ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

COMME EXIGENCE PARTIELLE
À L'OBTENTION DE LA
MAÎTRISE EN GÉNIE MÉCANIQUE
M.Ing.

PAR
ETIENNE FORTIN

CONCEPTION D'UNE ARCHITECTURE LOGICIELLE PERMETTANT LE CONTRÔLE DE MACHINES-OUTILS DANS UN CONTEXTE D'OUVERTURE DU CONTRÔLEUR

MONTRÉAL, LE 2 MAI 2003

© droits réservés de Etienne Fortin

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE :

Jean-François Chatelain, directeur de projet
Département de génie mécanique à l'École de technologie supérieure

Louis Rivest, codirecteur
Département de génie de la production automatisée à l'École de technologie supérieure

Louis Lamarche, président du jury
Département de génie mécanique à l'École de technologie supérieure

Roland Maranzana, professeur
Département de génie de la production automatisée à l'École de technologie supérieure

Stéphane Chalut, specialiste développement CFAO
Bombardier aéronautique

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 2 AVRIL 2003

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

CONCEPTION D'UNE ARCHITECTURE LOGICIELLE PERMETTANT LE CONTRÔLE DE MACHINES-OUTILS DANS UN CONTEXTE D'OUVERTURE DU CONTRÔLEUR

Etienne Fortin

SOMMAIRE

L'usinage par commande numérique est un domaine où bon nombre d'innovations ont pu être constatées depuis plusieurs années. Ces innovations ont eu lieu surtout du point de vue mécanique, avec de nouvelles techniques permettant de diminuer le temps d'usinage, améliorer la durée de vie des outils et augmenter la précision et la répétitivité des machines.

Cependant, tous ces changements n'ont pas nécessairement donné lieu à des innovations équivalentes dans la portion logicielle des machines. Celles-ci sont toujours contrôlées à l'aide d'une technologie logicielle vieille de plus de quarante ans. Le langage de programmation accessible à l'usager, les Codes-G, n'a que très peu évolué. Quelques problèmes peuvent être identifiés. Tout d'abord, les Codes-G ne forment en rien un véritable langage de programmation. Certaines extensions ont bien été ajoutées au fil des années, mais elles restent très limitées. De plus, l'ajout de fonctions de plus en plus variées sur les machines-outils favorise la création d'architectures ouvertes où l'usager peut étendre les fonctionnalités du contrôleur et de la machine au besoin. Les Codes-G n'offrent cependant qu'un contrôle très sommaire de la machine.

Il apparaît donc que l'utilisation des machines-outils à leur plein potentiel se heurte aux limites de la couche logicielle actuellement utilisée pour accéder à leurs fonctionnalités. Une nouvelle architecture logicielle permettant la mise en oeuvre de contrôleurs à architecture ouverte, l'architecture BNCL (*Basic Numerical Control Language*), a donc été conçue. La base de cette architecture est une machine virtuelle réalisant l'abstraction de l'environnement d'exécution du contrôleur, et un ensemble de matériel virtuel réalisant l'abstraction des particularités physiques de la machine-outil. Grâce à ces deux axes d'abstraction, les développements logiciels effectués sur un contrôleur sont portables. De plus, la prise en charge des extensions personnalisées de la machine-outil par l'utilisateur devient possible. Ce système permet également l'utilisation de tout langage informatique jugé pertinent par l'usager pour peu qu'un compilateur existe pour cette architecture. La liberté de choix est donc totale tant du point de vue de la machine physique que des outils logiciels utilisés. Les avantages escomptés de l'utilisation d'une telle architecture ont fait l'objet d'essais qui ont permis de confirmer la pertinence de cette approche. Cette nouvelle architecture logicielle constitue une contribution importante au domaine de recherche sur le contrôle des machines-outils à commande numérique.

DESIGN OF A SOFTWARE ARCHITECTURE ALLOWING MACHINE-TOOL CONTROL IN AN OPEN ENVIRONMENT

Etienne Fortin

ABSTRACT

Numerical control machining is a domain where a lot of innovations have occurred in the past years. These innovations took place mainly in the mechanics of machines, with new techniques to reduce machining time, improve tool life and enhance the precision and repeatability of machines.

But, these changes did not trigger similar innovations in the software portion of the machines. These are still controlled by a software technology that is more than forty years old. The programming interface accessible to the user, G-Codes, did not evolve much. Some problems can be identified. First of all, G-Codes do not constitute a true programming language. Some extensions have been added during the last years but they remain limited in capabilities. Furthermore, the development of various new functions on the machine-tool favours the creation of open architecture controllers where users can extend the functionalities of the controller and the machine. G-Codes, however, offer limited control over the machine's new functions.

It seems that using the machine-tools at their full potential faces the limits of the software used on controllers today. To solve these problems, a new software architecture allowing the implementation of open architecture controllers, the BNCL architecture (Basic Numerical Control Language), has been developed. The basis of this architecture is a virtual machine which abstracts the execution environment of the controller, and a group of virtual hardware components that abstract the physical particularities of the machine-tool. Combining these two axes of abstraction, software developments on a controller are portable to others. Custom extensions to the machine-tool are also made possible. This system also allows the use of any programming language that the user feels adequate for his work. The advantages resulting from the use of such an architecture have been validated. This new software architecture is an important contribution to the machine-tool control research field.

REMERCIEMENTS

J'aimerais tout d'abord remercier mes directeurs, Jean-François Chatelain et Louis Rivest, tous deux professeurs à l'École de technologie supérieure. Ils ont cru en mes capacités et mon projet dès le début et je leur en suis reconnaissant.

J'aimerais aussi remercier chaleureusement ma copine Nathalie Laforest qui a su être à mon écoute pendant toute la durée de ce projet et qui a été en mesure d'endurer mes quelques moments de découragement. Je la remercie du fond du coeur pour son soutient inconditionnel.

Et finalement, j'aimerais remercier tout particulièrement mon très bon ami Jean-François Chabot. Il a pris beaucoup de son temps pour me faire profiter de sa vaste expérience dans le domaine de l'usinage par commande numérique alors qu'il n'en retirait rien d'autre que la simple satisfaction d'aider un ami. Sans lui, mon travail n'aurait pu être aussi complet.

TABLE DES MATIÈRES

page

SOMMAIRE i
ABSTRACT ii
REMERCIEMENTS iii
TABLE DES MATIÈRES iv
LISTE DES TABLEAUX viii
LISTE DES FIGURES ix
INTRODUCTION 1

CHAPITRE 1 REVUE DE LITTÉRATURE 5
1.1 Contrôleur à architecture ouverte 5
1.1.1 Architecture ouverte versus architecture propriétaire 6
1.1.2 Langages de contrôle des contrôleurs à architecture ouverte 8
1.1.3 Les Codes-G et le contrôle poussé des machines-outils 12
1.2 Éléments de développement d'une architecture logicielle 15
1.2.1 La portabilité logicielle 16
1.2.2 L'extensibilité logicielle 17
1.2.3 La réactivité aux événements 18
1.2.4 Utilisation de plusieurs langages de programmation 19
1.3 Retour sur la revue de littérature 20

CHAPITRE 2 SURVOL DE L'ARCHITECTURE BNCL 22
2.1 Organisation conceptuelle 23
2.1.1 Jeu d'instructions BNCL 25
2.2 Portabilité logicielle 28
2.3 Extensibilité de la machine-outil 29
2.3.1 Extensibilité et portabilité 31
2.4 Réactivité aux événements 33

2.5 Utilisation de plusieurs langages de programmation 34
2.6 Résumé des éléments constitutifs de l'architecture BNCL 39

CHAPITRE 3 DÉTAILS DE L'ARCHITECTURE BNCL 41
3.1 Outils de développement utilisés 41
3.1.1 *Component Object Model* et *Object Linking and Embedding* 42
3.1.2 *Extensible Markup Language* (XML) 43
3.1.3 *Extensible Stylesheet* (XSL) 45
3.1.4 *Practical Extraction and Report Language* (Perl) 47
3.1.5 Cygnus Windows 48
3.1.6 Traitement multiprocessus 49
3.2 Spécification du jeu d'instructions 50
3.3 Machine virtuelle BNCL (BNCL Virtual Machine) 51
3.3.1 Fonctionnement 51
3.3.2 Fichier de configuration 53
3.4 Matériel virtuel BNCL (BNCL Virtual Hardware) 54
3.4.1 Mise en oeuvre 54
3.4.2 Fonctionnement 56
3.5 Répertoire de modules 57
3.6 Assembleur BNCL 57
3.6.1 Mise en oeuvre 58
3.7 Compilateur C 59
3.7.1 Mise en oeuvre 60
3.7.2 Fonctionnement 62
3.8 Utilisation des outils et techniques développés 63
3.8.1 Étapes de préparation 63
3.8.2 Étapes de réalisation 64
3.9 Résumé 65

CHAPITRE 4 BÉNÉFICES ESCOMPTÉS DE L'ARCHITECTURE BNCL 68
4.1 Contrôleur à architecture ouverte 68
4.1.1 La portabilité logicielle 69
4.1.2 L'extensibilité matérielle 72
4.1.3 La réactivité aux événements 73
4.1.4 Le support multi langage 74
4.2 Améliorations apportées par rapport au système actuel 75
4.2.1 Librairies de procédures 76
4.2.2 Structures de langage de programmation 77
4.2.3 Performances 77
4.3 Résumé 78

CHAPITRE 5 INTÉGRATION INITIALE AUX PROCESSUS EXISTANTS 80
5.1 Langage APT et logiciels de FAO 82
5.1.1 Méthodes d'intégration en amont 83
5.2 Codes-G 85
5.2.1 Méthodes d'intégration en aval 85
5.2.2 Conversion des Codes-G 86
5.3 Standard ISO14649 STEP-NC 88
5.3.1 Méthodes d'intégration 88
5.4 Résumé 90

CHAPITRE 6 VALIDATION DE L'ARCHITECTURE BNCL 92
6.1 Performance 93
6.1.1 Tests de performance réalisés 95
6.1.2 Analyse 98
6.2 Tests d'extensibilité 101
6.2.1 Analyse 104
6.3 Usinage de came 105

6.3.1 Objectifs et description du test 106
6.3.2 Scénario 107
6.3.3 Résultats et analyse 116
6.4 Résumé 118

DISCUSSION 119
CONCLUSION 122
RECOMMANDATIONS 126
ANNEXES
1 : Aide-mémoire des instructions BNCL 127
2 : Fichier de configuration du test d'extensibilité 135
3 : Definitions liées au programme ConsoleBVM 138
4 : Definitions liées au programme Thermocouple_Reader_BVH 176
5 : Fichier de description du BVH ThermocoupleReader 202
6 : Source du module BNCL utilisé avec le thermocouple 207
7 : Programme écrit en Codes-G paramétriques pour l'usinage de la came 212
8 : Programme écrit en langage C pour l'usinage de la came 218
BIBLIOGRAPHIE 229

LISTE DES TABLEAUX

Page

Tableau I Exemples de langages et de leurs possibles utilisations 75
Tableau II Évaluation des pertes de performance de chaque composante 97
Tableau III Positions générées pour chaque scénario de mesure de performance 98
Tableau IV Impact des composantes sur les performances de l'architecture BNCL .. 100
Tableau V Valeurs des paramètres pour l'usinage de la came 114
Tableau VI Méthodes utilisées dans la programmation de la came 115
Tableau VII Vitesses possibles de l'outil dans le test d'usinage d'une came 117

LISTE DES FIGURES

Page

Figure 1 Exemple de contrôle de machine à l'aide des Codes-G ... 12
Figure 2 Construction IF/THEN/ELSE en Custom Macro B de Fanuc ... 15
Figure 3 Abstraction de la machine-outil par la BVM et le BVH ... 24
Figure 4 Exemple de code source BNCL ... 26
Figure 5 Exemple de routine en assembleur x86 ... 27
Figure 6 Routine écrite en C pour calculer la distance entre deux points ... 27
Figure 7 Analogie entre l'architecture PC et l'architecture BNCL ... 30
Figure 8 Exemple d'extension par l'usager d'une machine-outils ... 31
Figure 9 Implications de la portabilité et de l'extensibilité ... 32
Figure 10 Réactivité aux événements de l'architecture BNCL ... 34
Figure 11 Cheminement de l'information dans le support multi-langage ... 36
Figure 12 Interopérabilité des langages dans l'architecture BNCL ... 38
Figure 13 Inclusion de documents grâce à COM et OLE ... 43
Figure 14 Exemple de fichier XML ... 44
Figure 15 Transformation de données XML à l'aide de XSL ... 46
Figure 16 Génération de la machine virtuelle BNCL ... 53
Figure 17 Génération automatique du code d'implémentation d'un BVH ... 56
Figure 18 Interaction entre un compilateur et l'assembleur BNCL ... 58
Figure 19 Utilitaire d'assemblage BASM ... 59
Figure 20 Création d'un compilateur C à l'aide de GCC ... 61
Figure 21 Génération d'un module BNCL exécutable à l'aide de GCC ... 62
Figure 22 Diagramme de séquence du test d'extensibilité ... 65
Figure 23 Résumé des composantes et utilitaires développés ... 66
Figure 24 Masquage par la BVM des particularités logicielles du contrôleur ... 70
Figure 25 Masquage par le BVH des particularités matérielles de la machine-outil ... 72
Figure 26 Transformation du modèle produit en instructions de programmation ... 81
Figure 27 Intégration de l'architecture BNCL au processus de programmation ... 84

Figure 28 Génération de Codes-G pour différentes machines-outils 86
Figure 29 Transformation de Codes-G entre différents contrôleurs 87
Figure 30 Intégration de l'architecture BNCL au standard STEP-NC 89
Figure 31 Flux d'information dans l'architecture actuelle 94
Figure 32 Nombre de points requis pour des trajectoires linéaires et curvilignes 95
Figure 33 Scénarios de test de performance de l'architecture BNCL 96
Figure 34 Scénario d'ajustement de la vitesse de la broche 102
Figure 35 Simulation de l'ajustement de la vitesse de la broche 103
Figure 36 Photo du montage réalisé pour le test d'extensibilité 104
Figure 37 Photo de l'interface usager prise lors du test d'extensibilité 105
Figure 38 Test d'usinage de came à l'aide de trois méthodes différentes 108
Figure 39 Came définie pour le test d'usinage dans un contexte de production 109
Figure 40 Déplacement du galet en fonction de l'angle de rotation 110

LISTE DES ABRÉVIATIONS ET SIGLES

HSM *High Speed Machining*. Usinage à grande vitesse à l'aide d'un outil de coupe.

RPM *Révolutions par minute*. Nombre de tours effectués par l'outil en une minute.

IPM *Inches Per Minute*. Avance d'un outil de coupe en nombre de pouces parcourus en une minute.

BNCL *Basic Numerical Control Language*. Langage ressemblant au jeu d'instructions d'un microprocesseur, à la base de l'architecture BNCL.

IBM *International Business Machine*. Entreprise oeuvrant dans le domaine informatique.

PC *Personnal Computer*. Ordinateur personnel.

BVH *BNCL Virtual Hardware*. Matériel virtuel BNCL. Composante logicielle effectuant l'abstraction de la machine-outil.

BVM *BNCL Virtual Machine*. Machine virtuelle BNCL. Composante logicielle faisant abstraction de l'environnement d'exécution du contrôleur.

COM *Component Object Model.* Standard définissant l'infrastructure pour la création et l'utilisation d'objets logiciels.

OLE *Object Linking and Embeding*. Standard complétant COM et permettant aux applications de communiquer des données entre elles.

XML *Extensible Markup Language*. Langage permettant de définir son propre format de données sous forme d'étiquettes semblables au html.

HTML *Hypertext Markup Language*. Langage permettant de définir la présentation de données. Utilisé sur la toile d'internet.

Perl *Practical Extraction and Report Language*. Langage script permettant d'automatiser des tâches complexes.

Mo *Méga-octets*. Unité de mesure représentant 1048576 octets (*bytes*).

INTRODUCTION[1]

Comme dans tous les domaines industriels de l'économie de marché actuelle, l'usinage par commande numérique a connu de nombreux changements au cours des dernières années. Les constructeurs offrent des machines de plus en plus complètes et de mieux en mieux intégrées à l'ensemble du réseau informatique du client. De plus, ces machines sont beaucoup plus performantes et offrent une bien meilleure précision que celles qui étaient disponibles il y a dix ans à peine. Par exemple, au début des années 1980 le terme *high-speed machining* (HSM), qui fait référence à l'enlèvement ultra-rapide de matériel sur une pièce à l'aide d'un outil, n'existait pas encore. Les vitesses de rotation des machines étaient de l'ordre de 5000 révolutions par minutes (RPM) et l'avance des outils ne dépassait guère les 20 pouces par minute (IPM). Aujourd'hui, par contre, des machines ayant des vitesses de révolution de l'ordre de 35000 RPM et permettant à l'outil d'avancer à des vitesses aussi élevées que 750 à 1000 IPM ne sont plus du domaine de la science-fiction. Les outils de coupe eux-mêmes ont subit des changements importants, surtout du point de vue des matériaux utilisés et de leur géométrie. Tous ces changements ont évidemment pour but d'augmenter la productivité de l'ensemble du processus de fabrication et ainsi diminuer les coûts.

Cependant, une portion du domaine de l'usinage par commande numérique n'a connu que peu de changements depuis ses débuts, soit le lien logiciel qui existe entre l'usager et la machine-outil. Ce lien, qui permet à l'usager de programmer les mouvements et le comportement de la machine-outil, est à peu de choses près le même que celui utilisé sur les premières machines-outils à commande numérique du milieu des années 1950. Cela est d'autant plus curieux que le domaine logiciel a connu des mutations et des améliorations continuelles, dues en partie au rapide déploiement des ordinateurs dans la population. Malgré la similitude entre la programmation d'un ordinateur et la programmation d'un contrôleur de machine-outil à commande numérique, qui dans les

[1] Les références pertinentes concernant les points abordés dans l'introduction se trouvent au Chapitre 1.

faits n'est qu'un ordinateur dédié à une tâche bien précise, le processus de programmation de ces machines repose encore aujourd'hui sur le vieux langage de contrôle Codes-G.

Les Codes-G sont des instructions de contrôle de machines conçus lors des premiers balbutiements de l'usinage par commande numérique. Les objectifs de conception de ces codes sont donc reliés aux réalités de ces premières machines pour répondre aux besoins de l'époque. Il est intéressant de mentionner que la forme simple et séquentielle de ces codes est intimement liée au fait que le support matériel disponible à l'époque était une bande perforée et un lecteur optique. Aujourd'hui le support matériel n'est évidemment plus une bande perforée, mais l'héritage de cette époque est toujours présent. Plusieurs caractéristiques de ces instructions font en sorte qu'elles sont mal adaptées aux nouveaux développements des contrôleurs de machines à commande numérique, et en particulier aux contrôleurs à architecture dite ouverte. Ces contrôleurs, dont le principe veut qu'ils soient flexibles et extensibles, doivent s'appuyer sur une architecture logicielle leur permettant de réaliser ces objectifs. Le langage de programmation le plus répandue présentement est toutefois les Codes-G. Ceux-ci ne sont ni extensibles, ni adaptés et n'offrent pas les performances, la portabilité et les fonctionnalités nécessaires pour être un candidat potentiel comme langage de programmation de base d'un contrôleur à architecture ouverte.

Même si les machines-outils actuellement sur le marché démontrent des caractéristiques mécaniques fort impressionnantes, les usagers n'ont pas la possibilité d'exploiter à fond ces fonctionnalités en raison de l'absence d'une architecture logicielle adéquate. La productivité de l'ensemble du processus de fabrication s'en trouve donc sous-exploitée. La décision de développer un ensemble d'outils permettant d'ouvrir le contrôleur de la machine-outil à l'usager, par une architecture logicielle adéquate, et ainsi lui permettre d'exploiter plus efficacement son investissement a donc été prise.

Cette architecture logicielle permet la portabilité des programmes de coupe entre des machines offrant des caractéristiques similaires, mais également la portabilité d'applications complexes affectées directement au contrôle de la machine. Ces modules, dans une architecture ouverte, peuvent être programmés par l'usager et la portabilité de ces modules entre les machines est donc souhaitable. Par exemple, une routine implémentant un cycle d'usinage personnalisé à une entreprise n'a pas à être développé pour chaque machine possédée par celle-ci.

Il était également souhaitable d'éviter de lier l'architecture à un langage de haut niveau en particulier. Un langage est dit de haut niveau lorsque sa syntaxe le rapproche plus du langage naturel que du langage primaire des circuits électroniques d'un ordinateur. L'histoire de l'informatique montre que les langages de programmation de haut niveau peuvent considérablement changer avec le temps. Être dépendant d'un seul langage peut constituer un frein à l'évolution future.

Cette architecture doit, de plus, bien s'intégrer avec les technologies et standards présents ou en cours de développement. Finalement, cette architecture doit permettre l'expression de toutes les caractéristiques propres à un contrôleur à architecture ouverte, dont la flexibilité et l'extensibilité, tant matérielle que logicielle. Le résultat de ce travail de conception est l'architecture BNCL, ou « Basic Numercial Control Language ».

La motivation initiale derrière cette démarche est basée sur l'observation que l'architecture actuellement exposée par les machines-outils à commande numérique est inadéquate et empêche l'utilisation de ces machines à leur plein potentiel. Cette hypothèse est au centre de l'étude et a mené à la création d'une toute nouvelle architecture logicielle pour le contrôle des machines-outils à commande numérique. L'hypothèse est maintenant posée qu'une architecture logicielle comportant une abstraction de la machine-outil, un langage de bas niveau pouvant être ciblé par une multitude de langages de programmation, et un protocole bien défini d'accès aux

périphériques de la machine permettrait d'augmenter la valeur productive des machines-outils à commande numérique.

Le premier chapitre de ce mémoire présente la revue de la littérature et l'état de l'industrie aujourd'hui. La poursuite de l'exposé se fera par une familiarisation avec l'architecture BNCL. Il sera question de l'organisation générale de l'architecture et de son fonctionnement. Le chapitre trois reprend les grandes lignes du précédent, mais en explicitant les détails techniques de la mise en oeuvre de l'architecture BNCL. Ces détails techniques, regroupés sous l'appellation *architecture BNCL de référence*, forment un exemple de réalisation des concepts développés au cours du projet. Au chapitre quatre, il sera question des avantages de la nouvelle approche mise de l'avant par ce projet. Des liens avec l'étude de la littérature exposée précédemment, ainsi que les contributions apportées par l'architecture proposée, seront établies. Cet exposé donnera lieu à un chapitre sur la place de l'architecture BNCL par rapport aux autres technologies utilisées ou présentement développées, tel le standard ISO14649 STEP-NC. Ce chapitre discutera de l'intégration de l'architecture BNCL par rapport à ces technologies et de son interaction possible avec l'ensemble des outils de programmation des contrôleurs. Au chapitre six, il sera question de la validation de l'architecture BNCL. L'accent sera mis sur la facilité de programmation du contrôleur de la machine par rapport aux langages actuellement disponibles. Différents scénarios seront étudiés. Une portion de cette validation portera sur les performances de l'architecture. Finalement, une discussion sur les résultats de la validation sera proposée, ainsi qu'une conclusion sur l'ensemble du projet. Les recommandations suivront dans la dernière partie du travail.

CHAPITRE 1

REVUE DE LITTÉRATURE

Le projet de développement d'une architecture logicielle pour le contrôle de machines-outils à commande numérique touche à plusieurs domaines qui ne sont pas naturellement associés. Ce développement touche le contrôle physique de la machine, où des considérations de très bas niveau de communication avec l'électronique de la machine-outil doivent être prises en compte. Ce développement touche également à la conception de logiciels et aux principes qui s'y rattachent. L'étude de l'architecture actuellement utilisée, et des efforts de changement reliés à l'évolution de cette architecture, passe donc par une revue tant de la recherche portant sur les contrôles de machines-outils que sur des concepts mis de l'avant dans le développement informatique.

1.1 Contrôleur à architecture ouverte

Le concept d'ouverture du contrôleur vise à éliminer les barrières et les contraintes qui empêchent l'usager d'accéder aux informations et spécifications internes de son produit. Plusieurs caractéristiques peuvent être associées à une architecture ouverte. Selon l'objectif des concepteurs d'une architecture donnée, il peut être question de portabilité, de modularité, d'interopérabilité, de flexibilité, d'extensibilité et autres termes se rapprochant. Les caractéristiques définies par Schofield et Wright et qui sont la flexibilité, l'intégration et la standardisation [1], seront toutefois utilisées. Selon ces auteurs, ces trois caractéristiques regroupent toutes les autres. Par flexibilité, ils entendent la capacité de modifier le domaine d'application de la machine dans le temps. Cette flexibilité demande donc que le contrôleur soit modulaire et extensible. L'intégration, quant à elle, traite des échanges de la machine avec son environnement extérieur. Un contrôleur qui présente une bonne intégration pourra facilement coopérer avec d'autres machines présentent sur un même réseau. Par exemple, il pourrait

communiquer avec d'autres machines sur le plancher concernant les données de coupe. Finalement, la standardisation demande la création de composantes qui interagissent bien entre elles et qui sont portables d'une machine à l'autre. Plusieurs travaux montrant les bénéfices et les applications des contrôleurs à architecture ouverte sont présentés dans la littérature [2][3][4][5][6].

Le contrôle de machines-outils à commande numérique n'est pas le premier domaine industriel réalisant les avantages d'une architecture ouverte et les limitations d'une architecture propriétaire. Avant même que l'expression « architecture ouverte » ne soit proposée, des auteurs parlaient déjà de concepts propres à cette approche. Par exemple, Grossman, en 1986, élaborait sur le contrôle adaptatif et l'utilisation intensive de capteurs dans la programmation des machines-outils à commande numérique [8]. Klein, lui, exposait déjà en 1985 les bénéfices escomptés de l'ouverture de l'architecture des contrôleurs, soit l'extension de la base des usagers et par conséquent l'extension de l'étendue du marché des contrôleurs [7]. Klein mentionnait également le gain de productivité probable associé à l'interaction des machines-outils avec leur environnement par l'entremise de capteurs. Ces deux auteurs font de plus tous deux directement référence au domaine des ordinateurs personnels où l'ouverture de l'architecture permettait déjà, quelques années après leur introduction, de constater d'importants bénéfices pour cette industrie. Un exemple est présenté à la section suivante.

1.1.1 Architecture ouverte versus architecture propriétaire

Généralement, il est reconnu que l'ouverture d'une architecture permet d'étendre la base d'utilisateurs, d'augmenter le nombre de personnes développant sur cette architecture et en bout de ligne, favoriser la croissance de l'industrie [1][7][8][9][10]. Par opposition, il peut être affirmé que la fermeture d'une architecture, qui permet à un constructeur de

protéger sa part de marché, a pour effet de limiter les innovations et de freiner le développement sur cette même architecture puisque la base de développeurs est plus restreinte. Il peut cependant y avoir plusieurs niveaux d'ouverture [11]. Un premier niveau se concentre sur les interfaces homme-machine et les interfaces de contrôle. La plupart des contrôleurs construits autour d'un ordinateur personnel se retrouvent à ce niveau. Ils permettent essentiellement la personnalisation de l'interface homme-machine. Un niveau plus profond dans la machine permet l'accès aux structures de données de contrôle, en plus de l'intégration de logiciels personnalisés. Plusieurs variations d'ouverture peuvent se retrouver entre les deux. Le principal avantage d'un contrôleur à architecture ouverte est son extensibilité, ou la possibilité qu'a l'usager d'ajouter des composantes personnalisées à la machine et d'utiliser ces composantes de différentes façons lors de l'usinage [1][12][13][14][15][16][17]. C'est ce que Schofield qualifie de flexibilité, ou le changement du domaine d'application de la machine-outil pendant sa vie utile [1]. De plus, l'utilisation de capteurs lors de l'usinage, présents ou non lors de l'achat de la machine, est donné comme étant un des traits caractéristiques d'un contrôleur à architecture ouverte.

Certains auteurs, comme Schofield, établissent un parallèle direct entre les bénéfices que l'industrie pourrait retirer d'une ouverture de l'architecture et ce qui s'est produit dans le domaine des ordinateurs personnels [1]. En effet, IBM a initié la création d'un vaste marché d'accessoires et de produits connexes en créant le premier ordinateur personnel extensible. À la même époque, Apple a décidé de fermer son architecture et n'a jamais été en mesure de reprendre l'avance obtenue par l'énorme marché des ordinateurs compatibles IBM. Une anecdote des plus révélatrices existe au sujet des ordinateurs personnels. Malgré le fait qu'en 1981 IBM soit passablement en retard sur ses concurrents pour la sortie de son ordinateur personnel, l'extensibilité de ceux-ci fait en sorte que les premiers PC remportent un succès sans précédent. Cette extensibilité est grandement due au fait que, dans la hâte, IBM achète plusieurs composantes et accessoires de fournisseurs externes. Mais une architecture extensible veut également

dire une architecture aisément copiée. IBM finit par perdre son monopole au profit d'une multitude de compagnies créant des ordinateurs en tout point compatibles aux leurs. Cependant, cette perte de monopole crée un immense marché pour les ordinateurs personnels et ultimement contribue à favoriser l'innovation et le développement sur cette architecture. Voulant reprendre le contrôle du marché en imposant une architecture propriétaire, IBM lance en 1989 les ordinateurs PS/2. Ceux-ci utilisent une nouvelle architecture développée deux ans plus tôt. Les ordinateurs PS/2 sont extensibles, mais toutes les spécifications internes sont propriétés exclusives d'IBM. Toute compagnie voulant créer des accessoires pour ces ordinateurs doit payer des redevances élevées. Cela empêche la création de compatibles IBM PS/2 de même qu'un vaste marché d'accessoires pour ces modèles. L'histoire montre que cette nouvelle architecture n'a jamais connu le succès qu'a connu l'architecture des premiers IBM PC [19]. Dans ce contexte particulier, une architecture ouverte a permis de créer un marché étendu et vigoureux alors qu'une architecture propriétaire et fermée a fini par s'éteindre par manque de développements et d'usagers.

Par ailleurs, un consensus semble exister sur le fait qu'une architecture fermée et propriétaire, qui expose le contrôleur comme une boîte noire, limite les gains en productivité de la machine et impose des limites arbitraires à ce qu'un usager peut réaliser avec sa machine-outil [1][15][17]. L'usager est contraint à se contenter de ce que le contrôleur lui permet de faire parce que les spécifications sont statiques et ne peuvent changer dans le temps. L'usager n'a pas accès à plus même si la machine, avec quelques modifications personnalisées, en serait capable.

1.1.2 Langages de contrôle des contrôleurs à architecture ouverte

En soit, un contrôleur à architecture ouverte ne peut apporter de bénéfice notable à un utilisateur que si cet utilisateur a la possibilité d'en utiliser efficacement les

fonctionnalités lors de l'usinage. En d'autres termes, un lien étroit entre le contrôleur et l'usager doit exister. Ce lien est réalisé par un pont logiciel, ou langage de programmation, qui permet de commander la machine via le contrôleur. C'est ce langage, ou cette architecture logicielle, qui expose les différentes fonctionnalités du contrôleur. L'importance de cette architecture dans le succès d'un contrôleur à architecture ouverte est clairement reconnue par différents auteurs dans la littérature [1][7][8][14][15][20][21][30]. Selon Engels [15], un langage de contrôle pour un contrôleur à architecture ouverte doit :

a. supporter des opérations de calcul et de contrôle en parallèle ;
b. permettre au programme de réagir à des événements ;
c. être compatible avec les standards précédents comme les Codes-G ;
d. supporter les structures classiques d'un langage de programmation.

Il peut être ajouté à ces caractéristiques que le langage doit permettre d'accéder de façon transparente à tout accessoire ajouté à la machine. Cette caractéristique d'extensibilité est centrale à tout contrôleur à architecture ouverte et doit être couverte par le langage de contrôle. Le langage en soit doit également être extensible, pour supporter tout développement futur, et être portable [14].

Tous les efforts de création de contrôleurs à architecture ouverte ont mené à la conception d'un langage orienté vers l'usinage et le contrôle de machines. Certains langages ont été créés de toutes pièces, comme par exemple OpenG [15]. Ce langage intègre une nouvelle syntaxe aux Codes-G et les complète avec plusieurs nouveaux concepts comme les événements et les processus. Les événements sont des signaux, de la machine-outil vers le contrôleur, qui initient l'exécution de routines associées à certaines conditions sur la machine. Par exemple, lorsqu'un capteur de fin de course est atteint, un événement est lancé et la routine appropriée est exécutée. Les processus, quant à eux, permettent l'exécution de tâches en parallèle.

D'autres langages ont été adaptés de langages plus généraux et modifiés pour supporter le contrôle de machines, comme Forth [14]. Ce langage fort simple et compact a vu le jour dans les années 1960 et son développement continue depuis. Ce langage a été conçu au départ pour des applications de contrôle dans le domaine de l'astronomie. L'extensibilité du dictionnaire du langage est unique en son genre et surtout très flexible, ce qui en fait un choix naturel pour la création d'un langage de contrôle pour un contrôleur à architecture ouverte. De plus, des compilateurs pour plusieurs microprocesseurs différents existent pour ce langage. Il est également possible de compiler un programme écrit en Forth en un ensemble d'instructions « machine » ciblant un microprocesseur virtuel. Plusieurs architectures peuvent par la suite exécuter ce code intermédiaire sans besoin de recompilation. Ceci procure une bonne portabilité des programmes entre différentes machines [22]. Il n'y a cependant pas de jeu d'instructions « machine » unique pour Forth. À chaque équipe d'implémentation le loisir de transformer le code source Forth de la manière dont elle le désire [23]. Pour ce langage, cela ne cause cependant pas véritablement de problème puisque tout est centré sur sa syntaxe particulière et non sur un format binaire de distribution de code exécutable.

Des langages à plus grand déploiement comme le C ont également été utilisés dans la mise en oeuvre et l'utilisation d'un contrôleur à architecture ouverte, dont MOSAIC-PM [1]. Cette architecture utilise un langage d'usinage par caractéristiques, c'est à dire modélisant les opérations d'usinage selon le procédé utilisé et non selon des parcours d'outils. Ce langage est du même type que celui élaboré dans le cadre du standard ISO14649 STEP-NC [24]. Le langage C est utilisé pour dialoguer avec la machine et pour le contrôle de bas niveau de celle-ci. Le langage C permet un développement rapide d'applications logicielles sur le contrôleur parce qu'il est un langage répandu et efficace. Un programme typique en C est compilé pour une architecture particulière. La portabilité pour ce genre d'architecture se situe donc au niveau des sources des programmes et non au niveau des modules exécutables.

L'importance de la normalisation autour d'un langage de programmation orienté vers le contrôle de machine est une préoccupation qui remonte aux tout débuts de cette discipline. On parlait déjà en 1973 des avantages d'une standardisation et des problèmes de développement du secteur qu'amènerait son abandon. À cette époque, Williams prévoyait déjà l'importance de l'utilisation intensive des capteurs et du contrôle adaptatif dans le développement du marché des contrôleurs de machines-outils à commande numérique [25]. Il proposait une librairie de procédures écrite en Fortran [26] et une normalisation autour d'un métalangage[2] et de macro compilateurs. Ces macro compilateurs permettent à un usager de créer ses propres langages de contrôle tout en maintenant la portabilité entre les machines.

L'ensemble des acteurs s'entendent pour dire qu'un langage adapté au contrôle est nécessaire pour la réalisation d'un contrôleur à architecture ouverte efficace. Mais, comme le fait remarquer Yellowley, la conception d'un tel langage se heurte aux nombreuses exigences divergentes d'usagers ayant chacun des besoins différents [14]. De plus, se limiter à un seul langage de programmation de haut niveau lie les contrôleurs à ce seul langage. Les innovations qui sont présentes dans tout nouveau langage de programmation, ou la popularité soudaine d'un autre langage, pourrait ainsi causer certains problèmes d'acceptation. Ceci s'est vu un grand nombre de fois dans l'histoire de l'informatique avec les débats parfois orageux portant, par exemple, sur les langages procéduraux versus les langages orientés objets, le langage C versus le langage Pascal, le langage C++ versus le langage Java et bien d'autres [27][28][29]. La réalité est que l'utilisation des langages de programmation de haut niveau change selon les époques et que la popularité de ceux-ci suit souvent des modes.

[2] Langage permettant de définir un autre langage.

1.1.3 Les Codes-G et le contrôle poussé des machines-outils

Depuis l'introduction des machines-outils dans les années 1950, un des seuls liens de programmation qui existe entre l'usager et la machine est un langage nommé les Codes-G. Ces codes alphanumériques simples modélisent les opérations élémentaires de la machine comme l'avance rapide, l'utilisation d'un fluide de refroidissement et la vitesse de rotation de la broche. La Figure 1 montre un exemple typique de programme écrit en Codes-G. Étant donné leur utilisation répandu, les Codes-G ont été à la base de quelques standards, dont ISO6983 et EIA274. Ces standards regroupent les codes les plus communs entre les machines-outils.

```
O1001
     Numéro de programme
N001 T01 M06
N002  T02
     Placer l'outil numéro 01 dans la broche et préparer l'outil numéro 02.
N003 G20
     Placer en mode impérial (pouces).
N001 G54 G90 G00 X0. Y0. Z0.
     Choisir un système de coordonnées [G54], le mode de coordonnées
     absolues [G90] et se déplacer en avance rapide [G00] jusqu'à la
     coordonnée 0,0,0.
N003 M03 S3500
     Démarrer la broche [M03] à une vitesse de 3500 RPM [S3500].
N004 G00 X0. Y0. Z-1.
     Se déplacer en avance rapide à la coordonnée 0,0,-1.
N005 G01 X2. Y0. Z-1. F20.0
     Se déplacer linéairement [G01] à la vitesse de 20.0 pouces/minute
     [F20.0] à la coordonnée 2,0,-1.
```

Figure 1 Exemple de contrôle de machine à l'aide des Codes-G

La littérature et plusieurs acteurs de l'industrie affirment que les Codes-G sont inadéquats et ne sont pas adaptés aux nouveaux développements et aux nouvelles tendances dans le domaine des contrôleurs [1][8][15][18]. Ce sentiment a été récemment

renforcé par le comité responsable de la rédaction de la norme ISO14649/STEP-NC qui stipule que le langage de programmation Codes-G est un frein au développement du marché et empêche l'utilisation des machines-outils à leur plein potentiel [24][30][31]. Le standard STEP-NC défini un modèle de données de très haut niveau permettant de modéliser les caractéristiques d'usinage au lieu des mouvements de la machine. Les critiques énoncées vis-à-vis des Codes-G, ou constaté par l'observation, sont que ceux-ci n'offrent notamment pas :

a. de structure adéquate pour supporter des librairies de procédures [32][33];
b. de réactivité aux événements [8][15];
c. d'extensibilité du langage [14];
d. de portabilité des programmes entre les machines [8][31];
e. de communication bidirectionnelle entre la machine et le programme [8];
f. de support pour l'extensibilité du matériel de la machine et l'ajout de capteurs [1][15][31];
g. de structures d'un véritable langage de programmation [1][15];
h. de performances adéquates [14][33].

Les constructeurs ont depuis longtemps reconnu les faiblesses des Codes-G. Les manufacturiers Fanuc, Okuma et Fadal, entre autres, ont tous créé des extensions à ceux-ci. Ces modifications aux Codes-G introduisent certaines syntaxes de langages de programmation plus évolués comme les boucles, les sauts conditionnels et les procédures. Certaines, comme les *custom macro B* de Fanuc, permettent de modifier un nombre de paramètres beaucoup plus grand que les Codes-G standards. Ces macros introduisent également la possibilité de communiquer de façon limitée avec des accessoires ajoutés à la machine via des ports d'entrées et de sorties [32].

Tout en diminuant l'importance des problèmes rencontrés dans l'utilisation des Codes-G, ces extensions ne règlent pas le problème de fond. En effet, celles-ci reposent

toujours sur la syntaxe rigide des Codes-G. Cette syntaxe origine du tout début de la commande numérique où le seul support de stockage disponible était une bande perforée [8][31]. Cette syntaxe empêche l'utilisation de méthodes de programmation structurée [33] souvent associées à la qualité d'un logiciel. Ce manque de structure se manifeste, entre autres, par la nécessité d'utiliser à profusion les sauts inconditionnels dans le code source (voir Figure 2). On peut voir qu'il n'existe pas de lien syntaxique entre le test de condition (IF) et la portion de code exécutée lorsque la condition est vraie ou lorsqu'elle est fausse. Pour cette raison, des sauts à des numéros de ligne (GOTO 10 et GOTO 11) doivent être utilisés. Cette pratique est pourtant considérée depuis longtemps comme une très mauvaise technique de programmation [34]. La façon de passer des paramètres entre les procédures et l'appel de ces procédures est également inflexible. De plus, le nombre de variables n'est pas limité par la mémoire disponible, mais par des décisions prises lors de l'élaboration du langage. Certains constructeurs vont même jusqu'à facturer des montants supplémentaires pour augmenter le nombre de variables disponibles [33]. Le nom que peut prendre une variable est également parfois soumis à des règles très strictes. Certaines extensions ne permettent d'ailleurs de référencer une variable que par un index numérique [32]. Les fonctions mathématiques disponibles sont souvent en nombre limité, cet état de chose variant toutefois d'une extension à l'autre. La vitesse d'exécution de ces instructions mathématiques est cependant un problème. Il est statué dans la littérature qu'une boucle de calcul ne comportant que quelques opérations mathématiques peut provoquer une pause perceptible par l'opérateur entre chaque itération [33].

```
...
...
...
IF [#12 EQ 2.] GOTO 10
M04
GOTO 11
N10 M03
N11
...
...
...
```

Figure 2 Construction IF/THEN/ELSE en Custom Macro B de Fanuc

1.2 Éléments de développement d'une architecture logicielle

La synthèse des points présentés aux sections précédentes fait ressortir certaines caractéristiques souhaitables que l'architecture logicielle devrait supporter. Ces caractéristiques principales sont résumées par,

a. la portabilité ;
b. l'extensibilité ;
c. la réactivité aux événements ;
d. le support de plusieurs langages différents.

