HAVING_CLASS (model_entity_expr, integer_expr, *identifier*) |
GET_NEIGHBOUR_NON_ORIENTED_HAVING_CLASS (model_entity_expr, integer_expr, *identifier*) |
GET_NEIGHBOUR_OUT_HAVING_CLASS (model_entity_expr, integer_expr, *identifier*) |
GET_NEIGHBOUR_IN_HAVING_CLASS (model_entity_expr, integer_expr, *identifier*) |
GET_NEIGHBOUR (model_entity_expr, integer_expr) |
GET_NEIGHBOUR_NON_ORIENTED (model_entity_expr, integer_expr) |
GET_NEIGHBOUR_OUT (model_entity_expr, integer_expr) |
GET_NEIGHBOUR_IN (model_entity_expr, integer_expr) |
$MET (*identifier*)

Les expressions de type arbre sémantique peuvent se référencer comme suit :

• la constante immédiate signifiant «arbre sémantique vide»;
• la référence à un arbre sémantique statique avec affectation de valeurs aux paramètres définis;
• la création d'un arbre pour lequel un contenu est précisé (constructeur prédéfini et/ou chaîne de caractères);
• l'ajout d'un fils a la suite des fils de la racine référencée. Le fils peut-être défini par son contenu (c'est alors une feuille) soit sous la forme d'un arbre (la non_bool_expression est alors forcément de type ***semantic_tree***);
• la référence à un paramètre ou une variable locale de type ***semantic_tree*** au sens défini page 20.

sm_tree_expr ::= **EMPTY_SEMANTIC_TREE** |
SM_TREE *identifier* ([param_affectation_list]) |
CREATE_SM_TREE (node_content) |
ADD_SM_SON (sm_tree_expr, non_bool_expr) |
ADD_SM_SON (sm_tree_expr, node_content) |
MERGE_SM_TREE (non_bool_expr, non_bool_expr) |
$SMT (*identifier*)

node_content	::=	*crochet_ouvrant* [sem_constructor_expr] **#** [string_expr] **#** [integer_expr] *crochet_fermant*(29)
non_bool_expr	::=	string_expr **|** integer_expr **|** sy_tree_expr **|** sem_constructor_expr **|** sm_tree_expr **|** text_expr **|** sm_rule_reference **|** model_entity_expr
sm_rule_reference	::=	**`SM_RULE`** *identifier* **(**[param_affectation_list]**)**
param_affectation_list	::=	*identifier* **=>** non_bool_expr {**,** *identifier* **=>** non_bool_expr}

5. Description des patrons syntaxiques (langage MSST)

Un patron syntaxique permet d'appliquer un «sucre syntaxique» aux arbres sémantiques construits par le patron sémantique. Un patron syntaxique est composé de règles. Une règle s'applique à un arbre sémantique et définit une écriture de «l'expression» correspondante à l'arbre. Il existe deux types de règles syntaxiques :

- Les règles externes : il y en a autant que de constructeurs prédéfinis(30) déclarées dans le patron sémantique associé au patron syntaxique. Ces règles peuvent être invoquées de manière explicite ou implicite;
- Les règles internes : ces règles ne sont pas associées à des constructeurs prédéfinis et ne peuvent être invoquées que de manière explicite.

Une règle syntaxique externe est en général «frontal» de plusieurs règles syntaxiques internes. Cela permet de générer des constructions syntaxiques compliquées qui ne pourraient être produites par la seule règle externe associée.

5.1. BNF d'un fichier principal

Le fichier principal d'un patron syntaxique contient le nom du patron syntaxique et identifie le patron sémantique auquel il est associé.

main_syn_pattern_def	::=	main_file global_declarations syn_pattern_body(31)
global_declarations	::=	**`SYNTACTIC_PATTERN_NAME`** *chaine_qcq* **`RELATED_TO`** *chaine_qcq*

5.2. BNF d'un fichier annexe

Il est à noter que des fichiers annexes peuvent référencer d'autres fichiers annexes au même titre que le fichier principal.

(29) ATTENTION : l'un des trois champs doit au moins être indiqué.
(30) Si aucune règle n'est associée à une catégorie syntaxique, la construction du moteur de réécriture échouera.
(31) Voir la définition page 29.

Structure générale du fichier

annex_syn_pattern_def	::=	**ANNEX_FILE** syn_pattern_body
syn_pattern_body	::=	one_entity {**,** one_entity}
one_entity	::=	**INCLUDE_FILE (***chaine_qcq***)** | one_syntactic_rule

Définition d'une règle syntaxique

one_syntactic_rule	::=	**SYNTACTIC_RULE** *identifier* **IS** {one_variable_declaration} **BEGIN** syn_rule_body**;**
one_variable_declaration	::=	*identifier* **:** variable_type**;**
variable_type	::=	**STRING** | **INTEGER**
syn_rule_body	::=	**EMPTY;** | instruction_list
instruction_list	::=	one_instruction {one_instruction}

L'instruction ***empty*** indique que la règle ne produit rien, ce qui permet d'ignorer des constructeurs prédéfinis. Il est possible de déclarer des variables locales afin d'effectuer des calculs simples.

