5

Le Méta-Générateur MetaGen

Après avoir défini les règles de transcription, il s'agit de les implanter dans un générateur de code. Le rôle de ce générateur de code est de transcrire la description textuelle du modèle de performance d'un système en un programme VHDL conformément aux règles de transcription décrites dans le chapitre 4. La description textuelle d'entrée et celle de sortie du générateur à développer sont conformes à une grammaire. Elles respectent une syntaxe et une sémantique données.

Développer un outil pour effectuer cette transformation n'est pas une tâche facile. L'effort est encore plus important quand la génération de code inclut également une phase de synthèse logicielle ou matérielle. On se limitera dans ce document uniquement à la génération de code, ce qui signifie que l'outil de génération à développer n'est pas capable de choisir une solution d'implantation de lui-même. Si plusieurs alternatives possibles existent, la solution retenue sera spécifiée dans le modèle source ou via l'interface utilisateur de l'outil. Pour cet objectif, l'effort de développement est fortement dépendant de la syntaxe et la sémantique des langages source et cible et de la complexité des règles de transformation.

Dans le passé, l'équipe MCSE a déjà développé trois types de générateurs de code pour le modèle fonctionnel de MCSE: un générateur de C [CALVEZ-93c], un générateur d'OCCAM [PASQUIER-93], un générateur de VHDL simulable [BAKOWSKI-92] et synthétisable [HELLER-92] [CALVEZ-93d]. Ces générateurs de code écrits en C utilisaient des structures internes spécifiques. Cette solution d'implantation qui dépendait également de l'environnement graphique SunView ou XMOTIF pour l'interface utilisateur s'est vite révélée pénalisante. Elle n'était pas portable. Elle ne facilitait pas la maintenance et la mise à jour des générateurs en fonction des modifications de la grammaire de la spécification d'entrée ou de celles des règles de transcription.

Afin d'obtenir des outils multi plate-formes et plus faciles à développer, maintenir et enrichir, l'équipe MCSE s'est alors intéressée à la technologie méta-case et en particulier à l'outil GraphTalk pour l'édition de graphes et l'outil LEdit pour la génération de code. Cette

approche a également été abandonnée. L'explication de cette abandon est donnée en début de chapitre. Mais cette expérience et en particulier le travail effectué sur la génération de code avec LEdit a amené l'équipe MCSE à revoir entièrement sa stratégie de développement des outils dont les générateurs de code comme support pour la méthodologie MCSE. La présentation de la nouvelle philosophie de développement de la plate-forme MCSE montre qu'elle repose sur les concepts d'analyseurs syntaxiques et de méta-structure.

Ces deux concepts nous ont permis de développer un principe générique de génération de code présenté dans ce chapitre et exploité pour implanter de nouveaux générateurs de code ayant comme spécification d'entrée le modèle de performance de MCSE décrit dans le chapitre 3. Un analyseur syntaxique est obtenu à partir de la spécification de la grammaire du langage concerné. Si la spécification de la grammaire est enrichie de règles de production, alors elle peut engendrer la structure de données interne image du texte analysé. La structure interne obtenue à partir d'un fichier texte est le coeur des générateurs. Pour que ceux-ci aient une architecture commune, nous avons défini un modèle générique de modèle de structure de données. Après avoir analysé la spécification d'une grammaire, nous présentons dans ce chapitre ce modèle générique appelé par la suite méta-structure ainsi que l'architecture commune des générateurs.

Le principe de génération de code qui est ensuite décrit repose également sur le concept de template qui est un fichier contenant toutes les constructions en langage cible nécessaires à la transcription. Pour un générateur donné, le fichier source et un fichier template sont analysés, puis les opérations de manipulation des structures de données résultantes sont exécutées pour réaliser la transcription. Pour avoir un générateur facilement reconfigurable, nous avons décidé de décrire les opérations de manipulation de structures de données dans un langage interprété nommé Script. La syntaxe et les constructions de ce langage dédié à la manipulation de structure de données sont définies. Cette définition sert par la suite à l'interprétation de l'implantation du générateur de code VHDL qui fait l'objet du chapitre 6.

Une fois saisi, le Script est interprété ou transcrit en code JAVA par un programme dont l'implantation en Java est décrite à l'aide du modèle statique de la méthode OMT-UML. Ce programme nommé MetaGen est donc un générateur de générateurs de code ou méta-générateur.

5.1 LA TECHNOLOGIE META-CASE

De 1993 à 1996, le développement de la plate-forme MCSE a reposé sur l'utilisation d'un méta-outil orienté éditeur de graphes (GRAPHTALK) et d'un méta-outil orienté éditeur syntaxique (LEdit). La mise en oeuvre d'un outil CASHE (Computer Aided Software Hardware Engineering) pour une méthodologie à l'aide de la technologie méta-Case est normalement réduite à la déclaration du formalisme de ses modèles. Les méta-outils permettent, en effet, de se focaliser sur la méthode et la sémantique des modèles plutôt que sur des aspects bas niveau tels que l'intégration d'outils, la gestion d'une base de données et l'environnement graphique.

L'outil GraphTalk était séduisant pour au moins 4 raisons. Il est multi plate-formes (PC, UNIX). Il est construit autour d'une base de données orientée objet. Les spécifications des modèles sont saisies graphiquement et l'outil permet d'adopter une démarche incrémentale facilitant le prototypage. La spécification graphique du méta-modèle fournit un bon support pour la documentation et le suivi du développement de l'outil.

Malheureusement, l'utilisation de GraphTalk a révélé progressivement un certain nombre de limitations:

- les concepts de méta-modélisation sont trop limités pour décrire complètement le modèle MCSE et il fallait écrire de plus en plus de code C (API C) pour définir des démons et actions sur les objets du méta-modèle. Au fur à mesure de l'implantation, on perdait en efficacité (interprétation des démons) et en facilités de prototypage.
- la personnalisation de l'interface utilisateur est très limitée.
- il n'y a quasiment plus de maintenance et d'évolubilité des outils GraphTalk et LEdit alors que ceux-ci sont pourtant fortement "buggés".
- l'utilisation nécessite de payer un "runtime" par machine.

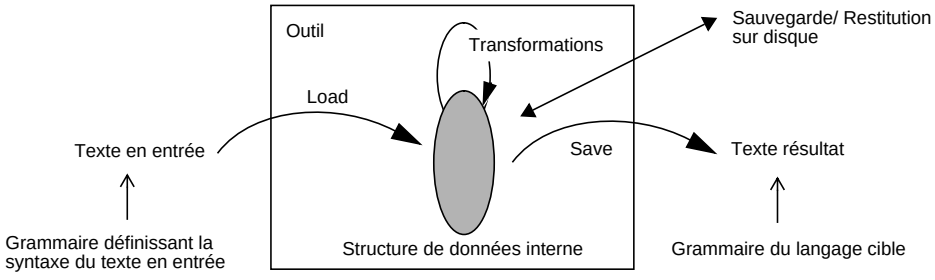
L'utilisation des méta-outils GraphTalk/LEdit a conduit à une impasse. Pour compenser leurs limites, l'équipe devait dépenser de plus en plus d'efforts alors que les résultats ne pouvaient être que médiocres, non propriétaires et sans perspective d'évolution.

Cependant, l'expérience de génération de code avec LEdit qui est une encapsulation de Lex&Yacc a permis d'appréhender les concepts des générateurs d'analyseurs syntaxiques et de méta-structure sur lesquels repose entièrement la nouvelle "philosophie" de développement des outils MCSE. Avec l'expérience de la technologie méta-Case, l'équipe a également pris conscience que le succès du développement d'un outil CASHE repose essentiellement sur la généricité et l'évolubilité de la structure des données internes aux outils. Début 1997, l'équipe a donc réfléchi à une autre orientation.

5.2 NOUVELLE STRATEGIE DE DEVELOPPEMENT DES OUTILS

Il est usuel de constater que dans la plupart des outils CASHE actuels le couplage de données entre outils est basé sur un échange par fichiers et non sur l'utilisation d'une base de données pour des raisons d'efficacité et d'indépendance. Sur la base de ce constat, l'équipe a donc décidé de développer une nouvelle plate-forme d'outils basée sur un couplage possible entre outils par fichiers. Dans ce contexte, comme le montre la figure 5.1, chaque outil de la plate-forme peut être implanté sous la forme d'une architecture générique basée sur trois fonctions principales:

- la fonction *Load* qui permet de créer la structure de données à partir du fichier texte,
- la fonction *Save* qui est la réciproque de la fonction Load et qui consiste à parcourir la structure de données et à exploiter la grammaire correspondante pour générer un fichier texte,
- la fonction *Transformations* qui est spécifique à l'outil et qui définit les manipulations effectuées sur la structure de donnée chargée.



-Figure 5.1- Architecture générique des outils de la plate-forme.

Cette approche nécessite de résoudre deux problèmes:

- La lecture d'un texte et sa conversion en une structure de données (Load) et l'opération inverse (Save). Pour résoudre ce problème, nous sommes partis du principe que toute structure de données peut être engendrée d'une manière automatique par un analyseur syntaxique enrichi des règles de production de la structure de données. Cet analyseur syntaxique est obtenu par un générateur d'analyseur syntaxique à partir de la spécification de la grammaire (syntaxe d'entrée du texte source) et des règles de production. Pour avoir un principe de création de la structure de données indépendant de la grammaire, nous avons défini pour la spécification des règles de production un modèle de modèle de structure de données ou méta-structure basé sur la composition de trois constructions de noeuds: le record ou ensemble fini d'éléments, l'alternative et la liste. Ce principe est présenté en détail dans le paragraphe 5.4. L'utilisation de cette technique permet de répondre aux critères de généricité et d'évolubilité des structures de données internes aux outils considérés indispensables pour le succès du développement de la plate-forme d'outils. Pour faire évoluer la structure de données, il suffit en effet de modifier la grammaire et de regénérer l'analyseur syntaxique associé. La fonction Save qui correspond à un parcours ordonné de la structure de données interne et à l'exploitation de la grammaire pour le formatage du texte de sortie ne pose pas de difficultés particulières d'implantation.

- la sauvegarde et restitution d'une structure de données sur disque sans contrainte de format, ce qui se résoud bien avec le concept d'objets persistants. Ce problème est résolu implicitement par le choix d'implantation des outils en Java qui dispose en standard (JDK 1.1) de la sérialisation/désérialisation d'objets.

5.3 ANALYSE D'UNE GRAMMAIRE

Le fichier d'entrée d'un outil respecte une syntaxe donnée. La syntaxe acceptée se décrit par une grammaire. Une grammaire se décrit elle-même selon une grammaire qui est alors appelée méta-grammaire. Un analyseur syntaxique est une fonction chargée de vérifier qu'un texte donné en entrée respecte une grammaire donnée. Si l'analyseur syntaxique est enrichi par des règles de production ou actions, il est alors à même de produire directement quelque chose en sortie. Nous sommes ici intéressés par la production de la structure de données interne équivalente au texte analysé et la fonction inverse pour la production de texte de manière à disposer d'une fonction bi-univoque texte <-> structure de données.

5.3.1 Définition d'une grammaire

Une grammaire est un ensemble fini de règles de la forme "coté gauche ::= coté droit" où le coté gauche représente un symbole non terminal et le coté droit est un ensemble éventuellement vide de symboles non terminaux et terminaux. Les symboles terminaux représentent les mots clés et les expressions régulières (identificateurs, nombre, etc).

Pour la compréhension de la technique employée par un générateur d'analyseur syntaxique, nous commençons par présenter quelques règles de grammaire pour un exemple très simple. Le texte suivant constitue un ensemble de règles de grammaire écrites selon la norme BNF (Backus Naur Form).

ComponentDescription ::= <**Component**> [Range] Name [GenericParameters] Parameters ";" [Attributes] ComponentDef <**EndComponent**>

Range ::= "**[**" Indice ":" Indice "**]**"

GenericParameters ::= **Generic** Name {"**,**" Name }

Parameters ::= "**(**" {Parameter";"} "**)**"

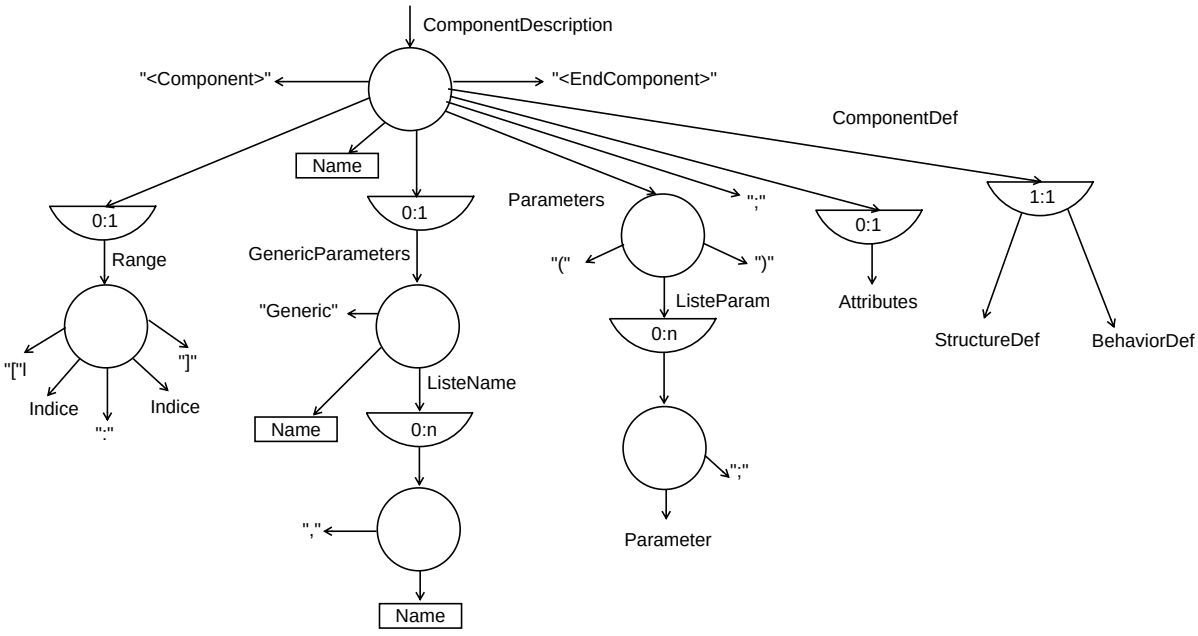
ComponentDef ::= StructureDef | BehaviorDef

La signification de la notation pour interpréter les règles est la suivante:

- **<Component>** et "**(**" sont des symboles terminaux (directement présents dans le texte). Les autres noms (GenericParameters, Parameters, etc.) sont appelés symboles non-terminaux et doivent être définis par des règles de grammaire.
- ::= définit une règle de grammaire. La partie gauche est régie selon la description à droite.
- Y ::= A B C veut dire que le texte doit comporter A puis B puis C.
- Y ::= A | B | C veut dire que le texte comprend A ou B ou C (Ou exclusif)
- Y ::= [A] veut dire que A est optionnel.
- Y ::= { A } veut dire qu'il s'agit d'une suite de longueur quelconque de A, la suite pouvant être vide.

5.3.2 Structure de données pour une grammaire

La figure suivante montre les règles de grammaire ci-dessus, mais cette fois représentées sous une forme graphique. La notation graphique est celle recommandée par la méthodologie MCSE pour la structuration des données [CALVEZ-90].



-Figure 5.2- Structure de données pour une grammaire.

La notation graphique utilise ici 3 types de symboles:

- Le *rectangle* qui contient un identificateur ou une valeur (symbole terminal),
- Un *rond* qui signifie le produit cartésien et qui veut aussi dire la composition de tous les champs qui en partent. C'est l'équivalent d'un record. Un champ peut être un mot clé, une valeur ou identificateur, la désignation d'un autre noeud de la structure.
- Un *demi-rond* pour l'ensemble qui signifie le regroupement d'une collection d'éléments. Un ensemble peut être vide ou non, peut aussi être borné. Dans ce dernier cas, les 2 nombres dans le symbole indiquent la borne minimale et la borne maximale.

Sur la base de cette notation, la structure de données devient plus lisible. La première règle de grammaire textuelle de l'exemple se traduit par le rond ComponentDescription. Comme Range est optionnel, on utilise le symbole ensemble (0:1) pour représenter l'alternative: Range ou absence de Range. On retrouve la même notation pour les paramètres génériques. La notation 0:n pour ListeName et ListeParam indique une suite vide ou de longueur quelconque. La notation 1:1 représente l'alternative des cas possibles. Une seule doit exister. C'est le cas pour ComponentDef.

A noter que dans le cas d'une récursivité, la structure de données est arrêtée dès la rencontre d'un symbole non-terminal déjà défini au préalable.

