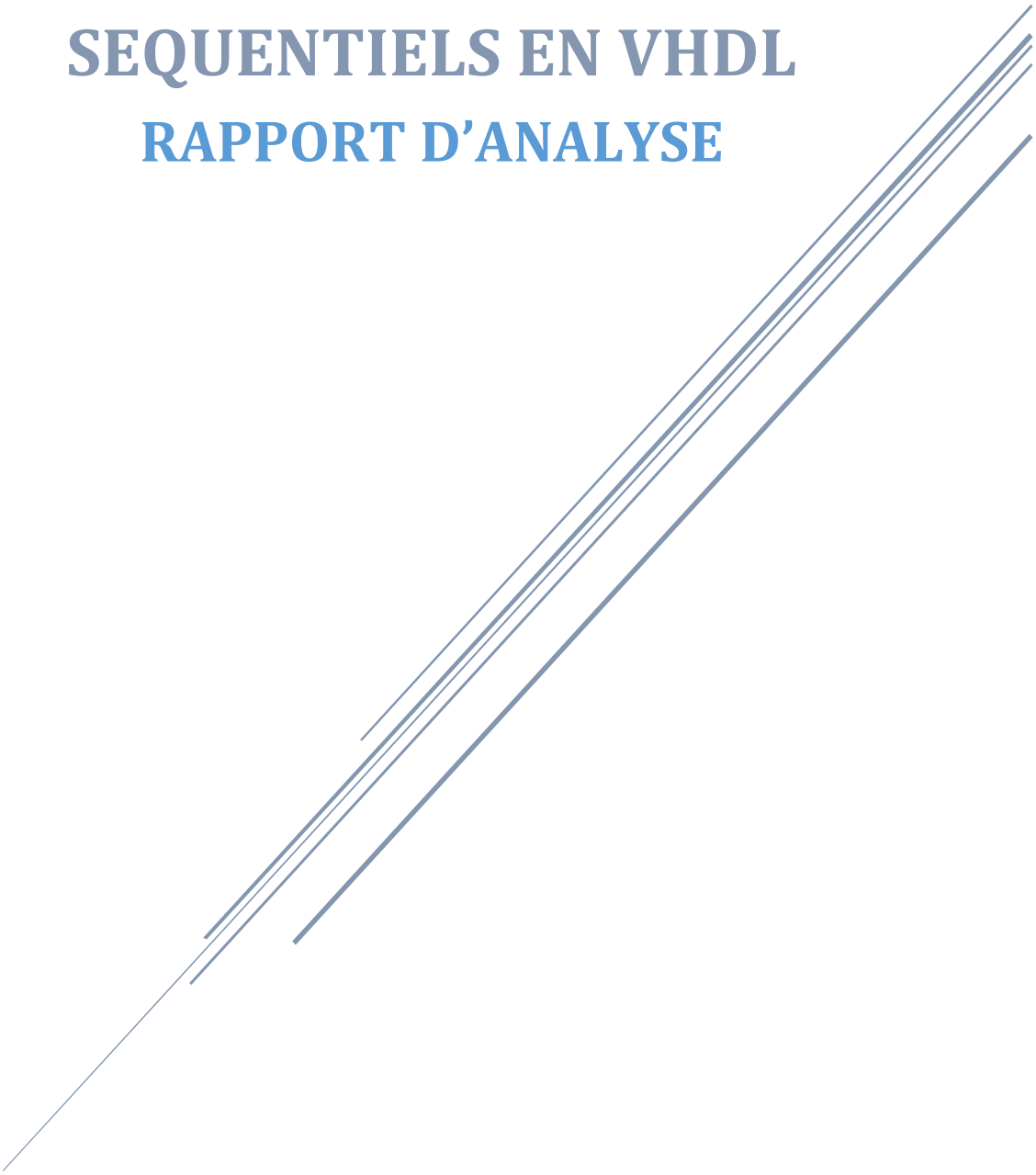
2A CSI PHELMA 08 octobre 2016

PROJET INFORMATIQUE : SYNTHESE DE CIRCUITS SEQUENTIELS EN VHDL

RAPPORT D'ANALYSE



Floriant PETERSCHMITT & Pénélope NICOLAS

TABLE DES MATIERES

INTRODUCTION

Pendant les quatre prochains mois, nous allons développer un outil informatique capable de faire la synthèse de circuits séquentiels en VHDL à partir d'un fichier texte contenant un programme en langage VHDL.

Le but de ce projet est de concevoir et d'implémenter en langage C++ un logiciel permettant de synthétiser une machine à états finis décrite en langage VHDL en une netlist VHDL, c'est-à-dire de passer d'une description matérielle à un niveau d'abstraction plus bas. En effet, la synthèse consiste à traduire un programme en langage informatique lisible par l'Homme en un ensemble de portes qui pourront être implémentées sur FPGA ou placées & routées pour un ASIC.

Le livrable de ce projet consistera en un logiciel ayant pour entrée un fichier texte contenant un programme en langage VHDL et produisant un arbre de structure de données représentant le programme compilé et une netlist VHDL au niveau porte. Pour ce faire, le fichier texte sera tout d'abord décodé à l'aide d'un parseur ce qui nous permettra d'obtenir l'arbre vérifié représentant le programme en entrée. Ensuite, l'étape de synthèse consistera à traduire le fichier texte et à générer une netlist VHDL en détectant et créant les portes logiques depuis l'arbre précédemment créé et vérifié. Ce livrable devra fonctionner sous linux sur les ordinateurs de PHELMA pour toutes les instructions considérées dans le sujet fourni lors de la première séance.

Pour réaliser ce projet, nous allons le décomposer en plusieurs parties facilement testables et assemblables pour pouvoir nous répartir le travail et valider au fur et à mesure chaque partie réalisée. La décomposition que nous avons choisie sera détaillée dans la suite du rapport.

Ce projet informatique présente une continuité avec la première année puisqu'il va nous permettre d'améliorer nos compétences en C++ mais il va également illustrer les notions de synthèse matérielle abordées en cours de VHDL, architecture et microélectronique. De plus, il représente pour nous un pas en avant vers le métier d'ingénieur puisque c'est le premier projet moyen terme que nous devons gérer depuis l'analyse en passant par la conception et l'implémentation jusqu'à la validation.

I- Décomposition modulaire du programme

Le synthétiseur VHDL à réaliser devra tout d'abord décrypter le fichier texte afin d'extraire un arbre représentant le programme en entrée et de le vérifier. Cette première étape sera effectuée à l'aide d'un parseur. Le parseur sera lui-même composé de 3 modules de type « analyseur » :

- Un analyseur de lexèmes qui permet le découpage du fichier texte en lexèmes et qui vérifie que ceux-ci respectent les règles d'orthographe basique

- Un analyseur de syntaxe qui permet de tagger chaque lexème afin de sortir l'arbre de structure de données représentatif du code VHDL en entrée

- Un analyseur de contexte qui permet de comparer l'arbre obtenu avec l'arbre de référence (annexe 1) afin de vérifier si l'enchainement des lexèmes dans le programme est cohérent.

Après que le fichier texte en entrée du logiciel a été parsé et retranscrit en arbre vérifié, celui-ci pourra ensuite passer à l'étape de synthèse qui se décompose en deux parties : la première étape permet de vérifier que le VHDL est synthétisable et la seconde consiste à détecter et créer les portes logiques dans la netlist VHDL qui est le fichier de sortie de notre logiciel.

La décomposition modulaire choisie est celle abordée en cours puisqu'elle nous est apparue très claire en cours et, lors de nos phases de réflexion, elle était totalement en accord avec nos idées d'implémentation. Cette découpe en quatre modules permet également de se répartir plus facilement les tâches puisqu'ils ne sont pas liés et peuvent donc être implémentés dans le désordre et non par une seule et même personne. De plus, chaque module a une entrée, un rôle et une sortie, ils peuvent donc être testés et validés indépendamment les uns des autres avant d'être testés et validés ensemble.

Le fonctionnement de chaque module sera détaillé dans la suite du rapport.

La décomposition modulaire détaillée précédemment est représentée ci-dessous.

