Langage de spécification de systèmes de pilotage d'expériences

Bertrand Blanc Frédéric Maccari

9 février 2003

Table des matières

Chapitre 1

Présentation

1.1 Introduction

Le second semestre de maîtrise propose aux étudiants de réaliser un **T**ravail d'**E**tudes et de **R**echerches sur un sujet libre, encadré par un enseignant. Ce projet "grandeur nature" a pour but d'introduire le 3^{eme} cycle pour sensibiliser d'une part les étudiants au domaine de la Recherche et d'autre part appliquer la théorie et les langages appris dans les différents modules depuis la Licence.

Le TER que nous avons choisi est mené en collaboration entre le CMI dont la représentante est Mme Pierre et le service d'électronique et de micro-informatique de l'Institut de Biologie Structurale et Micro-biologie du CNRS représenté par M. Zenatti. Ce service a pour but d'apporter des aides matérielles et logicielles dans les projets d'automatisation d'expériences des biologistes.

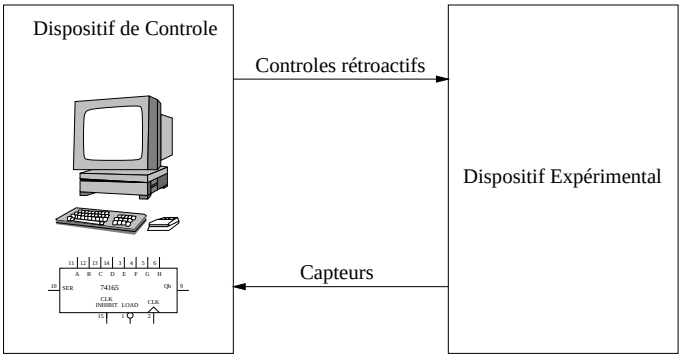

FIG. 1.1 – *Pilotage d'une expérience*

Comme le montre la figure 1.1, dans un tel projet, un dispositif de contrôle reçoit des paramètres de l'expérience via des capteurs et renvoie en conséquence des signaux de contrôle aux éléments matériels du dispositif expérimental, et ceci pendant toute la durée de l'expérience.

Le projet du TER est de proposer un langage simple et concis permettant à l'utilisateur de décrire l'expérience à piloter, et de mettre en oeuvre des outils de compilation et de traduction associés à ce langage. Le travail se découpe en

deux phases maîtresses:

- Analyse du contexte et de l'existant en fonction du cahier des charges, des spécifications et des contraintes. A l'issue de cette analyse, nous avons proposé un langage de modélisation de haut niveau adapté aux besoins.
- Réalisation des traducteurs et compilateurs. Le but est de mettre en oeuvre les différentes étapes entre la production d'un code source décrivant une expérience écrit dans notre langage et le code généré, pour un processeur donné.

1.2 Contexte

Le problème qui nous est proposé est de piloter des expériences. A priori, le contrôle peut être entièrement mis en oeuvre sous forme logicielle. Cependant, des contraintes de temps réel peuvent imposer une réalisation mixte matérielle/logicielle qui correspond à un bon compromis entre les contraintes de temps d'exécution et de coût des composants matériels. De ce fait, il peut s'avérer nécessaire d'avoir recours à des outils de codesign pour réaliser ce partitionnement logiciel/matériel.

Notre langage doit ainsi respecter deux points fondamentaux:

1. Il doit pouvoir facilement être transformé en un langage d'entrée d'outils de codesign.
2. Il est de plus souhaitable de pouvoir prototyper rapidement les modèles de pilotage d'expérience par des simulations. Ainsi, il doit avoir une sémantique proche d'un langage pour lequel il existe des outils de simulation.

Un langage ad hoc développé au sein du service de M. Zenatti il y a quelques années est actuellement utilisé. Cependant sa définition ne repose sur aucune sémantique formelle et ses spécifications ne satisfont pas les besoins et les contraintes définies par l'analyse du projet.

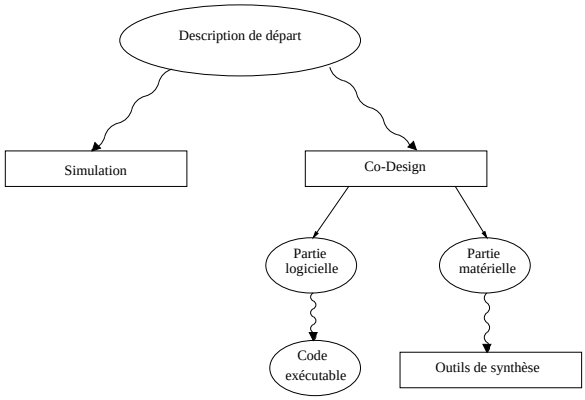

FIG. 1.2 – *Diagramme de présentation*

La figure 1.2 représente l'architecture sommaire d'inter-actions entre les différents modules. L'expérience décrite pourra soit être simulée (par exemple en

utilisant les outils de Synopsys) soit entrer dans un outil de co-design exhibant, en fonction des contraintes de temps, d'espace et de coût précisées par le biologiste, une partie logicielle et une partie matérielle (par exemple l'outil développé par R. Kamdem dans sa thèse [3]). Les éléments implantés en logiciel pourront être compilés, donnant un code exécutable pour un processeur donné. La partie matérielle pourra entrer dans des outils de synthèse (comme par exemple ceux de Mentor Graphics ou Compass), donnant une implantation physique de composants.

Le paragraphe suivant traite de l'élaboration du langage, de sa génération et traduction par les outils flex et bison. Nous transformerons, ensuite, le code en C grâce à des outils libres d'utilisation et proposerons un noyau de fonctionnement: dialogue avec le matériel, interruption périodique, ...

1.3 Langage et outils associés

1.3.1 Le langage

Besoins

Lors d'une phase d'analyse préalable, nous avons identifié les besoins relatifs à notre langage. Ces derniers peuvent se différencier en deux classes qui sont les besoins liés à l'environnement et les besoins concernant la partie contrôle.

Intéressons nous d'abord aux besoins environnementaux. Le langage doit pouvoir permettre de communiquer avec un dispositif expérimental, c'est à dire qu'il faut prévoir des constructions qui permettent de lui envoyer des signaux. Réciproquement, le langage doit permettre à l'utilisateur d'acquérir des données venant du dispositif, ceci afin de pouvoir faire un contrôle rétroactif[1] de l'expérience. Cette notion est représentée par la notion de *ports*.

En ce qui concerne la partie contrôle, nous avons identifié les besoins en étudiant des systèmes de ce type qui sont actuellement conçus au C.N.R.S.. Nous avons relevé une analogie entre les systèmes qui seront spécifiés dans notre langage et un système d'exploitation. Tout d'abord, ce type de système de contrôle est composé de *tâches concurrentes* qui sont *synchronisées* par une horloge commune. Chacune d'elle est composée d'une suite d'*instructions séquentielles*. Enfin, nous devons gérer la communication entre les différentes tâches au travers d'un système de *mémoire partagée*.

Solution proposée

Au vu des seuls besoins étudiés ci-dessus, il semble que le problème posé peut être résolu avec le langage V.H.D.L. [1] qui contient toutes les constructions sémantiques et sur lequel reposent des outils de simulation et de codesign dont nous avons besoin. En fait, nous avons développé un nouveau langage car les

1. Le terme de *contrôle rétroactif* désigne le fait que le système doit être capable de modifier son comportement au vu des résultats des acquisitions

utilisateurs cibles qui sont définis dans le cahier des charges (voir annexe B) n'ont pas les compétences nécessaires pour utiliser un langage tel que V.H.D.L..

Nous allons développer, un langage baptisé SP.A.C.E. (**SP**ecification l**A**nguage for **C**ontroling **E**xperiments), de sémantique proche de celle d'un sous-ensemble comportemental synchrone de V.H.D.L. permettant de décrire "facilement" des expériences.

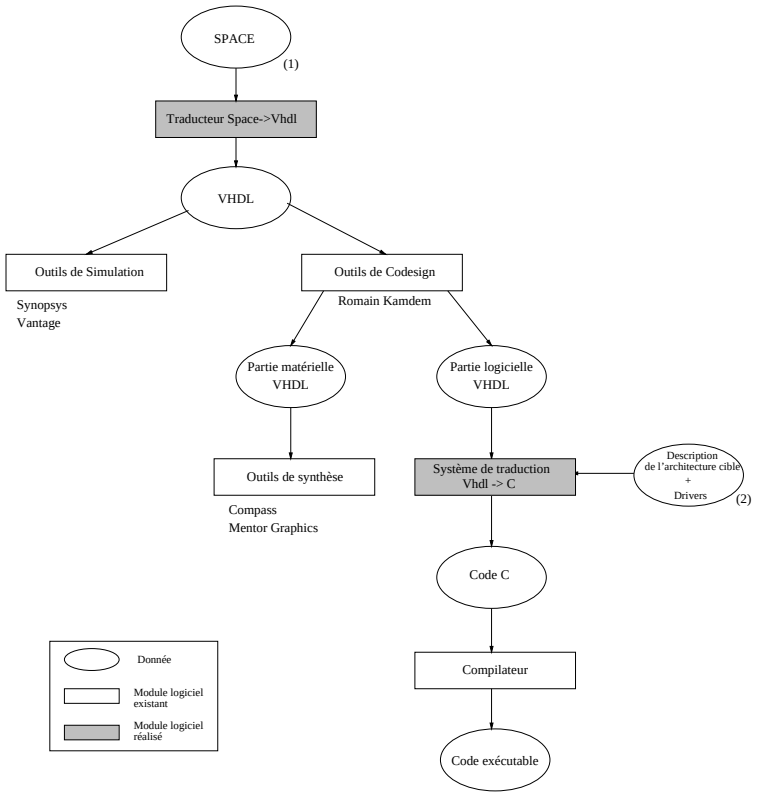

FIG. 1.3 – *Diagramme d'interaction entre nos outils et ceux existants*

La figure 1.3 présente de façon précise la transformation du code SPACE décrivant l'expérience tout au long du processus permettant de générer le code exécutable. Le code source SPACE est dans un premier temps traduit en VHDL. Nous avons par la suite, la possibilité de simuler l'expérience ou bien d'utiliser des outils de co-design. Nous obtenons alors deux ensembles de tâches en VHDL, l'un représentant la partie matérielle et l'autre la partie à implanter en logiciel. Des outils comme Compass ou Mentor Graphics permettent de synthétiser les tâches de la partie matérielle. D'autre part notre système de traduction de VHDL en C reposant sur un traducteur GPL et la description de l'architecture cible va permettre d'obtenir du code C compilable sur un processeur choisi.