Un langage exposant les fonctionnalités d'un contrôleur à architecture ouverte devrait supporter les caractéristiques décrites ci haut. Ce langage peut être de deux niveaux, haut ou bas. Par langage de bas niveau, il est entendu un langage proche de la machine et comportant des instructions élémentaires ressemblant au jeu d'instruction d'un microprocesseur. Un lien direct entre un langage de bas niveau et le langage machine d'un microprocesseur existe. Un langage de bas niveau permet donc de programmer

directement le microprocesseur d'un ordinateur. Le microprocesseur peut être réel ou virtuel, auquel cas une machine virtuelle devra émuler le comportement de cette composante. Un langage de haut niveau est un langage qui se rapproche plus du langage naturel. C'est un langage lisible et compréhensible par un être humain. Les programmes écrits avec des langages de haut niveau sont transformés en une forme de plus bas niveau avant que le code ne puisse être exécuté. Par exemple, un programme écrit en langage C doit être compilé pour qu'une architecture donnée soit ciblée.

1.2.1 La portabilité logicielle

Il a été démontré que la portabilité, dans un contexte de contrôleur à architecture ouverte, est souhaitable et même nécessaire [14]. Cette portabilité permet non seulement d'utiliser un programme de coupe d'une machine à l'autre, mais également de faire profiter du développement d'applications logicielles réalisé sur une machine à l'ensemble des machines ayant des caractéristiques semblables. La portabilité logicielle a toujours été souhaitée en programmation et est d'ailleurs l'objectif principal du langage Java [35].

La portabilité logicielle peut se situer au niveau des sources d'un programme ou au niveau du code exécutable compilé. Si la portabilité se situe au niveau des sources du programme, ce sont ces sources qui seront distribuées entre les machines, et le code devra être recompilé pour chacune des machines cibles. Ce type de portabilité est dit de haut niveau. La portabilité peut aussi se situer au niveau du code exécutable. Les modules pré compilés sont ainsi distribués entre les machines. Ces modules peuvent provenir de un ou plusieurs langages de haut niveau. Il est alors dit que la portabilité est de bas niveau. La littérature montre que la portabilité réalisée à l'aide d'un langage de bas niveau est plus efficace et plus générale que la portabilité réalisée au niveau des sources du programme [36][37]. La raison principale est que les sources n'ont pas

besoin d'être recompilées pour chaque nouvelle architecture ciblée. De plus, la portabilité au niveau des modules exécutables permet l'utilisation de plusieurs langages de haut niveau différents.

Un exemple d'architecture permettant la portabilité au niveau des exécutables est la machine virtuelle Java (JVM). Le code source compilé par un compilateur ciblant cette architecture de bas niveau est portable sur plusieurs machines différentes. Cependant, la machine virtuelle Java est si liée à un seul langage de haut niveau que l'utilisation d'autres langages est difficile. La spécification de la machine virtuelle Java indique clairement que le jeu d'instructions de bas niveau de cette architecture a été conçu explicitement pour supporter le langage Java [35]. Quelques tentatives ont été faites pour permettre de cibler la JVM à l'aide d'autres langages, mais ces essais ont eu des succès mitigés [38]. La portabilité du code exécutable sur une machine virtuelle Java est donc intimement liée à l'utilisation d'un seul langage de programmation.

1.2.2 L'extensibilité logicielle

L'extensibilité d'un langage de programmation peut se situer au niveau de l'ajout de mots-clés au langage. Le langage Forth permet, par exemple, de créer facilement de nouveaux mots-clés qui sont ensuite disponibles à l'utilisateur. Le jeu d'instructions du langage Forth est donc extensible [22]. Tous les langages de programmation permettent d'ajouter des procédures et des objets au dictionnaire de mots reconnus par le compilateur. Par exemple, en C++ il est possible de définir une fonction qui peut par la suite être appelée à partir d'une autre partie du programme. Ces fonctions n'étendent pas les instructions de base du langage, mais ajoutent des mots qui sont ensuite reconnus par le compilateur. Le corps de chacune de ces fonctions utilise les instructions de base et la syntaxe du langage. Une fonction peut également, à son tour, utiliser des fonctions ou des objets préalablement définis [39]. Dans ces langages, l'extension se fait par la

création de librairies de fonctions et d'objets qui peuvent être liées à tout nouveau programme. C'est cette approche de couches logicielles et de librairies de fonctions qui est préconisée, entre autre, par Yellowley [14] et Williams [25] pour l'extension du langage de contrôle. C'est aussi de cette façon que peut être étendu le langage Java et son architecture de bas niveau [35].

1.2.3 La réactivité aux événements

L'analyse de la littérature montre qu'un langage de contrôle compatible à une architecture ouverte doit permettre aux programmes, selon les besoins de l'usager, de répondre à des événements pouvant survenir sur la machine. Les événements sont à la base de l'utilisation des capteurs dans le contrôle des machines-outils à commande numérique. C'est ce mécanisme qui permet à des routines écrites par l'usager d'être exécutées lorsque certaines conditions sont rencontrées. Un exemple d'événement pourrait être l'arrêt d'urgence activé par l'opérateur. La réponse à cet événement serait d'arrêter le mouvement de la machine de façon sécuritaire [15]. L'exécution de la routine associée à un événement ne suit pas l'ordre d'exécution normale du programme et peut survenir à tout moment. C'est pour cette raison que dans certaines architectures, les événements sont aussi appelés interruptions [41].

Pour un langage de haut niveau, plusieurs stratégies peuvent être utilisées pour supporter les événements. Une manière de faire est d'inclure la sémantique des événements directement dans la syntaxe du langage. C'est l'avenue prise par le langage OpenG [15]. Ce langage possède des mots-clés spécialement utilisés pour définir et utiliser les événements. Il est par ailleurs possible d'étendre un langage pour le support des événements par une librairie de fonctions ou de classes. Le langage Java utilise cette technique pour implémenter les événements reliés à l'interface usager [40]. C'est

également cette technique qui est utilisée dans la plupart des autres langages de programmation.

Pour un système de bas niveau, un mécanisme d'interruption est généralement utilisé. Dans une architecture comportant un microprocesseur réel, un signal est envoyé à celui-ci qui s'occupe alors d'exécuter la routine associée à l'interruption déclenchée. C'est ce mécanisme qu'utilisent les microprocesseurs Intel [41]. Pour une architecture comportant un microprocesseur virtuel émulé de façon logicielle, ce mécanisme peut être simulé. C'est cette technique qui est utilisée dans le développement de la version 6 de l'engin d'exécution de Perl [42].

1.2.4 Utilisation de plusieurs langages de programmation

L'analyse de l'histoire de l'informatique et des développements dans le domaine des contrôleurs à architecture ouverte montre qu'une multitude de langages de programmation existe. Aucun de ces langages ne fait l'unanimité ou ne répond aux besoins de tous [27][28][29]. Une architecture logicielle pour un contrôleur à architecture ouverte devrait donc permettre l'utilisation de plusieurs langages différents. Il n'existe cependant pas de correspondance absolue entre toutes les fonctionnalités des langages de programmation. Le support de plusieurs langages différents passe donc par l'utilisation d'un code élémentaire se rapprochant du langage machine d'un microprocesseur. C'est cette avenue qui a été empruntée par la machine virtuelle Java. Cependant, des décisions de conception font en sorte que seul le langage Java peut être facilement supporté [35][38]. C'est également cette technique qui est utilisée par la nouvelle architecture Microsoft .NET qui a été conçue dès le départ pour supporter une multitude de langages [43]. Les objectifs de départ du développement de l'engin d'exécution de Perl 6 font également référence à la nécessité de ne pas lier l'engin d'exécution à un langage en particulier [42]. Cet engin d'exécution se rapproche

beaucoup d'un véritable microprocesseur. Ses concepteurs l'appellent d'ailleurs un microprocesseur logiciel. L'engin d'exécution de l'architecture Microsoft .NET implémente, pour sa part, certains idiomes de langage de haut niveau comme la notion d'objets et d'héritage simple. L'inclusion de ces idiomes peut occasionner certains problèmes lors de l'implémentation de langages de programmation qui diffèrent trop du modèle de base choisi pour l'architecture [44][45].

1.3 Retour sur la revue de littérature

La revue de littérature montre les avantages d'une approche d'ouverture dans le contexte des contrôleurs de machines-outils à commande numérique. Ces avantages portent sur la flexibilité qu'offre ce type de contrôleur, ainsi que sur la liberté d'action que les utilisateurs ont quant à l'utilisation et la modification de leurs machines. Il est escompté qu'une utilisation à grande échelle de ce type de contrôleur pourra favoriser l'innovation et les développements liés à l'utilisation des machines-outils à commande numérique.

Un des grands avantages des contrôleurs à architecture ouverte est l'extensibilité de la machine-outil qui est offerte à l'utilisateur. Celui-ci peut étendre les fonctionnalités de sa machine selon ses besoins. Le contrôleur a pour tâche de tenir compte de ces modifications de façon transparente. Les fonctionnalités de la machine-outil ne sont donc plus fixées dans le temps et deviennent ainsi dynamiques. Le potentiel de la machine-outil peut donc être mieux exploité et ajusté aux besoins de son utilisateur. De plus, les contrôleurs à architecture ouverte permettent la portabilité des développements logiciels d'un contrôleur à l'autre. Cette portabilité permet de capitaliser sur les développements logiciels effectués sur un contrôleur et de les transposer aux autres.

L'exposition à l'usager des fonctionnalités d'un contrôleur à architecture ouverte se fait à l'aide d'un langage de programmation. Ce langage expose les caractéristiques du

contrôleur et permet à l'usager d'en tirer partie. L'importance de cette architecture logicielle dans le succès d'un contrôleur à architecture ouverte a été démontrée. Cependant, un des obstacles à la mise en oeuvre d'un contrôleur à architecture ouverte est l'utilisation d'un langage de programmation, sur les contrôleurs, qui n'est pas adapté à ce genre de tâche. La revue de littérature a démontré que les Codes-G ont de grandes limitations, tant au point de vue des fonctionnalités que des performances et de la syntaxe. L'élaboration d'un langage adapté pour l'usinage, et tenant compte des impératifs des contrôleurs à architecture ouverte, est donc de mise.

Il a toutefois été possible de constater dans différents écrits qu'il pourrait être défavorable de standardiser tout sur un seul langage de programmation. Il a été démontré que les langages de programmation n'ont jamais fait l'unanimité et que chaque langage ne peut couvrir tous les besoins. Le support de plusieurs langages de programmation différents semble donc de mise. Ce support multi langage passe par l'utilisation non pas d'un langage de programmation de base, mais d'un jeu d'instructions élémentaires se rapprochant du jeu d'instructions d'un microprocesseur réel. Tous les langages utilisés pour l'architecture ciblent ce jeu d'instructions élémentaires. Des compilateurs pour chacun des langages supportés s'assurent de la validité du code élémentaire produit. C'est cette avenue qu'ont empruntée les concepteurs de l'architecture .NET de Microsoft dans leur désir de permettre l'utilisation de plusieurs langages de programmation différents.

CHAPITRE 2

SURVOL DE L'ARCHITECTURE BNCL

Comme il a été discuté dans la revue de la littérature au chapitre précédent, l'avènement de concepts tel que les contrôleurs à architecture ouverte ont fait ressortir les limites de l'architecture logicielle présentement utilisée pour le contrôle des machines-outils à commande numérique. Cette architecture logicielle, datant des débuts de l'usinage par commande numérique, n'offre pas les caractéristiques nécessaires pour permettre une réalisation efficace de ces nouveaux concepts. Il peut également être affirmé que les architectures fermées peuvent être un frein à l'innovation et peuvent empêcher les utilisateurs de profiter du plein potentiel de leur investissement. En effet, les machines, pendant leur vie utile, ne peuvent s'adapter aux besoins changeants des utilisateurs puisque les contrôleurs utilisés sont rigides.

Les changements qui peuvent survenir, pour rendre les machines-outils à commande numérique plus flexibles, peuvent se situer à plusieurs niveaux. Comme il a été présenté dans la revue de littérature, une architecture logicielle adéquate est un élément important favorisant l'ouverture des contrôleurs de machines-outils à commande numérique. L'étude se concentre donc sur cette architecture logicielle.

En synthétisant ce qui a été présenté dans la revue de littérature, il est permis d'affirmer qu'une architecture logicielle associée à un contrôleur à architecture ouverte doit :

a. permettre la portabilité des programmes de coupe et des logiciels développés pour un contrôleur;
b. permettre l'extensibilité de la machine-outil en donnant la possibilité à l'utilisateur d'ajouter des composantes et d'utiliser ces dernières dans ses programmes (les composantes peuvent être aussi variées que des capteurs de fin de course ou des tables d'indexation);

c. permettre aux programmes créés par l'utilisateur de réagir aux événements pouvant survenir sur la machine;
d. ne pas confiner les divers utilisateurs à un seul langage de programmation.

Une architecture logicielle pour le contrôle de machines-outils à commande numérique répondant le mieux possible à l'ensemble de ces critères a donc été développée. Cette architecture, nommée l'architecture BNCL pour « Basic Numerical Control Language », comporte différentes composantes permettant d'atteindre ces objectifs. Cette section du travail explicite ces composantes et l'architecture BNCL dans son ensemble.

Le survol débutera par un coup d'œil sur l'organisation conceptuelle de l'architecture BNCL. La description des différentes composantes qui forment l'architecture ainsi que leur utilité dans l'ensemble sera vue. Par la suite, une discussion sur la manière dont l'architecture BNCL répond aux quatre critères principaux qui ont été retenus seront discutés : la portabilité logicielle, l'extensibilité de la machine-outil, la réactivité aux événements et la possibilité d'utiliser plusieurs langages de programmation. Le tout se terminera sur une revue générale de l'architecture BNCL.

2.1 Organisation conceptuelle

L'architecture BNCL s'appuie sur deux abstractions pour ouvrir le contrôleur de la machine-outil, soit l'abstraction du matériel physique de la machine, et l'abstraction de l'environnement d'exécution des programmes et des logiciels. Une abstraction a pour but de cacher les détails d'un ensemble plus complexe. Dans l'architecture BNCL, la composante faisant abstraction du matériel physique de la machine-outil, comme les axes et les différents accessoires, est le BVH ou « BNCL Virtual Hardware ». Cette composante cache les détails de la machine-outil et expose les caractéristiques de chacune d'elles de façon uniforme et constante. La Figure 3 montre les relations existant

entre le BVH et les différentes composantes du contrôleur et de la machine-outil. À la lumière de cette figure, il peut être constaté que toute communication avec la machine-outil passe par le BVH et que les détails de fonctionnement de cette machine-outil sont complètement cachés par cette composante.

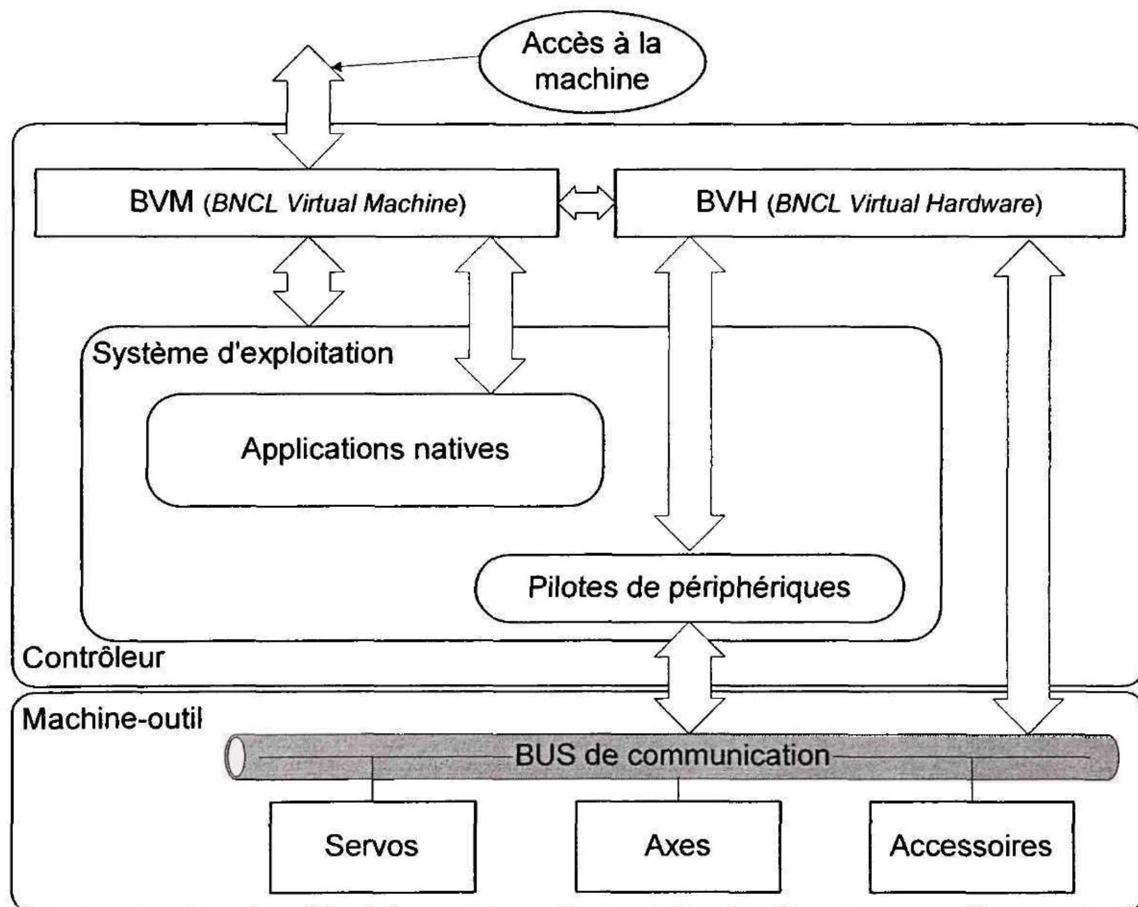

Figure 3 Abstraction de la machine-outil par la BVM et le BVH

L'environnement d'exécution, pour sa part, est l'endroit où les programmes de coupe et les différentes autres applications sont exécutés. Dans l'architecture BNCL, ce rôle est attribué à la BVM ou « BNCL Virtual Machine ». Il est à noter que le mot « machine » n'est pas utilisé au sens de la machine-outil mais au sens d'une machine de calcul tel un microprocesseur. Selon cette définition, une machine virtuelle BNCL est donc un

microprocesseur virtuel (machine virtuelle) reposant sur un jeu d'instructions nommé ici langage BNCL. Le jeu d'instructions BNCL est le noeud central de tout l'environnement d'exécution exposé par la BVM. Ce sont ces instructions que la machine virtuelle comprend et exécute. C'est également sous forme de modules formés d'une multitude de ces instructions que les programmes de coupe et les applications d'usinage se présentent à la machine virtuelle pour l'exécution. La Figure 3 montre comment la BVM fait l'abstraction de l'environnement d'exécution, soit le système d'exploitation et ses différentes parties constituantes sur lequel repose le contrôleur. Il est à noter que la machine virtuelle BNCL (BVM) et le matériel virtuel (BVH) résident tous deux sur le contrôleur. Ces composantes contrôlent l'accès aux fonctionnalités du contrôleur de la machine-outil et jouent un rôle central dans la programmation et l'utilisation de celle-ci.

2.1.1 Jeu d'instructions BNCL

Le langage BNCL comporte actuellement une centaine d'instructions (voir ANNEXE 1). Ces instructions permettent de manipuler des valeurs en mémoire, d'effectuer des calculs numériques sur des nombres entiers et décimaux et de communiquer avec le BVH. Seize instructions BNCL sont réservées pour la communication avec le matériel virtuel. Ces instructions permettent de lire ou d'écrire des valeurs sur un port. Toutefois, aucune instruction du langage BNCL n'est associée spécifiquement au contrôle de machines-outils. Au contraire, le jeu d'instruction BNCL est aussi général que peut l'être celui d'un microprocesseur réel. Le seul moyen de communiquer avec la machine-outil, et donc de la contrôler, passe par les seize instructions réservées pour la communication avec le BVH. Cette généralité du jeu d'instructions permet de ne pas limiter les applications s'exécutant sur la BVM uniquement à des programmes de coupe. Cela permet également de compiler pour la BVM des programmes écrits dans plusieurs langages différents. L'aspect multi langage de l'architecture sera abordé plus loin dans le survol de l'architecture BNCL. La Figure 4 montre un exemple de code source BNCL.

La routine présentée permet de calculer la distance entre deux points. La Figure 5 montre une routine semblable écrite en assembleur pour les processeurs Intel x86. La ressemblance entre les instructions BNCL et celles d'un microprocesseur réel peut être constatée. De plus, comme dans le cas d'un microprocesseur réel, l'usager n'a pas l'obligation de connaître ces instructions et d'écrire ses programmes avec celles-ci. Il a la possibilité de le faire, mais il n'en a pas l'obligation. Généralement, les instructions BNCL seront générées à partir de code source d'un langage de haut niveau et d'un compilateur, tel le langage C. La Figure 6 montre un exemple de code en langage C qui pourrait donner les flux d'instructions présentés aux Figure 4 et Figure 5.

```
sav.i;                   // Changer la fenêtre de registres des entiers.
sav.f;                   // Changer la fenêtre de registres des nombres réels.
ld.vf pfr0, [gr0];       // Charger de la mémoire le point A du vecteur.
ld.vf pfr1, [gr1];       // Changer de la mémoire le point B du vecteur.
sub.vf pfr2, pfr1, pfr0; // Calculer la différence entre le point A et B.
mul.vf pfr2, pfr2, pfr2; // Élever chaque membre de pfr2 au carré.
cadd.vf fr7, pfr2;       // Additionner tous les membres de pfr2.
sqrt.f fr7, fr7;         // Calculer la racine carré (la distance est dans fr7).
rst.f;                   // Réhabiliter la fenêtre précédente des nombres réels.
rst.i;                   // réhabiliter la fenêtre précédente des nombres entiers.
ret;                     // Retourner d'une sous-routine.
```

Figure 4 Exemple de code source BNCL

Contrairement à quelques architectures logicielles comme Microsoft .NET ou la machine virtuelle Java, le jeu d'instructions BNCL ne comporte aucun support direct de langages de programmation de haut niveau. Par exemple, il n'existe aucune instruction reliée à la programmation orientée objet. La notion de classe est donc tout à fait absente du langage. Le support pour les types évolués comme les structures de données est également absent. De plus, des pointeurs sur la mémoire peuvent être utilisés. La décision d'éviter les structures de langages de plus haut niveau dans le jeu d'instructions a été prise pour que les similitudes avec un microprocesseur réel soient les plus grandes possibles. En maintenant ces similitudes avec un microprocesseur au maximum, le

support d'un nombre étendu de langage de programmation de haut niveau devient possible.

```
push   ebp                     ; Sauvegarder le register ebp.
mov    ebp, esp                ; Création de la structure de pile.
sub    esp, 0                  ; Allouer de l'espace pour les variables locale.
movups xmm0, [ebp]             ; Charger le point A du vecteur.
movups xmm1, [ebp+16]          ; Charger le point B du vecteur.
subps xmm1, xmm0               ; Calculer la différence entre le point A et B.
mulps xmm1, xmm1               ; Élever chaque membre de xmm1 au carré.
movaps xmm0, xmm1              ; Créer une copie de xmm1 dans xmm0.
addss xmm1, xmm0               ; Additionner le premier membre.
shufps xmm0, xmm0, 93h         ; Placer le deuxième membre.
addss xmm1, xmm0               ; Additionner le deuxième membre au total précédent.
shufps xmm0, xmm0, D2h         ; Placer le troisième membre.
addss xmm1, xmm0               ; Additionner le troisième membre au total précédent.
shufps xmm0, xmm0, E1h         ; Placer le dernier membre.
addss xmm1, xmm0               ; Additionner le dernier membre au total précédent.
sqrtss xmm1, xmm1              ; Obtenir la racine carré du total.
pop ebp                        ; Rétablir la structure de pile précédent.
ret                            ; Retourner d'une sous-routine.
```

Figure 5 Exemple de routine en assembleur x86

```
float ComputeDistance(float[4] pt1, float[4] pt2) {
    float         tmp[4];
    float         total = 0;

    for(int i = 0;i < 4;i++) {
        tmp[i] = pt1[i] - pt2[i];
        tmp[i] = tmp[i] * tmp[i];

        total += tmp[i];
    }

    total = sqrt(total);

    return total;
}
```

Figure 6 Routine écrite en C pour calculer la distance entre deux points

2.2 Portabilité logicielle

L'abstraction faite par la BVM et le BVH permet dans les faits la portabilité des programmes entre les différents contrôleurs. En effet, comme tous les programmes sont formés des mêmes instructions BNCL et que les accès à la machine-outil passent tous par le BVH, les programmes communiquent donc toujours de la même manière avec le système, quel que soit la machine-outil et le contrôleur. Cependant, cette portabilité ne concerne pas uniquement les programmes de coupe. Les développements d'applications complètes d'usinage, les modules implémentant de nouvelles trajectoires ou des programmes exposant de nouveaux cycles sont tous portables d'un contrôleur à l'autre. Le développement fait sur une machine-outil est donc utilisable sur une autre machine, en autant que celles-ci possèdent des caractéristiques minimalement semblables, c'est à dire qu'elles permettent toutes deux des mouvements équivalents. Par exemple, une fraiseuse cinq axes est en mesure de reproduire les mouvements d'une fraiseuse trois axes.

Pour obtenir cette portabilité à l'aide des composantes BVM et BVH, certaines conditions doivent toutefois être rencontrées. Tout d'abord, la signification des ports du BVH doit être constante d'un contrôleur à l'autre. Par exemple, un port permettant de communiquer avec le changeur d'outils doit être à la même adresse quel que soit le contrôleur. Ensuite, le protocole de communication pour chacun de ces ports doit être identique d'un contrôleur à l'autre. Par exemple, si l'envoi de la valeur 01 au port du changeur d'outils sur un contrôleur prépare l'outil 01, alors l'envoi de la même valeur sur un autre contrôleur doit provoquer exactement la même action. En d'autres termes, la spécification de fonctionnement de chaque port doit être la même d'un contrôleur à l'autre, et donc d'un BVH à l'autre. Finalement, lorsqu'un accessoire secondaire n'est pas présent sur une machine, ou que les options disponibles ne couvrent pas la même étendue que celle proposée de façon standard, le BVH doit obligatoirement signaler l'absence de cette fonctionnalité par une erreur. L'opérateur, ou le module en court

d'exécution, pourra réagir en conséquence et jugera si la différence est assez significative pour arrêter l'application.

2.3 Extensibilité de la machine-outil

Un des principes centraux d'un contrôleur à architecture ouverte est la possibilité qu'il donne à l'usager de modifier la machine-outil et de tenir compte de ces modifications dans ses programmes. Dans l'architecture BNCL, l'abstraction du matériel de la machine-outil réalisée par le BVH permet cette extensibilité. Pour le programmeur, le BVH se présente comme une zone mémoire comportant des ports à partir desquels il peut envoyer ou recevoir des valeurs. La communication avec ces ports est en fait une communication avec les composantes de la machine-outil, par l'intermédiaire du BVH. Une analogie avec l'architecture interne d'un ordinateur personnel (voir Figure 7) peut permettre de comprendre ce qu'est un port machine. Prenons par exemple la carte graphique d'un ordinateur. Cette carte graphique est connectée au reste de l'ordinateur par des ports. Par exemple, lorsque le mode d'affichage graphique doit être changé, la notification est envoyée à un certain port. La carte graphique reçoit cette valeur et change le mode d'affichage selon les spécifications.

Dans le même esprit que la carte graphique sur un ordinateur personnel, chaque port du BVH est relié à une composante sur la machine-outil. Lorsqu'un programme qui s'exécute sur la BVM envoie une valeur à un de ces ports, le BVH interprète cette valeur et manipule la composante désirée selon les spécifications. Cette composante peut être un axe ou tout autre accessoire sur la machine. L'extensibilité de l'architecture BNCL vient du fait que ces ports peuvent être reliés à des accessoires présents sur la machine lors de son achat ou ajoutés sur la machine par l'utilisateur. Les détails de connexion de l'accessoire avec la machine ne sont pas importants puisque celle-ci se fait toujours de la même façon et passe toujours par le BVH. Seul le matériel virtuel BNCL a besoin de

connaître les détails de cette connexion, soit la façon dont l'accessoire est branché physiquement et électriquement à la machine-outil.

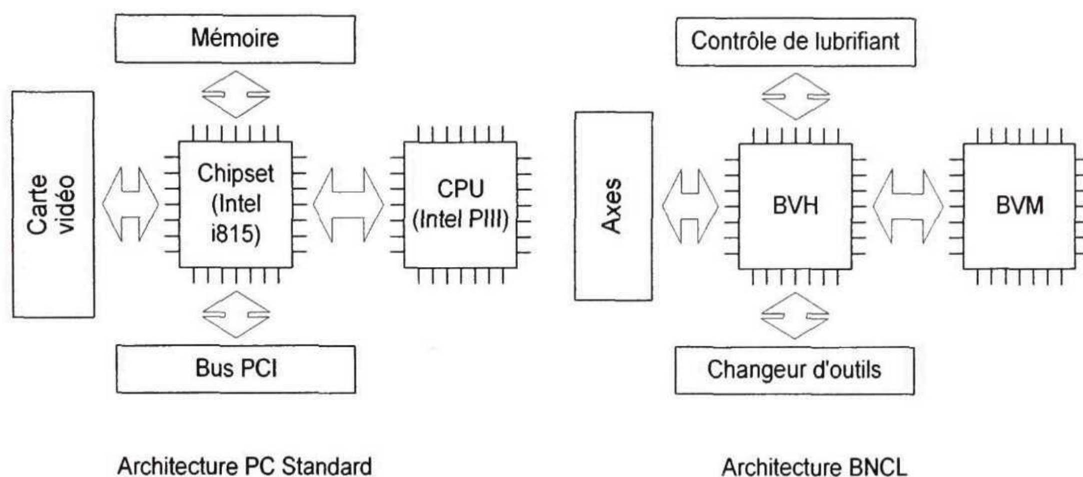

Figure 7 Analogie entre l'architecture PC et l'architecture BNCL

Dans un souci de bien séparer l'accès aux composantes standard de l'accès aux composantes qui peuvent être ajoutées par l'utilisateur, deux groupes de ports sont disponibles dans l'architecture BNCL. Le premier groupe est indépendant de la machine-outil sur laquelle il se trouve. Il regroupe les ports de communication communs à toute machine-outil d'une même catégorie. Par exemple, par convention, tous les BVH des machines-outils à commande numérique comportant trois axes exposent les mêmes ports dans ce premier groupe. Le second groupe est, quant à lui, utilisé pour communiquer avec les accessoires ajoutés par l'utilisateur de la machine. La convention d'attribution de ces ports est interne à une entreprise et peut changer d'une machine-outil à l'autre selon les besoins. La Figure 8 montre un exemple d'utilisation de l'extensibilité d'un contrôleur par l'ajout d'un capteur de température à la machine-outil. Ce capteur est utilisé pour ajuster la vitesse de révolution de la broche en fonction de la température captée sur l'outil de coupe. Avec l'architecture BNCL, l'accès aux informations provenant du capteur de température est contrôlé par le BVH et le groupe de ports

extensibles. Un programme usager interroge ce groupe de port pour en obtenir la température ou signifier au BVH qu'il désire recevoir un signal si la température atteint un certain seuil.

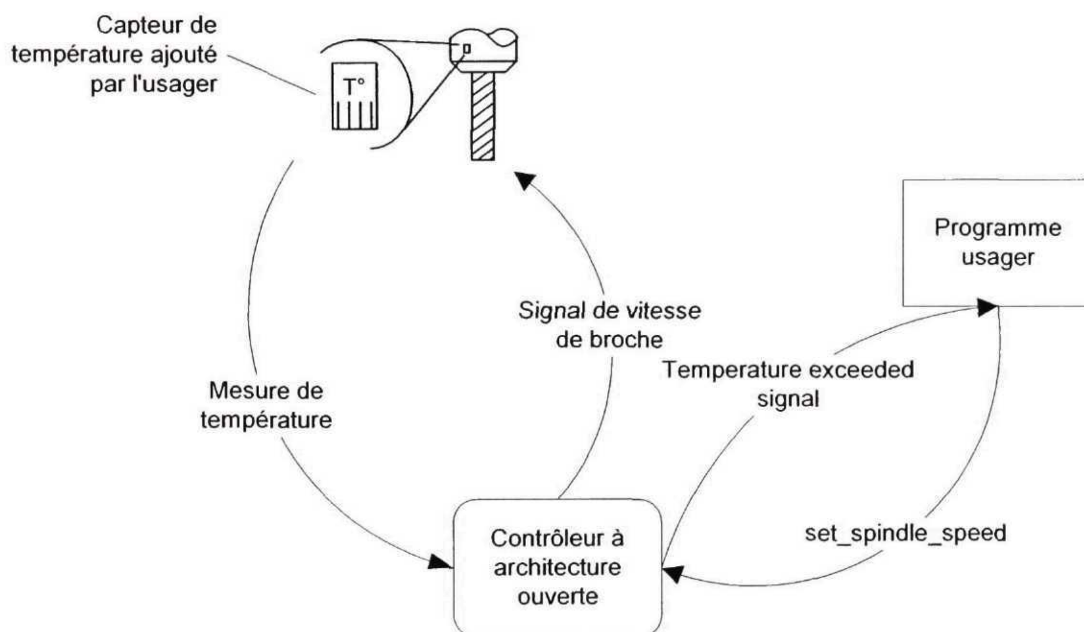

Figure **8** Exemple d'extension par l'usager d'une machine-outil

2.3.1 Extensibilité et portabilité

Le deuxième groupe de ports, qui permet la communication avec des accessoires ajoutés à une machine-outil, pose un problème au principe de portabilité. Si des accessoires ne se retrouvent que sur une machine, les applications développées pour cette machine ne sont plus portables d'un contrôleur à l'autre (voir Figure 9). Sur cette figure, il est indiqué que le programme qui fonctionne correctement sur la machine-outil #2 ne pourra fonctionner sur la machine-outil #1 puisque celle-ci ne possède pas de capteur de

température. Cette limite à la portabilité est nécessaire pour que l'extensibilité des machines-outils soit possible. En effet, le principe même de l'extensibilité veut que l'utilisateur puisse ajouter des accessoires qui n'étaient pas prévus lors de la conception de sa machine-outil en particulier. Un groupe de ports commun à toutes les machines ne pourrait tenir compte de toutes les modifications et de tous les besoins possibles des utilisateurs. Cependant, en instaurant des standards à l'intérieur d'une même entreprise, il est possible de s'assurer que les ports permettant l'extensibilité des machines soient constants pour l'ensemble des machines-outils de l'entreprise. Par exemple, un port permettant de communiquer avec un capteur de température serait toujours placé à la même adresse quelle que soit la machine-outil. De cette façon, les applications d'usinage profitant de ce capteur seraient portables pour l'ensemble des machines-outils de l'entreprise.

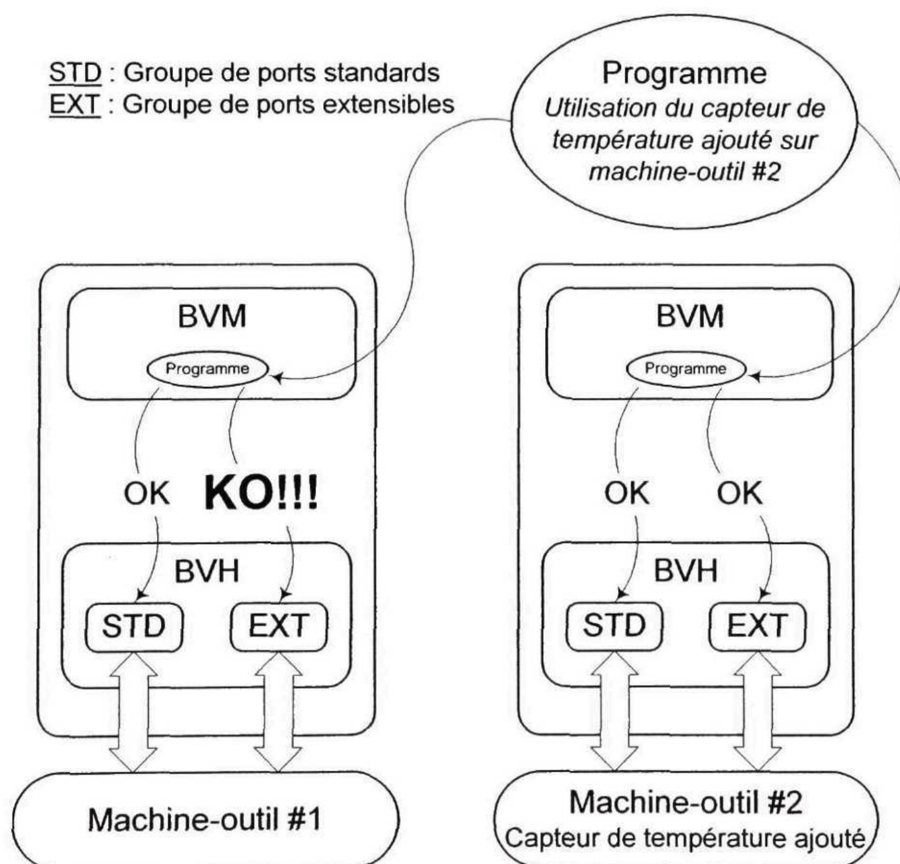

Figure 9 Implications de la portabilité et de l'extensibilité

2.4 Réactivité aux événements

Un événement est une condition d'exception qui survient sur une machine-outil pendant le déroulement normal du programme. Cette interruption de la séquence normale du programme en cours d'exécution s'accompagne de l'exécution d'une routine associée à l'événement. Cette routine effectue un travail en lien avec l'interruption et redonne ensuite le contrôle au programme en cours. Sur les contrôleurs actuels, sauf pour quelques exceptions, toutes les actions associées à ces interruptions sont contrôlés par le manufacturier du contrôleur. Dans de rares cas, l'usager peut demander l'exécution de son propre code lorsqu'une certaine interruption survient, mais ce qui peut être fait n'est pas très élaboré.

Par ailleurs, la gestion des événements est en lien direct avec l'extensibilité de la machine-outil et du contrôleur. En effet, plusieurs accessoires pouvant être ajoutés à la machine-outil ne sont pas passifs et doivent pouvoir avertir le contrôleur lorsqu'une certaine condition est rencontrée. Comme ces accessoires sont ajoutés par l'usager, c'est le code de l'usager qui doit être activé par les événements associés et non le code de base du contrôleur. Il peut donc être compris que la réactivité aux événements est liée à l'extensibilité du contrôleur et de la machine-outil.

L'architecture BNCL utilise les mêmes mécanismes d'interruption que ceux utilisés pour un microprocesseur réel. La machine virtuelle BNCL supporte un certain nombre de signaux d'interruption (voir Figure 10). Lorsqu'un accessoire désire lancer un événement à la BVM, le BVH envoie un signal et ainsi la BVM est alertée. Le code associé à cet événement particulier est exécuté par la BVM et le contrôle est ensuite redonné au programme en cours lorsque le traitement est terminé.

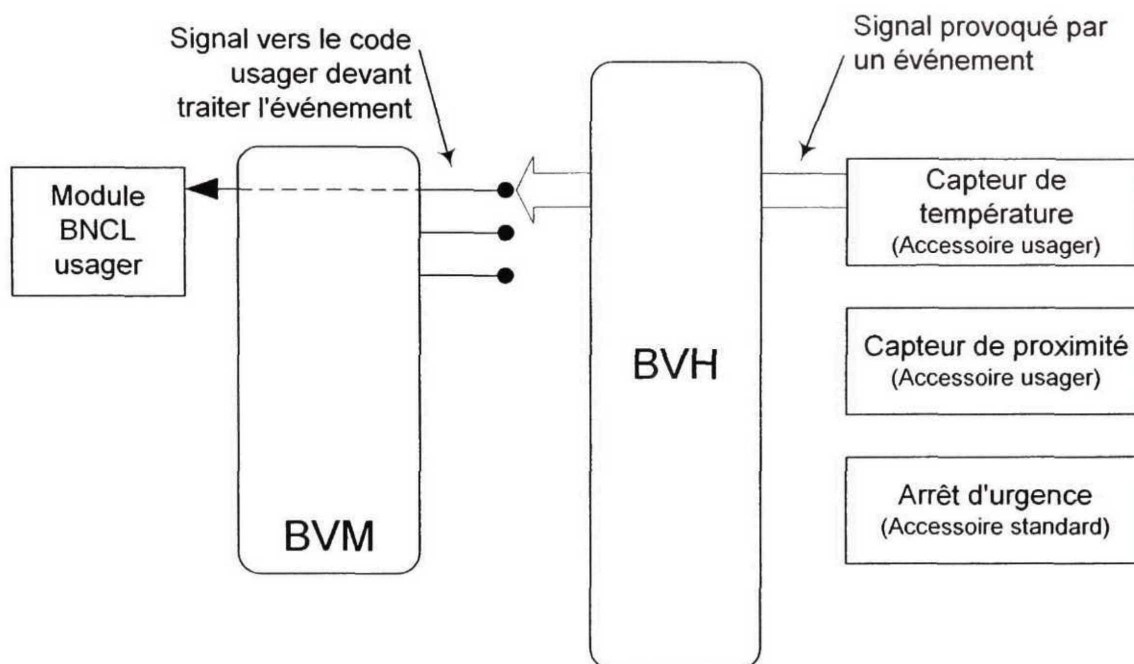

Figure 10 Réactivité aux événements de l'architecture BNCL

2.5 Utilisation de plusieurs langages de programmation

Comme le développement de l'informatique le montre, jamais dans l'histoire de la programmation un seul langage n'a fait l'unanimité. La raison en est qu'un seul langage ne peut répondre à tous les besoins et qu'une syntaxe peut être intéressante pour une personne et ne pas l'être pour une autre. De plus, l'engagement envers un langage relève parfois du domaine de l'émotion et de la connaissance, où chacun y va de son plaidoyer envers un langage en particulier sans autre argument que son sentiment favorable pour le langage. L'informatique a vu différentes guerres à ce sujet, comme l'affrontement entre le C et le Pascal et plus récemment entre le Java et le C++. En d'autres termes, l'acceptation d'un langage relève parfois plus du goût de ses utilisateurs potentiels que de ses réels mérites rationnels et désintéressés.

