6

Le Générateur VHDL

Ce chapitre décrit l'implantation du générateur de code VHDL. Rappelons que l'objectif à atteindre consiste à obtenir un outil capable de transcrire le modèle de performance de la méthodologie MCSE en une description VHDL comportementale. Le modèle de performance qui est l'entrée du générateur a été présenté dans le chapitre 3. Les règles de transcription qu'il s'agit d'implanter dans le générateur de code ont été détaillées dans le chapitre 4. Pour avoir un principe de génération commun à tous les générateurs de code et indépendant du langage cible, nous avons développé un méta-générateur qui a fait l'objet du chapitre 5.

Pour développer un générateur de code avec le méta-générateur nommé MetaGen, nous préconisons de suivre une démarche en trois phases:

- Entrer la grammaire du langage cible correctement afin d'obtenir un analyseur syntaxique. Généralement on n'utilisera pas toutes les constructions du langage cible et on peut évidemment se contenter de définir la syntaxe de celles qui nous seront utiles. Le générateur d'analyseur syntaxique fournit un rapport de la syntaxe saisie très utile lors du développement du script (recherche de la syntaxe d'une règle).

- Définir un ou plusieurs fichiers template. Ce fichier doit contenir toutes les constructions du langage utiles pour la génération de code. Une fois analysé, ce fichier est stocké sous la forme d'un arbre à partir duquel on dupliquera des branches pour générer l'arbre du programme de sortie. Comme il est plus facile de détruire une partie d'arbre que de construire des feuilles une par une, il est conseillé de définir les constructions utilisées sous leur forme complète. Cette définition est importante car elle influe directement sur la rapidité du développement du générateur et sur l'efficacité et la qualité de son implantation.

- Ecrire le script pour implanter les opérations nécessaires pour réaliser la transcription: d'un point de vue macroscopique, une grande partie du temps sert à dupliquer des branches et à mettre à jour des noms.

Nous commençons donc ce chapitre en présentant la spécification d'entrée de la grammaire VHDL qui sert à produire un analyseur syntaxique pour le langage VHDL. Les templates utilisés pour faciliter la génération du package et de l'entité VHDL sont ensuite décrits. La présentation du Script définissant les manipulations à effectuer sur les structures de données du modèle source et des templates est décomposée en deux parties: une partie Analyse du modèle MCSE et une partie Génération. La description de la partie Analyse détaille le parcours et l'analyse de la composante structurelle puis de la composante comportementale du modèle de performance de MCSE. La partie Génération regroupe l'ensemble des règles de production de la structure de données de sortie. Ces règles appelées dans la partie Analyse utilisent les informations fournies par la partie Analyse et les structures de données des templates pour générer la structure de données de sortie qui sera sauvegardée au final sous forme textuelle. Le générateur obtenu a été testé avec les exemples décrits dans le chapitre 7. Un tableau comparatif avec les autres générateurs de code prototypés pour la plate-forme d'outils "MCSE ToolBox" est fourni. Une fois transcrit en code Java par le méta-générateur, le script a été associé à une interface utilisateur et a été intégré au "framework" de la plate-forme "MCSE ToolBox". Une présentation succincte de l'interface utilisateur du générateur est également donnée. Au terme de ce développement, diverses améliorations sont envisageables à court terme. Avant de conclure, nous présentons donc une liste non exhaustive d'améliorations possibles.

6.1 SPECIFICATION DE LA GRAMMAIRE VHDL

Le générateur d'analyseur syntaxique sur lequel est basé la plate-forme des outils MCSE en cours de développement se nomme JAVACUP. C'est un générateur d'analyseur syntaxique du type LALR(1) comparable à YACC qui génère du code JAVA et qui a été développé par Scott E. Hudson [HUDSON-96].

Dans les paragraphes suivants, nous allons décrire le fonctionnement d'un analyseur syntaxique obtenu avec JAVACUP. La compréhension de ce fonctionnement est indispensable pour écrire les règles grammaticales d'une manière correcte et efficace.

6.1.1 Principe de fonctionnement de l'analyseur syntaxique utilisé

L'analyseur syntaxique généré par JAVACUP est en fait une machine d'états pour laquelle chaque état correspond à une position dans une ou plusieurs règles partiellement analysées. Il est couplé à un analyseur lexical dont le rôle est de découper le texte source en unités significatives que l'on nommera par la suite token. A chaque fois que l'analyseur syntaxique lit un token provenant de l'analyseur lexical et qui ne complète pas une règle, il l'empile et va dans l'état correspondant au nouveau token lu: cette action s'appelle "a **shift**".

Quand il a obtenu (empilé) tous les tokens nécessaires pour compléter une règle, l'analyseur syntaxique les désempile, empile le symbole correspondant à la règle réduite et va dans l'état correspondant à la nouvelle situation: cette action s'appelle "a **reduce**" (le nombre d'objets dans la pile a diminué). Si une action est associée à la règle réduite, elle est alors exécutée.

Dans notre cas, à chaque réduction de règle, il y a mise à jour de l'arbre du programme qui contiendra au final le code analysé.

6.1.2 Les conflits

JAVACUP peut échouer dans la translation de la spécification d'une grammaire si celle-ci est ambiguë ou contient des conflits. Si la grammaire est réellement ambiguë, il y a deux possibilités d'analyse pour une chaîne d'entrée et JAVACUP ne peut traiter ce cas. Si la grammaire est non ambiguë, il peut y avoir des conflits liés au fait que l'analyseur syntaxique aurait besoin de tenir compte de plus de un token d'entrée ("**limited lookahead**") pour décider laquelle des possibilités utiliser.

Il y a deux types de conflits qui peuvent apparaître quand JAVACUP essaie de créer un analyseur syntaxique: les "**shift/reduce**" conflits et les "**reduce/reduce**" conflits.

> Un "shift/reduce" conflit apparaît quand pour un token d'entrée il y a deux possibilités d'interprétation et que l'une des possibilités complète une règle (reduce option) et pas l'autre (shift option).
>
> Un "reduce/reduce" conflit apparaît quand un même token d'entrée peut compléter deux règles différentes.

Nous allons présenter dans les paragraphes suivants les principaux types de problèmes rencontrés lors de la génération d'un analyseur syntaxique pour le langage VHDL'93.

-A- Expressions arithmétiques

Considérons le cas des expressions arithmétiques qui sont décrites directement ou indirectement par une règle récursive telle que par exemple:

```
expr ::= expr + expr |
          expr - expr |
          expr * expr |
          expr / expr |
          factor
```

Avec la chaîne d'entrée suivante expr + expr + expr, il y aura un shitft/reduce conflit sur la deuxième occurrence du signe +. Contrairement à Yacc qui possède des opérateurs de priorité et d'associativité (%left, %right, %nonassoc), JAVACUP ne permet pas d'éviter ce type de conflit. On obtient alors un shift/reduce conflit où le shift l'importe: ce qui correspond à une associativité à droite.

-B- Problèmes liés au "limited lookahead"

Considérons le cas de la déclaration d'un sous-type en VHDL décrite par la syntaxe BNF suivante:

```
subtype_declaration::=subtype identifier is subtype_indication
subtype_indication::=[resolution_function_name] type_mark [constraint]
avec pour simplifier
resolution_function_name::=identifier
type_mark::=identifier
```

JAVACUP ne peut pas savoir si l'identificateur qui suit le mot clé "is" correspond au nom de la fonction de résolution ou au nom du type. Avec un analyseur syntaxique fonctionnant avec un "lookahead" d'au moins deux tokens, il n'y aurait pas de conflits dans ce cas.

Pour résoudre ce type de conflit, on n'a pas d'autres choix que de mettre à plat la description comme indiqué ci-dessous:

```
subtype_declaration::=
 subtype_declaration_with_resolution_function_name |
 subtype_declaration_without_resolution_function_name
```

```
subtype_declaration_with_resolution_function_name::=
 subtype identifier is resolution_function_name type_mark [constraint]
subtype_declaration_without_resolution_function_name::=
 subtype identifier is type_mark [constraint]
```

-C- Problèmes liés au recouvrement d'alternatives

Considérons les règles syntaxiques suivantes:

```
name::= identifier |
   operator_symbol |
   selected_name |
   indexed_name |
   slice_name |
   attribute_name
operator_symbol::=string_literal
primary::=name |
   literal |
   aggregate |
   function_call |
   qualified_expression |
   type_conversion |
   allocator |
   (expression)
literal::= numeric_literal |
   enumeration_literal |
   string_literal |
   bit_string_literal |
   null
```

Au niveau de la règle primary, il y a un recouvrement de la règle string_literal au travers des règles name et literal et par conséquent il y aura un reduce/reduce conflit entre les règles name et literal.

Pour éviter ce type de problème, il faut s'arranger pour que tous les éléments d'un choix multiple soient disjoints:

```
primary::=name_without_literal_string |
   literal_without_literal_string |
   literal_string |
   aggregate |
   function_call |
   qualified_expression |
   type_conversion |
   allocator |
   (expression)
```

Il n'est pas toujours immédiat de retrouver la source d'une erreur. La spécification d'une grammaire est une opération longue pour laquelle une maîtrise totale de la syntaxe est indispensable. Enfin, la représentation de la grammaire sous la forme du modèle graphique de méta-structure est très utile pour son optimisation.

6.1.3 L'analyseur syntaxique obtenu

La syntaxe du langage VHDL décrite dans le LRM (Langage Reference Manual) n'est pas du type LR(1). Elle est même ambiguë hors contexte. Pour éliminer les shift/reduce et reduce/reduce conflits de notre grammaire VHDL, nous avons fait quelques petites transgressions des règles syntaxiques décrites dans le LRM. Notre analyseur syntaxique n'est donc pas strictement "full VHDL'93".

Les entrées acceptées par l'analyseur syntaxique qui ne sont pas des constructions VHDL correctes peuvent être rejetées par des contrôles sémantiques. Comme notre but n'est pas de générer un éditeur orienté par la syntaxe, nous n'aborderons pas le sujet des contrôles sémantiques. D'ailleurs, l'élimination des conflits (mise à plat de la grammaire) se paie malheureusement aussi sur l'aspect et la convivialité de l'éditeur syntaxique.

Le code de la grammaire VHDL comporte 3084 lignes alors que celui de la grammaire du modèle MCSE ne nécessite que 1750 lignes.

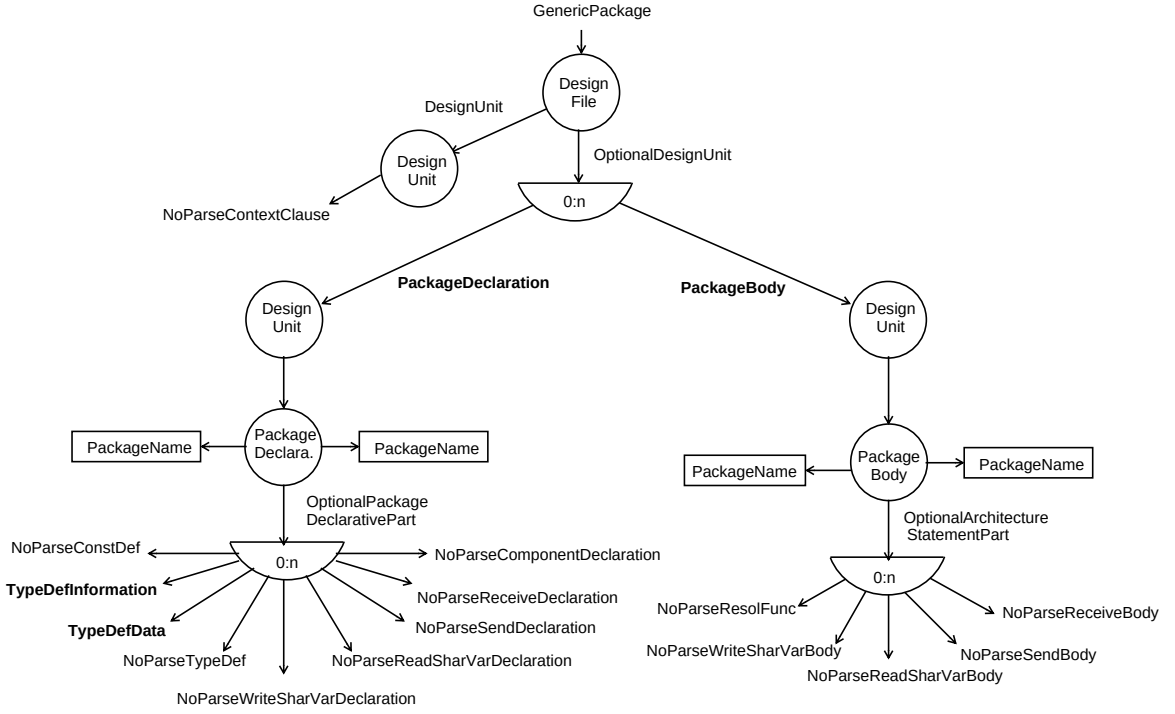
6.2 LES TEMPLATES UTILISES

Le fichier template contient toutes les constructions du langage VHDL nécessaires pour le générateur. Une fois analysé, ce fichier est stocké sous la forme d'un arbre à partir duquel on dupliquera des branches pour générer l'arbre du programme de sortie.

Pour la génération, nous utilisons deux templates: un pour la génération de l'entité VHDL représentant le système étudié et l'autre pour le package associé à cette entité.

La génération d'un package est due uniquement aux limitations sur la généricité des types du langage VHDL. Pour éviter de surcharger les primitives d'accès aux éléments de relations du modèle MCSE en fonction du type de la donnée transmise, on regroupe les types et les attributs concernant les ports de communication dans le record TypeDefInformation. De même, les types et attributs concernant les variables partagées sont regroupés dans le record TypeDefData.

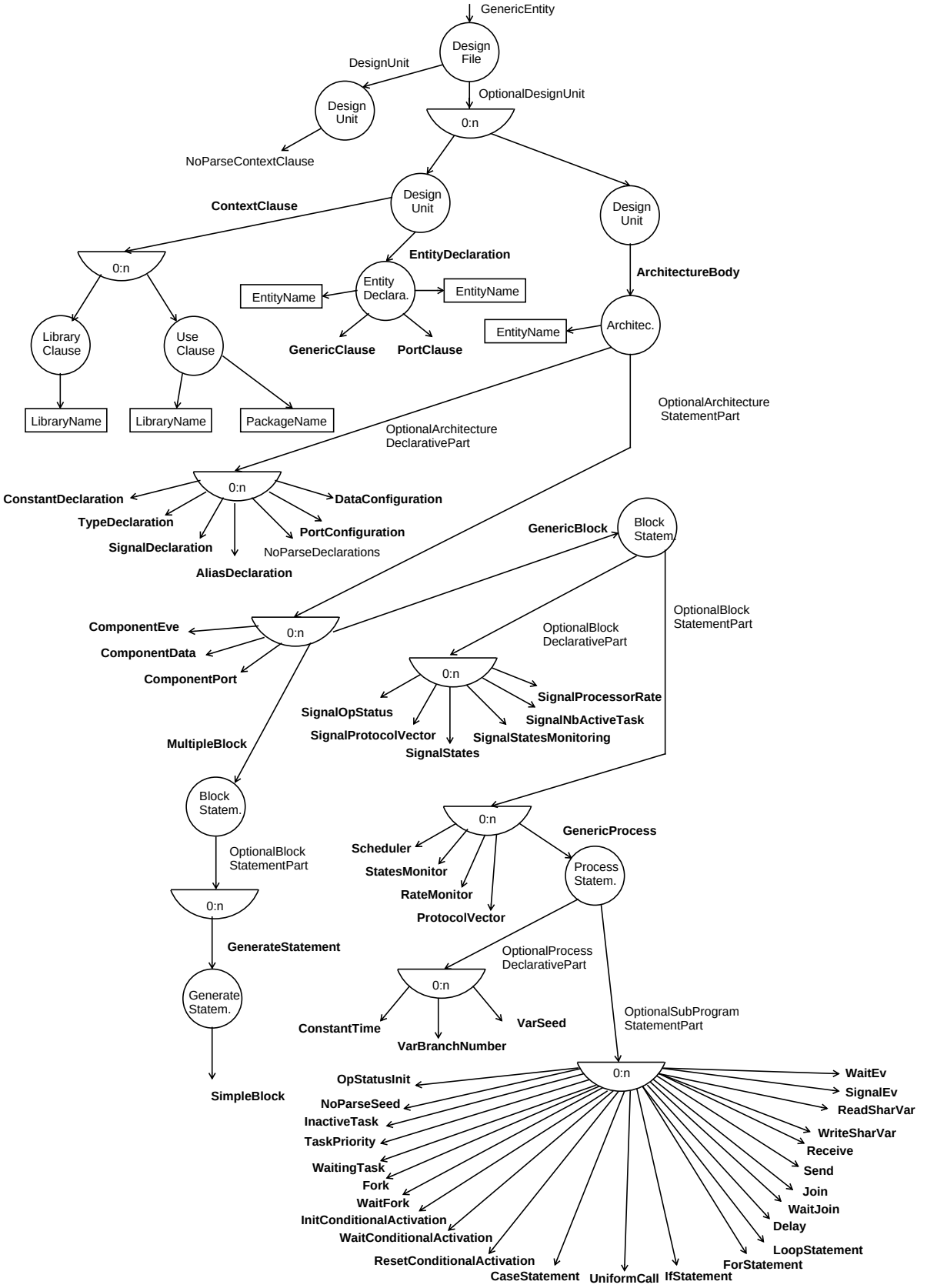
La représentation graphique du template du package est la suivante.