On peut conclure que tout texte respectant une grammaire peut se décrire sous la forme d'une structure de données construite sur la base des 2 opérateurs: composition et ensemble.

5.3.3 Structure de données pour une solution

Considérons maintenant un texte respectant les règles de grammaire ci-dessus. Un exemple est donné ci-après.

<Component> [1:k] F1 Generic n, k (E; S;)

<Structure>

.....

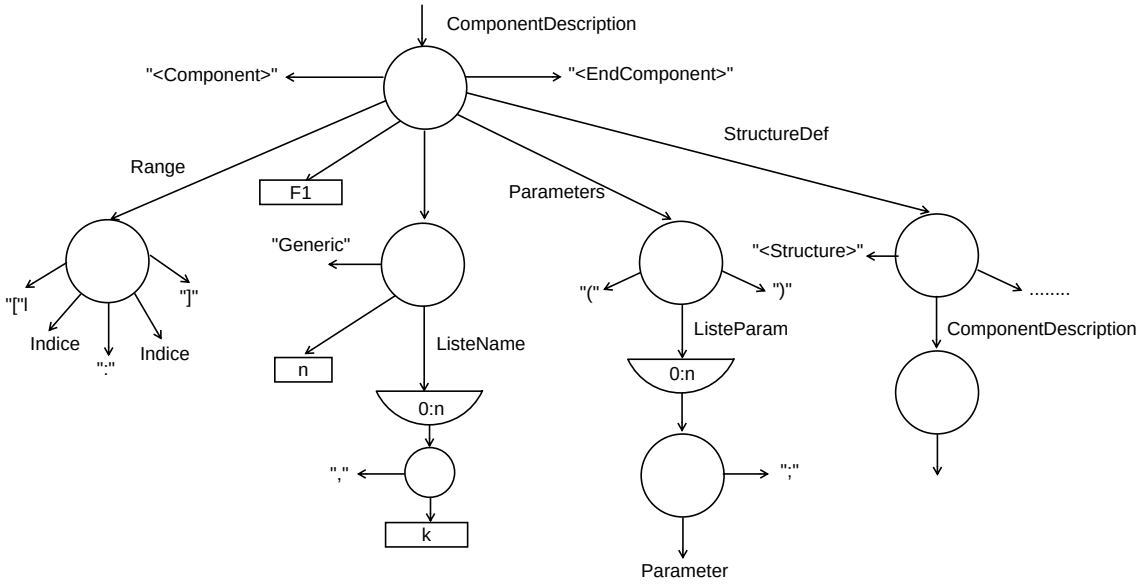
<EndComponent>

Intéressons-nous à sa représentation possible sous la forme d'une structure de données en exploitant la même notation que précédemment.

La structure de données de l'exemple représentée par la figure suivante montre la disparition des ensembles 0:1, 1:1. Ceci s'explique par le fait que l'alternative représentée par l'ensemble 0:1 se trouve maintenant spécifié pour un texte donné. On trouve par exemple ici Range avec sa définition. En l'absence de Range, le lien pointe sur Nil (qui veut dire vide). Pour un ensemble 1:1, on trouve le noeud correspondant au cas représenté dans le texte (ici StructureDef).

Les noeuds de composition se retrouvent intégralement dans la solution. Dans le cas d'une récursivité, la structure de données est prolongée. En effet, si un composant contient un autre composant, on doit retrouver la structure de données de cet autre composant (différence

essentielle par rapport à la structure de données de la grammaire). Le cas est représenté par la figure 5.3 pour le noeud StructureDef qui inclut des composants définis par le noeud ComponentDescription.



-Figure 5.3- Structure de données pour une solution.

5.4 SPECIFICATION D'UNE META-STRUCTURE

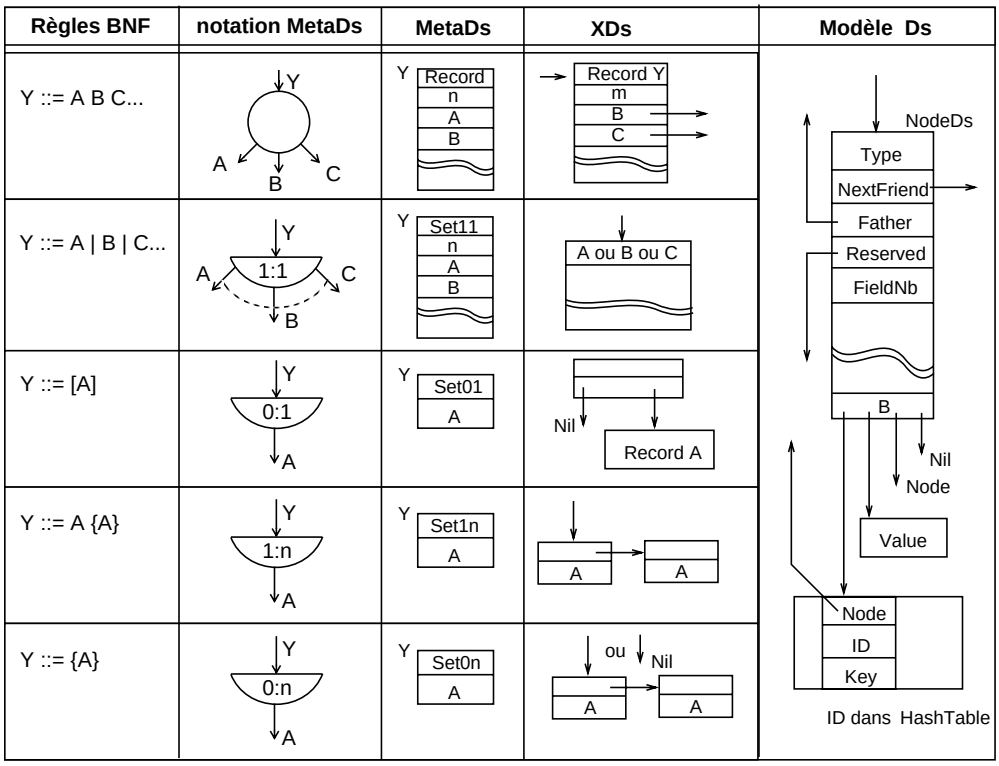
La présentation faite dans le paragraphe précédent permet maintenant de définir une méthode générale de construction d'un chargeur (loader) assurant la transformation d'un texte sous la forme d'une structure de données. Le texte à analyser est défini par un ensemble de règles de grammaire. A ces règles se trouvent associées des règles de production induisant la construction de la structure de données du texte analysé. Dans ce paragraphe nous montrons que les règles de grammaire peuvent se représenter sur la base d'une méta-structure appelée ici **MetaDs**. La MetaDs sert alors à la construction de la représentation interne de la grammaire appelée ici **GrammarDs**. La GrammarDs est utilisée par un analyseur syntaxique aussi appelé parser pour analyser et construire la structure de données de chaque solution appelée ici **XDs**. La XDs est basée sur un modèle de noeud générique appelé ici **DsModel**. Ceci est expliqué dans le paragraphe suivant.

Il s'agit donc de tirer des règles générales de transformations texte <-> structure à partir de l'exemple de présentation du paragraphe précédent.

La figure 5.4 indique sous la forme de tableau le résultat de synthèse. La première colonne indique les 5 cas de règles de grammaire selon la notation BNF. Les autres cas s'en déduisent par composition ou association. Y est un symbole non-terminal. A, B et C sont des symboles terminaux ou non-terminaux.

La deuxième colonne montre la représentation graphique équivalente sous la forme de la notation des structures de données. On retrouve le noeud de composition et 4 types d'ensembles, chacun défini par des bornes différentes.

La troisième colonne donne une implémentation appropriée pour chaque type de noeud. Ainsi une grammaire peut se représenter en interne sur la base de records. Le premier champ indique le type. Le deuxième champ indique le nombre de symboles qui suivent, sauf s'il un seul existe.



-Figure 5.4- Correspondances entre les règles BNF et les modèles de structure de données.

Les trois premières colonnes du tableau définissent ainsi la méta-grammaire et la méta-structure pour spécifier toute grammaire et la construction de structure équivalente. La transformation des règles textuelles d'une grammaire donne une structure de données appelée GrammarDs conforme à la notation MetaDs construite sur la base des règles qui viennent d'être définies.

Intéressons-nous maintenant à la structure de données qui doit être construite par la fonction Loader lors de la reconnaissance d'un texte. La quatrième colonne représente la notation de la structure de données pour une solution tandis que la cinquième colonne montre une solution pour l'implémentation.

Pour un noeud de composition, la notation est identique à celle de la grammaire, mais l'implantation diffère. Pour des raisons d'optimisation de la structure, seuls les champs non-terminaux sont placés dans le record (pour l'exemple, A est considéré ici comme symbole terminal). En effet, comme les champs terminaux sont constants, il ne sert donc à rien de les dupliquer dans ce record car leur existence et valeur sont définies dans la GrammarDs. Ainsi, l'interprétation de la structure de données pour tout texte se fera à l'aide de la GrammarDs. Pour cette raison, on ne trouve dans un noeud de composition que des références vers d'autres noeuds.

Lors d'une alternative Set1:1, on trouve la notation alternative. Pour une solution donnée, l'alternative disparaît car l'élément concerné est fixé dans le texte. Le cas est alors défini par le type de record (ici A ou B ou C...). Ainsi l'ensemble disparaît dans la structure XDs. Si la

structure doit pouvoir se modifier par un éditeur syntaxique par exemple, les alternatives possibles sont connues par l'analyse du noeud de la GrammarDs qui indique les possibilités par le Set 1:1.

Pour l'alternative Set 0:1, on trouve les 2 possibilités: absence ou symbole A. L'implantation est immédiate. Une référence Nil indique la non-existence de l'option. Dans le cas contraire on y trouve la référence du noeud.

Les ensembles Set 1:n et Set 0:n se retrouvent dans la structure. Elles sont implantables comme une liste chaînée des noeuds composant cet ensemble. Si l'ensemble est vide, alors la référence est Nil.

Il ressort de cette analyse que toute structure **XDs** peut se construire sur la base d'un modèle de noeud appelé ici **NodeDs**. Le modèle est représenté dans la dernière colonne. Chaque noeud est identifiable par son type qui est le même que le noeud équivalent dans la GrammarDs. On trouve ensuite les champs:

- NextFriend pour permettre son inclusion dans une liste et donc dans un ensemble,
- Father pour être capable de remonter la structure,
- Reserved pour des utilisations ultérieures,
- FieldNb qui définit le nombre de références qui suivent. Ces références désignent obligatoirement des noeuds non-terminaux. Elles sont définies dans le noeud du même type de la GrammarDs.
- Chaque référence désigne 4 cas possibles: vide, un autre noeud, une valeur, un identificateur.

Une valeur peut être un booléen, un entier ou un flottant. Un identificateur est une chaîne de caractères. Si un accès par le nom est souhaité, une hashtable pour la mémorisation des identificateurs peut être appropriée. Mais il faut faire très attention à la portée des identificateurs. Ce point essentiel conduit à un ensemble de hashtables et pas uniquement une seule.

En conclusion, tout texte source conforme à un langage X est représenté par une structure de données XDs dont la signification est fournie par une structure de donnée XGrammarDs qui est la transcription des règles de grammaire textuelles de X.

5.5 GENERATION DU LOADER

L'objectif est de disposer d'un programme Loader capable de transformer tout texte MCSE conforme aux règles de grammaire du modèle MCSE en une structure de données McseDs et un ensemble de tables des identificateurs. Une qualité essentielle du loader est son aptitude à pouvoir s'adapter à toute modification de la grammaire. Cette technique doit donc aussi pouvoir s'employer pour toutes autres grammaires telles que C et VHDL par exemple.

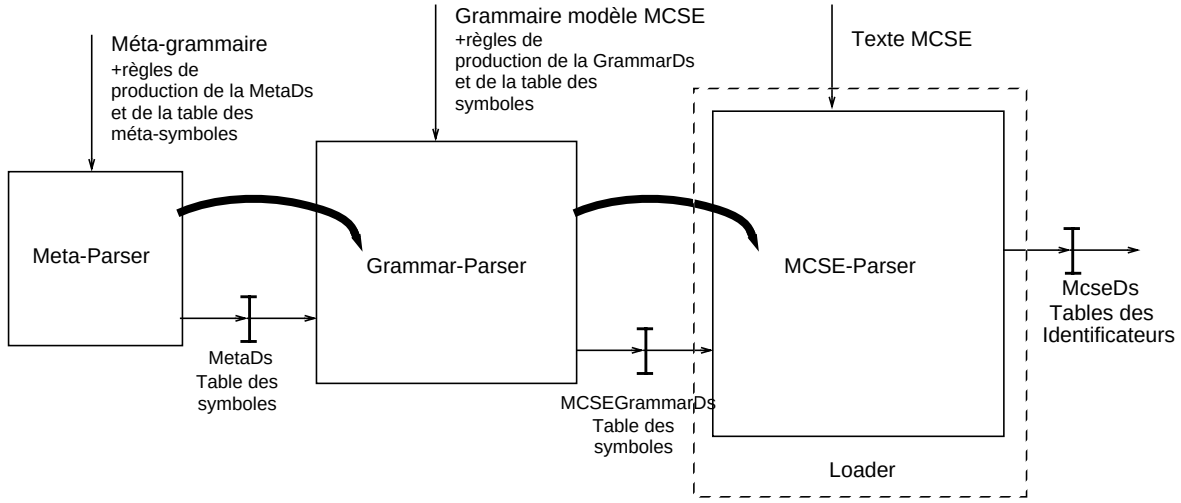
Le loader souhaité est en fait un parser (analyseur syntaxique). Le terme parser inclut ici tout d'abord un analyseur lexical pour identifier les tokens successifs et les présenter à l'analyseur syntaxique. La figure 5.5 explique la technique dite du "bootstrapping" que nous avons utilisée pour engendrer le parser souhaité.

Le loader est en fait réalisé par un programme dénommé MCSE-Parser. Il exploite en entrée tout texte MCSE pour l'analyser avec l'aide de la structure MCSEGrammarDS et de la table des symboles associée de la grammaire du modèle MCSE. La MCSEGrammarDs et la table

des symboles sont des classes d'objets fixes (statiques) pour une grammaire donnée intégrées au programme MCSE-Parser lors de sa compilation pour disposer d'une version exécutable. Cette compilation ne doit se faire que lors d'une modification de grammaire.

Il faut bien comprendre que le programme MCSE-Parser contient, en plus des tables de reconnaissance du texte, les procédures ou méthodes appropriées de production des noeuds de la structure de données McseDs. La question intéressante est: comment engendrer automatiquement ce programme?

La figure suivante fournit la réponse.



-Figure 5.5- Technique du "bootstrapping" pour la génération du loader.

Pour générer la MCSEGrammarDs et la table des symboles correspondante, mais aussi le programme source MCSE-Parser, il faut disposer d'un autre programme. Un parser peut à nouveau effectuer cette production à condition qu'il sache produire du code source. Un tel parser est ici appelé Grammar-Parser puisqu'il exploite pour cela en entrée la grammaire du modèle MCSE. Le lien en gras représente la production d'un code source.

Le programme Grammar-Parser doit disposer en entrée:

- de la grammaire du modèle MCSE qui est le texte parsé,
- intégrées à cette grammaire, les règles de production de la MCSEGrammarDs, de la table des symboles de la grammaire et du code source de MCSE-Parser. Ces règles sont directement écrites dans le langage du programme, c'est-à-dire ici en JAVA.
- de la MetaDs et de la table des symboles de la méta-grammaire, indispensable pour l'interprétation de la grammaire et la production de la GrammarDs.

On peut à nouveau continuer le raisonnement en se demandant comment obtenir automatiquement le programme Grammar-Parser. La réponse est maintenant clair. Il suffit d'utiliser à nouveau un parser appelé maintenant Meta-Parser.

Le méta-Parser exploite en entrée la méta-grammaire à laquelle se trouvent associées les règles de production du code source de Grammar-Parser, de la MetaDs, de la table des symboles de la méta-grammaire.

Pour résumer ou pour avoir une autre vision de cette technique, le méta-parser produit des classes d'objets appelées MetaDs et Grammar-Parser. L'exécution de l'objet Grammar-Parser produit des instances des classes d'objets MetaDs pour représenter la grammaire ce qui donne

les classes MCSEGrammarDs et MCSE-Parser. L'exécution de l'objet MCSE-Parser produit des instances des objets de la classe MCSEGrammarDs pour représenter une solution de description.

L'intérêt essentiel de cette technique est son évolutivité. Un changement souhaité de la structure interne McseDs se traduit par une modification de la grammaire du modèle MCSE ou par une modification de la MCSEGrammarDs ou par une modification des règles de production. Un nouveau loader est engendré par exploitation de Grammar-Parser.