Il doit également être mentionné que les langages évoluent dans le temps et qu'un langage adéquat pour une certaine période de l'histoire ne le sera peut-être plus dans une période ultérieure. De nouveaux développements sur les langages sont continuellement réalisés et ces nouveaux outils ne sont pas toujours facilement intégrables aux langages plus âgés. Il n'a qu'à être considéré l'avènement du paradigme de la programmation orientée objet, qui a entraîné l'abandon de plusieurs langages. Par exemple, le Cobol n'a aujourd'hui plus qu'une petite base d'utilisateurs. Assez peu de développement est maintenant effectué avec ce langage. D'autres paradigmes feront leur apparition dans le futur et rendront de nouveau certains langages désuets.

Même si cela ne fait pas partie des objectifs énoncés pour la réalisation d'un contrôleur à architecture ouverte, il est probable que confiner l'utilisation des contrôleurs à un seul langage est une limitation arbitraire qui aura tôt fait de créer des conflits et des débats dans la communauté. De plus, comme le développement d'applications ayant une étendue beaucoup plus large que de simples programmes de coupe devient chose possible sur ce type de contrôleurs, un seul langage ne peut répondre à tous les besoins des développeurs. Il semble donc essentiel que l'architecture logicielle mise de l'avant donne le choix à l'usager d'utiliser le langage de programmation qu'il désire en tenant compte de ses besoins particuliers.

L'architecture BNCL est bien adaptée à l'utilisation de plusieurs langages de programmation en raison de la présence d'un engin d'exécution de code de bas niveau s'apparentant à un microprocesseur réel. En effet, l'utilisation d'un langage en particulier ne demande que le développement d'un compilateur prenant les sources écrites dans ce langage en entrée et créant les modules BNCL en sortie. Ce processus n'est pas différent de ce qui est fait sur n'importe quel système informatique. Sur un ordinateur personnel, par exemple, plusieurs langages de programmation sont déjà utilisés. Une application peut avoir été programmée à l'aide d'une multitude de langages ou même à l'aide de plusieurs de ces langages en même temps. En bout de ligne, toutes

ces applications se retrouvent sous la même forme, soit un ensemble d'instructions machines. La même situation existe pour l'architecture BNCL. Les programmes peuvent tous provenir de langages différents, mais en bout de ligne ce qui est exécuté sur la BVM est un ensemble d'instructions élémentaires. La Figure 11 montre le cheminement de l'information pour la création de programmes à l'aide de l'architecture BNCL. Il est clairement indiqué qu'à la fin de la chaîne, tous les programmes se présentent à la BVM sous forme de module BNCL indépendamment de leur langage d'origine. Et comme c'est le cas pour les ordinateurs, où presque personne ne programme en assembleur, il est indiqué sur la figure que la programmation ne se fera pas directement dans le langage BNCL, mais à partir de langages de plus haut niveau.

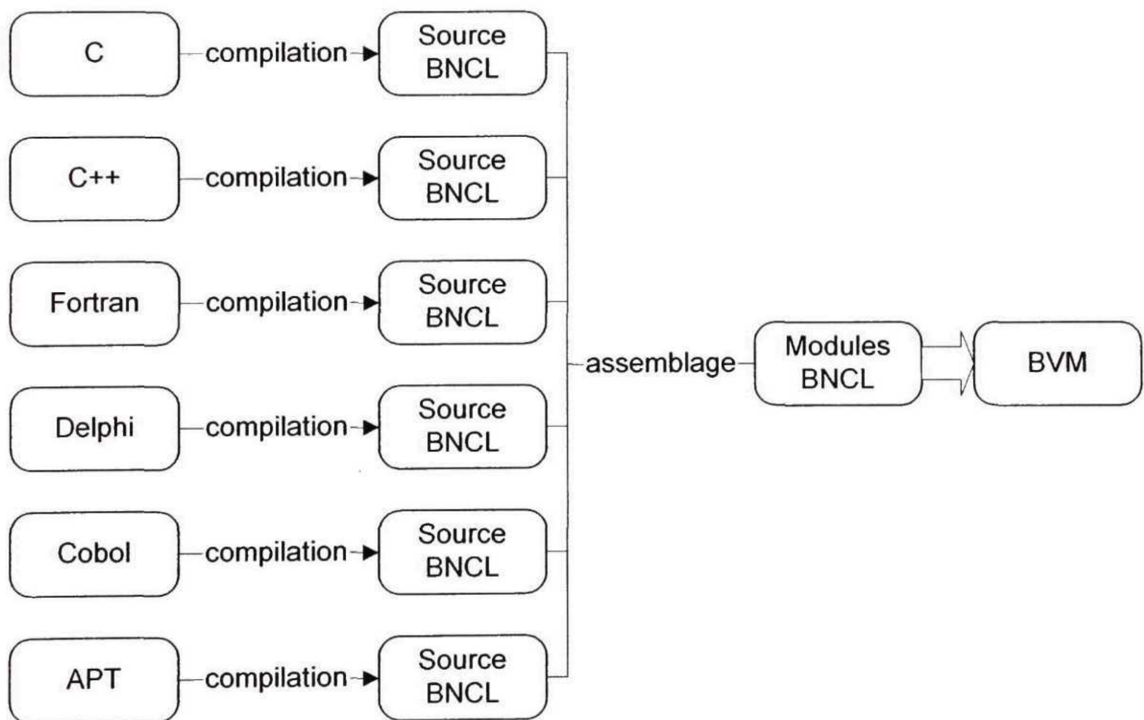

Figure 11 Cheminement de l'information dans le support multi langage

L'interopérabilité entre les langages de programmation peut se faire de plusieurs façons. Dans une architecture comme Microsoft .NET, qui supporte plusieurs langages, l'utilisation de notions de haut niveau dans le jeu d'instructions fait en sorte que tous les langages doivent supporter ces éléments s'ils veulent être en mesure de communiquer entre eux. Si l'architecture supporte directement les classes, chaque langage utilisé doit être orienté objet et supporter les classes. Si l'architecture supporte un autre paradigme quelconque comme les « templates »[3], alors ce paradigme doit être supporté par tous les langages voulant communiquer entre eux. Par exemple, comme Microsoft .NET ne supporte pas l'héritage multiple mais que le langage C++ le supporte, Microsoft a dû intégrer des mots-clés venant restreindre les fonctionnalités du langage C++ pour que celui-ci puisse être utilisé avec cette architecture.

L'interopérabilité entre les langages de programmation utilisés sur l'architecture BNCL se fait au niveau de la communication entre les modules exécutables et non au niveau du jeu d'instructions. L'architecture ne supporte aucune notion de plus haut niveau. Chaque langage est donc isolé des autres et a la liberté d'organiser la mémoire de la façon dont il le désire. Cependant, lorsque la communication avec un autre module est nécessaire, un protocole, cachant les détails de chaque langage, doit être supporté par ceux-ci. Il peut donc être affirmé que ce protocole de communication abstrait les particularités de chaque langage de programmation. Tout ce qu'un langage doit supporter est ce protocole de communication et les types de données fondamentales de l'architecture BNCL. Ces types simples sont les entiers, les nombres décimaux et les adresses mémoire. La Figure 12 montre la communication entre des modules implémentés dans des langages différents.

[3] Concept relié à un langage de programmation et qui permet d'automatiser la création de nouveaux types de données en utilisant ces types comme paramètre. Utilisé en programmation générique.

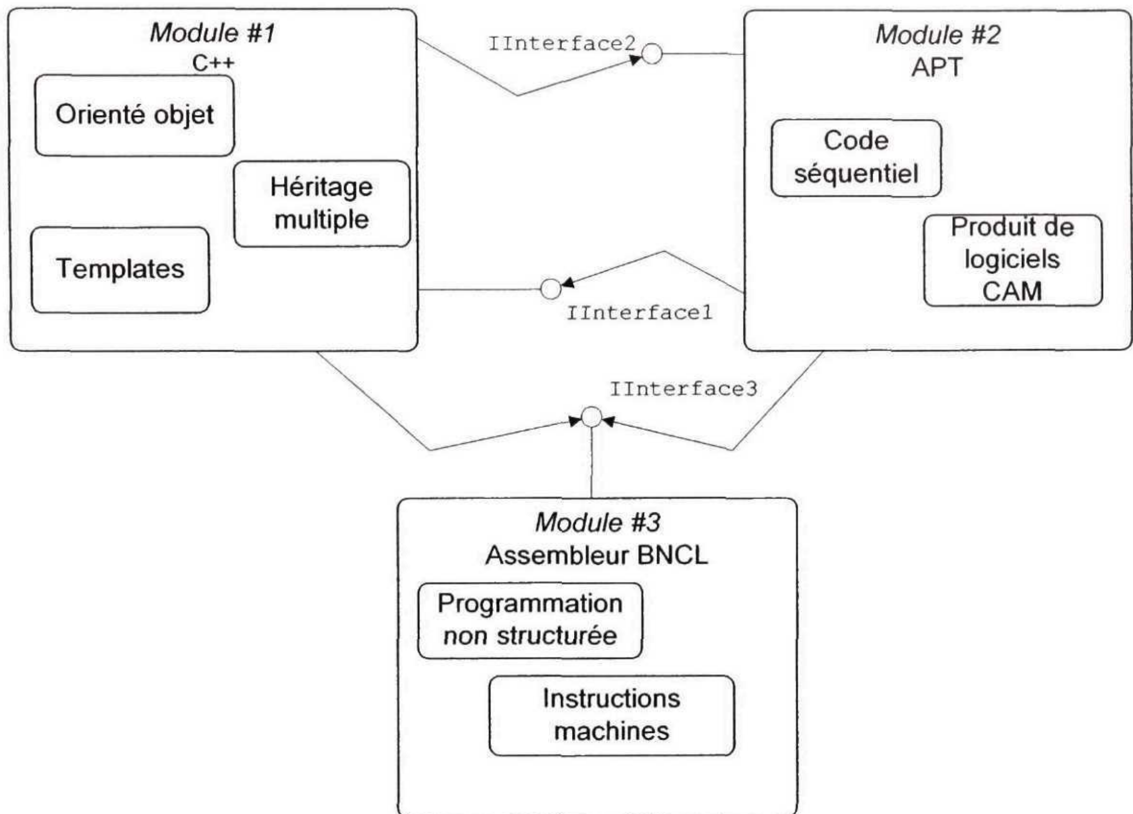

Figure 12 Interopérabilité des langages dans l'architecture BNCL

À partir de la Figure 12, un scénario pourrait être imaginé où une séquence de code est ajoutée manuellement à un fichier source CL-APT provenant d'un logiciel de fabrication assistée par ordinateur (FAO). Cette séquence fait appel à deux routines présentes sur le contrôleur, une se trouvant dans un module implémenté à l'aide du langage C++ et une autre à l'aide de l'assembleur BNCL. Le module programmé en C++ pourrait, par exemple, implémenter la gestion d'une boîte de dialogue. Le module programmé en assembleur BNCL pourrait, quant à lui, exposer un certain nombre de fonctions permettant de dialoguer avec un capteur de température ajouté à la machine-outil. Ce dialogue se ferait évidemment par l'entremise du BVH, mais il serait encapsulé par le module. Dans ce scénario d'exemple, l'usager ajouterait une ligne de code au fichier CL-APT. Ce code provoquerait l'affichage d'une boîte de dialogue demandant à l'opérateur la température maximale qu'il peut tolérer au niveau de l'outil. Ce premier

appel se ferait au module implémenté en C++. Ensuite, il utiliserait les fonctions implémentées dans le deuxième module pour signifier au BVH la température entrée par l'opérateur. Il utiliserait aussi ce module pour indiquer au BVH le signal à envoyer lorsque la température dépasse le seuil. Tous ces modules communiquent alors de façon transparente entre eux.

2.6 Résumé des éléments constitutifs de l'architecture BNCL

L'architecture BNCL repose sur deux composantes principales, soit la machine virtuelle BNCL (BVM) et le matériel virtuel BNCL (BVH). Ces deux composantes couplées ensemble sur un contrôleur permettent de cacher l'ensemble des détails internes de ce contrôleur et de la machine-outil qu'il contrôle. La BVM permet l'abstraction de l'environnement d'exécution. Cet environnement d'exécution comprend le système d'exploitation présent sur le contrôleur, et les différents logiciels fondamentaux gérant le contrôle de la machine. Le BVH, quant à lui, permet l'abstraction de la configuration mécanique de la machine-outil. Les détails sur la position et la dimension des axes, les protocoles électroniques de communication avec les accessoires et les pilotes de périphériques qui les contrôlent, ainsi que les différentes composantes reliées à la machine et leurs particularités sont cachés par cette abstraction. La BVM et la BVH exposent donc un ensemble d'éléments constants d'une machine à l'autre et d'un contrôleur à l'autre, en autant que ces machines-outils possèdent des caractéristiques semblables.

L'abstraction de la machine-outil réalisée par l'architecture BNCL permet de rencontrer les quatre critères principaux qui sont considérés être nécessaires à la réalisation d'un contrôleur à architecture ouverte :

a. la portabilité logicielle ;

b. l'extensibilité de la machine-outil ;

c. la réactivité aux événements ;

d. le support multi langage.

La portabilité logicielle est réalisée par l'utilisation d'un jeu d'instructions de bas niveau nommé jeu d'instructions BNCL. Ce jeu d'instructions possède des caractéristiques très proches du jeu d'instructions d'un microprocesseur réel. Les programmes se présentent donc tous sous la même forme d'un contrôleur à l'autre et sont ainsi portables. Les communications avec les accessoires de la machine-outil passent, pour leur part, par le BVH. Cette composante permet cette communication à l'aide de ports machines. Les programmes communiquant avec la machine-outil sont portables en autant que la signification de ces ports reste constante d'un contrôleur à l'autre. L'extensibilité de la machine-outil est également réalisée par le matériel virtuel BNCL. Les accessoires ajoutés par l'usager peuvent être reliés à des ports sur le BVH et ces ports sont ensuite accessibles par les programmes utilisateur. La réactivité aux événements est réalisée à l'aide d'un signal d'interruption que peut lancer le BVH pour retenir l'attention de la BVM. En réponse à ce signal, la BVM lance l'exécution d'un certain module BNCL en mémoire. Ce module peut être programmé ou non par l'usager. Finalement, le support multi langage provient directement de la standardisation qui est faite autour du jeu d'instructions BNCL. Tout langage possédant un compilateur ciblant ce jeu d'instructions peut alors être utilisé pour créer des modules exécutables sur cette architecture. De plus, ces langages peuvent communiquer entre eux à l'aide d'un protocole de communication simple et accessible à tous les modules BNCL. Ce protocole, l'interface, introduit un troisième niveau d'abstraction dans l'architecture BNCL.

CHAPITRE 3

DÉTAILS DE L'ARCHITECTURE BNCL

Les spécifications concernant l'architecture BNCL sont génériques et ne sont pas liées à un système informatique en particulier. Pour cette raison, cette architecture pourrait être utilisée tant sur des ordinateurs fonctionnant sous le système d'exploitation *Windows* que, par exemple, sous le système d'exploitation *Linux*. L'utilisation de différents microprocesseurs est également possible, tel les processeurs Intel ou Motorola. Il est possible de recréer les mêmes conditions de fonctionnement sur plusieurs types de systèmes puisque l'architecture BNCL possède un engin virtuel d'exécution. Cependant, dans le cadre de ce projet, certaines décisions ont été prises et des choix de technologies ont dû être fait afin de décrire une architecture de référence. Ces choix sont expliqués dans le présent chapitre. Une description des outils logiciels utilisés pour la mise en oeuvre de l'architecture BNCL de référence est d'abord proposée. Par la suite, chacune des composantes de l'architecture BNCL est exposée, de même que les outils créés pour faciliter leur développement.

3.1 Outils de développement utilisés

Plusieurs outils informatiques ont été utilisés pour développer l'architecture BNCL. Ces outils ont permis d'implémenter les concepts élaborés au cours du projet. Ils sont exposés sommairement dans la présente section. L'implication de chacun de ces outils dans la réalisation de l'architecture BNCL est discutée plus en détails tout au long du chapitre. Le lecteur pourra se référer à la présente section s'il désire plus d'informations sur un outil en particulier.

3.1.1 *Component Object Model* et *Object Linking and Embedding*

Le standard COM, ou *Component Object Model*, a été développé par la société Microsoft. C'est une spécification définissant différentes manières de faire permettant de relier des logiciels entre eux. Selon ce standard, tout se présente sous forme d'objets logiciels et ces objets dialoguent entre eux à l'aide d'un protocole bien défini [47]. Le standard COM est également à la base du standard OLE (*Object Linking and Embedding*), lui aussi développé par la société Microsoft. Ce deuxième standard est une extension du premier. Grâce à OLE, il est par exemple possible d'utiliser un diagramme développé à l'aide du logiciel Visio à même une figure d'un document Microsoft Word. La façon dont ce lien est fait est définie par OLE, alors que la communication entre les différents logiciels se fait à l'aide de COM. La Figure 13 présente l'interaction entre ces deux standards. Il peut être constaté que OLE se situe une couche au dessus de COM et que c'est ce dernier qui procure l'infrastructure de communication entre les différents documents.

Plusieurs composantes logicielles sont construites comme des objets COM dans l'architecture BNCL. Cette façon de faire permet de modulariser les programmes créés et favorise la réutilisation de ces modules entre les logiciels.

À retenir

- COM est un standard définissant l'infrastructure nécessaire à la création et à l'utilisation d'objets logiciels.
- COM est à la base de OLE, un standard plus étendu qui permet à des applications de communiquer entre elles et d'échanger des données.
- Un objet logiciel est une entité informatique qui expose des services bien précis à un utilisateur potentiel.

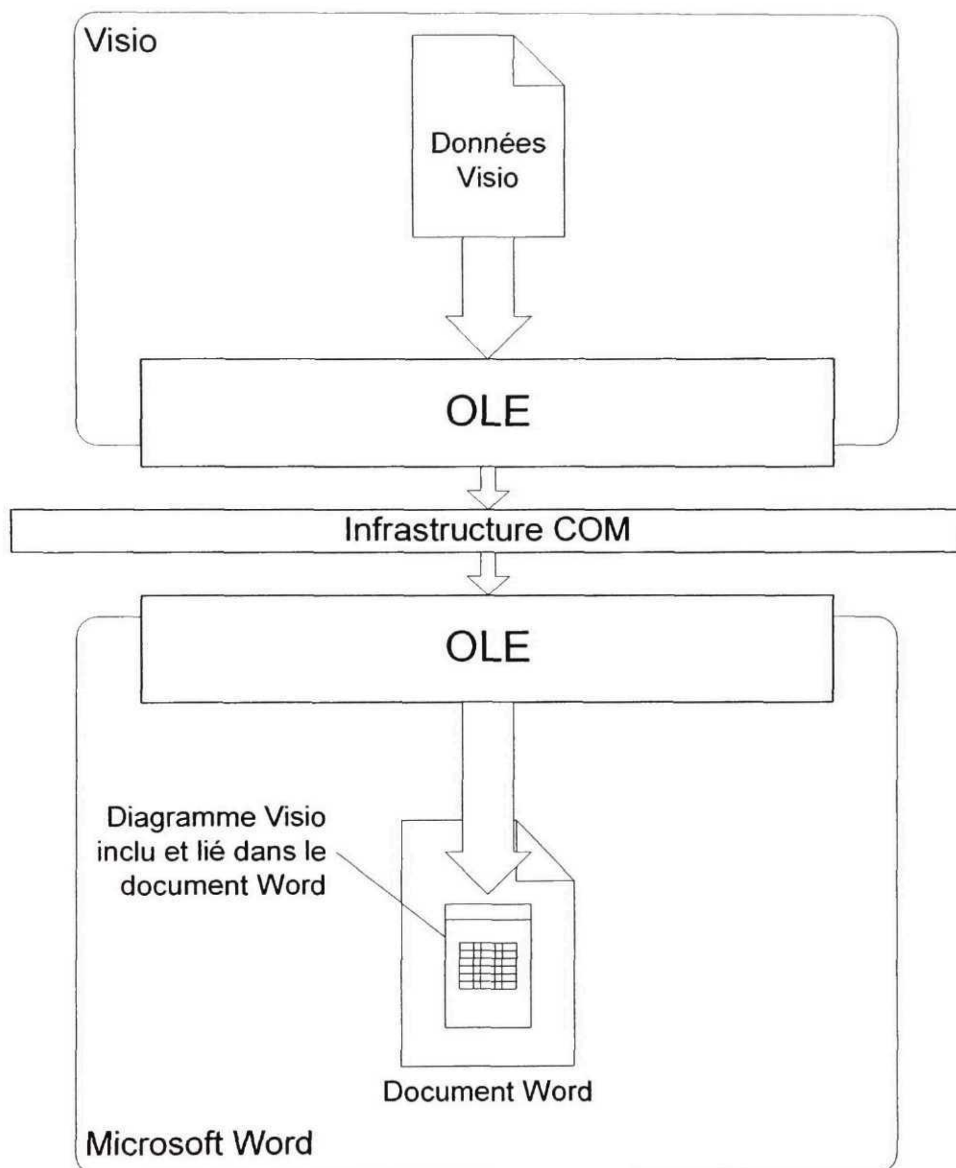

Figure 13 Inclusion de documents grâce à COM et OLE

3.1.2 *Extensible Markup Language* (XML)

Le standard XML est né du désir de faciliter l'échange de données. C'est une norme s'apparentant en syntaxe au HTML (*Hypertext Markup Language*) utilisé pour créer des pages web. La différence majeure entre XML et le HTML est que c'est le créateur des

données qui invente les étiquettes utilisées pour construire le fichier. Une étiquette est un mot-clé associé à une idée. Elle regroupe des données reliées à cette idée. Les étiquettes dans un fichier XML ne sont pas définies par un standard comme c'est le cas pour le HTML [48]. Par exemple, en HTML, pour indiquer au navigateur que le texte doit être en gras, l'étiquette <b> est utilisée :

```
<b>Ce bout de texte est en gras</b>
```

Avec XML, aucune étiquette n'est définie d'avance. Le créateur des données pourrait, par exemple, décider que le texte en gras est indiqué par l'étiquette <gras>, comme dans l'exemple suivant :

```
<gras>Ce bout de texte est en gras</gras>
```

```
<facture>
     <client>
          <nom>Etienne Fortin</nom>
          <ref>478744894382894</ref>
     </client>
     <items>
          <item no_inventaire='I437884' qte='5'/>
          <item no_inventaire='B588390' qte='2'/>
     </items>
</facture>
```

Figure 14 Exemple de fichier XML

Simplement remplacer les étiquettes HTML par d'autres n'est cependant pas très utile. HTML a été conçu pour la présentation des données et fait bien son travail. Cependant, avoir la possibilité d'inventer ses propres étiquettes ouvre tout un éventail de possibilités dans l'échange de données. Par exemple, une compagnie de commerce au détail pourrait définir un format d'étiquettes pour représenter une facture. Un exemple est présenté à la Figure 14. Les étiquettes <facture>, <client>, <nom>, <ref>, et les autres, sont toutes inventées par le créateur des données.

Deux entreprises qui s'entendent sur le format à donner à leurs factures pourront désormais échanger l'information sous forme de fichiers XML. La simplicité du fichier XML permet de traiter cette information rapidement à l'aide d'un logiciel. Le partage de l'information se fait donc de façon plus efficace. Il peut être également noté qu'il est possible de faire différents usages de ces données. Par exemple, il est possible de les traiter avec un logiciel de comptabilité, d'afficher l'information sur une page web, de créer un historique de transactions du client dans une base de données et bien d'autres. À partir du même fichier, de la même information, il est maintenant possible d'effectuer plusieurs tâches. Une source unique de données est utilisée, mais de multiples utilisations en sont faites. Les risques d'erreurs attribuables à la duplication des sources de données sont donc éliminés.

Lors de la création de l'architecture BNCL, toutes les spécifications qui ont été développées ont été écrites à l'aide de XML. L'information contenue dans ces spécifications a par la suite été utilisée pour documenter les spécifications, pour créer le squelette de code source d'implémentation des composantes logicielles et pour d'autres tâches connexes. Une seule source d'information a donc été utilisée pour effectuer ces différentes tâches.

À retenir

- XML est un moyen d'organiser l'information pour la rendre facilement communicable et utilisable par les différentes personnes ou applications devant l'exploiter.

3.1.3 *Extensible Stylesheet* (XSL)

Un standard a été élaboré pour faciliter la transformation des données se présentant sous forme XML. Ce standard définit la manière dont un fichier XML peut être transformé pour présenter ses données de différentes façons. Ces entités de transformation se

présentent elles aussi en format XML. Par exemple, un fichier XML représentant une facture pourrait être transformé en page HTML à l'aide d'une de ces entités de transformation. Ou encore, ces mêmes données pourraient être transformées d'un format de facture d'une entreprise à celui d'une autre selon les numéros d'inventaire, les numéros de client et les noms des étiquettes choisies pour définir la facture. Il serait également possible de générer des données Word ou PDF à partir de ces données et de ce système de transformation. Ce standard se nomme *Extensible Stylesheet* ou XSL [49].

Dans le projet, XSL est utilisé intensivement dans la génération automatique de code à partir des spécifications développées. Ces spécifications décrivent certaines parties de l'architecture BNCL, comme par exemple le jeu d'instructions. XSL est également utilisé pour générer la documentation associée à chacune de ces spécifications. Le résultat est présenté dans un format compatible avec le logiciel d'aide de Windows. Cette documentation se présente donc sous forme de fichiers d'aide et est utilisable comme telle.

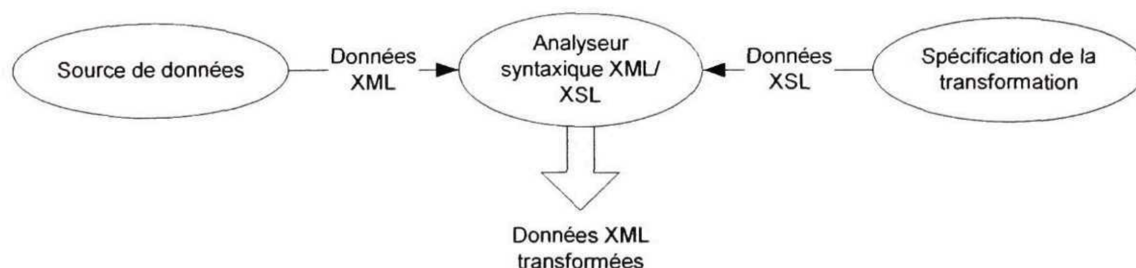

Figure 15 Transformation de données XML à l'aide de XSL

La Figure 15 montre les relations entre les données XML et les données XSL de transformation. Il peut être constaté que le coeur de tout le système de transformation se trouve dans l'analyseur syntaxique, communément appelé *parser*. C'est cette composante qui analysera le fichier de données XML et qui le transformera selon ce qui

est spécifié dans les données de transformation. Un aspect qui rend l'ensemble XML/XSL attrayant est qu'une multitude d'analyseurs syntaxiques XML existent sur le marché et que la plupart sont gratuits.

À retenir

- XSL signifie *Extensible Stylesheet*.
- Un fichier XSL permet de transformer, selon ses besoins, les données contenues dans un fichier XML.

3.1.4 *Practical Extraction and Report Language* (Perl)

Perl est un langage informatique permettant d'écrire des scripts, ou programmes automatisant des tâches. Le langage Perl a vu le jour en 1987 et a connu de nombreuses modifications depuis. Il est idéal pour traiter des fichiers texte. Son nom complet est d'ailleurs *Practical Extraction and Report Language ou* « langage de génération de rapports et d'extraction de données». Son usage n'est toutefois pas restreint à du simple traitement de données textuelles [50].

Plusieurs utilitaires ont été programmés avec Perl dans le développement de l'architecture BNCL. Perl a l'avantage de permettre un développement rapide avec un minimum d'effort. C'est pour cette raison que plusieurs utilitaires ont été programmés avec ce langage.

À retenir

- Perl est un langage informatique permettant d'écrire des scripts.
- Un script est un petit programme d'automatisation de tâches complexes.

3.1.5 Cygnus Windows

Les systèmes d'exploitation Windows et Unix diffèrent sur de nombreux aspects. Alors que Windows se concentrait à offrir une interface graphique toujours plus raffinée, Unix développait de nombreux utilitaires et outils permettant de faciliter le travail des utilisateurs et surtout des administrateurs de réseaux. Ces deux systèmes d'exploitation possèdent bien sûr leurs avantages et inconvénients propres.

Dans un souci de faire profiter des avantages de l'environnement de travail Unix aux utilisateurs de Windows, le projet Cygwin (Cygnus Windows) a été lancé. Ce projet a tout d'abord été dirigé par le groupe Cygnus. Il est maintenant parrainé par la compagnie Red Hat, concepteur du système d'exploitation Red Hat Linux [51]. Cygwin tente d'émuler l'environnement de travail de Unix à l'intérieur de Windows. De nombreux utilitaires présents sous Unix ont ainsi été portés vers Windows. Le comportement de ces utilitaires tente de reproduire le plus possible celui rencontré sous Unix. Cygwin offre donc un environnement Unix complet sous Windows.

Cet environnement d'émulation a été utilisé pour réaliser quelques composantes de l'architecture BNCL puisque certaines ressources nécessaires au développement n'étaient disponibles que sous Unix.

À retenir

- Cygwin est un environnement d'émulation de Unix pour Windows.
- Cygwin est développé par Red Hat, les concepteurs du système d'exploitation Red Hat Linux.

3.1.6 Traitement multiprocessus

De façon générale, un programme possède un seul flux de code. Les instructions de ce flux sont exécutées l'une après l'autre jusqu'à la fin de l'exécution du programme. Lorsqu'un logiciel possède plusieurs flux de code exécutés en même temps, celui-ci est qualifié de programme comportant un traitement multiprocessus (*multithread*). Évidemment, si l'ordinateur ne possède pas plusieurs processeurs, ces flux ne sont pas véritablement exécutés en même temps : ils ne le sont alors qu'en apparence. Le processeur travaille de façon partagée entre chacun de ces flux. Cette technique est pertinente lorsque l'ordinateur doit traiter plusieurs tâches en même temps, comme par exemple traiter une boîte de dialogue alors qu'un calcul est effectué en arrière-plan.

Dans un programme multiprocessus, un problème de synchronisation des flux survient. Le système d'exploitation propose des fonctions permettant d'effectuer cette synchronisation. Des standards comme COM (voir 3.1.1) proposent également des modèles de synchronisation [47]. La synchronisation faite par défaut par COM est cependant très lourde et demande beaucoup de ressource. Un objet COM peut limiter les ressources attribuées à la synchronisation en effectuant celle-ci à l'aide des fonctionnalités du système d'exploitation. C'est ce qui a été fait pour l'architecture BNCL de référence.

À retenir

- Un programme dit *multithread* est un programme dans lequel s'exécute plusieurs flux de code.
- Un objet COM est synchronisé par défaut, mais cette synchronisation est très lourde en terme de ressources.
- Un objet COM peut éviter la synchronisation par défaut, mais dans ce cas il doit se synchroniser lui-même.

3.2 Spécification du jeu d'instructions

À la base de la machine virtuelle BNCL se trouve le jeu d'instructions de bas niveau calquant les fonctionnalités d'un microprocesseur réel (voir exemple d'instructions à la Figure 4 de la section 2.1.1). Ce sont ces instructions, lorsqu'elles sont exécutées sur une machine virtuelle BNCL, qui procurent un travail utile. Le jeu d'instructions BNCL est donc au coeur de toute l'architecture. De nombreux outils et utilitaires, comme l'assembleur BNCL, se basent également sur ce jeu d'instructions.

La spécification du jeu d'instructions BNCL a été définie à l'aide de XML. Des étiquettes ont été définies pour décrire les instructions et les tâches qu'elles accomplissent. D'autres étiquettes ont été définies pour permettre de décrire les différents registres de l'architecture ainsi que le format binaire des instructions. Tout ce qu'il est nécessaire de connaître du jeu d'instructions pour développer une machine virtuelle BNCL est présent dans cette spécification.

Essentiellement, cette spécification est utilisée à trois endroits : pour créer le code d'implémentation de la machine virtuelle BNCL (voir section 3.3), pour créer le code d'implémentation de l'assembleur BNCL (voir section 3.6), et pour générer la documentation de chacune des instructions. Ces trois composantes utilisent donc la même source de données. Lorsqu'un changement est apporté à la spécification, le code d'implémentation de la machine virtuelle et de l'assembleur, ainsi que la documentation, peuvent être régénérés automatiquement.

Trois utilitaires ont été développés et sont employés pour réaliser le tout. Ces utilitaires sont programmés en partie à l'aide du langage Perl et en partie à l'aide de fichiers XSL. Ils transforment les données contenues dans la spécification selon la tâche qu'ils ont à effectuer. Le premier et le plus important de ces utilitaires est FVMgen (*Flexible Virtual Machine generator*). C'est cet outil qui permet de créer en entier le code du coeur de la

machine virtuelle. La machine virtuelle, après l'utilisation de FVMgen, est prête à être utilisée pour exécuter des modules BNCL. Le second utilitaire est celui permettant de créer le code d'implémentation de l'assembleur BNCL. C'est l'assembleur qui permet de générer les modules BNCL exécutables à partir de code source. Finalement, le troisième utilitaire permet de générer la documentation dans un format compatible avec l'aide de Windows. Les relations entre tous ces utilitaires sont présentées à la Figure 23, à la dernière section du chapitre.

3.3 Machine virtuelle BNCL (BNCL Virtual Machine)

Comme il en a déjà été fait mention à plusieurs reprises, l'architecture BNCL permet de faire l'abstraction des particularités physiques de la machine et des particularités logicielles de l'environnement d'exécution. La machine virtuelle BNCL (BVM) est au coeur de l'abstraction logicielle du contrôleur. Le matériel virtuel BNCL (BVH) fait, pour sa part, abstraction des particularités physiques de la machine-outil. La machine virtuelle BNCL cache les particularités du système d'exploitation ainsi que de toute l'infrastructure logicielle du contrôleur. Un module BNCL exécutable ne sait pas sur quel type d'ordinateur il est exécuté puisque tout passe par cette machine virtuelle. Pour réaliser la machine virtuelle BNCL et l'environnement d'exécution faisant abstraction du contrôleur, quelques choix technologiques ont dû être fait. Ils seront maintenant présentés.

3.3.1 Fonctionnement

La machine virtuelle BNCL se présente sous la forme d'un utilitaire s'exécutant à la ligne de commande. L'usager a la possibilité d'indiquer quel module BNCL il désire exécuter. Il peut également indiquer quelle portion de ce module doit être exécutée. Cette indication se fait sous la forme d'un nom de fonction. Par exemple, si l'usager

désire exécuter la fonction `fct1` du module `BnclModuleTest.bncl`, alors la ligne de commande pourrait être :

```
ConsoleBVM BnclModuleTest.bncl fct1 -config cfg.xml
```

Lorsque la ligne de commande précédente est entrée et que ConsoleBVM est exécuté, les événements suivants se produisent :

a. le fichier de configuration est chargé en mémoire (`cfg.xml`) ;
b. la composante logicielle faisant la gestion des ports standards est chargée ;
c. la composante logicielle faisant la gestion des ports personnalisables est chargée.
d. tous les modules BNCL ayant des fonctions système sont chargés en mémoire. Les modules système implémentent des fonctionnalités telles que l'accès aux disques, l'allocation de mémoire et autres. Ces modules sont spécifiés dans le fichier de configuration ;
e. le module de base, spécifié à la ligne de commande par l'usager (`BnclModuleTest.bncl`), est chargé en mémoire ;
f. la machine virtuelle est initialisée pour exécuter la méthode spécifiée à la ligne de commande (`fct1`) ;
g. la boucle de lecture des instructions est lancée, ce qui équivaut à lancer l'exécution de la fonction spécifiée à la ligne de commande.

Cette série d'événements se produit à chaque lancement de la machine virtuelle. Le coeur de cette machine virtuelle est la boucle de lecture des instructions. C'est cette boucle qui exécute une à une les instructions BNCL contenues dans un module. C'est également cette boucle de lecture des instructions qui est automatiquement implémentée par l'utilitaire FVMgen et la spécification XML du jeu d'instructions. Le reste du code d'implémentation de la machine virtuelle est fixe et n'est pas touché par l'utilitaire FVMgen. La Figure 16 montre le fonctionnement du système de génération automatique du code d'implémentation de la machine virtuelle. L'utilitaire FVMgen permet de

générer le code de lecture et d'exécution des instructions BNCL. Ce qui englobe cette boucle, le code invariable, n'est pas généré par cet utilitaire. Le code invariable supporte des fonctionnalités telles que l'interprétation des options entrées à la ligne de commande, l'interprétation du fichier de configuration, le chargement initial des modules et autres. Tout ce qui n'a pas de lien avec l'exécution en temps que tel des instructions BNCL se trouve dans le code invariable.

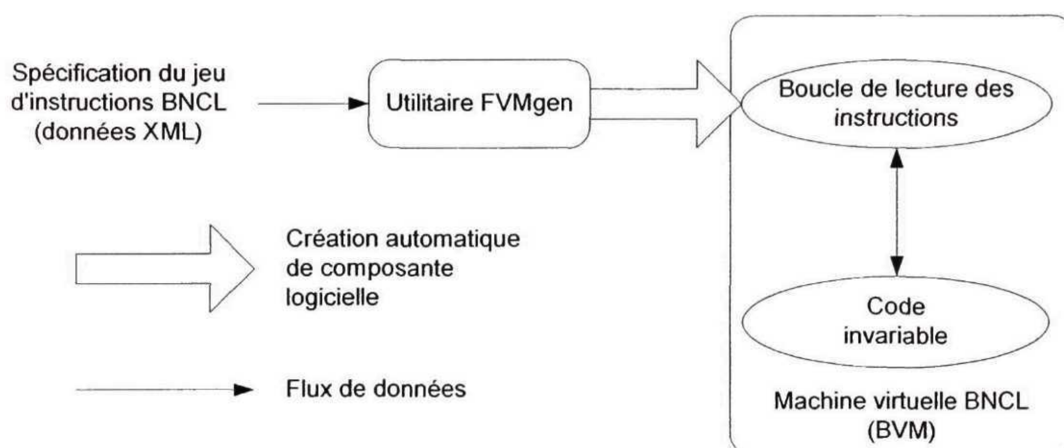

Figure 16 Génération de la machine virtuelle BNCL

3.3.2 Fichier de configuration

Dans la section précédente, il a été mentionné que lors du lancement de la machine virtuelle BNCL, un fichier de configuration est analysé. Ce fichier de configuration contient les informations suivantes :

a. l'identification du gestionnaire du groupe de ports standard (BVH) ;
b. l'identification du gestionnaire du groupe de ports personnalisables (BVH) ;
c. la liste des modules système à charger en mémoire.

Le fichier de configuration peut être spécifié à la ligne de commande. Ceci permet de changer de configuration rapidement tout en utilisant les mêmes modules BNCL. Par

exemple, les modules permettant de réaliser les tests de performance sont également utilisés pour vérifier si les points générés sont valides. Cet exemple est expliqué dans le chapitre sur la validation. Pour ce faire, différents fichiers de configuration sont utilisés. Un de ces fichiers spécifie un BVH (*BNCL Virtual Hardware*) permettant de calculer la performance en termes de points par seconde, alors qu'un autre spécifie un BVH traitant les points générés. Ces deux BVH ont des fonctions différentes, mais exposent les mêmes ports à un module BNCL. Un exemple de fichier de configuration, utilisé dans le cadre du test d'extensibilité, est présenté à l'ANNEXE 2.

3.4 Matériel virtuel BNCL (BNCL Virtual Hardware)

De la même manière que la machine virtuelle BNCL fait l'abstraction de l'environnement logiciel du contrôleur, le matériel virtuel BNCL fait l'abstraction de l'environnement physique de la machine. C'est cette composante qui permet de cacher les particularités physiques de la machine, telles que la disposition des axes, la position de la broche, la manière d'utiliser le fluide de refroidissement et bien d'autres. Cette composante est également au coeur de l'extensibilité de la machine-outil permise par l'architecture BNCL. En effet, avec cette architecture, deux groupes de ports sont utilisés : le groupe des ports standards et le groupe des ports personnalisables. Ces groupes sont tous deux contrôlés à l'aide de BVH. Ce sont les ports personnalisables qui permettent l'accès à des composantes ajoutées à la machine-outil par l'usager. Les détails de l'implémentation de ce matériel virtuel, ainsi que son fonctionnement, seront maintenant vus.