Rapport entre le langage S.P.A.C.E. et le langage V.H.D.L.

Nous avons identifié le sous-ensemble de V.H.D.L. qui est utile pour modéliser les systèmes de pilotage d'expérience et ainsi l'utilisateur aura la possibilité

de transformer tout programme en S.P.A.C.E. en un programme V.H.D.L. Le sous-ensemble que nous avons défini est le suivant:

- Chaque tâche qui fait partie du système peut être modélisée grâce à l'instruction "*process*". Cette instruction permet de définir une tâche dont le corps est composé d'instructions séquentielles.
- La synchronisation des tâches sur une horloge commune se décrit avec une instruction "*wait*".
- La communication entre les taches peut se modéliser par des *signaux* qui sont connus de tous les *process*.
- La communication avec le dispositif expérimental est assuré par la notion de ports d'entrée et de sortie en V.H.D.L..

Le détail de la grammaire et de la sémantique du langage se trouvent dans le dossier dans la partie "*Manuel de l'utilisateur*".

Organisation d'une spécification en langage S.P.A.C.E.

Pour résumer, un système de contrôle d'expérience est décrit dans notre langage par trois fichier différents et une collection de "*Drivers*".

Le premier fichier contient la description de la carte ou du système sur lequel les tâches vont s'exécuter. Cette description consiste en une série de déclarations de ports avec leur type et leur adresse de base, ainsi que la déclaration des possibilités du "*timer*". A chaque type de port doit correspondre un driver qui doit être situé dans un répertoire précis du système sur lequel l'application va être compilée. Ce fichier, de même que les "*drivers*", doit être réalisé une et une seule fois pour un type de plate-forme, ou de port, donné; il pourra être utilisé par plusieurs expériences différentes.

Le second fichier décrit la manière dont nous allons utiliser une plate-forme décrite par le premier fichier (fichier (2) sur la figure 1.3). En effet, ce fichier va nous permettre de donner le type des ports, d'en utiliser moins que ce que la carte en offre et de régler la fréquence du "*timer*".

Le troisième fichier contient la description du système de pilotage de l'expérience (fichier (1) sur la figure 1.3).

Grâce a ce découpage en trois fichiers, il sera très facile de porter un système de contrôle d'une plate-forme vers une autre.

1.3.2 Compilateur de S.P.A.C.E. vers V.H.D.L

Ce compilateur traduit une spécification de système en S.P.A.C.E. en une spécification en V.H.D.L.. Nous avons décidé de placer toutes les vérifications sémantiques et syntaxiques lors de cette compilation. Le compilateur produit, à l'aide des trois fichiers deux sorties: la première est le système proprement dit écrit en V.H.D.L. et le second est un fichier qui va permettre de créer le code *C* exécutable.

Nous avons réalisé ce traducteur en suivant les contraintes liées à la qualité qui sont définies dans le "*plan d'assurance qualité*" (voir annexe A). Ces contraintes nous ont poussé à utiliser des normes et à faire des choix quant à l'écriture de celui-ci.

Le premier choix que nous avons effectué fut d'utiliser les outils "*flex*" et "*bison*". Ce fut une partie assez ardue car nous ne connaissions pas ces deux programmes et nous avons donc dû apprendre à les utiliser à partir de la documentation disponible sur le site Internet de G.N.U.. Ce sont les avantages que nous avons tirés de l'adoption de ces programmes qui nous ont poussé à faire ce choix. En effet, le fait que ces outils constituent un standard permettent d'assurer une bonne et facile évolutivité; l'autre grand avantage vient du fait que le code produit est fiable et nous allège ainsi d'une partie du travail de vérification. Nous avons par ailleurs choisi de découper le traducteur en modules de façon à garder le plus de modularité possible. Le système logiciel se compose donc, entre autres, d'un module qui gère la table de hachage, d'un autre qui gère les appels et erreurs du système, d'un autre qui permet de manipuler des listes d'identificateurs, etc...

Dans toute la phase d'implémentation, nous avons gardé en tête cet aspect d'évolutivité, étant bien conscient qu'un projet de cette ampleur ne pourra être poursuivi par d'autres personnes que si son développement respecte certaines règles.

1.3.3 Traduction de V.H.D.L. en C

Nous avons trouvé un outil qui traduit une spécification en V.H.D.L. en un programme *C*. Cet outil se nomme "*V2C*". Il permet d'obtenir un fichier en *C* qui contient tous les mécanismes nécessaires au fonctionnement d'un système de tâches concurrentes.

Il manque toutefois une fonctionnalité majeure à ce traducteur: la communication avec du matériel n'est pas gérée. Par conséquent, nous avons choisi d'utiliser ce programme à l'intérieur d'un processus qui nous permettra d'obtenir le code *C* de l'application devant piloter le dispositif expérimental.

Notre travail à ce niveau a été constitué de trois phases. La première a été consacrée à la recherche et la mise en place du processus cité précédemment. La seconde concernait l'écriture d'une sorte de "*scheduler*" devant gérer le réveil périodique des tâches. Dans la troisième, nous avons écrit les modules qui permettent de faire le lien entre les ports du programme de contrôle et les ports effectifs de l'architecture cible.

Voici la description du processus nous permettant d'obtenir le code *C*. Ce processus est assez complexe et il n'est donc décrit ici qu'au travers des grandes lignes.

1. Traduction du code V.H.D.L. par V2C.modes
2. Analyse du fichier intermédiaire généré par le compilateur vers V.H.D.L. afin d'obtenir les renseignements nécessaires pour gérer les appels aux ports d'entrée et de sortie.
3. Création du fichier contenant le "*scheduler*" et la gestion des appels des procédures pour lire et écrire sur les ports.
4. Copie dans le répertoire courant des fichiers nécessaires à la compilation du programme de contrôle.
5. Création du "*Makefile*".

1.4 Conclusion

Ce travail d'études et de recherches a tout d'abord abouti à la création d'un langage de spécification de système de contrôle d'expérience. Ce langage est nouveau et prend place dans une niche qui n'est pas encore très bien fournie. Il présente le gros avantage de reposer sur deux standard de leur domaines respectif: le langage V.H.D.L. pour ses nombreux outils de simulation, de synthèse et de codesign; et le C qui nous permet de disposer de compilateurs pour tout microprocesseur ou microcontrôleur existant. Nous avons par ailleurs défini une méthode statique (contrairement à une approche interpréteur comme pour JAVA) qui permet d'obtenir un code pouvant s'exécuter sur tout type de plateforme. Les biologistes ont enfin à leur disposition un outil de spécification simple, puissant et adapté à leur besoin. Ils auront ainsi la possibilité d'imaginer des expériences complexes et sur une longue période qui seront entièrement automatisées pour des temps et des coûts de développement très largement inférieurs à ce qui se fait actuellement.

Une limitation est toutefois à noter. Elle concerne la modification des variables partagées. Ces dernières ne sont ne peuvent être modifiées que dans une seule des tâches constituant le système de pilotage. Une solution peut-être envisagée pour y remédier, elle consiste à définir la notion de fonction de résolution. Cette fonction permettra de décider en cas de conflit quelle est la valeur effectivement prise par cette variable.

Les perspectives de ce qui a été produit sont assez prometteuses. Une interface graphique liée à ce logiciel devrait permettre la création d'un environnement intégré de développement où la simplicité du langage associée à la convivialité d'une telle interface permettra à des biologistes de concevoir dans le plus grand confort des systèmes de contrôles pour leurs propres expériences. A terme, l'addition des possibilités du codesign produira un ensemble logiciel complet et homogène qui permettra de faire décroître encore une fois très sensiblement les temps et les coûts de conception de systèmes de contrôles d'expérience.

Sur un plan plus personnel, nous avons appris de très nombreuses notions durant ce projet. La plus importante est la notion de travail personnel et d'autoapprentissage. Nous avons en effet plus d'expérience pour rechercher une documentation ou pour apprendre à utiliser des produits complexes à partir de ce qu'on peut trouver sur Internet. Nous avons aussi évolué dans le domaine de la gestion de projet. Cela a été une nécessité pour éviter de finir sur un échec. Enfin, nous avons pu aborder le domaine de la communication entre un logiciel et du matériel qui est un domaine important à l'heure actuelle. Nous connaissons mieux les enjeux et les techniques qui entourent ce domaine.

Bibliographie

[1] R. Airiau J.-M. Bergé V. Olive J. Rouillard. *VHDL: du langage à la modélisation*. Presse Polytechniques et universitaires romandes, 1990.