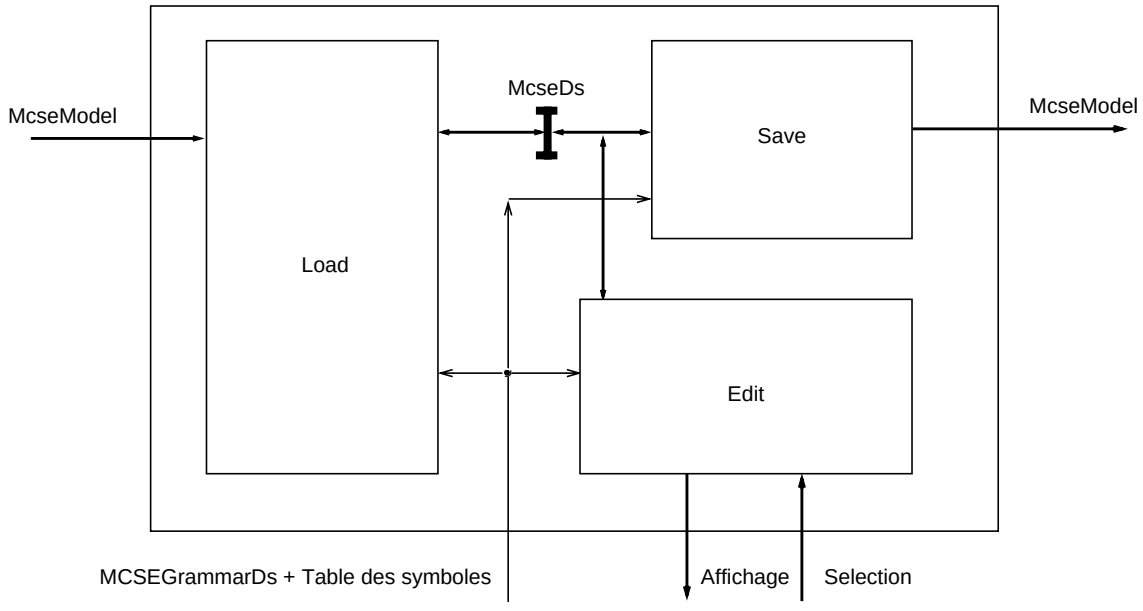
Ce résultat a été obtenu grâce à l'utilisation d'un classe java disponible sur le web réalisant la fonction de Meta-Parser [HUDSON-96].

5.6 STRUCTURE INTERNE GENERIQUE DE CHAQUE OUTIL

La technique de chargement du modèle MCSE étant maîtrisée, il s'agit de compléter l'organisation autour de la structure de données McseDs nécessaire pour chaque outil.

Il est clair maintenant que la structure de données interne est l'équivalent du texte source McseModel. Le texte est indispensable car il peut se lire et se mémoriser sous la forme de fichier. La structure de données en interne est aussi indispensable car c'est la seule forme exploitable efficacement par des programmes.

Disposant de la fonction Load, il faut aussi la fonction inverse appelée Save. De plus pour pouvoir modifier la structure interne, l'enrichir par exemple, il faut ajouter une fonction Edit.



-Figure 5.6- Structure interne générique de chaque outil.

La fonction Save est simplement un programme de parcours de la structure interne assurant sa transcription sous forme textuelle et son formatage par indentation et contrôle des retours de ligne. La signification de la structure McseDs se trouve par exploitation de la structure MCSEGrammarDs qui représente la grammaire. MCSEGrammarDs contient donc aussi les règles de formatage. La table des symboles est aussi indispensable pour l'écriture textuelle des symboles terminaux.

La fonction Edit peut avoir un rôle varié. Un premier rôle est la modification de la structure par une édition textuelle assistée par la syntaxe. Un deuxième rôle est la modification de la structure par une édition graphique assistée. Un autre catégorie de rôle est la transformation automatique ou assistée de la structure. Ce type de rôle permet d'entrevoir des possibilités intéressantes par exemple pour la réalisation de générateurs de solutions et de code.

5.7 PRINCIPE DE GENERATION DE CODE

Le principe de génération de code retenu est d'exploiter la structure de données équivalente d'un fichier template (ou fichier modèle) écrit dans le langage cible noté ici X. Le contenu de ce fichier template va comprendre toutes les constructions qui vont se retrouvées dans le programme cible. Chaque construction sera définie sous sa forme la plus complète et uniquement une fois. Un tel template permet alors de disposer en interne de la structure XGrammarDs, la table des symboles et la structure de données XTemplateDs pour toute production d'une structure représentative du code de sortie.

Ainsi, la production d'un programme ou d'un texte selon un langage cible nécessite de disposer de sa grammaire. Connaissant la grammaire, on peut alors disposer d'un analyseur syntaxique capable d'analyser un texte modèle et de produire la structure de données interne équivalente. Pour la compréhension de la méthode utilisée, nous commençons par une analyse des textes source et résultat voulu pour en déduire ce que doit être un template.

5.7.1 Analyse des textes source et résultat

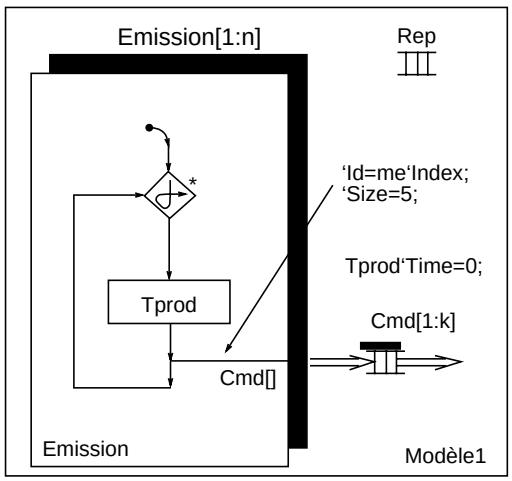
Pour comprendre le principe de génération basé sur l'emploi d'un template, considérons tout d'abord une très petite partie d'un exemple de modèle MCSE et du programme VHDL correspondant.

-A- Description du modèle MCSE

Ce modèle contient la description d'un système décrit sous la forme d'une structure qui elle même contient 2 ports et une fonction. Le port Cmd est un vecteur. La fonction Emission est multiple. Sa description est décrite par un comportement.

```
<Component> Serveur Generic n, k ( In Mess E: DefE; Out Var S: DefS;)
<Structure> Modele1;
     <Port> [1:k] Cmd : DefCmd;
     <Port> Rep: DefRep;
          <Component> [1:n] Emission
          (Out Mess Cmd[] : DefCmd;);
          <Attributes>
          Cmd[]'Id = me'Index;
          Cmd[]'Size = 5;
          Tprod'Time = 0;
          <Behavior> Modele2;
          Emission::{Tprod&!Cmd[]}*;
          <EndBehavior>
          <EndComponent>
<EndStructure>
<EndComponent>
```

a) Représentation textuelle



b) Représentation graphique

-Figure 5.7- Exemple de modèle MCSE partiel.

-B- Résultat VHDL souhaité

Compte tenu du chapitre 4, la traduction souhaitée pour avoir un modèle VHDL non-interprété est la suivante.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;
LIBRARY STANDARD;
USE STANDARD.TEXTIO.all;
LIBRARY ServeurLibrary;
USE ServeurLibrary.ServeurDeclaration.all;
ENTITY Serveur IS
GENERIC (      n : INTEGER := 1;
               k : INTEGER := 1);
PORT (    SIGNAL E: IN DefE;
          SIGNAL S: OUT DefS;);
END Serveur;
ARCHITECTURE behavioral OF Serveur IS
FOR all : PortObject
                    USE ENTITY Library.PortObject (behavioral);
.............................
SIGNAL CmdInputAccesses : DefPortInVector(1 TO n);
ALIAS EmissionCmd : DefPortInVector(1 TO n) IS CmdInputAccesses(1 TO n);
SIGNAL CmdNumberOfCurrentAccesses : DefNaturalVector(1 TO n);
SIGNAL CmdNumberOfMessagesInPort : DefNaturalVector(1 TO n);
BEGIN
 -- Instanciation des ports Cmd[1:k] et Rep
 MultipleCmd :
  FOR i IN 1 TO k GENERATE
  Cmd :
   COMPONENT PortObject
   GENERIC MAP (NbUsersInput =>1,
                NbUsersOutput =>1,
                policy =>Fifo,
                concurrency =>1,
                capacity =>1,
                WriteTime =>NullTime,
                ReadTime =>NullTime)
   PORT MAP (InputAccesses (1) =>CmdInputAccesses (i),
             OutputAccesses (1) =>CmdOutputAccesses (i),
             NumberOfCurrentAccesses =>CmdNumberOfCurrentAccesses (i),
             NumberOfMessagesInPort =>CmdNumberOfMessagesInPort (i));
  END GENERATE MultipleCmd;
Rep :
   COMPONENT PortObject
   GENERIC MAP (NbUsersInput =>1,
                NbUsersOutput =>1,
                policy =>Fifo,
                concurrency =>1,
                capacity =>1,
                WriteTime =>NullTime,
                ReadTime =>NullTime)
   PORT MAP (InputAccesses (1) =>RepInputAccesses (1),
             OutputAccesses (1) =>RepOutputAccesses (1),
             NumberOfCurrentAccesses =>RepNumberOfCurrentAccesses (1),
             NumberOfMessagesInPort =>RepNumberOfMessagesInPort (1));
-- Instanciation des fonctions
MultipleEmission :
  BLOCK
       PORT (Cmd : OUT DefPortInVector(1 TO n));
       PORT MAP (Cmd => EmissionCmd);
  BEGIN
   MultipleInstantiationEmission :
```

```
      FOR i IN 1 TO n GENERATE
      Emission :
       BLOCK
            PORT (Cmd : OUT DefPortIn);
            PORT MAP (Cmd => Cmd (i));
       CONSTANT me : INTEGER := i;
       BEGIN
        EmissionBehavior : PROCESS
                                  CONSTANT TprodTime : TIME := 0;
                                  VARIABLE InfoCmd : DefTprod;
        BEGIN
         LOOP
          Delay (TprodTime);
          InfoCmd.Id := me;
          InfoCmd.Size := 5;
          Send (Cmd, InfoCmd, NullTime, 0, now);
         END LOOP;
        END PROCESS EmissionBehavior;
       END BLOCK Emission;
      END GENERATE MultipleInstantiationEmission;
  END BLOCK Serveur;
 END behavioral;
```

Le programme commence par des déclarations fixes. On trouve ensuite la librairie utilisée puis la déclaration de l'entité. La partie Architecture contient la déclaration de signaux et alias nécessaires pour les ports. On trouve ensuite l'instanciation de composants PortObject pour les ports Cmd et Rep. Le port multiple Cmd exploite l'instruction VHDL Generate.

On trouve pour finir l'instanciation de la fonction multiple Emission. Celle-ci est décrite comme un block avec ses entrées et sorties. L'instanciation multiple s'obtient aussi par l'instruction Generate. Le corps de la fonction possède un comportement séquentiel. Un process est utilisé à cet effet. On trouve dans ce process les instructions représentant le comportement.

Pour cette description, nous avons mis en gras les mots qui sont spécifiques de l'exemple.

On peut de cette manière comprendre aisément l'idée du template. Ainsi, si on considère la transcription du port Cmd[1:k], il faut produire toutes les lignes comprises entre "Multiple**Cmd"** et "END GENERATE Multiple**Cmd**;". Il faut au préalable ajouter 3 signaux et un alias. Il en est de même pour l'instanciation d'une fonction décrite par un comportement. Toute la structure entre "Multiple**Emission"** et "END GENERATE MultipleInstantiation**Emission**;" doit être reproduite pour chaque fonction. L'adaptation porte alors sur certains champs: uniquement le nom **Cmd** pour le port.

5.7.2 Le concept de fichier Template

La description suivante correspond au template possible pour produire le programme précédent. Les lettres en gras sont **X** pour une chaîne de caractères, un **N** pour un nombre. Dans ce template certains champs alternatifs (par exemple IN, OUT ou INOUT) sont aussi indéfinis.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;
LIBRARY STANDARD;
USE STANDARD.TEXTIO.all;
LIBRARY XLibrary;
USE XLibrary.XDeclaration.all;
ENTITY X IS
 GENERIC (X : INTEGER := N;);
```

```
 PORT (SIGNAL X: X X;);      /*Choix entre IN, OUT, INOUT*/
END X;
ARCHITECTURE behavioral OF X IS
FOR all : PortObject
          USE ENTITY Library.PortObject (behavioral);
.............................
SIGNAL XInputAccesses : DefPortInVector(1 TO N);
ALIAS X : DefPortInVector(1 TO N) IS XInputAccesses(1 TO N);
SIGNAL XNumberOfCurrentAccesses : DefNaturalVector(1 TO N);
SIGNAL XNumberOfMessagesInPort : DefNaturalVector(1 TO N);
BEGIN
 -- Instanciation d'un port
 MultipleX :
  FOR i IN 1 TO N GENERATE
 X :
   COMPONENT PortObject
   GENERIC MAP (NbUsersInput =>1,
                NbUsersOutput =>1,
                policy =>Fifo,
                concurrency =>1,
                capacity =>1,
                WriteTime =>NullTime,
                ReadTime =>NullTime)
   PORT MAP (InputAccesses (1) => XInputAccesses (i),
             OutputAccesses (1) => XOutputAccesses (i),
             NumberOfCurrentAccesses => XNumberOfCurrentAccesses (i),
             NumberOfMessagesInPort => XNumberOfMessagesInPort (i));
  END GENERATE MultipleX;
-- Instanciation d'une fonction
MultipleX :
  BLOCK
      PORT (X : X DefPortInVector(1 TO N));     /* X vaut IN, OUT, INOUT*/
    PORT MAP (X => X);
  BEGIN
   MultipleInstantiationX :
    FOR i IN 1 TO N GENERATE
    X :
     BLOCK
          PORT (X : X DefPortIn);
          PORT MAP (X => X (i));
     CONSTANT me : INTEGER := i;
     BEGIN
      XBehavior : PROCESS
           CONSTANT XTime : TIME := N;
           VARIABLE InfoX: X;
      BEGIN
       LOOP
        Delay (XTime);
        InfoX.X := X;
          Send (X, InfoX, NullTime, 0, now);
       END LOOP;
      END PROCESS XBehavior;
     END BLOCK X;
    END GENERATE MultipleInstantiationX;
 END BLOCK X;
END behavioral;
```

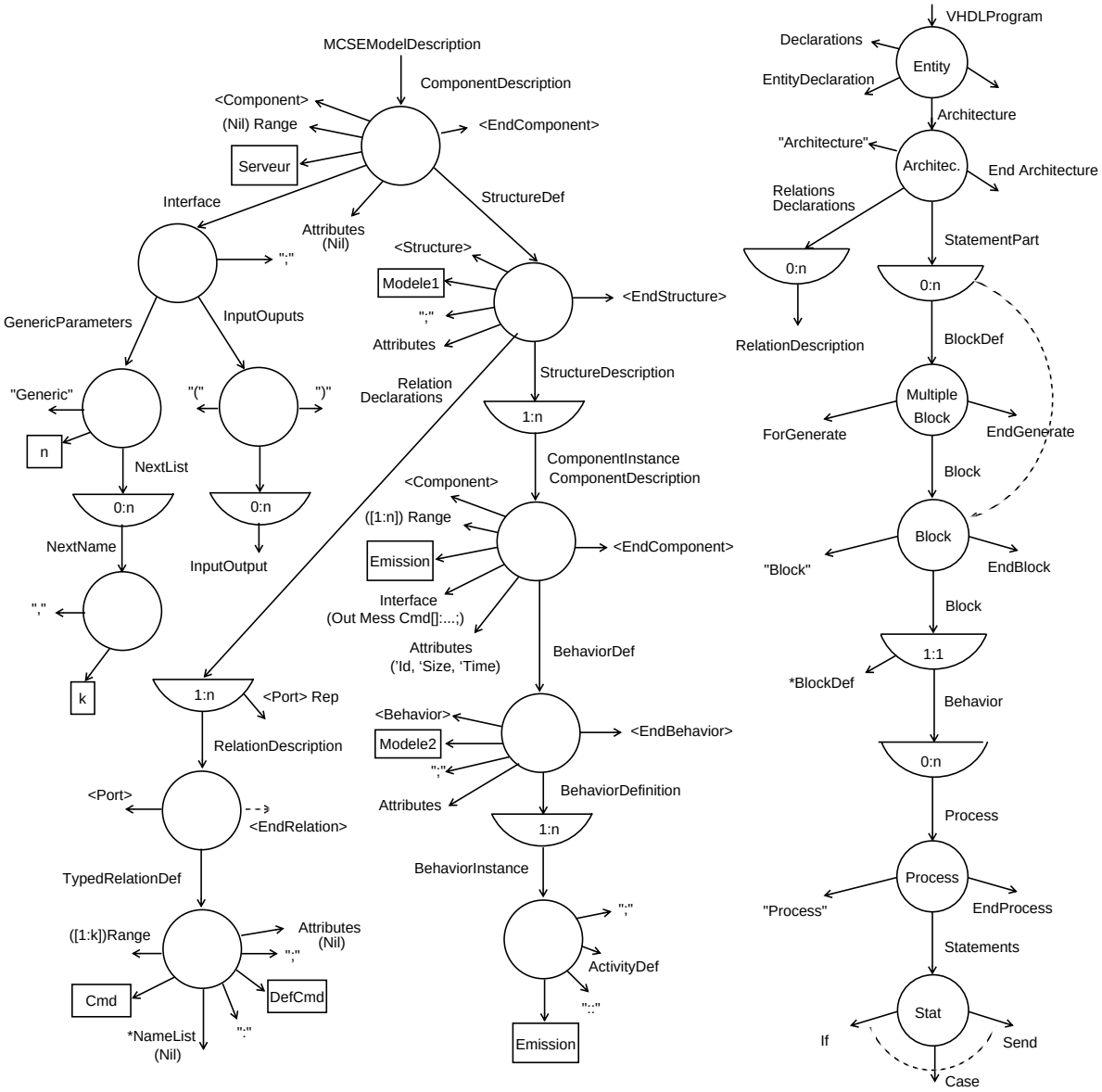
Ce template n'est bien sûr pas exhaustif pour la génération d'un programme VHDL non-interprété complet. Il est ici simplement considéré suffisant pour l'exemple. Il ne sert donc qu'à illustrer la méthode utilisée.