Les instructions contenues dans le corps d'une règle se divisent en plusieurs groupes :

- Les instructions d'écriture dans le fichier en cours de production(32);
- les instructions de présentation pour gérer l'indentation dans le fichier produit;
- une instruction de contrôle (test);
- une instruction pour gérer les erreurs;
- les instructions d'invocation des règles syntaxiques.

Il existe deux façons d'invoquer une règle syntaxique :

- ***invocation implicite*** : aucun identificateur de règle n'est spécifié. La règle qui est invoquée sera choisie en fonction du constructeur prédéfini de la racine de l'arbre sur lequel s'applique la règle. Cela suppose bien sûr que cet identificateur est présent dans le nœud(33);
- ***invocation explicite*** : quelque soit le contenu de la racine de l'arbre, la règle désignée par son identificateur sera appliquée.

Seules les règles externes peuvent être implicitement invoquées.

one_instruction	::=	***manipulation de variables*** *identifier* **:=** non_bool_expression**;** ***écriture dans le fichier résultat*** **PUT (**string_expression**);** | **PUT_LINE (**string_expression**);** |

(32) Celui dans lequel on génère le résultat de l'application du patron syntaxique.

(33) Un contrôle dynamique est effectué et une erreur est générée si le patron sémantique n'a pas construit un arbre respectant cette convention.

```
PUT_UPPER (string_expression); |
PUT_LOWER (string_expression); |
PUT_LINE_UPPER (string_expression); |
PUT_LINE_LOWER (string_expression); |
NEW_LINE (valeur_positive); |
      présentation
INCREASE_MARGING; |
ALING_MARGING_ON_WORD (valeur_positive); |
DECREASE_MARGING; |
      contrôle
IF bool_expr THEN
  instruction_list
{ELSIF bool_expr THEN
  instruction_list}
[ELSE
  instruction_list]
END IF; |
      gestion des erreurs
ERROR (chaine_qcq); |
      invocation des règles syntaxiques
APPLY (subtree_designation); |
APPLY identifier (subtree_designation);
```

Les instructions élémentaires correspondent aux traitements suivants :

- `PUT` : écriture dans le fichier cible de la chaîne de caractère transmise en paramètre,
- `PUT_LINE` : écriture dans le fichier cible de la chaîne de caractère transmise en paramètre suivit d'un passage à la ligne suivante,
- `PUT_LOWER` : écriture dans le fichier cible de la chaîne de caractère transmise en paramètre après sa conversion en minuscules,
- `PUT_UPPER` : écriture dans le fichier cible de la chaîne de caractère transmise en paramètre après sa conversion en majuscules,
- `PUT_LINE_LOWER` : écriture dans le fichier cible de la chaîne de caractère transmise en paramètre après sa conversion en minuscules suivit d'un passage à la ligne suivante;
- `PUT_LINE_UPPER` : écriture dans le fichier cible de la chaîne de caractère transmise en paramètre après sa conversion en majuscules suivit d'un passage à la ligne suivante;
- `NEW_LINE` : insère des lignes vides dans le fichier cible,
- `INCREASE_MARGING` : provoque un décalage à droite(34) de la marge courante du fichier cible. Ce décalage est applicable à la ligne courante si rien n'y a déjà été écrit. Dans le cas contraire, il s'appliquera à la ligne suivante,
- `ALING_MARGING_ON_WORD` : positionne la marge courante sous le début du N^{ième} mot de la dernière ligne non vide. Ce décalage est applicable à la ligne courante si rien n'y a déjà été écrit. Dans le cas contraire, il s'appliquera à la ligne suivante. Si la dernière ligne non vide ne contient pas assez de mots, une erreur est générée.

(34) Le nombre d'espaces du décalage est paramétré lors de l'invocation du moteur de réécriture.

- `DECREASE_MARGING` : provoque un décalage à gauche de la marge courante du fichier cible. Ce décalage est applicable à la ligne courante si rien n'y a déjà été écrit. Dans le cas contraire, il s'appliquera à la ligne suivante,
- `ERROR` : Affichage d'un message et arrêt du moteur de réécriture.

Accès à l'arbre sémantique transmis en paramètre

Toute règle syntaxique s'applique à un arbre sémantique implicitement transmis en paramètre. Cet arbre peut-être désigné et manipulé via les opérateurs :

- ***$0*** représente l'arbre complet;
- ***$N*** représente le sous-arbre correspondant au N^ième^ fils de la racine;
- ***$#*** désigne le nombre de fils de la racine de l'arbre;
- ***$SEM(N)*** désigne le constructeur prédéfini associée à la racine du sous-arbres correspondant au N^ième^ fils ("$SEM(0)" désigne constructeur prédéfini associé à la racine de l'arbre passé en paramètre).
- ***$STR (N)*** désigne la chaine de caractères associée à la racine du sous-arbres correspondant au N^ième^ fils ("$STR(0)" désigne la chaîne de caractère associée à la racine de l'arbre passé en paramètre);
- ***$INT(N)*** désigne l'entier associée à la racine du sous-arbres correspondant au N^ième^ fils ("$INT(0)" désigne l'entier associé à la racine de l'arbre passé en paramètre).