-Figure 6.1- Structure de données pour le template du package VHDL.

Les éléments en gras correspondent à des branches référencées par des variables du script.

La structure de données du template de l'entité est représentée par la figure 6.2.



-Figure 6.2- Structure de données pour le template VHDL de l'entité.

Le fichier template de l'entité contient comme constructions de base du langage VHDL:

- la déclaration d'une entité avec sa clause générique et son port,
- la déclaration des modèles de définition de constante, variable, signal, alias et type,
- la déclaration des modèles de composant VHDL représentant les éléments de relation du modèle MCSE et leur instanciation,
- la déclaration d'un block multiple contenant un block simple utilisé pour la transcription des éléments actifs du modèle MCSE,
- la déclaration de l'ordonnanceur et des signaux qui lui sont associés,
- la déclaration de l'ensemble des primitives utilisées pour la transcription du modèle de comportement d'un élément actif.

On trouve également dans le template des portions de code délimitées par les mots clés NO_PARSE_TEXT et END_NO_PARSE_TEXT et qui sont chargées uniquement sous forme de chaînes de caractères. Normalement, ces chaînes de caractères ne seront pas modifiées par la génération.

Pour une exploitation efficace du template, la structure de données est parcourue une seule fois au début de la génération et un ensemble de pointeurs sur ses éléments caractéristiques est mis à jour.

6.3 CONCEPTION D'UN SCRIPT

Pour bien comprendre quel doit être le contenu d'un script, rappelons tout d'abord le fonctionnement macroscopique d'un générateur:

- 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 initialisée à l'image de XTemplateDs.
- Il met ensuite à jour la structure OutputDs en se basant sur une analyse de la structure 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 grammaire XGrammarDs pour l'interprétation et sa table des symboles pour l'écriture des champs terminaux.

La difficulté repose sur les phases numérotées 3 et 4 pour lesquelles il faut traduire les règles de transcription décrites en langage naturel en un ensemble de règles de script. Au moins deux approches sont possibles pour ce problème.

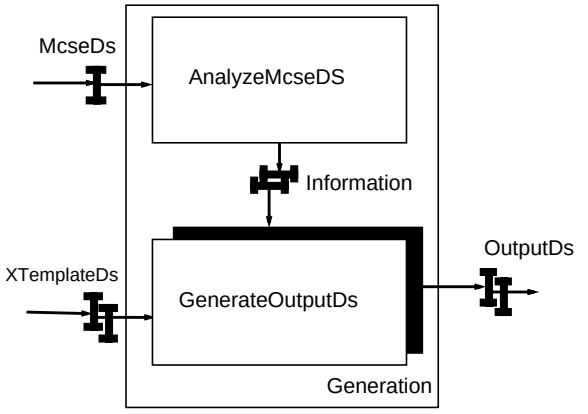
La première approche consiste à partir du modèle final (copie du template) et à utiliser un ensemble de règles de la forme "pour chaque <élément_modèle_final> faire si <condition_modèle_source> alors <mise_a_jour_modèle_final>". Cette solution est uniquement possible si le modèle final est une description structurée et de préférence hiérarchique et descendante. Son principal inconvénient est la recherche des conditions de mise à jour qui peut nécessiter un parcours multiple du modèle source. Ce principe est à rapprocher

de la technologie d'un système expert dans lequel les règles de transcriptions constituent une base des connaissances et l'analyse du modèle MCSE fournit les faits qui sont introduits dans un moteur d'inférence avec chaînage arrière.

La seconde approche consiste à partir du modèle source (modèle MCSE) et à utiliser un ensemble de règles de la forme "si <condition_modèle_source> alors <génération_modèle_final>. Cette fois, le modèle source est a priori parcouru une seule fois. Nous avons retenu cette solution plus avantageuse et plus conventionnelle.

6.3.1 Structure de la fonction Génération

Compte tenu du principe utilisé, la fonction Génération de la figure 5.9 sur la structure interne d'un générateur du chapitre 5 peut se raffiner comme le montre la figure 6.3. A noter que nous exploitons ici la méthodologie MCSE pour la conception de la solution. Une bonne conception fonctionnelle nécessite de chercher d'abord les variables internes caractéristiques indispensables pour déterminer les fonctions qui l'exploitent. La variable caractéristique retenue nommée Information représente les informations pertinentes extraites de d'analyse du modèle source MCSE et qui sont indispensables pour la génération du code de sortie.



-Figure 6.3- Raffinement de la fonction Génération.

Cette décomposition fonctionnelle fait apparaître qu'un script peut donc se décomposer en deux parties:

- une partie Analyse qui correspond au parcours ordonné de la structure de données McseDs et à l'extraction d'informations pertinentes pour la génération,
- une partie Génération qui exploite les informations obtenues de l'analyse et les structures génériques des templates pour générer la structure de données de sortie.

La partie Analyse est la partie dans laquelle on trouve le plus de similitudes entre des générateurs de code utilisant le même modèle source (modèle MCSE par exemple). Les différents types de parcours possibles sur la structure de données McseDs du modèle de performance de MCSE sont détaillés dans le paragraphe suivant.

La partie Génération exploite les informations obtenues de l'analyse et les règles de transcription. Cette partie est spécifique pour chaque générateur. On peut imaginer qu'un générateur de code C ciblant sur différents exécutifs temps-réel utilisera une seule fonction d'analyse et plusieurs fonctions de génération et templates.

La tendance naturelle du concepteur de générateur va être d'écrire les règles de cette partie de script de façon à obtenir un comportement du générateur identique à celui d'une transcription manuelle: c'est à dire qu'il va suivre l'ordre chronologique et structurel suivi lors d'une saisie manuelle du code cible. En procédant ainsi, il risque d'obtenir un script non optimisé. En effet, comme on travaille sur des structures de données, on n'est pas limité par l'accès séquentiel des fichiers et une même information peut très bien servir à mettre simultanément à jour des parties de code cible complètement disjointes. Cependant ce constat est à relativiser si les modèles source et destination sont structurellement très proches comme c'est le cas pour notre générateur VHDL. Pour faire face à la complexité, la meilleure solution est probablement d'écrire son script de manière naturelle et de l'optimiser dans un second temps. Pour cette phase d'optimisation, il est indispensable de trouver un formalisme adéquat pour représenter les règles de transcription décrite en langage naturel. Mais pour l'instant, nous n'avons pas fait de recherche dans cette direction.

La décomposition du script en deux parties facilite sa lisibilité et sa maintenance. Une modification de la grammaire du langage cible se veut être sans conséquence sur la partie Analyse. Il est souhaitable que le format des informations échangées entre la partie analyse et génération soit complètement indépendant des grammaires utilisées. Ainsi, une modification de la grammaire du modèle MCSE serait également invisible pour la partie génération. Ceci implique du point de vue implantation qu'il faut utiliser des listes intermédiaires et non pas directement les ensembles de la structure de données source. Comme la création de ces listes intermédiaires a un coût en temps d'exécution et mémoire occupée et que la grammaire du modèle MCSE est maintenant relativement stable, nous n'avons pas retenu ce principe pour l'échange d'informations entre les deux parties.

6.4 PARCOURS ET ANALYSE DU MODELE MCSE

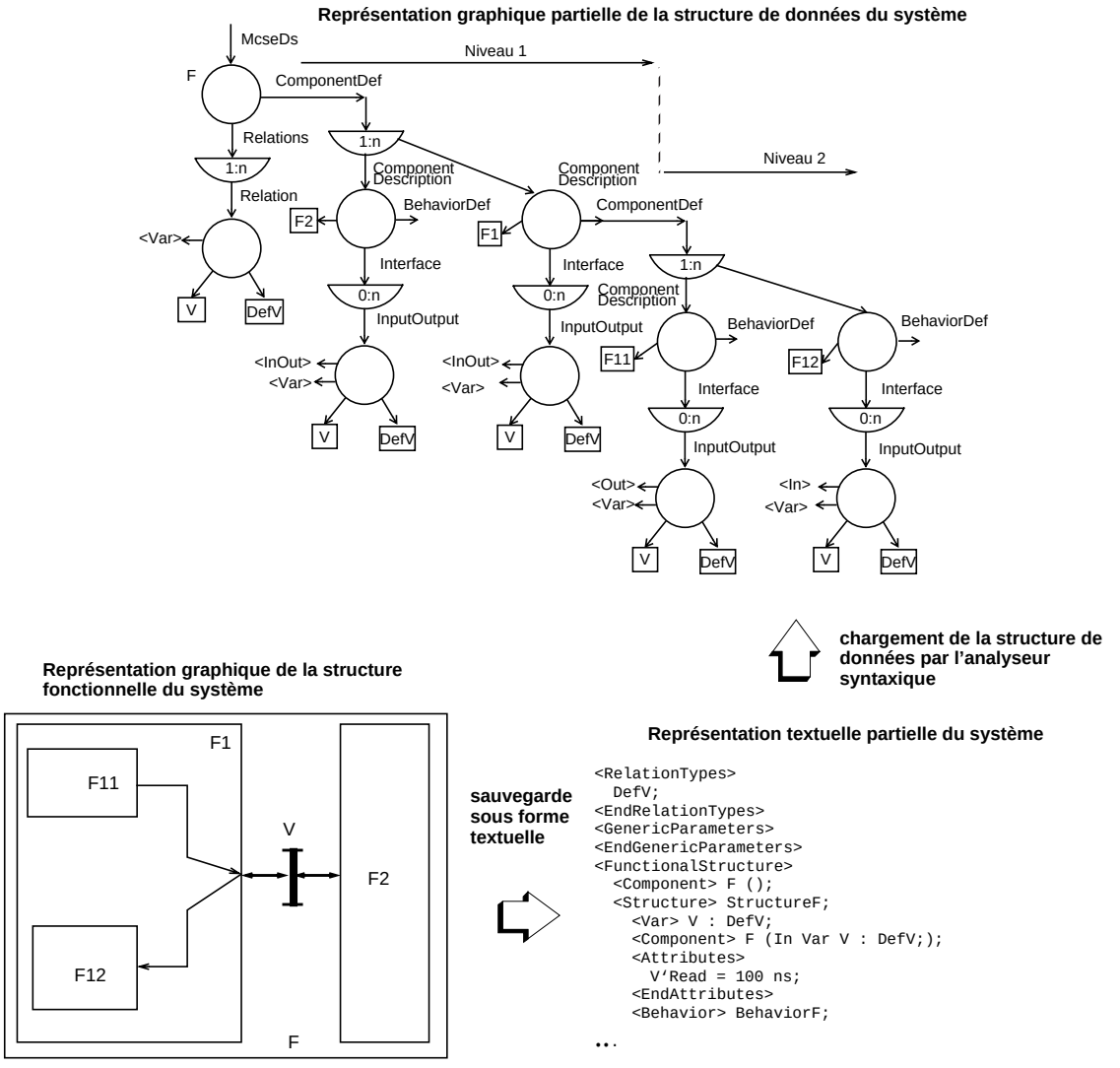
Le modèle de performance de MCSE se compose de deux vues complémentaires: la vue structurelle (dimension organisationnelle) et la vue comportementale (dimension temporelle). Le parcours du modèle MCSE se décompose donc en deux étapes: un parcours du modèle structurel et pour chaque fonction non raffinée un parcours de son modèle de comportement.

6.4.1 Parcours du modèle structurel de MCSE

La vue structurelle permet de décrire les éléments actifs (fonction, processeur) d'un système et leurs interconnexions (événement, variables partagée et port de communication). Le modèle structurel est hiérarchique: un élément actif peut être raffiné. Un élément actif peut aussi être une instance d'un modèle.

Pour illustrer nos propos, nous allons partir de l'exemple représenté sur la figure 6.4. Le système se compose de 2 fonctions F1 et F2 couplées par une variable partagée V. La fonction F1 est raffinée par 2 fonctions F11 et F12 utilisant toutes les deux la variable V.

On s'aperçoit avec la figure 6.4 que la structure de données d'un modèle MCSE est un graphe particulier dans lequel il existe une et une seule suite d'arêtes permettant de relier tout couple de sommets. C'est donc un arbre et plus précisément une arborescence. En effet par définition, une arborescence est un arbre dans lequel on particularise un sommet appelé la racine, ici le sommet McseDs, et à partir duquel sont placés les autres sommets par niveaux successifs. A partir de cette caractéristique, nous analysons les possibilités de parcours de la structure.



-Figure 6.4- Exemple de système avec sa structure de données.

-A- Les différents types d'algorithme de parcours d'une structure MCSE

Il y a deux catégories de parcours pour les arborescences: le parcours en largeur d'abord abrégé BFS (Breath First Search) et le parcours en profondeur d'abord abrégé DFS (Depth First Search).

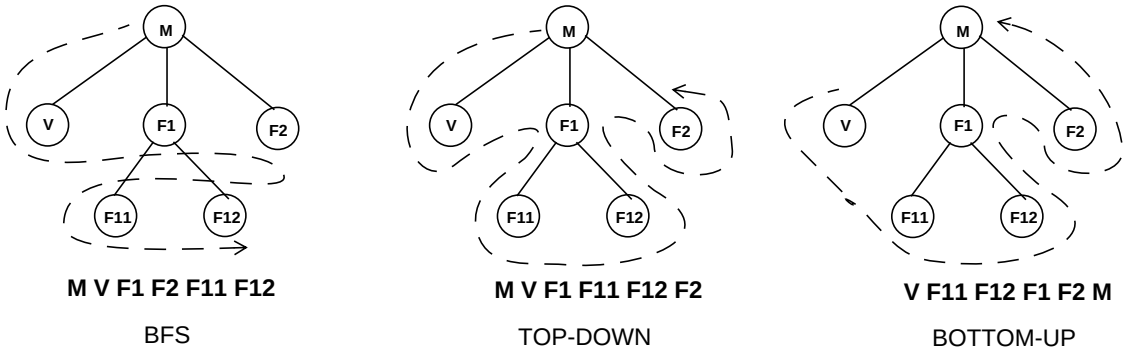
Le parcours BFS consiste à parcourir l'arborescence par niveau, de gauche à droite dans chaque niveau. Ce type de parcours est possible pour une analyse du modèle MCSE si la génération des éléments d'un niveau ne nécessite pas d'informations provenant de l'analyse de niveaux inférieurs. C'est le cas le plus simple de génération de code. Avec ce type de parcours, la hiérarchie du modèle de départ n'est pas respectée dans le modèle final.

On distingue deux types de parcours DFS:

- En préordre ou préfixé ou parcours descendant (top-down). On commence par exploiter la racine puis la sous-arborescence de la gauche vers la droite. Tout comme pour le parcours BFS, l'utilisation de ce type de parcours n'est possible que si la génération des éléments d'un niveau ne nécessite pas d'informations provenant de l'analyse de niveaux inférieurs. Par contre, la hiérarchie du modèle de départ est respectée dans le modèle final.

- En postordre ou postfixé ou parcours ascendant (bottom-up). On commence par exploiter les feuilles pour terminer par la racine. Lorsque l'on transcrit une feuille, on dispose forcément de toutes les informations nécessaires car celles-ci ont été collectées durant le parcours d'accès à cette feuille. Mais le modèle de départ est mis à plat.