Il est aisé de comprendre que, par exemple, la création de chaque port ou chaque fonction va nécessiter la copie des lignes de VHDL du template pour chaque élément. Il faut ensuite assurer la mise à jour, par exemple modifier le nom **X** en **Cmd**. Un port simple ou une fonction simple se déduit d'un élément multiple par simplification. Ceci est mis en évidence dans le paragraphe suivant.

Pour faciliter la production, un template doit contenir la description la plus complexe. Toute copie du modèle est alors ajustée par changement de valeurs et par élimination des parties non-utiles.

5.7.3 Analyse des structures de données

Pour comprendre la technique de génération, la figure suivante représente (partiellement) la structure de données du modèle MCSE de l'exemple ci-dessus et la structure de données du template VHDL.



-Figure 5.8- Structure de données pour le modèle MCSE et le template VHDL.

La démarche de génération peut être la suivante:

1- Copie de la structure complète du template comme structure du programme de sortie.

2- Mise à jour du champ Declarations en remplaçant X par Serveur.

3- Mise à jour du champ EntityDeclaration en utilisant le nom Serveur puis en créant les paramètres génériques et entrées/sorties par exploitation de la structure Interface de MCSE.

4- L'architecture est ensuite mise à jour par le nom du modèle (Modele1) et le nom du composant Serveur.

5- Il s'agit ensuite de créer autant de relations que dans le modèle MCSE avec mise à jour des noms X pour les signaux et alias de chaque relation. Si une relation n'est pas multiple, il faut supprimer la structure Generate, ce qui se fait simplement par élimination du noeud MultipleBlock (idem à la déclaration des composants).

6- La définition de blockDef est ensuite mise à jour pour l'instanciation de n blocs Emission. Il s'agit de modifier la valeur n, le nom du block, de créer ses entrées et sorties conformément à la déclaration de l'interface de Emission dans le modèle MCSE. Si le composant n'est pas multiple, il suffit de détruire le noeud MultipleBlock (lien en pointillé sur la figure).

7- Il faut ensuite assurer la mise à jour de la partie comportementale (Behavior). Le process est mis à jour compte-tenu de la déclaration du comportement par ActivityDef. Si plusieurs process sont nécessaires, chacun est créé à l'image du process dans le template.

8- il suffit ensuite de transcrire la structure sous forme textuelle pour avoir le programme résultat souhaité.

Ainsi l'explication précédente montre que dans le cas de cet exemple, la structure du résultat s'obtient par une seule copie de la structure complète du template, puis par sa mise à jour. La mise à jour inclut la destruction des champs inutiles car le template est une version complète, la modification de noms et valeurs de champs, l'ajout d'éléments supplémentaires dans les ensembles par copie du modèle dans le template.

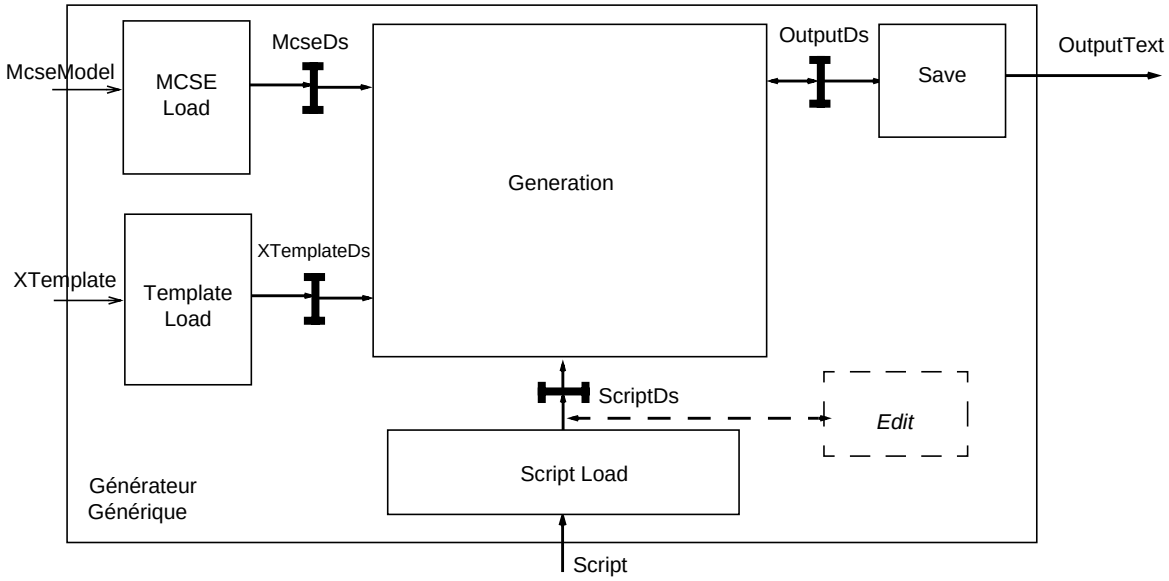
5.8 STRUCTURE ET FONCTIONNEMENT D'UN GENERATEUR

Dans ce paragraphe nous expliquons la structure d'un générateur et son principe de fonctionnement.

5.8.1 Structure du générateur

Un générateur basé sur le principe expliqué dans les paragraphes précédents exploite en entrée les 2 structures McseDs et XTemplateDs. Le résultat en sortie est la structure OutputDs. Les 2 structures en entrée sont obtenues par l'analyseur syntaxique Loader paramétré par la grammaire du texte. Chacune des structures possède une référence sur sa grammaire (GrammarDs) en incluant la table des symboles pour permettre son interprétation. La structure de sortie nécessite aussi de se référer à une grammaire, ici celle de XtemplateDs. La sortie textuelle se fait avec la fonction Save qui exploite alors cette structure de données et la grammaire pour le formatage.

La figure 5.9 montre l'organisation d'un générateur.



-Figure 5.9- Structure interne d'un générateur.

Un générateur pour un programme ou texte donné en sortie peut se concevoir comme un programme écrit en C, C++ ou Java assurant les opérations nécessaires sur les structures de données et selon l'ordre approprié. Cette solution conduit à devoir développer autant de générateurs que de résultats différents en sortie. On peut aisément imaginer que les générateurs devraient posséder des opérations en commun par suite de similitudes. Ne peut-on pas alors pousser plus loin le raisonnement en cherchant une solution d'un seul générateur programmable de l'extérieur. On arrive alors à l'idée d'un script définissant le comportement souhaité. Nous avons retenu cette idée car elle devrait permettre à toute autre personne de développer un générateur spécifique sans avoir à connaître et à intervenir dans le code du générateur. Pour cela, il suffit que toutes les opérations nécessaires soient possibles par le script.

5.8.2 Fonctionnement du générateur

La génération est le résultat de l'exécution d'un programme se conformant à un script de génération. Plutôt que d'effectuer l'interprétation du texte du script pour l'exécution, le script est aussi chargé en mémoire sous la forme d'une structure de données comme le montre la figure 5.9. Le programme de génération doit donc assurer le parcours de la structure du script pour effectuer les opérations de manipulation de OutputDs.

Le fonctionnement possible pour un générateur piloté par un script est alors le suivant:

- Il commence par lire le fichier texte McseModel de manière à construire la structure interne McseDs. De la même manière, il va créer la structure interne XTemplateDs à partir du fichier texte XTemplate.

- La structure de sortie OutputDs est créée à l'image de XTemplateDs par copie de parties de structures en se basant sur une analyse de la structure McseDs.

- Il met ensuite à jour la structure créée dans OutputDs compte-tenu des champs des noeuds de McseDs. La mise à jour veut dire: placer les noms appropriés des identificateurs, supprimer les champs inutiles car le template contient la structure la plus complète, mettre à jour les listes correspondant aux ensembles.
- Il procède ainsi pour toute la structure McseDs par récurrence et/ou récursivité.
- Lorsque la structure est complète, le programme code est produit dans un fichier par la fonction Save qui exploite la XGrammarDs pour l'interprétation et sa table des symboles pour l'écriture des champs terminaux.

Ce principe de générateur s'avère ainsi relativement indépendant du langage à produire. Il s'agit donc de développer la fonction Generation selon une approche d'automate d'exécution s'appliquant sur le parcours du script représenté en interne par sa structure de données. Le script doit être construit sur la base des opérations générales de parcours, copie et mise à jour de structures.

Sur la figure précédente nous avons grisé la fonction Edit. La question est: peut-on exploiter Edit dans un générateur? Edit peut par exemple servir à proposer des alternatives. Ainsi ne peut-on pas utiliser Edit pour sélectionner une solution particulière pour la génération lorsque plusieurs sont possibles? C'est le cas par exemple en VHDL pour une variable partagée à plusieurs écritures. Les différentes solutions peuvent être dans le template.

Une autre suggestion est de constituer plusieurs templates différents et de choisir le template le mieux adapté. Ainsi dans le cas du langage C avec l'emploi d'un exécutif temps-réel, il est utile de pouvoir "fitter" sur plusieurs exécutifs temps réel. La différence est au moins dans les primitives: noms et arguments, ainsi que dans les déclarations.

Un éditeur orienté par la syntaxe peut aussi servir à l'édition de tout texte défini par une grammaire. Ainsi, il peut donc tout à fait servir pour l'édition du script. Une telle édition peut s'imaginer se faire d'une manière interactive; choix d'une action et visualisation immédiate du résultat.

5.9 DEFINITION DU SCRIPT

La question fondamentale pour définir la grammaire du script concerne les opérations nécessaires pour assurer toute transformation de structure. On peut donc se dire qu'il s'agit de rechercher un langage de manipulation de structures.

Pour répondre à cette question, nous commençons par analyser l'ensemble des opérations élémentaires nécessaires puis nous proposons une technique d'exécution.

5.9.1 Opérations sur les structures

La présentation de la méthode faite dans le paragraphe précédent permet d'extraire assez facilement les opérations de base nécessaires. Nous retenons la liste suivante:

- Chargement d'une structure à partir d'un texte,
- Sauvegarde d'une structure sous forme textuelle,
- Copie d'une structure complète à partir du point de désignation,
- Copie d'un noeud d'une structure,
- Destruction d'une structure,
- Destruction d'un noeud,
- Mise à jour d'un champ d'un noeud,

- Ajout d'un élément dans un ensemble.

-A- Chargement d'une structure à partir d'un texte: LoadDs

Les structures nécessaires pour la génération (par exemple McseDs et XTemplateDs) sont chargées par l'opération **RefDs := LoadDs(AccessFileName, Grammar, mode);**.

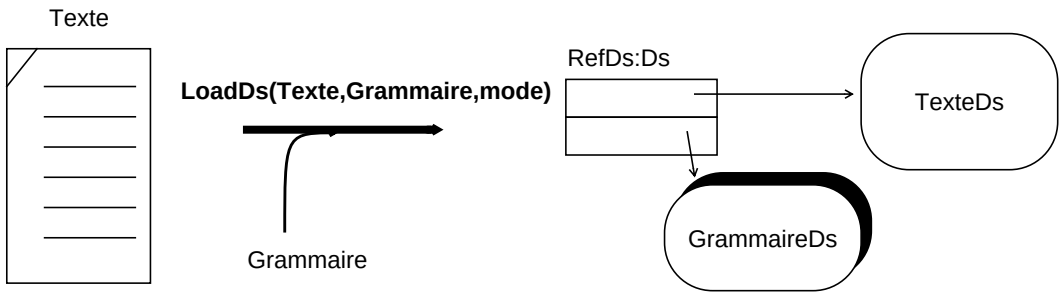
- **AccessFileName** est le chemin d'accès au fichier. Pour que l'outil soit indépendant du système de fichiers, le chemin est défini à l'aide d'une variable prédéfinie FileSeparator qui représente le caractère de séparation d'un "path". L'utilisateur du script peut également utiliser d'autres variables systèmes: ProjectPath pour le chemin du projet, ToolsPath pour le chemin des outils et Date pour la date courante.

- **Grammar** définit la grammaire à utiliser pour l'analyse et la construction de la structure de données.

- **mode** indique le format du fichier à charger. Le ficher peut être du type Ascii (mode=0) et dans ce cas il est chargé par un analyseur syntaxique. Mais il peut également être du type Binaire (mode=1) et dans ce cas on utilise le mécanisme de sérialisation/ désérialisation du langage JAVA pour le charger. Un fichier sérialisé sera chargé plus rapidement qu'un fichier texte, mais il n'est pas lisible.

- Le résultat **RefDs** est une référence sur la structure de données créée. Mais comme une structure de données n'a de signification qu'avec sa grammaire, le résultat va contenir aussi une référence sur la structure de données de la grammaire GrammarDs qui a été chargée pendant l'opération LoadDs. GrammarDs contient aussi bien-sûr la table des symboles.

L'opération peut se représenter par la figure suivante.



-Figure 5.10- Opération LoadDs.

-B- Sauvegarde d'une structure sous forme textuelle: SaveDs

La sauvegarde est l'opération strictement inverse du LoadDs. La syntaxe est: **SaveDs(RefDs, AccessFileName);**. RefDs et AccessFileName ont les mêmes significations et définitions que ci-dessus.

Une bonne vérification consiste à écrire le script suivant:

RefDs := LoadDs(Texte1, McseModel,0); SaveDs(RefDs, Texte2);

On doit alors trouver dans Texte2 la même description que Texte1 mais avec en plus un formatage qui peut être meilleur si Texte1 ne respectait pas les règles de présentation.

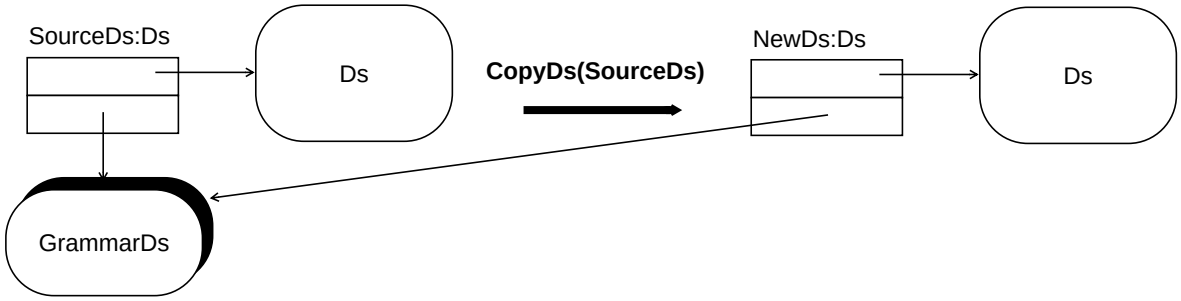
-C- Copie d'une structure complète: CopyDs

L'objectif est d'obtenir une nouvelle structure image complète de sa source. Une telle copie concerne la duplication de tous les noeuds de la structure, de tous les identificateurs et valeurs, de la mise à jour de tous les champs de cette nouvelle structure.