En MSST, le parcours des arbres sémantiques se fait récursivement. L'opérateur ***$N**** permet de construire l'arbre correspondant à "$0" ôté des N-1 premiers fils (Figure 8).

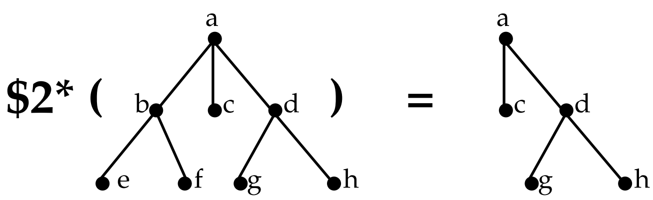

Figure 8 : Effet de l'opérateur "**`$N*`**"sur un exemple.

subtree_designation	::=	**`$`** *valeur_positive* \| **`$`** *valeur_positive* **`*`**

On peut également manipuler des expressions simples :

- des constructeurs prédéfinis (en lecture seulement),
- des chaînes de caractères (transformation d'entier en une suite de caractères, lecture de variables, concaténation etc.),
- des entiers (manipulation de variables, opérations élémentaires etc.).

sem_constructor_expr	::=	*identifier* | **`$SEM (`***valeur_positive***`)`**
string_expression	::=	*chaine_qcq* | **`$STR (`***valeur_positive***`)`** \| **`$STR (`***identifier***`)`** [35] |

(35) Référence à la variable identifier de type chaîne de caractères.

$STR_INT (*valeur_positive***)** [36] |
{string_expression **&** string_expression**}** |
TO_STRING (integer_expression**)**

integer_expression ::= *valeur* |
$# |
$INT (*valeur_positive***)** |
$INT (*identifier***)** [37] |
{integer_expression math_operator integer_expression**}** |
STRLEN (string_expression**)**

math_operator ::= **+** |
- |
/ |
*****

Conditions de l'instruction de test

Les expressions booléennes de l'instruction de test sont classiques. Les opérateurs de comparaison s'appliquent :

- à des chaînes de caractères ;
- à des constructeurs prédéfinis du patron sémantique (test du constructeur prédéfini contenu dans un nœud de l'arbre);
- à des entiers (pour tester le nombre de fils d'un arbre sémantique).

Les opérateurs de comparaison lexicographique (<, >, ≤, ≥) ne s'appliquent qu'aux chaînes de caractères et aux entiers. Les constructeurs prédéfinis déclarées dans un patron sémantique n'étant pas ordonnés.

bool_expr ::= bool_argument bool_operator bool_argument |
(bool_expr**) OR (**bool_expr**)** |
(bool_expr**) AND (**bool_expr**)** |
NOT (bool_expr**)**

bool_argument ::= string_expression |
sem_constructor_expr |
integer_expression

bool_operator ::= **=** |
/= |
< |
<= |
> |
>=

6. Illustration de l'utilisation de *M*eta*S*cribe à travers un exemple simple

Pour utiliser *M*eta*S*cribe, il faut caractériser les entités du triplet <*formalisme, patron sémantique, patron syntaxique*> dans l'ordre suivant :

1) Définition du formalisme,
2) Définition du patron sémantique,

[36] `$str_int (1)` est équivalent à `to_string ($int (1))`.
[37] Référence à la variable identifier de type entier.

3) Définition du patron syntaxique.

Un patron sémantique se définit par rapport à un formalisme puisqu'il décrit des manipulation sur les entités composant ce formalisme. De même, la définition d'un patron syntaxique se fait par rapport à des éléments caractérisés au niveau d'un patron sémantique (il y a forcément association d'un patron syntaxique à un patron sémantique).

Nous considérons ici un exemple extrêmement simple dont l'objectif est d'illustrer la philosophie d'utilisation de *M*eta*S*cribe. Il s'agit de réaliser un outil calculant des éléments statistiques simples à partir d'un réseau de Petri de type place/transition. Les éléments que l'on souhaite afficher sont :

- le nombre de places avec la liste des noms de places,
- le nombre de transitions avec la liste des noms de transitions.

Dans ce qui suit, nous donnons la description MSF, MSSM, MSST de cet exemple et montrons un exemple d'exécution.