3.4.1 Mise en oeuvre

Dans l'architecture BNCL de référence, le matériel virtuel, ou BVH, est construit à l'aide d'un objet COM. Cet objet COM encapsule un groupe de ports et en contrôle

l'accès. Pour manipuler l'information, l'objet expose certaines fonctions pour écrire ou lire sur les ports. Par l'accès à ces ports, un programme BNCL peut contrôler la machine sous-jacente. De plus, ces BVH sont accessibles à partir de scripts comme Perl ou VBScript. Cette possibilité d'accéder au matériel virtuel BNCL à l'aide de scripts a été conçue pour faciliter les tests sur cette composante pendant leur développement.

Le code d'implémentation d'un BVH est généré automatiquement à partir d'une spécification des ports. Cette spécification, écrite à l'aide de XML, permet de décrire chacun des ports contrôlés par le BVH (voir ANNEXE 5). Le code permettant la gestion de ces ports par le BVH est généré à l'aide d'un script Perl et d'un fichier XSL. Le reste du code d'implémentation du BVH est généré à l'aide d'un système d'assistants[4] (*wizards*) conçu dans le cadre du projet. Ce système d'assistants, nommé FWS[5], a été développé à l'aide de Perl et de fichiers XML et XSL. La Figure 17 présente le fonctionnement de ce système dans la création automatique du code d'implémentation d'un BVH. Les patrons de code présentés sur cette figure contiennent le code source de la structure du programme et de plusieurs fonctions qui ne sont pas touchées par l'utilitaire *bvh_code_generator*.

Il doit également être mentionné que pour assurer des performances adéquates, le BVH est implémenté à l'aide de plusieurs flux de code, ou en d'autres mots de façon *multithread*. Par exemple, un flux est utilisé pour contrôler les demandes de modification des ports alors qu'un autre flux, exécuté en même temps, permet de traiter ces modifications et de les valider. La synchronisation des flux de code est effectuée à l'aide des fonctionnalités du système d'exploitation.

---

[4] Composante logicielle permettant l'automatisation de tâches répétitives de création de documents ou de code.

[5] Flexible Wizard System

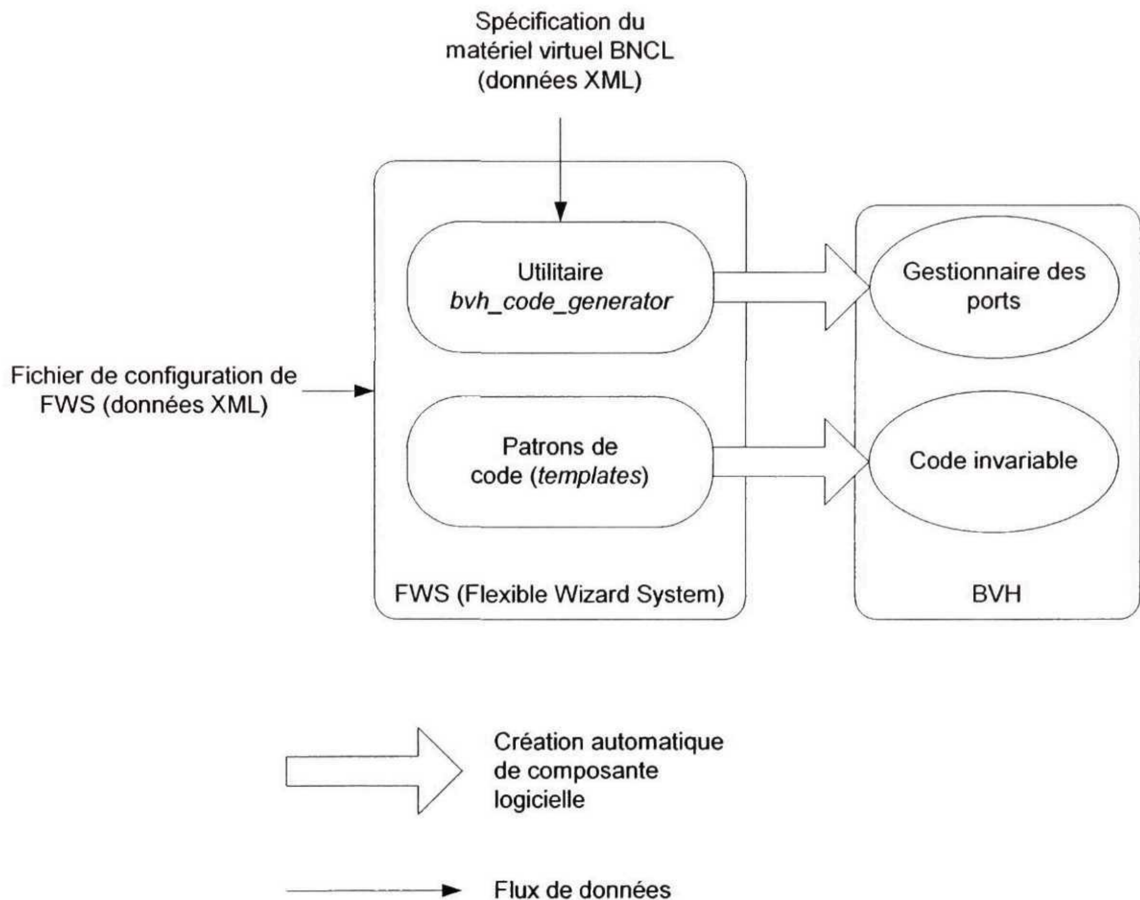

Figure 17 Génération automatique du code d'implémentation d'un BVH

3.4.2 Fonctionnement

Le matériel virtuel permet de contrôler l'accès aux ports. Dans l'architecture BNCL, les ports peuvent permettre l'accès à une donnée entière ou à une donnée décimale. Les ports sont identifiés comme étant des ports entiers, ou des ports de nombres réels. L'accès à un port entier avec une donnée réelle provoque une erreur, et vice-versa. Ces conditions d'erreur sont signifiées à la machine virtuelle BNCL à l'aide de signaux d'interruption. Ces signaux sont mis en place lorsque la machine virtuelle charge le BVH en mémoire.

Les interruptions du BVH vers la machine virtuelle permettent de signaler, outre les erreurs, que certaines conditions ont été observées. Par exemple, lorsque la température atteint un certain niveau dans l'exemple employant un thermocouple (voir Figure 8 section 2.3), un signal est envoyé à la machine virtuelle pour que l'action appropriée soit exécutée. Les signaux d'interruption utilisent les fonctionnalités d'événements de COM. Ce système de signaux est à la base de la réactivité aux événements de l'architecture BNCL. Cette réactivité aux événements est un des concepts menant à la réalisation d'un contrôleur à architecture ouverte, comme il a déjà été mentionné.

3.5 Répertoire de modules

Pour faciliter le chargement des modules BNCL, il a été décidé de créer une composante logicielle faisant la gestion de ceux-ci. Tout comme le matériel virtuel BNCL, cette composante est implémentée en suivant le standard COM. L'objet COM gère les différents modules pouvant être chargés en mémoire. Ces modules peuvent se trouver à deux endroits, soit dans le répertoire central ou dans un fichier. Le répertoire central permet de faciliter l'accès aux modules, indépendamment de leur localisation. De plus, le répertoire rend disponible de l'information concernant ces modules, comme par exemple les fonctions implémentées par le module, leurs noms et autres. Cette fonctionnalité est d'ailleurs utilisée par la machine virtuelle BNCL lorsqu'il est nécessaire de vérifier si la fonction de départ spécifiée à la ligne de commande existe pour le module de base spécifié.

3.6 Assembleur BNCL

Lorsqu'un compilateur compile des sources de langages de haut niveau, celui-ci produit un fichier source BNCL. Ce fichier source n'est cependant pas encore exécutable. Il ne

contient que des instructions sous forme textuelle. Pour devenir un module exécutable à part entière, ce fichier source doit être assemblé. Cet assemblage produit un code binaire et c'est ce code qui est exécuté par la machine virtuelle. La Figure 18 présente l'interaction entre le compilateur, l'assembleur et la machine virtuelle BNCL. Il y est indiqué que l'assembleur est la composante qui produit un module exécutable, alors que le compilateur s'organise pour générer un fichier source BNCL compatible avec cet assembleur. Cette compilation se fait à partir de code source d'un langage de haut niveau, tel le C ou le Fortran.

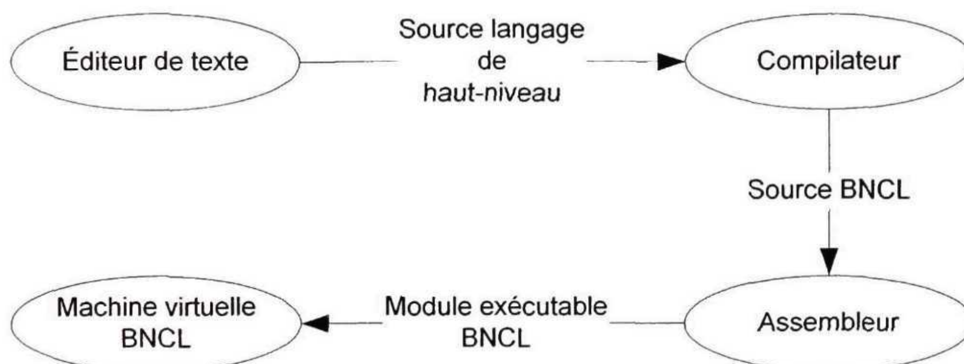

Figure 18 Interaction entre un compilateur et l'assembleur BNCL

3.6.1 Mise en oeuvre

L'assemblage, dans l'architecture BNCL de référence, est réalisé à l'aide d'un utilitaire nommé *basm*. Cet utilitaire est implémenté en partie par un objet COM et en partie par un script Perl. L'objet COM permet l'assemblage brut, c'est à dire l'encodage du segment de données du module et du code de chacune des fonctions. Il traite une partie du fichier source à la fois, une fonction à la fois. Le script Perl, quant à lui, permet de récupérer l'encodage réalisé par l'objet COM et d'inclure cette information dans un fichier que le répertoire des modules pourra lire et charger à la demande de la machine

virtuelle. C'est le script Perl qui interprète le code source BNCL et qui s'assure de la validité du module BNCL exécutable créé. Le fonctionnement de l'assembleur BNCL est montré à la Figure 19. L'interaction entre le script Perl (*basm.pl*) et l'objet COM d'assemblage (*BnclAssembler*) y est montrée. La figure montre également le rôle de chacune des composantes, à savoir que *BnclAssembler* assemble des morceaux de code source et produit un code binaire assemblé. Le script Perl récupère ces portions de code assemblé et les intègre au fichier du module BNCL exécutable.

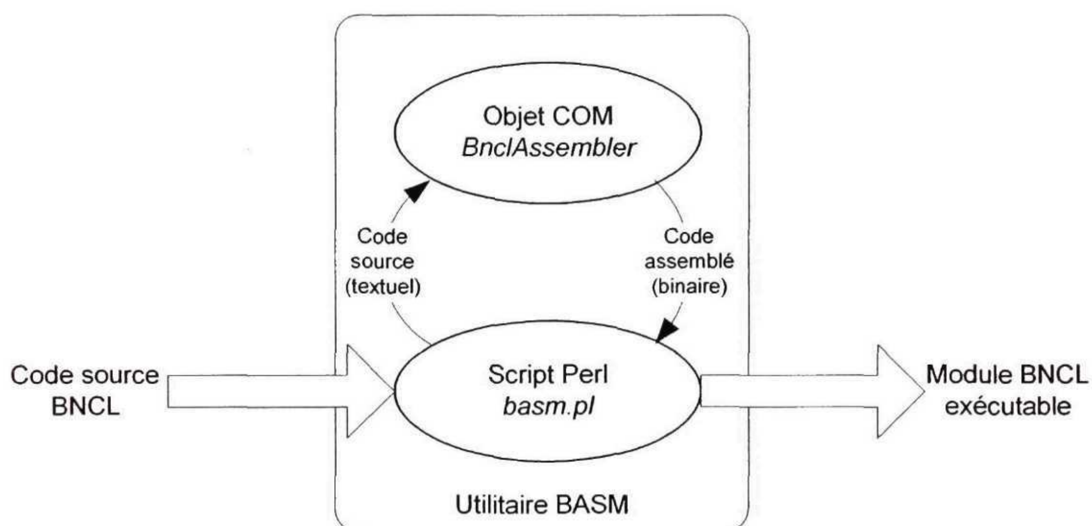

Figure 19 Utilitaire d'assemblage BASM

3.7 Compilateur C

Dans un souci de vérifier la faisabilité de l'utilisation de plusieurs langages de programmation sur l'architecture BNCL, un compilateur C a été créé. Le langage C est assez général, et son implémentation assez commune, pour laisser supposer que le support multi langage est possible sur l'architecture BNCL. La disponibilité d'un tel compilateur permet également à l'usager d'éviter d'écrire ses programmes à l'aide de la syntaxe du jeu d'instruction BNCL (voir Figure 4 à la section 2.1.1 pour un exemple d'instructions). Le langage C est de plus un langage très utilisé et bien reconnu dans le

monde informatique. La disponibilité d'un compilateur pour le langage C sur l'architecture BNCL procure donc un atout non négligeable.

3.7.1 Mise en oeuvre

Il existe sous l'environnement Unix une collection de compilateurs qui se nomme GCC, ou *Gnu Compiler Collection*. Cette collection contient des compilateurs pour de nombreux langages, tels les langages C, C++, Ada et Fortran. Elle est, de plus, disponible sur de nombreux ordinateurs fonctionnant sous une variante de Unix [52]. Cette popularité provient de deux aspects : la collection est entièrement gratuite et elle a été conçue pour faciliter la création de variantes pour des architectures différentes. Elle était donc un choix idéal pour la création d'un compilateur C pour l'architecture BNCL.

Un problème majeur à contourner résidait dans le fait que l'architecture BNCL de référence est développée sous Windows, alors que GCC n'est disponible que sous Unix. Pour contourner ce problème, et pour profiter des avantages qu'offrent GCC, un environnement d'émulation Unix a été utilisé. Cet environnement, nommé Cygwin, procure pour Windows les utilitaires et l'interface de ligne de commande disponibles sur une station Unix. La plupart des programmes destinés à être exécutés dans un environnement Unix peuvent être utilisés avec Cygwin. Cet environnement d'émulation a également l'avantage d'être gratuit.

L'implémentation du compilateur C pour l'architecture BNCL à l'aide de GCC passait par la création d'une nouvelle variante de GCC. Pour se faire, il a été nécessaire d'instruire GCC sur les caractéristiques de l'architecture BNCL : les registres disponibles, les instructions arithmétiques, les instructions pour appeler des procédures, les instructions de contrôle du flux du programme et autres. Chacune des caractéristiques

de l'architecture devait être documentée. La Figure 20 montre la génération du compilateur C à l'aide de GCC et des informations sur l'architecture BNCL.

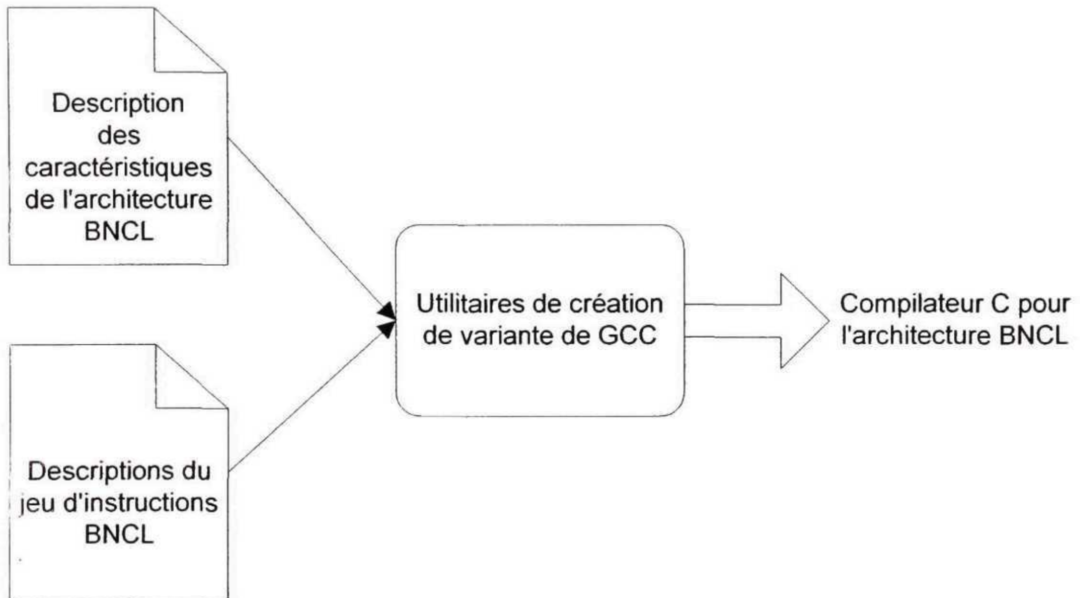

Figure 20 Création d'un compilateur C à l'aide de GCC

3.7.2 Fonctionnement

Le compilateur C pour l'architecture BNCL fonctionne sous l'environnement Unix. Pour l'architecture de référence, cet environnement est émulé sur une station Windows par Cygwin. L'usager doit donc compiler ses programmes C dans cet environnement à l'aide du compilateur C pour l'architecture BNCL créé à partir de GCC. Après avoir créé le code source BNCL à partir de ses sources en langage C, l'utilisateur retourne dans l'environnement Windows et utilise l'assembleur BNCL pour générer un module BNCL exécutable. Cependant, un problème de transfert des données entre l'environnement Unix et Windows se pose. En effet, le format des données générées par GCC n'est pas compatible avec le format des données que s'attend à retrouver l'utilitaire *basm*. De plus, l'utilitaire *basm* utilise un objet COM alors que ceux-ci ne sont pas compatibles avec l'environnement Unix. Forcer GCC à générer des données compatibles directement avec l'assembleur BNCL serait également ardu. Le transfert se fait donc à l'aide d'un script Perl, *blink*, qui permet de convertir les données générées par GCC en des données compatibles avec l'assembleur BNCL. L'ensemble du processus de création d'un module BNCL exécutable à partir de code source en langage C est montré à la Figure 21.

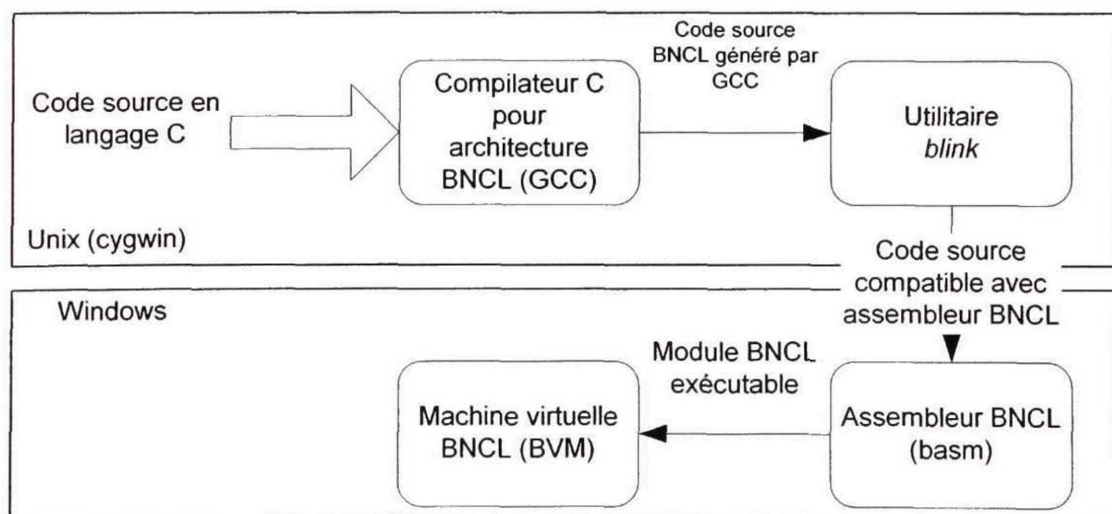

Figure 21 Génération d'un module BNCL exécutable à l'aide de GCC

3.8 Utilisation des outils et techniques développés

Un test sur l'extensibilité permise par l'architecture BNCL est décrit au chapitre portant sur la validation (voir Chapitre 6). Ce test simule l'adaptation de la vitesse de rotation de la broche de la machine-outil en fonction de la température observée sur un capteur ajouté à la machine par l'usager. Les détails des composantes logicielles utilisées pour réaliser ce test seront maintenant exposés. Ces détails permettent de saisir le travail réalisé dans le développement de l'architecture BNCL.

3.8.1 Étapes de préparation

Les étapes suivantes ont été franchies pour réaliser le test portant sur l'extensibilité. Il est à noter que certaines des composantes qui sont énumérées, comme la machine virtuelle BNCL, sont réutilisables d'un essai à l'autre :

a. création de la machine virtuelle BNCL. Une portion du code d'implémentation C++ de cette composante est présentée à l'ANNEXE 3. La création de cette composante a nécessité l'utilisation de l'utilitaire FVMgen (voir section 3.3.1). L'exécutable résultant est nommé *ConsoleBVM* ;
b. création du BVH *ThermocoupleReader*. Une portion du code d'implémentation C++ de cette composante est présentée à l'ANNEXE 4. La création de cette composante a nécessité l'utilisation de l'utilitaire *bvh_code_generator* (voir section 3.4.1). Le fichier XML de description du BVH, utilisé par *bvh_code_generator*, est présenté à l'ANNEXE 5 ;
c. assemblage du module BNCL effectuant le test. Le module BNCL, avant et après assemblage, est présenté à l'ANNEXE 6 ;
d. création du fichier de configuration pour la machine virtuelle (voir ANNEXE 2) ;
e. connexion de la carte d'acquisition et du thermocouple ;
f. lancement de la machine virtuelle à la ligne de commande :

BvhThermocoupleTest1.bncl = Nom du module BNCL à exécuter.

reader1 = Nom de la fonction à être exécutée.

```
"ConsoleBVM BvhThermocoupleTest1.bncl reader1 -filename
-verbose  -config vmconfig_ThermocoupleReader.xml"
```

Les annexes mentionnées dans cette liste contiennent une partie du code utilisé dans la génération des composantes logicielles de l'architecture BNCL. Une des spécifications développées pour décrire les composantes de l'architecture BNCL est également présente.

3.8.2 Étapes de réalisation

Une certaine séquence d'événements se produit lorsque la machine virtuelle est lancée. Cette séquence d'événements est montrée à la Figure 22. Au démarrage de la machine virtuelle, la fonction *reader1* du module *BvhThermocoupleTest1.bncl* communique au BVH la valeur de la température à partir de laquelle un signal est désiré. Cette fonction communique également la vitesse de révolution initiale désirée. Par la suite, la fonction entre dans une boucle d'attente. Dans un environnement réel de production, cette boucle d'attente serait remplacée par un travail utile. Pour fin de test, ce travail supplémentaire n'est pas nécessaire. L'usager fait par la suite augmenter la température sur le thermocouple en utilisant, par exemple, une chandelle. Lorsque le BVH détecte que la température atteint le niveau spécifié, un signal est envoyé au module *BvhThermocoupleTest1.bncl.* Ce signal est alors capté par la fonction *temperatureTrigger*. Cette fonction affiche un message et demande au BVH de modifier la vitesse de rotation de la broche. L'usager termine le test en appuyant sur CTRL-C pour fermer la machine virtuelle.

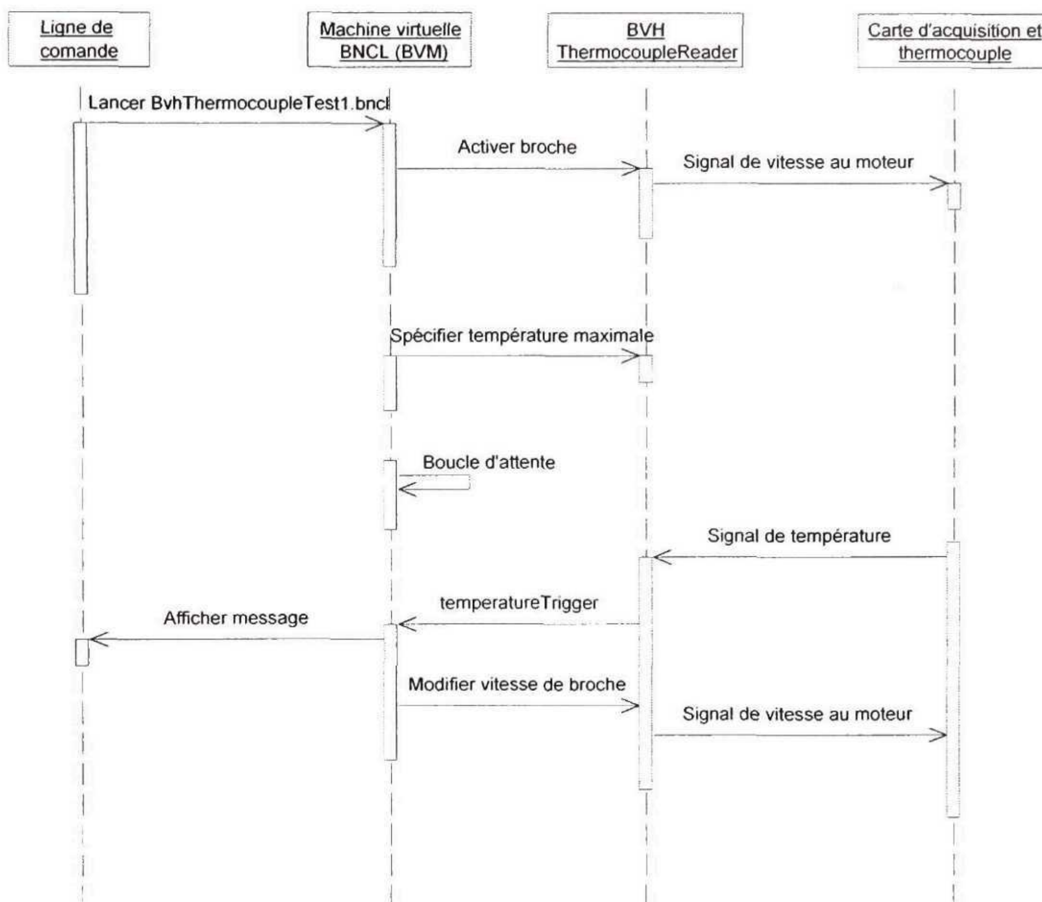

Figure 22 Diagramme de séquence du test d'extensibilité

3.9 Résumé

Plusieurs outils ont été conçus et développés dans le cadre du développement de l'architecture BNCL de référence. Ces outils étaient nécessaires pour passer des concepts à la réalité. Une machine virtuelle BNCL a été créée et est en mesure d'exécuter des modules BNCL exécutables. Différents BVH, pour effectuer des tests sur les fonctionnalités de l'architecture et sur la performance, ont également été définis. Un compilateur permet de générer un code source BNCL à partir de programmes écrits dans

le langage C. Ce langage est très répandu et très utilisé dans le monde de l'informatique. De plus, un assembleur BNCL permet de générer un module BNCL valide à partir d'une série d'instructions textuelles d'une source de données. Ce module exécutable peut être exécuté sur la machine virtuelle BNCL, ou être archivé dans le répertoire de modules créé à cette fin.

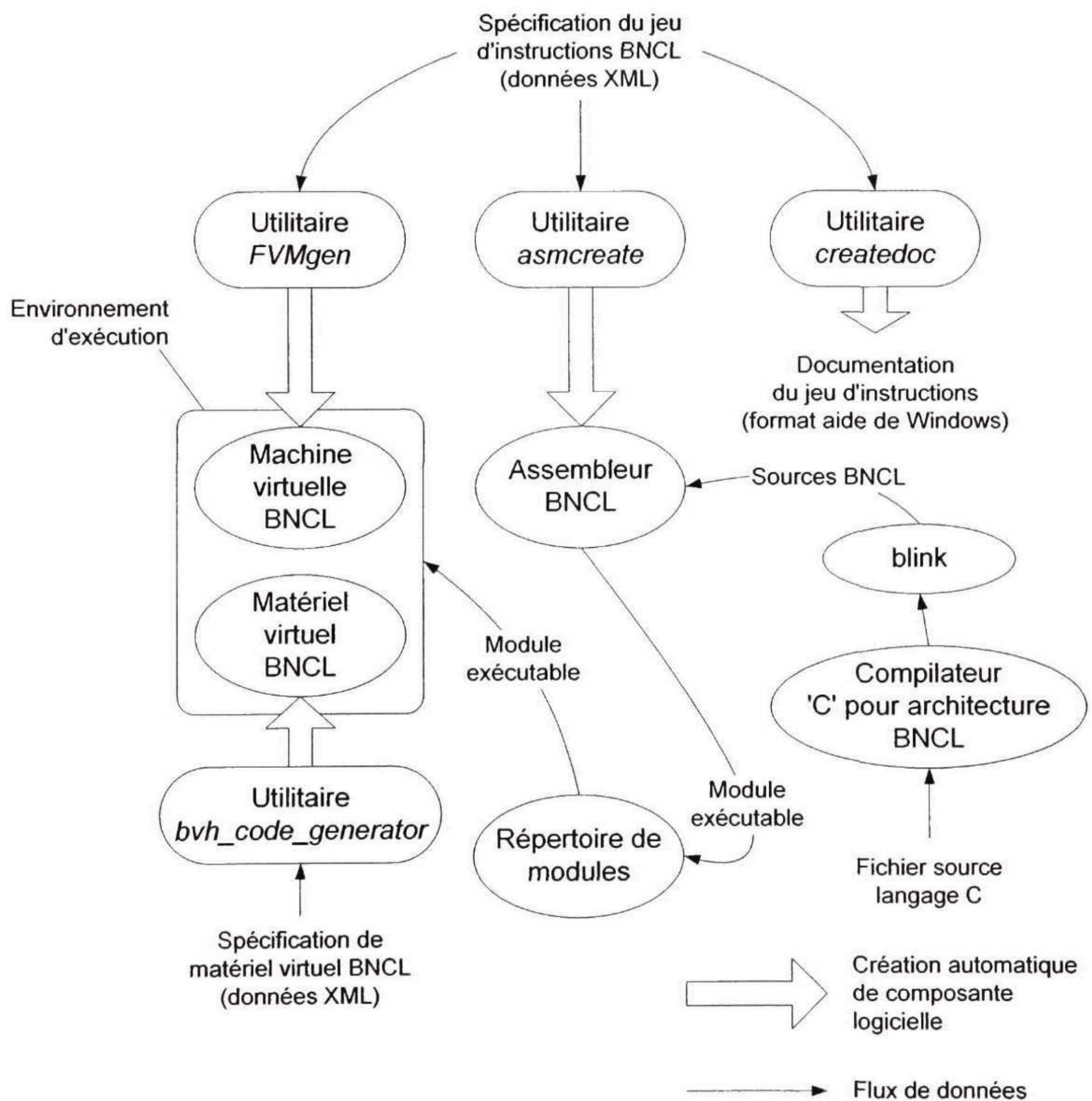

Figure 23 Résumé des composantes et utilitaires développés

Pour créer les outils et composantes de l'architecture, plusieurs techniques de génération automatique de code ont été développées. Ces techniques s'appuient fortement sur la syntaxe XML et permettent de diminuer le temps nécessaire à la maintenance et au changement des différentes composantes de l'architecture BNCL de référence. Plusieurs des outils conçus, et leurs interrelations, sont représentés à la Figure 23.

CHAPITRE 4

BÉNÉFICES ESCOMPTÉS DE L'ARCHITECTURE BNCL

L'architecture BNCL propose une nouvelle manière de faire en matière de contrôle de machine-outil à commande numérique. Cette architecture permet l'abstraction de la machine-outil tant du point de vue de l'environnement d'exécution que du matériel physique de cette machine (voir Chapitre 2). Certains bénéfices sont escomptés de la façon dont l'architecture BNCL aborde le contrôle de la machine-outil. Cette manière de faire pourrait permettre, en outre, de supporter les caractéristiques principales d'un contrôleur à architecture ouverte tel qu'énoncé au Chapitre 1.

Il a été constaté lors de la revue de littérature que l'architecture logicielle actuellement utilisée par les contrôleurs de machine-outil semble être désuète et inadéquate. Cette architecture ne permet pas la création d'un contrôleur ouvert à l'usager et certaines de ses caractéristiques font en sorte qu'elle empêche l'usager d'utiliser la machine-outil à son plein potentiel (voir section 1.1.3). L'architecture BNCL possède, du point de vue conceptuel, des qualités qui permettent de répondre à certains des problèmes rencontrés actuellement. Les différents avantages escomptés et les éléments nouveaux apportés par l'architecture BNCL au contrôle de machines-outils à commande numérique seront maintenant explorés.

4.1 Contrôleur à architecture ouverte

La conception de l'architecture BNCL s'est appuyée sur quatre caractéristiques souhaitées d'une architecture logicielle dans un contexte d'ouverture du contrôleur (voir section 1.2). Ces caractéristiques sont la portabilité, l'extensibilité, la réactivité aux

événements et le support multi langage. Chacune de ces caractéristiques offrent des avantages, énumérés précédemment, qui seront maintenant explicités.

4.1.1 La portabilité logicielle

Que ce soit en informatique ou dans n'importe quel autre domaine nécessitant un certain effort de programmation, la portabilité logicielle est souhaitable (voir section 1.2.1). En effet, une multitude de systèmes différents existent sur le marché, chacun possédant ses caractéristiques propres. Une architecture offrant la portabilité des programmes permet de capitaliser sur le développement réalisé sur un système en l'utilisant sur un autre. Cette utilisation multiple du code des programmes entre différents systèmes permet de réduire les coûts.

Traditionnellement, la portabilité logicielle s'est située au niveau des sources des programmes et non au niveau des modules exécutables compilés. C'est donc le code source qui est distribué entre les systèmes et non les modules exécutables. Ce code source est ensuite compilé sur chacun des systèmes. Cette portabilité n'a toutefois jamais été aisée. Même si des langages comme le C existent, où la syntaxe et les bibliothèques de fonctions standards sont les mêmes pour tous les systèmes, des problèmes surviennent lors de la compilation. La création de code portable au niveau des sources du programme est un processus long et difficile.

Par l'abstraction que fait l'architecture BNCL de l'environnement d'exécution sur lequel elle s'appuie, il est possible de cacher complètement les détails du système d'exploitation et des logiciels installés sur un contrôleur. Dans cette architecture, le programmeur écrit des programmes qui ciblent non pas un système d'exploitation en particulier, mais la BVM. Cette machine virtuelle fait le lien avec les particularités de l'environnement d'exécution sur laquelle elle est utilisée. Par exemple, si le système

d'exploitation présent est Windows CE[6], alors la gestion de l'interface usager sera différente que dans le cas où le système d'exploitation est Linux ou un système d'exploitation interne à une entreprise. La Figure 24 schématise l'abstraction que fait la machine virtuelle BNCL de l'environnement d'exécution présent sur le contrôleur.

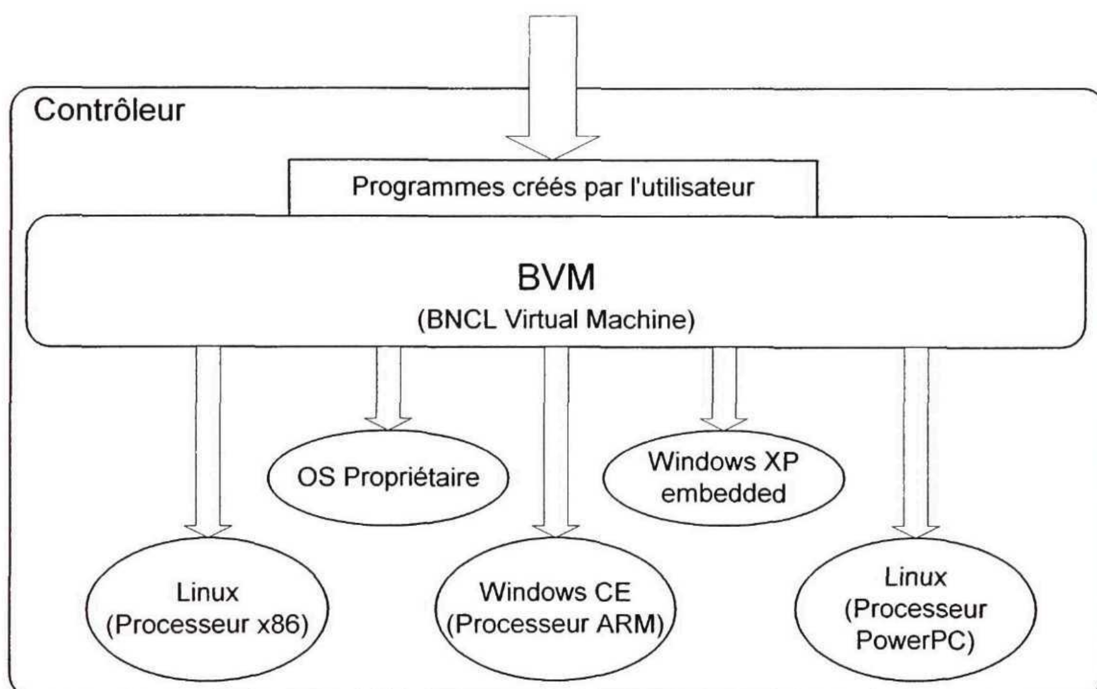

Figure 24 Masquage par la BVM des particularités logicielles du contrôleur

Parce que l'environnement d'exécution est caché par la BVM, le programmeur ne sait pas sur quel système informatique il se trouve. Tout est programmé en fonction de la BVM. De façon plus concrète, pour le programmeur il n'existe pas d'autre système d'exploitation que la BVM. Son code pourrait être utilisé sur un contrôleur de machine-outil ayant Linux comme système d'exploitation ou bien sur un autre ayant Windows NT. La BVM pourrait même ne pas se trouver sur un contrôleur de machine-outil, mais sur un ordinateur effectuant des simulations. Ceci montre un avantage de l'approche

[6] Version compacte et temps réel de Windows.

prise par l'architecture BNCL, soit que les programmes sont non seulement portables d'un contrôleur de machine à l'autre, mais que ces programmes peuvent être exécutés sur n'importe quel système permettant l'implémentation d'une BVM. Cet avantage ouvre de multiples possibilités au niveau de la simulation, des tests des programmes ainsi que du développement logiciel.

De plus, par l'abstraction qu'elle fait du matériel physique de la machine-outil, l'architecture BNCL cache à l'usager les détails des accessoires et des fonctionnalités présentes sur chaque machine-outil. De cette façon, un programme de coupe fait pour une machine-outil est portable vers une autre machine. Le programme créé par l'usager voit toujours la même machine et c'est l'architecture BNCL qui fait les ajustements nécessaires selon la véritable machine-outil sur laquelle il s'exécute. La Figure 25 schématise l'abstraction des composantes et accessoires physiques de la machine-outil. Sur cette figure, il est montré que des machines ayant des capacités d'usinage semblables, ici trois axes de mouvement, sont accessibles de façons identiques à partir du BVH.

Finalement, l'abstraction de l'environnement d'exécution que procure la BVM donne la liberté au manufacturier du contrôleur d'utiliser le système d'exploitation ainsi que les logiciels qu'il désire dans la création de son produit. À l'opposé, la standardisation autour d'un seul langage et d'un seul système d'exploitation oblige ce même manufacturier à suivre le standard et brime ainsi sa liberté de choix. Il en va de même pour l'abstraction des composantes et des accessoires de la machine-outil. Les protocoles de communication avec ces composantes et les détails de leur connexion électrique sont laissés à la discrétion du manufacturier. Celui-ci dispose alors de toute la marge de manoeuvre voulue pour concevoir la machine-outil, ainsi que l'interface avec le contrôleur, selon ses objectifs.

4.1.2 L'extensibilité matérielle

Tel que déjà mentionné, l'extensibilité de la machine-outil, soit l'ajout de composantes par l'usager, est un argument central en faveur des contrôleurs à architecture ouverte (voir section 1.2.2). Dans un tel contexte, la machine devient un outil dynamique qui peut évoluer dans le temps au gré des besoins de l'entreprise qui l'exploite. La manière dont l'architecture BNCL abstrait la machine-outil permet d'y ajouter des composantes et de les utiliser par la suite dans les programmes créés par l'usager. La Figure 25 montre l'abstraction faite par le BVH de la machine sous-jacente. Sur cette figure, une des machines est une fraiseuse verticale 3 axes, modifiée. Cette machine pourrait, par exemple, avoir été modifiée pour y ajouter un capteur de température sur la broche.