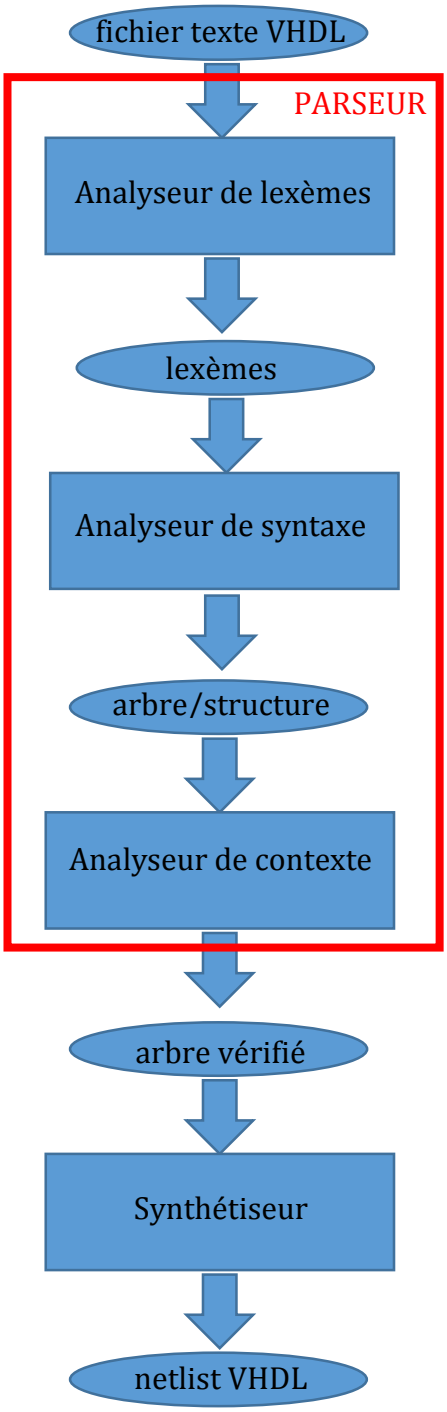


Figure 1: Décomposition modulaire du programme

II- Description des modules

a) L'analyseur de lexèmes

Ce module prend en entrée le fichier texte contenant un programme en VHDL et doit le découper en lexèmes. Il doit également vérifier l'orthographe des lexèmes et traiter les cas spéciaux tels que les lignes commentées ou les majuscules.

Principe de fonctionnement : Un pointeur va balayer le fichier texte caractère par caractère jusqu'à rencontrer un séparateur, tel que « espace, retour à la ligne, tabulation », ou un caractère spécial considéré comme un lexème, tel que « ; : ( ) [ ] ». Les caractères spéciaux cités précédemment doivent être considérés comme des lexèmes car ils ne sont pas nécessairement séparés du mot qui les précède ou de celui qui les succède. Une solution pour répondre à cette contrainte est donc de préciser dans notre fonction de découpage que ces caractères spéciaux doivent être vus comme des séparateurs, mais qu'ils doivent également être reconnus comme lexèmes puisqu'ils sont nécessaires à certaines instructions du VHDL. Lorsqu'un lexème est identifié, son orthographe doit être vérifiée. En effet si un lexème ne respecte pas les règles citées ci-dessous, alors l'analyseur devra retourner une erreur.

Règles d'orthographe des lexèmes en VHDL :

- ✓ Un lexème peut être composé de caractères alphanumériques et d'underscore(s)
- ✓ Les underscore(s) peuvent être utilisés dans un lexème à condition qu'ils ne soient ni le premier, ni le dernier caractère de celui-ci
- ✓ Le premier caractère du lexème doit être une lettre
- ✓ Un caractère spécial est un lexème
- ✓ Un lexème ne peut contenir un autre lexème

Chaque lexème reconnu et vérifié sera placé dans un tableau à double entrée, comme présenté sur la figure ci-dessous, afin de conserver la position du mot dans le fichier texte, c'est-à-dire son numéro de ligne et sa place dans la ligne représentée par un numéro de colonne, pour pouvoir localiser les erreurs facilement et pour faciliter la création de l'arbre de structure de données.

	1	2	3	4	5	6	7	…
1	entity	alpha	is					
2	port	(						
…								

Figure 2 : Tableau en sortie de l'analyseur de lexèmes

L'analyseur de lexèmes doit également traiter des cas spéciaux tels que les commentaires et les majuscules.

Un commentaire en code VHDL commence par « -- » et se termine par un retour à la ligne. Leur gestion se fera donc de la manière suivante : lors de la détection d'une succession de deux lexèmes « - », l'analyseur de lexèmes devra sauter la ligne et le pointeur sera automatiquement placé sur le premier caractère de la ligne suivant en ne faisant pas apparaître la ligne commentée dans le tableau.

En ce qui concerne les majuscules, elles peuvent tout à fait être utilisées dans du code VHDL puisque le compilateur n'est pas sensible à la casse. Cependant, l'analyseur de syntaxe doit tagger chaque lexème qu'il reconnait parmi la liste fournie. Une contrainte de cette liste est qu'elle ne reconnaitra pas un lexème s'il contient une ou plusieurs majuscules puisque le code ASCII d'une majuscule est différent d'une minuscule. L'analyseur de lexème devra alors détecter la présence d'une majuscule et la remplacer par une minuscule.

Pour résumer, l'analyseur de lexème prend comme entrée un fichier texte VHDL, qu'il découpe et traite à l'aide des différentes fonctions détaillées ci-dessous pour sortir un tableau à double entrée rempli de lexèmes à l'orthographe vérifiée et écrit en minuscules. Ce tableau sera l'entrée de l'analyseur de syntaxe.

b) L'analyseur de syntaxe

Ce module consiste à traiter tous les lexèmes identifiés grâce au module précédent en les associant à un tag. Il doit également créer un arbre de structure de données à partir du tableau de lexèmes extrait par l'analyseur de lexèmes.

Principe de fonctionnement : Tous les types de lexèmes possibles seront répertoriés dans une classe à laquelle chaque lexème du tableau extrait par l'analyseur de lexèmes sera comparé. Lorsqu'un lexème sera reconnu, il sera taggé dans un autre tableau à double entrée qui sera le reflet du premier. Ce tableau sera donc l'image du fichier texte en entrée, à la seule différence qu'il sera rempli par des tags et non par des lexèmes. On considèrera ce tableau comme arbre de structure de données du fichier texte VHDL et il sera la sortie du fichier texte de référence. Sur les figures ci-dessous sont présentés les tableaux en entrée et en sortie de l'analyseur de syntaxe.

	1	2	3	4	5	6	7	…
1	entity	alpha	is					
2	port	(						
…								

Figure 3 : Tableau en entrée de l'analyseur de syntaxe

	1	2	3	4	5	6	7	…
1	t_entity	name_entity	t_is					
2	t_port	t_(						
…								

Figure 4 : Tableau en sortie de l'analyseur de syntaxe faisait office d'arbre de structure de données

Le second rôle de l'analyseur de syntaxe est de créer un dictionnaire permettant de répertorier les identifiants attribués lors de la déclaration d'une entité, d'une architecture, d'un process ou d'un signal. Ce dictionnaire sera sous la forme d'une table de symboles comme présenté sur la figure ci-dessous. Chaque identifiant sera répertorié de par son nom, son rôle (= port, signal, nom_entity, …), son type (=entrée ou sortie) et permettra l'accès à son nœud pour avoir plus d'informations. Ce dictionnaire sera utile pour éviter qu'un même identifiant soit utilisé plusieurs fois et pour vérifier lors de l'étape suivante, que lorsqu'un identifiant est appelé, il a bien été préalablement déclaré.

Pour résumer, l'analyseur de syntaxe prend en entrée le tableau à double entrée rempli de lexèmes à l'orthographe vérifiée et écrit en minuscules pour sortir un tableau à double entrée de tags faisant office d'arbre de structure de données. Ce tableau sera l'entrée de l'analyseur de contexte. Un deuxième outil disponible à la sortie de l'analyseur de syntaxe est le dictionnaire contenant tous les identifiants déclarés dans le code VHDL. Dans la prochaine étape, l'arbre de structure extrait sera vérifié.

c) L'analyseur de contexte

Ce module consiste à comparer l'arbre de structure représentatif du code VHDL extrait sous la forme d'un tableau à double entrée lors de l'analyse de la syntaxe à un arbre de référence (annexe 1).