[2] D. Campo. Manuel de commandes pour pilote. Technical report, C.N.R.S., 1998.

[3] Romain Bertrand Kamdem. Contribution au partitionnement matériel/logiciel des systèmes temps réel pour les applications de contrôle d'expériences. Master's thesis, Laboratoire d'informatique de Marseille, 1999.

[4] MOTOROLA. *MC68HC16Z1 user's manual*, 1992.

Annexe A

Plan d'assurance qualité

A.1 But, domaine d'application et responsabilités

Ce plan d'assurance qualité concerne le développement d'un langage de spécification de systèmes de contrôle d'expérience ainsi que de deux outils de compilation associés. Le premier compliateur produira du source en VHDL, et le deuxième tranformera le code en C incluant des appels de fonctions en assembleur (pour les drivers notamment). Ce logiciel est réalisé dans le cadre du T.E.R. de l'année 1999-2000.

La réalisation de ce plan d'assurance qualité est assurée par M. Maccari. Le suivi de ce plan sera réalisé par celui-ci et il prend donc comme responsabilité de le suivre et de le faire suivre à la lettre et dans ses moindres détails.

Au cours de différentes discussions avec M. Zénatti, nous sommes arrivés à dégager des facteurs principaux de qualité auxquels le présent projet devra répondre. Ces derniers sont l'évolutivité et la maintenabilité. Ces deux facteurs impliquent donc une série de critères qui pourront permettre, grâce à une métrique de juger de la qualité du produit final. Ces critères seront décrits plus loin dans ce plan.

Afin de présenter une bonne flexibilité, ce plan qualité, qui doit être validé à la fin de sa réalisation par les tuteurs de ce T.E.R. qui sont Mlle Laurence Pierre et M. André Zénatti, contient une procédure d'évolution[1] décrite ci-après:

- identification du paragraphe du plan d'assurance qualité à modifier.
- rédaction du texte correspondant à l'évolution.
- validation de ce nouveau texte par le responsable de la qualité.
- ajout de ce texte sous forme d'un addendum au plan qualité.

En cas de non application de ce plan, il est prévu de suivre la procédure suivante:

- identification des points qui ne sont pas respectés.
- étude par les responsables de la qualité de moyens à mettre en oeuvre pour remédier à ce non respect.
- s'il est possible d'y remédier alors présentation de la solution aux tuteurs.

1. On impose que ces évolutions aillent dans le sens de l'amélioration de la qualité, pas dans son amoindrissement.

- s'il est impossible d'y remédier, ou si les tuteurs ne considèrent pas la procédure comme viable, alors le responsable de la qualité devra joindre au dossier de réalisation une feuille de non respect de la qualité précisant le ou les points qui ne sont pas respectés, les raisons de ce non respect et la ou les conséquences sur la qualité du produit final.
- si une solution a été trouvée pour remédier au problème alors mise en oeuvre de cette solution et rédaction d'une fiche d'incident de la qualité précisant la nature de la non application et le remède employé.

A.2 Documents applicables et documents de référence

A.2.1 Documents applicables:

- manuel de commandes pour PILOTE. CAMPO Denis. Version 10/98. [2]

A.2.2 Documents de référence:

- VHDL: du langage à la modélisation. R. Airiau, J.-M. Bergé, V. Olive, J. Rouillard. Ed. Presse Polytechniques et universitaires romandes. 1990. [1]
- MC68HC16Z1 user's manual. Motorola. 08/92. [4]

A.3 Terminologie

A.3.1 Abréviations:

- HC16 désigne le processeur MC68HC16Z1 de Motorola.
- Pentium désigne les processeurs Intel et compatibles (AMD K6, CYRIX, ...) à partir de la génération des i586.
- VHDL désigne le langage de modélisation V.H.D.L. (VHSIC [2] Hardware Description Language).

A.4 Organisation

A.4.1 Intervenants du projet

Ce projet est réalisé par le binôme Blanc-Maccari, sous la conduite de Mlle. Pierre du C.M.I. et tutoré par M. Zénatti du C.N.R.S..

A.4.2 rôles et missions

Mlle. Pierre a pour rôle de guider ses élèves dans leur choix. Ses conseils interviennent à deux niveaux:

- Au niveau de la définition du langage et de la traduction vers le langage VHDL.

2. Very High Speed Integrated Circuit.

- Au niveau de la conception globale du système, car son recul vis-à-vis du projet est plus important et ses remarques forcément pertinentes.

Elle peut intervenir quand elle est sollicitée par ses élèves ou bien quand elle ressent que le projet ne va plus dans le sens où elle l'entend.

M. Zénatti a pour mission d'encadrer le binôme Blanc-Maccari au sein du C.N.R.S.. Il doit exprimer ses besoins pendant la conception du cahier des charges; il peut intervenir en tant que référence concernant la mise en place de programmes pour micro-contrôleurs de par son expérience; il peut enfin émettre des remarques sur la forme du projet s'il sent que le résultat final ne pourra pas s'intégrer dans le projet global.

Binôme Blanc-Maccari sont les personnes chargées de la réalisation du projet. Ils doivent produire les outils et toute la documentation nécessaire relativement à ce plan d'assurance qualité. Ils doivent donc concevoir le programme, mettre le programme en service et produire toute la documentation nécessaire. Ils doivent tenir compte des remarques de Mlle. Pierre et de M. Zénatti.

M. Franck Tison a été désigné par M. Zénatti pour aider le binôme à réaliser les drivers qui seront utilisés pour tester l'ensemble logiciel. Il existe donc un lien fonctionnel entre cette personne et le binôme. Ces derniers fourniront des spécifications de fonctions à M. Tison qui sera chargé d'en réaliser une implémentation.

A.4.3 Liens hiérarchiques et fonctionnels

Il n'y a pas de lien hiérarchique entre Mlle. Pierre et M. Zénatti, leur lien est purement fonctionnel. Ces deux personnes dirigent le projet et leurs points de vue étant complémentaires, il est important qu'elles expriment clairement au binôme les objectifs à atteindre.

Le binôme Blanc-Maccari est donc lié hiérarchiquement à ces deux personnes par le fait qu'elles sont son tuteur et son professeur. Il doit donc leur rapporter ses avancements et tenir compte de leurs remarques/suggestions. Le lien fonctionnel est plus évident: le binôme Blanc-Maccari à besoin de l'expérience et du savoir de Mlle. Pierre et de M. Zénatti.

A.5 Démarche de développement

Cette section fait référence au plan de développement (Figure A.1).

A.5.1 Analyse préalable

C'est la première phase dans le développement, elle est démarrée dès le début du projet.

Activités

Nous allons axer nos recherches sur deux domaines distincts

Analyse de l'existant: cette phase a pour but de chercher ce qui a déjà été

fait dans les domaines qui nous intéressent. Il sera effectué des recherches portant sur les outils disponibles qui pourraient répondre, en partie ou totalement, au problème posé par la création des traducteurs. Nous étudierons par ailleurs les contraintes liées aux drivers existants qui pourront ainsi être éventuellement réutilisés.

Analyse des besoins: Auprès de M. Zénatti, demande d'informations complémentaires concernant les besoins relatifs à cette traduction, et plus particulièrement sur les configurations cibles possibles et leur particularités. Nous analyserons aussi les constructions sémantiques et syntaxiques indispensables dans le langage. Enfin, nous définirons un utilisateur type du langage.

Points d'entrée nécessaires

Le sujet du T.E.R. doit être rédigé et par conséquent fixé.

Documents réalisés

Il sera édité deux documents portant sur les besoins et les contraintes relatifs pour l'un à la définition du langage et pour l'autre à la conception des traducteurs . Ces documents seront inclus au cahier des charges. Les résultats portant sur l'analyse de l'existant seront joints au présent document sous la forme d'un annexe composée de suex parties: une pour les outils de traduction vers VHDL et une autre pour ceux qui nous aiderons pour la traduction en C. Enfin, une annexe sera consacrée à la présentation de toutes ou une partie des architectures cibles qui seront, éventuellement, des plates-formes de test.

Conditions de passage à la phase suivante

La phase suivante sera accessible si les conditions suivantes sont remplies:

1. Au moins deux plates-formes cibles sont analysées.
2. Les besoins sont analysés.
3. Les documents décrits précédemment sont réalisés.
4. Les document d'analyse des besoins et des contraintes à inclure dans le cahier des charges doivent être validés par les tuteurs.

A.5.2 Mise en oeuvre du cahier des charges

Cette phase commence après la phase d'analyse préalable.

Activités

Les contraintes et les besoins ayant déjà été analysés lors de la phase précédente, le travail de cette phase se trouve quelque peu réduit. Il s'agit principalement de rédiger, avec le concours de M. Zénatti, un plan de recette concernant l'ensemble logiciel.

Points d'entrée

Les documents d'analyse des besoins et des contraintes réalisés lors de la phase précédente.

Documents réalisés

Un plan de recette qui se présentera, étant donné que le langage n'est pas encore défini, sous la forme de description de problèmes que le langage et les traducteurs seront sensés résoudre. A la fin de cette phase, le cahier des charges sera terminé et rédigé. Il sera inclus au présent document.

Condition de passage à la phase suivante

Il est nécessaire que le cahier des charges soit rédigé et validé par les tuteurs avant de passer à la phase suivante.

A.5.3 Spécification externe du système

Cette phase prend place juste après la phase de création du cahier des charges.

Activités

Pendant cette période, le binôme Blanc-Maccari va, compte tenu du cahier des charges, produire les spécifications externes de chaque composant du système logiciel. C'est à dire que nous allons décrire les fonctionnalités du langage et des deux traducteurs.