L'ordre des traitements des feuilles de l'arborescence de l'exemple pour les 3 types de parcours possibles est le suivant.



-Figure 6.5- Les 3 types de parcours possibles de l'arborescence de l'exemple.

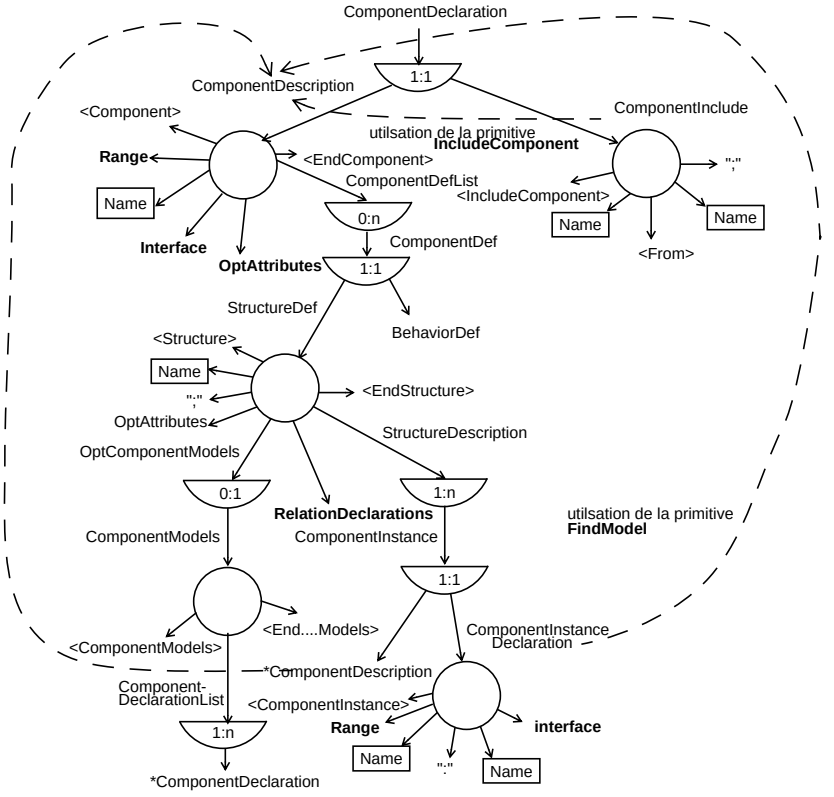
Si le modèle final ne permet pas de garder la hiérarchie du modèle source, l'algorithme de parcours le plus efficace pour la fonction analyse du générateur est le parcours ascendant (bottom-up). Si l'on souhaite respecter la hiérarchie du modèle source, il faut utiliser un algorithme de parcours descendant (top-down). Ce type d'algorithme ne pose aucun problème si la génération d'une feuille ne dépend que des informations des feuilles précédentes ou si il y a transcription quasi-directe entre un élément du modèle source et son correspondant dans le modèle cible. Mais la majorité des générateurs de code ne correspondront pas à cette situation. Sinon la structure de données ne sert à rien, car il aurait mieux valu faire une transcription directe au niveau de la règle de production de l'analyseur syntaxique. Il faut alors utiliser un algorithme mixte Top-Down/Bottom-Up permettant d'obtenir pour l'exemple précédent l'ordre de traitement M V F1 F11 F12 F1 F2 M. Concrètement, cela signifie que la génération d'une feuille se fera en deux phases. La première phase correspondant au parcours descendant consiste à générer même de manière incomplète tous les éléments du modèle final permettant de respecter la hiérarchie du modèle source. La second phase correspondant au parcours ascendant consiste à mettre à jour si nécessaire les éléments d'un niveau donné du modèle final en fonction d'informations provenant de niveaux inférieurs.

-B- Description des algorithmes de parcours

Nous allons maintenant donner dans le formalisme du langage script les algorithmes de parcours possibles du modèle MCSE. Comme il s'agit d'effectuer un parcours ordonné d'une structure de données d'un modèle MCSE, il est utile de rappeler auparavant le méta-modèle d'une structure de données MCSE. La figure 6.6 représente la structure de données simplifiée d'un élément actif.

Un élément actif peut être raffiné (StructureDef) ou décrit par un comportement (BehaviorDef). Le raffinement d'un élément actif se compose d'éléments de relation et d'éléments actifs (ComponentDescription) et/ou d'instances de modèle (ComponentInstanceDeclaration). Un modèle peut être défini en interne (ComponentModels)

ou importé d'une librairie (ComponentInclude). Ce méta-modèle fait clairement apparaître des bouclages indiquant que l'algorithme de parcours est récursif ou utilise une pile.



-Figure 6.6- Structure de données MCSE simplifiée d'un élément actif.

Dans un parcours d'une arborescence en profondeur d'abord, après avoir rencontré une feuille, on doit remonter dans l'arbre. Pour cela soit on utilise une pile pour empiler en descendant et désempiler en remontant (instruction Push et Pop du script), soit le système gère lui-même une pile par le biais de l'utilisation de la récursivité (clause Localvisibility d'un règle de script). La différence entre un parcours en préordre (top-down) et un parcours en postordre (bottom-up) est l'emplacement du traitement de la feuille: traitement de la feuille puis parcours de la sous-arborescence pour le préordre et l'inverse pour le postordre.

Les différents algorithmes de parcours du modèle MCSE vont donc se différencier par l'emplacement des appels des règles de génération et l'utilisation d'une pile implicite (LocalVisibility) ou explicite (Push et Pop).

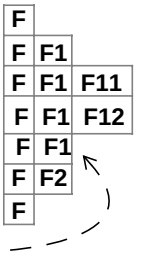
L'algorithme du parcours du type TopDown est décrit ci-après sous forme script.

```
TopDownAnalyzeStructuralDescription:: {
  AnalyzeComponentModelOrInstance;
  ComponentDef:=ComponentModel.ComponentDefList;
  /* we assume that the first model is the model of the current configuration */
  TmpType:=TypeOf(ComponentDef);
  Case(TmpType='StructureDef' :
        OptRelationDescriptions:=ComponentDef.StructureDef.RelationDeclarations;
        ForEach(OptRelationDescriptions : RelationLinks(OptRelationDescriptions);
                  CurrentRelation:=OptRelationDescriptions;
                  AnalyzeRelationElement;);
        StructureDescription:=ComponentDef.StructureDef.StructureDescription;
        ForEach(StructureDescription :
           TmpType:=TypeOf(StructureDescription);
           Case (TmpType = 'ComponentDescription' :
                   ComponentModel:=StructureDescription.ComponentDescription;
                   ComponentInstance:=Nil;
                 | Else :
                   ComponentInstance:=StructureDescription.ComponentInstanceDeclaration;
                   ComponentModel:=FindModel(ComponentInstance,1););
```

```
            TopDownAnalyzeStructuralDescription;);
     | Else : InternalRelationsList:=ComponentDef.BehaviorDef.InternalRelationsList;
        ForEach(InternalRelationsList :  RelationLinks(InternalRelationsList);
              CurrentRelation:=InternalRelationsList;
              AnalyzeInternalRelationElement;);
       BehaviorInstance:=ComponentDef.BehaviorDef.BehaviorDefinition;
       AnalyzeBehavioralDescription;
   );
  }
```

L'algorithme du parcours du type BottomUp est le suivant.

```
BottomUpAnalyzeStructuralDescription:: {
  ComponentDef:=ComponentModel.ComponentDefList;
  TmpType:=TypeOf(ComponentDef);
  Case(TmpType='StructureDef' :
        OptRelationDescriptions:=ComponentDef.StructureDef.RelationDeclarations;
        ForEach(OptRelationDescriptions : RelationLinks(OptRelationDescriptions););
        StructureDescription:=ComponentDef.StructureDef.StructureDescription;
        ForEach(StructureDescription :
          Push(ComponentModel);
          Push(ComponentInstance);
          TmpType:=TypeOf(StructureDescription);
          Case (TmpType = 'ComponentDescription' :
                  ComponentModel:=StructureDescription.ComponentDescription;
                  ComponentInstance:=Nil;
                | Else :
                  ComponentInstance:=StructureDescription.ComponentInstanceDeclaration;
                  ComponentModel:=FindModel(ComponentInstance,1);
          );
          BottomUpAnalyzeStructuralDescription;
          Pop(ComponentModel);
          Pop(ComponentInstance);
        );
        AnalyzeComponentModelOrInstance;
        OptRelationDescriptions:=ComponentModel.ComponentDefList.StructureDef.RelationDeclarations;
        ForEach(OptRelationDescriptions : CurrentRelation:=OptRelationDescriptions;
                AnalyzeRelationElement;);
     | Else : InternalRelationsList:=ComponentDef.BehaviorDef.InternalRelationsList;
        ForEach(InternalRelationsList :  RelationLinks(InternalRelationsList);
                CurrentRelation:=InternalRelationsList;
                AnalyzeInternalRelationElement;);
       AnalyzeComponentModelOrInstance;
       BehaviorInstance:=ComponentDef.BehaviorDef.BehaviorDefinition;
       AnalyzeBehavioralDescription;
   );
}
```



Le contenu de la variable ComponentModel (pile) pour l'exemple précédent est représenté sur la droite de l'algorithme.

L'algorithme mixte se déduit facilement des deux autres. La génération se passe en deux phases: une phase de déclaration lors du parcours descendant et une phase de mise à jour lors du parcours remontant.

```
AnalyzeStructuralDescription:: {
  AnalyzeComponentModelOrInstance;
  ComponentDef:=ComponentModel.ComponentDefList;
  TmpType:=TypeOf(ComponentDef);
  Case(TmpType='StructureDef' :
        OptRelationDescriptions:=ComponentDef.StructureDef.RelationDeclarations;
        ForEach(OptRelationDescriptions : RelationLinks(OptRelationDescriptions);
                  CurrentRelation:=OptRelationDescriptions;
                 AnalyzeRelationElement;);
        StructureDescription:=ComponentDef.StructureDef.StructureDescription;
        ForEach(StructureDescription :
           Push(ComponentModel);Push(CompnentInstance);Push(PathLevel);
           TmpType:=TypeOf(StructureDescription);
          Case (TmpType = 'ComponentDescription' :
                  ComponentModel:=StructureDescription.ComponentDescription;
                  ComponentInstance:=Nil;
                  PathLevel:=PathLevel&FileSeparator&ComponentModel.Name;
                | Else :
                  ComponentInstance:=StructureDescription.ComponentInstanceDeclaration;
                  ComponentModel:=FindModel(ComponentInstance,0););
                  TmpType:=TypeOf(ComponentModel);
                  Case (TmpType = 'ComponentInclude' :
                        PathLevel:=GetLibraryPath(ComponentModel.ComponentInclude);
                       ComponentModel:=IncludeComponent(ComponentMode.ComponentIncludel););
                  PathLevel:=PathLevel&FileSeparator&ComponentInstance.Name;);
           AnalyzeStructuralDescription;
           UpdateComponentModel;
           Pop(ComponentModel);Pop(ComponentInstance);Pop(PathLevel);
           );
```

```
        OptRelationDescriptions:=ComponentModel.ComponentDefList.StructureDef.RelationDeclarations;
        ForEach(OptRelationDescriptions : CurrentRelation:=OptRelationDescriptions;
               UpdateRelationElement;);
     | Else : InternalRelationsList:=ComponentDef.BehaviorDef.InternalRelationsList;
        ForEach(InternalRelationsList :  RelationLinks(InternalRelationsList);
                 CurrentRelation:=InternalRelationsList;
                 AnalyzeInternalRelationElement;);
       BehaviorInstance:=ComponentDef.BehaviorDef.BehaviorDefinition;
       AnalyzeBehavioralDescription;
   );
}
```

Les algorithmes précédents utilisent des instructions du script qui sont spécifiques au modèle MCSE (mots en italiques):

- *IncludeComponent* importe un composant défini dans une librairie.
- *RelationLinks* met à jour une association bidirectionnelle entre l'élément de relation et les interfaces de composants qui lui sont directement ou indirectement rattachées.
- *FindModel* retourne la description interne du modèle d'une instance.

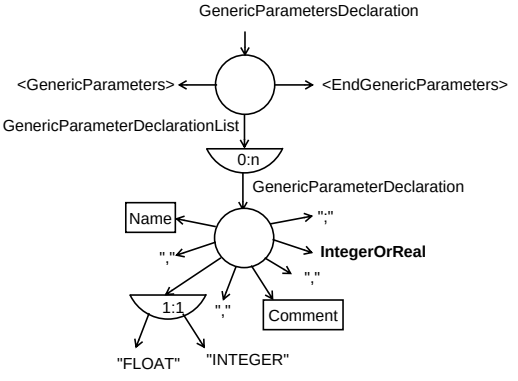
Ils font tous appel à trois autres règles de script:

- *AnalyzeComponentModelOrInstance* pour l'analyse d'un élément actif,
- *AnalyzeRelationElement* pour l'analyse d'un élément de relation,
- *AnalyzeInternalRelationElement* pour l'analyse d'un élément de relation interne défini dans le modèle de comportement d'un élément actif.

-C- Analyse d'un élément actif

L'interface d'un composant comprend sa dimension (Range), les paramètres génériques (GenericParameters) et la définition des entrées et sorties du composant et le type de chacune.

Le type et la valeur par défaut des paramètres génériques utilisés dans la description d'un système sont définis dans une liste située à la racine de la structure de données du modèle MCSE. Ces définitions s'accompagnent d'un message d'information concernant le rôle joué par le paramètre générique défini. Le modèle de structure pour la déclaration des paramètres génériques est le suivant.



-Figure 6.7- Structure de données pour la déclaration des paramètres génériques.

L'algorithme d'analyse des paramètres génériques qui exploite la structure de données ci-dessus est le suivant.

```
AnalyzeGenericParameters :: {
  GenericParameterDeclatationList:=McseDs.FunctionalDescription.GenericParametersDeclaration.
                                GenericParameterDeclarationList;
  Case(OptGenericParameters # Nil :
    ParameterName:=OptGenericParameters.Name;
    FindGenericParameterDeclaration;
    Case(GenericParameterDeclatation=Nil :
       Error('Generic Parameter '&ParameterName&' not defined');
    | Else : GenericParameterType:=TypeOf(GenericParameterDeclatation.FloatOrInteger);
       GenericParameterDefaultValue:=GenericParameterDeclatation.IntegerOrReal;
       GenerateFirstGenericParameter;);
```

```
    NameList:=OptGenericParameters.NameList;
    ForEach(NameList : ParameterName:=NameList.Name;
      FindGenericParameterDeclaration;
      Case(GenericParameterDeclatation=Nil :
          Error('Generic Parameter '&ParameterName&' not defined');
      | Else : GenericParameterType:=TypeOf(GenericParameterDeclatation.FloatOrInteger);
          GenericParameterDefaultValue:=GenericParameterDeclatation.IntegerOrReal;
          GenerateOneGenericParameter;);
    );
  | Else :
          GenerateNoGenericParameters;
  );
}
```

avec

```
FindGenericParameterDeclaration :: {
  GenericParameterDeclatation:=Nil;
  While ((GenericParameterDeclatationList # Nil AND
      GenericParameterDeclatation=Nil) :
      Case(GenericParameterDeclatationList.Name = ParameterName :
        GenericParameterDeclatation:=
                    GenericParameterDeclatationList;
      | Else :
        GenericParameterDeclatationList:=
                    GenericParameterDeclatationList.nextFriend;);
  );
}
```

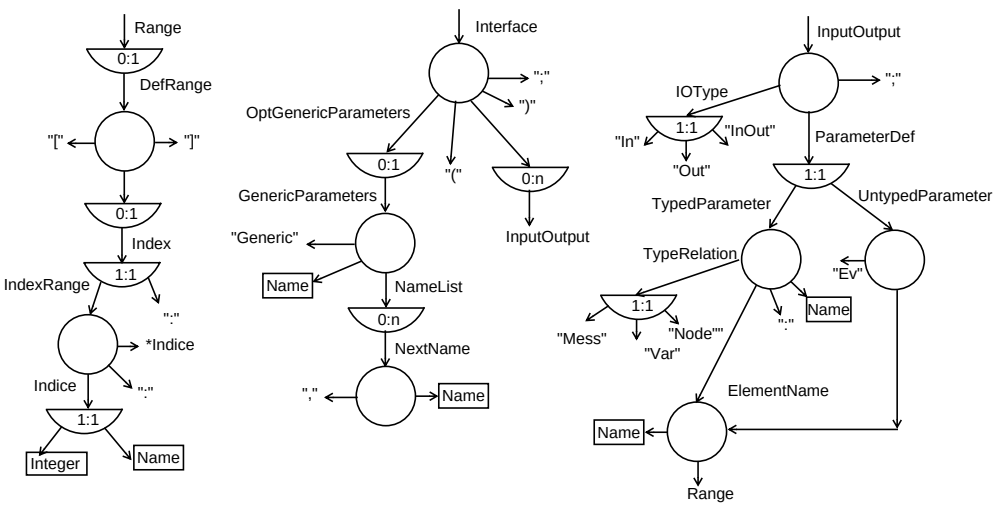
Lorsqu'il n'y a pas de paramètres génériques, il faut détruire si nécessaire la construction correspondante dans la structure de données cible (clause générique d'une entité VHDL par exemple). Sinon pour chaque paramètre générique, on recherche son type et sa valeur par défaut.