Principe de fonctionnement : L'annexe 1 présente l'arbre de référence qui correspond à la base de la structure de données attendue. Il s'agit d'un arbre de syntaxe. L'annexe 2 présente quant à elle le workflow qui détaille toutes les combinaisons d'instructions VHDL possibles dans le sous-ensemble VHDL considéré pour ce projet. L'analyseur de contexte prendra en entrée l'arbre de structure du code VHDL extrait par l'analyseur de syntaxe et le comparera à l'arbre de référence en utilisant le workflow de l'annexe 2 comme une FSM puisqu'il présente toutes les possibilités de syntaxe et contexte du code dans le sous-ensemble VHDL considéré comme des états. L'arbre de structure VHDL devra traverser tous les états du workflow et si ce n'est pas le cas, l'analyseur de contexte devra retourner une erreur sur l'enchainement de lexème erroné et présenter les enchainements de lexèmes autorisés.

L'analyseur de contexte devra également vérifier que chaque identifiant utilisé a bien été déclaré en le comparant au dictionnaire en sortie de l'analyseur de syntaxe.

Pour résumer, l'analyseur de syntaxe prend en entrée le tableau à double entrée de tags faisant office d'arbre de structure de données et le compare à l'arbre de référence via un workflow détaillé pour sortir un arbre vérifié. Ce tableau sera l'entrée du synthétiseur. L'arbre vérifié n'est disponible en sortie uniquement si tous les identifiants utilisés sont présents dans le dictionnaire, sinon l'analyseur de contexte retourne une erreur.

A la sortie de l'analyseur de contexte, le fichier a été parsé, ce qui signifie qu'il peut être compilé mais pas qu'il ne peut être synthétisé. Cette seconde vérification se fera à la synthèse.

d) Le synthétiseur

Ce module procède en deux étapes : une première qui permet de vérifier que le code est synthétisable et qui prépare le fichier pour l'élaboration, et une seconde étape qui élabore la synthèse, c'est-à-dire qui traduit le code VHDL en portes et génère une netlist VHDL.

Principe de fonctionnement : Le synthétiseur va tout d'abord vérifier que le code VHDL du fichier en entrée est synthétisable. Le fichier en entrée sera l'arbre vérifié disponible en sortie du parseur.

Pour être synthétisable, un composant VHDL doit suivre les règles suivantes :

- ✓ Implémentation de la logique séquentielle et de la logique combinatoire dans des process différents
- ✓ Affectation d'un signal par un seul et unique process
- ✓ La liste de sensibilité d'un process doit contenir tous les signaux qu'il lit
- ✓ Seulement les éléments séquentiels peuvent tester un front sur un signal
- ✓ Lecture et affection d'un même signal dans un process impossible
- ✓ Affectation d'une valeur par le process à tous ces signaux de sortie
- ✓ La partie contrôle et la partie opérative doivent toujours être séparées
- ✓ Initialisation des variables obligatoire avant leur lecture
- ✓ Interdiction d'utiliser les instructions d'attente « wait » et les assignations inertielles « after ».

Pour vérifier que le code VHDL est synthétisable, le fichier va parcourir l'arbre de décisions présenté ci-dessous qui vérifiera les règles citées ci-dessus.

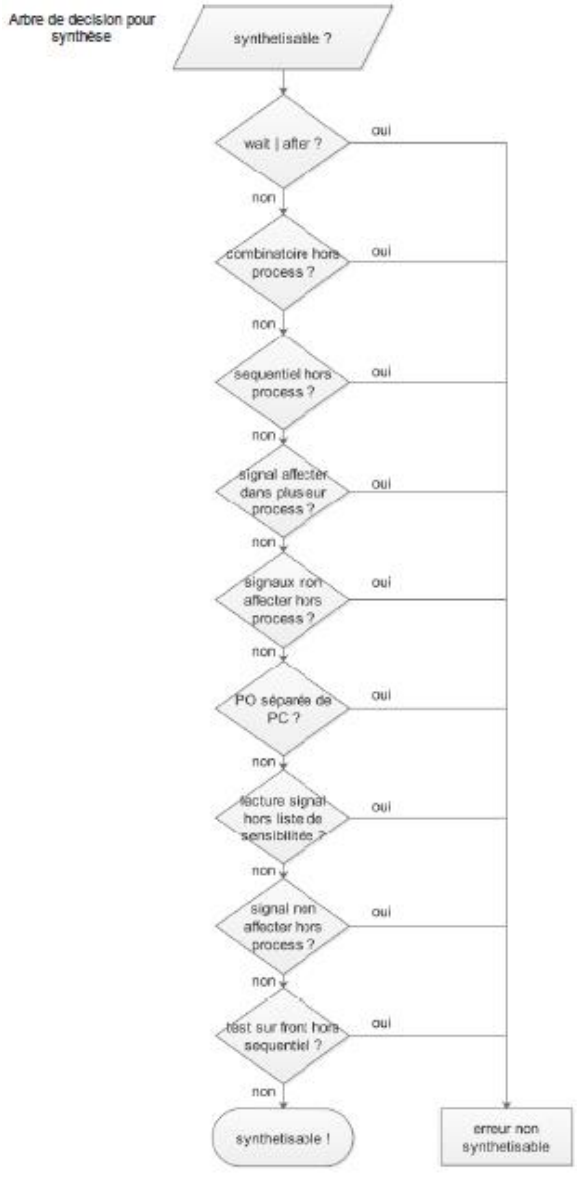


Figure 5 : Arbre de décision permettant de vérifier que le VHDL est synthétisable

Cependant, dans notre cas, la dernière règle concernant les « wait, after » n'a pas besoin d'être vérifié puisque ces deux instructions ne sont pas considérées dans le sous-ensemble VHDL à traiter.

Si le fichier réussit à sortir de l'arbre de décisions alors l'élaboration de la synthèse pourra être effectuée. Cependant, le fichier contenant l'arbre vérifié et synthétisable devra tout d'abord être préparé, c'est-à-dire qu'il devra être découpé en petit blocs de taille progressive, comme présenté sur la figure ci-dessous, qui seront analysés et traduits par une routine en portes lors de l'élaboration. Chaque bloc correspondra à une structure VHDL car chaque structure VHDL correspond à une porte logique.

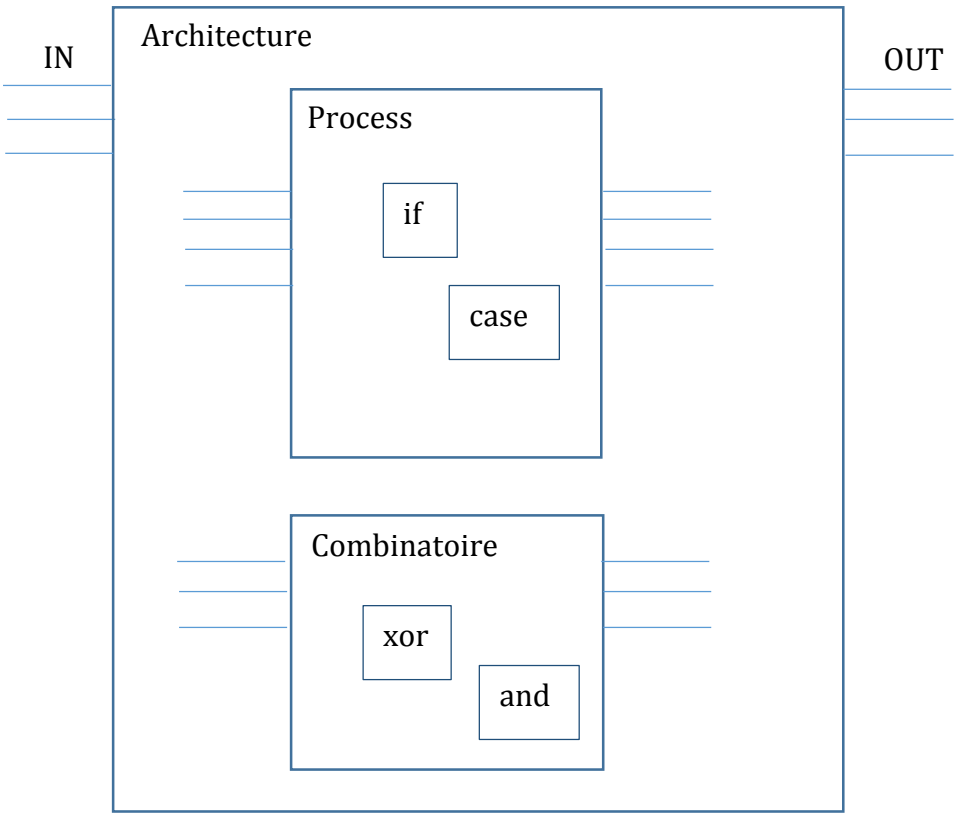


Figure 6 : Exemple de découpage du programme en petits blocs

En ce qui concerne la routine de traduction, elle va premièrement analyser chaque bloc. Pour chacun elle devra tout d'abord repérer le type du bloc puis elle devra détecter quels sont les signaux en entrée et en sortie, quel est leur type et combien sont-ils. Enfin, elle utilisera les informations extraites précédemment pour écrire le code VHDL de chaque porte puisque chaque type de bloc correspond à un type de porte logique contenu dans une bibliothèque prédéfinie. Ce fichier sera la sortie du synthétiseur et donc du programme complet et sera appelé netlist VHDL.