Points d'entrée nécessaires

Cette partie dépend du cahier des charges ainsi que des documents relatifs à l'analyse de l'existant.

Documents réalisés

Trois annexes seront rédigées et donneront les spécifications du langage et des deux composantes du système logiciel. Ces spécifications seront figées pour tout le reste du projet.

Conditions de passage à la phase suivante

Les spécifications du programme seront présentées aux tuteurs et devront être soumise à critique. Si ces spécifications ne se révellent pas satisfaisantes alors le binôme devra revenir sur celles-ci pour en proposer d'autres. On ne pourra clôturer cette phase que si les spécifications sont validées par les tuteurs.

A.5.4 Définition du langage

Cette phase commence après la spécification du langage.

Activités

A partir des spécifications décrites précédemment, définition de la grammaire et de la sémantique du langage. Après cette définition, nous réaliserons des tests afin de déterminer si les possibilités sémantiques du langage permettent de spécifier aisément le type de problème qui nous concerne.

Points d'entrée nécessaires

Les spécifications du langage.

Documents réalisés

Un document expliquant les choix, leurs avantages et leurs inconvénients, notamment en ce qui concerne l'évolutivité du langage, sera attaché à celui-ci. La liste des mots clé et des règles de production de notre grammaire feront l'objet d'une annexe à lequelle se référera le document précédent.

Conditions de passage à la phase suivante

Cette phase ne s'achèvera que quand le langage aura été défini. Il devra avoir été présentée aux tuteurs qui devront le critiquer. Les remarques devront être étudiées afin de définir si quelque chose peut-être ajouté ou supprimé. Après ces éventuelles modifications, la prochaine phase pourra débuter.

A.5.5 Conception de l'architecture des traducteurs

Cette phase suit la phase de définition du langage.

Activités

Il s'agit en premier lieu de présenter une architecture globale du système logiciel présentant les interactions entre les trois composantes. Ensuite, pour chaque traducteur, il faut définir une architecture interne, c'est à dire que nous allons décrire la manière de fonctionner des traducteurs, identifier des modules autonomes et décrire leurs intéractions. Cette phase nous permettra de garantir une bonne modularité des traducteurs. Il sera réalisé une maquette de ces architectures pour pouvoir tester la cohérence de celle-ci.

Points d'entrée nécessaires

Il faut avoir à sa disposition dans cette phase toutes les spécifications de la phase précédente. Le binôme devra en outre avoir acquis une bonne maîtrise du langage VHDL afin de pouvoir obtenir une traduction de qualité.

Documents réalisés

Pour l'architecture globale du système, un document sera réalisé afin d'expliquer les choix pris dans cette architecture ainsi que les avantages et les inconvénients de celle-ci. Il en sera de même pour le découpage des traducteurs. Ces

documents contiendront des figures pour présenter l'architecture. Chacun fera l'objet d'un chapitre.

Conditions de passage à la phase suivante

Pour passer à la phase suivante, l'architecture des deux composantes devra être définie. Elle devra, en outre, avoir été soumise à critique auprès d'un tuteur au moins afin que des remarques puissent être émise pour la dernière fois. Ces remarques devront être discutées et cela pourra alors entraîner des modifications quant à la structure du logiciel.

A.5.6 Écriture du système logiciel

Départ dès que la conception est finie.

Activités

Chaque module identifié lors de la conception va être implementé par le binôme. Chacun va faire l'objet de la définition d'un plan de recette, une écriture, une vérification et d'une documentation de l'interface. Cette procédure va nous permettre d'avoir un suivi du code aisé dans le cycle de vie du logiciel.

Points d'entrée nécessaires

Tous les documents produits dans la définition de l'architecture serviront de base à ce travail qui consistera à répartir le travail entre les protagonistes du binôme et M. Tison.

Documents réalisés

Pour chaque module implémenté, une documentation qui fera partie d'une annexe consacrée aux manuels. Pour certains d'entre eux, une documentation plus précise expliquant les choix dans les algorithmes ou dans les structures de données sera intégré à un chapitre sur l'implémentation des modules.

Conditions de passage à la phase suivante

La phase suivante ne sera accessible qu'au moment où tous les modules seront implémentés, toute la documentation sera produite et où ils auront tous été testés.

A.5.7 Intégration et mise en service du logiciel

Début quand tous les modules sont implémentés, documentés et vérifiés.

Activités

Intégration de chaque module pour créer les deux traducteurs. Tests sur les traducteurs. Debogage. Mise en place du système logiciel pour utilisation. Vérification du fonctionnement sur au moins un exemple réel. Présentation du résultat aux tuteurs.

Points d'entrée nécessaires

Tous les modules implémentés.

Documents réalisés

Les traducteurs seront décrits au travers de leur manuel d'utilisation, joints au présent document. Enfin, des notes à l'attention des programmeurs décriront les éventuelles particularités du code et les manières de faire évoluer le programme.

Condition de fin du projet

Toute la documentation devra être produite et le logiciel mis en place et prêt à fonctionner. Le produit final devra être montré aux tuteurs. Le présent document sera bouclé.

A.6 Gestion des modifications

A.6.1 Modifications dues à des erreurs

Les erreurs peuvent intervenir à plusieurs niveaux dans le système logiciel. Les procédures à suivre sont différentes selon le cas. Voici la description des procédures à suivre en cas d'erreur au niveau de:

Spécifications des programmes: c'est un type d'erreur assez grave, deux cas peuvent se présenter. Si l'erreur est découverte avant la phase d'écriture des modules, alors on peut peut encore y revenir si une décision est prise en ce sens dans une réunion du binôme. Si l'erreur est découverte après la phase de conception de l'architecture alors aucune modification ne sera permise. Dans tous les cas, une fiche de détection d'erreur devra être jointe au présent document; si une solution a été mise en oeuvre alors elle fera partie de cette fiche. Cette fiche montrera les conséquences de cette erreur sur le produit final.

Grammaire du langage: c'est certainement l'erreur la plus grave. Si elle est découverte avant la phase d'écriture des modules alors elle pourra être rattrapable si une décision de correction est prise par le binôme. Si elle est découverte pendant ou après cette phase, on va estimer qu'il est trop tard pour revenir en arrière. De même que précédemment, une fiche sera éditée.

A.6.2 Modifications pour évolution

Comme nous l'avons décrit précédemment, les spécifications du langage sont figées juste après la création du cahier des charges. Une évolution proposée après cette phase sera refusée sauf si l'évolution est mineure[3] et alors l'évolution sera mise en oeuvre ainsi que la rédaction d'une fiche d'évolution qui la décrira. Cette fiche devra identifier le point qui a subit une évolution, préciser si cette évolution concerne une fonction existante ou entraîne une création, et enfin décrire cette évolution (choix, avantages, inconvénients).

A.7 Méthodes, outils et règles

A.7.1 Méthodes et outils

On ne va imposer que deux outils et une méthode. Tout d'abord chaque document sera rédigé sous la forme d'un fichier LaTeX. Ensuite, les fichiers des programmes devront être édités à l'aide du logiciel *Emacs*. Enfin, la grammaire sera définie à l'aide de la notation *Backus-Naur*.

A.7.2 Règles et normes à respecter

Les seules règles et normes que nous allons imposer dans ce plan d'assurance qualité concernent l'implémentation. Tout d'abord, le langage utilisé pour implémenter les traducteurs sera le langage C. Chaque module devra être séparé en deux fichiers dont un sera l'interface proposée (fichier *header '.h'*) et l'autre le fichier de code proprement dit (fichier *'.c'*). L'indentation utilisée sera celle du '*c-mode*' du logiciel *Emacs*. Il y aura, dans les fichier d'implémentations au moins un commentaire toutes les cinq lignes. Chaque fonction citée dans les fichiers d'interface devra être commentée par les informations suivantes: description succincte du service rendu et des ses entrées/sorties. Toutes les fonctions des fichiers d'implémentation qui ne sont pas décrites dans les fichiers d'interface devront l'être dans ceux-ci. La description de ces fonctions se limitera à la descriptions des entrées/sorties.

A.8 Reproduction, livraison

L'ensemble logiciel que nous allons produire ne sera pas destiné à être reproduit. Dans l'éventualité où il devrait l'être, les conditions de reproduction devront être négociées entre le binôme et M. Zénatti. La distribution de ce logiciel se fera sous la forme d'une archive *.tgz*[4] qui est un format très répandu sur les systèmes *UNIX*. Cette archive ne contiendra que les sources et le logiciel devra donc être compilée grâce à un '*Makefile*' qui sera créé par un fichier '*configure*'. D'autre part, il ne sera pas permis de ne reproduire ou de distribuer une partie seulement du système.

3. Une évolution sera qualifiée de mineure si elle n'entraine pas de modifications dans les spécifications du système.

4. Archive crée avec le logiciel *tar* et compressée avec le logiciel *gzip*.

A.9 Suivi de l'application du plan qualité

Nous allons ici décrire les moyens mis en oeuvre par le responsable du plan qualité (cf. Section A.1) pour maîtriser la qualité tout au long du cycle de production.

Le responsable vérifiera par lui même si toutes les conditions de passages entre deux phases sont respectées. Il vérifiera pour chaque module implémenté s'il correspond à la norme décrite en A.7.2. Il contrôlera la qualité de tout document produit qui doit s'intégrer au présent document; pour ce faire, il vérifiera que le document rempli les critères enoncés dans la description des documents produits de la phase auquel il se rapporte (section A.5). La vérification sera donc constante et ainsi elle permet de penser que le produit final respectera ce plan qualité.