L'analyse de l'interface d'un composant doit tenir compte des différents cas de l'instanciation multiple.

On utilise pour cela deux indicateurs:

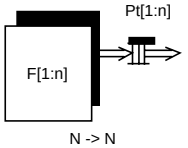
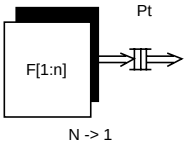
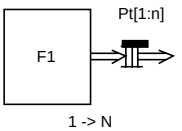
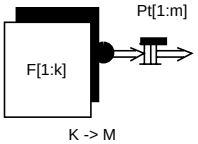
- *BlackBall* indique que l'élément d'interconnexion associé à l'entrée/sortie analysée doit être de dimension 2 (un vecteur de vecteur par exemple),
- *Vector* indique que l'élément d'interconnexion doit être de dimension 1 (un vecteur par exemple).

La figure suivante montre le méta-modèle pour la définition de l'interface.



-Figure 6.8- Structure de données de la grammaire pour la déclaration de l'interface.

Les différents cas possibles de l'instanciation multiple des fonctions et des éléments de relation sont représentés dans le tableau suivant (voir chapitre 3).

Graphique	IO Range	Observations	Black Ball	Vector
	**[ ]**	Il y a correspondance indice par indice.	0	0
	**Null**	Toutes les fonctions F[i] exploitent le port Pt en respectant son degré de partage.	0	0
	**[:]**	La Fonction F1 peut utiliser l'un quelconque des ports du vecteur Pt[1:n] en le désignant par son indice (utilisation de l'attribut 'Path pour la sélection)	0	1
	**[:]**	Chaque fonction F[i] peut accéder à tous les éléments du vecteur Pt[1:m] (attribut 'Id pour désigner la source et attribut 'Path pour la destination). La valeur par défaut de l'attribut 'Id est la valeur 'me représentant l'indice courant de la fonction.	1	–

-Figure 6.9- Les différents cas de l'instanciation multiple.

L'algorithme d'analyse d'un élément actif est donc le suivant.

```
AnalyzeComponentModelOrInstance :: {
  LocalVisibility NbTask;
  NbTask:=0;
  InterfaceModel:=ComponentModel.Interface;
  Case (ComponentInstance # Nil :
    ComponentName:=ComponentInstance.Name;
    Range:=ComponentInstance.Range;
    InterfaceInstance:=ComponentInstance.Interface;
    GetAttributes(ComponentInstance,AttributesList);
  | Else : ComponentName:=ComponentModel.Name;
    Range:=ComponentModel.Range;
    InterfaceInstance:=ComponentModel.Interface;
    GetAttributes(ComponentModel,AttributesList););
  GetRangeExpression;
  ComponentRangeExpression:=RangeExpression;
  ComponentMultiple:=MultipleComponent;
  GenerateComponentFromNameAndRange;
  OptGenericParameters:=InterfaceInstance.OptGenericParameters;
  AnalyzeGenericParameters;
  CurrentInputOutputModel:=InterfaceModel.InputOutputList;
  InputOutput:=InterfaceInstance.InputOutputList;
  GenerateBeginInterface;
  ForEach(InputOutput :
    AnalyzeInputOutputInterface;
    CurrentInputOutputModel:=CurrentInputOutputModel.nextFriend;);
  GenerateEndInterface;
}
```

avec

```
AnalyzeInputOutputInterface :: {
  Vector:=0;
  BlackBall:=0;
  TmpType:=TypeOf(InputOutput.ParameterDef);
  Case(TmpType = 'TypedParameter' :
    ParameterName:=InputOutput.ParameterDef.TypedParameter.ElementName.Name;
    ParameterModelName:=CurrentInputOutputModel.ParameterDef.TypedParameter.ElementName.Name;
    TypeName:=InputOutput.ParameterDef.TypedParameter.Name;
```

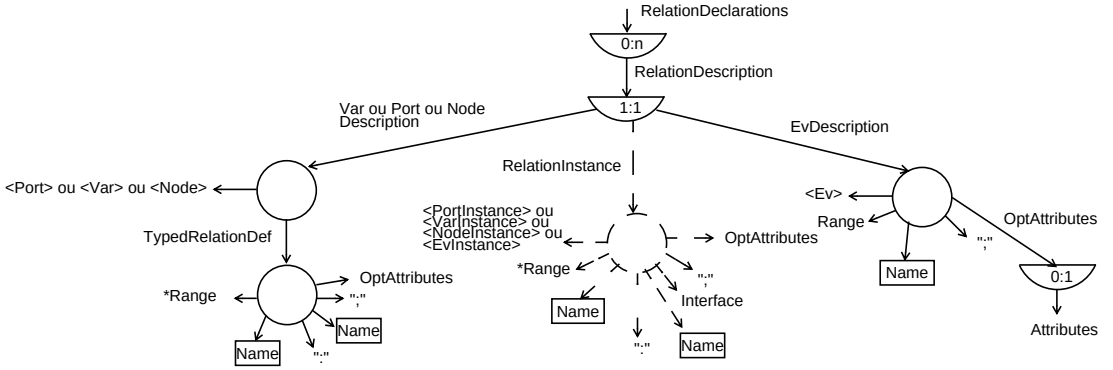
```
    ParameterType:=TypeOf(InputOutput.ParameterDef.TypedParameter.TypeRelation);
    Range:=InputOutput.ParameterDef.TypedParameter.ElementName.Range;
  | Else : ParameterName:=InputOutput.ParameterDef.UntypedParameter.ElementName.Name;
    ParameterModelName:=CurrentInputOutputModel.ParameterDef.UntypedParameter.ElementName.Name;
    TypeName:=Null;
    ParameterType:=TypeOf(InputOutput.ParameterDef.UntypedParameter.Ev);
    Range:=InputOutput.ParameterDef.UntypedParameter.ElementName.Range;);
  GetRangeExpression;
  Case (RangeExpression # Null : /* Null si Range=[] */
    Vector:=1;
    BlackBall:=ComponentMultiple;
    RelationalElement:=FindRelationalElement(InputOutput);
    TmpType:=TypeOf(RelationalElement);
    Case(TmpType='EvDescription' :
      Range:=RelationalElement.EvDescription.Range;
    | Else : Range:=RelationalElement.PortDescription.TypedRelationDef.Range;);
    GetRangeExpression;
    VectorExpression:=RangeExpression;
  | Else : VectorExpression:=ComponentRangeExpression;);
  FindNumberOfAccessInLevel;
  TmpType:=TypeOf(InputOutput.IOType);
  Case (TmpType = 'In' : IsInput:=1;
  | TmpType = 'InOut' : IsInput:=2;
  | Else : IsInput:=0;);
  GenerateFromInputOutput;
}
```

Les informations mises à disposition de la règle de génération d'un élément actif sont: sa dimension, son nom et sa liste d'attributs. Pour la génération de chaque interface du composant, on fournit également le nom de l'interface, son type, le type de la donnée, la dimension de l'élément de relation qui lui est associé, le nombre d'interfaces du raffinement du composant qui lui sont associés et les indicateurs Vector, BlackBall et IsInput.

-D- Analyse d'un élément de relation

Les composants sont couplés entre eux par l'intermédiaire d'éléments de relation. Ils sont de 3 types pour la structure fonctionnelle (Evénement, Variable partagée, Port) et de 3 types pour la structure d'exécution (Signal, Mémoire commune, Noeud de communication). Les 2 premiers types sont identiques pour les 2 structures. Pour la simplification, on a conservé uniquement les noms Ev et Var. On ne traitera pas ici le problème du raffinement des éléments de relation.

Le méta-modèle pour la définition d'un élément de relation est donné par la figure suivante.

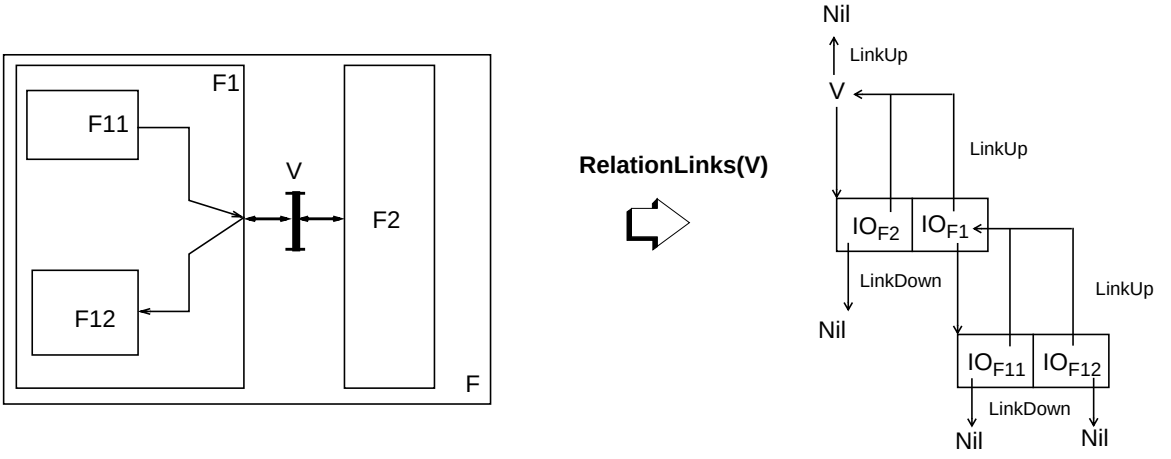


-Figure 6.10- Structure de données de la grammaire pour la déclaration d'un élément de relation.

Pour gérer le problème de la traçabilité des liens entre éléments de relation et interfaces de composants, on utilise un lien bidirectionnel implanté dans la structure de données à l'aide de deux références (LinkUp et LinkDown). La mise à jour de cette association bidirectionnelle se fait par l'instruction **RelationLinks** qui s'applique sur un élément de relation ou l'interface

d'un composant. Après exécution de cette instruction, le champ LinkUp référence l'élément de relation ou l'interface de niveau supérieur. Le lien LinkDown référence un vecteur contenant les références de toutes les interfaces du niveau associées à l'élément concerné.

La figure ci-dessous illustre cette association bidirectionnelle dans le cas de l'exemple du début de chapitre.



-Figure 6.11- Liens entre éléments de relation et interfaces de composants.

La règle d'analyse d'un élément de relation est la suivante.

```
AnalyzeRelationElement:: {
  TmpType:=TypeOf(CurrentRelation);
  Case((TmpType='PortDescription' OR  TmpType='NodeDescription'):
    RelationType:='Port';TypedRelationDef: = CurrentRelation.PortDescription.TypedRelationDef;
    Range:=TypedRelationDef.Range;Name:=TypedRelationDef.Name;
  | TmpType = 'VarDescription' :
    RelationType:='Var';
    TypedRelationDef: = CurrentRelation.VarDescription.TypedRelationDef;Range:=TypedRelationDef.Range;
    Name:=TypedRelationDef.Name;
  | TmpType = 'EvDescription' :
    RelationType:='Ev';Range:=CurrentRelation.EvDescription.Range;
    Name:=CurrentRelation.EvDescription.Name;);
  GetRangeExpression;
  ElementRangeExpression:=RangeExpression;
  ElementMultiple:=MultipleComponent;
  FindNumberOfAccessInLevel;
  GetAttributes(CurrentRelation,AttributesList);
  GenerateFromRelationElement;
}
```

Les informations mises à disposition de la règle de génération d'un élément de relation sont le type de l'élément (Port, Var, Ev), sa dimension, son nom, sa liste d'attributs, le nombre d'interfaces du niveau et le nombre total d'interfaces du système rattachées à l'élément.

L'analyse d'un élément de relation interne d'une description comportementale est à rapprocher de celle d'un élément de relation d'une structure fonctionnelle ou exécutive. Contrairement au modèle structurel, le modèle comportemental n'est pas forcément complètement hiérarchique. Un élément de relation interne peut en effet être relié à une interface d'instance d'une activité ou directement sur une condition d'évolution ou une action. L'instruction RelationLinks crée donc une association bidirectionnelle entre l'élément de relation interne et l'interface de l'activité ou le champ ElementName associé à une condition d'évolution ou une action. On utilise ce champ ElementName car une condition d'évolution ou une action peut être simple ou composée. Les instructions FindRelationElement et FindInternalRelationElement devront donc être appliquées sur le champ ElementName d'une condition ou d'une action pour retrouver l'élément de relation associé à cette condition ou action.

6.4.2 Parcours du modèle comportemental

La vue comportementale permet de décrire le comportement des fonctions. Le modèle de comportement repose sur:

- des activités dynamiques décomposables (raffinement) ou élémentaire (algorithme ou temps d'exécution),
- des opérateurs de composition d'activités (séquence, parallélisme, alternative, répétition et attente conditionnelle),
- des opérateurs de construction de condition (attente sur une entrée),
- des opérateurs de construction d'action (génération d'une sortie) et des attributs pour paramétrer le modèle.

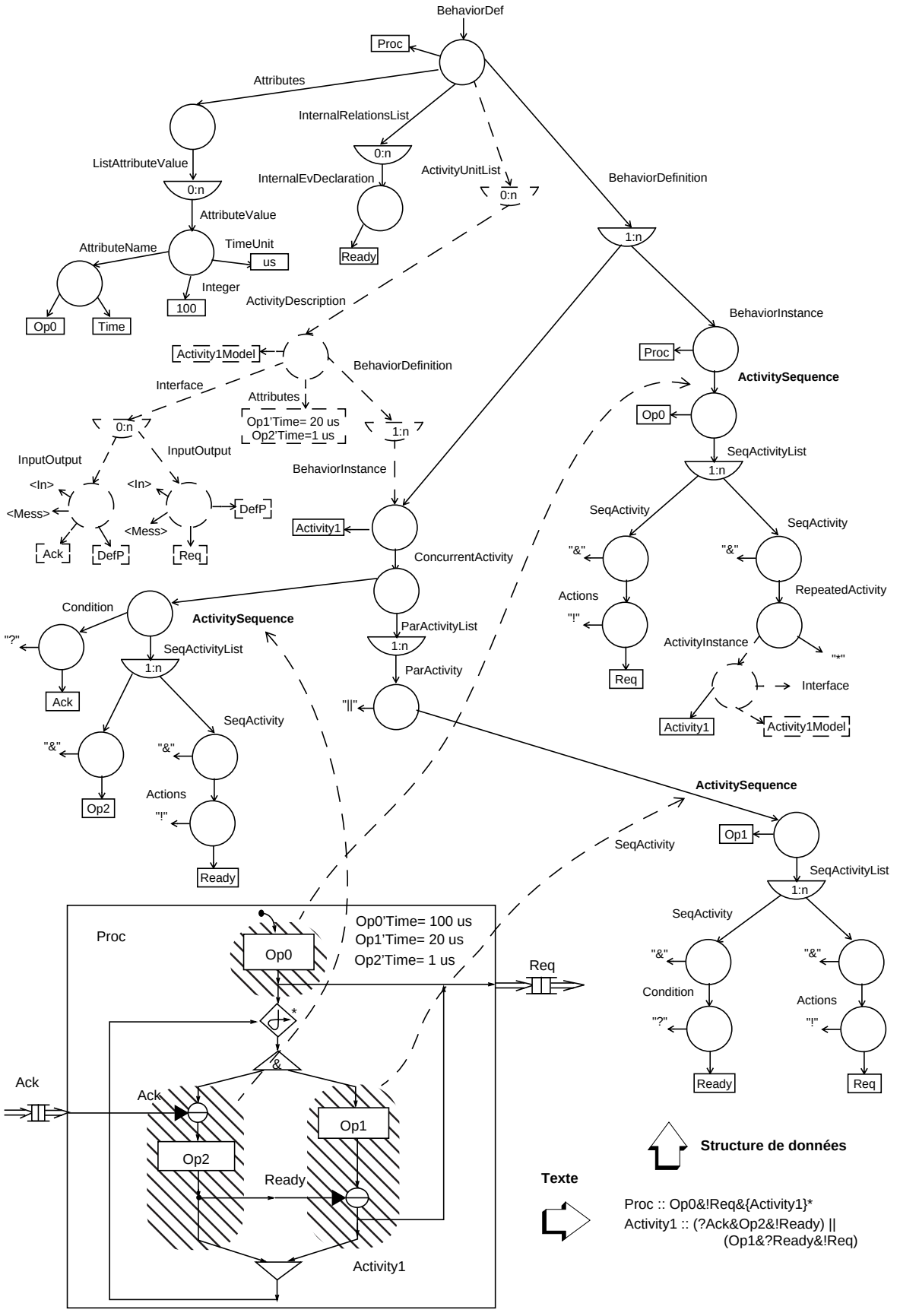
Contrairement au modèle structurel, le modèle comportemental est plus riche en concepts et n'est pas strictement hiérarchique, ce qui ne facilite son analyse. Dans le chapitre 4 sur les règles de transcription, nous avons également vu que les règles de transcription sont plus complexes pour la composante comportementale du modèle de performance de MCSE.