III- Problèmes techniques soulevés

Au fil de cette phase d'analyse, nous avons pu rencontrer différents problèmes auxquels nous avons pu partiellement répondre.

Le premier problème concerne l'appel des commandes système en C/C++. En effet, nous avons dû chercher des renseignements sur la manière d'ajouter les commandes bash dans les codes sources. Nous avons trouvé qu'une fonction système existe en C++ et qu'elle permet de traduire les commandes bash par simple appel de celle-ci sur la commande à traduire.

Le second problème concerne la gestion des données et le « versioning » du projet. En effet, nous devons être capables de travailler en parallèle sur nos ordinateurs personnels mais chacun doit pouvoir suivre l'avancée de l'autre. Pour répondre à ces contraintes, nous allons utiliser un logiciel tel que Git qui est un système de gestion de versionnement décentralisé utiliser pour Linux qui permet modifier localement un programme puis de transmettre la version aux autres utilisateurs.

IV- Test et validation des modules

Une étape du projet qui ne fait pas partie de la conception ni de l'implémentation, mais qui reste nécessaire pour garantir le fonctionnement du programme final, est celle de test et de validation des modules. En effet, chaque module composant notre programme va être développé séparément et nécessitera d'être testé avant d'être assemblé aux autres. Mettre en place une méthodologie de test et d'évaluation systématique de chacun des modules va permettre d'éviter un debug seulement sur le programme final qui sera long, difficile et qui n'identifiera pas le module source du problème.

Dans cette partie, les méthodes pour tester et valider chaque module seront présentées. Chaque module sera testé depuis sa phase de développement jusqu'à sa phase finale.

- Test & Validation de l'analyseur lexical :
  - Création d'un fichier template sans erreur
  - Création d'un fichier template représentatif des erreurs, copie du premier fichier template avec injection d'erreurs
  - Appel de l'analyseur lexical sur les deux fichiers et vérification que les erreurs sortent bien aux endroits où elles ont été injectées

- Test & Validation de l'analyseur de syntaxe :
  - Récupérer le fichier template sans erreur en sortie de l'analyseur lexical
  - Appel de l'analyseur de syntaxe sur le fichier
  - Création manuelle de l'arbre correspondant au fichier
  - Comparaison des deux arbres et vérification de la correspondance

- Test & Validation de l'analyseur de contexte :
    - Création d'un arbre correct issu de l'arbre de référence
    - Création d'un arbre avec des enchainements de lexèmes incorrects
    - Appel de l'analyseur de contexte sur les deux arbres et vérification de la détection et de la position des erreurs

- Test & Validation du synthétiseur :
    - Créer des arbres vérifiés sans erreur puis avec des erreurs correspondant à chacune des étapes de l'arbre de décisions permettant de vérifier que le code est synthétisable
    - Faire tourner le synthétiseur sur chacun des arbres et vérifier qu'il affiche bien une erreur et qu'elle correspond bien à celle introduite dans l'arbre
    - Après avoir vérifié que l'arbre sans erreur réussit à franchir l'arbre de décisions, il faut vérifier que le fichier après la routine de traduction est bien en code VHDL et qu'il traduit les bonnes portes logiques

Chacune de ces méthodes sera appelée à l'aide d'une option debug au logiciel final pour éviter que la phase debug ait lieu à chaque lancement du programme.

V- Organisation des fichiers

La figure ci-dessous présente le l'arborescence Unix illustrant l'organisation des fichiers choisie.

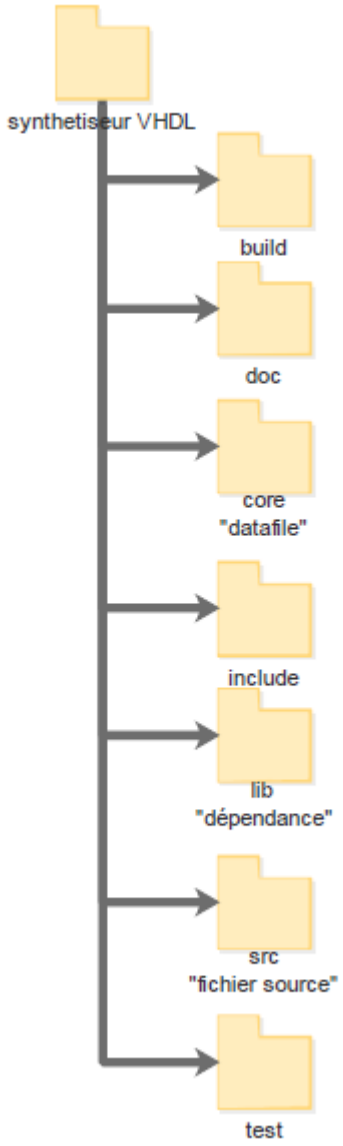


Figure 7 : Arborescence Unix d'organisation des fichiers

VI- Planning prévisionnel et répartition du travail

La remise en temps et en heure d'un projet repose sur l'organisation de l'équipe en charge de ce projet et sur le respect d'un planning préalablement défini.

Pour commencer, nous avons commencé par établir un planning prévisionnel qui nous semble cohérent actuellement mais qui est susceptible d'être légèrement modifié au fil de la progression du projet. Cependant, nous devons veiller à garder assez de temps pour assembler nos modules et tester et valider le programme complet. Ce planning est présenté sur la figure ci-dessous.

	W40	W41	W42	W43	W44	W45	W46	W47	W48	W49	W50	W51	W52	W01	W02	W03	W04	W05	W06
analyseur de lexèmes + validation																			
analyseur de syntaxe + validation																			
analyseur de contexte + validation																			
assemblage du parseur + validation																			
synthétiseur + validation																			
assemblage du parseur et du synthétiseur																			
test et validation du programme complet																			

W** : période en entreprise
W** : séances de tutorat
W** : deadlines
W40 : 09/10/16 : rendu du rapport d'analyse
W05 : 02/02/17 : rendu des sources
W06 : 09/02/17 : soutenance

Figure 8 : Planning prévisonnel de gestion du projet

Pour pouvoir respecter ce planning, nous avons décidé de nous répartir les tâches. Cependant un module sera développé par une seule et même personne puisque nous n'avons pas la même manière d'écrire le code. Une personne s'occupera donc du développement de l'analyseur de lexème, de l'analyseur de syntaxe et de la vérification que le code VHDL est synthétisable. L'autre personne sera donc en charge de l'analyseur de contexte, de la préparation à la synthèse et de la routine de traduction. Chacun sera le back-up de l'autre sur chaque module pour assurer un bon développement de ceux-ci. L'avantage du travail séparé est de pouvoir avancer le projet chacun à notre rythme et de notre côté. Cependant, la mise en commun régulière, peut-être chaque semaine selon l'avancée de chacun, reste indispensable pour suivre l'avancée du projet dans sa globalité et pour détecter rapidement un problème qui pourrait engendrer un retard et y remédier. De plus, nous utiliserons un logiciel de versioning de type de Git pour que chacun puisse travailler localement et faire suivre son travail facilement à l'autre. Pour finir, les étapes d'assemblage, de test et de validation seront effectuées par les deux membres du binôme pour bénéficier d'une covalidation.