La copie doit être telle que la destruction de la structure source doit permettre son remplacement par sa copie.

La syntaxe est la suivante: **NewDs := CopyDs(SourceDs);**

L'opération peut se représenter par la figure suivante.



-Figure 5.11- Opération CopyDs.

La référence sur la nouvelle structure inclut la référence sur sa grammaire pour l'interprétation. La structure de la grammaire n'est pas copiée.

Une vérification de cette opération peut être la suivante:

```
SourceDs := LoadDs(Texte1, Grammaire,0);
NewDs := CopyDs(SourceDs);
DelDs(SourceDs);
Save(NewDs, Texte2);
```

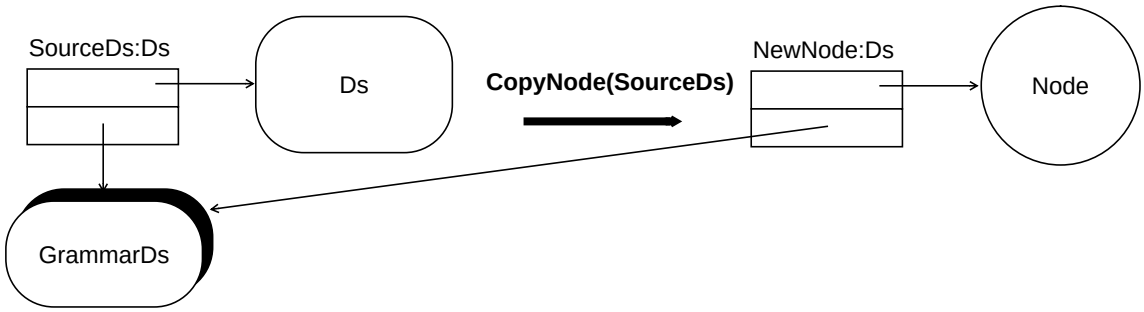
On doit alors retrouver en sortie Texte2 le même contenu que Texte1.

-D- Copie d'un noeud d'une structure: CopyNode

Il s'agit ici de dupliquer uniquement le noeud (et donc pas ses noeuds fils). La duplication exploite le modèle du noeud dans la grammaire. Tous les champs terminaux sont donc existants. Les champs contenant des références sur d'autres noeuds ou sur des valeurs ou identificateurs sont vides (Nil).

La syntaxe est la suivante: **NewNode := CopyNode(SourceDs);**

L'opération peut se représenter par la figure suivante.



-Figure 5.12- Opération CopyNode.

La référence sur la nouvelle structure inclut aussi la référence sur la grammaire source pour l'interprétation. Comme chaque noeud possède son type même après duplication, les champs du noeud résultat sont définis par ceux de son noeud modèle dans la structure de la grammaire. En fait, un noeud est simplement le cas particulier d'une structure à un seul élément.

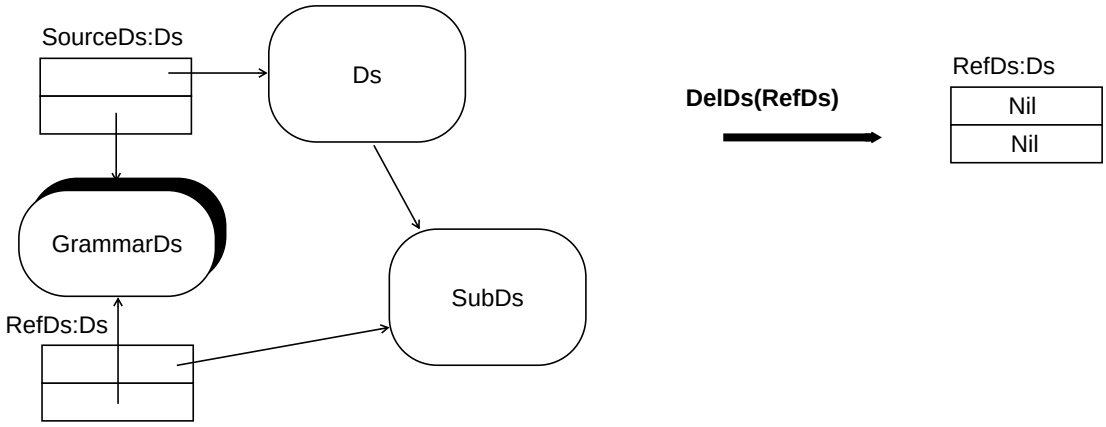
Nous avons aussi considéré utile d'ajouter l'opération **CreateNode(GrammarDs, NodeType)**; qui a pour objectif de créer un noeud sans avoir un exemplaire disponible. La création se fait à partir du modèle de la grammaire. L'argument est alors la grammaire et le type de noeud.

-E- Destruction d'une structure: DelDs

La destruction d'une structure définie par une référence permet de récupérer de la place en mémoire ou permet de simplifier une structure plus globale.

La syntaxe est la suivante: **DelDs(RefDs);**

L'opération peut se représenter par la figure suivante.



-Figure 5.13- Opération DelDs.

La référence sur la structure détruite n'est plus utilisable car possédant un contenu Nil. Pour l'exploitation correcte de cette opération pour le cas représenté par la figure, il faut que la référence sur SubDs dans Ds doit aussi être supprimée. Ceci est possible avec l'opération de modification d'un champ décrite ci-après.

La destruction de toutes les structures d'une même grammaire peut conduire à la destruction de la grammaire de manière à réduire la place mémoire.

-F- Destruction d'un noeud: DelNode

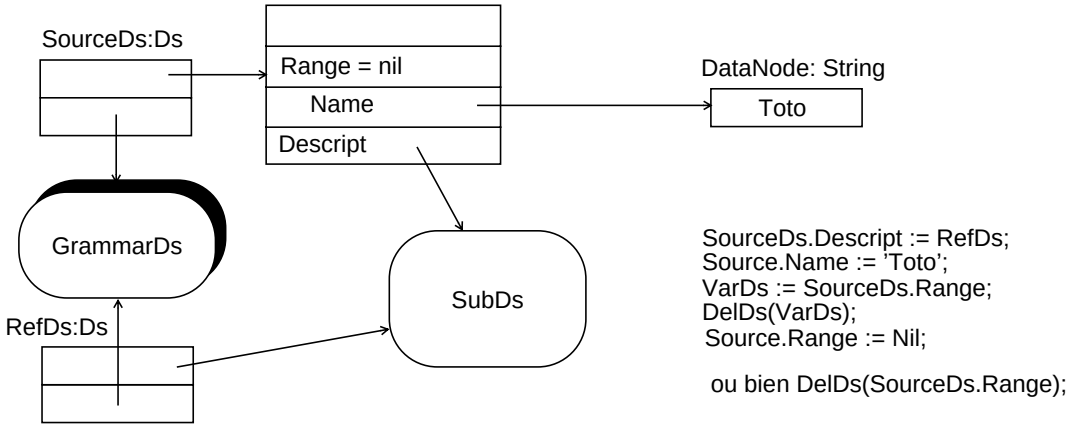
Cette opération n'a de signification que s'il s'agit d'un noeud isolé (sans références). Dans ce cas il s'agit d'un cas particulier d'une structure. La destruction d'un seul noeud d'une structure plus globale n'a pas de sens car alors les noeuds fils (ceux référencés) ne sont plus référencés et donc inutilisables.

-G- Mise à jour d'un champ: ModifField

La modification d'un champ est une opération importante. Il s'agit de pouvoir affecter une nouvelle valeur. Il y a 2 catégories de noeuds: le noeud de composition (NodeDs), le noeud de valeur (DataNode). Le noeud de valeur est un noeud élémentaire contenant un nombre ou un identificateur. Le noeud de composition contient un ensemble de champs de référence. Chaque champ de référence pointe sur un autre noeud et donc sur une structure. L'assignation d'un champ est donc fonction de la catégorie concernée. Un champ peut aussi être mis à vide (Nil).

La syntaxe est la suivante: **RefDs.Field := [Ds | Val | Nil];**

La figure suivante représente les 3 types d'opérations.



-Figure 5.14- Opération ModifField.

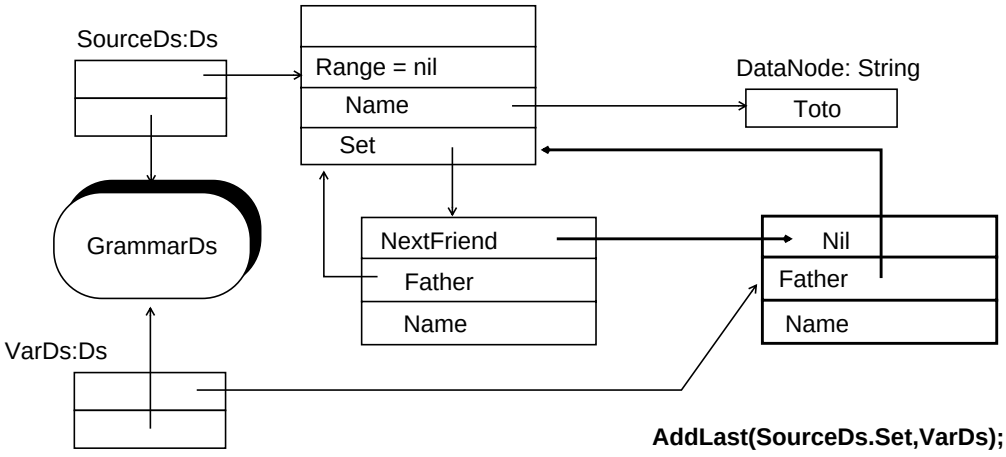
Une variante est indiquée pour le champ Range. Dans la grammaire du modèle MCSE, le champ Range est une référence sur une structure. Si Range ne doit pas exister, il s'agit de détruire la structure et d'affecter le champ à Nil. Dans ce cas, l'opération DelDs est suffisante.

-H- Ajout d'un élément dans un ensemble: AddFirst, AddLast, AddAfter

les 3 opérations AddFirst, AddLast et AddAfter servent à la mise à jour d'un ensemble d'éléments. Il s'agit de pouvoir ajouter une structure ou une valeur en début de liste, en fin de liste ou après un élément donné de la liste.

La syntaxe est la suivante: **AddLast(RefDs.Field,VarDs);**

La figure suivante représente la structure d'un ensemble et l'effet de l'opération.



-Figure 5.15- Opération AddLast.

SourceDs.Set désigne l'ensemble pour lequel il faut ajouter la structure référencée par VarDs. Compte-tenu du choix adopté pour l'implantation d'un ensemble, il s'agit d'effectuer les opérations plus élémentaires suivantes:

TmpDs := SourceDs.Set;
While TmpDs # Nil do TmpDs := TmpDs.NextFriend;
Tmps.NextFriend := VarDs;

VarDs.NextFriend := Nil;
VarDs.Father := SourceDs;

L'ajout en début de l'ensemble ou après un élément donné est assez similaire.

5.9.2 Instructions d'enchaînement

Après avoir défini les opérations de base, il faut pouvoir construire une instruction ou une transformation plus complexe. Pour décider des constructions nécessaires, nous partons des bases de la programmation structurée qui recommande l'emploi des 3 constructions suivantes:

- Exécution séquentielle,
- Répétition,
- Exécution conditionnelle.

Un ensemble d'opérations peuvent se regrouper en une procédure, ce qui va se traduire par une règle de construction ou de transformation. Nous retenons les instructions suivantes:

- Règle = suite d'instructions ou de règles exécutées en séquence,
- While Condition do règle;
- ForEach élément do règle; instruction appropriée pour les ensembles
- Case pour l'exécution conditionnelle.

Ces instructions sont décrites dans les paragraphes suivants.

-A- Définition d'une règle composite

Une règle de transformation ou de construction se définit comme une suite de règles plus élémentaires ou d'opérations telles que celles décrites auparavant.

La syntaxe est la suivante: **R ::= { R1; R2; ..... Rn; }**

La traduction de cette règle sous la forme d'une structure conduit à un vecteur des règles plus élémentaires. La récursivité est possible en permettant l'utilisation de règles plus englobantes.

-B- Instruction While

Il est indispensable de pouvoir répéter une règle jusqu'à une condition d'arrêt. La syntaxe est la suivante: **While(Condition : Règle);**

La condition peut être élémentaire ou composée de conditions élémentaires avec les opérateurs logiques ou (OR), et (AND) et ou exclusif (XOR). Une condition élémentaire est booléenne et limitée aux tests simples Egal (=), Différent (#), Inférieur (<) et supérieur (>). Il s'agit de tester la valeur de champs.

-C- Instruction Case

L'exécution conditionnelle englobe l'instruction If et l'instruction Case. On retient l'instruction Case car elle englobe la première. La syntaxe retenue est la suivante:

Case(Condition1 : Règle1 | Condition2 : Règle2 ..... [| Else : RègleN]);

Le cas Else est optionnel. Une condition se définit comme pour le While.

-D- Instruction ForEach

Cette instruction porte sur la manipulation d'un ensemble d'éléments. Il s'agit par exemple de pouvoir exécuter une règle pour chaque élément de l'ensemble. La syntaxe retenue est la suivante: **ForEach(Set : Règle);**

Set est une référence sur un ensemble. Les éléments de l'ensemble constituent une liste chaînée avec le pointeur NextFriend.

5.9.3 Définition des variables

L'analyse qui précède montre que les références ou valeurs sont de 3 natures: Référence sur une structure de données, chaîne de caractères, Nombre. La clarté d'un script passe par la déclaration des variables utilisées dans celui-ci. Des opérations particulières sont ensuite nécessaires pour faciliter l'écriture du script.

-A- Déclarations

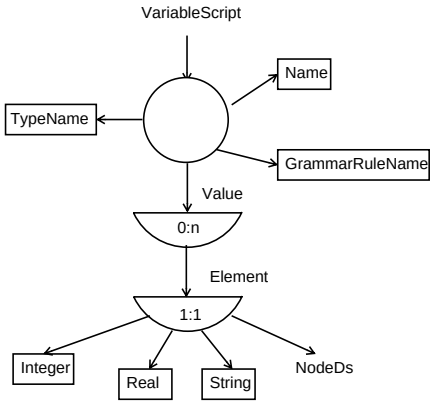
Pour éviter les erreurs et faciliter la transcription du script pour une exécution efficace, il est judicieux d'associer un type à chacun.

Les variables désignant un scalaire sont du type String ou du type Value.

Les variables désignant une structure ou un noeud doivent être typées avec la grammaire. Ceci est nécessaire pour donner une signification aux champs de la structure ou du noeud référencé. Ceci a été montré comme indispensable pour les opérations CopyDs et CopyNode.

D'où la déclaration: **Var : [Value | String | Grammar];**

La structure de données d'une variable du script pour l'exécution est représentée par la figure suivante.



Pour une variable désignant une structure, la variable doit aussi avoir une référence sur la règle de grammaire concernée. Les valeurs des variables sont gérées sous la forme de pile pour permettre la récursivité des règles du script. Ceci explique l'ensemble 0:n

-Figure 5.16- Structure de données pour une variable du script.

Certaines opérations nécessitent d'employer des données prédéterminées. Il s'agit alors de constantes qu'il faut pouvoir déclarer pour faciliter leur modification globale.

D'où la déclaration: **Const [Number | 'String'];**

-B- Portée des variables

Par rapport à un langage classique et pour des raisons de simplicité, les règles d'un script n'ont pas d'arguments d'appel ni de variables locales. Ceci oblige le développeur de Script à maîtriser parfaitement la traçabilité des variables qu'il utilise et à empiler et désempiler explicitement les variables utilisées par des règles récursives. Pour éviter d'avoir une liste trop longue d'instructions **Push(Var);** et **Pop(Var);** , une clause LocalVisibility a été rajoutée à la définition d'une règle. Les variables déclarées dans cette liste sont implicitement empilées lors de l'appel de la règle et désempiler à la fin de son exécution.

LocalVisibility::=**LocalVisibility** name {**,** name}**;**

Travailler uniquement par variables globales offre un avantage du point de vue efficacité du code, mais se paie évidemment par une plus mauvaise lisibilité du code.

5.9.4 Manipulations des variables

Des manipulations des variables du type Value et String sont indispensables.

-A- Les opérations sur les chaînes de caractères

Les opérations de manipulations des chaînes de caractères retenues sont:

- la conversion d'un nombre sous la forme d'une chaîne de caractères (ToString),
- la concaténation de plusieurs chaînes de caractères (&),
- l'obtention du type (nom de la règle de grammaire) d'un noeud de structure (TypeOf),
- l'obtention de la première chaîne rencontrée dans un record (LabelOf),
- l'obtention de la longueur d'une chaîne de caractères (LengthOf),
- l'extraction d'une sous-chaîne dans une chaîne de caractères (SubStringOf).