La figure 6.12 représente la représentation graphique du modèle de comportement d'un exemple, sa description textuelle équivalente et le modèle graphique de la structure de données obtenue à partir de l'analyse de la description textuelle.

Le modèle de comportement décrit un mécanisme de requête/acquittement. Après une opération Op0 représentant des opérations d'initialisation, une première requête est envoyée via le port de communication Req. Comme le suggère la boucle infinie, cette initialisation n'est effectuée qu'une seule fois lors de la création (ou instanciation) de l'activité Proc. Cette activité a alors une durée de vie illimitée.

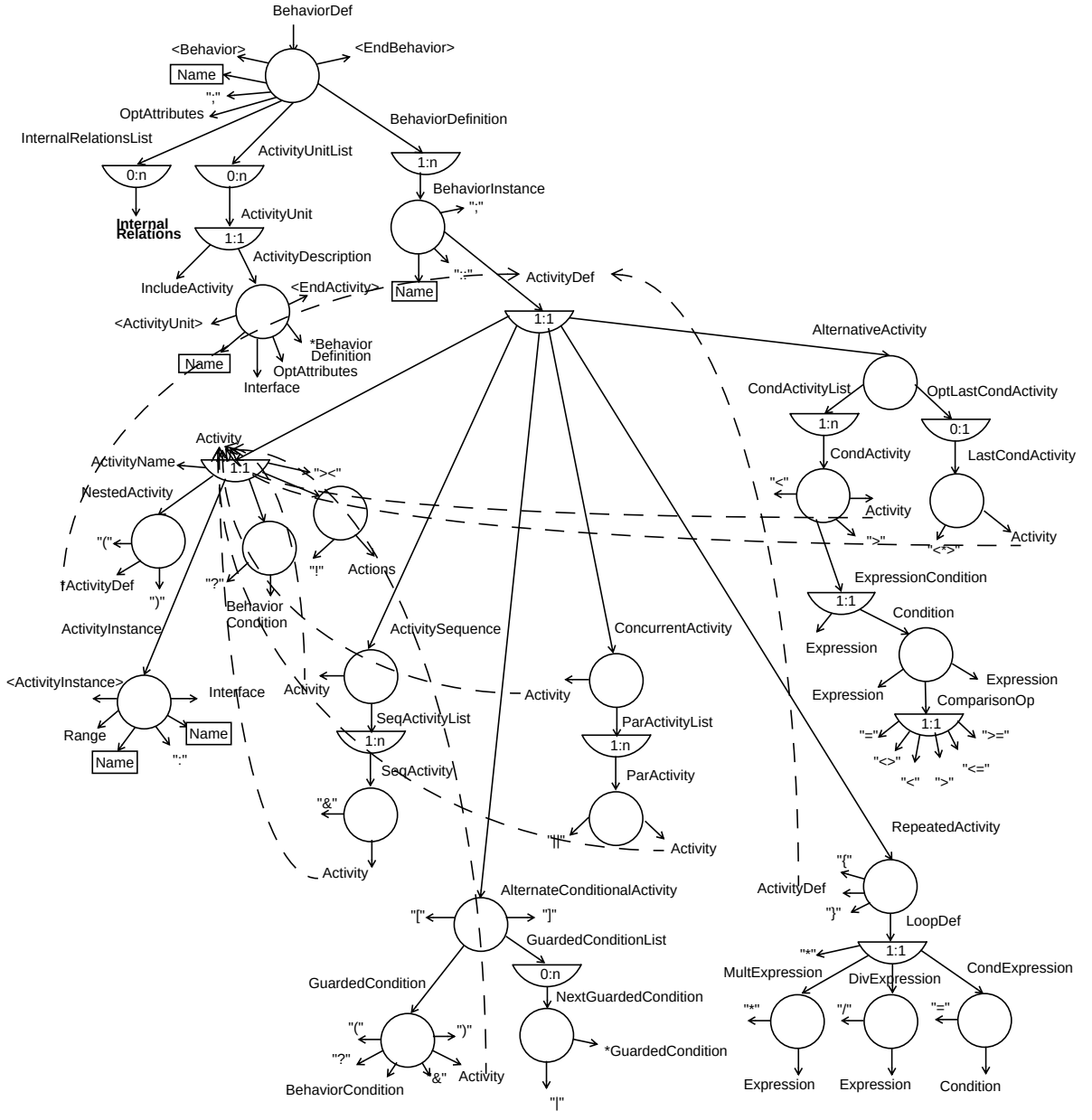
La description du comportement se poursuit avec un parallélisme à deux branches. Ce parallélisme peut éventuellement être englobé dans une activité nommée ici Activity1. Dans ce cas, l'activité Activity1 est déclarée comme un modèle d'activité nommé Activity1Model dans la liste ActivityUnitList (partie en pointillé des figures) et comme une instance de ce modèle au niveau du comportement de l'activité Proc. Le parallélisme tel qu'il est décrit permet d'occuper la ressource d'exécution (opération Op1) même si l'activité Proc est en attente de l'acquittement (attente de message provenant du port de communication Ack). Après avoir reçu l'acquittement, la branche de gauche signale (signalisation ready) à la branche de droite qu'elle peut envoyer une nouvelle requête.

Lorsque l'on regarde la représentation graphique et textuelle de l'exemple, on pense a priori que la difficulté va être d'extraire les séquences d'opérations (zones hachurées) qui seront générées sous forme de process (VHDL) ou de tâches (C/ETR, ADA) ou de thread (Java). En réalité, grâce à une bonne méta-modélisation du modèle comportemental, ce travail est fait implicitement par l'analyseur syntaxique. En effet, chaque séquence est une branche de la structure de données chargée dont la racine est du type ActivitySequence.



-Figure 6.12- Exemple de modèle comportemental avec sa structure de données.

L'algorithme de parcours du modèle comportemental se déduit de l'analyse du méta-modèle de structure de données pour la description des activités dont la représentation graphique est donnée ci-dessous.



-Figure 6.13- Méta-modèle pour la description des activités.

Cette représentation graphique fait clairement apparaître un rebouclage entre les noeuds de composition d'activités (ActivitySequence, ConcurrentActivity, MultipleActivity, AlternativeActivity, et AlternateConditionalActivity) et le noeud Activity. Il y a également un bouclage entre les noeuds Activity, RepeatedActivity et ActivityDef. Par conséquent, l'algorithme de parcours est un algorithme itératif utilisant une pile ou un algorithme récursif. Pour faciliter la lisibilité et la compréhension du programme, il est préférable de définir une règle de script pour l'analyse de chaque type de noeud. L'algorithme de parcours du modèle comportemental ainsi obtenu comporte une récursivité indirecte et se compose de l'ensemble des règles suivantes:

- AnalyzeAlternateActivity pour l'analyse de l'alternative,
- AnalyzeRepeatedActivity pour l'analyse de l'itération,
- AnalyzeConcurrentActivity pour l'analyse du parallélisme,
- AnalyzeActivitySequence pour l'analyse de la séquence,
- AnalyzeAlternateConditionalActivity pour l'analyse de l'attente conditionnelle,
- AnalyzeActivity et AnalyzeActivityDef pour l'analyse d'une activité composée,
- AnalyzeActivityName pour l'analyse d'une activité élémentaire,
- AnalyzeBehaviorCondition pour l'analyse d'une condition d'attente,
- AnalyzeActions pour l'analyse d'une génération de sortie,
- AnalyzeActivityInstance pour l'analyse d'une instance d'activité.

A titre d'exemple, le code script de certaines de ces règles d'analyse est donné dans les paragraphes suivants. Les mots en italiques représentent les noms de règles de génération de la structure de données de sortie appelées lors de l'analyse du modèle source. Les mots en gras sont des appels aux règles de la partie Analyse du Script.

-A- Analyse du parallélisme

```
AnalyzeConcurrentActivity :: {
  LocalVisibility ParActivityList,Activity,NbTask;
  Activity:=ConcurrentActivity.Activity;
  GenerateBeginTaskSynchronisation;
  AnalyzeActivity;
  GenerateEndTaskSynchronisation;
  ParActivityList:=ConcurrentActivity.ParActivityList;
  ForEach(ParActivityList : NbTask:=ValueOf(NbTask) + 1;
    GenerateFromParActivity; /* create new process or task */
    GenerateBeginTaskSynchronisation;
    Activity:=ParActivityList.Activity;
    AnalyzeActivity;
    GenerateEndTaskSynchronisation;
  );
}
```

La séparation de la première des N branches d'une concurrence permet de continuer la description de cette branche dans la séquence en cours et d'obtenir une traduction en N-1 tâches. La traduction nécessite la création/instanciation de tâches et l'utilisation de mécanismes de synchronisation surtout si la création des tâches ne peut être dynamique comme c'est le cas pour VHDL.

-B- Analyse d'une activité simple

Un nom d'activité peut représenter une activité élémentaire ou un découpage de l'activité analysée défini dans l'ensemble BehaviorDefinition d'où la règle suivante.

```
AnalyzeActivityName :: {
ElementName:=Activity.ActivityName.ElementName;
  Name:=ElementName.Name;
  ActivityDef:=FindActivityDescription(Activity.ActivityName);
  Case (ActivityDef = Nil :
    Case (ForPerformanceEvualation = 'YES' :
      TmpType:=TypeOf(ElementName.superFather.BehaviorDef);
      Case(TmpType = 'ActivityDescription' :
        Attributes:=ElementName.superFather.ActivityDescription.OptAttributes;
        ComponentAttributes:=Nil;
      | Else :
        BehaviorDef:=ElementName.superFather.BehaviorDef;
        Attributes:=BehaviorDef.OptAttributes;
        ComponentAttributes:=BehaviorDef.superFather.ComponentDescription.OptAttributes;
      );
      SearchAttributeDesignator:='Time';
      Case(Attributes # Nil : FindAttributeValue;);
      Case ((Expression=Nil AND ComponentAttributes#Nil) :
        Attributes:=ComponentAttributes;
        FindAttributeValue;
      );
      Case(Expression=Nil : Error('Attribut Time not defined for '&Name);
      | Else : GenerateElementaryActivity;
      );
    | Else : FileName:=PathLevel&FileSeparator&Name&SuffixFile;
```

```
      FileInclude:=IncludeOp(FileName,1);
     GenerateFromIncludeFile;
    );
  | Else : AnalyzeActivityDef;
  );
}
```

avec

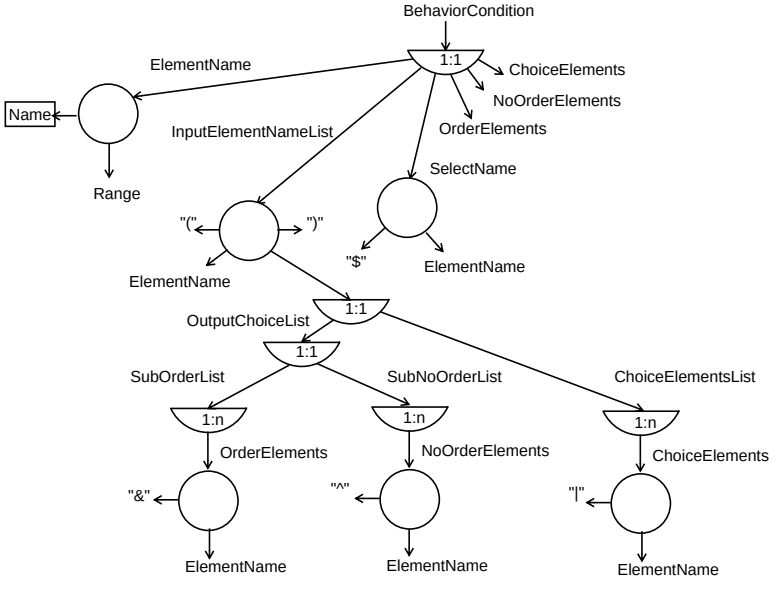
```
FindAttributeValue :: {
  AttributeValue:=Attributes.ListAttributeValue;
  Expression:=Nil;TimeUnit:=Nil;
  While ((AttributeValue#Nil AND Expression=Nil) :
    AttributeDesignator:=AttributeValue.AttributeName.AttributeDesignator;
    TmpType:=TypeOf(AttributeDesignator);
    Case ((TmpType = SearchAttributeDesignator AND
      AttributeValue.AttributeName.OptElementName.Name=Name):
      Expression:=AttributeValue.Expression;
      TimeUnit:=TypeOf(AttributeValue.OptTimeUnit);
    | Else : AttributeValue:=AttributeValue.nextFriend;
    );
  );
}
```

L'instruction prédéfinie FindActivityDescription recherche la description interne d'une activité dans l'ensemble BehaviorDefinition de la description comportementale de l'élément actif concerné. Si la recherche échoue, l'activité est une activité élémentaire. Il faut alors retrouver l'attribut 'Time pour une évaluation des performances ou inclure le code de l'algorithme pour une simulation interprétée ou une synthèse.

-C- Analyse d'une condition d'évolution

Une condition d'évolution s'élabore à partir des entrées de l'activité ou de la fonction et des éléments de relation internes à la description comportementale. Les opérateurs de composition sont le ET séquentiel (&) qui impose une relation d'ordre strict d'apparition, le ET logique (^) sans ordre d'apparition, le OU logique (!) et la sélection ($).

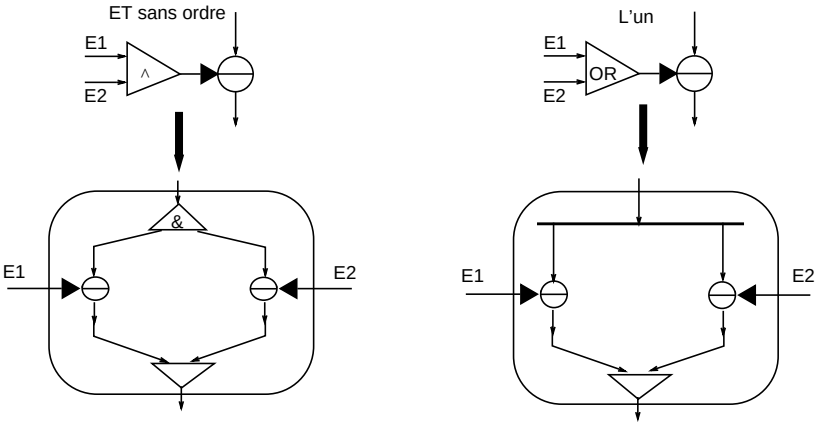
Le méta-modèle de structure de données pour la description des conditions d'évolution est le suivant.