CONCLUSION

Ce rapport d'analyse nous a permis d'avoir une approche sur la globalité du projet afin de visualiser la quantité de travail à fournir et d'estimer la durée de chacune des tâches pour pouvoir rendre le projet fini en temps et en heure.

Grâce à cette analyse, nous avons pu opter pour une logique de programme C++ orientée objet.

Nous avons également mis en place une stratégie de groupe afin d'être le plus efficace possible et le plus fiable possible.

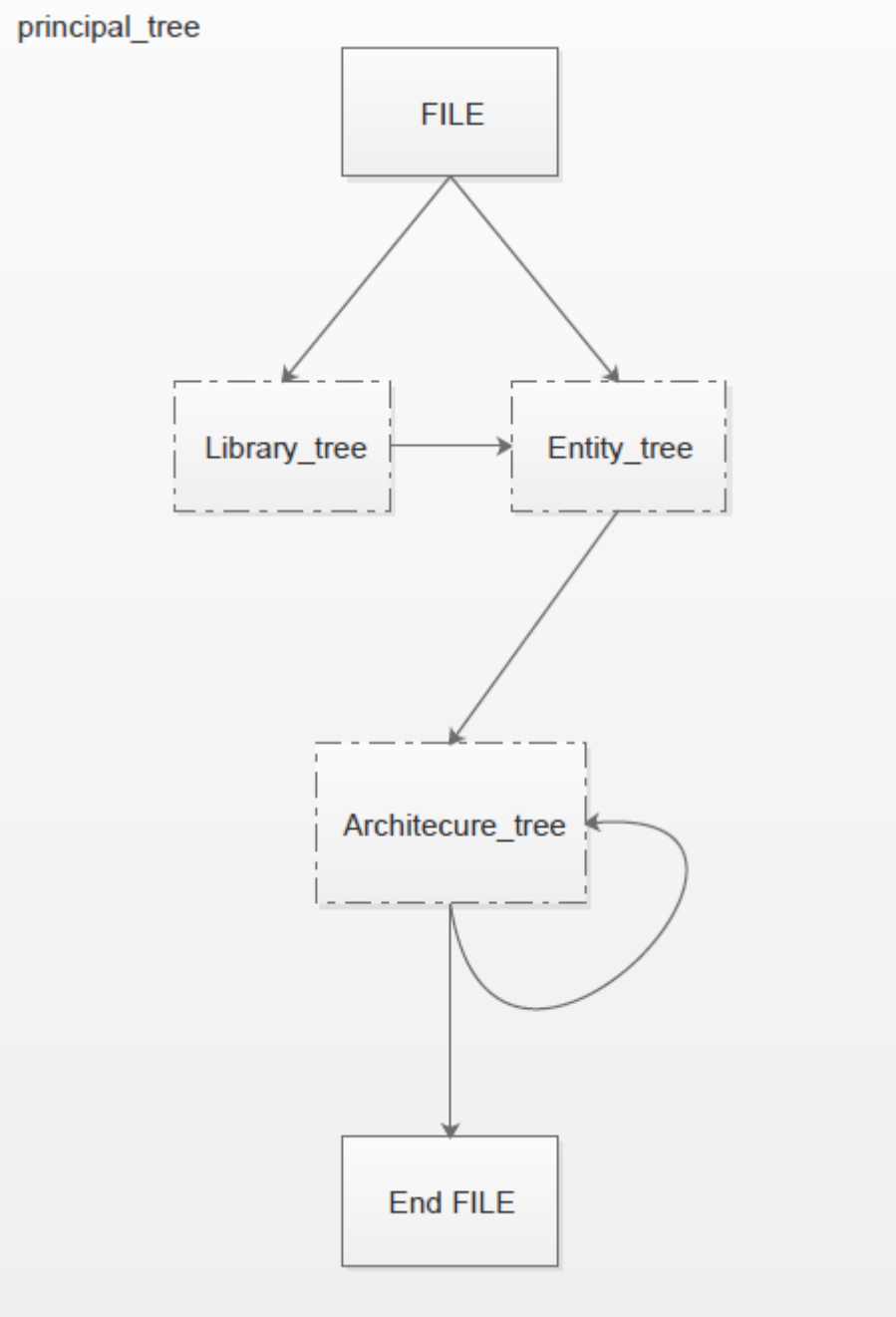
La méthodologie de test et de validation a été choisie pour éviter un debug trop fastidieux en phase finale et pour valider chaque module pas à pas.

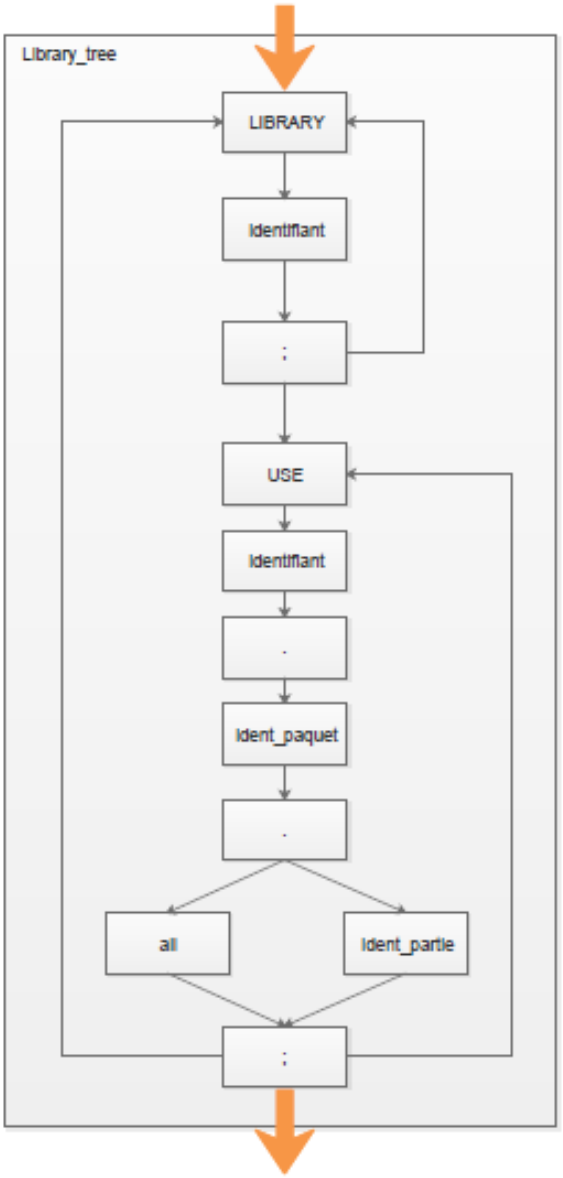
Certaines optimisations sont dès à présent envisageables comme par exemple l'introduction d'un module de gestion d'erreur parallèle à notre parseur. En effet, nous pouvons voir que chaque module du parseur entraîne une génération d'erreurs, ce qui pourrait être amélioré en laissant la gestion d'erreurs à ce module parallèle. Cependant, les différentes optimisations seront effectuées par ordre d'importance et d'efficacité en fonction du temps disponible restant.

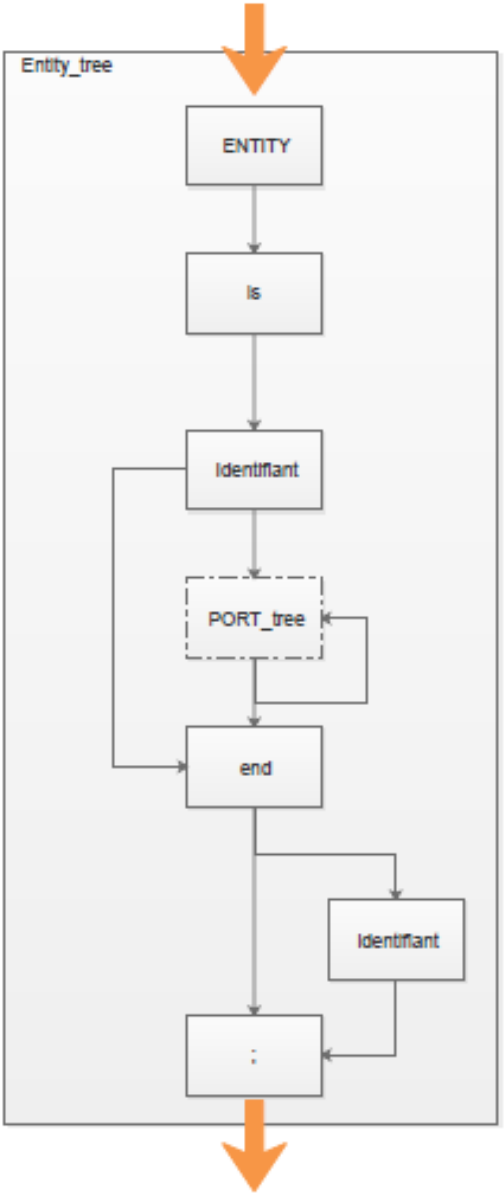
Annexe 1 : Arbre de structure de données de référence