Les syntaxes de ces instructions sont:

- **VarString := ToString(VarValue);**
- **VarString := VarString1 & VarString2 .....& VarStringN;**
- **VarString := TypeOf(RefDs.Field);**
- **VarString := LabelOf(RecordDs);**
- **VarValue := LengthOf(VarString);**
- **VarString2 := SubStringOf(VarString1, IndexMin, IndexMax);**

Les générateurs de code qui seront obtenus à partir du méta-générateur MetaGen doivent permettre de produire un programme complet pour faire une simulation fonctionnelle ou une synthèse. Dans ce cas, des comportements sont écrits directement sous une forme algorithmique. Le générateur doit alors intégrer ce code saisi manuellement au code généré automatiquement. Pour cela, nous avons défini l'instruction IncludeOp dont la syntaxe est:

VarString := IncludeOp(FileAccessName, mode)

L'inclusion d'un fichier texte peut se faire par copie (mode=0) ou par référence (mode=1). Pour le mode par référence, le fichier est inclut uniquement au moment de la sauvegarde de la structure de données concernée sous forme textuelle.

Lorsque l'on analyse un fichier template, certaines parties déclaratives de ce fichier n'auront pas à être modifiées pour la transcription. Au lieu de créer une sous-structure de données pour ces textes figés, le concepteur a la possibilité de les charger uniquement sous forme de chaînes de caractères en les délimitant par des mots clés spécifiques (NO_PARSE_TEXT et END_NO_PARSE_TEXT par exemple). L'exemple ci-dessous est une partie d'un fichier de template illustrant ces propos.

```
NO_PARSE_TEXT
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;
LIBRARY STANDARD;
USE STANDARD.TEXTIO.all;
END_NO_PARSE_TEXT
LIBRARY work;
```

```
USE work.Declaration.all;
ENTITY ApplicationName IS
...
```

Lors de la sauvegarde sous forme textuelle, il faut éliminer les délimiteurs de zones non analysées syntaxiquement. Pour cela, le concepteur dispose de l'instruction DelDelimiter dont la syntaxe est la suivante:

DelDelimiter(StringRef, BeginDelimiterString, EndDelimiterString);

Bien que ce soit déconseillé, le concepteur peut modifier ces zones de texte figées avec l'instruction ReplaceCharAt qui permet de changer un caractère d'une chaîne de caractères et dont la syntaxe est:

ReplaceCharAt(StringRef, Index, NewChar);

-B- Les opérations sur les nombres

Les opérations du script concernant les nombres sont:

- la conversion d'une chaîne de caractères en un nombre (ValueOf),
- l'obtention de l'heure courante (GetTime),
- l'obtention de la mémoire système libre (GetFreeMemory),
- le calcul d'expression avec les opérateurs +, - , * et /.

La syntaxe de ces instructions est:

- **VarValue := ValueOf(VarString);**
- **VarValue := GetTime(Unit);** avec Unit ::=ms | sec | min
- **VarValue := GetFreeMemory();**

5.9.5 Manipulations de liste

Une variable du script est implantée sous la forme d'une pile. Donc pour créer une liste d'éléments, il suffit de les empiler dans une variable à l'aide des instructions Push et d'assignation: **Push(Var); Var:=NewElement;**. Pour gérer ces piles, les instructions à disposition sont:

- l'obtention de la taille de la liste (SizeOf),
- l'obtention d'un élément de la liste à partir de son index (PeekAt),
- la recherche d'un élément dans la liste à partir de sa référence (IsIn),
- la recherche d'un élément dans la liste à partir de son type et de son nom (Search),
- la réinitilisation d'une variable (ResetVariable).

La syntaxe est la suivante:

- **VarValue := SizeOf(ListRef);**
- **NodeRef := PeekAt(ListRef,Index);**
- **VarValue := IsIn(ListRef,NodeRef);**
- **NodeRef := Search(ListRef,TypeName,NodeName);**
- **ResetVariable(VarRef);**

5.9.6 Configuration d'un script

-A- Configuration

L'interprétation du script et sa transcription en code Java peuvent être pilotées en fonction de paramètres contenus dans un fichier de configuration situé dans le répertoire du projet. Il s'agit dans le principe d'une sorte de pré-compilateur tel que celui du langage C. Seules les parties de Script validées par ces paramètres sont compilées et interprétées ou transcrites en code JAVA. Par exemple, un même script peut ainsi fournir un générateur pour C/VxWorks ou un générateur pour C/NoyauETRIreste.

La syntaxe est la suivante:

Configuration ExecutifETR;
...
Case (ExecutifETR='VxWork': ...;
| Else : ...):

-B- Importation de règles

Un script est un ensemble de règles. Il est possible d'importer et d'utiliser des règles d'un autre script. Ceci permet à l'utilisateur de pouvoir découper son code en différents petits morceaux de script. Cette possibilité de découpage a au moins deux avantages:

- elle réduit la taille des fichiers de code et donc améliore leur lisibilité,
- elle permet de créer des librairies de règles qui sont utilisables par différents générateurs et donc améliore la réutilisabilité du code.

La syntaxe est la suivante: **ImportOp NomRegle1,NomRegle2;**

Les chemins d'accès aux scripts contenant les règles à importer sont définis dans le fichier de configuration de l'outil.

-C- Contrôle de l'exécution

En mode interprété, le concepteur peut mettre des points d'arrêt dans son code script avec l'instruction **"BreakPoint;"**. Il a alors la possibilité de visualiser le contenu des variables, faire du pas à pas ou continuer l'exécution.

Le concepteur dispose également de l'instruction **"Error(msg);"** qui permet d'afficher un message et stopper l'exécution du générateur. A tout moment, il peut aussi envoyer un message avec les instructions **"Display(msg);"** et "**Warning(msg);"** sans stopper l'exécution du générateur.

Enfin, on retrouve comme instructions du script les méthodes spécifiques à l'exploitation du modèle MCSE et qui permettent de résoudre les problèmes de:

- portée des noms,
- traçabilité des liens entre éléments de relations et interfaces de composant,
- inclusion de modèle.

Ces méthodes ont été détaillées dans le chapitre 3 sur le modèle de performance de MCSE.

5.10 SPECIFICATION DE LA GRAMMAIRE DU SCRIPT

La syntaxe du langage Script a été inspirée de celle de langages existants. Elle devrait ainsi être assez familière pour tous nouveaux développeurs de scripts. Sa description au format BNF est donnée dans le paragraphe suivant.

5.10.1 Syntaxe BNF de la grammaire du script

```
Script ::= ScriptName  [Configuration] [Importation] [Constants] [Variables] Operations
Configuration ::= "Configuration" ParameterName {"," ParameterName} ";"
Importation ::= "ImportOp" RuleName {"," RuleName} ";"
Constants ::= "Constants" ConstName ":" ConstValue ";" {ConstName ":" ConstValue ";"}
ConstValue ::= Integer | String
Variables ::= "Variables" VarName ":" VarType ";" {VarName ":" VarType ";"}
VarType ::= "Value" | "String" | GrammarName
Operations ::= "Operations" {OpDefinition}
OpDefinition ::= OpName "::"  "{" [LocalVisibility] OpDef; {OpDef;} "}"
LocalVisibility ::= "LocalVisibility" VarName {"," VarName} ";"
OpDef ::= OpName | SaveOp |AssignOp | DelDs | DelNode | ExecOp | SetOp |
              CaseOp | WhileOp | PushOp | PopOp | GetAttributes | RelationLinks |
              UnLinkInstance | AddFirst | AddLast | AddAfter | ResetVariable | Error |
              Warning | Display | "BreakPoint;"
SaveOp ::= "SaveDs(" IdName "," StringDef ");"
AssignOp ::= IdName ":=" Value ";"
IdName ::= VarName {.FieldName}
Value ::= "Nil" | StringDef | Expression
StringDef ::= "LabelOf(" IdName ")" | StringDef {"&" StringDef} |
              "IncludeOp(" StringDef "," Integer ")" | "TypeOf(" IdName")" |
              "Date" | "ProjectPath" | "ToolsPath" | "FileSeparator" |
              DataStructure | "DelDelimiter(" IdName "," StringDef "," StringDef ")" |
              SubStringOf(" StringDef "," Expression "," Expression ")" | "ToString(" IdName ")" |
              "HierarchyName(" IdName "," IdName "," StringDef ")" |
              "ReplaceCharAt(" StringDef "," Expression "," StringDef ")"
DataStructure ::= IdName |
       "LoadDs(" StringDef "," Name "," Integer ")" |
       "CopyDs(" DataStructure ")" | "CopyNode(" DataStructure ")" |
       "CreateNode(" GrammarName "," NodeType ")" |
       "PeekAt(" Name "," Expression ")" | "Search(" Name "," StringDef "," StringDef ")" |
       "IncludeComponent(" IdName ")" | "GetLibraryPath(" IdName ")" |
       "FindModel(" IdName "," Integer ")" | "FindRelationalElement(" IdName ")" |
       "FindStartComponent(" IdName "," StringDef "," StringDef "," Integer ")" |
       "FindActivityDescription(" IdName ")"
Expression ::= BinaryExpression | Factor
BinaryExpression ::= Expression Op Expression
Op ::= "+" | "-" | "*" | "/"
factor ::= Integer | Real | "SizeOf(" Name ")" | "IsIn(" Name "," IdName ")" |
       "ValueOf(" IdName ")" | "GetTime(" Name ")" | "GetFreeMemory()" | "LengthOf(" IdName ")"
DelDs ::= "DelDs(" IdName ");"
DelNode ::= "DelNode(" IdName ");"
ExecOp ::= "Exec(" String ");"
SetOp ::= "ForEach(" VarName ":" OpDef; {OpDef;} ");"
While ::= "While(" Condition ":" OpDef; {OpDef;} ");"
CaseOp ::= "Case(" Condition ":" OpDef; {OpDef;} {"|" Condition ":" OpDef; {OpDef;}}
        ["| Else" : OpDef; {OpDef;}] ");"
Condition ::= IdName ComparisonOp |alue | Value | LogicalExpression
ComparisonOp ::= "=" | "#" | "<" | ">"
LogicalExpression ::= "(" Condition LogicalOp Condition ")"
LogicalOp ::= "AND" | "OR" | "XOR"
PushOp ::= "Push(" VarName ");"
PopOp ::= "Pop(" VarName ");"
GetAttributes ::= "GetAttributes(" Idname "," VarName ");"
RelationLinks ::= "RelationsLinks(" IdName ");"
UnLinkInstance ::= "UnLinkInstance(" IdName ");"
AddFirst ::= "AddFirst(" IdName "," DataStructure ");"
AddLast ::= "AddLast(" IdName "," DataStructure ");"
AddAfter ::= "AddAfter(" IdName "," DataStructure ");"
ResetVariable ::= "ResetVariable(" VarName ");"
Error ::= "Error(" StringDef ");"
Warning ::= "Warning(" StringDef ");
Display ::= "Display(" StringDef ");
```

5.10.2 Exemple de Script

L'exemple suivant est un script permettant de traiter partiellement le problème de génération présenté dans le paragraphe 5.7.1.

```
ExempleDeScript1
Variables VarMcseDs: McseGrammar; VarTemplateDs: VhdlGrammar;
      VarOutputDs: VhdlGrammar; Tmp1Ds: VhdlGrammar;
      Tmp2: VhdlGrammar;
Operations
      VHDLGenerate :: { Load; CreateEntity; CreateRelations; CreateBlocks; }
      Load :: { VarMcseDs :=LoadDs('McseDescription', McseGrammar,0);
                VarTemplateDs := LoadDs('template', VhdlGrammar,1);
              }
      CreateEntity :: { OutputDs:=CopyDs(TemplateDs);
```

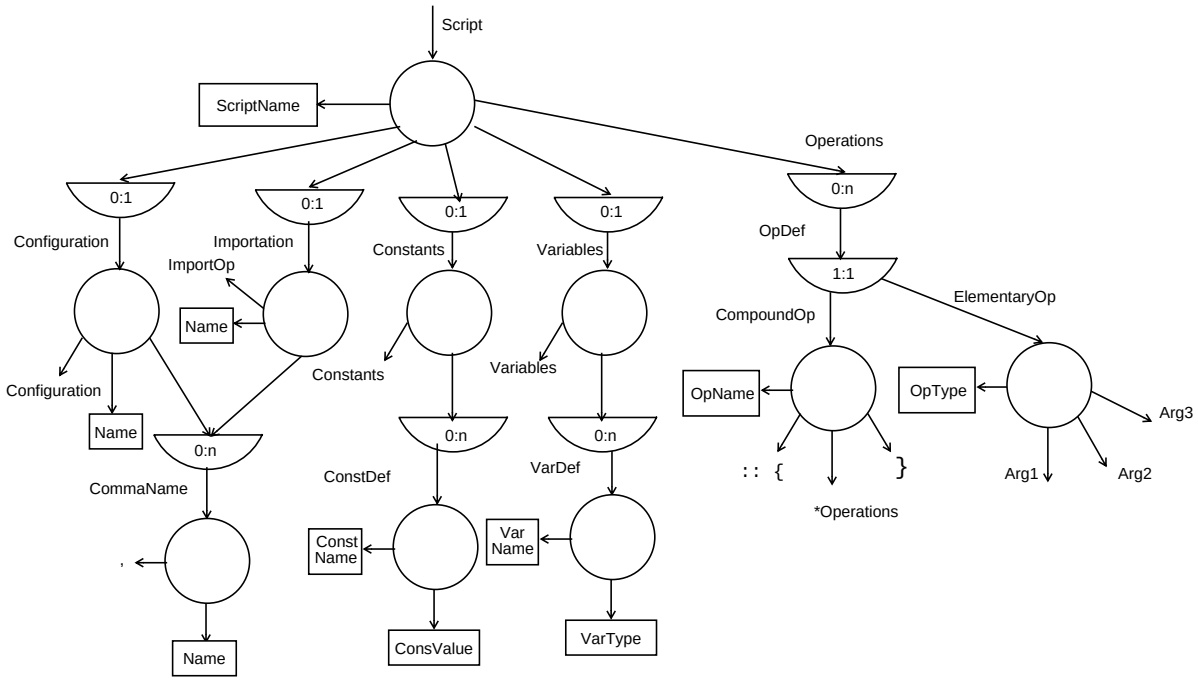
```
      OutputDs.Name:=McseDs.Name;
      }
CreateRelations :: { VarMcseDs:=McseDs.RelationDef;
      VarTemplateDs:=TemplateDs.RelationDef;
      VarOutputDs:=OutputDs.RelationDef;
      ForEach( VarMcseDs : AddLast(VarOutputDs,copyDs(VarTemplateDs));
      }
CreateBlocks ::  { VarMcseDS:=McseDs.StructureDef;
      VarTemplateDs:=TemplateDs.BlockDef;
      VarOutputDs:=OutputDs.BlockDef;
      ForEach( VarMcseDs : CreateBlock);
      }
CreateBlock :: { Tmp1Ds:=copyDs(VarTemplateDs);
      Case( VarMcseDs.Range = Nil : DeleteMultipleBlock);
      AddLast(VarOutputDs,TempDs);
      }
DeleteMultipleBlock :: { Tmp2:=Tmp1Ds.BlockDef;
      Tmp1Ds.BlockDef:=Tmp2.block;
      }
```

5.11 AUTOMATE D'EXECUTION

Le script sert à définir la fonctionnalité du générateur. Le comportement sera obtenu par interprétation de la structure de données ScriptDs qui est le résultat de son chargement par l'analyseur syntaxique approprié. Dans ce paragraphe nous analysons la structure de données ScriptDs pour déduire la spécification du moteur d'exécution.

5.11.1 Analyse de ScriptDs

La structure de données ScriptDs est associée à sa structure de grammaire ScriptGrammarDs comme c'est le cas pour toute structure. Compte-tenu de la grammaire, la structure de données souhaitée pour tout script est représentée sur la figure suivante.



-Figure 5.17- Structure de données pour la grammaire du Script.

Un script se décompose en deux parties: une partie déclaration et une partie description des opérations. Dans la partie déclaration, il y a:

- le nom du script qui est utilisé pour nommer la classe d'objet obtenue par la transcription du script en code JAVA,
- la clause de configuration qui permet de paramétrer l'exécution et la génération du script. Les paramètres de configuration sont déclarés et initialisés dans le fichier de configuration du méta-générateur.
- la clause d'importation qui indique les règles importées d'un autre script. Le nom des règles et le chemin du fichier dans lequel elles sont définies sont également déclarés dans le fichier de configuration du méta-générateur.
- la déclaration des constantes et des variables.