Pour donner encore plus de crédibilité au suivi de ce plan qualité, les deux tuteurs sont invités, quand ils le désirent à vérifier son suivi en demandant des comptes au responsable de la qualité. Comme nous l'avons décrit dans la section A.1, certains critères permettent d'obtenir les facteurs de qualité attendus. Nous allons donc décrire ici les différentes procédures à suivre pour vérifier le suivi de ces critères. Au vu des résultats observés quand on applique une des procédures suivantes, nous pouvons savoir si oui ou non la qualité est respectée.

Traçabilité: Les tuteurs choisiront un point particulier de ce plan et le responsable devra être capable de leur fournir tous les documents et/ou produits qui ont déjà du être réalisés relativement à ce point.

Réutilisabilité: Les tuteurs, ou les protagonistes du binôme, devront vérifier certains modules implémentés. Ces derniers devront être autonomes et contenir une interface donnée au travers d'un fichier '*header*'. Il sera observé la plus grande attention sur la qualité de l'interface et de l'éventuelle encapsulation des méthodes qui sont des marqueurs clairs de réutilisabilité.

Modularité: Les auditeurs pourront vérifier que l'implémentation se fait en modules. Des modules qui doivent être documentés et, dans la mesure du possible, vérifiés. Le projet final doit être composé de différents modules compilés séparément. Ces modules devront êtres en accord avec la découpe effectuée lors de la phase de conception (voir en A.5.5). On pourra donc mesurer la modularité en étudiant, avec l'aide du document précédant, la qualité de la découpe modulaire dans le contexte de l'application.

Lisibilité: On pourra mesurer la lisibilité en étudiant tout d'abord l'indentation des fichiers des programmes. Ensuite, la densité de commentaires devra être en accord avec celle préconisée dans en A.7.2, mais on pourra par ailleurs vérifier leur sémantique et donc leur utilité.

Expansibilité: L'expansibilité peut s'observer à trois niveaux différents. Tout d'abord, au niveau de chaque module; chacun devra permettre d'être amélioré en aillant le moins de répercutions possibles sur le reste du programme. Ensuite, au niveau de l'application dans son intégralité. Il s'agira de la faculté de l'architecture logicielle à être améliorée en rajoutant plus de fonctionnalités. Enfin, au niveau du langage, il sera intéressant d'étudier

les éventuelles possibilités d'évolution.

Cohérence: La cohérence se fera par construction. C'est à dire que, puisque toutes les étapes jusqu'à celle de la conception (section A.5.5), sont validées par les tuteurs, on peut être assuré que l'ensemble des trois composantes sera cohérent. On pourra aussi vérifier que tous les documents produits sont aussi cohérents et, enfin, que tous les modules implémentés le sont de la même manière.

Si une défaillance est décelée alors le responsable de la qualité devra tout mettre en oeuvre pour combler le manque, quitte à stopper le déroulement du processus de développement; il sera de plus rédigé une fiche de défaillance ponctuelle de la qualité qui sera joint à ce document dans une annexe.

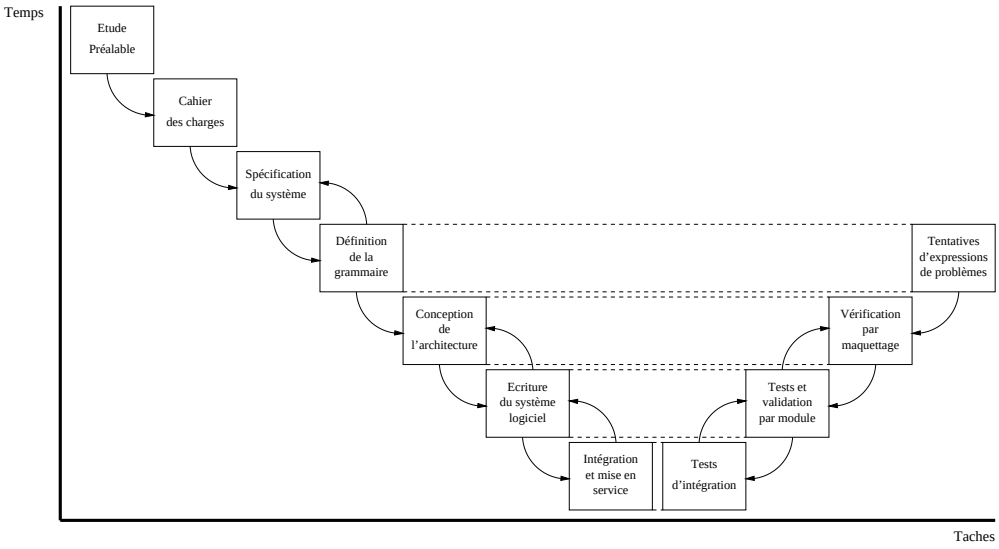

FIG. A.1 – *Plan de développement du logiciel*

Annexe B

Cahier des charges

B.1 Besoins et contraintes

Durant la phase d'analyse préalable, nous avons pu identifier les besoins et les contraintes liés à chaque partie du projet. Ces derniers seront à la base de tout notre travail. Ils vont nous être extrêmement utiles lors des phases de conception.

B.1.1 Par rapport au langage

Besoins

La principale motivation qui a été à l'origine du lancement de ce projet est la réduction du temps de développement des automates de pilotage d'expériences, notamment dans le domaine de la micro-biologie. Le langage devra donc permettre un accès et une mise en oeuvre rapide des fonctionnalités requises pour la description des systèmes de contrôle d'expériences automatisées. Les utilisateurs auront besoin de décrire un système de tâches concurrentes. Il serait souhaitable que le langage soit doté de constructions syntaxiques simples qui ne diffèrent que peut d'autres langages existants afin que les utilisateurs non-informaticiens puissent quand me l'utiliser.

Contraintes

La principale contrainte liée au langage est due au fait que le niveau d'abstraction soit être suffisamment élevé pour qu'un utilisateur type décrive son expérience sans se soucier des contraintes techniques. Il doit seulement décrire le pourquoi et jamais à se demander comment. Une deuxième contrainte impose que le langage soit indépendant de toute architecture cible. Il doit être possible, à partir du même code source, de compiler le programme aussi bien pour un système embarqué que pour un P.C.. Ensuite, un point important pour la réussite de ce projet est que l'ensemble logiciel soit avant tout utilisable. Dans notre cas, cela correspond au fait que le langage doit être adapté à l'utilisateur cible défini en B.2.

B.1.2 Par rapport aux traducteurs

Besoins

En ce qui concerne les traducteurs, le besoin de compilateurs très faciles d'utilisation, de même que le langage, est très vif. Il faudrait aussi que la traduction en VHDL donne une entité simulable et synthétisable afin de pouvoir vérifier le bon fonctionnement du système avant de passer à une éventuelle réalisation. Au niveau du code C généré, il faudra, dans la mesure du possible, qu'il soit le plus efficient possible.

Contraintes

Nous avons tout d'abord identifié des contraintes liées à l'existant. En effet, des travaux précédents ayant déjà étés réalisés dans le domaine qui nous concerne, il était utile d'étudier ce qui a été fait pour éventuellement en réutiliser une partie. En fait, en dehors des outils trouvés sur Internet, nous n'avons récupéré que des drivers écrits pour piloter une carte LABPC+ de chez National Instruments sous un environnement Windows. Nous étudierons la possibilité de les réutiliser dans la partie consacrée à la conception de l'architecture.

B.2 Utilisateur type

Le langage doit pouvoir être utilisé par un scientifique non-informaticien, un biologiste ou un chimiste par exemple. Nous estimons qu'il est plus raisonnable de nécessiter que l'utilisateur possède des compétences dans les domaines de la micro-électronique et de la programmation. Pour être plus précis, l'utilisateur type devra connaître au moins l'architecture matérielle qu'il utilise. Ses connaissances en informatiques devront être un peu moins importantes. Elles pourront se limiter à une initiation aux langages de programmation et à des notions sur les tâches concurrentes. Il semblerait enfin que des notions plus grandes de l'informatique pourront lui permettre d'obtenir des automates plus performants.

B.3 Plateformes cibles

Le langage permettra de spécifier des systèmes de contrôle d'expériences et le traducteur vers le langage C devra permettre d'utiliser ces spécifications sur différents systèmes hôtes. La diversité de ces derniers est pour ainsi dire infinie et notre langage doit permettre de spécifier des systèmes pour chacune de ces plateformes. Nous allons ici présenter deux exemples de plateformes envisageables.

B.3.1 Plate–forme PC/LABPC+

Piloter une expérience automatisée avec un ordinateur requiert, en plus des outils fournis par le PC, l'utilisation de composants matériels traduisant les données envoyées par le robot en données compréhensibles par la machine

et vice-versa. Il s'agit en général de conversions Analogique/Digital et Digital/Analogique. Ces outils nous sont offerts par la carte National Instruments LABPC+ que nous utiliserons d'une part comme interface entre l'ordinateur et le robot, et d'autre part comme générateur d'un signal d'interruption périodique. De telles conversions prennent du temps, de mêmes que positionner les sorties du système en fonction des entrées. Il est donc primordial que toutes ces opérations soient synchronisées sur un signal commun de période connue.