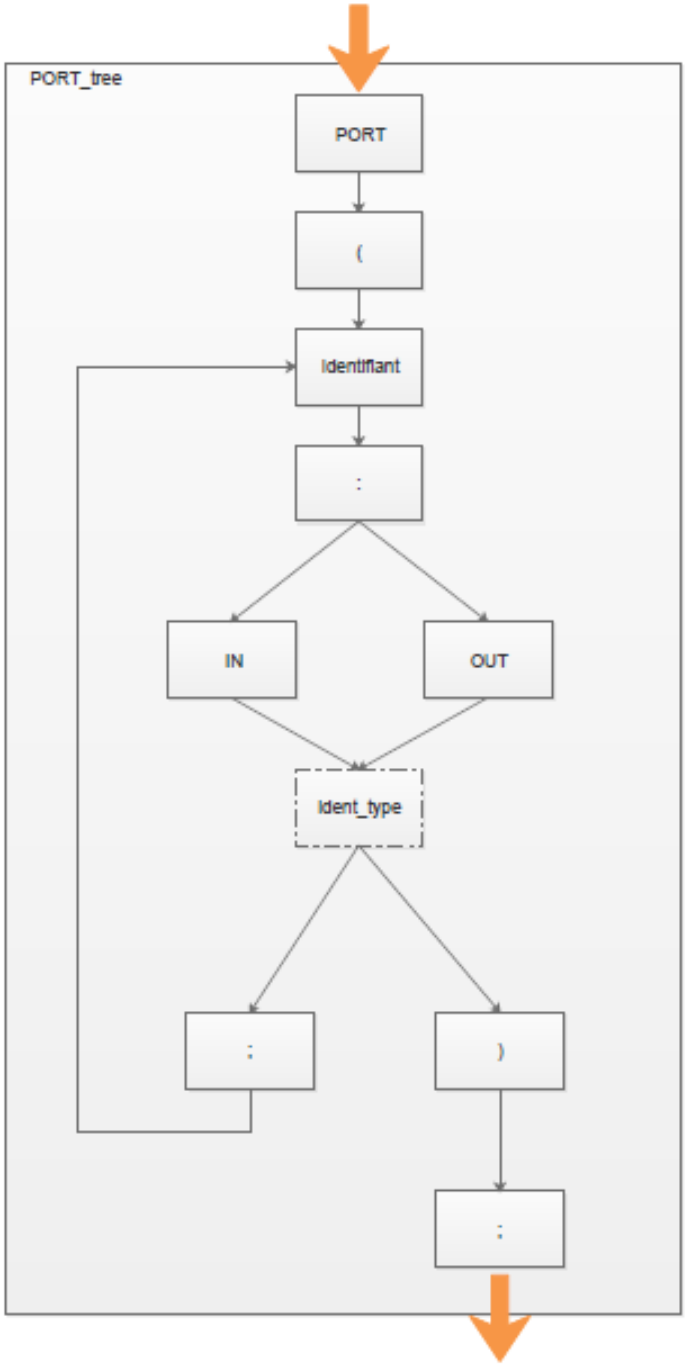
Annexe 2 : Workflow permettant le test et la validation de la syntaxe et du contexte du fichier VHDL

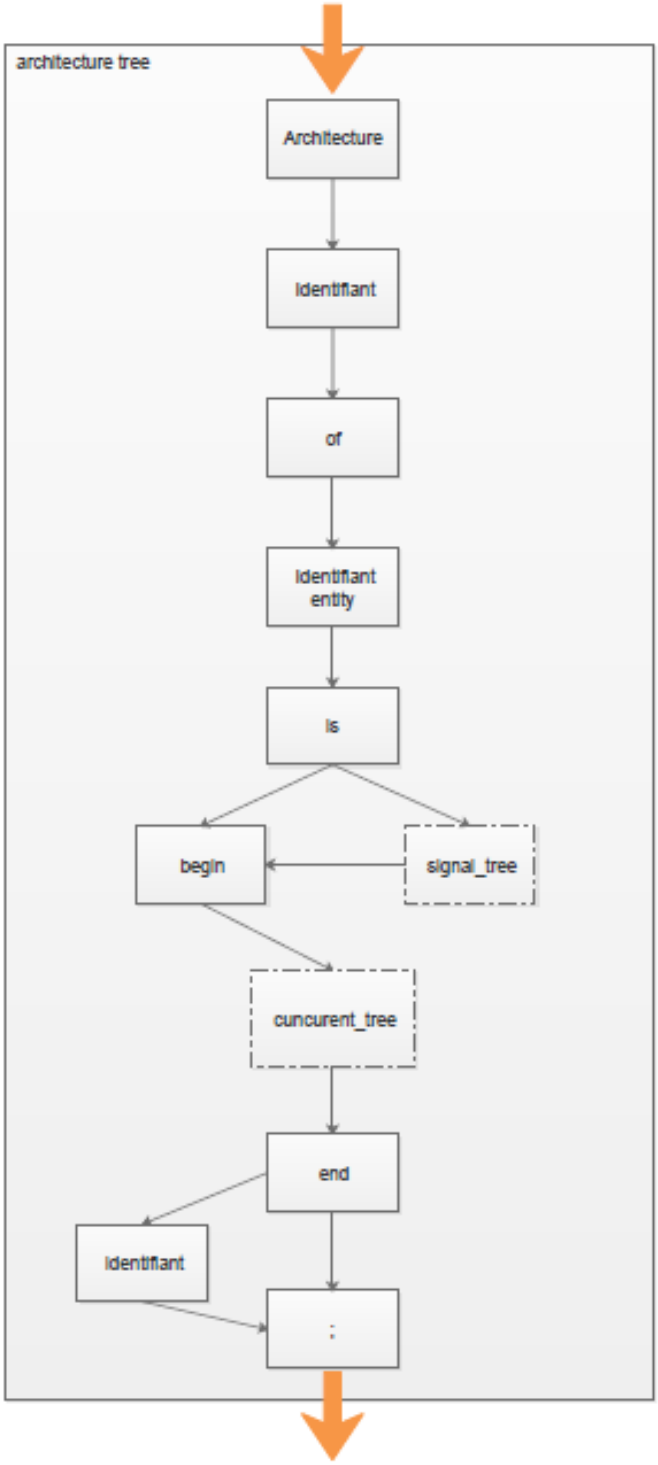
Note : Chaque structure en pointillés représente un sous-arbre décrit par la suite