Chaque opération est soit une opération composite et donc définie par un ensemble d'opérations, soit une opération élémentaire. Chaque opération élémentaire est caractérisée par son type et l'ensemble de ses arguments. Par exemple, l'opération LoadDs possède les 3 arguments: FileName, GrammarName et mode. A noter qu'un argument peut se définir comme un noeud opération qui est alors le résultat de cette opération (donc une variable implicite). C'est par exemple le cas pour l'opération LoadDs qui est l'argument de droite de l'opération d'assignation.

L'ordre de déclaration des règles n'a pas d'importance: les règles sont exécutées suivant l'ordre d'appel de ces règles.

La récursivité apparaît lorsqu'une opération comprend une opération de niveau supérieur.

5.11.2 Principe d'exécution

La structure du script doit être construite pour avoir une efficacité d'exécution. On analyse dans ce paragraphe les transformations à apporter à la structure ci-dessus avant exécution pour permettre une exécution efficace.

-A- Objectif d'efficacité

L'exécution s'obtient par un programme se comportant comme un automate d'exécution assurant le parcours ordonné de la structure de données du script et exécutant au fur et à mesure les opérations élémentaires.

L'efficacité d'exécution résulte:

- d'une efficacité du parcours,
- d'une efficacité d'exécution des opérations.

Le parcours de la structure est efficace car l'implantation des ensembles d'opérations est faite sous la forme de listes chaînées.

L'efficacité des opérations résulte d'un accès efficace aux opérandes puis d'une exécution rapide. Le point important concerne l'accès aux éléments (variables, champs) comme opérandes qui doit être le plus direct possible. Les opérandes sont de 2 natures:

- une référence sur une donnée ou une structure. Il s'agit alors d'un accès direct.
- la désignation d'un champ dans un noeud d'une structure qui est faite par son nom symbolique. La difficulté consiste à connaître directement la position de ce champ dans le noeud. Cette position est dépendante du type de noeud, la structure du type de noeud étant définie dans la structure de données de la grammaire GrammarDS.

Le champ concerné est défini dans le script par son nom. Il s'agit donc d'assurer la conversion du nom du champ en un index dans le noeud. Cette conversion est faite avant exécution. L'index dans le champ se trouve par la grammaire de la structure du texte considéré. Comme le type d'une variable (type de la règle de grammaire) dépend de l'historique de l'exécution du script, cette conversion nécessite de faire une pseudo-exécution qui ne tient compte que du type des variables (et non leurs valeurs): c'est la phase dite d'optimisation d'un script.

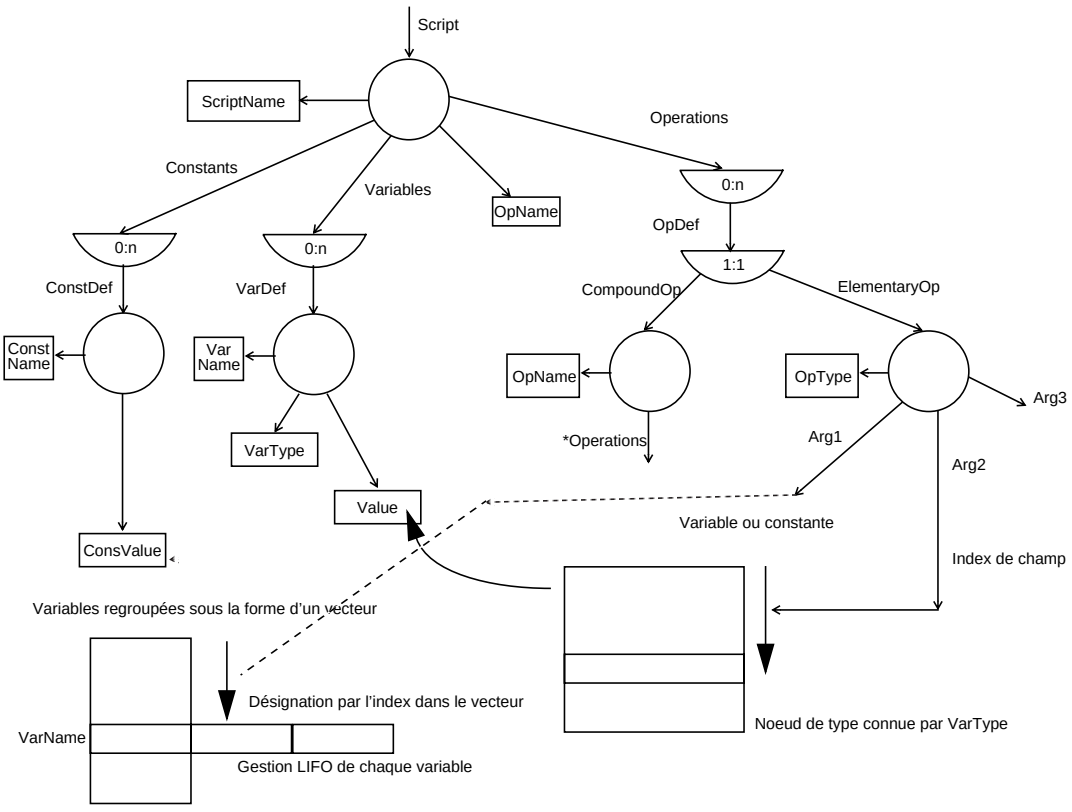
-B- Exploitation des variables

Les opérandes sont en fait les variables du script. L'efficacité s'obtient en exploitant une désignation directe de la variable. Ceci impose donc de remplacer le nom dans le script par un index dans la structure de données des variables (voir le paragraphe suivant).

-C- Principe d'exécution

La figure suivante montre la technique retenue pour obtenir une exécution efficace. Il s'agit de l'organisation de la structure de données du script pour qu'un programme conçu comme un automate soit efficace.

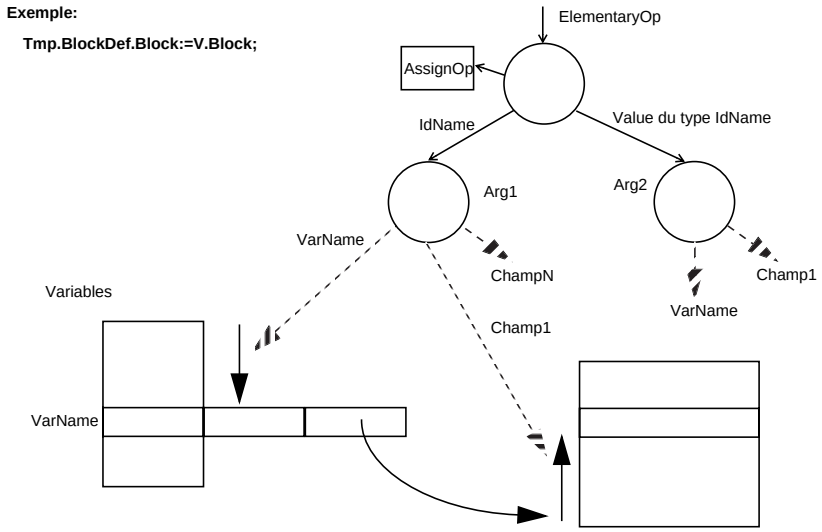
Les variables sont regroupées sous la forme d'un vecteur. Chaque variable se désigne alors par son index dans le vecteur. Chaque variable est également gérée comme une pile qui permet la récursivité et la gestion de listes.



-Figure 5.18- Structure de données pour une exécution du Script.

Le noeud ElementaryOp illustre le cas des 2 types d'opérandes: une variable ou constante, un champ d'un noeud. S'il s'agit d'une constante, l'opérande est une référence directe sur cette constante. S'il s'agit d'une variable, l'opérande est l'index de la variable dans le vecteur. S'il s'agit d'un champ, il s'agit de l'index dans le noeud.

La figure 5.19 illustre le cas de l'opération d'assignation.



-Figure 5.19- Structure de données pour l'opération d'assignation.

L'assignation est définie par **IdName := Value**. IdName et Value sont ici représentés comme l'emploi d'un champ d'une variable. Chaque opérande est soit une désignation directe (cas d'une variable sans champ ou d'une constante), soit un champ d'un noeud désigné par une succession de champs (indirections multiples comme pour Arg1 de la figure 5.19). Les champs sont traités dans l'ordre pour faire toutes les indirections déclarées.

5.11.3 Implantation de l'automate

L'automate d'exécution se déduit assez directement de la technique décrite dans le paragraphe précédent. Il s'agit d'implanter:

- le programme de chargement du script de manière à aboutir à la structure de données (phase de vérification syntaxique et chargement),
- le programme de parcours de la structure du script pour son optimisation avant exécution (phase d'optimisation et de contrôles sémantiques),
- le programme de parcours et d'exécution de toutes les opérations élémentaires (phase d'exécution).

5.11.4 Traduction d'un script en Java

Au lieu d'interpréter le Script en parcourant sa structure de données image, il est également possible de traduire le Script en un programme Java. On obtient alors un programme de génération indépendant du méta-générateur pour son exécution et plus rapide que le fonctionnement en mode interprété. En effet, que le script soit en mode interprété ou transcrit en code Java, les mêmes méthodes sont utilisées pour l'exécution des opérations élémentaires. La transcription se limite donc simplement à convertir une structure de données en un enchaînement d'instructions. Cette conversion repose sur la dualité qui existe entre opérateurs de composition d'une structure de données et structures de contrôle d'un programme. Ainsi un record de la structure de données du script est traduit par une séquence d'instructions, l'alternative par la sélection et l'ensemble par l'itération.

Comme un script peut contenir un ou plusieurs appels récursifs directs ou indirects, nous sommes obligés de définir une méthode pour chaque règle. Cette solution donne également la possibilité de surcharger chaque méthode ou règle et donc de piloter le script de l'extérieur.

5.12 LA TECHNOLOGIE JAVA

Le changement de philosophie de conception des outils MCSE a offert l'opportunité d'utiliser pour l'implantation des outils, le dernier né des langages orienté objet, c'est à dire le langage Java. Jamais encore un nouveau langage de programmation n'a reçu autant d'attention et est devenu très populaire aussi rapidement. Les raisons de ce succès sont principalement liées à la sélection minutieuse des constructions du langage et son ouverture sur le web.

Pour le développement d'une plate-forme d'outils, le langage Java offre un certain nombre d'avantages non négligeables par rapport à une implantation en C++ ou l'utilisation d'un méta-outil. Il est en effet [JAWORSKI-97]:

- *objet* (et non *orienté* objet). Contrairement à C++ qui permet de programmer sans utiliser nécessairement une approche et des concepts objet, Java n'autorise de travailler qu'avec des classes d'objets et supporte tous les concepts de la programmation orientée objet: hiérarchie de classes, héritage, encapsulation (package), polymorphisme (méthode de même nom), réutilisation (Java Beans), etc.

- *interprété* et *portable*. Un programme Java tourne sur toute plate-forme disposant de la machine virtuelle Java. Si le logiciel doit satisfaire des contraintes de temps sévères, un programme Java peut aussi être compilé en code machine à la volée (compilateur Just-In- Time) ou définitivement (compilateur natif). Le mode interprété permet d'instancier des classes d'objets dont on ne connaît pas le nom (nom saisi par l'utilisateur par exemple) lors de la compilation en byte-code. Le pseudo-code (byte-code) utilisé par la machine virtuelle a aussi la particularité d'être facilement retransformable en code Java ("reverse-coding"). Un déassembleur est livré en standard dans le JDK.

- *simple* et *familier*. Pour l'écriture, la syntaxe est proche de celle du C++ avec quelques concepts plus formalisés tel que l'absence de pointeurs, la persistance d'objets (sérialisation), etc. La similitude syntaxique entre Java et C++ facilite la translation d'un programme C++ en Java.

- *multi-thread*. L'ordonnancement des threads est préemptif et à priorité fixe. La protection des données partagées entre threads est assurée en standard (synchronisation). Java offre ici une propriété très intéressante pour la simulation du modèle de performance de MCSE. En plus de la programmation multi-thread, d'autres constructions ont aussi été inspirées d'ADA: les concepts de package et d'exception.

- *fiable*. Les constructions qui sont souvent source de problèmes en C/C++ telles que par exemple les pointeurs génériques (*void) et la conversion implicite de type ne sont pas autorisés en Java. Le garbage collector s'occupe de libérer automatiquement la mémoire désallouée. Le mécanisme d'exception permet de retrouver plus facilement la source d'une erreur survenue lors de l'exécution du programme.

- *riche*. La richesse des classes prédéfinies facilite en autre la génération d'interfaces utilisateurs conviviales (menus, boutons, boite de dialogue, ScrollBar, animations, gestion des événements, etc),

- *facile à documenter* et à *maintenir*. Si le code source respecte un format donné, l'utilitaire javadoc génère un document HTML contenant la hiérarchie des classes, une description détaillée de chaque classe (attributs, constructeur, méthodes), un index, etc.
- *adapté à l'implantation de systèmes distribués sur un réseau local ou distant*. Java permet de décrire facilement des architectures client/serveur et autorise l'utilisation de différents protocoles de communication (TCP/IP, UDP, IPX, HTTP, etc). Comme il autorise également le lancement de différents process (programme C, simulateur spécifique), Java est donc très intéressant pour faire de la co-simulation hétérogène.
- *sécurisé*. Dédié à programmer des applications sur Internet, Java a été conçu avec différentes couches de sécurité. Le package Security Manager contient des méthodes pour contrôler les accès des utilisateurs et encrypter les données.
- *ouvert vers le web*. Un programme Java décrit sous forme d'applet est téléchargeable par le web. Si de plus, l'application est implantée sur le modèle client-serveur et utilise le protocole RMI (Remote Method Invocation), alors elle est utilisable par tous mais sa propriété intellectuelle (code source) est préservée.
- *plein de vitalité*. La technologie Java est en plein essor (prolifération de livres, outils et classes). Le système d'exploitation JavaOs qui est multi-tâches et orienté réseaux de communication devrait connaître un franc succès avec l'apparition des "Network Computers" à bon marché et la banalisation d'Internet.

5.13 REALISATION DE L'OUTIL META-GEN

Ce paragraphe décrit la solution retenue pour l'implantation de l'outil MetaGen. Pour représenter les classes d'objets implantées, nous utilisons le modèle statique de la méthode OMT-UML. Pour le lecteur non familiarisé avec le dialecte de cette notation pour la conception orientée objet, les concepts du modèle statique de la méthode unifiée sont présentés dans [RUMBAUGH-91] [RUMBAUGH-95].

5.13.1 Les classes d'objets

Toutes classes d'objets sont regroupées en deux catégories. La première catégorie concerne l'interface utilisateur de l'outil. La classe StartScript sert à lancer l'outil sous forme d'application locale ou d'applet. La classe GenEdit implante l'interface utilisateur qui sera détaillée dans le paragraphe suivant. La seconde catégorie regroupent les classes qui constituent le coeur du méta-générateur.