-Figure 6.14- Structure de données de la grammaire pour les conditions.

Pour réutiliser ce qui a été fait du point de vue de l'analyse et de la génération pour le parallélisme et l'attente conditionnelle, il est préférable de convertir le ET sans ordre sous la

forme d'une concurrence et le OU logique sous la forme d'une attente conditionnelle comme le montre la figure ci-dessous.



-Figure 6.15- Conversion du ET sans ordre et du OU logique.

Lorsque l'élément de composition de conditions ne comporte qu'un vecteur d'entrée, la plupart des générateurs de code traduiront cette construction sous une forme particulière.

L'algorithme d'analyse d'une condition d'évolution distingue donc ce cas par rapport au cas général. Le code script de l'algorithme est le suivant:

```
AnalyzeBehaviorCondition :: {
  LocalVisibility ConcurrentActivity,AlternateConditionalActivity,GuardedCondition;
  Vector:=0;
  IsInput:=1;
  TmpType:=TypeOf(BehaviorCondition);
  Case(TmpType = 'ElementName' :
    ElementName:=BehaviorCondition.ElementName;
    AnalyzeInputOutputName;
  | TmpType = 'InputElementNameList'  :
    InputChoiceList:=BehaviorCondition.InputElementNameList.InputChoiceList;
    TmpType:=TypeOf(InputChoiceList);
    Case(TmpType='OrderElements' :
      SubOrderList:=InputChoiceList.OutputChoiceList.SubOrderList;
      ElementName:=BehaviorCondition.InputElementNameList.ElementName;
      AnalyzeInputOutputName;
      ForEach(SubOrderList : ElementName:=SubOrderList.ElementName;
        AnalyzeInputOutputName;
      );
    | TmpType = 'NoOrderElements'  :
      SubNoOrderList:=InputChoiceList.OutputChoiceList.SubNoOrderList;
      ElementName:=BehaviorCondition.InputElementNameList.ElementName;
      ConcurrentActivity:=CreateNode(Mcse,ConcurrentActivity);
      ConditionalActivity:=CreateNode(Mcse,ConditionalActivity);
      ConditionalActivity.BehaviorCondition.ElementName:=CopyDs(ElementName);
      ConcurrentActivity.Activity.ConditionalActivity:=ConditionalActivity;
      ForEach(SubNoOrderList : ElementName:=SubNoOrderList.ElementName;
        ParActivity:=CreateNode(Mcse,ParActivity);
        ConditionalActivity:=CreateNode(Mcse,ConditionalActivity);
        ConditionalActivity.BehaviorCondition.ElementName:=CopyDs(ElementName);
        ParActivity.Activity.ConditionalActivity:=ConditionalActivity;
        AddLast(ConcurrentActivity.ParActivityList,ParActivity);
      );
      AnalyzeConcurrentActivity;
      DelNode(ConcurrentActivity);
    | TmpType = 'ChoiceElements'  :
      ChoiceElementsList:=InputChoiceList.ChoiceElementsList;
      ElementName:=ChoiceElementsList.ElementName;
      AlternateConditionalActivity:=CreateNode(Mcse,AlternateConditionalActivity);
      GuardedCondition:=CreateNode(Mcse,GuardedCondition);
      GuardedCondition.BehaviorCondition.ElementName:=ElementName;
      AlternateConditionalActivity.GuardedCondition:=GuardedCondition;
      ChoiceElementsList:=ChoiceElementsList.nextFriend;
      ForEach(ChoiceElementsList : ElementName:=ChoiceElementsList.ElementName;
        NextGuardedCondition:=CreateNode(Mcse,NextGuardedCondition);
        NextGuardedCondition.GuardedCondition:=CopyDs(GuardedCondition);
        NextGuardedCondition.GuardedCondition.BehaviorCondition.ElementName:=ElementName;
        AddLast(AlternateConditionalActivity.GuardedConditionList,NextGuardedCondition);
      );
      AnalyzeAlternateConditionalActivity;
      DelNode(AlternateConditionalActivity);
```

```
    );
  | TmpType = 'SelectName'  : ElementName:=BehaviorCondition.SelectName.ElementName;
     Vector:=1;
     AnalyzeInputOutputName;
  | TmpType = 'OrderElements'  : ElementName:=BehaviorCondition.OrderElements.ElementName;
     GenerateFromNoOrderElementsVector;
  | TmpType = 'NoOrderElements'  : ElementName:=BehaviorCondition.NoOrderElements.ElementName;
     GenerateFromOrderElementsVector;
  | TmpType = 'ChoiceElements'  : ElementName:=BehaviorCondition.ChoiceElements.ElementName;
     GenerateFromChoiceElementsVector;
  );
}
```

Pour une condition (ou une action), il faut également retrouver l'interface du composant ou l'élément de relation interne correspondant. On utilise pour cela le champ LinkUp du record ElementName associé à la condition.

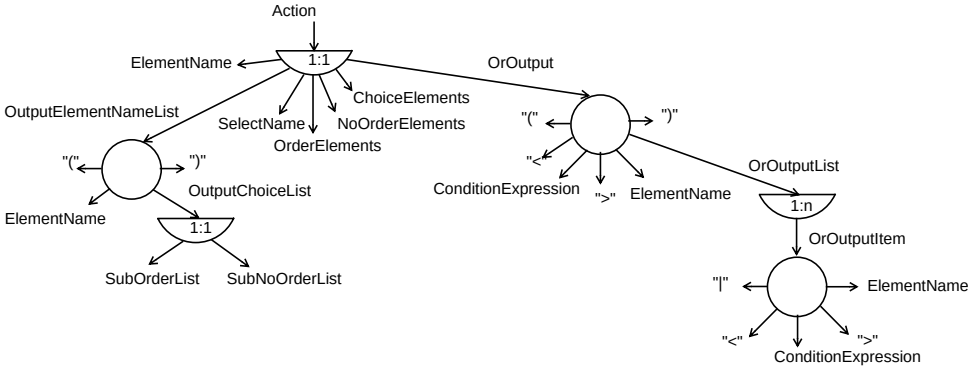
```
AnalyzeInputOutputName :: {
  TmpType:=TypeOf(ElementName.linkUp.InputOutput);
  Case(TmpType = 'InputOutput' : InputOutput:=ElementName.linkUp.InputOutput;
    GetAttributes(InputOutput,AttributesList);
    BlackBall:=0;
    TmpType:=TypeOf(InputOutput.ParameterDef);
    Case(TmpType = 'TypedParameter' :
      ParameterName:=InputOutput.ParameterDef.TypedParameter.ElementName.Name;
      ParameterType:=TypeOf(InputOutput.ParameterDef.TypedParameter.TypeRelation);
      TypeName:=InputOutput.ParameterDef.TypedParameter.Name;
      Range:=InputOutput.ParameterDef.TypedParameter.ElementName.Range;
    | Else :
      ParameterName:=InputOutput.ParameterDef.UntypedParameter.ElementName.Name;
      ParameterType:='Ev';
      TypeName:=Null;
      Range:=InputOutput.ParameterDef.UntypedParameter.ElementName.Range;
    );
  | Else : InternalRelations:=ElementName.linkUp.InternalRelations;
    GetAttributes(InternalRelations,AttributesList);
    TmpType:= TypeOf(InternalRelations);
    Case(TmpType='InternalInfoDeclaration':
      TypedRelationDef:=InternalRelations.InternalInfoDeclaration .TypedRelationDef;
      ParameterType:='Mess';
      Range:=TypedRelationDef.Range;
      ParameterName:=TypedRelationDef.Name;
      TypeName:=TypedRelationDef.SecondName;
    | TmpType = 'InternalDataDeclaration' :
      TypedRelationDef: = InternalRelations.InternalDataDeclaration.TypedRelationDef;
      ParameterType:='Var';
      Range:=TypedRelationDef.Range;
      ParameterName:=TypedRelationDef.Name;
      TypeName:=TypedRelationDef.SecondName;
    | TmpType = 'InternalEvDeclaration' :
      Range:=InternalRelations.InternalEvDeclaration.Range;
      ParameterName:=InternalRelations.InternalEvDeclaration.Name;
      ParameterType:='Ev';
      TypeName:=Null;
    );
  );
  GetRangeExpression;
  Case (RangeExpression # Null :
    BlackBall:=MultipleComponent;
  );
  GenerateFromInputOutputName;
}
```

Les informations mises à disposition pour la génération d'une condition d'évolution simple sont: son nom, son type, le type de la donnée transmise, la dimension de l'interface ou de l'élément de relation qui lui sont rattachés, ses attributs et l'indicateur BlackBall utilisé pour l'instanciation multiple.

-D- Analyse d'une action

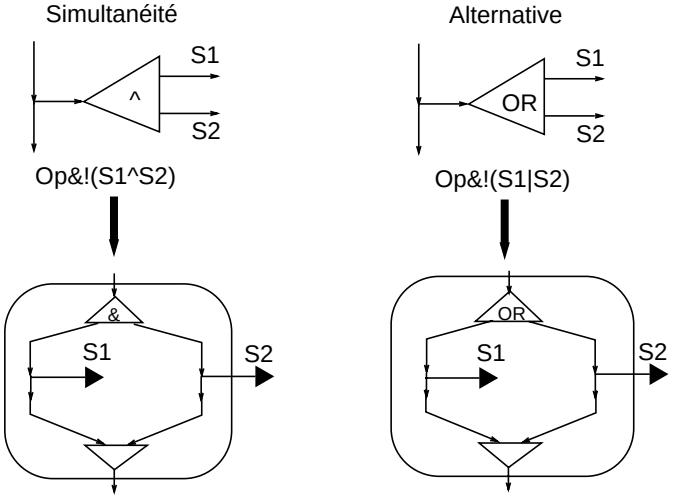
Les actions concernent la génération d'informations ou d'événements par les sorties du composant contenant l'activité ou vers d'autres activités internes au composant par l'intermédiaire d'éléments de relation internes à la description comportementale. Les opérateurs de composition d'actions sont la séquence qui fixe l'ordre de génération, la simultanéité, l'alternative qui discerne une des sorties en fonction d'une condition ou d'une

probabilité et la sélection. Le méta-modèle de structure de données pour la description des actions est le suivant.



-Figure 6.16- Structure de données de la grammaire pour les actions.

Comme pour les conditions d'évolution, on transforme la simultanéité et l'alternative comme indiqué ci-dessous.



-Figure 6.17- Conversion de la simultanéité et de l'alternative.

L'algorithme d'analyse d'une action se déduit facilement par dualité de celui d'une condition d'évolution.

6.4.3 Test de la partie analyse du script

La partie analyse du modèle source est essentielle car elle conditionne la partie génération et donc influe directement sur l'efficacité et la qualité d'implantation du générateur. La difficulté repose ici sur la sélection des informations pertinentes pour la génération et le choix de l'emplacement des appels des règles de génération.

Comme le modèle MCSE sera la source pour d'autres générateurs de code, nous avons essayé d'implanter une partie analyse suffisamment générique pour qu'elle soit commune à tous les générateurs de code du modèle MCSE vers un langage cible quelconque. De toute façon, un concepteur de script pourra optimiser ou compléter cette base à sa guise. Ainsi, il peut concentrer ses efforts sur les règles de la partie génération.

Cette partie d'Analyse a été testée avec l'exemple du serveur vidéo du chapitre 7. Dans ce test, les règles de génération affichent uniquement les informations dont elles disposent.

Le résultat partiel de ce test est présenté ci-dessous.

```
Loading McseModel...
Time use for loading McseModel: 18
PathLevel = ApplicationServeurVideo
GenerateComponentFromNameAndRange with
                ComponentName = ApplicationServeurVideo
                ComponentRangeExpression =
                ComponentMultiple = 0
GenerateOneGenericParameter with
                n:INTEGER=10
GenerateOneGenericParameter with
                k:INTEGER=10
GenerateOneGenericParameter with
                nd:INTEGER=10
GenerateOneGenericParameter with
                TaccDisk:INTEGER=20
GenerateFromRelationElement with
                RelationType = Port
                Name = CmdUsager
                ElementMultiple = 1
                ElementRangeExpression = k
                NumberOfInputAccessInLevel = 1
                NumberOfOutputAccessInLevel = 1
                NumberOfInputAccessInAllLevel = 2
                NumberOfOutputAccessInAllLevel = 2
GenerateFromRelationElement with
                RelationType = Port
                Name = RepUsager
                ElementMultiple = 1
                ElementRangeExpression = k
                NumberOfInputAccessInLevel = 1
                NumberOfOutputAccessInLevel = 1
                NumberOfInputAccessInAllLevel = 2
                NumberOfOutputAccessInAllLevel = 3
GenerateFromRelationElement with
                RelationType = Port
                Name = SequenceIn
                ElementMultiple = 1
                ElementRangeExpression = n
                NumberOfInputAccessInLevel = 1
                NumberOfOutputAccessInLevel = 1
                NumberOfInputAccessInAllLevel = 3
                NumberOfOutputAccessInAllLevel = 2
                AttributesList:
                        `Capacity=0
GenerateFromRelationElement with
                RelationType = Port
                Name = SequenceOut
                ElementMultiple = 1
                ElementRangeExpression = n
                NumberOfInputAccessInLevel = 1
                NumberOfOutputAccessInLevel = 1
                NumberOfInputAccessInAllLevel = 2
                NumberOfOutputAccessInAllLevel = 3
                AttributesList:
                        `Capacity=0
PathLevel = ApplicationServeurVideo\Usagers
GenerateFromInputOutput with
                ParameterType = Mess
                ParameterName = RepUsager
                TypeName = DefRepUsager
                VectorExpression = k
                Vector = 1
                BlackBall = 0
                IsInput = 1
                NumberOfInputAccessInLevel = 1
                NumberOfOutputAccessInLevel = 0 ...
```

-Figure 6.18- Résultat partiel du test de la partie analyse du script.

La figure donne la liste des règles de génération appelées lors de l'analyse du modèle MCSE source et des informations fournies à ces règles. Sur un Pentium 90, le test s'effectue en 135 secondes en mode interprété et 28 en code java. Sur les 28 secondes, 18 sont utilisées par l'analyseur lexical et l'analyseur syntaxique pour le chargement de la structure de données de l'exemple à partir de sa description textuelle. L'algorithme de parcours du modèle MCSE est donc relativement rapide (10 secondes). Pour le script du générateur de VHDL comportemental, la partie analyse représente 31% (1020 lignes) du code total (3253 lignes).

6.5 PARTIE GENERATION DU SCRIPT

La partie génération représente la plus grosse partie du code (69%) car le langage VHDL est un langage très déclaratif et aussi parce que nous avons décidé de garder la hiérarchie du modèle source dans le modèle final (redondance d'information), ceci pour bien exploiter VHDL et favoriser la réutilisation de modèles internes.

6.5.1 Règles de génération

Les règles de génération concernant la composante structurelle du modèle de performance de MCSE sont définies dans le tableau suivant. Pour chaque règle, nous précisons les informations qui lui sont fournies et les opérations effectuées pour la fonction de génération.

Règles de Génération appelées lors de l'analyse	Opérations effectuées par le générateur VHDL
GenerateComponentFromNameAndRange *Informations fournies*: ComponentName ComponentRangeExpression ComponentMultiple, AttributesList	Si premier composant généré alors mise à jour des noms de l'entité et du package et leurs contextes Sinon création d'un block multiple et/ou simple Si l'attribut 'Concurrency est limité alors instanciation d'un composant ordonnanceur et déclaration des signaux associés

Règles de Génération appelées lors de l'analyse	Opérations effectuées par le générateur VHDL
GenerateFirstGenericParameter *informations fournies*: ParameterName, GenericParameterType GenericParameterDefaultValue	Mise à jour de la clause générique de l'entité ou d'un block.
GenerateOneGenericParameter	Identique à GenerareFirstGenericParameter
GenerateNoGenericParameters	Destruction de la clause générique
GenerateFromInputOutput *informations fournies:* ParameterType, ParameterName ParameterModelName, TypeName VectorExpression, Vector BlackBall, IsInput, NumberOfIOSubLevel	Mise à jour du Port et Port Map de l'entité ou du block. Déclaration ou mise à jour du type, signal et alias associés à l'interface du block.
GenerateFromRelationElement *informations fournies:* RelationType, Name, ElementMultiple ElementRangeExpression NumberOfAccessInLevel, AttributesList	Instanciation du composant implantant l'élément de relation. Mise à jour de la clause générique du composant en fonction des attributs de l'élément de relation. Mise à jour de la clause port map du composant et déclaration des types et signaux associés.