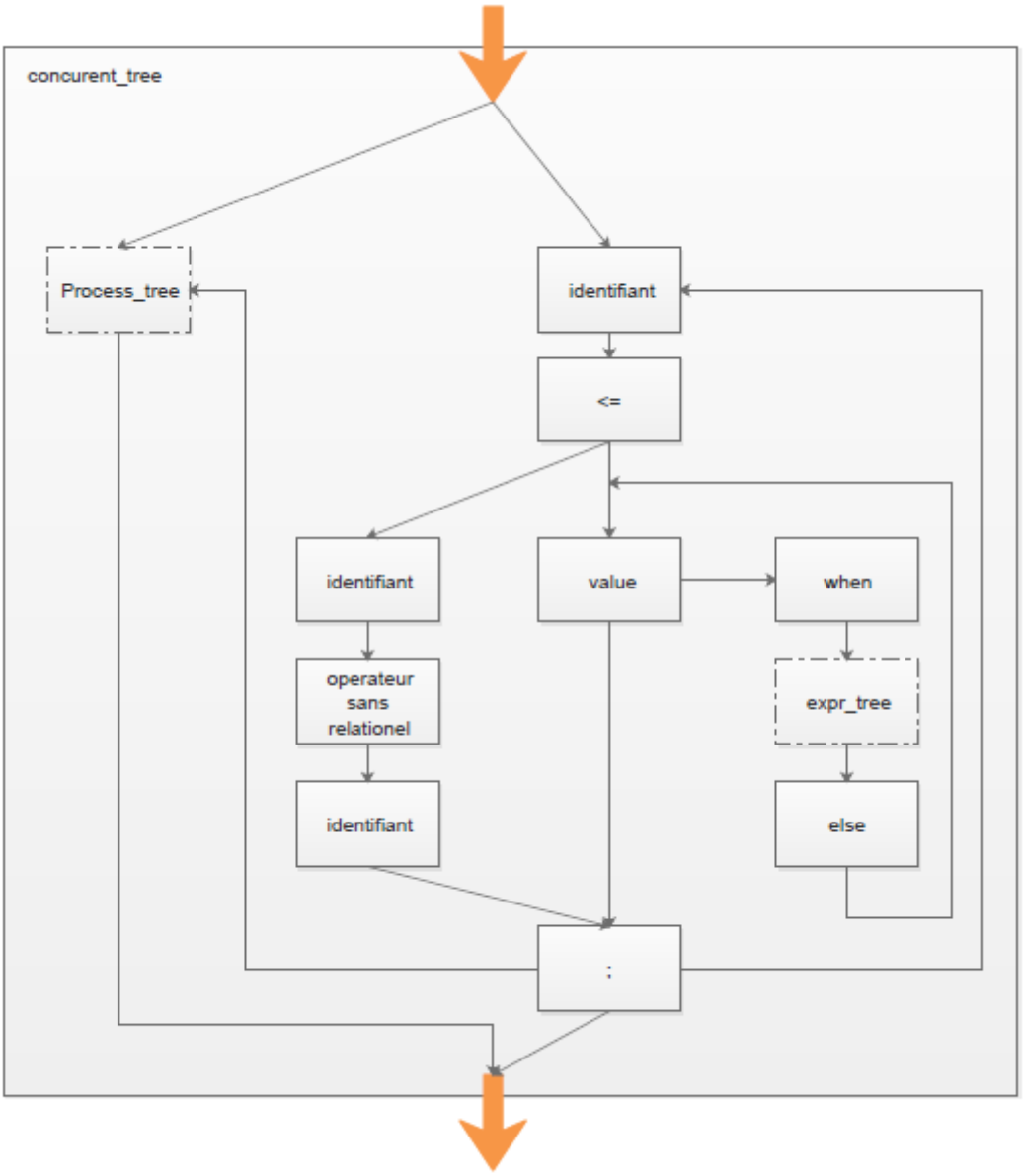


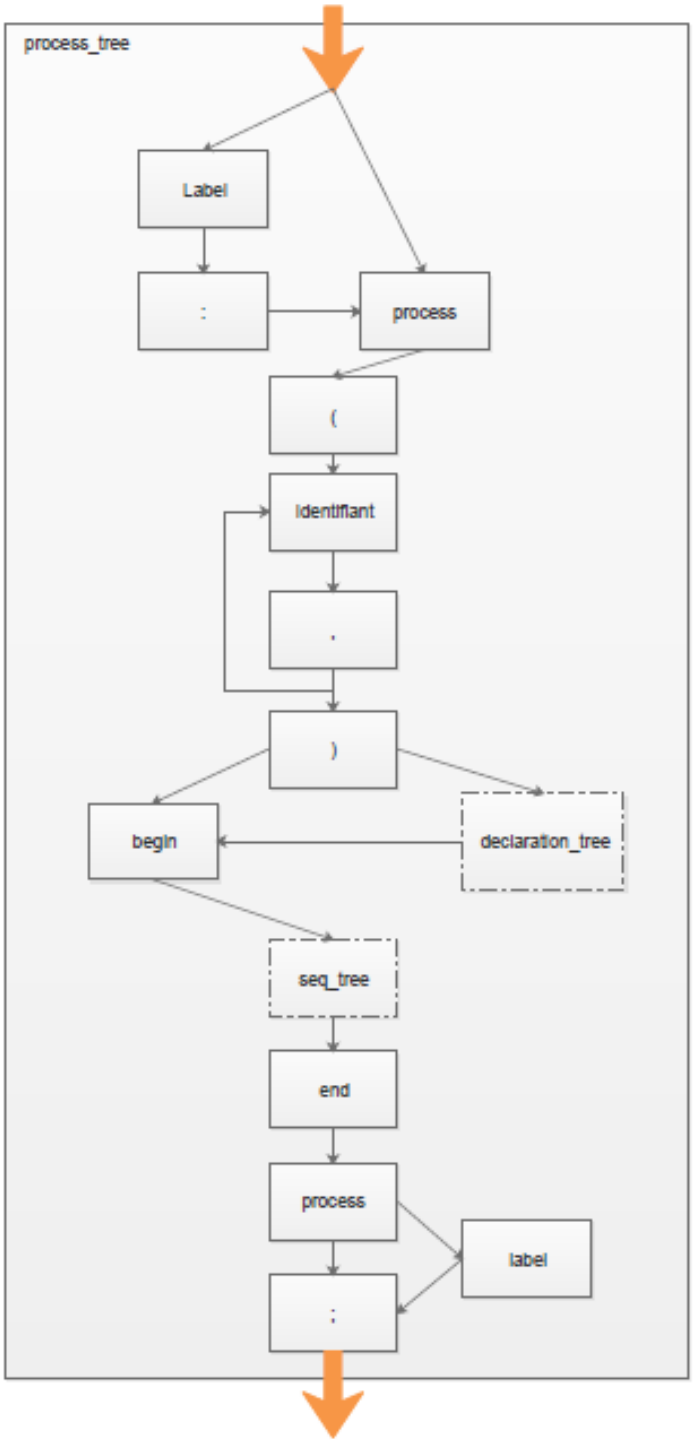

Library_tree
LIBRARY
Identifiant
;
USE
Identifiant
.
Ident_paquet
.
all
Ident_partie
;