6.1. La description du formalisme

Le formalisme `RDP_P_T` que nous construisons est composé de deux types de nœuds (places et transitions) et d'un type de liens (arcs). Il comporte un seul attribut global de type chaîne de caractères indiquant le nom du modèle. On considère qu'il n'y a pas de construction source prédéfinie (i.e. aucun attribut n'est de type expression). Le contenu du fichier principal est donné ci après.

```
main_file
formalism ('RDP_P_T');

// =========================================================
// Les entites d'un RdP Place/Transition

entity_list
   PLACE : node,
   TRANSITION : node,
   ARC : link;

// =========================================================
// Les attributs globaux d'un RdP Place/Transition

global_attributes
   attribute string  : NET_NAME;
end;

// =========================================================
// Pas d'operateurs car on n'a pas d'attribut de type expression.

construction_list (NONE);

// =========================================================
// L'unique Fichier annexe

file_list
   file ('rdp_form.annex.msf');
```

Les nœuds possèdent deux attributs : un nom (chaîne de caractères) et un marquage initial (valeur entière indiquant le nombre de marques). Les transitions ne possèdent qu'un seul attribut : leur nom. Les arcs sont valués par des entiers.

Places et transitions sont connectées au moyens d'arcs qui peuvent être entrants ou sortants. Il n'y a pas de limite au nombre d'arcs partant ou arrivant sur un nœud du modèle.

L'unique fichier annexe de la description MSF est donné ci après.

```
annex_file
// ==========================================================
// Les places

node (PLACE) is
   attribute_list
      attribute string  : NAME;
      attribute integer : MARKING;
   end ;
   connectability_list
      with ARC
         direction out ,
         maximum none ;
      with ARC
         direction in ,
         maximum none ;
   end ;
end PLACE ;

// ==========================================================
// Les transitions

node (TRANSITION) is
   attribute_list
      attribute string : NAME;
   end ;
   connectability_list
      with ARC
         direction out ,
         maximum none ;
      with ARC
         direction in ,
         maximum none ;
   end ;
end TRANSITION ;

// ==========================================================
// Les arcs

link (ARC) is
   attribute_list
      attribute integer : VALUE ;
   end ;
end ARC ;
```

6.2. La description du patron sémantique

Considérons que l'arbre sémantique[38] produit par le patron respecte la structure donnée en Figure 9. Il possède une racine unique de type `ANALYSIS_RESULT` et dont la chaîne de caractères associée contient le nom du modèle. Cette racine possède deux fils (toujours présents) respectivement de type `LIST_OF_PLACE` et `LIST_OF_TRANSITION`. La chaîne de caractères associée à ces nœuds contient respectivement le nombre d'occurrence de place ou de transition.

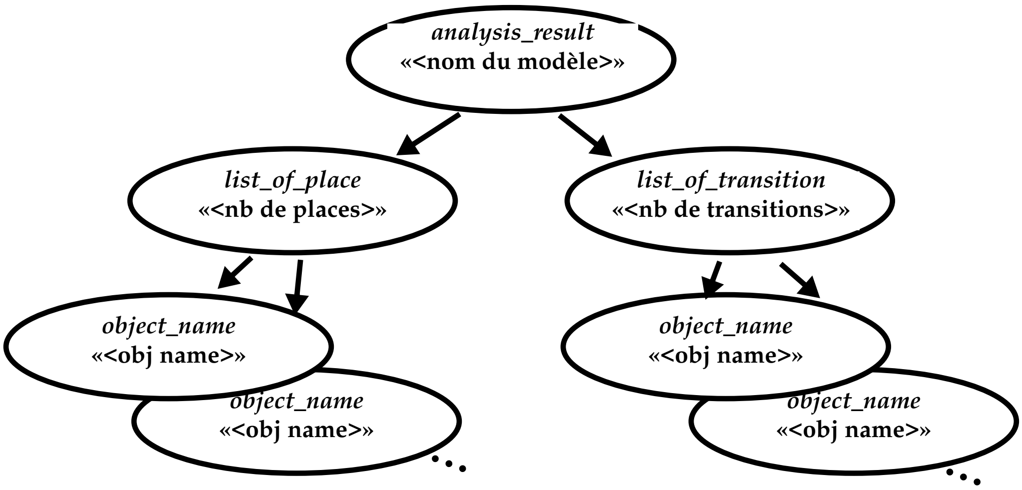

Figure 9 : Structure attendue de l'unique arbre produit par le patron sémantique.

La description des noms de places ou de transitions se fait sur le même modèle. A chaque entité est associé un nœud de type `OBJECT_NAME` et dont le texte associé contient le nom de l'objet considéré. Ces nœuds ont pour père les nœuds `LIST_OF_PLACE` ou `LIST_OF_TRANSITION` selon la classe des objets ainsi désignés.