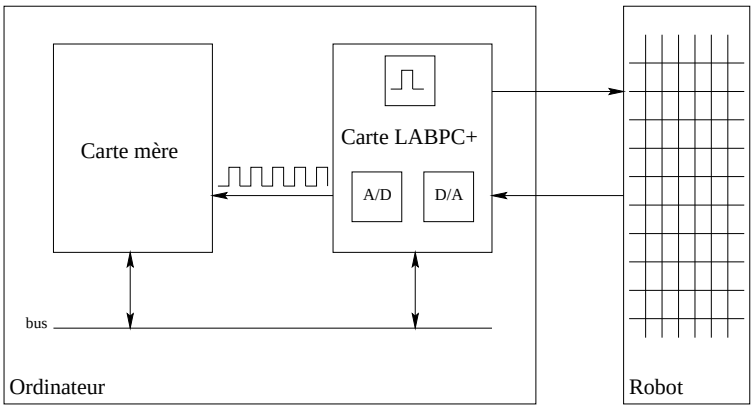

FIG. B.1 – *Plate-forme PC/LABPC+*

Une telle plate–forme est très souple à utiliser et permet d'obtenir rapidement une interface graphique agréable à l'utilisateur, cependant elle l'est un peu moins du point de vue du contrôle des contraintes de temps des fonctions implantées. En effet, le système d'exploitation (que ce soit Windows ou Linux) gère toutes les interruptions du matériel présent sur le PC et ne considère celle de la carte que comme une parmis les autres. Or chaque évènement produit par le système expérimental doit être traité en une période connue: celle générée par le timer de la carte. Ainsi, en plus de la gestion parallèle des tâches du robot, il faut aussi gérer le système d'exploitation de notre ordinateur. Pour éviter qu'une interruption soit traitée trop tard par le système, il faut faire en sorte que le système d'exploitation ne prenne plus que le signal du timer de la carte en compte. La technique est simple:

- sauvegarder le vecteur d'interruptions
- désactiver toutes les interruptions logicielles et matérielles
- activer le timer de la carte
- exécuter le programme de pilotage de l'expérience
- désactiver le timer
- réactiver les interruptions logicielles et matérielles
- restaurer le vecteur d'interruptions

Cependant cette méthode ne satisfait pas le cas où des processus gérés par le système d'exploitation doivent terminer dans un temps imparti. Nous considérerons que la machine qui fait tourner un programme de pilotage d'expérience ne devra faire tourner que ce programme, et ne recevra, pendant le temps de l'expérience, que l'interruption périodique de la carte.

B.3.2 Système embarqué avec micro-contrôleur HC16

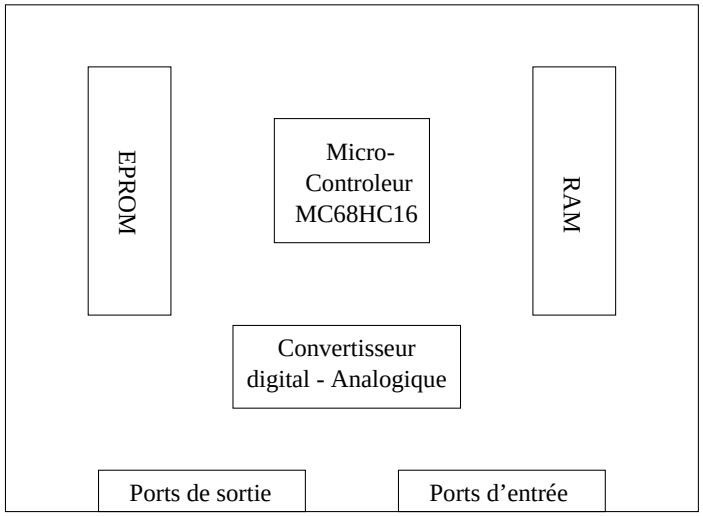

FIG. B.2 – *Exemple de carte contenant un micro-contrôleur HC16*

Il est très intéressant de pouvoir piloter un automate grâce à un système embarqué car cela permet d'obtenir plus de mobilité et plus d'autonomie. Le CNRS met à notre disposition une carte comprenant un micro-contrôleur HC16 ainsi que d'autres composants lui permettant de fonctionner. Un des gros avantages d'une telle plateforme vient du fait qu'elle ne fait fonctionner que notre système, nous permettant de pouvoir gérer plus aisément les contraintes de temps liés au contrôle d'une expérience. Un autre de ses avantages est la possibilité de réutilisation de ce type d'architecture. En effet, si on veut faire évoluer le programme contenu sur la carte ou bien si on veut tout simplement réutiliser cette carte pour une autre application, il suffit simplement de changer la R.O.M.[1] pour faire totalement autre chose.

Le micro-contrôleur HC16 peut être vu comme une boite contenant un processeur capable d'effectuer les opérations classiques ainsi qu'un panel d'outils permettant son utilisation en quasi otartie. Ceux-ci sont, par exemple, un timer programmable[2], un convertisseur analogique/digital, etc...

La carte qui nous est fournie contient donc un micro-contrôleur HC16 ainsi que des composants permettant de rendre le système totalement autonome et intégrable. Nous avons donc de la RAM[3], des ports d'entrée et de sortie qui seront reliés au dispositif expérimental, et bien d'autres composants. Différents modèles de carte existent car il est clair que ce sont les besoins de l'utilisateur qui dicteront la présence de tel ou tel composant.

Nous pouvons donc voir l'intérêt que représente la capacité de notre système de pouvoir générer du code utilisable sur un tel type de plate-forme. En effet, il

1. Read Only Memory
2. Un timer programmable est un outil qui permet de contrôler, par des algorithmes, la génération d'interruptions périodiques
3. Random Acces Memory

semble évident que la majorité des systèmes informatiques dans le futur ne sera pas des ordinateurs mais plutôt des systèmes embarqués dans tous les objets courants qui nous faciliteront la vie. Le meilleur exemple en est la voiture; le nombre de puces qu'elle contient est en augmentation continuelle et notre projet peut très bien s'adapter à la conception, par exemple, d'un système d'aide au freinage, qui est un système embarqué où les notions de temps sont très importantes.

B.4 Plan de recette

Le plan de recette que nous considérons ici est présenté sous la forme d'une description de problème. Ce plan de recette va nous permettre de vérifier que le système que nous allons produire est capable de résoudre des problèmes réels.

B.4.1 Automatisation d'une expérience d'électrochimie

Tout d'abord, cette expérience nécessite l'utilisation d'un potentiostat, d'un convertisseur digital vers analogique, d'un spectrophotomètre, d'un capteur de température et d'un générateur de fonction qui doit fournir une tension dont l'évolution sera automatisée.

Le potentiostat sera réalisé par une équipe du CNRS, nous le considérons donc comme existant et fonctionnel. Le convertisseur utilisé sera considéré comme un convertisseur générique du marché des composants. Le générateur de tension sera réalisé par un programme dans le langage que nous allons définir.

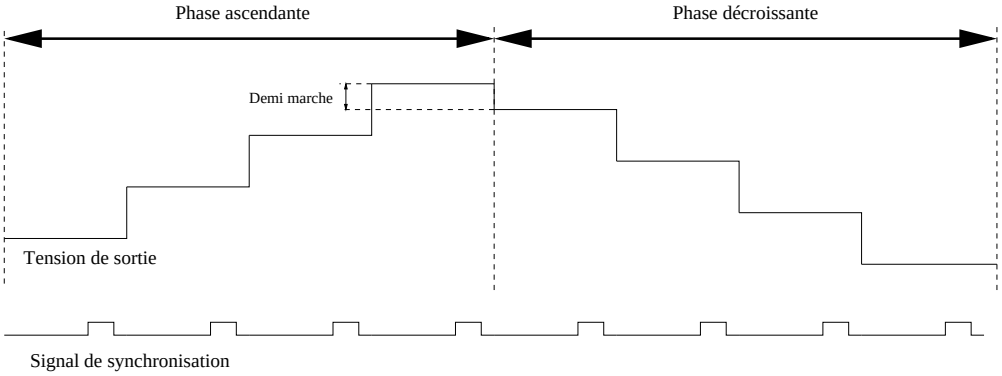

FIG. B.3 – *Tension de sortie désirée et signal de synchronisation*

Les fonctions que devra assurer le système sont diverses. En premier lieu, le générateur de tension doit fournir une tension croissante sous la forme de marches d'escalier, puis décroissante mais la valeur de départ pour cette phase sera diminuée d'une demi marche avant de commencer (voir figure B.3). Par ailleurs, le système devra générer un signal de synchronisation pour le déclenchement d'un spectrophotomètre. L'acquisition de mesures et la communication de celles-ci par un port série en vue d'un traitement sur une autre machine.

Enfin, on va supposer que les paramètres fournis par l'utilisateur sont chargés via un port série au début du contrôle de l'expérience.

La mesure de température sera réalisée par un capteur qui fourni une tension de 10mV par degré centigrade avec une précision d'un quart de degré. Il peut-être intéressant de prévoir une fonction de régulation de température qui peut être un bon exemple de contrôle rétroactif.

Certains paramètres doivent être déterminés par l'utilisateur:

- La tension de départ formulée en mV.
- L'incrément de tension d'une marche à l'autre exprimée en mV.
- Le nombre de marches qui permettrons de déterminer la tension maximum atteinte.
- La durée des paliers sera définie en minutes.
- Délai de déclenchement du spectrophotomètre.

B.5 Conditions relatives à la qualité

Dans le plan d'assurance qualité que vous trouverez dans l'annexe A, nous décrivons les principaux facteurs de qualité qui seront garant de celle de ce projet. Ces facteurs peuvent être considérés comme des contraintes. Les procédures décrites en A.9 nous permettent d'assurer que ces facteurs seront respectés dans le produit final. En particulier, toutes les contraintes relatives à l'évolutivité du système y sont décrites.