Les règles de génération concernant la composante comportementale du modèle de performance de MCSE sont données dans le tableau suivant.

Règles de Génération appelées lors de l'analyse	Opérations effectuées par le générateur VHDL
GenerateFromFirstCondActivity *informations fournies:* Expression, ComparisonOp, SecondExpression	Appel de la procédure de génération de nombres aléatoires et génération de la branche "IF" de la construction VHDL "IF... ELSIF... ELSE..."
GenerateFromCondActivity *informations fournies:* Expression, ComparisonOp, SecondExpression	Rajout d'une branche "ELSIF..." dans la construction "IF... ELSIF... ELSE...".
GenerateFromLastCondActivity	Rajout d'une branche "ELSE..." dans la construction "IF... ELSIF... ELSE...".
GenerateBeginTaskSynchronisation *information fournie*: NbTask	Appel des procédures Fork/WaitFork.
GenerateEndTaskSynchronisation *information fournie*: NbTask	Appel des procédures WaitJoin/Join.
GenerateFromParActivity *information fournie*: NbTask	Déclaration et initialisation d'un nouveau process.
GenerateInfiniteLoop	Génération de la construction "LOOP... END LOOP;"
GenerateFiniteLoopExpression *information fournie:* *Expression*	Génération de la construction "FOR j IN 1 TO *Expression* LOOP ... END LOOP;"
GenerateFiniteUnLoop *information fournie:* *Expression*	Génération de la construction "IF (cpt=0) THEN cpt := *Expression*; ... ELSE cpt := cpt - 1; END IF;" et déclaration de la variable cpt.

Règles de Génération appelées lors de l'analyse	Opérations effectuées par le générateur VHDL
GenerateConditionalLoop *informations fournies:* Expression, ComparisonOp, SecondExpression	Génération de la construction "WHILE (Expression ComparisonOp SecondExpression) LOOP ... END LOOP;"
GenerateFromFirstGuardedCondition *information fournie*: InputName	Déclaration du signal ProtocolVector et de la variable BranchNumber. Génération de: - l'assignation concurrente du signal ProtocolVector - l'appel des primitives InitConditionnalActivation, WaitConditionnalActivation et ResetConditionalActivation - La première branche WHEN ... de la construction CASE
GenerateFromGuardedCondition *information fournie*: InputName	Appel aux primitives InitConditionnalActivation, WaitConditionnalActivation et ResetConditionalActivation Rajout d'une branche WHEN à la construction CASE.
GenerateFromLastGuardedCondition	Mise à jour de la dimension du signal ProtocolVector.
GenerateElementaryActivity *informations fournies*: Name, Expression, TimeUnit	Appel de la primitive Delay et déclaration d'une constante ou d'une variable pour l'expression du temps. Appel la primitive TaskPriority si l'attribut 'Priority est défini et si la concurrence est limitée.
GenerateFromIncludeFile *information fournie*: FileName	Le fichier contenant l'algorithme de l'opération élémentaire est inclu par référence.
GenerateFromNoOrderElementsVector	Pas implanté pour l'instant.
GenerateFromInputOutputName *informations fournies*: ParameterType, ParameterName TypeName, BlackBall IsInput, AttributesList IsInternalElement, LocalAtttributes	Appel de la primitive d'accès à l'élément de relation. Déclaration de la variable contenant l'information à transmettre ou recue. Les attributs locaux sont utilisés pour surchager ceux de l'élément de relation. Mise à jour des records TypeDefInformation et TypeDefData du package.
GenerateFromActivityInstance *informations fournies*: ActivityName, ActivityRangeExpression ActivityMultiple, AttributesList	création d'un block multiple et/ou simple.
GenerateFromBehaviorInstance *information fournie*: BehaviorName	Déclaration d'un nouveau process.
GenerateExit	Pas implanté pour l'instant
UpdateGenerationOfComponentModel *informations fournies*: NbTask, ComponentRangeExpression	Mise à jour du paramètre générique des composants de relation du niveau représentant le nombre d'accès du niveau à ce composant. Rajout de l'interface StateVector dans le port et leport map du block si la concurrence est limitée. Gestion des indices du signal StateVector en fonction du nombre de process(tâches) du niveau.

6.5.2 Difficultés rencontrées

Certaines constructions du modèle MCSE et leurs principes de transcription en VHDL posent des difficultés pour la génération automatique de code. On peut citer en autres:

- le problème de connexion entre éléments de relation et composants actifs qui a été déjà présenté dans le chapitre 4 consacré aux règles de transcription du modèle de performance en VHDL,

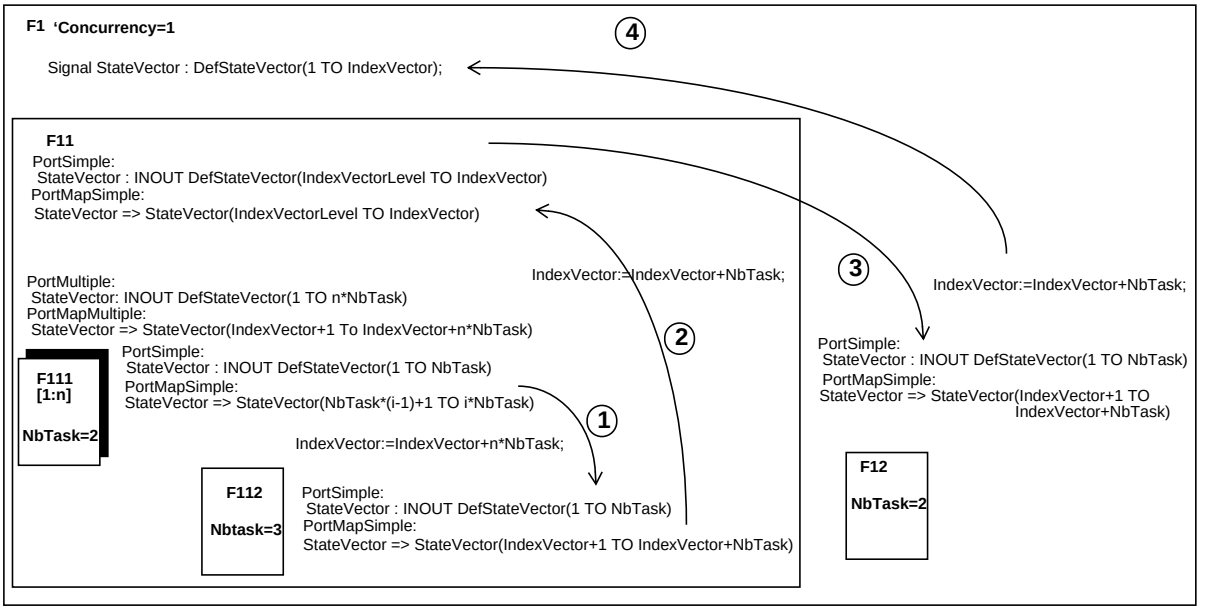
- le problème de la gestion des indices du vecteur d'état associé à un ordonnanceur,

- le problème du partage d'informations entre différentes branches d'un parallélisme.

-A- Gestion des indices du vecteur d'état des tâches

La limitation du degré de concurrence d'un élément actif par l'attribut 'Concurrency nous oblige à instancier un composant ordonnanceur et à gérer les indices d'un vecteur regroupant l'état des tâches (process VHDL) qui peuvent être situées à des niveaux hiérarchiques différents.

Le principe de gestion des indices du vecteur d'état est schématisé ci-dessous.



-Figure 6.19- Principe de gestion des indices du vecteur d'états des tâches.

Lors du parcours descendant du modèle source, on empile à chaque niveau hiérarchique la valeur courante de l'indice IndexVector dans la variable IndexVectorLevel. Puis lors du parcours ascendant, comme on connaît alors le nombre de tâches du niveau (analyse du modèle de comportement de la fonction terminée), on met à jour le port et le port map des blocks VHDL concernés et la dimension du signal StateVector.

-B- Partage des informations entre différentes branches d'un parallélisme

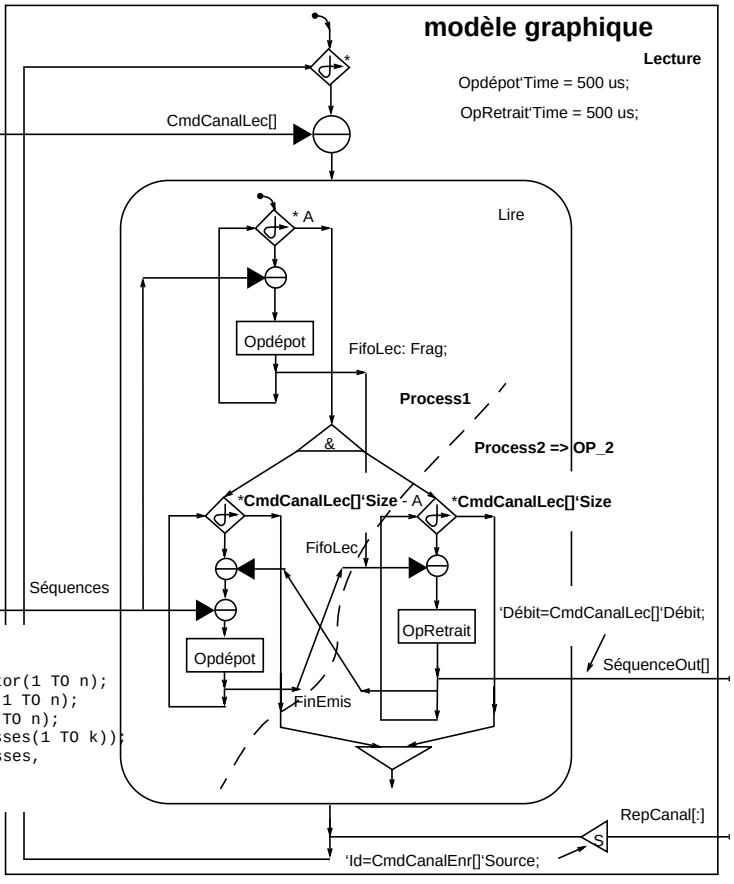
L'utilisation des attributs d'une information reçue par des branches différentes d'une concurrence nécessite de déplacer la déclaration de la variable utilisée pour stocker l'information reçue de la partie déclarative d'un process vers la partie déclarative du block et de déclarer cette variable sous la forme d'une "shared variable" (VHDL'93).

La fonction Lecture de l'exemple du serveur vidéo temps réel illustre ce cas. La variable InfCmdCanalLec liée à l'entrée CmdCanalLec[] est déclarée comme shared variable au niveau

du block Lecture. Elle est en effet utilisée par les process Lecture et OP_2 comme le montre le code suivant.

code source

```
<Component> [1:n] Lecture
 (In Mess CmdCanalLec[] : DefCmdCanalLec;
  InOut Var Sequences : DefSequences;
  Out Mess SequenceOut[] : DefSequenceOut;
  Out Mess RepCanal[:] : DefRepCanal;);
 <Attributes>
  SequencesOut[]`Debit = CmdCanalLec[]`Debit;
  RepCanal[]`Id = CmdCanalLec[]`Source;
  RepCanal[]`Size = 8;
  RepCanam[:]`Path = `Id;
  OpDepot`Time = 500 us;
  OpRetrait`Time = 500 us;
 <EndAttributes>
 <Behavior> BehaviorLecture;
  <InfoLink> FifoLec : Frag;
   <Attributes>
    `Capacity = A;
   <EndAttributes>
   <EndInfoLink>
  <EvLink> FinEmis;
  <EndEvLink>
  Lecture::{?CmdCanalLec[]&Lire&!$RepCanal[:]}*;
  Lire::({?Sequences&OpDepot&!FifoLec}*A)&
        (({?FinEmis&?Sequences&OpDepot&!FifoLec}
        *CmdCanalLec[]`Size - A)||
        ({?FifoLec&OpRetrait&!SequenceOut[]&
         !FinEmis}*CmdCanalLec[]`Size));
<EndBehavior>
<EndComponent>
```



code généré

```
MultipleLecture :
BLOCK
  PORT (SIGNAL CmdCanalLec : INOUT DefPortOutVector(1 TO n);
    SIGNAL Sequences : INOUT DefDataAccessVector(1 TO n);
    SIGNAL SequenceOut : INOUT DefPortInVector(1 TO n);
    SIGNAL RepCanal : INOUT DefRepCanalInputAccesses(1 TO k));
  PORT MAP (CmdCanalLec => CmdCanalLecOutputAccesses,
    Sequences => LectureSequencesAccesses,
    SequenceOut => SequenceOutInputAccesses,
    RepCanal => RepCanalInputAccesses);
  BEGIN
  MultipleInstantiationLecture :
    FOR i IN 1 TO n GENERATE
    Lecture :
    BLOCK
    PORT (SIGNAL CmdCanalLec : INOUT DefPortOut;
      SIGNAL Sequences : INOUT DefDataAccess;
      SIGNAL SequenceOut : INOUT DefPortIn;
      SIGNAL RepCanal : INOUT DefRepCanalInputAccesses(1 TO k));
     PORT MAP (CmdCanalLec => CmdCanalLec (i),
       Sequences => Sequences (i),
       SequenceOut => SequenceOut (i),
       RepCanal => RepCanal);
 CONSTANT me : INTEGER := n + i;
   SIGNAL FifoLecInputAccesses : DefPortIn;
   ...
   SIGNAL OpStatus_1 : DefForkJoinStatus := InitForkJoinStatus;
   SHARED VARIABLE InfCmdCanalLec : DefInformation;
 BEGIN
     FifoLec : COMPONENT PortObject
       GENERIC MAP (NbUsersInput => 1, NbUsersOutput => 1,
         ...
         Capacity => A,
         WriteTime => NullTime
         ReadTime => NullTime)
       PORT MAP (InputAccesses (1) => FifoLecInputAccesses,
         OutputAccesses (1) => FifoLecOutputAccesses, ...);
     FinEmis : COMPONENT EveObject
       GENERIC MAP (NbUsersInput => 1,
         NbUsersOutput => 1,
         ...
         ReadTime => NullTime)
       PORT MAP (InputAccesses (1) => FinEmisInputAccesses,
         OutputAccesses (1) => FinEmisOutputAccesses, ...);
     Lecture : PROCESS
         VARIABLE InfSequences : DefData;
         VARIABLE InfFifoLec : DefInformation;
         VARIABLE InfRepCanal : DefInformation;
         CONSTANT OpdepotTime : TIME := 500 us;
     BEGIN
       OpStatus_1 <= InitForkJoinStatus;
       LOOP
         receive (CmdCanalLec, InfCmdCanalLec, NullTime, 0, now,
                  FALSE);
         FOR j_1 IN 1 TO A LOOP
           ReadSharVar (Sequences, InfSequences, NullTime,
                        NullTime, 0, now);
         Delay(OpdepotTime);
         send (InfFifoLec, FifoLecInputAccesses, NullTime, 0
               now, FALSE);
       END LOOP;
     Fork (OpStatus_1);
     FOR j_2 IN 1 TO InfCmdCanalLec.Size - A LOOP
       WaitEv (FinEmisOutputAccesses, NullTime, 0, now);
       ReadSharVar (Sequences, InfSequences, NullTime, NullTim
                    0, now);
       Delay(OpdepotTime);
       send (InfFifoLec, FifoLecInputAccesses, NullTime, 0, no
            FALSE);
     END LOOP;
     WaitJoin (OpStatus_1);
     InfRepCanal.Id := InfCmdCanalLec.Source;
     InfRepCanal.Size := 8;
     send (InfRepCanal, RepCanal, InfRepCanal.Id, me, NullTime
          now, FALSE);
 END LOOP;
 END PROCESS Lecture;
     OP_2 : PROCESS
       VARIABLE InfFifoLec : DefInformation;
         VARIABLE InfSequenceOut : DefInformation;
         CONSTANT OpRetraitTime : TIME := 500 us;
     BEGIN
         WaitFork (OpStatus_1);
         FOR j_1 IN 1 TO InfCmdCanalLec.Size LOOP
           receive (FifoLecOutputAccesses, InfFifoLec,NullTime
                    0, now, FALSE);
           Delay(OpretraitTime);
           InfSequenceOut.Debit := InfCmdCanalLec.Debit;
           send (InfSequenceOut, SequenceOut, NullTime, 0, now
                TRUE);
           SignalEv (FinEmisInputAccesses, NullTime, 0, now);
         END LOOP;
         Join (OpStatus_1);
     END PROCESS OP_2;
   END BLOCK Lecture;
  END GENERATE MultipleInstantiationLecture;
 END BLOCK MultipleLecture;
```

-Figure 6.20- Test du générateur avec la fonction lecture du serveur vidéo.