Le fichier principal du patron sémantique que nous définissons doit déclarer les constructeurs prédéfinis pré-cités et déclare quatre règles dont une principale. Un arbre statique décrit les constructions associées à la référence d'un objet (place ou transition) du réseau de Petri.

```
main_file (P_T_NETS for 'RDP_P_T')

// ========================================================
// patron semantique permettant de comptabiliser les objets
// (places et transitions) d'un réseau de Petri).
// ========================================================

semantic_rule_list is
    // La règle sémantique principale systématiquement invoquee par le moteur de transformation
    main DO_THE_WORK (none) return void,
    // La regle traitant le cas de toutes les places du modele
    WORK_ON_PLACES (none) return semantic_tree,
    // La regle traitant le cas de toutes les transitions du modele
    WORK_ON_TRANSITIONS (none) return semantic_tree,
    // La regle construisant le nom du fichier de statistique produit par le moteur de transformation
    FILE_NAME (MODEL_NAME : string) return string
```

(38) Le travail d'un patron sémantique ne se résume pas forcément à un unique arbre. On peut raisonnablement qu'il y aura un arbre par fichier produits dans le formalisme cible.

```
end;

semantic_tree_list is
   // l'arbre semantique statique decrivant la reference a
   // une place ou une transition du modele
      ONE_OBJECT (NAME : string)
end;

// La liste des identificateurs de construction semantiques reconnues dans les arbres semantiques
// produits par le patron
constructor_list is LIST_OF_PLACES,
                    LIST_OF_TRANSITIONS,
                    ANALYSIS_RESULT,
                    OBJECT_NAME;

// La liste (vide) des variables globales du patron semantique
global_variables none end;

// Les fichiers annexes (un pour le corps des regles, un pour le corps de l'arbre sémantique statique)

include_file ('rdp_sem.trees.mssm');
include_file ('rdp_sem.rules.mssm');
```

Pour définir le patron sémantique, nous avons également écrit deux fichiers annexes : le premier pour l'arbre sémantique statique, l'autre pour les règles.

```
annex_file
// ===========================================================
// L'arbre semantique ONE_OBJECT
// ===========================================================

semantic_tree ONE_OBJECT (NAME : string) is
    semantic_leaf ([OBJECT_NAME # $str (NAME) #])
end;
```

L'expression des règles sémantiques est plus riche puisque l'on dispose de tests, de mécanismes d'itérations simples, et de la récursivité.

```
annex_file

// ===========================================================
// regle semantique principale DO_THE_WORK
// ===========================================================
semantic_rule DO_THE_WORK (none) return void is

THE_TREE : semantic_tree;

begin
   // Creation de la racine de l'arbre resultat
   THE_TREE := create_sm_tree ([ANALYSIS_RESULT # $atrv_str (attribute NET_NAME)#]);
  // Construction du sous-arbre regroupant les places
   message ('Working on places...');
   THE_TREE := add_sm_son ($smt (THE_TREE), sm_rule WORK_ON_PLACES ());
   // Construction du sous-arbre regroupant les transitions
   message ('Working on transitions...');
   THE_TREE := add_sm_son ($smt (THE_TREE), sm_rule WORK_ON_TRANSITIONS ());
   // Application du patron syntaxique associe a l'arbre ainsi produit.
   generate $smt (THE_TREE) in {'semantic_' & $atrv_str (attribute NET_NAME)};
   message ('Done...');
   return;
end;

// ===========================================================
```

```
// regle semantique WORK_ON_PLACES
// ===========================================================
semantic_rule WORK_ON_PLACES (none) return semantic_tree is

RETV : semantic_tree;

begin
  // construction de la racine du sous-arbre indiquant le nombre de places
  RETV := create_sm_tree ([LIST_OF_PLACES #
                           to_string (nb_node_instance (PLACE))#]);
  if nb_node_instance (PLACE) > 0 then
     // Il y a au moins une place, on peut y aller
     for PLACE in 1 .. nb_node_instance (PLACE) do
        // on applique l'arbre statique a toutes les occurences de places
        RETV := add_sm_son ($smt (RETV),
                            sm_tree ONE_OBJECT (NAME => $atrv_str (attribute NAME
                                                from get_node_reference (PLACE,
                                                   $int (PLACE)))));
     end for;
  end if;
  return $smt (RETV);
end;

// ===========================================================
// regle semantique WORK_ON_TRANSITIONS
// ===========================================================
semantic_rule WORK_ON_TRANSITIONS (none) return semantic_tree is

RETV : semantic_tree;

begin
  // construction de la racine du sous-arbre indiquant le nombre de transitions
  RETV := create_sm_tree ([LIST_OF_TRANSITIONS #
                           to_string (nb_node_instance (TRANSITION))#]);
  if nb_node_instance (TRANSITION) > 0 then
    // Il y a au moins une transition, on peut y aller
     for TRANS in 1 .. nb_node_instance (TRANSITION) do
        // on applique l'arbre statique a toutes les occurences de transitions
        RETV := add_sm_son ($smt (RETV),
                            sm_tree ONE_OBJECT (NAME => $atrv_str (attribute NAME
                                                from get_node_reference (
                                                   TRANSITION , $int (TRANS)))));
     end for;
  end if;
  return $smt (RETV);
end;

// ===========================================================
// regle semantique FILE_NAME
// ===========================================================
semantic_rule FILE_NAME (MODEL_NAME : string) return string is

RETV : string;

begin
  RETV := {'statistics_on_' & $str (MODEL_NAME)};
  return $str (RETV);
end;
```

6.3. La description du patron syntaxique

Le patron syntaxique définit l'habillage que l'on veut appliquer à l'arbre produit par le patron sémantique. Nous souhaitons avoir l'habillage suivant :

- identification du modèle analysé
- affichage du nombre de places, puis de la liste des places,
- affichage du nombre de transitions puis, de la liste des transitions.