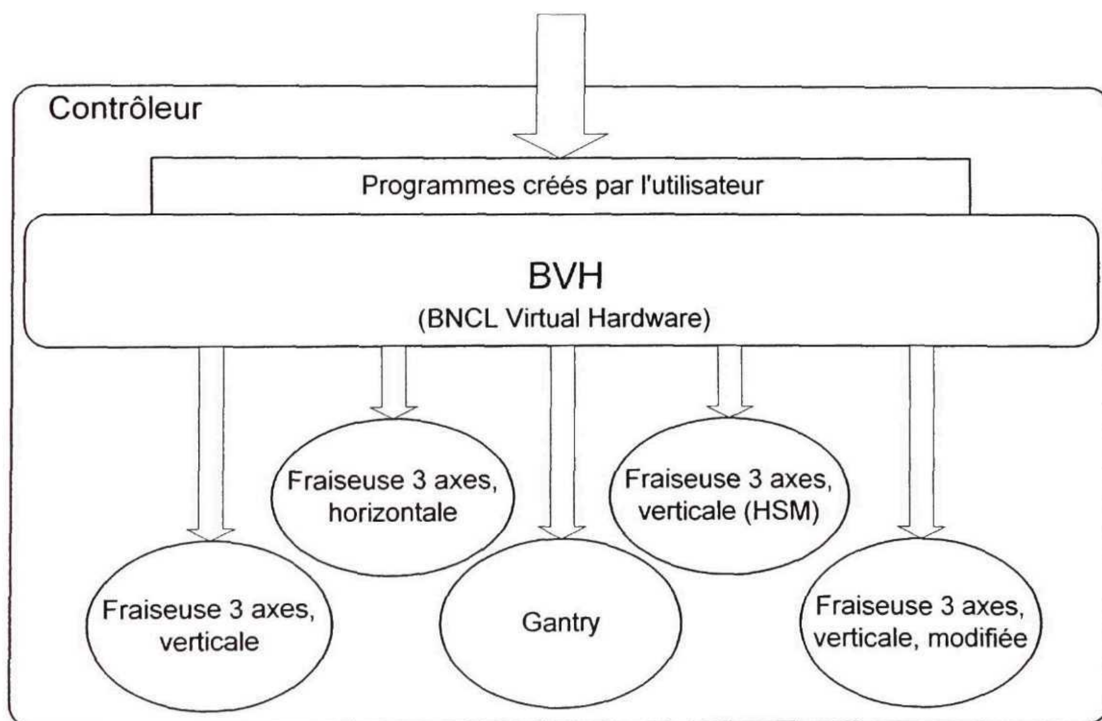

Figure 25 Masquage par le BVH des particularités matérielles de la machine-outil

Le mécanisme qui permet la portabilité logicielle entre les machines permet également l'extensibilité de celles-ci. Le BVH propose en effet deux groupes de ports pour l'accès à la machine-outil : un groupe standard partagé par toutes les machines de même type, et un local à la machine où les extensions sont visibles. Cette façon de faire permet de simplifier l'accès aux composantes ajoutées, tout en permettant une certaine portabilité de ces extensions d'une machine-outil à l'autre. L'explication détaillée du concept de groupes de ports a été vue à la section 2.3.

4.1.3 La réactivité aux événements

La réactivité aux événements est le fait de permettre à l'usager de traiter des événements dans ses programmes. L'architecture BNCL permet à l'usager de lier des routines à une multitude d'événements pouvant se produire sur la machine-outil. Par exemple, l'usager pourrait décider la façon dont la machine doit réagir à un arrêt d'urgence. Il pourrait également provoquer l'exécution de son propre code lorsqu'un capteur ajouté à la table de la machine est activé. Ce mécanisme de réactivité aux événements est au coeur du principe permettant d'adapter l'usinage aux conditions rencontrées. L'architecture BNCL simplifie le traitement des événements en permettant au BVH de signaler ces conditions à la BVM pour lui permettre d'exécuter les routines appropriées. Encore une fois, le mécanisme mis de l'avant est indépendant de la manière dont l'usager doit interfacer ses modifications avec le contrôleur. Tous ces détails sont entièrement cachés et la seule partie visible pour les programmes est le BVH. Ceci procure l'avantage de simplifier le développement logiciel relié à la réaction aux événements captés sur la machine-outil.

4.1.4 Le support multi langage

Comme il a été présenté au Chapitre 1, le support de plusieurs langages de programmation est une caractéristique bénéfique pour l'architecture logicielle d'un contrôleur de machine-outil (voir section 1.2.4). Le support multi langage donne à l'usager la liberté de choisir ses outils logiciels. Cette liberté lui permet de sélectionner le langage avec lequel il se sent le plus à l'aise et qui répond le mieux à ses besoins. En effet, un langage peut être bien adapté pour une tâche, mais l'être moins pour une autre ; un langage est conçu dans une certaine optique et ne peut couvrir tous les besoins d'un environnement industriel de production. Par exemple, le langage APT est tout à fait approprié pour la création d'un programme de coupe, mais serait peut-être moins efficace dans la création d'une application de gestion d'une banque d'outils. C'est pour cette raison que la standardisation autour d'un engin d'exécution comme la machine virtuelle BNCL, dont le jeu d'instructions n'est relié à aucun langage de programmation, permet le support d'autant de langages que nécessaire. Le Tableau I énumère des tâches reliées au développement logiciel sur les contrôleurs de machine-outil à commande numérique et quelques langages qui pourraient être utilisés dans chacun des cas.

La manière choisie pour aborder le problème du support de multiples langages de programmation assure que les langages sont libres de posséder toutes les fonctionnalités qu'ils désirent, sans pour autant nécessiter de support explicite de ces fonctionnalités par le jeu d'instructions et la machine virtuelle. De nouveaux langages pourront être créés dans le futur et n'entreront pas en conflit avec les langages existants puisque les détails internes de fonctionnement de chacun des langages de programmation sont cachés entre les modules. L'avantage principal de cette manière de faire est que la grande majorité des langages de programmation peuvent être utilisés sur l'architecture BNCL. Cela permet une grande liberté de choix et une excellente flexibilité en ce qui a trait aux outils logiciels pouvant être utilisés. Ce n'est pas le cas, par exemple, de la machine virtuelle

Java qui, à toutes fins pratiques, ne permet de supporter efficacement qu'un seul langage de programmation.

Tableau I

Exemples de langages et de leurs possibles utilisations

Tâche	Langages possibles
Programme de coupe	APT, Codes-G, OpenG
Gestionnaire d'événements	C, Forth, OpenG, Assembleur BNCL
Application de monitoring	C, C++
Implémentation de l'algorithme d'une nouvelle trajectoire	C, Fortran
Interface usager	VB, C++
Gestionnaire d'accès à une composante ajoutée par l'usager sur la machine-outil	C

4.2 Améliorations apportées par rapport au système actuel

L'architecture logicielle présentement utilisée sur les contrôleurs présente certains problèmes et certaines lacunes qui limitent l'étendue de l'utilisation d'une machine-outil (voir section 1.1.3). Ces lacunes sont liées, entre autre, à l'utilisation d'un langage de

contrôle désuet dans le contexte actuel d'ouverture des contrôleurs de machines-outils. Quelques problèmes déjà identifiés sont :

a. l'absence de librairies de procédures ;
b. l'absence de structures de langage de programmation ;
c. l'absence de performances adéquates dans la création d'applications complexes d'usinage.

L'architecture BNCL corrige ou améliore la situation concernant chacun de ces éléments. Les apports de l'architecture BNCL seront maintenant présentés.

4.2.1 Librairies de procédures

De façon naturelle, l'architecture BNCL supporte les librairies de procédures. Un module BNCL est en soit une librairie de procédures. Ce module expose des points d'entrée vers des fonctions qui peuvent être utilisées par d'autres modules. Cette façon de faire permet une certaine extensibilité du langage. Un nouvel algorithme de calcul peut être implémenté dans un langage quelconque et par la suite être inclu dans un module. Ce module peut être utilisé par tout programme nécessitant ce nouvel algorithme. Un exemple pourrait être l'implémentation d'un nouveau cycle d'usinage. Ce cycle pourrait concerner l'usinage d'une came. Le cycle permettrait l'usinage d'un nombre indéterminé de segments, chacun étant paramétré d'une façon différente. Avec l'architecture BNCL, de tels cycles sont accessibles en autant qu'ils soient inclus dans un module. Tous ces développements sont également portables d'une machine à l'autre. Le cycle de came serait programmé une fois, et pourrait être utilisé sur plusieurs machines différentes (voir section sur la portabilité logicielle en 4.1.1).

4.2.2 Structures de langage de programmation

Il a été constaté, lors de la revue de littérature, que le langage de contrôle actuellement utilisé sur les machines-outils, les Codes-G, ne possède pas les structures syntaxiques des langages de programmation évolués (voir section 1.1.3). Cela a pour effet de complexifier le processus d'écriture de programmes évolués. De plus, il est pratiquement impossible d'utiliser les principes de la programmation structurée avec ce langage. Les programmes en Codes-G sont donc peu lisibles et difficiles à maintenir.

Dans l'architecture BNCL, il n'y a pas de langage de programmation à proprement parler. Ceci est d'ailleurs un de ses avantages puisque l'architecture n'est pas liée à un seul langage de programmation. Il y a bien sûr le jeu d'instructions BNCL, qui est parfois appelé langage BNCL, mais ce langage ne représente que des instructions élémentaires ressemblant au langage de tout microprocesseur. Le support des structures de langages de programmation est réalisé à l'extérieur du jeu d'instructions par les différents langages de programmation dont les compilateurs peuvent cibler l'architecture BNCL. Ceci procure l'avantage de permettre le support de structures de programmation de styles très variés, tout en permettant à ces langages hétérogènes de communiquer entre eux.

4.2.3 Performances

Comme il a été présenté dans la revue de littérature, l'architecture logicielle actuellement utilisée sur les contrôleurs de machines-outils à commande numérique ne livre pas des performances adéquates pour la réalisation d'application d'usinage complexes (voir 1.1.3). Le goulot d'étranglement de cette architecture est en grande partie situé au niveau de l'interprétation des Codes-G. Cette interprétation lente des instructions d'usinage peut occasionner des délais dans la génération des signaux de

contrôle vers la machine-outil. Par exemple, lorsque des calculs mathématiques sont présents, la vitesse de déplacement de l'outil peut être ralentie à des niveaux parfois très bas. Même lorsque les positions ont déjà été calculées à l'aide d'un post-processeur, l'extraction de ces positions des Codes-G par l'interpréteur peut ralentir tout le processus de contrôle (voir sections 1.1.3 et 6.3).

L'architecture BNCL permet, en contrepartie, des performances nettement meilleures. L'utilisation d'une machine virtuelle et d'une couche d'abstraction du matériel physique de la machine-outil permet un accès plus direct aux composantes de la machine. Ceci n'est pas le cas des Codes-G utilisés comme langage de contrôle de base. En plus de fournir l'infrastructure pour la réalisation d'un contrôleur à architecture ouverte, l'architecture BNCL offre également les performances nécessaires pour l'implémentation d'algorithmes mathématiques complexes. Pour les Codes-G, cette complexité n'est la plupart du temps pas possible. Toute la question de la performance de l'architecture BNCL, ainsi que des résultats des tests réalisés, sont présentés au Chapitre 6, portant sur la validation.

4.3 Résumé

L'architecture BNCL présente des qualités importantes et offre des améliorations par rapport à l'architecture logicielle actuellement utilisée avec les contrôleurs de machines-outils. Dans un premier temps, l'architecture BNCL possède des caractéristiques qui permettent la réalisation d'un contrôleur à architecture ouverte. Ce type de contrôleur demande des caractéristiques de portabilité, d'extensibilité, de réactivité aux événements et de support multi langage.

L'architecture BNCL offre également des améliorations par rapport aux outils de programmation actuellement utilisés avec les contrôleurs. Les principales lacunes

rencontrées au niveau des Codes-G, soit l'absence de librairie de procédures, la faiblesse des structures de langage de programmation ainsi que les performances, sont toutes comblées par l'architecture BNCL. Cette architecture permet en effet la création de librairies de procédures, allant de l'algorithme mathématique de génération de parcours d'outil jusqu'au cycle d'usinage complexe, en passant par la gestion de l'interface usager. Le support multi langage de l'architecture BNCL assure, de plus, que des structures valables de langages de programmation pourront être utilisées. Finalement, les performances escomptées de l'utilisation de l'architecture sont prévues être supérieures à celles rencontrées lors de l'utilisation des Codes-G.

CHAPITRE 5

DÉMARCHES D'INTÉGRATION AUX PROCESSUS EXISTANTS

Le présent chapitre décrit des méthodes permettant d'intégrer l'architecture BNCL aux processus existants en entreprise. Cette intégration vise à faciliter l'acception de l'architecture BNCL et la création d'une base d'utilisateurs. L'adaptation aux processus existants est essentielle puisque ceux-ci sont utilisés à grande échelle et ne peuvent être remplacés du jour au lendemain. Ce chapitre présente donc des démarches qui peuvent être entamées pour réaliser efficacement cette intégration.

De nombreux outils existent aujourd'hui pour programmer les machines-outils à commande numérique. Ces outils ont parfois pour origine des travaux effectués dans les années cinquante, comme les Codes-G ou le langage APT. Ceux-ci n'ont jamais été remplacés puisque la compatibilité d'une nouvelle génération d'outils avec la précédente a toujours été une nécessité. De fait, des substituts n'ont jamais véritablement été utilisés à grande échelle. Il en résulte que ces outils sont aujourd'hui essentiellement les mêmes que ceux utilisés au tout début de l'usinage à commande numérique. Par ailleurs, des organismes internationaux travaillent actuellement à l'élaboration de nouvelles méthodes de programmation concernant les machines-outils à commande numérique. Le standard STEP-NC est un exemple d'effort majeur en ce sens. Ce standard fait l'objet d'une étude intensive et sera plus tard utilisé dans de nombreux systèmes.

La Figure 26 montre le flux de transformation de la définition du produit en instructions de programmation des machines-outils à commande numérique, tel qu'il existe actuellement. Les principaux outils y sont présentés soit : le logiciel de fabrication assistée par ordinateur (FAO), le post-processeur et l'interpréteur de Codes-G du contrôleur. Le format des données à chacune des étapes y est indiqué : CL-APT entre le logiciel FAO et le post-processeur et Codes-G entre le post-processeur et la machine.

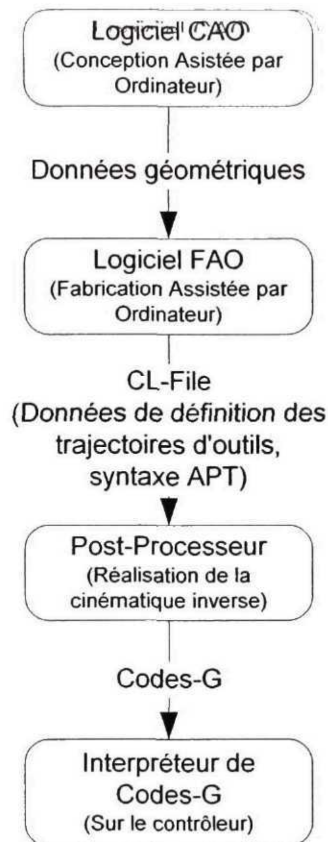

Figure 26 Transformation du modèle produit en instructions de programmation

Le constat peut être fait que tout nouvel outil ne peut ignorer les réalités de l'usinage par commande numérique. Ne pas tenir compte du processus actuel de programmation des machines-outils, et des changements anticipés, pourrait handicaper l'acceptation de ce nouvel outil. Le succès potentiel d'un nouveau développement dans la programmation des machines-outils passe donc par une intégration aux processus actuellement employé en industrie. Cette intégration se veut temporaire. Le but consiste à favoriser l'acceptation initiale et de promouvoir une plus grande utilisation de la nouvelle architecture par les usagers. Quatre éléments ont été ciblés dans le plan d'intégration de l'architecture BNCL : le langage APT, les logiciels de FAO, les Codes-G et le standard ISO14649 STEP-NC.

5.1 Langage APT et logiciels de FAO

Au tout début de la commande numérique, alors que les logiciels de FAO graphique étaient presque inexistants, le langage APT permettait de faciliter la programmation des machines-outils à commande numérique. Le programmeur utilise des mots-clés (APT) pour décrire la pièce à usiner, plutôt que de programmer directement la machine en Codes-G. Un compilateur permet par la suite de transformer le code APT de description de l'usinage, en code APT décrivant les positions du centre de l'outil. Un post-processeur permet finalement d'obtenir les Codes-G pour un contrôleur spécifique. Le langage APT fournit plusieurs outils de calculs géométriques et de génération de parcours d'outils. Les Codes-G n'offrent aucune chose comparable. Lorsque les logiciels de FAO ont été utilisés à grande échelle, la syntaxe APT a été conservée comme format de données intermédiaire entre le logiciel et le post-processeur. Cette utilisation des outils existant a facilité l'intégration des logiciels de FAO dans le processus de programmation qui était utilisé à l'époque.

Le code pourvu d'une syntaxe APT provient aujourd'hui de deux sources : soit d'un programme écrit à la main par un programmeur, soit d'un logiciel de FAO. La programmation manuelle de l'usinage de pièces est de moins en moins utilisée. Toutefois, elle l'est encore en certains endroits où de nombreux programmes existants doivent être supportés. Les logiciels de FAO sont aujourd'hui la plus grande source de code en syntaxe APT retrouvée dans le processus de programmation des machines-outils à commande numérique. Une stratégie d'intégration doit tenir compte de ces deux sources.

5.1.1 Méthodes d'intégration en amont

L'intégration, reposant sur le langage APT et les logiciels de FAO, de la solution proposée peut se faire selon deux méthodes. Premièrement, il peut être fait en sorte que l'architecture BNCL comprenne le langage APT directement. Ce code en syntaxe APT peut provenir soit d'un logiciel de FAO, soit d'un programme écrit à la main. Une deuxième méthode consiste à obtenir des instructions BNCL, plutôt qu'une syntaxe APT, du système de FAO.

La Figure 27 montre les deux méthodes d'intégration en amont qui sont envisagées et l'effet de chacune sur le flux de transformation des données. La solution 1 permet d'utiliser le code en syntaxe APT en provenance de logiciels de FAO actuels. La solution 2 permet d'utiliser du code APT écrit à la main, représenté sur la figure par la composante *éditeur de texte*. La solution 3 représente la méthode où les logiciels de FAO sont adaptés pour produire des instructions BNCL. Pour les trois solutions, le contrôleur est utilisé tel quel et n'est pas modifié.

Les méthodes d'intégration présentées permettent d'inclure l'architecture BNCL au processus actuel de programmation des machines-outils sans demander de changements radicaux aux façons de faire en vigueur. De fait, avec la première méthode d'intégration, le programmeur verra l'inclusion de l'architecture BNCL comme un nouveau type de post-processeur avec lequel il peut travailler de façon habituelle. L'intégration à ce niveau permet de capitaliser sur les logiciels de FAO existants. La deuxième méthode peut être utilisée lorsqu'une intégration plus poussée est désirée. Elle nécessite des changements aux logiciels de FAO existants ou le développement de modules s'intégrant directement à ces logiciels dans le but de leur ajouter des fonctionnalités.

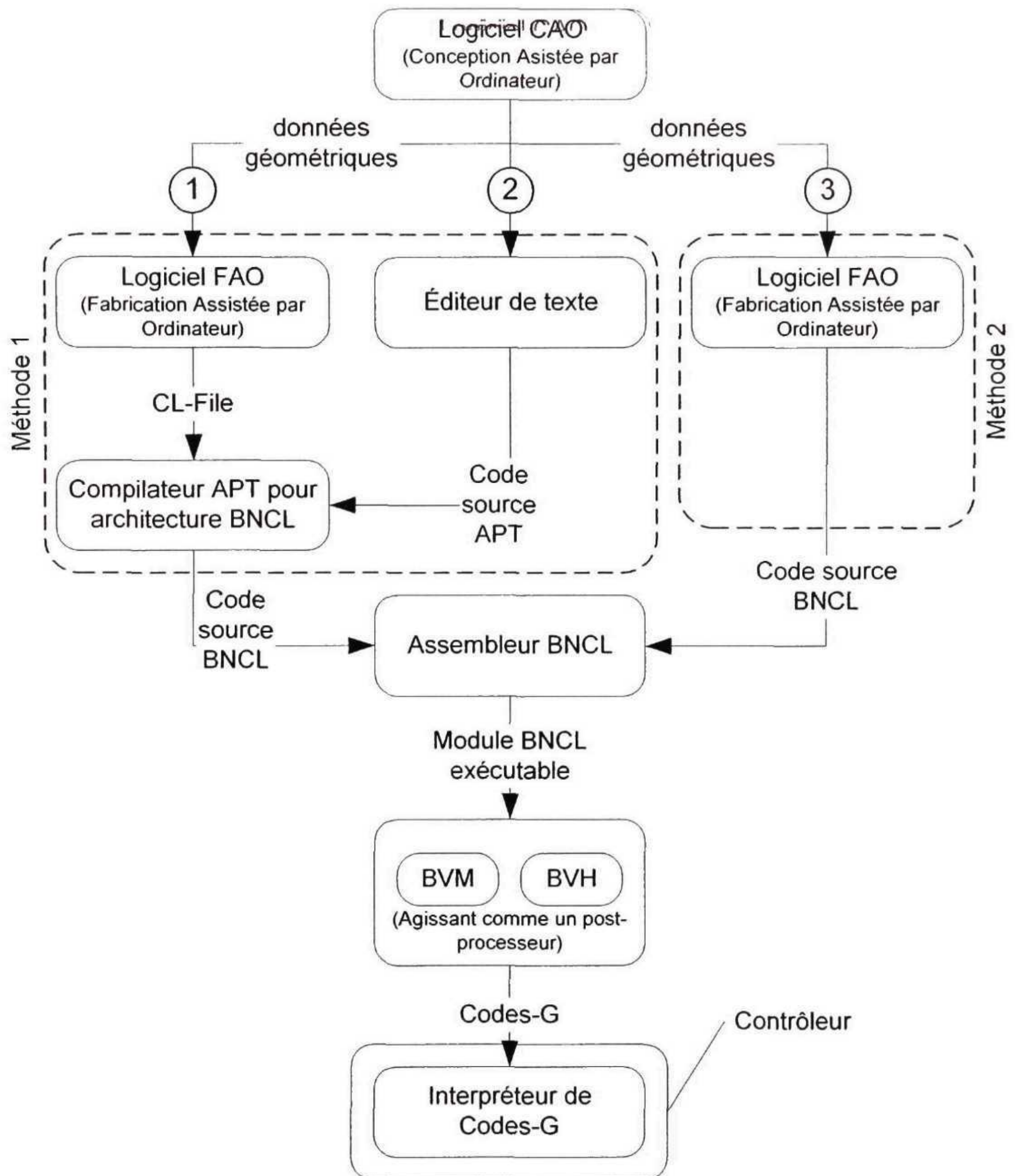

Figure 27 Intégration de l'architecture BNCL au processus de programmation

5.2 Codes-G

La revue de littérature effectuée a exposé les lacunes des Codes-G tels qu'ils sont utilisés pour la programmation des machines-outils à commande numérique (voir section 1.1.3). Malgré ces lacunes, les Codes-G sont utilisés dans presque la totalité des contrôleurs de machine. Une intégration de l'architecture BNCL au processus actuel de programmation des machines-outils nécessite donc de tenir compte de l'étendue de l'utilisation de ces codes.

5.2.1 Méthodes d'intégration en aval

La manière d'intégrer l'architecture BNCL en tenant compte des Codes-G est semblable à celle utilisée dans le cas du langage APT. Dans un premier temps, il est possible d'utiliser les logiciels de FAO tels quels et de travailler au niveau des sources APT/CL-APT. Ensuite, après modifications des logiciels FAO utilisés, il est possible de générer des instructions BNCL qui pourront être exécutées par une machine virtuelle BNCL. Ces deux méthodes ont été vues à la section 5.1.1. Elle sont présentées à la Figure 27 en tant que méthode 1 et méthode 2. Avec ces deux méthodes, ce qui est appelé post-processeur (voir Figure 26) est remplacé par une machine virtuelle BNCL (BVM) et un matériel virtuel BNCL (BVH) (voir Figure 27). Ces deux composantes clés de l'architecture BNCL sont ici utilisées à l'extérieur du contrôleur et permettent de générer les Codes-G spécifiques à une machine-outil donnée. Un BVH est alors associé à chacune des machines devant être supportées. C'est cette composante qui génère les Codes-G pour la machine qu'elle abstrait. La Figure 28 montre l'utilisation de trois BVH permettant la génération de Codes-G pour autant de machines. Il est important de rappeler qu'un BVH permet l'abstraction du matériel physique qu'il contrôle. Cette composante est donc en mesure d'abstraire un générateur de Codes-G au même titre qu'elle est en mesure d'abstraire la véritable machine-outil. Dans ce scénario d'intégration, la machine

virtuelle BNCL exécute un programme. Ce programme envoie des données aux ports contrôlés par un BVH. Le BVH répond à ces envois de données en effectuant des transformations et en générant les Codes-G appropriés. Lorsque l'usager désire générer les Codes-G pour une autre machine, il change de BVH.

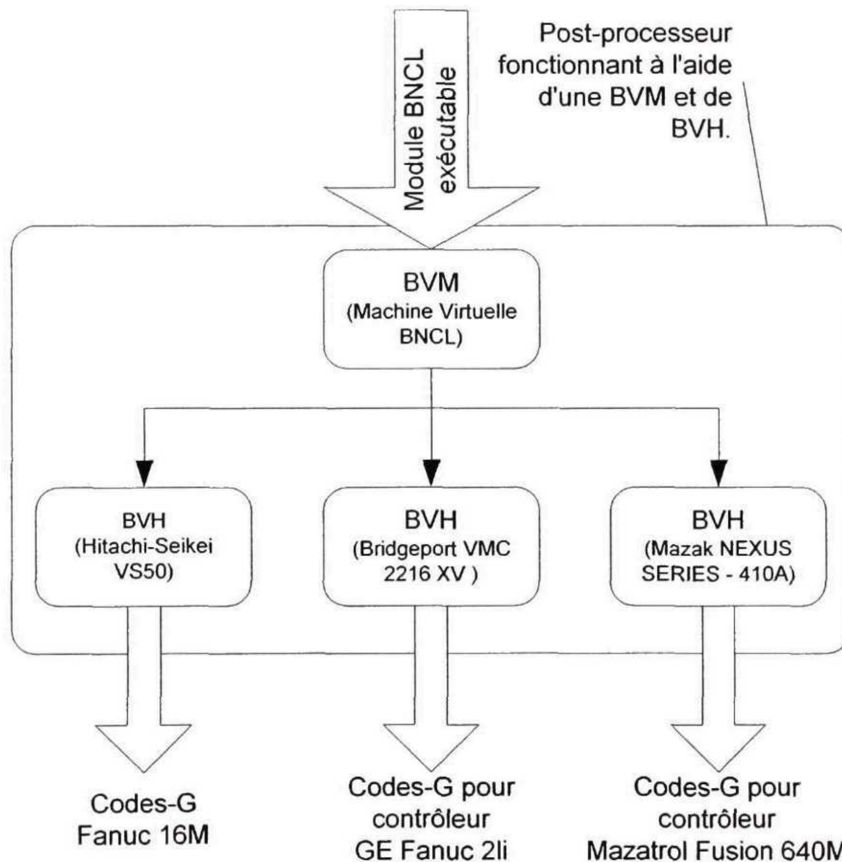

Figure 28 Génération de Codes-G pour différentes machines-outils

5.2.2 Conversion des Codes-G

On peut envisager utiliser l'architecture BNCL pour convertir des programmes Codes-G d'une variante à l'autre et donc d'un contrôleur à l'autre. Par exemple, il serait possible de transformer les Codes-G d'un contrôleur Mazatrol pour qu'ils puissent être exécutés

sur un contrôleur Fanuc 2li. La Figure 29 montre comment cette transformation est possible. Tout d'abord, les Codes-G des différentes machines-outils sont compilés pour former un code source BNCL. Ce code source est par la suite assemblé à l'aide de l'assembleur BNCL et exécuté sur une machine virtuelle BNCL (BVM). Un BVH pour la machine-outil et le contrôleur visé est utilisé pour la conversion finale. Cette méthode est une autre façon d'intégrer l'architecture BNCL au processus actuel de programmation des machines-outils à commande numérique.

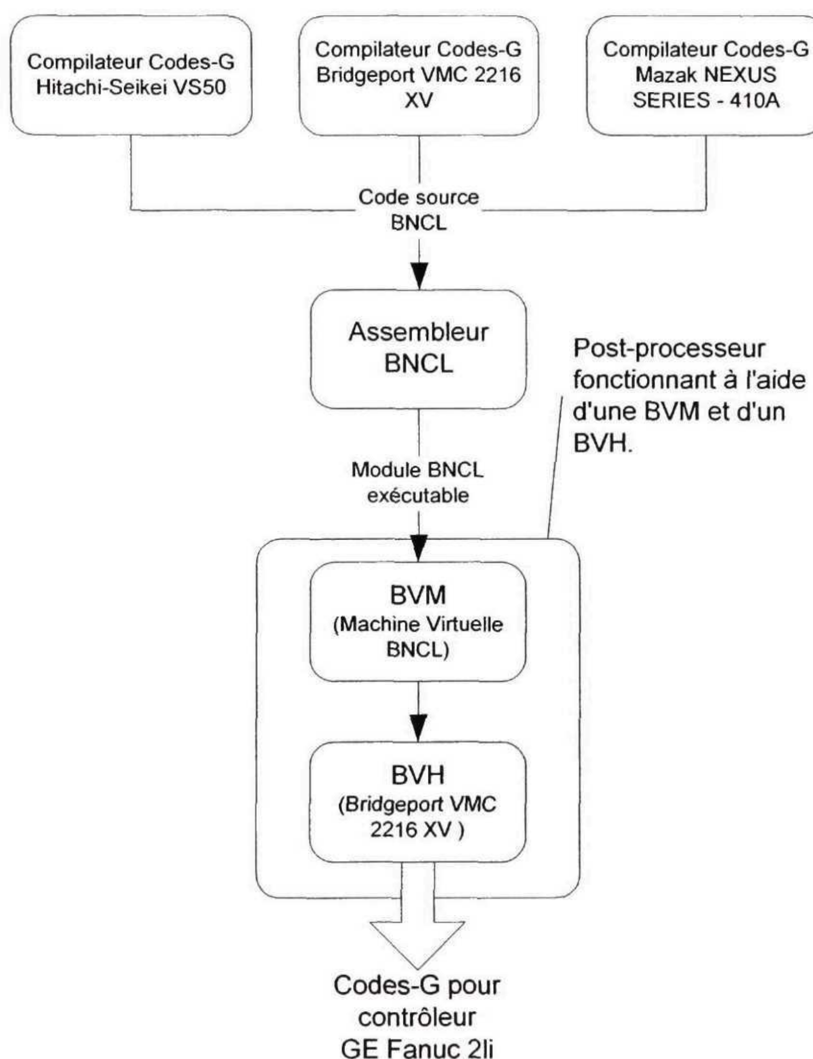

Figure 29 Transformation de Codes-G entre différents contrôleurs

5.3 Standard ISO14649 STEP-NC

Les lacunes du processus actuellement utilisé pour la programmation des machines-outils à commande numérique ont suscité de nouvelles façons de faire. Une de ces façons de faire est élaborée autour du standard ISO14649, ou STEP-NC. Ce standard reprend la syntaxe de STEP, qui se veut un standard de représentation et d'échange de modèles produit, et en étend les caractéristiques pour y inclure les informations d'usinage [24]. Les données STEP-NC sont destinées à être interprétées directement par un contrôleur de machine.

Au moment de la rédaction de ce mémoire, le standard ISO14649 en était à la phase d'approbation. Plusieurs mois peuvent s'écouler avant que l'acceptation finale ne soit obtenue et que le standard ne soit publié. Certains projets sont cependant déjà en cours pour permettre aux machines-outils de la prochaine génération d'utiliser le format de données ISO14649. Quelques exemples sont les projets en cours à l'Université de technologie d'Aachen en Allemagne et ceux de l'Université Pohang en Corée du sud [53].

Le standard ISO14649 est destiné à prendre une importance grandissante dans le contrôle des machines-outils puisqu'il est développé par une organisation internationale à laquelle participent de nombreux pays et plusieurs acteurs clés de l'industrie. L'intégration de l'architecture BNCL au processus de programmation des machines-outils doit donc tenir compte de ce nouveau standard et de ses implications.

5.3.1 Méthodes d'intégration

L'architecture BNCL étant une couche d'abstraction très proche de la machine-outil, tout système de plus haut niveau peut être construit en utilisant cette architecture comme

base. Par exemple, l'architecture BNCL propose un compilateur pour le langage C. Ce compilateur, qui supporte donc un langage de haut niveau, est construit en utilisant l'architecture BNCL comme base. Le langage APT, pour lequel un compilateur est actuellement en cours de réalisation, en est un autre exemple. Tout langage de haut niveau peut être porté vers l'architecture BNCL, puisque celle-ci a été conçue pour supporter plusieurs langages de programmation différents. Cela est vrai pour un langage de programmation, mais l'est aussi pour un format de données comme le standard ISO14649 STEP-NC.

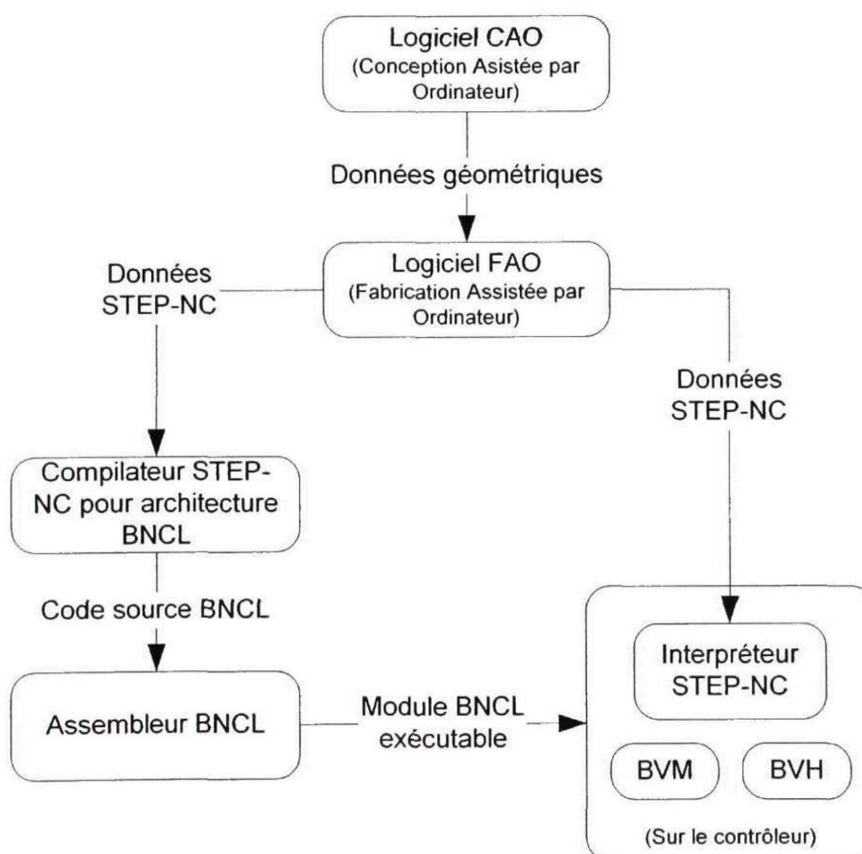

Figure 30 Intégration de l'architecture BNCL au standard STEP-NC

Un compilateur pour les données STEP-NC peut donc être conçu. Ce compilateur transformerait les données STEP-NC pour en faire des modules BNCL exécutables. Il serait également possible de concevoir un interpréteur de code STEP-NC et d'inclure cet

interpréteur dans un module BNCL. Ce module, utilisé sur un contrôleur de machine-outil à commande numérique, permettrait d'utiliser les données ISO14649 comme entrée de contrôle de la machine. La Figure 30 montre les différentes possibilités d'intégration de l'architecture BNCL au standard ISO14649 STEP-NC.

5.4 Résumé

Le processus de programmation des machines-outils comprend une série d'outils avec lesquels l'architecture BNCL doit initialement cohabiter pour favoriser son acceptation et l'élargissement de sa base d'utilisateurs. Cette acceptation initiale pourra par la suite faire place à l'utilisation pour laquelle cette architecture a été développée.

L'architecture BNCL, de par sa flexibilité d'utilisation, est capable de s'intégrer de plusieurs manières au processus actuel de programmation. L'intégration peut se faire en amont, en travaillant sur le produit des logiciels de FAO. Le langage APT est alors utilisé comme source d'entrée pour un compilateur produisant des instructions BNCL. Des modules peuvent également être ajoutés aux logiciels de FAO pour que ceux-ci génèrent des instructions BNCL. L'intégration peut également se faire en aval, où l'architecture BNCL agit comme un post-processeur ou comme un convertisseur de Codes-G. Finalement, il a été démontré que l'architecture BNCL peut s'intégrer aux nouveaux développements dans le domaine du contrôle des machines-outils à commande numérique. Par exemple, l'architecture BNCL peut être utilisée comme interpréteur de données STEP-NC sur le contrôleur. Toutes ces méthodes d'intégration peuvent être utilisées ensemble, ou séparément, selon les besoins et les objectifs de l'utilisateur.

En ayant la possibilité de s'intégrer au processus actuel de programmation des machines-outils à commande numérique, l'architecture BNCL favorise son acceptation

initiale dans un domaine où les changements sont généralement acceptés avec difficulté. Ces différentes méthodes d'intégration montrent, de plus, la flexibilité d'utilisation que l'architecture BNCL offre.

CHAPITRE 6

VALIDATION ET ANALYSE DES RÉSULTATS

L'architecture BNCL présente de nouveaux concepts et de nouvelles idées visant à changer les manières de faire dans le domaine du contrôle des machines-outils à commande numérique. Cette nouvelle architecture permet d'envisager des bénéfices en termes de flexibilité de programmation et de l'extension des fonctionnalités de la machine-outil. Ces bénéfices ont été discutés au Chapitre 4. Pour être en mesure de réaliser ces bénéfices, cette architecture doit cependant répondre réellement, et non uniquement conceptuellement, aux besoins pour lesquels elle a été conçue. Une validation des concepts fondateur de l'architecture BNCL est donc essentielle pour s'assurer que l'architecture fonctionne telle qu'imaginée et qu'elle apporte les bénéfices prévus.

La validation de l'architecture BNCL comprend trois volets : la validation de la performance logicielle de l'architecture, la validation du concept d'extensibilité propre aux contrôleurs à architecture ouverte et la validation de l'architecture du point de vue des performances et de la flexibilité d'utilisation dans un contexte de production. Cette dernière validation inclue une comparaison avec l'architecture logicielle actuellement utilisée dans l'industrie.

La validation des performances est essentielle puisqu'elle indique si l'architecture peut être utilisée dans un contexte réel de production. La validation porte sur les performances nécessaires dans un contexte de contrôle des machines-outils à commande numérique. De plus, puisque l'accent à travers le projet a été mis sur la réalisation de contrôleurs à architecture ouverte, la validation des concepts de ces contrôleurs est également essentielle. Finalement, la comparaison avec l'architecture logicielle actuellement utilisée en industrie permet de mettre en lumière les avantages de la

nouvelle proposition, tout en s'assurant de la validité de celle-ci. Toutes les simulations de l'architecture BNCL ont été effectuées sur un ordinateur de classe Pentium II cadencé à 450 MHz. Cet ordinateur peut être considéré bas de gamme.

6.1 Performance

Les tests de performance visent à s'assurer que l'architecture BNCL permet de répondre aux exigences de performance d'un environnement réel de production. Le but est de vérifier que l'architecture BNCL n'est pas la composante limitant les performances de la machine-outil. Les tests visent donc à déterminer la composante où se situe la limitation de performance lorsque l'architecture BNCL est utilisée. Le métrique utilisé est le nombre de positions d'outil pouvant être générées en une seconde. La vitesse avec laquelle l'outil peut suivre un parcours étant directement proportionnelle au nombre de positions que le contrôleur peut générer dans un certain laps de temps, le calcul de la performance de l'architecture BNCL dans ce domaine est important.

La Figure 31 montre l'architecture de base d'un contrôleur de machine-outil. L'interpréteur du programme NC est le module qui décode le programme de contrôle et qui envoie les positions désirées au module contrôlant les axes de la machine. Si cet interpréteur n'est pas en mesure de produire suffisamment de positions dans un laps de temps donné, le module de contrôle des axes n'aura plus de positions sur lesquelles travailler et devra soit diminuer la vitesse de l'outil, pour laisser la possibilité à l'interpréteur de fournir les positions, soit complètement arrêter le mouvement. Il est important de noter que plus le mouvement est complexe, plus le nombre de positions nécessaires dans un certain laps de temps est grand. En effet, un mouvement complexe demande la génération de points très rapprochés dans l'espace. La distance que l'outil doit parcourir entre les points est donc petite. L'interpréteur doit alors fournir des points à une cadence plus élevée. À l'opposé, un mouvement strictement linéaire sur une

grande distance ne nécessite que deux points. La Figure 32 illustre cette différence. Dans un mouvement linéaire pur, seulement deux points sont nécessaires. Par contre, dans un mouvement curviligne complexe (trajectoire du bas sur la figure), le changement fréquent de direction demande la génération de beaucoup plus de positions.