6.6 TEST DU GENERATEUR OBTENU

Le générateur ainsi développé a été testé sur deux exemples qui sont décrits dans le chapitre suivant. Il s'agit d'un serveur vidéo temps-réel et d'un système de communication basé sur un ensemble de cartes identiques. Ces deux exemples ont été très utiles pour mettre au point le générateur car ils regroupent la plupart des constructions possibles (alternative, parallélisme, attente conditionnelle, concurrence limitée...) d'un modèle source MCSE.

Le tableau suivant donne pour chaque exemple la taille en K octets du code source MCSE, la taille du code VHDL généré (entité uniquement), le temps de génération en mode interprété et le temps de génération en mode compilé (transcription du script en code Java). La machine utilisée pour les tests est un PC PENTIUM PRO 200 Mhz avec 32 Mo de mémoire vive. Le compilateur Java (ou plutôt dans ce cas machine virtuelle) utilisé est le compilateur "Just In Time" de Symantec (VisualCafe 1.1).

Système	taille du code source Mcse	taille du code VHDL généré	Temps en mode interprété	Temps en mode compilé
ComSys	4.28 Ko	28.6 Ko	462 s	228 s
ServeurVideo	5.45 Ko	31 Ko	492 s	245 s

Le temps de génération qui est de l'ordre de 4 mn, peut sembler pénalisant si le parcours du domaine des solutions possibles d'un partitionnement logiciel/matériel nécessite un nombre important de générations de code. En réalité, la recherche de la solution optimale ne nécessitera pas forcément plusieurs générations lorsqu'il y a utilisation judicieuse des paramètres génériques et le temps de génération sera certainement négligeable par rapport au temps de simulation. Pour l'instant, aucune étude d'optimisation du code n'a donc été effectuée. On peut cependant réduire le temps de génération en utilisant un compilateur Java natif mais on perd alors la portabilité.

Le tableau suivant permet de comparer le générateur VHDL comportemental avec les autres générateurs développés avec l'outil MetaGen. Il s'agit d'un générateur de programme C utilisant l'exécutif temps-réel VxWorks et un générateur de programme VHDL synthétisable (niveau RTL). Le tableau comporte la taille du code script, la taille du code source MCSE utilisé comme exemple, la taille des templates utilisés par les générateurs, la taille du code généré, le temps de génération en mode interprété, la taille du code Java obtenu par transcription du script et le temps d'exécution du programme Java obtenu.

Generateur	Script	Mcse	Template	Sortie	Interprété	Java	Compilé
CVxworks	150 Ko	6.33 Ko	2.23 Ko	3.2 Ko	63 s	660 Ko	23 s
VhdlSyn	194 Ko	6.93 Ko	18.6 Ko	21.7 Ko	111 s	669 Ko	83 s
VhdlPerf	176 Ko	5.45 Ko	27 Ko	31 Ko	492 s	827 Ko	245 s

Ce tableau met en évidence que le temps de génération du générateur de code pour l'évaluation des performances est beaucoup plus long (rapport 3) que celui des générateurs de code pour la synthèse logicielle et matérielle. Cette différence de temps est due:

- aux différences de complexité des modèles sources,
- aux différences de complexité des modèles de sortie générés.

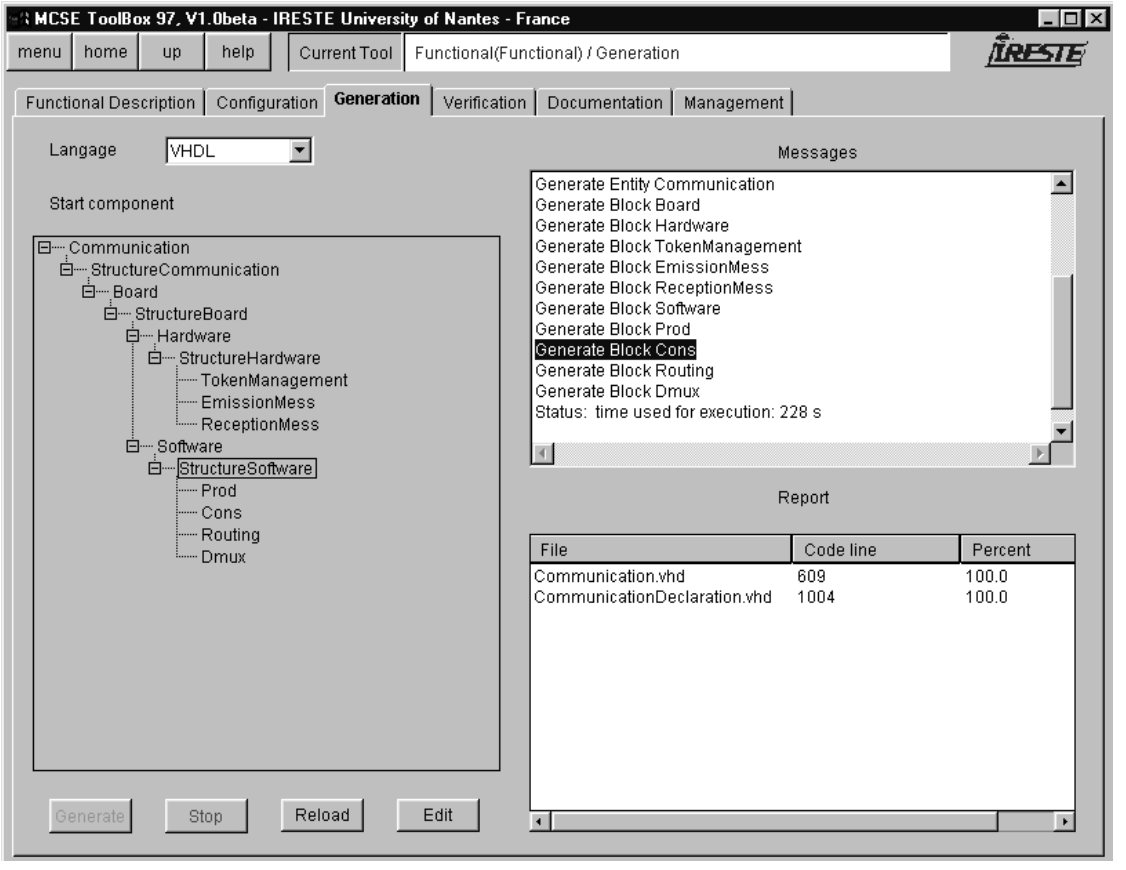
Pour le générateur VHDL synthétisable, au niveau structurel, les combinaisons problématiques de l'instanciation multiple (K->M->N) disparaissent par raffinement (fonction d'arbitrage de bus par exemple). Au niveau comportemental, ce générateur de code synthétisable n'accepte que des comportements de fonctions qui sont séquentielles et déterministes.

Le code de sortie du générateur de code synthétisable est également pour l'instant plus simple:

- Il n'y a pas de problèmes liés aux limitations sur la généricité des types puisqu'il travaille uniquement avec des bits (niveaux logiques),
- un port ne peut avoir qu'un seul producteur et un seul consommateur,
- un événement ne peut avoir qu'un seul producteur,
- une variable partagée peut avoir plusieurs accès simultanés mais obligatoirement du même niveau hiérarchique. Pour des accès à une variable partagée à des niveaux hiérarchiques différents, on instancie une interface (décodeur de priorité). La priorité de l'accès devient locale au niveau mais comme il ne tient pas compte de l'attribut 'Concurrency ceci n'a aucune importance.
- Le générateur n'a pas à gérer les activités concurrentes et la limitation du degré de concurrence d'une ressource ainsi que leurs problèmes sous-jacents.

6.7 INTERFACE UTILISATEUR DE L'OUTIL OBTENU

Le générateur obtenu est intégré dans une plate-forme d'outils en cours de développement nommée McseTools. L'interface utilisateur du générateur est représentée par la figure 6.21.



-Figure 6.21- Face avant du générateur de programme VHDL comportemental.

L'interface utilisateur comporte trois zones principales:

- une zone de sélection du composant de départ pour la génération. Pour une évaluation des performances, on générera plutôt le système complet et son environnement pour ne pas être obligé de définir de stimuli,
- une zone de messages permettant aux générateurs d'afficher des informations durant la génération de code (instructions Display et Warning du script),
- une zone de rapport contenant la liste des fichiers générés avec leur nombre de lignes de code et le pourcentage de code généré automatiquement.

L'outil peut être utilisé:

- durant la phase de conception fonctionnelle: vérification fonctionnelle et/ou allocation d'exigences sur les constituants du système (taille d'un port, temps d'exécution max,...) utiles pour la conception architecturale,
- durant la phase de conception architecturale: aide au partitionnement logiciel/matériel (co-simulation non-interprétée).

6.8 AMELIORATIONS

Les améliorations possibles à court terme concernent la réduction du temps de génération du code VHDL, le mode de couplage entre le simulateur du modèle de performance et l'outil d'analyse de trace et la transcription automatique de VHDL vers C ou réciproquement des algorithmes des activités élémentaires du modèle de performance d'un système.

6.8.1 La réduction du temps de génération du code VHDL

La génération de code VHDL comportemental avec l'outil développé est relativement lente. Une analyse du code Java produit avec un outil de "profiling" tel que Visual Quantify de Pure Atria permettrait de trouver les méthodes gourmandes en temps C.P.U. et d'optimiser le temps d'exécution du générateur. A priori, le générateur consomme beaucoup de temps à rechercher avant de déclarer un élément si la déclaration de ce dernier (type, signal, variable, procédure) n'a pas déjà été générée. Au lieu de parcourir plusieurs fois des listes de la structure de données interne, il semble plus judicieux d'offrir au niveau du Script la possibilité d'utiliser des hashtables. Comme les outils ont été implantés en Java, cette extension du Script ne pose aucune difficulté.

Le parcours du modèle MCSE nécessite un parcours descendant pour respecter la hiérarchie du modèle source et un parcours ascendant qui permet d'utiliser les informations recueillies à un niveau donné pour la mise à jour du niveau supérieur. Si le modèle MCSE source l'autorise, il est préférable de laisser le choix au concepteur de générer un modèle VHDL hiérarchique ou un modèle à plat. Un modèle à plat s'obtient par algorithme de parcours du modèle MCSE du type ascendant (Bottom-Up) qui est plus rapide que celui utilisé pour générer un modèle hiérarchique.

6.8.2 Couplage avec l'outil d'analyse de trace

Le couplage entre le couple générateur/simulateur VHDL et l'outil d'analyse de trace est pour l'instant du type hors ligne. Les informations nécessaires pour la génération de la trace sont contenues dans le modèle MCSE source sous la forme d'attributs 'Probe et transformées

sous la forme d'une table de codage par le générateur de code. Lors de la simulation, la trace générée ne contient que des occurrences de trace dont les codes sont ceux définis dans la table de codage. Avec cette approche, le contenu de la trace ne peut pas être modifié en direct par l'outil d'analyse des performances. Pour avoir un couplage en ligne entre le simulateur VHDL et l'outil d'analyse de trace, il est possible de déclarer automatiquement une table de codage complète et de ne sortir par le simulateur que les occurrences de trace dont les codes sont nécessaires pour calculer les indices de performances définis au niveau de l'outil d'analyse de trace. La procédure de génération d'une occurrence de trace doit alors être asservie à l'état d'un indicateur qui sera positionné par le simulateur en fonction des ordres provenant de l'outil d'analyse de trace. Le couplage entre le simulateur et l'outil d'analyse de trace peut s'effectuer grâce à l'interface Foreign Language Interface (VHDL'93) et à un mécanisme de communication inter-outils. Le mécanisme de communication (socket par exemple) doit être capable de relier le simulateur VHDL s'exécutant sur une machine sous unix et l'outil d'analyse de trace qui fonctionne dans l'environnement Windows 95/NT.

6.8.3 Transcription VHDL en C et C en VHDL

Avec les générateurs obtenus, pour faire une vérification fonctionnelle, le concepteur doit saisir manuellement le code de chaque opération élémentaire du modèle de comportement des fonctions non raffinées de la description fonctionnelle. Lorsque la conception est avancée, la description fonctionnelle est très détaillée et le comportement des fonctions est purement séquentiel. La description peut alors être synthétisée. Pour qu'une modification de l'allocation d'une fonction au niveau du partitionnement matériel/logiciel avec passage d'une implantation matérielle en logiciel ou réciproquement, ne nécessite par une nouvelle saisie de code des opérations, il faut transcrire automatiquement une description VHDL séquentielle en C et réciproquement. Cette translation peut facilement être réalisée avec l'outil MetaGen pour lequel on dispose déjà des grammaires C et VHDL. La syntaxe des constructions VHDL séquentielles et celle de leurs équivalences en C étant proches, il est aussi possible de faire une transcription directe lors de l'analyse syntaxique sans utiliser de structures de données intermédiaires [PARKINSON-94].

6.9 CONCLUSIONS

Ce chapitre a présenté l'implantation du générateur de code permettant de transcrire la description textuelle MCSE d'un système en une description VHDL comportementale. L'outil obtenu permet de faire:

- une vérification fonctionnelle. L'utilisateur doit alors entrer les algorithmes des opérations élémentaires (modèle interprété). Comme le langage VHDL est très déclaratif, le générateur permet de générer 60-80% du code automatiquement.
- une évaluation des performances pour laquelle le code est entièrement généré automatiquement. Il s'agit en effet d'un modèle non-interprété où seuls comptent les temps des opérations et les dépendances temporelles.

Le principe de génération repose sur l'utilisation d'un analyseur syntaxique et d'un template. L'obtention d'un analyseur syntaxique nécessite de saisir la grammaire du langage voulu dans le formalisme du générateur d'analyseur syntaxique. Le template est un fichier contenant sous leur forme la plus complète les constructions du langage cible qui sont utiles pour la

transcription. La génération consiste alors à effectuer un parcours ordonné de la structure de données d'entrée obtenue avec l'analyseur syntaxique et à générer la structure de données de sortie à partir des constructions contenues dans le template. L'ensemble de ces opérations de manipulations de structures de données est décrit sous la forme d'un script. Ce script sert de point d'entrée au méta-générateur MetaGen qui permet d'interpréter le script ou de le transcrire en code Java. Une fois transcrit en code java, le générateur obtenu reste un outil multi plate-forme (Unix, PC, Mac...).

Le script se compose d'une partie analyse du modèle source et d'une partie génération. La partie analyse extrait les informations pertinentes et appelle les règles de la partie génération. Si plusieurs générateurs utilisent le même modèle source, il faut essayer d'obtenir une partie Analyse suffisamment générique pour qu'elle soit commune à tous les générateurs. Comme le modèle MCSE servira d'entrée à d'autres générateurs que le générateur VHDL, nous avons donc détaillé dans ce chapitre la partie analyse du modèle MCSE. Le concepteur d'un nouveau script pourra ainsi l'utiliser tel quel ou optimiser et compléter cette base à sa guise et concentrer ses efforts sur les règles de la partie génération. La partie Génération est spécifique à chaque générateur. Les règles de transcription qui seront à priori décrites en langage naturel doivent être converties en règle de script.

La technologie de génération utilisée permet:

- de réduire le temps de développement des générateurs qui peuvent être developpés de manière incrémentale,
- de faciliter la maintenance des générateurs: la taille du code script est plus petite et plus lisible que celle d’un code équivalent en langage de programmation classique (C++, JAVA...).

Un générateur de code C++ pour l’évaluation des performances est également en cours de développement. Cette solution basée sur l’exécution d’un programme C++ est plus avantageuse que l’utilisation du couple VHDL/simulateur. L’intérêt est que l’exécution du modèle obtenu est beaucoup plus rapide que la simulation du modèle VHDL (rapport de 3 à 4), et ceci permet donc de parcourir plus rapidement l’espace des solutions possibles d’un partitionnement matériel/logiciel.