Le fichier principal du patron syntaxique permet de définir la relation avec le patron sémantique et d'identifier la liste des fichiers annexes à analyser.

```
main_file

// ==========================================================
// ce patron permet de formuler les statistiques calculées au niveau du patron sémantique P_T_NETS
// ==========================================================

syntactic_pattern_name STATISTICS related_to P_T_NETS

include_file ('rdp_stats.annex.msst');
```

Le patron syntaxique étant associé au patron sémantique précédemment défini, il est impératif de fournir au moins une règle externe par constructeur prédéfini identifié dans la section précédente. Pour parcourir l'arbre des références aux objets (places comme transition), il est pratique de définir des règles internes qui seront invoquées explicitement.

```
annex_file
// ================================================================
// Les regles externes permettant de generer des statistiques
// ================================================================

// Traitement de tous les objets du reseau de Petri
syntactic_rule ANALYSIS_RESULT is
begin
   // generation d'un en-tete clair et lisible
   put_line ('-----------------------------------');
   put ('Voici le contenu du modele ');
   put_line ($STR (0));
   put_line ('-----------------------------------');
   // Application (implicite) de la règle associée l'identificateur de construction sémantique de la racine
   // du premier fils
   apply ($1);
   // Meme chose avec la racine du second fils
   apply ($2);
end;

// Traitement d'un arbre de type "liste de transition"
syntactic_rule LIST_OF_TRANSITIONS is
begin
   put ('Il y a ');
   put ($STR (0));
   put_line (' transitions, les voici :');
   if $# > 0 then
      // il y a au moins un fils...appel explicite à la règle interne effectuant le travail
      apply INTERNAL_LIST_OF_TRANSITIONS ($0);
   end if;
end;
```

```
// Traitement d'un arbre de type "liste de place"
syntactic_rule LIST_OF_PLACES is
begin
   put ('Il y a ');
   put ($STR (0));
   put_line (' places, les voici :');
   if $# > 0 then
      // il y a au moins un fils... appel explicite à la règle interne effectuant le travail
      apply INTERNAL_LIST_OF_PLACES ($0);
   end if;
end;

// Traitement des objets (uniquement le nom!)
syntactic_rule OBJECT_NAME is
begin
   put_line ($STR (0));
end;

// ==========================================================
// Les regles internes permettant de generer des statistiques
// ==========================================================

syntactic_rule INTERNAL_LIST_OF_TRANSITIONS is
begin
   // appliquer la regle OBJECT_NAME au premier fils
   put ('   Transition ');
   apply ($1);
   if $# > 1 then
      // il reste d'autres fils a traiter => appel recursif avec reduction de l'arbre en largeur...
      apply INTERNAL_LIST_OF_TRANSITIONS ($2*);
   end if;
end;

syntactic_rule INTERNAL_LIST_OF_PLACES is
begin
   // appliquer la regle OBJECT_NAME au premier fils
   put ('   Place ');
   apply ($1);
   if $# > 1 then
      // il reste d'autres fils a traiter => appel recursif avec reduction de l'arbre en largeur...
      apply INTERNAL_LIST_OF_PLACES ($2*);
   end if;
end;
```

6.4. Invocation de *M*eta*S*cribe

Pour construire le patron sémantique, nous invoquons l'outil de la manière suivante :

```
meta_scribe -s [-t| {-T<RULE_ID>}] dsc_form dsc_semp dsc_synp
```

L'option «-s» est liée à FrameKit (*M*eta*S*cribe est développé avec les API de FrameKit). L'option «-t» permet de passer tracer l'exécution des règles pour la mise au point des patrons (on peut limiter le volume de trace en indiquant explicitement les règles que l'on souhaite tracer).

les paramètres `dsc_form`, `dsc_semp` et `dsc_synp` doivent identifier les fichiers principaux MSF, MSSM et MSST décrivant respectivement le formalisme, le patron sémantique et le patron syntaxique.