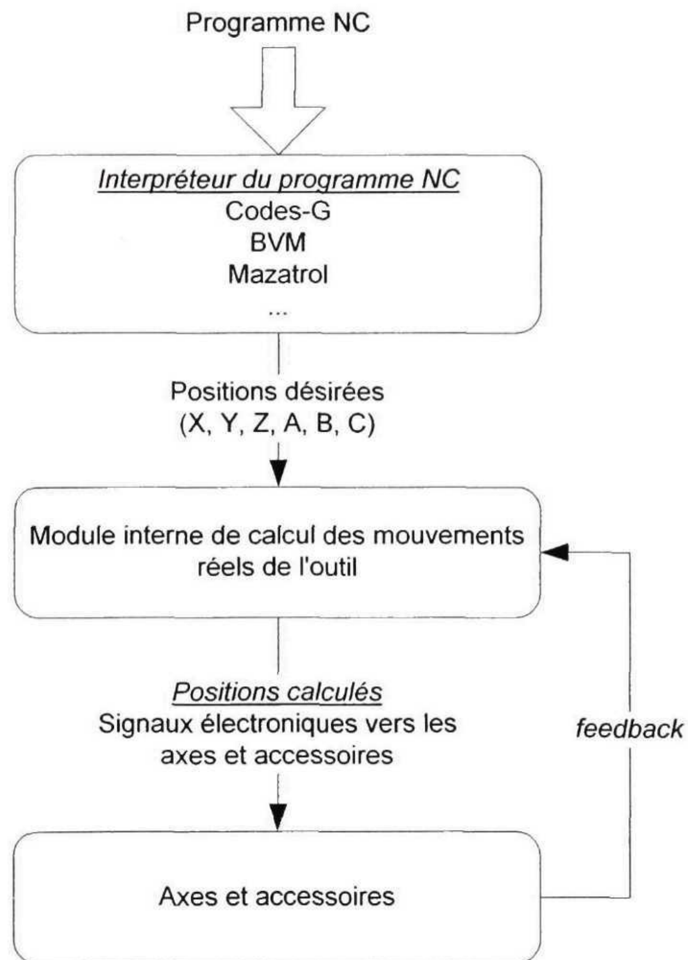

Figure 31 Flux d'information dans l'architecture actuelle

L'équation (1) permet de calculer le nombre de positions nécessaires par unité de temps en fonction de la vitesse d'avance de l'outil et de l'incrément de distance entre deux positions.

$$F = \frac{IPM}{(60 \cdot INC)} \qquad (1)$$

Dans cette équation, *F* représente le flux de positions nécessaires par seconde, *IPM* représente la vitesse d'avance de l'outil en pouces/minute et *INC* représente l'incrément de distance en pouce entre deux positions. Par exemple, pour un outil avançant à 1000 pouces/minute sur des mouvements d'une distance de 0.001 pouce, le nombre de positions nécessaires par seconde est de 16666. Si l'interpréteur n'est pas en mesure de fournir ce nombre de positions, le module de calcul des mouvements (Figure 31) devra ralentir la vitesse de l'outil pour compenser.

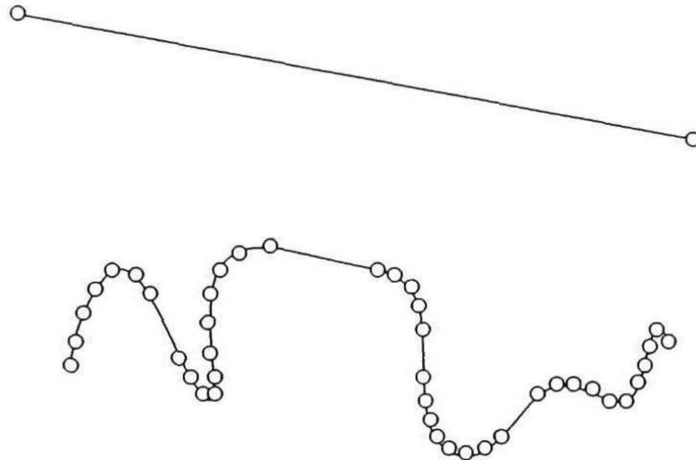

Figure 32 Nombre de points requis pour des trajectoires linéaires et curvilignes

6.1.1 Tests de performance réalisés

Les tests de performance de l'architecture BNCL tentent de déterminer, entre autre, la vitesse à laquelle la machine virtuelle BNCL (BVM) et le matériel virtuel BNCL (BVH) peuvent produire des positions d'outil. Les scénarios étudiés dans la validation de la performance du lien de communication des composantes de l'architecture BNCL sont montrés à la Figure 33. L'ensemble de l'architecture BNCL est présenté à la Figure 3 de

la section 2.1. Cinq scénarios sont proposés. Les trois premiers utilisent un pilote de périphérique pour simuler l'accès à l'électronique du contrôleur. Les deux derniers scénarios n'ont pas ce pilote. Ce pilote de périphérique a été conçu spécialement pour l'architecture BNCL. Il répond au standard des pilotes de périphérique de l'environnement Windows 2000 [54]. Les deuxième, troisième et cinquième scénarios utilisent un exécutable Windows au lieu d'un module exécutable BNCL. Cet exécutable Windows communique directement avec le BVH, ou le pilote de périphérique, et n'implique donc pas le temps d'interprétation des instructions BNCL par la BVM. Finalement, le troisième scénario contourne complètement l'architecture BNCL et communique directement avec le pilote de périphérique. Ceci permet de déterminer la performance maximale pouvant être atteinte, sans autre élément que la communication avec l'électronique du contrôleur. Pour les cinq scénarios, aucun point n'est calculé en temps réel. Seule la communication entre les différentes composantes est évaluée.

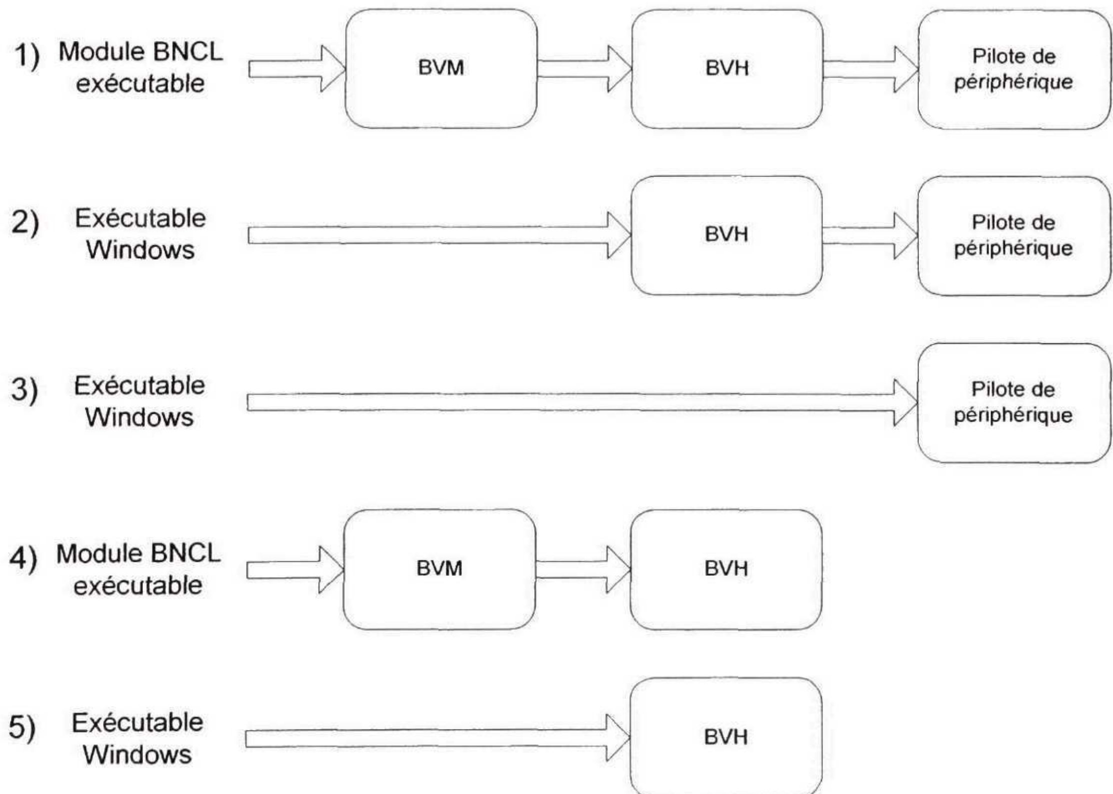

Figure 33 Scénarios de test de performance de l'architecture BNCL

Plusieurs résultats peuvent être déduits en combinant les valeurs obtenues pour chacun des cinq scénarios présentés à la Figure 33. La perte de performance associée à chacune des composantes de l'architecture BNCL peut ainsi être évaluée, tel que résumé au Tableau II.

Tableau II

Évaluation des pertes de performance de chaque composante

Scénarios comparés (voir Figure 33)			Résultat déduit
1 bvm/bvh/pilote	VS	2 bvh/pilote	Perte de performance due à la BVM
4 bvm/bvh	VS	5 bvh	Perte de performance due à la BVM
2 bvh/pilote	VS	3 pilote	Perte de performance due au BVH
1 bvm/bvh/pilote	VS	4 bvm/bvh	Perte de performance due au pilote de périphérique
2 bvh/pilote	VS	5 bvh	Perte de performance due au pilote de périphérique

Les tests ont été réalisés sur un ordinateur de classe Pentium-II ayant une cadence d'horloge de 450 MHz. La mémoire vive du système est de 384 Mo (méga-octets). Le module BNCL exécutable et l'exécutable Windows n'ont aucun calcul de position à réaliser. Des points calculés à l'avance sont utilisés. Ainsi, seule la performance du flux des données à l'intérieur de l'architecture est évaluée à l'aide de ces tests. Les résultats des tests sont présentés au Tableau III. Les calculs présentés au Tableau II, et utilisant les données du Tableau III, sont répertoriés au Tableau IV (voir section 6.1.2). Le matériel virtuel (BVH) utilisé pour les tests est un compteur de performance. Il calcule le nombre de positions transmises par seconde. Le module BNCL, et l'exécutable

Windows, transmettent tout d'abord un certain nombre de positions au BVH. Ils récupèrent par la suite le résultat de performance à partir d'un port du matériel virtuel et affichent cette valeur à l'écran. L'essai est alors terminé. Cinq essais sont réalisés dans le but d'obtenir une valeur moyenne pour chaque test.

Tableau III

Positions générées pour chaque scénario de mesure de performance

Scénario	Performance (positions/seconde)					
	Essai 1	Essai 2	Essai 3	Essai 4	Essai 5	Moyenne
1 (bvm/bvh/pilote)	33400	32520	34294	33278	34353	*33569*
2 (bvh/pilote)	37272	36711	35689	37879	37425	*36995*
3 (pilote)	45809	46838	47214	47081	46041	*46597*
4 (bvm/bvh)	192678	189394	194932	187617	191939	*191312*
5 (bvh)	197239	191571	187266	186916	190476	*190694*

6.1.2 Analyse

L'analyse des résultats est basée sur les différences entre les scénarios présentés à la Figure 33. Selon le Tableau II, la comparaison des scénarios 1 et 2 et des scénarios 4 et 5 permet de déterminer la perte de performance causée par la machine virtuelle BNCL. Au Tableau IV, il peut être constaté qu'entre les scénarios 4 et 5, une différence infime de 0.3% existe. Cette différence n'est donc pas significative. La présence d'une BVM dans un de ces scénarios ne semble pas avoir influencé le résultat. En comparant les scénarios 1 et 2 toutefois, une différence de près de 10% est constatée. Comme la seule différence

entre les scénarios 1 et 2 et les scénarios 4 et 5 est la présence d'un pilote de périphérique, il est probable que cette différence de 10% soit attribuable à ce pilote. Mais comme la seule composante ajoutée entre les scénarios 1 et 2 est une machine virtuelle BNCL, cette différence de 10% doit être le résultat de l'ajout de celle-ci. La machine virtuelle, dans le contexte étudié, a un impact négatif d'environ 10% sur les performances du système. La différence entre les comparaisons des scénarios 1 et 2 et des scénarios 4 et 5 pourrait indiquer qu'à une valeur de 190000 positions générées par seconde (voir Tableau III), le BVH atteint sa limite de traitement sur l'ordinateur utilisé pour les tests. Ceci expliquerait l'écart de seulement 0.3% observé lors de la comparaison des scénarios 4 et 5. Puisque ces scénarios ne comportent pas de pilote, ils ne sont donc pas ralentis par celui-ci et le BVH peut ainsi potentiellement atteindre sa vitesse maximale de traitement.

Toujours selon le Tableau II, la comparaison entre les scénarios 2 et 3 permet de déterminer l'influence du BVH sur les performances de transmission de l'information. Les données du Tableau IV indique que l'impact est de l'ordre de 20%. La seule différence entre les deux scénarios étant la présence d'un BVH, cet impact de 20% peut être attribué à cette composante. Le traitement effectué par le BVH pour faire abstraction du matériel physique sous-jacent impose donc une pénalité de l'ordre de 20%.

Les deux dernières comparaisons présentées au Tableau II permettent de déterminer l'influence d'une communication avec l'électronique du contrôleur. Il est toutefois important de rappeler qu'un pilote de périphérique simule cet accès. Les pilotes de périphérique peuvent varier beaucoup en performance selon la composante électronique qu'ils contrôlent et la façon dont ce contrôle est effectuée. Les données recueillies donnent une idée de l'impact d'une telle communication. Selon le Tableau IV, cet impact est de l'ordre de 80%. La présence de la machine virtuelle BNCL a peu d'influence sur le résultat. Cette donnée importante pourrait indiquer que le goulot

d'étranglement se situe au niveau de la communication avec l'électronique du contrôleur et non au niveau des composantes logicielles de l'architecture BNCL.

Tableau IV

Impact des composantes sur les performances de l'architecture BNCL

Scénario	Composante testée	Écart moyen
1 VS 2 (bvm/bvh/pilote VS bvh/pilote)	BVM	*-9.3%*
4 VS 5 (bvm/bvh VS bvh)	BVM	*0.3%*
2 VS 3 (bvh/pilote VS pilote)	BVH	*-20.6%*
1 VS 4 (bvm/bvh/pilote VS bvm/bvh)	Pilote	*-82.5%*
2 VS 5 (bvh/pilote VS bvh)	Pilote	*-80.6%*

Ces tests permettent de déterminer l'impact sur les performances de chacune des composantes de l'architecture BNCL. Pour le scénario 1, le plus exigeant en termes de performance en raison du nombre de composantes interagissant, une valeur de 33569 positions générées par seconde a été mesurée. Cette valeur correspond à une avance de l'outil, selon l'équation (1), de 2014 pouces/minute. Cette vitesse est nettement suffisante dans un contexte de production. Par comparaison, une machine-outil haute performance comme la *Mazak Nexus 510C-HS* permet une vitesse maximale en avance rapide de 1968 pouces/minute. L'architecture BNCL ne serait donc pas le goulot

d'étranglement qui limiterait les performances de la machine-outil. De plus, selon les tests effectués, les limitations de performance semblent se trouver au niveau de la communication avec l'électronique du contrôleur et non au niveau de l'architecture BNCL.

6.2 Tests d'extensibilité

Un des principes de base des contrôleurs de machine-outil à architecture ouverte est celui d'extensibilité. L'extensibilité est la possibilité qui est donnée à l'usager d'ajouter des fonctionnalités à une machine-outil. Le contrôleur doit tenir compte de ces modifications et les rendre disponibles dans les programmes de coupe de l'usager. Les quatre principes sur lesquels se base l'architecture BNCL sont :

a. permettre la portabilité des programmes de coupe et des logiciels développés pour un contrôleur;
b. permettre l'extensibilité de la machine-outil en donnant la possibilité à l'utilisateur d'ajouter des composantes et d'utiliser ces dernières dans ses programmes. Les composantes peuvent être aussi variées que des capteurs de fin de course ou des tables d'indexation;
c. permettre aux programmes créés par l'utilisateur de réagir aux événements pouvant survenir sur la machine;
d. ne pas confiner l'utilisateur à un seul langage de programmation.

L'extensibilité de la machine-outil est reliée aux points b) et c). Le point b) prescrit que l'usager doit pouvoir interroger des composantes ajoutées sur la machine-outil par l'usager. Le point c) vient compléter le précédent et prescrit qu'une composante ajoutée à la machine-outil par l'usager doit pouvoir signaler à un programme de coupe qu'une certaine condition vient d'être rencontrée. Ce point c) est à la base de la communication

bidirectionnelle avec les composantes ajoutées à la machine-outil. Le test d'extensibilité de l'architecture BNCL se concentre sur le point c). Il est considéré ici que si l'architecture BNCL répond efficacement au point c), celle-ci répondra également au point b). Cela est attribuable au fait que ces deux aspects ont pour base la communication avec des composantes ajoutées à la machine-outil par l'usager. La différence entre le point b) et le point c) est le travail utile réalisé avec cette communication.

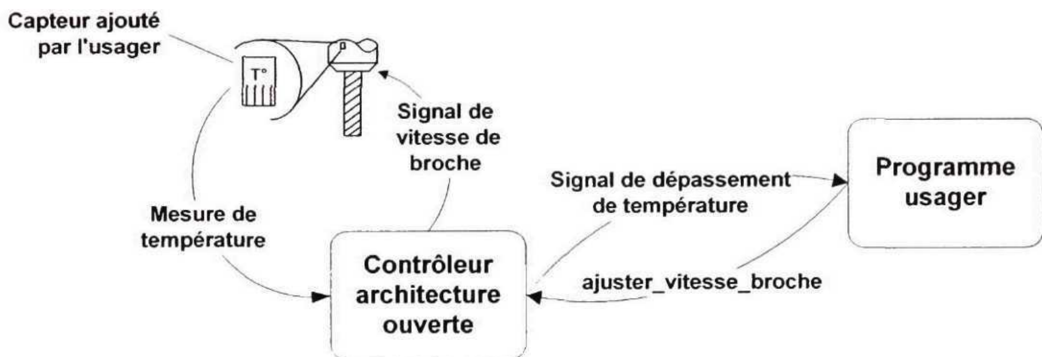

Figure 34 Scénario d'ajustement de la vitesse de la broche

Pour valider l'extensibilité de la machine-outil, un scénario d'adaptation de la vitesse de rotation de la broche en fonction de la température mesurée au niveau de l'outil a été élaboré. Le scénario réel est présenté à la Figure 34. La simulation réalisée est présentée à la Figure 35.

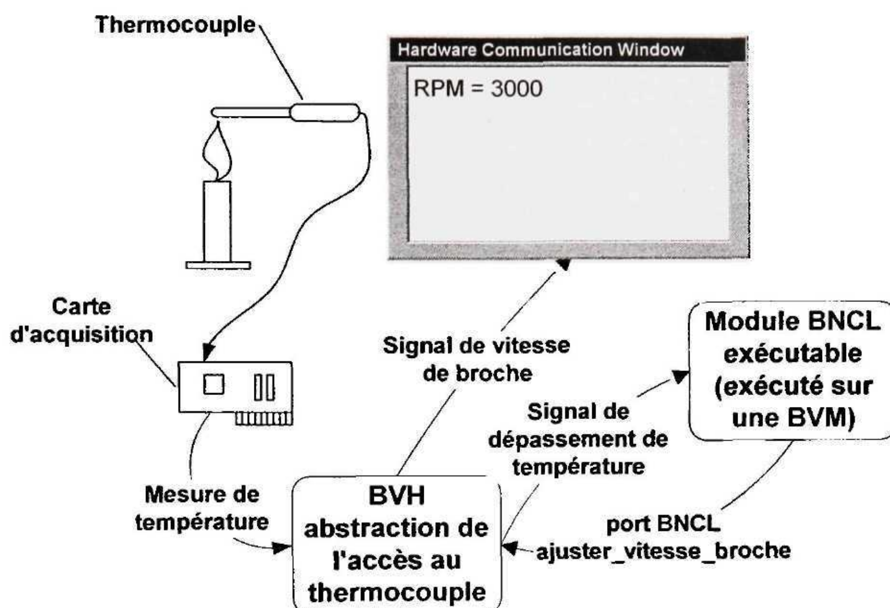

Figure 35 Simulation de l'ajustement de la vitesse de la broche

Dans la simulation (Figure 35), le signal de vitesse de la broche n'est pas envoyé à une machine-outil réelle, mais à une boîte de dialogue associée au BVH. La vitesse courante de la broche est affichée dans cette fenêtre. La température est mesurée à l'aide d'un thermocouple connecté à une carte d'acquisition de données. Le BVH abstrait l'accès à cette carte et rend la valeur de la température disponible au programme s'exécutant sur la machine virtuelle BNCL. Lorsque la température dépasse un certain niveau, un signal est envoyé du BVH vers la BVM. Un module BNCL associé à ce signal est alors exécuté. Celui-ci modifie la vitesse de la broche en communicant en retour avec le BVH. Pour la simulation, la température est modifiée à l'aide d'une flamme.

6.2.1 Analyse

Le test d'extensibilité a été un succès et la vitesse de rotation de la broche a pu être modifiée en fonction de la température mesurée par le thermocouple. La facilité avec laquelle le montage a pu être réalisé a permis de constater la flexibilité d'utilisation de l'architecture BNCL dans un contexte d'extensibilité de la machine-outil. Les utilitaires utilisés pour générer automatiquement le code d'implémentation du BVH (voir Figure 23 à la section 3.9), qui fait l'abstraction de la carte d'acquisition, ont permis d'accélérer sensiblement la préparation du test. De tels outils pourraient également être utilisés dans un contexte de production. Les détails du test d'extensibilité sont présentés à la section 3.8. La Figure 36 et la Figure 37 montrent des photos prises lors de la réalisation du test d'extensibilité. La Figure 36 présente le montage complet, alors que la Figure 37 présente l'interface usager telle que visualisée lors du test.

Figure 36 Photo du montage réalisé pour le test d'extensibilité

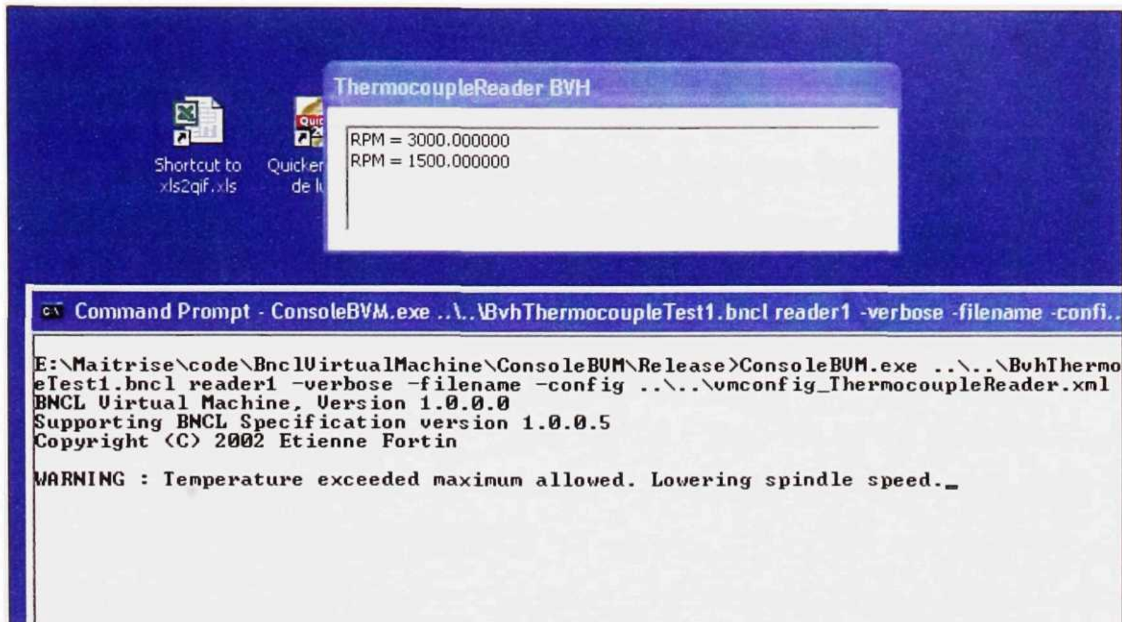

Figure 37 Photo de l'interface usager prise lors du test d'extensibilité

6.3 Usinage de came

Plusieurs concepts ont été employés pour définir l'architecture BNCL. Comme cette architecture a été conçue pour corriger les lacunes de l'architecture logicielle existante, une comparaison avec celle-ci est de mise. Un test réalisé dans un environnement réel de production a donc été prévu. Ce test vise à montrer la flexibilité de l'architecture BNCL dans un contexte de contrôle de machine-outil. Par flexibilité, il est entendu la souplesse donnée à l'usager dans la programmation de la machine. Le présent test vise également à démontrer les performances pouvant être atteintes dans un contexte réel de production. L'idée retenue consiste à usiner une pièce réelle avec l'architecture logicielle actuelle et de simuler l'usinage de cette même pièce avec l'architecture BNCL. Il n'est en effet pas encore possible de relier l'architecture BNCL à une véritable machine-outil puisqu'un certain travail de développement reste à faire en ce sens. Le test n'en est pas moins valable. En effet, la démonstration de la flexibilité d'utilisation de l'architecture ne

nécessite pas l'usinage réel d'une pièce. C'est le processus de programmation qui est évalué. L'usinage à partir de l'architecture BNCL constituerait la dernière étape du processus. Ainsi, les performances nécessaires peuvent être déterminées théoriquement tel que vu à la section 6.1. La vitesse théorique maximale que pourrait atteindre l'architecture BNCL lors d'un usinage réel peut donc être déterminée sans devoir effectuer cet usinage. Par contre, le seul moyen d'évaluer les performances de l'architecture logicielle actuellement utilisée est d'effectuer un usinage réel de pièce. Cet usinage est donc essentiel pour tester l'architecture actuelle, mais non pour l'architecture BNCL.

6.3.1 Objectifs et description du test

La pièce retenue pour le test d'usinage dans un environnement réel de production est une came. Ce type de pièce a été choisi pour ses contours souvent complexes obéissant à des équations mathématiques bien définies. Ces équations peuvent parfois être de degré élevé (sept ou plus) et nécessiter des calculs trigonométriques. Cette complexité permet donc une bonne comparaison des performances des architectures dans un contexte de production. L'usinage d'une came se fait également par segments, chacun suivant une équation mathématique distincte. Ce découpage de la came en morceaux indépendants permet de paramétrer l'usinage.

Du point de vue de l'architecture BNCL, le test d'usinage a pour objectif principal de démontrer la facilité d'utilisation de celle-ci dans un contexte d'application réelle. Démontrer que l'architecture BNCL offre des performances adéquates dans un contexte de production est également un objectif visé. De plus, ce test contribue à démontrer la faisabilité du support de multiples langages de programmation sur l'architecture BNCL, ainsi que la capacité de cette dernière à gérer des calculs mathématiques complexes. Finalement, un des buts de ce test est de démontrer les lacunes de l'architecture

logicielle actuellement utilisée, en termes de manque de flexibilité de l'environnement de programmation et de performances.

6.3.2 Scénario

Pour réaliser ce test comparatif, la programmation de la came à usiner est réalisée par trois méthodes : programmation à l'aide du module de fabrication assistée par ordinateur (FAO) du logiciel Pro/Engineer, programmation à l'aide des fonctionnalités paramétriques d'un contrôleur Fanuc 16M et programmation à l'aide de l'architecture BNCL. L'étalon utilisé pour les trois phases est l'usinage réalisé avec un logiciel de fabrication assistée par ordinateur. En effet, puisque toutes les positions de l'outil sont connues depuis la phase du post-processeur, il n'est pas nécessaire d'effectuer de calcul numérique sur le contrôleur. C'est donc ce test qui aura théoriquement les meilleures performances pour l'architecture logicielle actuellement utilisée. Les trois tests effectués sont présentés à la Figure 38. On peut y voir que dans le cas de l'architecture BNCL, le contrôleur est simulé et l'accès à l'électronique du contrôleur est émulé par un pilote de périphérique. Cette technique est la même que celle utilisée à la section 6.1.

La came à usiner comporte quatre segments. Chacun de ces segments suit une courbe mathématique distincte. La Figure 39 montre le profil de la came. Les quatre portions de la came y sont indiquées. Le parcours de l'outil est également présent, de même que certaines variables et paramètres utilisés pour décrire la came. Le déplacement instantané du galet, en fonction de l'angle et du parcours, est noté S_1, S_2, S_3 et S_4. Les positions instantanées des points situés au centre du galet sont notées (x_1, y_1), (x_2, y_2), (x_3, y_3) et (x_4, y_4). Le rayon primitif de la came est noté R_0. Les déplacements se font à partir de ce rayon primitif.

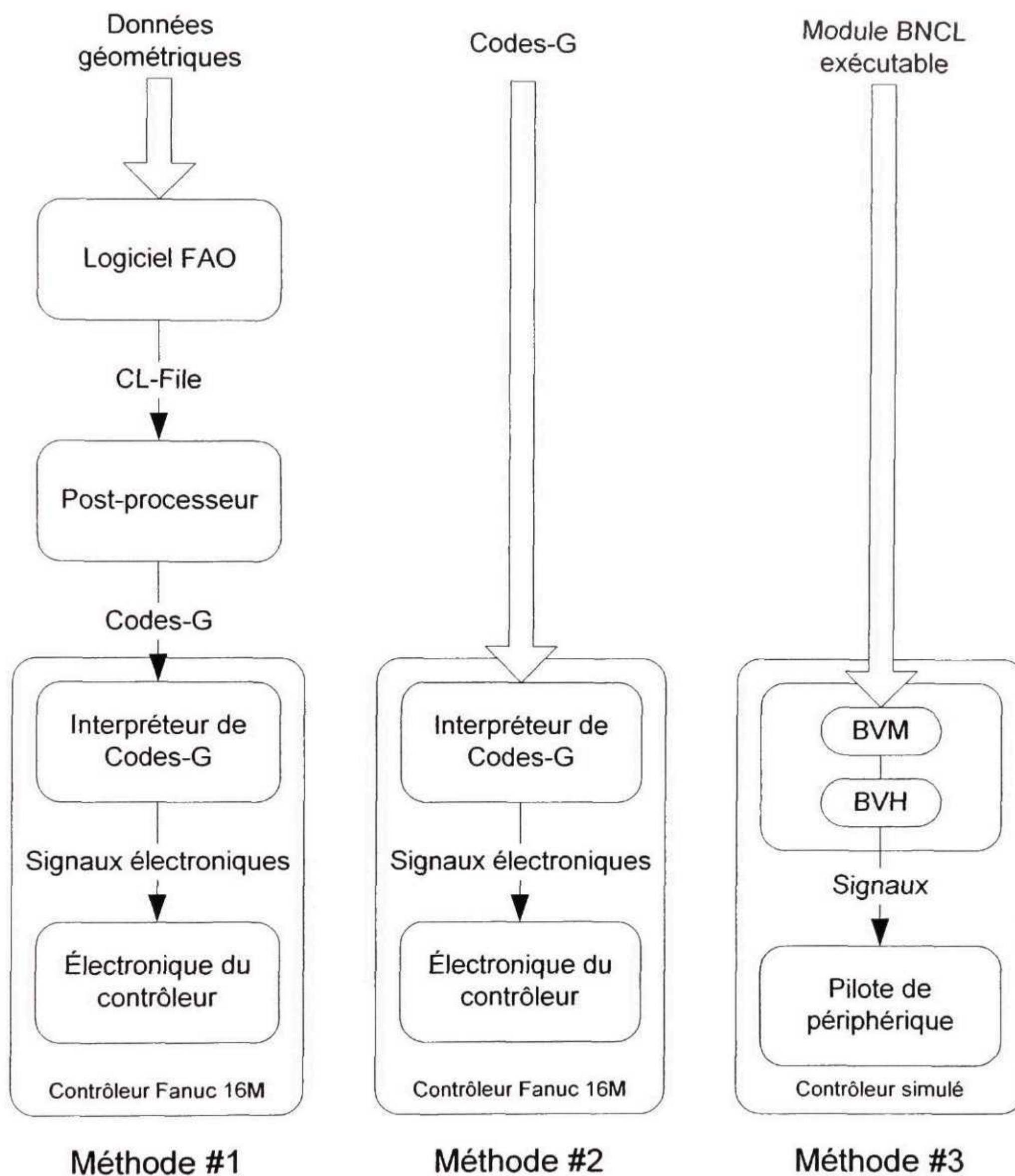

Figure 38 Test d'usinage de came à l'aide de trois méthodes différentes

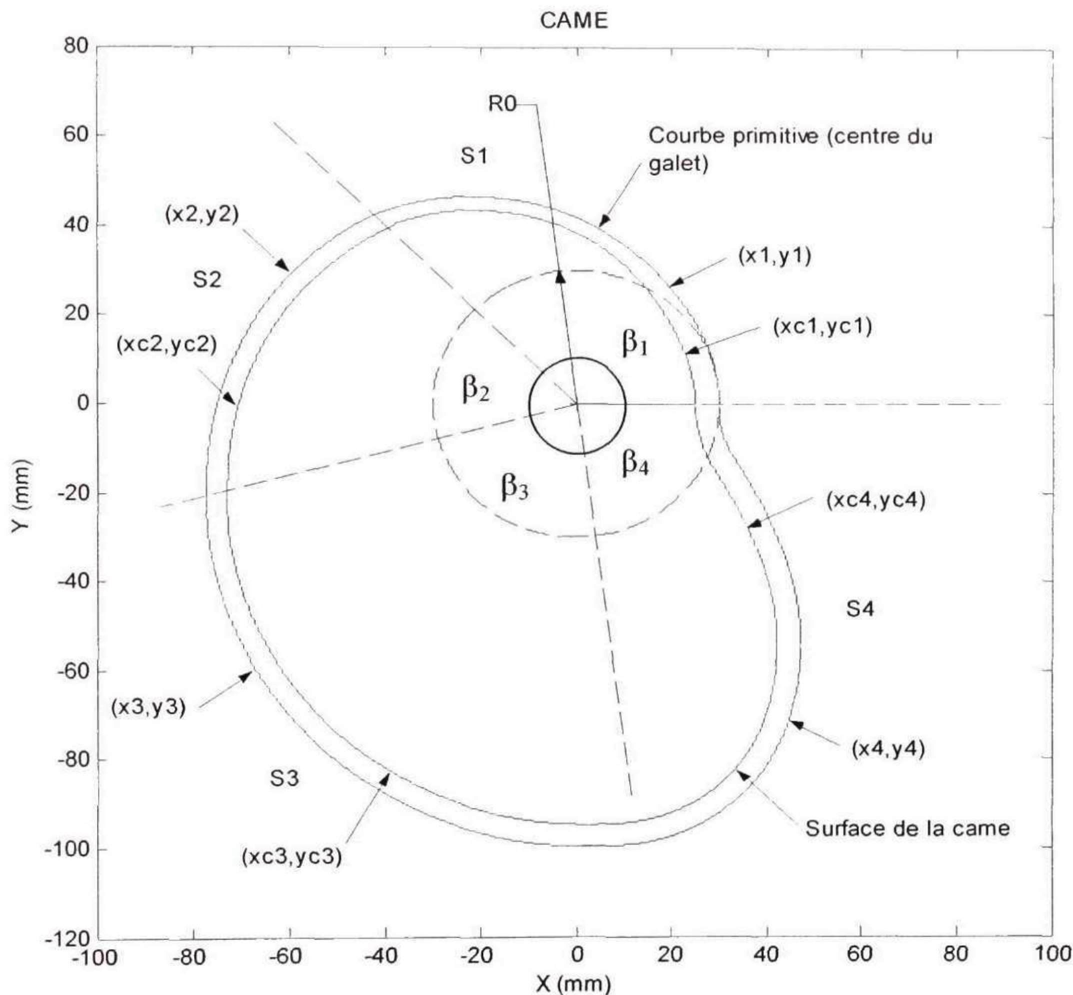

Figure 39 Came définie pour le test d'usinage dans un contexte de production

La rotation de la came autour de son axe produit un mouvement de va-et-vient du galet en contact avec le contour de la came. La position zéro de la came est montrée à la Figure 39. Le galet se trouve confondu avec l'axe des x. La rotation de la came se fait dans le sens des aiguilles d'une montre. Le déplacement horizontal du galet, en fonction de l'angle de rotation de la came, est montré à la Figure 40. Quelques paramètres sont définis sur cette figure. L'ouverture d'angle de chacun des segments est notée β_1, β_2, β_3

et β_4. Le déplacement relatif du galet lors du passage sur chacun des segments est noté L_1, L_2, L_3 et L_4. Les angles sont exprimés en radians et les déplacements en mm.

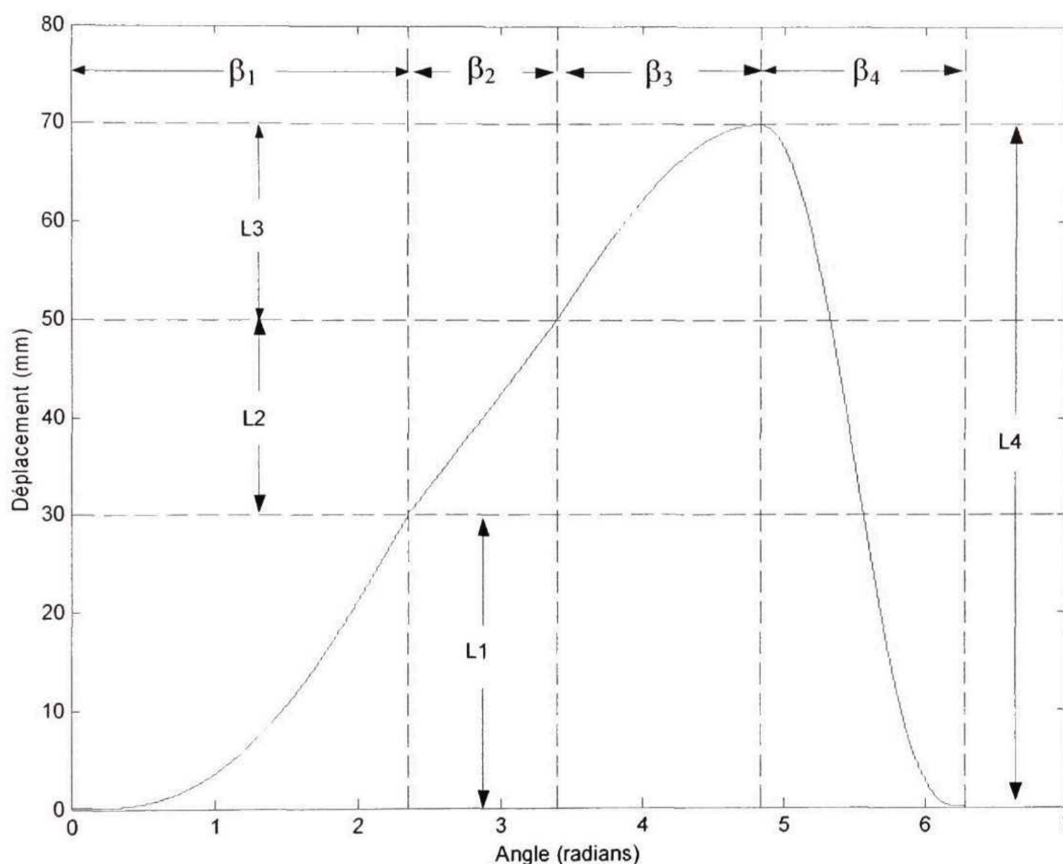

Figure 40 Déplacement du galet en fonction de l'angle de rotation

Le profil de chaque segment obéit à une équation mathématique distincte. Ces équations ont été choisies pour assurer une continuité de tangence du mouvement entre chacune des portions de la came. Les équations sont tirées du livre *Mechanisms and Dynamics of Machinery* [46]. Il a de plus été décidé que l'usinage se ferait avec un outil de diamètre correspondant à celui du galet. Le parcours d'outil se confond alors avec le parcours du centre du galet et les mêmes équations peuvent ainsi être utilisées. Les équations suivantes ont été utilisées :

Segment S1

La portion S1 couvre la partie de la came balayée lorsque $\theta \in [0, \beta_1[$

Le mouvement est cycloïdal.

Calculs du déplacement et de la vitesse du galet.

$$s_1 = L_1\left(\frac{\theta}{\beta_1} - \frac{1}{\pi}\sin\left(\frac{\pi\theta}{\beta_1}\right)\right)$$

$$v_1 = \frac{L_1}{\beta_1}\left(1 - \cos\left(\frac{\pi\theta}{\beta_1}\right)\right)$$

Calculs de la position du centre du galet.

$$R = (R_0 + s_1)$$

$$x_1 = R\cos(\theta)$$

$$y_1 = R\sin(\theta)$$

Segment S2

La portion S2 couvre la partie de la came balayée lorsque $\theta \in [\beta_1, \beta_1+\beta_2[$

Le mouvement est à vitesse constante.

Calculs du déplacement et de la vitesse du galet.

$$\mu = \theta - \beta_1$$

$$s_2 = L_1 + L_2\left(\frac{\mu}{\beta_2}\right)$$

$$v_2 = \frac{L_2}{\beta_2}$$

Calculs de la position du centre du galet.

$$R = (R_0 + s_2)$$
$$x_2 = R\cos(\theta)$$
$$y_2 = R\sin(\theta)$$

Segment S3

La portion S3 couvre la partie de la came balayée lorsque $\theta \in [\beta_1+\beta_2, \beta_1+\beta_2+\beta_3[$

Le mouvement est harmonique.

Calculs du déplacement et de la vitesse du galet.

$$\mu = \theta - (\beta_1 + \beta_2)$$
$$s_3 = L_1 + L_2 + L_3\left(\sin\left(\frac{\pi\mu}{2\beta_3}\right)\right)$$
$$v_3 = \frac{\pi L_3}{2\beta_3}\left(\cos\left(\frac{\pi\mu}{2\beta_3}\right)\right)$$

Calculs de la position du centre du galet.

$$R = (R_0 + s_3)$$
$$x_3 = R\cos(\theta)$$
$$y_3 = R\sin(\theta)$$

Segment S4

La portion S4 couvre la partie de la came balayée lorsque $\theta \in [\beta_1+\beta_2+\beta_3, 2\pi]$

Le mouvement est polynomial de huitième ordre.