Annexe C

Manuel

C.1 Définition de la grammaire du langage

C.1.1 Méta-langage

|	: ou
.	: et
(...)	: priorité d'exécution des règles à l'intérieur des parenthèses
'a'	: caractère a
"a"	: chaine de caractères a
a-b	: a et b appartiennent à un ensemble énuméré {..., a, a_1, ..., b, ...}. a-b est équivalent à a | a_1 | ... | b

Les chaines de caractères en caractères gras représentent les mots clés du langage. Par convention les règles de production commencent par une lettre majuscule.

C.1.2 La grammaire

La grammaire du langage est représentée par un quadruplet (V_N,V_T,$Programme$,P) où

- V_N est l'ensemble des règles non terminales.
- V_T est l'ensemble des règles terminales.
- Programme est la règle initiale.
- P est l'ensemble des règles de production.

Règles non terminales

Constante, Ident, Reel, Entier
Math_Expr, Math_Expr_Bis, Math_ExprA, Math_ExprB
Bool_Expr, E_Bool_Expr
Affectation
Bloc
If, ElseIf
Case, Case_Lig
Macro
Variable, Shared
Type_Var, Type_Shared
Tache
Const
Programme
Corps

Règles terminales

Digit : '0'-'9'
Lettre : 'a'-'z' | 'A'-'Z'
Caractere : Digit | Lettre | '_'
Booleen : "true" | "false"
Plus : '+'
Moins : '-'
Mult : '*'
Div : '/'
Mod : "//"
Et : '&'
Ou : '|'
Inf : '<'
Infe : "<="
Sup : '>'
Supe : ">="
Eg : '='
Diff : "<>"
PtVirg : ';'
Virg : ','
Point : '.'
DeuxPts : ':'
Aff : ":="
ParOuv : '('
ParFer : ')'
AccOuv : '{'
AccFer : '}'
Space : ' ' | tabulation
FinL : retour à la ligne

Ident → Lettre . E_Ident
E_Ident → Caractere . E_Ident | ϵ

Entier → Digit . E_Entier
E_Entier → Digit . E_Entier | ϵ

Reel → Entier . Point . Entier

String → Caractere . String | FinL

Type_Var → **real** | **integer** | **bit** | **vector** . ParOuv . Entier . ParFer
Type_Shared → **port_numerique** | **part_analogique** | **port_serie** | **port_parallele**

Règles de production

Affectation → Ident . Aff . (Math_Expr | E_Bool_Expr | Constante) . PtVirg

Bloc	→ (Affectation | Case | If) . Bloc | Affectation | Case | If
Bool_Expr	→ Math_Expr . Bool_Op . Math_Expr
Case	→ **case** . Math_Expr . **of** . Case_Lig . **endcase** . PtVirg
Case_Lig	→ Constante.DeuxPts.Bloc.PtVirg.Case_Lig | **else**.DeuxPts.Bloc
Const	→ **const** . Ident . Eg . (Math_Expr | E_Bool_Expr) . PtVirg . Const | ϵ
Constante	→ Ident | Reel | Entier
Corps	→ Tache . Corps | Tache
Elseif	→ **elsif** . E_Bool_Expr . **then** . Bloc . ElseIf | **else** . Bloc | ϵ
E_Bool_Expr	→ Booleen | Cond_Expr . Verite_Op . Bool_Expr | Bool_Expr
If	→ **if** . E_Bool_Expr . **then** . Bloc . EndIf . **endif** . PtVirg
Macro	→ **macro** . Ident . DeuxPts . Bloc . **end** . PtVirg . Macro | ϵ
Math_Expr	→ Terme . Math_Expr_Bis
Math_ExprA	→ (Mult | Div | Mod | Et | Ou) . Terme
Math_ExprB	→ (Plus | Moins) . Math_Expr
Math_Expr_Bis	→ (Math_ExprA | Math_ExprB) . Math_Expr_Bis | ϵ
Programme	→ **system**.Ident.DeuxPts.Const.Shared.Macro.Corps.**endsystem**.PtVirg
Shared	→ **shared** . Ident . DeuxPts . Type_Shared . PtVirg . Shared | ϵ
Tache	→ **task**.Ident.DeuxPts.Const.Variable.**begin**.Bloc.**endtask**.Ptvirg | **separate**.**task**.Ident.PtVirg
Terme	→ ParOuv . Math_Expr . ParFer | Constante
Variable	→ **var** . Ident . DeuxPts . Type_Var . PtVirg . Variable | ϵ

C.2 L'unité de description de la cible

c'est le chapitre sur l'unité de description de la cible

C.3 L'unité d'interface

C'est le chapitre sur l'unité d'interface.

C.4 L'unité d'architecture

C.4.1 Le programme

Le programme est composé de quatre blocs séquentiels:

- **Déclaration des constantes:** On déclare ici les constantes du programme, c'est à dire des valeurs figées pour la suite.
- **Déclaration des variables partagées:** Elles représentent les canaux de communication entre les tâches.
- **Les macros:** Ce sont des morceaux de programme; Le macrocompilateur remplacera l'appel à ces macros par le code correspondant.
- **Le corps:** Il est composé par la suite de tâches

C.4.2 Le séparateur

Le séparateur d'instructions est le point-virgule **;**. Les blocs comme ceux des tâches, de la conditionnelle, du programme lui-même, ... sont initiés et cloturés par des mots clés spécifiques. Ces mots clés de fin de bloc marquent une fin et non la séparation d'avec un autre bloc. Pour cette raison, le point-virgule marque aussi la séparation entre les blocs. Ainsi, les mots clés de fin de bloc sont toujours suivis du point-virgule.

C.4.3 Les constantes

Les constantes sont déclarées par le mot clé **const** suivi de l'identificateur représentant cette constante. L'affectation de la constante est faite une seule et unique fois lors de l'élaboration. L'identificateur de la constante est tout de suite suivi du signe d'égalité = puis d'une valeur réelle, entière, booléenne, d'une expression ou d'une chaine de caractères. Les expressions sont évaluées une seule fois, si une constante les compose, elle aura dû être au préalable évaluée donc sa déclaration doit impérativement précéder celle dans laquelle est utilisée en partie droite.

C.4.4 Les macros

Lorsqu'une séquence d'instructions est redondante ou que sa présence alourdie le corps d'un programme, il est utile de faire appel à des macros. Elles sont déclarées par le mot clé **macro** suivi d'un identificateur, de deux points **:** et de la séquence d'instructions suivie de **end**. Le macrocompilateur remplacera l'appel à ces macros par la suite d'instructions composant son corps. Afin d'éviter des problèmes de cycle, on ne pourra pas appeler d'autres macros à l'intérieur d'une macro.

C.4.5 Les tâches

Déclaration

Toutes les tâches sont concurrentes, c'est à dire qu'elles réalisent des séquences d'instructions indépendamment les unes des autres. Ainsi, l'ordre de

déclaration des tâches est peu important. Une tâche est initiée par le mot clé **task** suivi d'un identificateur, de deux points, de la déclaration des constantes et des variables locales, du mot **begin**, des instructions séquentielles la composant et du mot clé de fin **endtask** suivi d'un point-virgule.

Les tâches séparées

Certraines tâches peuvent être très longues ou pénibles à écrire en même temps que les autres. Le système des macros n'est ici pas assez puissant pour soulager le programmeur. Il faut alors laisser la possibilité à celui-ci d'implanter la tâche dans un fichier indépandant. Cette construction est possible en déclarant la tâche comme différée par le mot **separate** suivi de **task**, du chemin complet image de la tâche puis d'un point virgule marqueur de séparation.

Dans ce fichier, la tâche devra être déclarée et implantée comme d'habitude. Le macrocompilateur remplacera cette ligne par le contenu du fichier spécifié.

C.4.6 Les instructions

Nous utiliserons trois instructions:
- l'affectation
- la condition
- le choix

L'affectation

On affecte les variables en les faisant suivre directement par := puis par une expression. En partie gauche se trouve un identificateur préalablement déclaré soit dans la partie déclarative de la tâche pour des variables locales soit en partie globale, relative aux variables partagées.

Le choix

Il est initié par le mot prédéfini **case** suivi d'une expression mathématique évaluée à chaque entrée dans la boucle, de **of**, des choix possibles et de **endcase**. Les choix sont figés lors de l'élaboration et sont de même type que celui de la valeur de l'expression évaluée. Ils sont suivis de deux points et d'un bloc d'instructions séquentielles à réaliser. Un cas par défaut est prévu, il est unique, placé en dernière position et précédé d'au moins un choix. Il est représenté par le mot clé **else** suivi de deux points et d'un bloc d'instructions.

La condition

L'instruction conditionnelle est initiée par le mot clé **if** suivi d'une expression booléenne puis par deux blocs d'instructions séquentielles et terminée par **endif**. Le premier bloc est initié par **then**, les instructions sont exécutées si l'expression booléenne est évaluée à vrai. Le second bloc optionnel représente la suite d'instructions à exécuter si l'expression conditionnelle est évaluée à faux. Dans ce cas la, il est initié par le mot clé **else**.

Pour éviter la lourdeur des expressions conditionnelles en cascade, une construction a été créée à cet effet. Ainsi, le second bloc sera initié par **elsif** suivi d'une expression booléenne et de deux blocs dont la construction est identique à celle de la conditionnelle initiée par **if**. Le cas par défaut commencera par **else** et les instructions le suivant sont exécutées si aucune des conditions n'aura été évaluée à vrai.