Dans le cas de notre exemple, nous appliquons la commande suivante qui donne le résultat indiqué ci après.

```
$ meta_scribe -s rdp_form.main.smf rdp_sem.main.mssm rdp_stats.msst
MetaScribe (1.1b2) by F.Kordon, July 1997 (FrameKit API: DF=2.0, EC=2.0, UI=2.0)
====================================================
= Laboratoire d'Informatique de Paris 6,           =
= Universite P. & M. Curie, 4 place Jussieu,       =
= 75252 Paris Cedex 05, France                     =
====================================================
generating transformation engine for :
Input formalism is   :rdp_form.main.msf
Semantic pattern is  :rdp_sem.main.mssm
Syntactic pattern is :rdp_stats.main.msst
====================================================
Analysing input formalism...
Analysing semantic pattern...
        Working on static tree ONE_OBJECT
        Working on rule DO_THE_WORK
=======================================================
Warning : cannot verify dynamic reference to an objet attribute
=======================================================
        Working on rule WORK_ON_PLACES
=======================================================
Warning : cannot verify dynamic reference to an objet attribute
=======================================================
        Working on rule WORK_ON_TRANSITIONS
        Working on rule FILE_NAME
Analysing syntactic pattern...
        Working on syntactic rule ANALYSIS_RESULT
        Working on syntactic rule LIST_OF_TRANSITIONS
        Working on syntactic rule LIST_OF_PLACES
        Working on syntactic rule OBJECT_NAME
        Working on syntactic rule INTERNAL_LIST_OF_TRANSITIONS
        Working on syntactic rule INTERNAL_LIST_OF_PLACES
Finalizing semantic pattern...
Generation of the transformation engine completed.
```

Cette commande provoque la génération de fichiers Ada implémentant les fonctions définies pour l'exemple. Le nom du programme généré est obtenu par concaténation du nom des deux patrons considérés (dans le cas présent, cela donne `p_t_nets_statistics`).

Les programmes ainsi obtenus doivent être compilé dans l'environnement de développement de FrameKit[39]. Pour effectuer une transformation à partir d'un modèle décrit en MSM, il suffit ensuite d'invoquer la commande suivante :

```
p_t_nets_statistics -s dsc_form dsc_mod
```

Les paramètres `dsc_form` et `dsc_mod` désignent respectivement les fichiers principaux MSF et MSM décrivant respectivement le formalisme source et le modèle à traiter. La référence au formalisme source permet de vérifier que la description MSM ne contient pas d'éléments étrangers au formalisme traité.

[39] A terme, les fichier correspondant aux éléments minimaux de FrameKit nécessaire à la mise en œuvre du patron sémantique seront fournis en standard avec le script de compilation ad-hoc.

6.5. Exemple d'utilisation du moteur de transformation produit

Considérons le fichier MSM correspondant à un modèle de dix-huit places nommées P#1 à P#18, treize transitions nommées T#1 à T#3 et aucun arc. La description MSM d'un tel modèle est indiquée ci-dessous. Le nom des attributs globaux, des objets possibles et des attributs associés à ces objets est directement déduit de la description en langage MSF que nous avons déjà donnée.

```
MAIN_FILE
FORMALISM ( 'RDP_P_T' ) ;
// Attributs globaux
WHERE (ATTRIBUTE net_name => 'petit modele de test') ;

// Places

NODE 'place_1' IS place
   WHERE (ATTRIBUTE name => 'P#1') ;
NODE 'place_2' IS place
   WHERE (ATTRIBUTE name => 'P#2') ;
NODE 'place_3' IS place
   WHERE (ATTRIBUTE name => 'P#3') ;
NODE 'place_4' IS place
   WHERE (ATTRIBUTE name => 'P#4') ;
NODE 'place_5' IS place
   WHERE (ATTRIBUTE name => 'P#5') ;
NODE 'place_6' IS place
   WHERE (ATTRIBUTE name => 'P#6') ;
NODE 'place_7' IS place
   WHERE (ATTRIBUTE name => 'P#7') ;
NODE 'place_8' IS place
   WHERE (ATTRIBUTE name => 'P#8') ;
NODE 'place_9' IS place
   WHERE (ATTRIBUTE name => 'P#9') ;
NODE 'place_10' IS place
   WHERE (ATTRIBUTE name => 'P#10') ;
NODE 'place_11' IS place
   WHERE (ATTRIBUTE name => 'P#11') ;
NODE 'place_12' IS place
   WHERE (ATTRIBUTE name => 'P#12') ;
NODE 'place_13' IS place
   WHERE (ATTRIBUTE name => 'P#13') ;
NODE 'place_14' IS place
   WHERE (ATTRIBUTE name => 'P#14') ;
NODE 'place_15' IS place
   WHERE (ATTRIBUTE name => 'P#15') ;
NODE 'place_16' IS place
   WHERE (ATTRIBUTE name => 'P#16') ;
NODE 'place_17' IS place
   WHERE (ATTRIBUTE name => 'P#17') ;
NODE 'place_18' IS place
   WHERE (ATTRIBUTE name => 'P#18') ;

// Transitions

NODE 'trans_1' IS transition
   WHERE (ATTRIBUTE name => 'T#1') ;
NODE 'trans_2' IS transition
   WHERE (ATTRIBUTE name => 'T#2') ;
NODE 'trans_3' IS transition
   WHERE (ATTRIBUTE name => 'T#3') ;
```

```
NODE 'trans_4' IS transition
   WHERE (ATTRIBUTE name => 'T#4') ;
NODE 'trans_5' IS transition
   WHERE (ATTRIBUTE name => 'T#5') ;
NODE 'trans_6' IS transition
   WHERE (ATTRIBUTE name => 'T#6') ;
NODE 'trans_7' IS transition
   WHERE (ATTRIBUTE name => 'T#7') ;
NODE 'trans_8' IS transition
   WHERE (ATTRIBUTE name => 'T#8') ;
NODE 'trans_9' IS transition
   WHERE (ATTRIBUTE name => 'T#9') ;
NODE 'trans_10' IS transition
   WHERE (ATTRIBUTE name => 'T#10') ;
NODE 'trans_11' IS transition
   WHERE (ATTRIBUTE name => 'T#11') ;
NODE 'trans_12' IS transition
   WHERE (ATTRIBUTE name => 'T#12') ;
NODE 'trans_13' IS transition
   WHERE (ATTRIBUTE name => 'T#13') ;
```