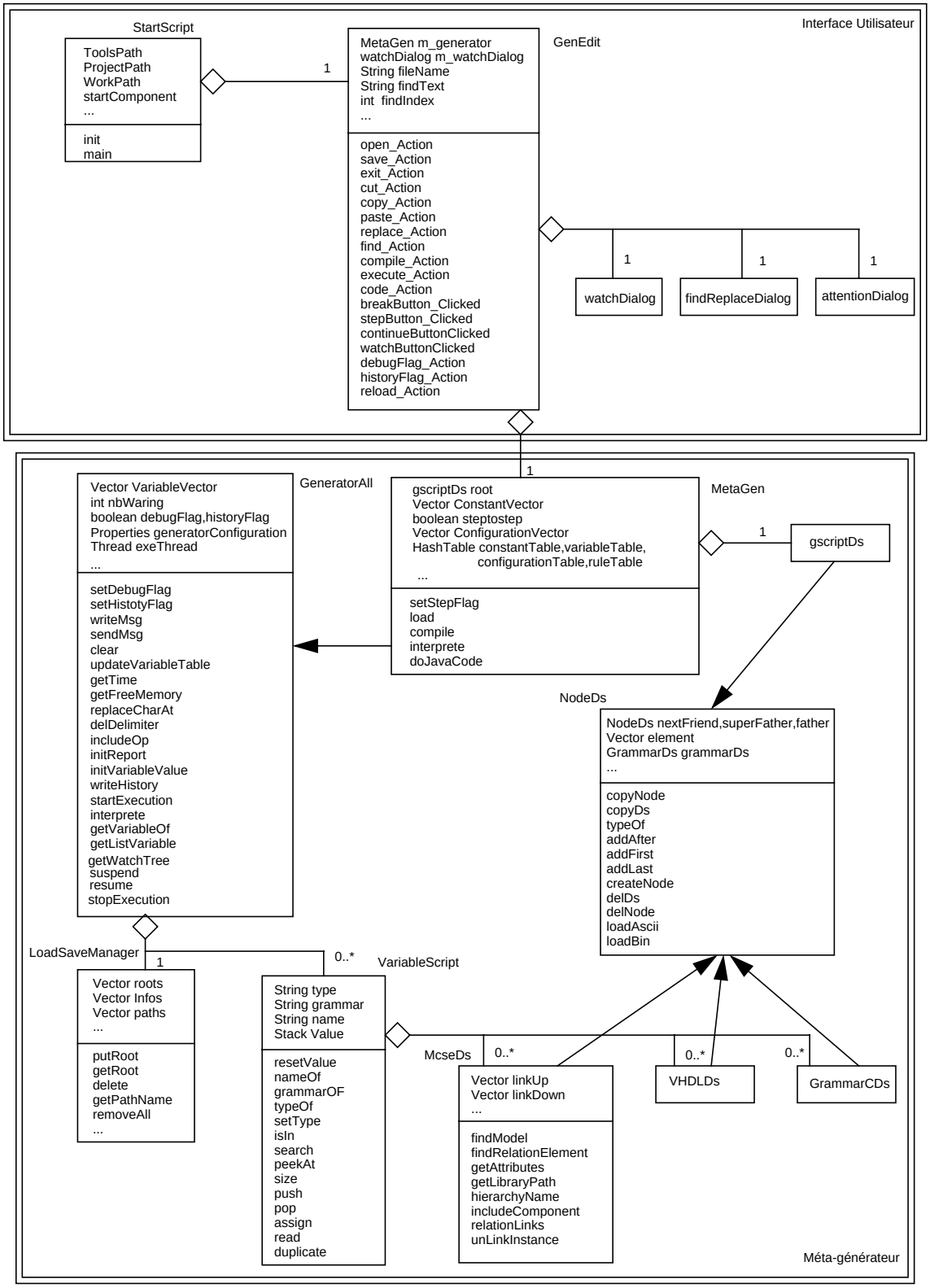
L'ensemble des instructions du script est réparti sur trois classes de la seconde catégorie:

- la classe VariableScript qui implante les variables du script,
- la classe NodeDs qui implante un noeud de structure de données,
- la classe GeneratorAll qui contient les méthodes qui ne sont rattachées ni aux variables du script ni aux noeuds d'une structure de données.

-A- La classe VariableScript

Chaque variable du script possède une référence sur la grammaire et la règle de grammaire concernées. Les valeurs d'une variable sont gérées par une pile dont la tête est la valeur courante de la variable. Lors d'une assignation, il y a systématiquement vérification de la cohérence entre le type de la variable et la valeur qui lui est assignée.

La hiérarchie des classes est représentée sur la figure 5.20.



-Figure 5.20- Hiérarchie des classes d'objets de l'outil MetaGen.

-B- Les classes des noeuds de structure

Les classes des noeuds de structure héritent toutes de la classe NodeDs. Parmi ces classes, la classe McseDs se distingue par le fait qu'elle possède des attributs (linkUp, linkDown) et des méthodes spécifiques à l'exploitation du modèle MCSE.

-C- La classe LoadSaveManager

Cette classe sert à mémoriser les structures de données chargées (fichiers templates) lors de l'exécution d'un script. Lors d'une nouvelle exécution, les classes mémorisées qui n'auront pas été volontairement effacées par l'utilisateur ne seront pas rechargées d'où un gain de temps non négligeable. Ce mécanisme est également valable pour les générateurs engendrés par le méta-générateur pour lesquels plusieurs générations de code successives seront nécessaires pour parcourir le domaine des solutions possibles.

-D- La classe GeneratorAll

La classe GeneratorAll est une classe abstraite dont héritent la classe MetaGen et les classes obtenues par transcription d'un script en code Java. Elle contient comme principaux attributs le vecteur des variables du script et un "Thread" qui sert pour l'interprétation ou l'exécution du script. L'utilisation de ce thread permet de suspendre l'exécution du script à tout moment. On peut remarquer qu'il n'y a pas de table de constantes. En effet, lors de la génération de code Java, les constantes sont remplacées par leurs valeurs. Les erreurs de syntaxe, compilation ou exécution sont gérées par le mécanisme d'exception de Java. En cas d'erreur, un événement contenant un message d'erreur est envoyé vers l'interface graphique. On utilise également un événement mais sans message pour signaler la fin d'exécution d'un script.

-E- La classe MetaGen

En plus de ceux hérités de la classe GeneratorAll, les attributs importants de la classe MetaGen sont:

- la racine de la structure de données du script (root),
- la table des constantes (constantVector),
- la table des paramètres de configuration du script (configurationVector),

Les principales méthodes de cette classe sont chargées de:

- la vérification syntaxique et le chargement de la structure de données du script (load),
- l'optimisation de la structure de données avant son interprétation (compile),
- l'exécution ou plutôt interprétation (interprete),
- la transcription en code java (doJavaCode).

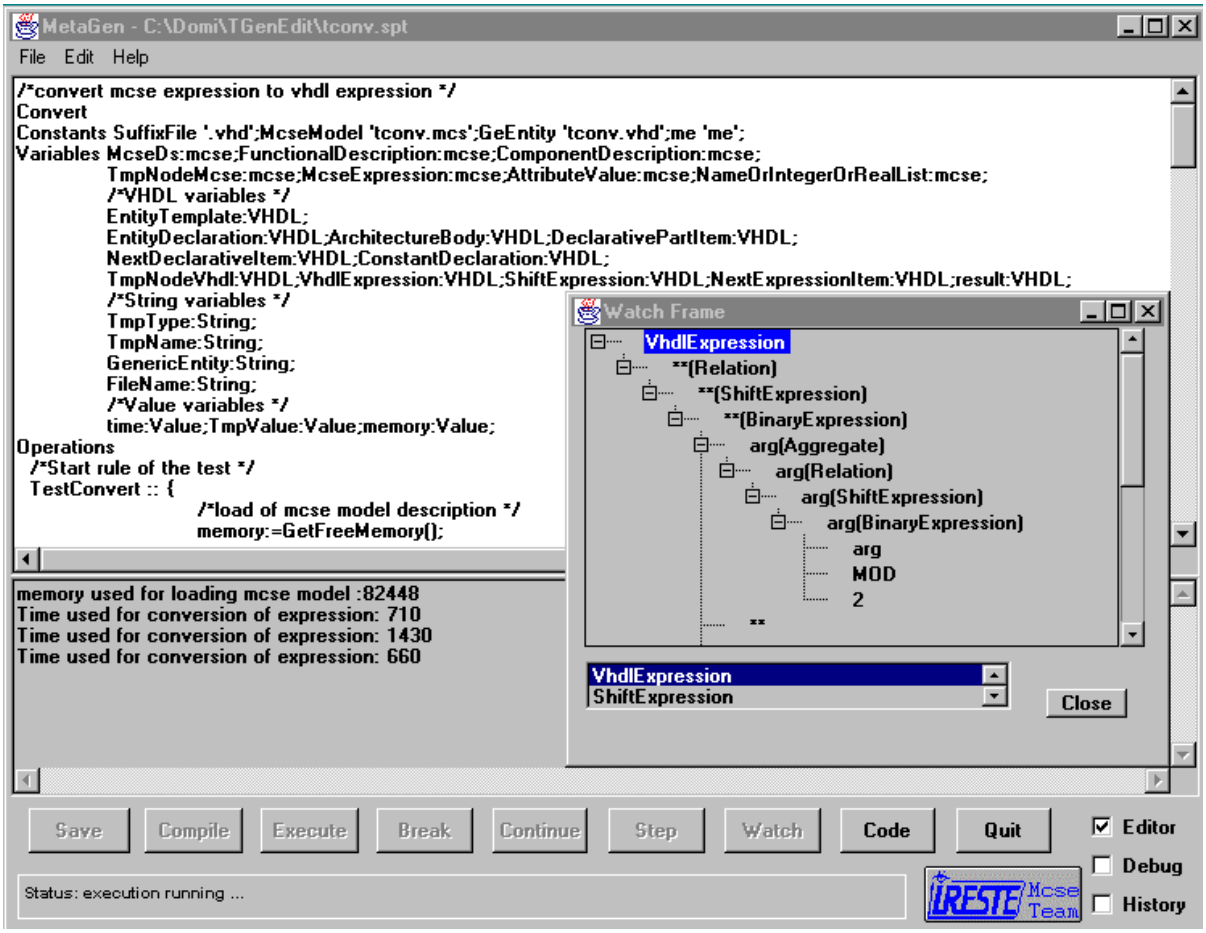
Durant la phase de chargement et vérification syntaxique, on met à jour des "hashtables" concernant les paramètres de configuration, les constantes, les variables et les règles du script. Ces hashtables sont ensuite utilisées durant la phase d'optimisation de la structure de données qui consiste à remplacer une variable par son index dans la table des variables et un nom d'un champ de record pour son index. Cette phase d'optimisation est une pseudo-interprétation du script qui ne tient compte que du type des variables. La phase d'exécution est un parcours ordonné de la structure du script et une exécution des opérations élémentaires.

5.13.2 L'interface utilisateur de l'outil MetaGen

L'interface de l'outil se compose:

- d'une barre de menu principal,
- d'une zone d'édition permettant de saisir le script et où l'on retrouve les commandes classiques d'un éditeur textuel (open, save, copy, paste, cut, find, replace, etc.),
- d'une zone d'affichage de messages,
- d'une zone de boutons de commande,
- d'une ligne de statut,
- d'une zone d'affichage du contenu des variables sous forme d'arbre.

La figure suivante montre l'interface utilisateur du méta-générateur.



-Figure 5.21- Interface utilisateur de l'outil MetaGen.

La mise au point d'un script se fait d'une manière interactive. Le concepteur assure d'abord son édition et sa sauvegarde. Le script est ensuite compilé. Durant la phase de compilation, l'outil fait une vérification syntaxique et charge la structure de données du script. Une fois chargé, des contrôles sémantiques et des optimisations sont faites (Compile) sur cette structure. Lorsque la compilation s'est passée sans erreur, le concepteur peut lancer l'interprétation de son script (Execute). Pour la mise au point, il a la possibilité de placer des points d'arrêt dans le code (instruction BreakPoint) ou de suspendre l'interprétation à tout moment (Break). Il peut

alors visualiser le contenu des variables (Watch) et relancer l'interprétation en continu (Continue) ou en pas à pas (Step). Des indicateurs permettent d'avoir une trace sur l'écran (Debug) ou dans un fichier (History) de la compilation ou de l'exécution. Enfin, il peut transcrire le script en un programme Java dont l'exécution sera beaucoup plus rapide (Code).

5.14 CONCLUSION

Ce chapitre a décrit la stratégie adoptée pour disposer d'un outil général permettant une conception aisée et efficace de tout générateur de code.

Le travail sur la génération de code présenté dans ce chapitre a aussi amené l'équipe MCSE à revoir sa stratégie de développement des outils comme support pour la méthodologie MCSE. Dans la nouvelle plate-forme en cours de développement "MCSE ToolBox", le couplage entre outils est basé sur un échange par fichiers. Ce fichier d'échange doit être transformé en une structure de données qui est ensuite manipulée par l'outil concerné. Or, toute structure de données peut être engendrée d'une manière automatique par un analyseur syntaxique enrichi des règles de production de la structure de données. Cet analyseur syntaxique est obtenu par un générateur d'analyseur syntaxique à partir de la spécification de la grammaire (syntaxe d'entrée du texte source) et des règles de production.

Pour avoir un principe de création de la structure de données indépendant de la grammaire, nous avons défini pour la spécification des règles de production un modèle de modèle de structure de données ou méta-structure basé sur la composition de quatre éléments de base qui sont la séquence d'éléments, l'optionnel, l'alternative et la liste ou ensemble.

Cette solution offre au moins deux avantages. Tout d'abord, elle est *évolutive*. En effet, pour faire évoluer la structure de données, il suffit de modifier la grammaire et de regénérer l'analyseur syntaxique associé. Elle est aussi *générique*. Comme tous les outils de la plate-forme MCSE ToolBox, un générateur de code repose sur une architecture générique comportant une fonction de chargement, de modification et de sauvegarde de structure de données. La fonction de chargement charge sous la forme d'une structure interne le texte source. La fonction de manipulation parcourt la structure de données du modèle source et crée à partir des informations recueillies une structure de données de sortie dont la grammaire est celle du langage cible. La fonction de sauvegarde qui correspond à un parcours ordonné de la structure de données créée et à l'exploitation de la grammaire pour le formatage du texte de sortie génère le fichier texte final.

Pour faciliter les manipulations des structures de données à effectuer pour mener à bien une transcription texte à texte, nous avons également utilisé deux autres concepts: *le concept de template et le concept de script.* Un template est un fichier écrit dans le langage cible souhaité et contenant toutes les constructions nécessaires pour la translation texte à texte. Il s'agit d'un modèle générique du résultat attendu. Ainsi, la production de la structure de sortie consiste à parcourir la structure de données du modèle source, copier des parties de la structure du template puis les mettre à jour. Le script est un langage que nous avons défini pour décrire de manière concise les manipulations de structures de données à effectuer: chargement d'une structure à partir d'un texte, sauvegarde d'une structure sous forme textuelle, copie d'une structure complète ou d'un noeud de structure, destruction d'une structure complète ou d'un noeud de structure, mise à jour d'un champ d'un noeud et ajout d'un élément dans un ensemble.

Un générateur est alors le résultat de la définition des grammaires des langages source et cible afin d'obtenir les analyseurs syntaxiques associés, la définition d'un ou plusieurs fichiers

template et l'écriture d'un script. Ce script sert de point d'entrée au générateur de générateurs de code ou méta-générateur nommé MetaGen qui permet d'interpréter le script ou de le transcrire en code JAVA. En mode interprété, le script est chargé et la génération se fait par parcours de la structure de données du script. Bien que relativement lent, ce mode est très utile pour:

- la mise au point d'un script: vérification incrémentale, point d'arrêt, exécution pas à pas, visualisation du contenu des variables, etc,
- la modification de la structure de données du script et donc du résultat de la génération par un programme extérieur (éditeur orienté par la syntaxe par exemple): le script peut ainsi être modifié de manière interactive par l'utilisateur en fonction de choix d'implantation. Cette propriété est intéressante pour le développement d'un générateur d'interface matériel/logiciel qui doit laisser le choix au concepteur entre plusieurs solutions d'implantation possibles.

Lorsque le script est transcrit en code Java, l'exécution est plus rapide, sachant qu'en contre-partie il ne peut plus être modifié par un programme extérieur que par une surcharge de méthodes.

L'intérêt du script est que toute la définition de la génération se fait par des règles de production fournies au générateur pour son paramétrage. Ainsi une modification dans le script est immédiatement reportée dans le générateur concerné. Un autre aspect intéressant est la possibilité de construire un script (c'est-à-dire sa structure de données) par un autre programme. Il peut s'agir d'un programme interactif permettant au concepteur de spécifier ce qu'il désigne comme solution en sortie. Ceci peut par exemple être le cas pour le choix d'une solution pour l'implantation d'une variable partagée ou d'une fonction. Il peut aussi s'agir de l'emploi d'un autre script qui par son exécution engendre le script de production du code!. La modification de la structure de données d'un script peut aussi se faire en interactif par une édition orientée par la syntaxe car toutes les règles de grammaire sont connues par l'outil. L'éditeur exploite alors la méta-structure et la table des symboles pour proposer à l'utilisateur les solutions possibles. La sélection faite respecte ainsi obligatoirement la grammaire.

Actuellement, les concepts retenus et le méta-générateur développé ont été utilisés pour générer trois générateurs de code:

- un générateur de code VHDL comportemental pour la vérification fonctionnelle et l'évaluation des performances qui est présenté dans le chapitre 6. Ce générateur a demandé un temps de développement plus long (6 mois) que les deux autres (3 mois) car d'une part les règles de transcription sont plus complexes et d'autre part il n'a pas pu bénéficier contrairement aux deux autres de l'expérience acquise sur l'implantation d'un générateur équivalent mais limité au modèle fonctionnel de MCSE. Ce générateur est également celui qui est le plus avancé, debuggé et testé.
- un générateur de code VHDL au niveau RTL pour la synthèse matérielle
- un générateur de code C/Noyau Temps-réel pour la synthèse logicielle pour lequel la possibilité de configuration permet de changer facilement de noyau temps-réel cible.

Un générateur de code C++ pour l'évaluation des performances est également en cours de développement. Pour tester efficacement l'intérêt de notre solution, ce générateur est développé par des personnes nouvelles au projet (utilisation de candide pour ne pas biaiser les résultats de cette expérience).