C.4.7 Les variables

Les variables partagées

Les variables partagées sont les uniques moyens de communication entre les tâches. Elles sont déclarées par le mot clé **shared** suivi d'un identificateur, de deux points et de son type. Une telle variable peut prendre un des types prédéfinis:

- entier: **integer**
- réel: **real**
- booléen: **bit**
- vecteur de bits: **vector(n)**, $n \in \mathbb{N}$

Le terme de variable partagée entraine les problèmes d'exclusion mutuelle. Ainsi on pose quelques contraintes sur leur utilisation;

1. La lecture est possible par toutes les tâches. On peut donc l'utiliser en partie droite de l'affectation, dans les expressions booléennes ou comme argument de choix
2. l'écriture est autorisée que dans une seule et unique tâche ie elle apparait en partie gauche de l'affectation que dans une seule tâche. Les instructions à l'intérieur des tâches étant séquentielles, rien ne nous empêche d'affecter cette variable plusieurs fois à l'intérieur d'une même tâche.

Les variables locales

Les variables sont locales aux tâches, leur visibilité est restreinte au corps de la tâche. Elles sont déclarées par le mot **var** suivi d'un identificateur, de deux points et du type entier, réel, booléen ou tableau de bits.

Les instructions à l'intérieur d'une tâche étant séquentielles, les variables ne présentent aucun problème de lecture/écriture. Elles pourront donc apparaitre dans toutes les expressions arithmétiques ou logiques, dans le choix et en partie droite ou gauche de l'affectation.

C.4.8 Les ports

Ils sont déclarés dans l'unité d'interface. Ce sont grâce à eux que l'on peut communiquer avec l'extérieur. Les ports ont été préalablement configurés soit en entrée soit en sortie. Selon le type de ports, il peut accepter par défaut le duplex. Lors de l'élaboration, la configuration des ports est figée.

Si le port a été configuré en entrée par le mot clé **in**, il ne pourra jamais apparaitre en partie gauche d'une affectation. De même, s'il a été déclaré en

sortie par **out**, il pourra uniquement être utilisé en partie gauche de l'affectation. Les ports sont des variables tellement particulières que plusieurs types ont dû être spécifier pour leur utilisation: cf paragraphe sur les types non triviaux.

C.4.9 Les types

1. **Les types triviaux**: Les valeurs des variables de ces types sont sauvegardées et accédées par la mémoire. Ils sont uniquement déclarés et utilisés dans le fichier d'architecture.
2. **Les types non triviaux**: Ce sont les types des ports déclarés dans l'unité d'interface et dont les variables sont utilisées dans le fichier d'architecture.

Les types triviaux

- Entier: **integer**, représentant $\mathbb{N}$
- Réel: **real**, représentant $\mathbb{R}$ énumérable
- Booléen: **boolean**, l'ensemble (**false**, **true**)
- Tableau ou vecteur de bits de taille n: **vector(n)**, $n \in \mathbb{N}^*$. Il peut être vu soit comme un entier positif — avec quelques opérations de plus que sur les entiers —, soit comme un tableau de valeurs positionnées à vrai ou faux — états d'un automate, par exemple—.

Les variables de ces types sont accédées ou affectées en mémoire dans la RAM. Les opérations permises sur ces types sont précisées dans le paragraphe sur les expressions et dans celui sur l'affectation. Tous ces types sont ordonnés.

Les types non triviaux

Les données transitant entre le pilote de contrôle de l'expérience et le dispositif expérimental doivent passer par des ports numériques, analogiques, série ou parallèle.

- **port_numerique**
- **port_analogique**
- **port_serie**
- **port_parallele**

Accéder à ces ports n'est pas trivial et doit faire appel à des routines assembleur spécifiques. Ainsi chacun de ces types est une structure à au moins trois champs:

1. **initialisation**: permet lors de l'élaboration de déclarer et d'initialiser le port
2. **lecture**: permet d'accéder au port en lecture et fournit la valeur acquise
3. **écriture**: permet d'accéder au port en écriture et envoie une donnée

Selon le matériel utilisé, ces ports ne pourront pas être en full-duplex, l'utilisateur devra donc le contraindre soit en entrée par le mot clé **in** précédé du type de port soit en sortie par **out**. Il est de plus possible pour les ports analogiques

de certaines cartes de préciser une plage d'utilisation afin que la conversion numérique soit la plus précise possible. On utilisera la syntaxe **constraint** a **to** b, où a et b sont des réels, b>a.

type	in/out	duplex
port_numerique	×	-
port_analogique	×	×
port_serie	-	×
port_parallele	×	×

La lecture et l'écriture sur un port sont transparents à l'utilisateur, il les utilise comme de simples variables. Il ne faut cependant pas perdre de vue que l'accession aux ports suscite fortement le matériel, composante lente du système. L'utilisation abusive de ceux-ci peut nuire gravement aux contraintes de temps imposées par l'expérience. Si un port doit être lu plusieurs fois dans une même tâche, il est fortement conseillé de le faire une seule fois et de stocker sa donnée acquise dans une variable locale de type trivial.

Ces ports étant considérés comme des variables, l'écriture vers ceux-ci se fait donc par une affectation.

C.4.10 Les expressions

Les expressions sont au nombre de deux:

1. **Les expressions arithmétiques**: opérations mathématiques usuelles sur les nombres entiers, les réels et tableaux comme l'adition, la multiplication, ...
2. **Les expressions logiques**: les ensembles représentés par les types réel, entier, vecteur, ...sont ordonnés. Les éléments de ces ensembles sont donc comparables, on peut donc leur appliquer les opérateurs de comparaison comme $<$, $>$, ...Il est de plus possible pour les tests conditionnels, par exemple, de faire des disjonctions, des conjonctions, ...de clauses par **AND**, **OR**, ...

Les expressions arithmétiques

Les expressions arithmétiques sont composées d'opérateurs et d'opérandes. Les opérateurs agissent sur des opérandes de même type et l'expression, une fois évaluée, donne un résultat de type identique à celui des opérandes.

Opérateur sur les entiers

- L'adition: + d'arité 2
- La soustraction: - d'arité 2
- La division entière: / d'arité 2
- Le reste de la division entière: // d'arité 2
- La multiplication: * d'arité 2
- Le signe: +,- d'arité 1

Opérateur sur les réels

- L'adition: + d'arité 2
- La soustraction: - d'arité 2
- La division: / d'arité 2
- La multiplication: * d'arité 2
- Le signe: +,- d'arité 1

Opérateur sur les vecteurs de bits

Le nombre codé par un vecteur de bits peut être considéré comme un entier. On peut donc lui appliquer les mêmes opérateurs que sur les entiers. Il en existe cependant trois autres spécifiques à ce type, représentant les masques:

- L'adition: + d'arité 2
- La soustraction: - d'arité 2
- La division entière: / d'arité 2
- Le reste de la division entière: // d'arité 2
- La multiplication: * d'arité 2
- Le "et" bit à bit: & d'arité 2
- Le "ou inclusif" bit à bit: | d'arité 2
- Le "ou exclusif" bit à bit: ^ d'arité 2

Opérateur sur le type des booléens

Ce type déroge à la règle de stabilité car les opérandes peuvent être de types différent que celui du résultat. En effet, la comparaison de nombres est de type booléen alors que les nombres sont soit des entiers, soit des réels, soit des vecteurs de bits. Les opérateurs utilisés sont donc les mêmes que ceux utilisés dans les expressions logiques du test de la conditionnelle. On retrouvera cependant les trois masques des vecteurs de bit.

Les expressions logiques

Deux types d'opérateurs peuvent être utilisés:

- La comparaison: $<$, $>$, ...
- La concaténation: AND, OR, ...

La comparaison

Les opérateurs de comparaison sont utilisés sur des opérandes de même type dont les valeurs de ce type sont ordonnées: les entiers, les réels, les vecteurs de bits — puisqu'on peut les voir commes des entiers —, les booléens.

Sur les booléens, seuls les opérateurs d'égalité et d'inégalité sont pertinents, les autres ne représentent que peu d'interêt. L'ensemble étant énuméré comme (**false**, **true**), on pourra néanmoins s'il y a nécessité les utiliser.

- Inférieur: $<$
- Inférieur ou égal: $<=$

- Supérieur: >
- Supérieur ou égal: >=
- Egalité: =
- Inégalite: <>

Le résultat des comparaisons est de type booléel **bit**.

La concaténation

Le test d'une expression conditionnelle consiste à vérifier si une expression logique est évaluée à vrai **true** ou faux **false**. Dans le cas où l'on veuille tester plusieurs expressions booléennes, une construction adequat est nécessaire. On peut utiliser les opérateurs de concaténation suivant:

- Toutes les clauses sont vraies: **and**
- Au moins une clause est vraie: **or**
- Une seule clause est vraie: **xor**
- L'expression est fausse: **not**
- Au moins une clause est fausse: **nand**
- Toutes les clauses sont fausses: **nor**

C.4.11 Les priorités

- Les opérateurs **-** et + d'arité 1 sont prioritaires sur tous les opérateurs arithmétiques.
- L'opérateur // est prioritaire sur tous les autres opérateurs arithmétiques d'arité 2.
- Les opérateurs de masque **&**, **|** et **^** sont prioritaires devant +, **-**, ***** et / d'arité 2 et ont le même niveau de priorité.
- Les opérateurs ***** et / dont prioritaires sur + et **-** d'arité 2 et ont le même niveau de priorité.
- Les opérateurs + et **-** ont le même niveau de priorité.
- L'opérateur logique **not** est plus prioritaire que tous les autres opérateurs logiques.
- Les opérateurs logique **and**, **or**, **xor**, **nand**, **nor** ont le même niveau de priorité.

Le priorité est néanmoins la plus forte pour le parenthésage **()**, il règle en outre tous les problèmes.