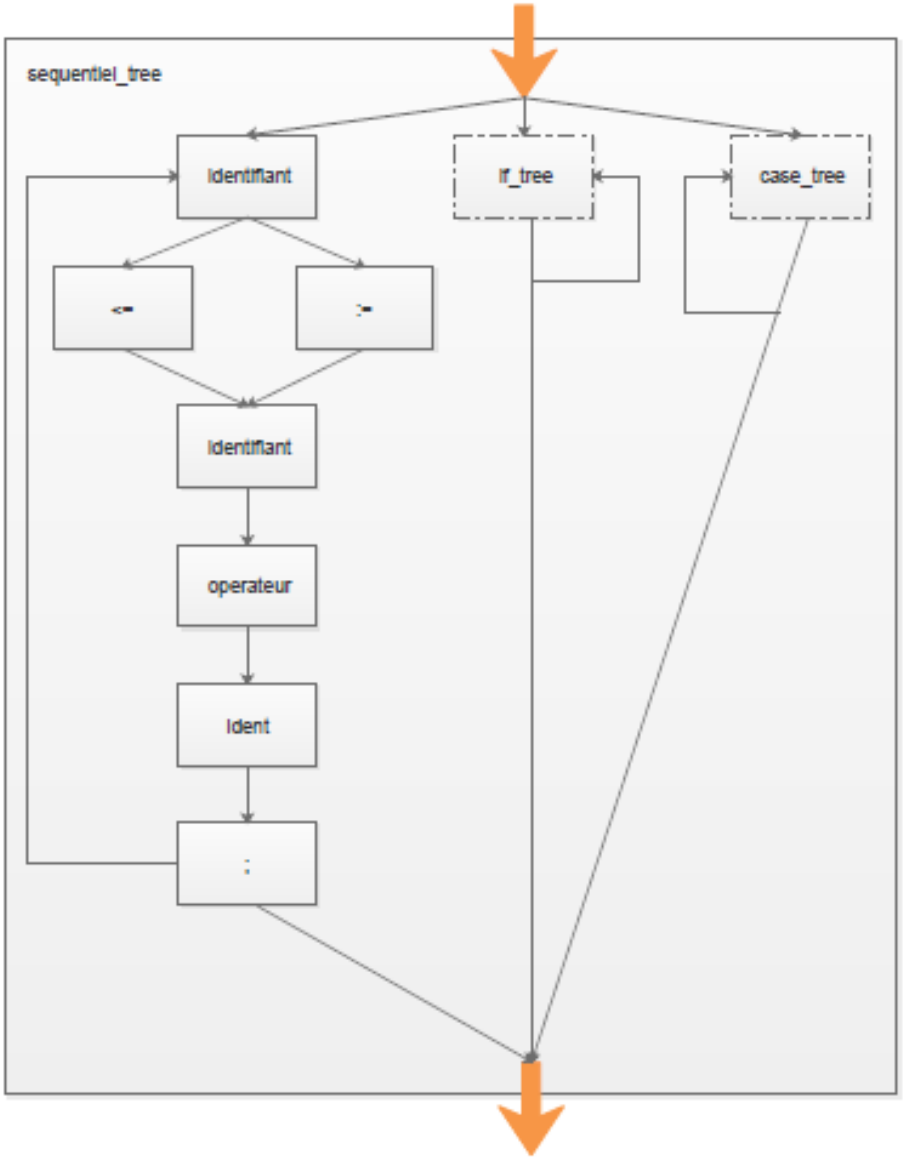

Entity_tree
ENTITY
is
Identifiant
PORT_tree
end
Identifiant
;

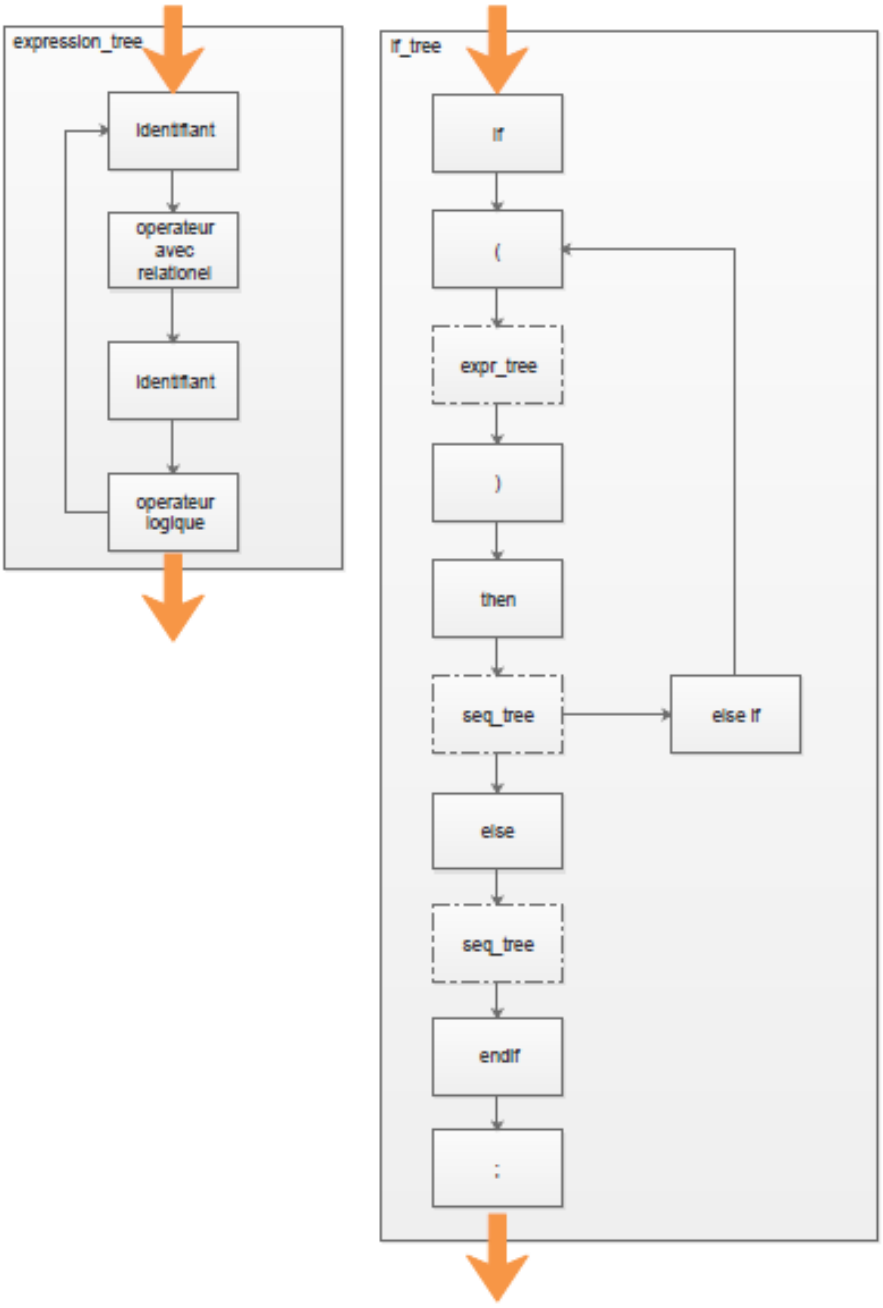

PORT_tree
PORT
(
Identifiant
:
IN
OUT
Ident_type
;
)
;

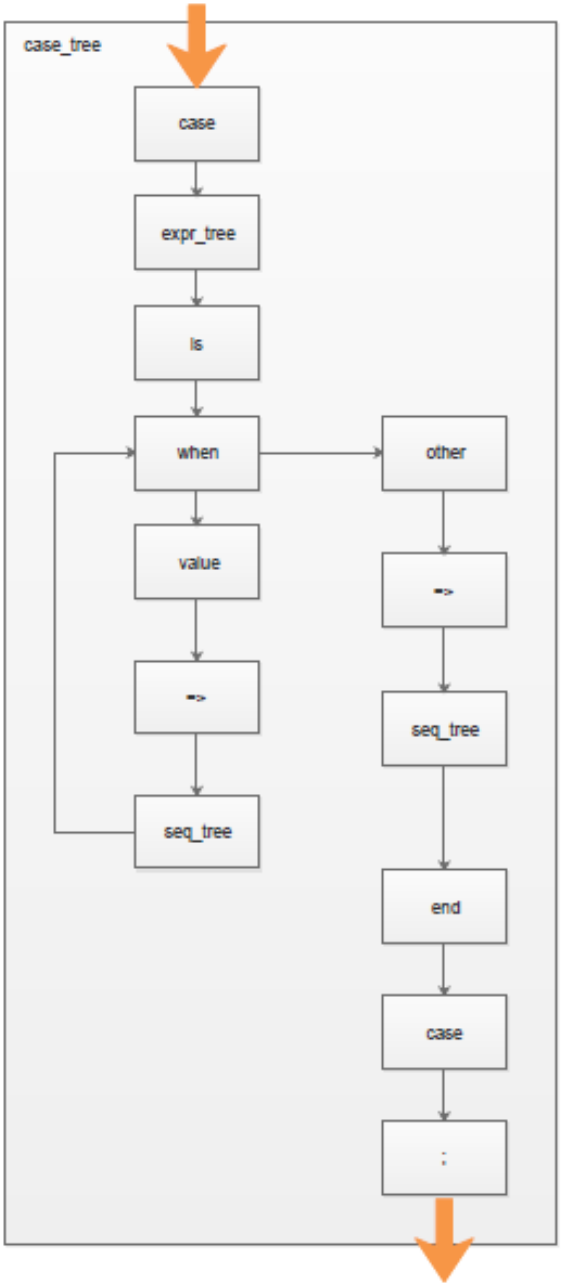

architecture tree
Architecture
Identifiant
of
Identifiant
entity
is
begin
signal_tree
cuncurent_tree
end
Identifiant
;


concurent_tree
Process_tree
identifiant
<=
identifiant
value
when
operateur sans relationel
expr_tree
identifiant
else
;


process_tree
Label
:
process
(
Identifiant
,
)
begin
declaration_tree
seq_tree
end
process
label
;


sequentiel_tree
identifiant
if_tree
case_tree
<=
:=
identifiant
operateur
ident
;


expression_tree
Identifiant
operateur avec relationel
Identifiant
operateur logique
If_tree
If
(
expr_tree
)
then
seq_tree
else If
else
seq_tree
endif
;


case_tree
case
expr_tree
is
when
value
=>
seq_tree
other
=>
seq_tree
end
case
;