L'application du moteur de transformation au fichier MSM indiqué ci dessus donnera :

```
hephaistos>p_t_nets_statistics -s rdp_form.main.msf un_modele_rdp.msm
Analysing input formalism...
Analysing input model...
Working on places...
Working on transitions...
Done...
```

Dans le patron sémantique (règle `DO_THE_WORK`), nous avons spécifié que le résultat devait être généré dans un fichier nommé `statistic_result`. Ce fichier, après exécution du moteur de transformation, contient :

```
hephaistos>cat stat_result
----------------------------------
Voici le contenu du modele petit modele de test
----------------------------------
Il y a  18 places, les voici :
   Place P#1
   Place P#10
   Place P#11
   Place P#12
   Place P#13
   Place P#14
   Place P#15
   Place P#16
   Place P#17
   Place P#18
   Place P#2
   Place P#3
   Place P#4
   Place P#5
   Place P#6
   Place P#7
   Place P#8
   Place P#9
Il y a  13 transitions, les voici :
   Transition T#1
   Transition T#10
   Transition T#11
```

```
Transition T#12
Transition T#13
Transition T#2
Transition T#3
Transition T#4
Transition T#5
Transition T#6
Transition T#7
Transition T#8
Transition T#9
```

7. Références Bibliographiques

[1] G. Araujo, S. Devadas, K. Keutzer, S. Liao, S. Malik, A. Sudarsanam, S. Tjiang & A. Wang, "Challenges in Code Generation for Embedded Processors", Chapter 3, pp. 48-64, in "Code Generation for Embedded Processors", P. Marwedel and G. Goossens editors, Kluwer Academic Publishers, ISBN 0-7923-9577-8, 1995

[2] M. J. Chen, "Developing a Multidimensional Synchronous Dataflow Domain in Ptolemy", ERL Technical Report UCB/ERL No. 94/16, University of California, Berkeley, CA 94720, May, 1994

[3] P. Desfray, "Modélisation par Objets", Masson 1996

[4] C. Donnelly & R. Stallman, "Bison: The YACC-compatible Parser Generator", GNU documentation, <http://www.cl.cam.ac.uk/texinfodoc/bison_toc.html>, November 1995

[5] C. Hylands, E. Lee & H. Reekie, "The Tycho User Interface System", The 5th Annual Tcl/Tk Workshop '97, Boston, Massachusetts, pp 149-157, July 14-17, 1997

[6] F. Kordon & J-L. Mounier, "FrameKit and the prototyping of CASE environments", 8th IEEE International Workshop on Rapid System Prototyping, Research Triangle Park Institute, IEEE comp Soc Press N° 97TB100155, pp 91-97, June 1997

[7] F.Kordon & J-L. Mounier, "FrameKit, an Ada Framework for a Fast Implementation of CASE Environments", to appear in proceedings of the ACM/SIGAda ASSET'98 symposium, July 1998

[8] F.Kordon, "tha FrameKit Home Page", `<http://www-src.lip6.fr/framekit>`

[9] J-L. Mounier, "the Macao Home page", `<http://www-src.lip6.fr/macao>`

[10] MultiQuest Corporation, "S-CASE 3.0 User's guide", 1931 N Meachanm Road, Suite 318, Schaumburg, IL 60173, USA, 1996

[11] V. Paxson, "Flex: A fast scanner generator, Edition 2.5", GNU documentation, <http://www.cl.cam.ac.uk/texinfodoc/flex_toc.html>, March 1995

[12] Ptolemy project, "Ptolemy Programmer's Manual", University of California berkeley, 1996

[13] SofTeam, "Manuel utilisateur d'*Objecter*ing v4", 1996