Calculs du déplacement et de la vitesse du galet.

$$\mu = \theta - \left(\beta_1 + \beta_2 + \beta_3\right)$$

$$s_4 = L_4 \begin{pmatrix} 1 - 2.63415\left(\dfrac{\mu}{\beta_4}\right)^2 + 2.78055\left(\dfrac{\mu}{\beta_4}\right)^5 + 3.17060\left(\dfrac{\mu}{\beta_4}\right)^6 \\ -6.87795\left(\dfrac{\mu}{\beta_4}\right)^7 + 2.56095\left(\dfrac{\mu}{\beta_4}\right)^8 \end{pmatrix}$$

$$v_4 = \frac{L_4}{\beta_4} \begin{pmatrix} -5.26830\left(\dfrac{\mu}{\beta_4}\right) + 13.90275\left(\dfrac{\mu}{\beta_4}\right)^4 + 19.02360\left(\dfrac{\mu}{\beta_4}\right)^5 \\ -43.14565\left(\dfrac{\mu}{\beta_4}\right)^6 + 20.48760\left(\dfrac{\mu}{\beta_4}\right)^7 \end{pmatrix}$$

Calculs de la position du centre du galet.

$$R = \left(R_0 + s_4\right)$$
$$x_4 = R\cos(\theta)$$
$$y_4 = R\sin(\theta)$$

Certaines variables utilisées dans le contrôle du profil de la came, soit les variables L_1 à L_4, β_1 à β_4 et R_0, ont été identifiées comme étant paramétrables. L'usager, si les programmes étaient utilisés en production, aurait la possibilité de changer ces valeurs pour modifier la géométrie de la came. Le programme doit alors permettre de modifier facilement les valeurs de ces paramètres. Les valeurs attribuées aux paramètres pour réaliser le présent test, tant pour la programmation à l'aide du logiciel de FAO que pour les deux autres méthodes, sont présentées au Tableau V. Les programmes utilisés pour l'usinage de la came sont présentés en annexe. L'ANNEXE 7 contient le programme réalisé à l'aide des Codes-G paramétriques et l'ANNEXE 8, celui pour l'architecture BNCL.

Tableau V

Valeurs des paramètres pour l'usinage de la came

Paramètre	Valeur
β_1	$\frac{3\pi}{4}$ radians
β_2	$\frac{\pi}{3}$ radians
β_3	$\frac{11\pi}{24}$ radians
β_4	$\frac{11\pi}{24}$ radians
L_1	30 mm
L_2	20 mm
L_3	20 mm
L_4	70 mm
Ro	25 mm

Pour la programmation réalisée à l'aide du module de FAO du logiciel Pro/Engineer, les valeurs des paramètres sont fixes. Pour la programmation réalisée à l'aide des deux autres méthodes, soit les Codes-G et l'architecture BNCL, ces paramètres sont modifiables dans le programme. Le programme réalisé à l'aide de l'architecture BNCL incorpore une flexibilité supplémentaire. En effet, au lieu de programmer une came paramétrable comportant quatre segments toujours basés sur les mêmes équations mathématiques, un cycle de came complet a été programmé. Ce cycle de came accepte un nombre variable de segments, chacun pouvant obéir à une équation mathématique au

choix. Cela permet d'usiner une infinité de géométries de cames. Cette flexibilité supplémentaire est ajoutée afin de vérifier la souplesse avec laquelle l'architecture BNCL peut être utilisée dans la programmation de la machine. Le cycle de came est utilisé pour générer le parcours d'outil pour la came demandée. Le Tableau VI résume les caractéristiques de chacun des moyens de programmation. Il est important de souligner que l'usinage est réalisé avec un outil de diamètre correspondant au galet. Les positions du centre de l'outil et du centre du galet sont donc confondues. Ceci permet d'utiliser sans modifications les équations du centre du galet lors de la génération du parcours d'outil.

Tableau VI

Méthodes utilisées dans la programmation de la came

Outil	Caractéristiques
Logiciel FAO	• Programmation à partir d'un solide 3D de la came. • Aucun paramètre modifiable. • Aucun calcul numérique de position d'outil n'est nécessaire sur le contrôleur. • Le programme a été réalisé à l'aide de Pro/Engineer.
Codes-G paramétriques	• Les paramètres sont modifiables. • Came de quatre segments comportant toujours les mêmes équations. • Calculs en temps réel des positions de l'outil. • Écrit en Custom Macro B pour un contrôleur Fanuc 16M.
Architecture BNCL	• L'usinage est simulé. • Plus de quatre segments possible. • Les paramètres sont modifiables. • Une simulation du calcul des positions est réalisée. • Le programme est écrit à l'aide du C.

6.3.3 Résultats et analyse

Les résultats sont résumés au Tableau VII. Les valeurs simulées pour l'architecture BNCL ont été calculées à l'aide de l'équation (1), présentée à la section 6.1. La distance maximale entre deux points (INC) utilisée est de 0.001 pouces pour cette simulation. Un incrément d'angle de 0.001 radians permet d'atteindre cette valeur maximale pour le profil de came choisi. Cet incrément d'angle a été utilisé pour les Codes-G paramétriques et l'architecture BNCL. Dans le cas du logiciel de FAO, la précision au niveau du post-processeur a été ajustée pour générer un nombre équivalent de positions. L'usinage a été réalisé sur une machine-outil *Hitachi-Seiki VS50* comportant un contrôleur Fanuc 16M.

Le test d'usinage de la came permet d'estimer le comportement de l'architecture BNCL dans un contexte réel de production. Le Tableau VII montre que l'impact des calculs sur la performance, dans le cas du programme écrit en Codes-G, est significatif. En effet, ses performances, par rapport au programme ne comportant aucun calcul numérique, sont dix fois inférieures. Ceci est causé par le fait que le contrôleur doit calculer chaque position de l'outil, en temps réel, pendant l'usinage. L'architecture logicielle actuellement utilisée pour le contrôle des machines-outils comporte donc une lacune au niveau des performances. En contrepartie, l'architecture BNCL a été en mesure de générer assez de positions pour permettre à l'outil d'avancer à la vitesse théorique de 880 pouces/minute. Évidemment, cette valeur ne tient pas compte de tous les calculs d'accélération et de décélération de l'outil effectués par le contrôleur. Cependant, en tenant compte des résultats des tests de la section 6.1, où l'électronique du contrôleur a été identifié comme composante limitative des performances, il peut être affirmé que lors d'un usinage réel l'outil aurait été capable d'avancer à une vitesse au moins équivalente à 11.81 pouces/minute. Il est possible que la performance aurait été meilleure puisque le programme Codes-G provenant du logiciel de FAO comporte, comme son homologue paramétrique, une perte de performance causée par

l'interprétation des Codes-G. Dans l'éventualité où l'électronique du contrôleur serait en mesure d'accepter toutes les positions envoyées, une vitesse de 880 pouces/minute est atteignable avec l'architecture BNCL. Il est important de rappeler que ces tests ont été effectués sur un ordinateur bas de gamme (Pentium II 450 MHz) et qu'un ordinateur plus puissant pourrait être utilisé en production.

Tableau VII

Vitesses possibles de l'outil dans le test d'usinage d'une came

Architecture	Vitesse Demandée	Vitesse atteinte ou simulée (mm/min)	Vitesse atteinte ou simulée (pouce/min)
Logiciel de FAO (Pro/Engineer)	5000 mm/min (196.97 po/min)	300	11.81
Codes-G paramétriques (Custom Macro B de Fanuc)	5000 mm/min (196.97 po/min)	35	1.38
Architecture BNCL (Langage C)	Sans objet	22375.37	880.92

Par ailleurs, le test d'usinage de la came montre la flexibilité d'utilisation de l'architecture BNCL pour le contrôle des machines-outils. Au lieu de limiter l'usinage à une came comportant quatre segments, chacun obéissant à des équations mathématiques définies à l'avance, l'architecture BNCL propose un cycle de came permettant un nombre variable de segments et d'équations. La programmation d'un cycle aussi souple à l'aide des Codes-G paramétrique n'a pas été possible, ces codes n'exposant pas les fonctionnalités nécessaires pour y arriver. Finalement, le test d'usinage de la came indique la faisabilité de l'utilisation de multiples langages de programmation avec

l'architecture BNCL. Tous les programmes sont en effet écrits à l'aide du langage C, un langage très général représentatif de beaucoup d'autres langages, et de l'assembleur BNCL. La disponibilité d'autres compilateurs, par exemple un compilateur APT ou FORTRAN, aurait permis d'écrire certaines routines du cycle de came dans ces langages.

6.4 Résumé

Plusieurs tests de validation de l'architecture BNCL ont été effectués. Une partie des tests vise à déterminer la performance de l'architecture dans un contexte de contrôle de machine-outil. Ces tests permettent de déterminer si l'architecture BNCL possède des capacités adéquates au niveau de la performance. Le test d'extensibilité de la machine-outil permet, pour sa part, de valider la possibilité d'utiliser l'architecture BNCL dans un contexte de contrôleur à architecture ouverte. Finalement, une partie des essais valide l'ensemble dans un contexte réel de production et permet de comparer l'architecture BNCL avec l'architecture logicielle actuellement utilisée.

Les performances dans un contexte de production sont adéquates. De plus, comme la machine virtuelle en est à sa première version, des améliorations de performance sont possibles. Il est intéressant de noter que les tests ont tous été effectués sur un ordinateur de classe Pentium-II ayant une cadence d'horloge de 450 MHz. Cet ordinateur peut être considéré bas de gamme. Pourtant, les performances de l'architecture dans un contexte de production sont tout de même adéquates. Le goulot d'étranglement se situerait, selon les essais effectués, au niveau de la communication avec l'électronique du contrôleur et non au sein de l'architecture BNCL. Le test d'usinage de came a permis de valider cette hypothèse dans un contexte de production. Les performances rencontrées ont été très satisfaisantes. De plus, la réalisation d'un cycle de came complet pour l'usinage de la came a permis de constater la flexibilité d'utilisation de l'architecture BNCL.

DISCUSSION

La validation de l'architecture BNCL s'est concentrée sur trois aspects : la validation de la performance de l'architecture, la validation de l'extensibilité que procure l'architecture et finalement la simulation de l'usinage d'une pièce réelle. Le test de performance a permis d'identifier la composante limitant les performances sur un contrôleur implémentant l'architecture BNCL. La validation de l'extensibilité a permis de vérifier que l'architecture BNCL répond efficacement aux critères de réalisation d'un contrôleur à architecture ouverte. Et finalement, la simulation d'usinage, et la comparaison avec l'architecture actuellement utilisée, a permis de vérifier si les lacunes de l'architecture actuelle sont corrigées et si l'architecture BNCL peut être utilisée efficacement dans un contexte réel d'usinage.

La validation de performance a permis de déterminer l'impact de chaque composante de l'architecture BNCL sur les performances. Ces essais ont été réalisés en calculant le nombre de positions pouvant être générées en une seconde. Ce nombre de positions influence la vitesse avec laquelle l'outil peut se déplacer. Plus ce nombre est élevé, plus l'outil est en mesure de se déplacer rapidement. Il a été démontré qu'avec un ordinateur bas de gamme, l'architecture BNCL peut générer assez de positions pour permettre à l'outil d'avancer à une vitesse aussi élevée que 2000 pouces/minute. Cette vitesse est théorique et ne tient pas compte des autres composantes du contrôleur qui pourraient ralentir les performances. Les modules devant calculer l'accélération et la décélération de l'outil sont des exemples de telles composantes. Cependant, cette valeur montre que dans un contexte d'usinage réel, l'architecture BNCL ne serait pas la composante qui limiterait les performances. La composante limitative a été identifiée comme étant l'électronique du contrôleur et la communication avec celle-ci. Cette communication a été simulée à l'aide d'un pilote de périphérique. Il est évident que les pilotes de périphérique présentent des performances variables selon le type de composante

contrôlée. Cependant, le très grand impact sur les performances de l'utilisation du pilote de périphérique permet de prévoir que c'est bien l'électronique du contrôleur qui limiterait les performances. Cette hypothèse a été confirmée lors de l'usinage de la came, où l'architecture BNCL a permis des performances très supérieures à ce que la machine-outil réelle a été en mesure d'effectuer.

Le test d'extensibilité, quant à lui, a pour but la validation de l'architecture BNCL dans un contexte de contrôleur à architecture ouverte. Le test choisi a consisté en l'ajustement de la vitesse de la broche de la machine-outil en fonction de la température observée lors de l'usinage. Un montage utilisant l'architecture BNCL comme architecture logicielle, un thermocouple et une carte d'acquisition comme matériel physique, a été réalisé. La vitesse de la broche simulée a pu être ajustée en fonction du dépassement d'un certain seuil de température. Le peu de temps nécessaire au développement de cet essai a permis de constater la flexibilité de l'architecture BNCL dans un tel contexte de contrôle de machine. La conception des composantes logicielles n'a présenté que très peu de problèmes. Les outils développés dans le cadre du projet, et présentés au Chapitre 3, ont grandement contribué à ce développement rapide.

Finalement, la simulation d'usinage d'une pièce réelle poursuivait deux buts : comparer l'architecture BNCL avec l'architecture logicielle actuellement utilisée et valider la performance de l'architecture BNCL dans un contexte de production. La pièce à usiner était une came comportant quatre segments. Certains paramètres étaient modifiables. Dans le cas de l'architecture actuelle, l'usinage a été réalisé sur une machine-outil. L'usinage à l'aide de l'architecture BNCL a été simulé, et les performances théoriques ont été calculées. Il en est ressorti que l'architecture BNCL, dans un contexte réel d'usinage de pièce, permet des performances très satisfaisantes. L'usinage à l'aide des Codes-G paramétriques a démontré que cette architecture logicielle est mal adaptée à l'usinage algorithmique où les positions de l'outil doivent être calculées en temps réel. Le programme réalisé à l'aide d'un logiciel de FAO n'a pas non plus présenté de

performances exceptionnelles. Cependant, les calculs effectués par l'électronique du contrôleur, comme les calculs d'accélération et de décélération de l'outil, sont probablement en cause. Ceci corrobore une des conclusions des tests de performance qui identifiait l'électronique du contrôleur et la communication avec celle-ci comme le maillon limitant les performances. Avec l'architecture BNCL, une très grande vitesse d'outil est possible. En autant que le contrôleur puisse accepter toutes les positions qui lui sont envoyées, l'architecture BNCL sort gagnante de ce test du point de vue des performances. La flexibilité de programmation de l'architecture BNCL est également démontrée par ce test. Les programmes de coupe sont en effet programmés à l'aide du langage C. Ce langage est très général. La possibilité de l'adapter pour l'architecture BNCL laisse supposer que celle-ci peut supporter plusieurs langages de programmation, comme il a été imaginé au départ. Finalement, le cycle d'usinage programmé permet l'usinage d'une came comportant autant de segment que désiré. Ces segments peuvent, de plus, obéir à n'importe quelle équation mathématique. Une telle flexibilité n'a pu être atteinte avec les outils logiciels utilisés actuellement sur les contrôleurs de machine-outil.

CONCLUSION

L'architecture BNCL vise à répondre aux besoins du contrôle des machines-outils à commande numérique. L'étude de la littérature indique que l'architecture logicielle actuellement utilisée présente certaines lacunes qui l'empêche de répondre efficacement aux nouveaux besoins du domaine du contrôle des machines-outils. Entre autres, la recherche sur les contrôleurs à architecture ouverte montre que l'architecture logicielle actuelle est inadéquate. Ce nouveau type de contrôleur permet la portabilité tant des programmes de coupe que des logiciels utilisés pour contrôler la machine. Il permet également à l'usager d'étendre les fonctions de la machine-outil selon ses besoins. Ces nouvelles fonctionnalités ne peuvent être prises en charge par l'architecture logicielle actuelle. De plus, cette architecture présente des lacunes au niveau des performances, ce qui empêche l'implémentation de certains algorithmes de coupe, ou de cycles d'usinage, dans le contrôleur.

La revue de littérature a également fait ressortir quatre principes majeurs sur lesquelles se base la conception de l'architecture BNCL :

a. la portabilité logicielle ;
b. l'extensibilité de la machine-outil ;
c. la réactivité aux événements ;
d. le support multi langage.

Il est souhaitable que ces caractéristiques soient prises en compte par une architecture logicielle permettant la création d'un contrôleur à architecture ouverte. Le Chapitre 2 a montré que l'architecture BNCL rencontre le premier et le dernier principe en proposant une architecture de machine virtuelle. Cette machine virtuelle, ou microprocesseur virtuel, permet de réaliser l'abstraction de l'environnement logiciel à partir duquel le

programme s'exécute. Cette abstraction permet de rendre les programmes portables, puisqu'ils ne visent que l'architecture BNCL et rien d'autre. De plus, l'abstraction rend réalisable le support de plusieurs langages de programmation. En effet, comme la machine virtuelle BNCL se présente comme un microprocesseur virtuel, des compilateurs de différents langages peuvent être créés pour cette architecture. Un compilateur du langage C est actuellement fonctionnel et un compilateur APT est en développement.

Les deuxième et troisième principes sont respectés grâce à une couche d'abstraction du matériel physique de la machine-outil. Tous les accès à ce matériel, que ce soit les axes, les accessoires ou les composantes ajoutées à la machine par l'usager, sont accessibles via cette couche d'abstraction. Cette couche, nommée matériel virtuel BNCL ou BVH, permet de communiquer avec ce qui se branche sur la machine. Elle permet également aux accessoires d'envoyer des signaux vers les programmes usagers. La communication avec les accessoires est donc bidirectionnelle. La machine virtuelle BNCL (BVM) et le matériel virtuel BNCL (BVH) forment les deux modules principaux de l'architecture et constituent le fondement de ce projet.

La façon dont l'architecture BNCL a été implémentée pour répondre aux objectifs des contrôleurs à architecture ouverte, et permettre la réalisation des différentes validations, a été présentée au Chapitre 3. Ce chapitre décrit l'architecture BNCL de référence. Cependant, en aucun cas cette manière de faire ne doit être considérée unique. Ce fait est d'ailleurs essentiel pour la portabilité de l'architecture. L'architecture BNCL permet cette portabilité de l'architecture et la mise en oeuvre décrite au Chapitre 3 n'est qu'une indication de ce qui peut être fait.

Il a été montré que l'architecture BNCL répond aux quatre principes énumérés ci haut. De plus, cette architecture propose plusieurs avantages et améliorations qui ont été présentés au Chapitre 4. Entre autres, la portabilité logicielle est possible grâce à la

machine virtuelle BNCL et au matériel virtuel BNCL. L'extensibilité de la machine est également possible. De plus, l'architecture BNCL constitue une amélioration notable par rapport au système actuel de programmation des machines-outils à commande numérique. L'extensibilité du langage de contrôle, les structures syntaxiques de langages de programmation permettant la réalisation d'applications complexes et les performances sont tous des points ayant fait l'objet d'une attention particulière dans l'élaboration de l'architecture BNCL. La pertinence de ces éléments, et la façon dont l'architecture BNCL y satisfait, a été démontrée dans l'analyse.

La validation présentée au Chapitre 6 a permis de confirmer la validité des concepts mis de l'avant dans l'élaboration de l'architecture BNCL. Cette architecture possède des performances adéquates pour un environnement de production. Cette constatation fait suite aux différentes simulations informatiques, de calculs et d'analyse. L'extensibilité de la machine-outil permise par l'architecture BNCL a également fait l'objet d'une validation. Un scénario d'adaptation de la vitesse de la broche en fonction de la température de l'outil a permis de constater l'efficacité de l'architecture BNCL dans ce genre de situation. La facilité avec laquelle le scénario a été implémenté a de plus montré la flexibilité d'utilisation de l'architecture et l'utilité des outils développés dans le cadre du projet. Ces outils sont présentés au Chapitre 3. L'architecture a finalement été validée dans un contexte réel d'usinage de pièce. La simulation, et la comparaison avec l'architecture logicielle actuellement utilisée dans l'industrie, ont permis de constater une fois de plus que les performances de l'architecture BNCL sont adéquates. La flexibilité de l'architecture, ainsi que la souplesse avec laquelle elle peut être utilisée, a pu être constatée. Tous ces aspects étaient des avantages escomptés de l'architecture lors de la conception de celle-ci et sont présentés au Chapitre 4.

Finalement, le Chapitre 5 présente des méthodes d'intégration de l'architecture BNCL au processus actuel de programmation des machines-outils à commande numérique. En effet, imposer un nouveau cadre de travail n'est pas une manière efficace d'amener un

changement et une cohabitation, au moins temporaire, sera nécessaire. Différentes avenues d'intégration au processus actuel sont possibles et ont été analysées.

L'architecture BNCL présente des avantages intéressants du point de vue de la programmation des machines-outils à commande numérique. Les essais réalisés jusqu'à maintenant semblent démontrer que l'architecture proposée permet de répondre aux lacunes de l'architecture logicielle actuellement utilisée. Cette nouvelle solution pourrait également permettre la réalisation efficace des concepts associés aux contrôleurs à architecture ouverte. Cependant, des tests supplémentaires seront nécessaires. Entre autre, les performances de l'architecture, bien que suffisantes théoriquement dans un contexte de production, devront être analysées de plus près avec de nouveaux scénarios. Le tout ne pourra cependant être fait que dans un contrôle réel de la machine-outil où la simulation fait place à une utilisation réelle de l'architecture. Un doctorat portant sur cette question est envisagé.

RECOMMANDATIONS

Pour donner suite au développement de l'architecture BNCL et à l'analyse de celle-ci, les recommandations suivantes peuvent être considérées :

a. créer un lien entre l'architecture BNCL et les logiciels de FAO, permettant à ceux-ci d'exploiter les concepts des contrôleurs à architecture ouverte;

b. réaliser un contrôleur de machine-outil ayant pour base l'architecture BNCL. Ce contrôleur permettrait la validation du point a) et l'usinage réel de pièces. Le comportement de l'architecture BNCL dans un contexte de production pourrait donc être évalué;

c. créer des compilateurs pour de nombreux langages de programmation, prouvant plus à fond les capacités multi langage de l'architecture;

d. tester la justesse des méthodes d'intégration présentées au Chapitre 5 dans un contexte réel de production. Ces tests pourraient prendre la forme d'intégration graduelle des méthodes, dans le processus de programmation, pour en arriver à terme à une intégration complète sur le contrôleur;

e. élaborer des scénarios d'usinage à la limite des capacités des machines-outils et analyser le comportant de l'architecture BNCL dans ce contexte. Ceci permettrait d'évaluer plus en détail les performances de l'architecture.

Toutes ces démarches permettraient de favoriser l'utilisation de l'architecture et d'en démontrer les avantages par rapport à la situation actuelle. Une commercialisation serait par la suite envisageable.

ANNEXE 1

AIDE-MÉMOIRE DES INSTRUCTIONS BNCL

Cette annexe dresse la liste complète des instructions compatibles avec la machine virtuelle BNCL. Ces instructions sont utilisées par l'assembleur BNCL pour produire un module exécutable. Elles sont également générées par le compilateur C pour l'architecture BNCL ou tout autre compilateur ciblant cette architecture.

abs.f	Real number absolute value	*Calculate the absolute value of a real number.*
acos.f	Inverse cosine	*Calculate the arc-cosine of an angle. Angles are always expressed in radians.*
add.f	Real addition	*Add two real numbers together.*
add.i	Integer addition	*Add two integer numbers together.*
add.bi	Big integer addition	*Add two 128-bits numbers together. Big integer are contained in 4 consecutive integer registers or in a 128-bits memory location.*
add.vf	Real vector addition	*Add two real vectors together. A real vector is contained in four consecutive real registers or in a 256-bits memory location.*
add.vi	Integer vector addition	*Add two integer vectors together. An integer vector is contained in four consecutive integer registers or in a 128-bits memory location.*
and.i	Logical AND	*Combine two integer with a logical AND. Result for each bit will be 1 only if two source bits are also set to 1.*
and.vi	Logical AND on integer vector	*Combine two integer vector with a logical AND. This results in a logical AND applied on 128-bits data. Result for each bit will be 1 only if two source bits are also set to 1.*
asin.f	Inverse sine	*Calculate the arc-sine of an angle. Angles are always expressed in radians.*
atan.f	Inverse tangent	
cadd.vf	Convert real vector to scalar with addition	
cadd.vi	Convert integer vector to scalar with addition	
call	Local procedure call	

cmp.f	Real comparison	
cmp.i	Integer comparison	
cmp.bi	Big integer comparison	
cos.f	Cosine	
dec.i	Decrementation	
div.f	Real division	
div.vf	Real division on real vector	
epil	Destroy stack frame in procedure exit	*Destroy the stack frame created by the prol instruction. The first operand contains the local stack pointer and the second operand contains the local frame pointer. Procedure is not exited and register windows are not affected. This procedure is used to speed up epilogue creation in GCC.*
icall	Call to interface	
in.f	Extract real from BNCL port	*Communication with standard ports BVH.*
in.i	Extract integer from BNCL port	*Communication with standard ports BVH.*
in.vf	Extract real vector from BNCL port	*Communication with standard ports BVH.*
in.vi	Extract integer vector from BNCL port	*Communication with standard ports BVH.*
inc.i	Incrementation	
ins.f	Extract real from specific port	*Communication with custom ports BVH.*
ins.i	Extract integer from specific port	*Communication with custom ports BVH.*
ins.vf	Extract real vector from specific port	*Communication with custom ports BVH.*
ins.vi	Extract integer vector from specific port	*Communication with custom ports BVH.*
j	Conditional jump	
jmp	Inconditional jump	

ld.f	Load real
ld.i	Load integer
ld.if	Convert real register value to integer register
ld.fi	Convert integer register value to real register
ld.qi	Load byte
ld.is	Load system register to integer register
ld.hi	Load half-word
ld.si	Load integer register to system register
ld.ss	Load system register to system register
ld.vf	Load real vector
ld.vi	Load integer vector
log.f	Base e logarithme
mcall	External procedure call
mmpl	Match module privilege level
mul.f	Real Multiplication
mul.vf	Real vector multiplication
neg.i	Two complement
neg.bi	Big integer two complement
neg.vi	Integer vector two complement
nop	No operation
not.i	Logical NOT
not.vi	Integer vector logical NOT

or.i	Logical OR	
or.vi	Integer vector logical OR	
isolb.i	Read byte from integer	
isolh.i	Read halfword from integer	
out.f	Send real to BNCL port	*Communication with standard ports BVH.*
out.i	Send integer to BNCL port	*Communication with standard ports BVH.*
out.vf	Send real vector to BNCL port	*Communication with standard ports BVH.*
out.vi	Send integer vector to BNCL port	*Communication with standard ports BVH.*
outs.f	Send real to specific port	*Communication with custom ports BVH.*
outs.i	Send integer to specific port	*Communication with custom ports BVH.*
outs.vf	Send real vector to specific port	*Communication with custom ports BVH.*
outs.vi	Send integer vector to specific port	*Communication with custom ports BVH.*
pop.f	Pop real from stack	
pop.i	Pop integer from stack	
pop.vf	Pop real vector from stack	
pop.vi	Pop integer vector from stack	
pow.f	Raise real number to power	
prol	Create stack frame in procedure entry	*Create a stack frame on procedure entry. The first operand contains the local stack pointer. The second operand contains the local frame pointer and operand three contains the size of the stack frame to allocate. This procedure is used to speed up prologue creation in GCC.*

push.f	Push real on stack
push.i	Push integer on stack
push.vf	Push real vector on stack
push.vi	Push integer vector on stack
ret	Procedure return
rnd.f	Rounding
rol.i	Left bit rotation
rol.bi	Big integer left rotation
rol.vi	Integer vector left rotation
ror.i	Right bit rotation
ror.bi	Big integer right rotation
ror.vi	Integer vector right rotation
rst.f	Restore real register window
rst.i	Restore integer register window
sar.i	Left shift
sar.vi	Integer vector right shift
sav.f	Save real register window
sav.i	Save integer register window
scall	System call
scnd	Set condition register
sdiv.i	Signed division
sdiv.bi	Big integer signed division

sdiv.vi	Integer vector signed division
shcdl	Left shift of condition register
shcdr	Right shift of condition register
shl.i	Left shift
shl.bi	Big integer left shift
shl.vi	Integer vector left shift
shr.i	Right shift
sar.bi	Big integer right shift
shr.vi	Integer vector right shift
sin.f	Sine
smul.i	Signed integer multiplication
smul.bi	Big integer signed multiplication
smul.vi	Integer vector signed multiplication
sqrt.f	Square root
sub.f	Real subtraction
sub.i	Integer subtraction
sub.bi	Big integer subtraction
sub.vf	Real vector subtraction
sub.vi	Integer vector subtraction
tan.f	Tangent
tjmp	Jump using dispatch table
trap	Call trap
udiv.i	Unsigned integer division

udiv.bi	Big integer unsigned division
udiv.vi	Integer vector unsigned division
umul.i	Unsigned integer multiplication
umul.bi	Big integer unsigned multiplication
umul.vi	Integer vector unsigned multiplication
xchg.i	Exchange integer registers values
xchg.f	Exchange real registers values
xor.i	Integer logical EXCLUSIVE OR
xor.vi	Integer vector logical EXCLUSIVE OR

ANNEXE 2

FICHIER DE CONFIGURATION DU TEST D'EXTENSIBILITÉ

La machine virtuelle BNCL nécessite un fichier de configuration pour fonctionner. Ce fichier contient : la liste des modules devant être chargés en mémoire lors du lancement de la machine virtuelle, la référence sur le répertoire de module devant être utilisé, les références sur les composantes logicielles devant contrôler les ports standard et les ports personnalisables, et les modules devant traiter les événements. La description du fichier de configuration a été faite à la section 3.3.2.

Fichier vmconfig_ThermocoupleReader.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bvm:configuration xmlns:bvm="urn:bvm:configuration">
	<bvm:input-output>
		<bvm:std-iospace
			uuid="{1E174F77-A623-464C-886C-BCAB07D9B8CC}"/>
		<bvm:custom-iospace
			uuid="{1ADCD2F7-5F77-4D39-B1DE-835D782379CD}"/>
	</bvm:input-output>

	<bvm:repositories>
		<bvm:repository
			id="MAIN_REPOSITORY"
			uuid="{22AA1401-2697-4209-BCB5-EB732D634A29}"/>
	</bvm:repositories>

	<bvm:exceptions>
		<bvm:exception
			vector="64"
			module="MODULE_NATIVE"
			method="interruptHandler"/>
		<bvm:exception
			vector="65"
			module="MODULE_THERMOCOUPLE_READER"
			method="temperatureTrigger"/>
	</bvm:exceptions>

	<bvm:sysmodules main="MODULE_SYSTEM">
		<bvm:module
			id="MODULE_NATIVE"
			uuid="{5C08285F-6F20-4179-B697-0FB5AF0D96CE}"/>
		<bvm:module
			id="MODULE_SYSTEM"
			uuid="{7ECA1944-9896-48C2-8047-010140A1B58A}"/>
		<bvm:module
			id="MODULE_THERMOCOUPLE_READER"
			uuid="{D5658875-48FB-415c-BE03-670D811F83DB}"/>
	</bvm:sysmodules>
</bvm:configuration>
```

Les éléments `<bvm:std-iospace>` et `<bvm:custom-iospace>` permettent de définir les composantes logicielles qui doivent contrôler l'accès aux ports BNCL. Dans le fichier de configuration utilisé pour le test d'extensibilité, la référence `{1E174F77-A623-464C-886C-BCAB07D9B8CC}` pointe sur l'objet COM *ThermocoupleReaderBVH.*

L'élément `<bvm:sysmodules>` contient les références sur les modules systèmes. Ces références pointent sur des modules BNCL dans le *BnclRepository*. Le répertoire de modules utilisé est définie, pour sa part, par l'élément `<bvm:repository>`. Finalement, les gestionnaires d'événements sont définies par l'élément `<bvm:exceptions>`. C'est à cet endroit que la fonction *temperatureTrigger*, exécutée lorsque la température dépasse le seuil acceptable, est associée à l'événement correspondant.

ANNEXE 3

DEFINITIONS LIÉES AU PROGRAMME CONSOLEBVM

Cette annexe contient les fichiers d'entête C++ (*header files*) utilisés dans la génération du code d'implémentation de la machine virtuelle BNCL utilisée lors des tests de validation. Ces fichiers sont pour la plupart générés automatiquement par l'utilitaire FVMgen à partir du fichier de description de l'architecture BNCL. Ce fichier de description n'est pas montré ici.

Fichier ConsoleBVM.h

```
////

#ifndef _CONSOLEBVM_H_
#define _CONSOLEBVM_H_


#include "stdafx.h"


#include "cmdline.h"
#include "msxml2.h"
#include "fvmgen.h"
#include <exception>



#define             DEFAULT_CONFIG_FILE             "vmconfig.xml"

////////////////////////////////////////////////////////////////////////////////
////
//// Functions
////
////////////////////////////////////////////////////////////////////////////////

bool checkOptions(CommandLineProperties& clCmdLine,
                  IXMLDOMDocument2** pConfiguration);
void startVirtualMachine(CommandLineProperties& clCmdLine);
void initVirtualMachine(CommandLineProperties& clCmdLine,
                        IXMLDOMDocument2* pConfiguration,
                        fvmgen::VirtualMachine& vm);
void printHeader();



////////////////////////////////////////////////////////////////////////////////
////
//// Interfaces
////
////////////////////////////////////////////////////////////////////////////////
```

```
/**/
// <summary>
// Interface de validation d'un fichier XML.
// </summary>
// <remarks>
// Une classe implémentant la validation d'un fichier XML implémente
// cette interface. Par exemple une classe XmlDtdValidator pourrait
// implémenter une validation à l'aide d'un DTD. XmlXslValidator
// pourrait implémenter une validation à l'aide d'une page XSL. Etc...
// </remarks>
struct IXmlValidator {
      /**/
      // <summary>
      // Déterminer si le validateur est connecté à un schéma XML ou
      // non.
      // </summary>
      // <returns>Retourne true si l'objet est connecté à un schéma et
      // false sinon.
      // </returns>
      virtual bool isConnected() throw() = 0;

      /**/
      // <summary>
      // Valider un fichier XML par rapport au schéma associé à l'objet
      // XmlSchemaValidator.
      // </summary>
      // <remarks>
      // Cette méthode retourne un pointeur sur une interface
      // IXMLDOMDocument2 qui pointe sur le document XML spécifié
      // chargé en mémoire et validé.
      // </remarks>
      // <param name="rsUrl">URL spécifiant où se trouve le fichier XML
      // à valider.</param>
      // <param name="ppXmlDocument">Endroit où placer la valeur du
      // pointeur sur l'interface IXMLDOMDocument2.</param>
      // <returns>Retourne true si le fichier XML est valide et false
      // sinon.</returns>
      virtual bool validate(const std::string& rsUrl,
                            IXMLDOMDocument2** ppXmlDocument)
            throw() = 0;

};

/////////////////////////////////////////////////////////////////////////
////
//// Classes
////
/////////////////////////////////////////////////////////////////////////

/**/
// <summary>
```

```
// Classe de validation de fichier XML à l'aide d'un schéma XSD.
// </summary>
class XmlSchemaValidator : public IXmlValidator {
private:

	/**/
	// <summary>Pointeur sur l'interface d'accès au schéma.</summary>
	IXMLDOMSchemaCollection2* pXMLSchemaM;


public:

	/**/
	XmlSchemaValidator(const std::string& rsSchemaUrl,
	                   const std::string& rsNamespaceUri);

	/**/
	~XmlSchemaValidator();

	// Interface IXmlValidator
	/**/
	virtual bool isConnected() throw();

	/**/
	virtual bool validate(const std::string& rsUrl,
	                      IXMLDOMDocument2** ppXmlDocument)
	            throw();

};



/**/
class InterruptHandler : public fvmgen::IBVMExceptionEvent {
private:

	fvmgen::VirtualMachine* pVirtualMachine;


public:

	InterruptHandler(fvmgen::VirtualMachine* pVM) {
		pVirtualMachine = pVM;
	}


	// Interface fvmgen::IBVMExceptionEvent
	virtual HRESULT __stdcall send(int iExceptionVector) {
		::InterlockedExchange(
			(LPLONG)&pVirtualMachine->pending,
			iExceptionVector);
		::WaitForSingleObject(pVirtualMachine->synchro, INFINITE);
```

```
        return S_OK;
    }

    // Interface IUnknown

    virtual ULONG __stdcall AddRef() {
        return 1;
    }

    virtual ULONG __stdcall Release() {
        return 1;
    }

    virtual HRESULT __stdcall QueryInterface(
            REFIID iid, void ** ppvObject) {
        if(IsEqualGUID(iid, __uuidof(fvmgen::IBVMExceptionEvent))) {
            *ppvObject = (void*)this;
            return S_OK;
        } else {
            return E_NOINTERFACE;
        }
    }

    // Interface IDispatch

    virtual HRESULT STDMETHODCALLTYPE Invoke(
            /* [in] */ DISPID dispIdMember,
            /* [in] */ REFIID riid,
            /* [in] */ LCID lcid,
            /* [in] */ WORD wFlags,
            /* [out][in] */ DISPPARAMS *pDispParams,
            /* [out] */ VARIANT *pVarResult,
            /* [out] */ EXCEPINFO *pExcepInfo,
            /* [out] */ UINT *puArgErr)
{
    (void) riid;
    (void) dispIdMember;
    (void) lcid;
    (void) wFlags;
    (void) pExcepInfo;
    (void) puArgErr;
    HRESULT hr = S_OK;
    if (pDispParams == 0) {
        return DISP_E_BADVARTYPE;
    }
    if (pDispParams->cArgs > 1) {
        return DISP_E_BADPARAMCOUNT;
    }
    if (pVarResult != 0) {
        ::VariantInit(pVarResult);
    }
```

```
    VARIANT* rgpVars[1];
    UINT index = 0;
    for (; index < pDispParams->cArgs; ++index) {
        rgpVars[index] = &pDispParams->rgvarg[index];
    }
    VARIANT v0;
    VARIANT* v;
    switch (dispIdMember) {
    case 1:
        {
            if (pDispParams->cArgs != 1) {
                return DISP_E_BADPARAMCOUNT;
            }
            v = rgpVars[0];
            if(v->vt
               !=
               VT_I4
               &&
               FAILED(VariantChangeType(v, &v0, 0, VT_I4))) {
                if (puArgErr != 0) {
                    *puArgErr = 0;
                }
                return DISP_E_TYPEMISMATCH;
            }
            int i1 = (int)v->lVal;
            hr = ((::fvmgen::IBVMExceptionEvent*) this)->send(i1);
            break;
        }
    default:
        return DISP_E_MEMBERNOTFOUND;
    }

    return hr;
}

virtual HRESULT STDMETHODCALLTYPE GetIDsOfNames(
            /* [in] */ REFIID riid,
            /* [size_is][in] */ LPOLESTR *rgszNames,
            /* [in] */ UINT cNames,
            /* [in] */ LCID lcid,
            /* [size_is][out] */ DISPID *rgDispId)
{
    (void) riid;
    (void) rgszNames;
    (void) cNames;
    (void) lcid;
    (void) rgDispId;
    static LPOLESTR names[] = { L"send" };
    static DISPID dids[] = { 1 };
    for(unsigned int i = 0; i < cNames; ++i) {
        int fFoundIt = 0;
        for(unsigned int j = 0;
            j < sizeof(names)/sizeof(LPOLESTR);
```

```
                ++j)
            {
                if (lstrcmpiW(rgszNames[i], names[j]) == 0) {
                    fFoundIt = 1;
                    rgDispId[i] = dids[j];
                    break;
                }
            }
            if (fFoundIt == 0) {
                return DISP_E_UNKNOWNNAME;
            }
        }
        return S_OK;
    }

    virtual HRESULT STDMETHODCALLTYPE GetTypeInfoCount(
                unsigned int*  pctinfo) {
        if (pctinfo == NULL) {
            return E_POINTER;
        }

        *pctinfo = 0;

        return S_OK;
    }

    virtual HRESULT STDMETHODCALLTYPE GetTypeInfo(
                unsigned int iTInfo, LCID lcid, ITypeInfo** ppTInfo) {
        return E_FAIL;
    }

};



////////////////////////////////////////////////////////////////////////
////
//// Exceptions
////
////////////////////////////////////////////////////////////////////////


/**/
class ConfigFileNotFoundException : public std::exception {
};

/**/
class ConfigFileInvalidException : public std::exception {
};

/**/
```

```
class RepositoryNotFoundException : public std::exception {
};

/**/
class MsxmlNotPresentException : public std::exception {
};

/**/
class BnclVirtualHardwareNotFoundException : public std::exception {
};

/**/
class MethodIndexInvalidException : public std::exception {
};

/**/
class MethodTypeInvalidException : public std::exception {
};

/**/
class ModuleNotFoundException : public std::exception {
};


#endif
```

Fichier cmdline.h

```
//#include "stdafx.h"

#include <map>
#include <vector>
#include <string>
#include <exception>


////////////////////////////////////////////////////////////////////////////
//
// Structures
//
////////////////////////////////////////////////////////////////////////////


/**
 *
 */
struct SCommandLine {
```

```
	int		argc;
	char**		argv;

	SCommandLine(int count, char** psarg)
		{ argc = count; argv = psarg; }
	~SCommandLine() { /*free(argv);*/ }
};


////////////////////////////////////////////////////////////////////////
//
// Fonctions
//
////////////////////////////////////////////////////////////////////////


/**
 *
 */
SCommandLine extractCommandLine(char* command_line);



////////////////////////////////////////////////////////////////////////
//
// Classes
//
////////////////////////////////////////////////////////////////////////

#define			COMMANDLINE_PARAMETER_SET
	"COMMANDLINE_PARAMETER_SET"


/**
 *
 */
class CommandLineProperties {
private:

	std::vector<std::string> clArgumentsM;
	std::map<std::string, std::string> clParametersM;


public:

	CommandLineProperties(char* pacCmdLine);
```

```
        CommandLineProperties(SCommandLine& rstCommandLine);
        ~CommandLineProperties();

        int getArgumentsSize();
        int getParametersSize();

        std::string getArgument(int iIndex);

        std::string getParameter(const char* pacParameter);
        std::string getParameter(std::string& rsParameter);
};
```

Fichier fvmgen.h

```
/**/
// Definition of structures, classes and types for the
// virtual machine code generated by fvmgen.
//
//
//

#ifndef _FVMGEN_H
#define _FVMGEN_H


#include <cmath>
#include <exception>
#include <iostream>
#include <string>


#define           REGISTER_sfr_CF   0
#define           REGISTER_sfr_OF   1
#define           REGISTER_sfr_SF   2
#define           REGISTER_sfr_ZF   3
#define           REGISTER_sfr_NZ   4
#define           REGISTER_sfr_ND   5
#define           REGISTER_sfr_NU   6
#define           REGISTER_sfr_NO   7
#define           REGISTER_sfr_NP   8
#define           REGISTER_sfr_NI   9
#define           REGISTER_sfr_UCD1 10
#define           REGISTER_sfr_UCD2 11
#define           REGISTER_sfr_UCD3 12
#define           REGISTER_sfr_CT   28
#define           REGISTER_sfr_TF   31
//#include "stdafx.h"

#include "atlbase.h"
#include "atlcomcli.h"
#include "comutil.h"
```

```
#include "atlsafe.h"
#include "atlmem.h"

#include <vector>

//#include "gmp.h"

#pragma warning( disable : 4312 4101 4244 4311 4018 4307)


#define          BNCL_STACK_SIZE              (1024*1024)/4
#define          BNCL_NUMBER_WINDOWS          20
#define          BNCL_WINDOW_SIZE             32
#define          BNCL_WINDOW_INREG_SIZE       8
#define          BNCL_WINDOW_LOCALREG_SIZE    16
#define          BNCL_WINDOW_OUTREG_SIZE      8

#define          PTR_BNCL_TO_NATIVE(src)      (src)

#define          PTR_NATIVE_TO_BNCL(src)      (src)


namespace fvmgen {



struct SVirtualMachineState;

/**/
// Integral type
typedef          int          int32;
/**/
// Floating-point type.
typedef          double       float64;
/**/
// Target to BNCL code.
typedef int32* TCodeTarget;
/**/
// Exception vector type.
typedef int32 TExceptionVector;


/**/
// <summary>
// Structure containing the various registers of the architecture.
// </summary>
struct SRegisters {
    int32*  integer;
    float64*     real;
    int32*  system;

};
```

```
enum ERegisters_integer {
    reg_gr0=0,
    reg_pgr0=0,
    reg_bi0=0,
    reg_gr1=1,
    reg_gr2=2,
    reg_gr3=3,
    reg_gr4=4,
    reg_pgr1=4,
    reg_bi1=4,
    reg_gr5=5,
    reg_gr6=6,
    reg_gr7=7,
    reg_gr8=8,
    reg_pgr2=5,
    reg_bi2=5,
    reg_gr9=9,
    reg_gr10=10,
    reg_gr11=11,
    reg_gr12=12,
    reg_pgr3=12,
    reg_bi3=12,
    reg_gr13=13,
    reg_gr14=14,
    reg_gr15=15,
    reg_gr16=16,
    reg_pgr4=16,
    reg_bi4=16,
    reg_gr17=17,
    reg_gr18=18,
    reg_gr19=19,
    reg_gr20=20,
    reg_pgr5=20,
    reg_bi5=20,
    reg_gr21=21,
    reg_gr22=22,
    reg_gr23=23,
    reg_gr24=24,
    reg_pgr6=24,
    reg_bi6=24,
    reg_gr25=25,
    reg_gr26=26,
    reg_gr27=27,
    reg_gr28=28,
    reg_pgr7=28,
    reg_bi7=28,
    reg_gr29=29,
    reg_gr30=30,
    reg_gr31=31
};
```

```
enum ERegisters_real {
    reg_fr0=0,
    reg_pfr0=0,
    reg_fr1=1,
    reg_fr2=2,
    reg_fr3=3,
    reg_fr4=4,
    reg_pfr1=4,
    reg_fr5=5,
    reg_fr6=6,
    reg_fr7=7,
    reg_fr8=8,
    reg_pfr2=8,
    reg_fr9=9,
    reg_fr10=10,
    reg_fr11=11,
    reg_fr12=12,
    reg_pfr3=12,
    reg_fr13=13,
    reg_fr14=14,
    reg_fr15=15,
    reg_fr16=16,
    reg_pfr4=16,
    reg_fr17=17,
    reg_fr18=18,
    reg_fr19=19,
    reg_fr20=20,
    reg_pfr5=20,
    reg_fr21=21,
    reg_fr22=22,
    reg_fr23=23,
    reg_fr24=24,
    reg_pfr6=24,
    reg_fr25=25,
    reg_fr26=26,
    reg_fr27=27,
    reg_fr28=28,
    reg_pfr7=28,
    reg_fr29=29,
    reg_fr30=30,
    reg_fr31=31
};


enum ERegisters_system {
    reg_sr0=0,
    reg_str=0,
    reg_sr1=1,
    reg_gpr=1,
```

```
    reg_sr2=2,
    reg_lpr=2,
    reg_sr3=3,
    reg_epr=3,
    reg_sr4=4,
    reg_gl0=4,
    reg_sr5=5,
    reg_gl1=5,
    reg_sr6=6,
    reg_gl2=6,
    reg_sr7=7,
    reg_gl3=7,
    reg_sr8=8,
    reg_gl4=8,
    reg_sr9=9,
    reg_gl5=9,
    reg_sr10=10,
    reg_gl6=10,
    reg_sr11=11,
    reg_gl7=11,
    reg_sr12=12,
    reg_smr=12,
    reg_sr13=13,
    reg_sr14=14,
    reg_sr15=15,
    reg_sr16=16,
    reg_ipr=17,
    reg_sr17=17,
    reg_sr18=18,
    reg_sr19=19,
    reg_sr20=20,
    reg_sfr=20,
    reg_sr21=21,
    reg_sr22=22,
    reg_sr23=23,
    reg_sr24=24,
    reg_sr25=25,
    reg_sr26=26,
    reg_sr27=27,
    reg_sr28=28,
    reg_sr29=29,
    reg_sr30=30,
    reg_sr31=31
};


/* HASH: Uncomment and replace with correct hash function supplied by
         Perfect Hash Function generator. The values supplied
         here are for example only. By default, no hash is
         made. The formats switch are then unoptimized. */
```

```
/*#define FVMGEN_FORMAT_HASHING*/

#ifdef FVMGEN_FORMAT_HASHING
typedef  unsigned long  int  ub4;
#define HASH_INT(v) (((v >> 10) & 1) | ((v >> 26) & 62))
#define HASH(v, rsl)    (rsl = (((v >> 10) & 1) | ((v >> 26) & 62)));
#else
#define     HASH_INT(v) (v)
#define HASH(v, rsl) (rsl = v);
#endif




////////////////////////////////////////////////////////////////////////////////
////
//// Exceptions
////
////////////////////////////////////////////////////////////////////////////////


class ObjectRequestBrokerException : public std::exception {
private:

    std::string                 sReason;


public:

    ObjectRequestBrokerException() {
        sReason = "Uncatchable ObjectRequestBrokerException : ";
    }
    ObjectRequestBrokerException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable ObjectRequestBrokerException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class InterfaceRequestBrokerException : public std::exception {
private:

    std::string                 sReason;


public:
```

```
    InterfaceRequestBrokerException() {
        sReason = "Uncatchable InterfaceRequestBrokerException : ";
    }
    InterfaceRequestBrokerException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable InterfaceRequestBrokerException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class MachineShutdownException : public std::exception {
private:

    std::string             sReason;


public:

    MachineShutdownException() {
        sReason = "Uncatchable MachineShutdownException : ";
    }
    MachineShutdownException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable MachineShutdownException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class InvalidAddressException : public std::exception {
private:

    std::string             sReason;


public:

    InvalidAddressException() {
        sReason = "Uncatchable InvalidAddressException : ";
    }
    InvalidAddressException(const char* reason) {
        using namespace std;
```

```
        sReason = "Uncatchable InvalidAddressException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class UnknownInstructionException : public std::exception {
private:

    std::string                 sReason;


public:

    UnknownInstructionException() {
        sReason = "Uncatchable UnknownInstructionException : ";
    }
    UnknownInstructionException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable UnknownInstructionException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class StackUnderflowException : public std::exception {
private:

    std::string                 sReason;


public:

    StackUnderflowException() {
        sReason = "Uncatchable StackUnderflowException : ";
    }
    StackUnderflowException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable StackUnderflowException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
```

```
};

class StackOverflowException : public std::exception {
private:

    std::string                 sReason;


public:

    StackOverflowException() {
        sReason = "Uncatchable StackOverflowException : ";
    }
    StackOverflowException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable StackOverflowException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class RegisterWindowUnderflowException : public std::exception {
private:

    std::string                 sReason;


public:

    RegisterWindowUnderflowException() {
        sReason = "Uncatchable RegisterWindowUnderflowException : ";
    }
    RegisterWindowUnderflowException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable RegisterWindowUnderflowException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class RegisterWindowOverflowException : public std::exception {
private:

    std::string                 sReason;
```

```
public:

    RegisterWindowOverflowException() {
        sReason = "Uncatchable RegisterWindowOverflowException : ";
    }
    RegisterWindowOverflowException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable RegisterWindowOverflowException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class InvalidRegisterUseException : public std::exception {
private:

    std::string             sReason;


public:

    InvalidRegisterUseException() {
        sReason = "Uncatchable InvalidRegisterUseException : ";
    }
    InvalidRegisterUseException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable InvalidRegisterUseException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class UnsupportedFeatureException : public std::exception {
private:

    std::string             sReason;


public:

    UnsupportedFeatureException() {
        sReason = "Uncatchable UnsupportedFeatureException : ";
    }
    UnsupportedFeatureException(const char* reason) {
```

```
        using namespace std;

        sReason = "Uncatchable UnsupportedFeatureException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class InvalidModuleIndexException : public std::exception {
private:

    std::string             sReason;


public:

    InvalidModuleIndexException() {
        sReason = "Uncatchable InvalidModuleIndexException : ";
    }
    InvalidModuleIndexException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable InvalidModuleIndexException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class InvalidMethodIndexException : public std::exception {
private:

    std::string             sReason;


public:

    InvalidMethodIndexException() {
        sReason = "Uncatchable InvalidMethodIndexException : ";
    }
    InvalidMethodIndexException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable InvalidMethodIndexException : ";
        sReason += reason;
    }

    const char *what() const throw() {
```

```
        return sReason.c_str();
    }
};

class PrivilegeViolationException : public std::exception {
private:

    std::string                 sReason;


public:

    PrivilegeViolationException() {
        sReason = "Uncatchable PrivilegeViolationException : ";
    }
    PrivilegeViolationException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable PrivilegeViolationException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class ZeroDivideException : public std::exception {
private:

    std::string                 sReason;


public:

    ZeroDivideException() {
        sReason = "Uncatchable ZeroDivideException : ";
    }
    ZeroDivideException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable ZeroDivideException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class NumericZeroDivideException : public std::exception {
private:
```

```
    std::string               sReason;


public:

    NumericZeroDivideException() {
        sReason = "Uncatchable NumericZeroDivideException : ";
    }
    NumericZeroDivideException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable NumericZeroDivideException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class NumericDenormalException : public std::exception {
private:

    std::string               sReason;


public:

    NumericDenormalException() {
        sReason = "Uncatchable NumericDenormalException : ";
    }
    NumericDenormalException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable NumericDenormalException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class NumericUnderflowException : public std::exception {
private:

    std::string               sReason;


public:

    NumericUnderflowException() {
        sReason = "Uncatchable NumericUnderflowException : ";
```

```
    }
    NumericUnderflowException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable NumericUnderflowException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class NumericOverflowException : public std::exception {
private:

    std::string              sReason;


public:

    NumericOverflowException() {
        sReason = "Uncatchable NumericOverflowException : ";
    }
    NumericOverflowException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable NumericOverflowException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class NumericPrecisionException : public std::exception {
private:

    std::string              sReason;


public:

    NumericPrecisionException() {
        sReason = "Uncatchable NumericPrecisionException : ";
    }
    NumericPrecisionException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable NumericPrecisionException : ";
        sReason += reason;
    }
```

```
    const char *what() const throw() {
        return sReason.c_str();
    }
};

class NumericInvalidNumberException : public std::exception {
private:

    std::string                 sReason;


public:

    NumericInvalidNumberException() {
        sReason = "Uncatchable NumericInvalidNumberException : ";
    }
    NumericInvalidNumberException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable NumericInvalidNumberException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class NumericGeneralExceptionException : public std::exception {
private:

    std::string                 sReason;


public:

    NumericGeneralExceptionException() {
        sReason = "Uncatchable NumericGeneralExceptionException : ";
    }
    NumericGeneralExceptionException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable NumericGeneralExceptionException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class NumericEmulationException : public std::exception {
```

```
private:

    std::string                sReason;


public:

    NumericEmulationException() {
        sReason = "Uncatchable NumericEmulationException : ";
    }
    NumericEmulationException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable NumericEmulationException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class DebugExceptionException : public std::exception {
private:

    std::string                sReason;


public:

    DebugExceptionException() {
        sReason = "Uncatchable DebugExceptionException : ";
    }
    DebugExceptionException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable DebugExceptionException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

class UserExceptionException : public std::exception {
private:

    std::string                sReason;


public:
```

```
    UserExceptionException() {
        sReason = "Uncatchable UserExceptionException : ";
    }
    UserExceptionException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable UserExceptionException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};

enum EIOSpaces {
    iospace_TARGET_INDEPENDANT=0, iospace_TARGET_DEPENDANT=1
};


//////////////////////////////////////////////////////////////////////
////
//// Types definitions
////
//////////////////////////////////////////////////////////////////////

/**/
// BNCL instruction opcode.
typedef int32 TOpcode;
/**/
// BNCL module static data target.
typedef int32* TDataTarget;
/**/
// BNCL module index.
typedef int32 TModuleIndex;
/**/
// BNCL method index.
typedef int32 TMethodIndex;
/**/
// BNCL method block index;
typedef int32 TCodeIndex;

/**/
// Enumeration used for method flags.
enum EMethodFlags {
    /**/
    // Method is of type BNCL.
    TYPE_CODE_BNCL = 1,
    /**/
    // Method is of type NATIVE. Native call point to code compiled
    // for the current architcture.
    TYPE_CODE_NATIVE = 2,
    /**/
```

```
    // Method is not valid.
    TYPE_CODE_INVALID = 31
};

/**/
// Enumeration used for module flags.
enum EModuleFlags {
    /**/
    // Module has USER access.
    TYPE_MODULE_USER = 1,
    /**/
    // Module has SUPERVISOR access.
    TYPE_MODULE_SUPERVISOR = 2,
    /**/
    // Module has an unknown access. Typically, the module is not
    // accessible for security reasons.
    TYPE_MODULE_UNKNOWN = 31,
    /**/
    // Module index is not associated with a module.
    MODULE_INVALID = 0xFFFFFFFF
};


/**/
typedef void (*FNativeCall)(SVirtualMachineState* vm);

struct SModule;

/**/
typedef std::vector<SModule*>              TModuleVector;


/**/
// <summary>
// Union for a method table target. A target is a function in BNCL
// or in a DLL.
// (@author = fvmgen-extension)
// </summary>
union UTarget {
    /**/
    // The target is a native method. Value is a full 32-bits
    // address to the function.
    FNativeCall          native;
    /**/
    // The target is a BNCL method. Value is an index
    // into the method block segment.
    TCodeIndex           bncl;
};


//////////////////////////////////////////////////////////////////////////
////////////
```

```
////
//// COM Interfaces
////
///////////////////////////////////////////////////////////////////////
////////////

// IBVHCore
MIDL_INTERFACE("587A2F44-8CE5-44B3-8C6A-B628FD3380FC")
IBVHCore :
public IDispatch
{
    public:
    virtual HRESULT STDMETHODCALLTYPE get_Visual(
                        IDispatch **pVal) = 0;

    virtual HRESULT STDMETHODCALLTYPE get_PortsAccessor(
                        IDispatch **pVal) = 0;

    virtual HRESULT STDMETHODCALLTYPE get_TargetIndustry(
                        BSTR *pVal) = 0;

    virtual HRESULT STDMETHODCALLTYPE get_OperationsType(
                        BSTR *pVal) = 0;

    virtual HRESULT STDMETHODCALLTYPE get_VirtualHardwareType(
                        BSTR *pVal) = 0;

    virtual HRESULT STDMETHODCALLTYPE get_TargetIndustryDescription(
                        BSTR *pVal) = 0;

    virtual HRESULT STDMETHODCALLTYPE get_OperationsTypeDescription(
                        BSTR *pVal) = 0;

    virtual HRESULT STDMETHODCALLTYPE
                              get_VirtualHardwareTypeDescription(

                        BSTR *pVal) = 0;

    virtual HRESULT STDMETHODCALLTYPE get_Author(
                        BSTR *pVal) = 0;

    virtual HRESULT STDMETHODCALLTYPE get_Version(
                        BSTR *pVal) = 0;

    virtual HRESULT STDMETHODCALLTYPE get_Description(
                        BSTR *pVal) = 0;

    virtual HRESULT STDMETHODCALLTYPE get_PortMapType(
                        BSTR *pVal) = 0;

    virtual HRESULT STDMETHODCALLTYPE get_PortMapTypeDescription(
                        BSTR *pVal) = 0;
```

```
};

// IBVHPortsAccessor
MIDL_INTERFACE("2595CBF3-C741-4d07-AA05-0D72433AD5D0")
IBVHPortsAccessor :
public IDispatch
{
    public:
    virtual HRESULT STDMETHODCALLTYPE writeInteger(
                        int iPort,
                        int iValue) = 0;

    virtual HRESULT STDMETHODCALLTYPE writeReal(
                        int iPort,
                        double dValue) = 0;

    virtual HRESULT STDMETHODCALLTYPE writeVInteger(
                        int iPort,
                        int *paiVector) = 0;

    virtual HRESULT STDMETHODCALLTYPE writeVReal(
                        int iPort,
                        double *padVector) = 0;

    virtual HRESULT STDMETHODCALLTYPE readInteger(
                        int iPort,
                        int *piValue) = 0;

    virtual HRESULT STDMETHODCALLTYPE readReal(
                        int iPort,
                        double *pdValue) = 0;

    virtual HRESULT STDMETHODCALLTYPE readVInteger(
                        int iPort,
                        int *paiVector) = 0;

    virtual HRESULT STDMETHODCALLTYPE readVReal(
                        int iPort,
                        double *padVector) = 0;
};

MIDL_INTERFACE("19BD7F16-6515-4ABC-B80E-6D74A52605CA")
IBVMExceptionEvent :
public IDispatch
{
```

```
    public:
    virtual HRESULT STDMETHODCALLTYPE send(
                        int iExceptionVector) = 0;

};




/**/
// <summary>
// Interface de base de l'objet COM BnclRepository. Cette interface
// peut être utilisée par un script puisqu'elle dérive de IDispatch.
// </summary>
MIDL_INTERFACE("EA41C4C3-2320-4E95-B02D-2BA50661A3DF")
IBnclRepositoryComponent :
public IDispatch
{
    public:
    virtual HRESULT STDMETHODCALLTYPE get_modules(
                        VARIANT *pvModules) = 0;

    virtual HRESULT STDMETHODCALLTYPE get_namespaces(
                        VARIANT *pvNamespaces) = 0;

    virtual HRESULT STDMETHODCALLTYPE get_interfaces(
                        VARIANT *pvInterfaces) = 0;

    virtual HRESULT STDMETHODCALLTYPE get_tags(
                        VARIANT *pvTags) = 0;

    virtual HRESULT STDMETHODCALLTYPE load(
                        BSTR sModuleUuid,
                        VARIANT *pvModule) = 0;

    virtual HRESULT STDMETHODCALLTYPE loadFromFile(
                        BSTR sModulePath,
                        VARIANT *pvModule) = 0;

    virtual HRESULT STDMETHODCALLTYPE isModulePresent(
                        BSTR sModuleUuid,
                        VARIANT_BOOL *pbPresent) = 0;

    virtual HRESULT STDMETHODCALLTYPE isNamespacePresent(
                        BSTR sNamespaceUuid,
                        VARIANT_BOOL *pbPresent) = 0;

    virtual HRESULT STDMETHODCALLTYPE isInterfacePresent(
                        BSTR sInterfaceUuid,
                        VARIANT_BOOL *pbPresent) = 0;

    virtual HRESULT STDMETHODCALLTYPE isInterfaceImplementedBy(
```

```
                        BSTR sModuleUuid,
                        BSTR sInterfaceUuid,
                        VARIANT_BOOL *pbImplemented) = 0;

virtual HRESULT STDMETHODCALLTYPE isInterfacePartOf(
                        BSTR sNamespaceUuid,
                        BSTR sInterfaceUuid,
                        VARIANT_BOOL *pbPartOf) = 0;


virtual HRESULT STDMETHODCALLTYPE findInterfaceIndex(
                        BSTR sModuleUuid,
                        BSTR sInterfaceUuid,
                        int *piIndex) = 0;

virtual HRESULT STDMETHODCALLTYPE findInterfaceIndexFromFile(
                        BSTR sModuleFilename,
                        BSTR sInterfaceUuid,
                        int *piIndex) = 0;

virtual HRESULT STDMETHODCALLTYPE findInterfaceNamespace(
                        BSTR sInterfaceUuid,
                        BSTR *psNamespaceUuid) = 0;

virtual HRESULT STDMETHODCALLTYPE findModuleUuid(
                        BSTR sModuleName,
                        BSTR *psModuleUuid) = 0;

virtual HRESULT STDMETHODCALLTYPE findNamespaceUuid(
                        BSTR sNamespaceName,
                        BSTR *psNamespaceUuid) = 0;

virtual HRESULT STDMETHODCALLTYPE findInterfaceUuid(
                        BSTR sNamespaceUuid,
                        BSTR sInterfaceName,
                        BSTR *psInterfaceUuid) = 0;

virtual HRESULT STDMETHODCALLTYPE findModuleName(
                        BSTR sModuleUuid,
                        BSTR *psModuleName) = 0;

virtual HRESULT STDMETHODCALLTYPE findNamespaceName(
                        BSTR sNamespaceUuid,
                        BSTR *psNamespaceName) = 0;

virtual HRESULT STDMETHODCALLTYPE findInterfaceName(
                        BSTR sInterfaceUuid,
                        BSTR *psInterfaceName) = 0;

virtual HRESULT STDMETHODCALLTYPE findMethodIndex(
                        BSTR sModuleUuid,
                        BSTR sMethodName,
                        int *piIndex) = 0;
```

```
    virtual HRESULT STDMETHODCALLTYPE findMethodIndexFromFile(
                        BSTR sModuleFilename,
                        BSTR sMethodName,
                        int *piIndex) = 0;

};


///////////////////////////////////////////////////////////////////////////
////
//// Structures
////
///////////////////////////////////////////////////////////////////////////


/**/
struct SInteger128 {
    //constructors / destructors
    SInteger128();
    SInteger128(int val);
    SInteger128(int32* pVal);
    SInteger128(const SInteger128& val);

    //assignment operator
    SInteger128& operator=(const SInteger128& val);

    //arithmetic operators
    SInteger128 operator+(const SInteger128& val);
    SInteger128 operator-(const SInteger128& val);
    friend SInteger128 operator-(const SInteger128& val);
    SInteger128& operator++();
    SInteger128& operator--();
    SInteger128  operator*(const SInteger128& val) const;
    SInteger128  operator/(const SInteger128& val) const;
    SInteger128  umul(const SInteger128& val) const;
    SInteger128  udiv(const SInteger128& val) const;
    SInteger128  operator%(const SInteger128& val) const;
    SInteger128& operator*=(const SInteger128& val);
    SInteger128& operator/=(const SInteger128& val);
    SInteger128& operator+=(const SInteger128& val);
    SInteger128& operator-=(const SInteger128& val);
    SInteger128& operator%=(const SInteger128& val);
    SInteger128  operator~() const;

    //equality operators
    bool operator==(const SInteger128& val) const;
    bool operator!=(const SInteger128& val) const;
    bool operator>(const SInteger128& val) const;
    bool operator>=(const SInteger128& val) const;
    bool operator<(const SInteger128& val) const;
    bool operator<=(const SInteger128& val) const;
```

```
    //Misc operators
    SInteger128  operator>>(int nShift) const;
    SInteger128  operator<<(int nShift) const;
    SInteger128& operator>>=(int nShift);
    SInteger128& operator<<=(int nShift);
    SInteger128  operator^(const SInteger128& val) const;
    SInteger128  operator|(const SInteger128& val) const;
    SInteger128  operator&(const SInteger128& val) const;
    SInteger128& operator^=(const SInteger128& val);
    SInteger128& operator|=(const SInteger128& val);
    SInteger128& operator&=(const SInteger128& val);


    //Misc. Functions
    bool GetBit(int nIndex) const;
    void SetBit(int nIndex, bool val);
    bool IsZero() const;
    void Zero();
    void Negate();
    bool IsNegative() const;
    bool IsPositive() const;
    void Modulus(const SInteger128& divisor,
                 SInteger128& Quotient,
                 SInteger128& Remainder,
                 bool Signed) const;
    void CopyTo(int32* pVal);


protected:
    void TwosComplement();
    void InverseTwosComplement();


public:
    //Actual member data variables
    int32         value[4];
};

/**/
struct SMethodTableEntry {
    EMethodFlags  flags;
    UTarget             target;
};

/**/
// <summary>
// Structure encapsulating a UUID.
// (@author = fvmgen-extension)
// </summary>
struct UUID : public GUID {
    UUID();
    UUID(const GUID& rgidGuid);
```

```
    UUID(const UUID& rstUuid);
    UUID(const char* pacGuid);
    UUID(const std::string& rsGuid);


    UUID& operator=(const UUID& rstUuid);
    std::string asString() const;
    bool isNull() const;
    UUID& affect(const GUID& rgidUuid);
};

struct SModule {
    UUID uuidModuleM;

    CComSafeArray<int32> module_info_source;
    CComSafeArray<int32> static_data_source;
    CComSafeArray<int32> static_code_source;
    CComSafeArray<int32> method_table_source;
    CComSafeArray<int32> method_block_source;

    EModuleFlags flags;
    TDataTarget static_data;
    TCodeTarget static_code;
    SMethodTableEntry* method_table;
    int32 table_size;
    TCodeTarget method_block;

    SModule(const UUID& uuidModule, CComVariant& clModule);
    ~SModule();
};

/**/
// <summary>
// Structure encapsulating a collection of modules in memory.
// (@author = fvmgen-extension)
// </summary>
struct SModuleContainer : public TModuleVector {
    SModuleContainer();
    ~SModuleContainer();

    TModuleIndex  load(IBnclRepositoryComponent* pRepository,
                       GUID* pgidModuleUuid);
    TModuleIndex  loadFromFile(IBnclRepositoryComponent* pRepository,
                               const std::string& rsFilename);
};


/**/
// <summary>
// Structure encapsulating a BNCL stack.
// (@author = fvmgen-extension)
// </summary>
struct SStack {
```

```
    /**/
    // Pointer to the end of the stack.
    void*               tos;
    /**/
    // Size of the stack in 32-bits increment.
    int                         iSize;
    /**/
    // Pointer to beginning of memory region of stack.
    int32*              pBegin;
    /**/
    // Pointer to end of memory region of stack.
    int32*              pEnd;

    SStack();
    ~SStack();

    void push(int32 value);
    void push(float64 value);
    void pop(int size = 1);
};

/**/
struct SExceptionTableEntries {
    int32          module;
    int32          method;

    SExceptionTableEntries() {
        module = 0xFFFFFFFF;
        method = 0;
    }
};


/**/
struct SExceptionTable {
    SExceptionTableEntries          vectors[1024];
};

/**/
template<class typ>
struct SRegisterWindows {
    typ*           pInRegisters;
    typ*           pLocalRegisters;
    typ*           pOutRegisters;

    typ*           pBegin;
    typ*           pEnd;

    SRegisterWindows();
    ~SRegisterWindows();

    void next();
    void prev();
```

```
};



////////////////////////////////////////////////////////////////////////////////
////
//// Custom Exceptions
////
////////////////////////////////////////////////////////////////////////////////


/**/
class InvalidExceptionTableException : public std::exception {
private:

    std::string              sReason;


public:

    InvalidExceptionTableException() {
        sReason = "Uncatchable InvalidExceptionTableException : ";
    }
    InvalidExceptionTableException(const char* reason) {
        using namespace std;

        sReason = "Uncatchable InvalidExceptionTableException : ";
        sReason += reason;
    }

    const char *what() const throw() {
        return sReason.c_str();
    }
};




/**/
// <summary>
// Data structure containing the current state of the virtual machine.
// </summary>
// <remarks>
// This structure is defined both by fvmgen and by extensions. This
// structure is passed to any natively implemented method.
// </remarks>
struct SVirtualMachineState {


    SRegisters   registers;
```

```
    TCodeTarget   instptr;


    /**/
    // Registers windows.
    SRegisterWindows<int32>        window_integer;
    SRegisterWindows<float64>      window_real;
    /**/
    // Stack region of the VM.
    SStack        stacks;
    /**/
    // BNCL Modules Container.
    SModuleContainer   modules;
    /**/
    // Current BNCL module index.
    TModuleIndex        current_module;
    /**/
    // IO spaces accessors.
    // SPACE #1 = TARGET_INDEPENDANT.
    // SPACE #2 = TARGET_DEPENDANT.
    IBVHCore*                iospaces_core[2];
    IBVHPortsAccessor*  iospaces[2];
    /**/
    // Module repository.
    IBnclRepositoryComponent* repository;
    /**/
    // Structure containing the exceptions table. Each exception points
    // to a method in a
    // module.
    SExceptionTable                excptable;
    /**/
    // Synchronization object for BVH exceptions execution.
    HANDLE                         synchro;
    /**/
    // Tell the number of exception to execute next. If 0, no exception
    // to execute. Used by BVH communications facilities.
    TExceptionVector          pending;

};


/**/
// <summary>
// Classe abstracting the virtual machine. It inherit from
// SVirtualMachineState.
// </summary>
class VirtualMachine : public SVirtualMachineState {
public:

    VirtualMachine();
    ~VirtualMachine();

    void run();
```

```
};

} // namespace fvmgen

#endif
```

ANNEXE 4

DEFINITIONS LIÉES AU PROGRAMME THERMOCOUPLE_READER_BVH

Cette annexe contient les fichiers d'entête C++ (*header files*) utilisés dans la génération du code d'implémentation d'un BVH. Un BVH se présente comme un objet COM à la machine virtuelle BNCL (BVM). Ces fichiers sont générés automatiquement par l'utilitaire *bvh_code_generator* et l'assistant de création du BVH. Pour plus d'informations, se référer au Chapitre 3 et en particulier à la Figure 17.

Fichier BVHCore.h

```
//
// BVHCore.h
// ------------------------------------------------------------------------
// Définition de la composante COM BVHCore.
//
// ------------------------------------------------------------------------
// 2002-8-22 - FWS
//          - Création
//
//


#ifndef __BVHCORE_H_
#define __BVHCORE_H_


#pragma once
#include "resource.h"        // main symbols


#include "BVHVisual.h"
#include "BVHPortsAccessor.h"

#include "COMInterfaces.h"



// CBVHCore

[
      coclass,
      threading("free"),
      aggregatable("never"),
      vi_progid("ThermocoupleReader.BVHCore"),
      progid("ThermocoupleReader.BVHCore.1"),
      version(1.0),
      uuid("1E174F77-A623-464C-886C-BCAB07D9B8CC"),
      helpstring("BVHCore Class")
]
```

```
class ATL_NO_VTABLE CBVHCore :
	public IBVHCore
{
private:

	/**/
	CComPtr<IBVHVisual>             pVisualM;
	/**/
	CComPtr<IBVHPortsAccessor>      pPortsM;
	/**/
	CComPtr<IBVHProcessingHook>     pVisualHookM;
	/**/
	CComPtr<IBVHProcessingHook>     pPortsHookM;


public:

	CBVHCore();
	~CBVHCore();

	DECLARE_PROTECT_FINAL_CONSTRUCT()

	HRESULT FinalConstruct();
	void FinalRelease();

public:

	/* Interface COM IBVHCore */
	STDMETHOD(get_Visual)(IDispatch* *pVal);
	STDMETHOD(get_PortsAccessor)(IDispatch* *pVal);
	STDMETHOD(get_TargetIndustry)(BSTR *pVal);
	STDMETHOD(get_OperationsType)(BSTR *pVal);
	STDMETHOD(get_VirtualHardwareType)(BSTR *pVal);
	STDMETHOD(get_TargetIndustryDescription)(BSTR *pVal);
	STDMETHOD(get_OperationsTypeDescription)(BSTR *pVal);
	STDMETHOD(get_VirtualHardwareTypeDescription)(BSTR *pVal);
	STDMETHOD(get_Author)(BSTR *pVal);
	STDMETHOD(get_Version)(BSTR *pVal);
	STDMETHOD(get_Description)(BSTR *pVal);
	STDMETHOD(get_PortMapType)(BSTR *pVal);
	STDMETHOD(get_PortMapTypeDescription)(BSTR *pVal);

};


#endif
```

Fichier BVHPortsAccessor.h

```
//
// BVHPortsAccessor.h
// -------------------------------------------------------------------
// Définition de la composante COM local BVHPortsAccessor.
//
// -------------------------------------------------------------------
// 2002-8-22 - FWS
//          - Création
//
//

#ifndef __BVHPORTSACCESSOR_H_
#define __BVHPORTSACCESSOR_H_


#pragma once
#include "resource.h"       // main symbols

#include "COMInterfaces.h"
#include "motions.h"



// CBVHPortsAccessor

[
	coclass,
	threading("free"),
	aggregatable("never"),
	vi_progid("ThermocoupleReader.BVHPortsAccessor"),
	progid("ThermocoupleReader.BVHPortsAccessor.1"),
	event_source("com"),
	version(1.0),
	uuid("AE67BB80-482B-4127-8509-BECD0B7EE3BE"),
	helpstring("BVHPortsAccessor Class"),
	noncreatable
]
class ATL_NO_VTABLE CBVHPortsAccessor :
	public CComCoClass<CBVHPortsAccessor,
				&__uuidof(CBVHPortsAccessor)>,
	public IBVHPortsAccessor,
	public IBVHProcessingHook,
	public IBvmException,
	public IWorkerThreadClient
{
private:

	/**/
	MachineManager*			pclMachineM;
```

```
        /**/
        MotionManager*                 pclMotionsM;

        /**/
        IBVHProcessingHook*            pinVisualLinkM;

        /**/
        CComPtr<IBVHDisplayOutput>     pOutputM;

        /**/
        // Thread handling ports changes.
        CWorkerThread<DefaultThreadTraits>          clChangesThreadM;


        /**/
        // Variable telling if the thread has been initialized.
        int                                   iThreadInitializedM;
        /**/
        // Event permettant de laisser dormir la thread qui s'occupe de
        // traiter les changements de ports.
        HANDLE                                hPortSynchronizerM;


public:

        __event __interface IBVMExceptionEvent;


        CBVHPortsAccessor();


        DECLARE_PROTECT_FINAL_CONSTRUCT()

        HRESULT FinalConstruct();
        void FinalRelease();



public:

        /* Interface COM IBVHPortsAccessor */
        STDMETHOD(writeInteger)(int iPort, int iValue);
        STDMETHOD(writeReal)(int iPort, double dValue);
        STDMETHOD(writeVInteger)(int iPort, int * paiVector);
        STDMETHOD(writeVReal)(int iPort, double * padVector);
        STDMETHOD(readInteger)(int iPort, int * piValue);
        STDMETHOD(readReal)(int iPort, double * pdValue);
        STDMETHOD(readVInteger)(int iPort, int * paiVector);
        STDMETHOD(readVReal)(int iPort, double * padVector);

        /* Interface COM IBVHProcessingHook */
        STDMETHOD(init)(IUnknown* pInt);
        STDMETHOD(chain)(IBVHProcessingHook* pInt);
```

```
    STDMETHOD(link)(IBVHProcessingHook* pInt);
    STDMETHOD(hook)();
    STDMETHOD(clean)();

    /* Interface IWorkerThreadClient */
    HRESULT CloseHandle(HANDLE hHandle);
    HRESULT Execute(DWORD_PTR dwParam, HANDLE hObject );

    /* Interface IBvmException */
    void raise(int vec);

};


#endif
```

Fichier BVHVisual.h

```
//
// BVHVisual.h
// -----------------------------------------------------------------
// Définition de la composante COM local BVHVisual.
//
// -----------------------------------------------------------------
// 2002-8-22 - FWS
//           - Création
//
//


#ifndef __BVHVISUAL_H_
#define __BVHVISUAL_H_


#pragma once
#include "resource.h"       // main symbols


#include "COMInterfaces.h"


/**/
// <summary>
// Classe d'implémentation de la fenêtre du visuel de l'objet.
// </summary>
// <remarks>
// Cette classe gère les interactions avec l'usager et propose
// certaines méthodes pour permettre à d'autre classe de connaitre
// l'état de ces interactions. Dans certaines limites cette classe gère
```

```
// l'orientation de la vue sur la machine.
// </remarks>
class ViewportDialog : public CDialogImpl<ViewportDialog> {
public:
	enum { IDD = IDD_BVHVISUAL_DIALOG };

	ViewportDialog();

	BEGIN_MSG_MAP(ViewportDialog)
		MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
	END_MSG_MAP()



	LRESULT OnInitDialog(UINT uMsg,
	                     WPARAM wParam,
	                     LPARAM lParam,
	                     BOOL& bHandled) {
		return 1;
	}
	LRESULT OnBnClickedButton1(WORD /*wNotifyCode*/,
	                           WORD /*wID*/,
	                           HWND /*hWndCtl*/,
	                           BOOL& /*bHandled*/);

};



// CBVHVisual

[
	coclass,
	threading("free"),
	aggregatable("never"),
	vi_progid("ThermocoupleReader.BVHVisual"),
	progid("ThermocoupleReader.BVHVisual.1"),
	version(1.0),
	uuid("DAD9C25F-06F0-429A-9AAA-85AE364ACBD7"),
	helpstring("BVHVisual Class"),
	noncreatable
]
class ATL_NO_VTABLE CBVHVisual :
	public CComCoClass<CBVHVisual, &__uuidof(CBVHVisual)>,
	public IBVHVisual,
	public IBVHProcessingHook,
	public IBVHDisplayOutput,
	public IWorkerThreadClient
{
private:

	/**/
	ViewportDialog		coViewportM;
```

```
	/**/
	IBVHProcessingHook*	pinPortsHookM;
	/**/
	CWorkerThread<DefaultThreadTraits> clDialogThreadM;

public:
	CBVHVisual();
	~CBVHVisual();

	DECLARE_PROTECT_FINAL_CONSTRUCT()

	HRESULT FinalConstruct();

	void FinalRelease();

public:

	/* Interface COM IBVHVisual */
	STDMETHOD(show)(VARIANT_BOOL bStatus);
	STDMETHOD(enabling)(VARIANT_BOOL bStatus);
	STDMETHOD(isVisible)(VARIANT_BOOL * pbStatus);
	STDMETHOD(isEnable)(VARIANT_BOOL * pbStatus);

	/* Interface COM IBVHProcessingHook */
	STDMETHOD(init)(IUnknown* pInt);
	STDMETHOD(chain)(IBVHProcessingHook* pInt);
	STDMETHOD(link)(IBVHProcessingHook* pInt);
	STDMETHOD(hook)();
	STDMETHOD(clean)();

	/* Interface COM IBVHDisplayOutput */
	STDMETHOD(display)(BSTR sInformation);

	/* Interface IWorkerThreadClient */
	HRESULT CloseHandle(HANDLE hHandle);
	HRESULT Execute(DWORD_PTR dwParam, HANDLE hObject );

};

#endif
```

Fichier COMInterfaces.h

```
//
// COMInterfaces.h
// -------------------------------------------------------------------
// Définiction des interfaces des composantes COM.
//
// -------------------------------------------------------------------
```

```
// 2002-8-22 - FWS
//          - Création
//
//


#ifndef _COMINTERFACES_H_
#define _COMINTERFACES_H_



// IBVHCore
[
      object,
      uuid("587A2F44-8CE5-44B3-8C6A-B628FD3380FC"),
      dual, helpstring("IBVHCore Interface"),
      pointer_default(unique)
]
__interface IBVHCore : IDispatch
{
      [propget, id(1), helpstring("property Visual")]
            HRESULT Visual([out,retval] IDispatch* *pVal);
      [propget, id(2), helpstring("property PortsAccessor")]
            HRESULT PortsAccessor([out, retval] IDispatch* *pVal);
      [propget, id(3), helpstring("property TargetIndustry")]
            HRESULT TargetIndustry([out, retval] BSTR *pVal);
      [propget, id(4), helpstring("property OperationsType")]
            HRESULT OperationsType([out, retval] BSTR *pVal);
      [propget, id(5), helpstring("property VirtualHardwareType")]
            HRESULT VirtualHardwareType([out, retval] BSTR *pVal);
      [propget, id(6), helpstring("propertyTargetIndustryDescription")]
            HRESULT TargetIndustryDescription(
                  [out, retval] BSTR *pVal);
      [propget, id(7), helpstring("propertyOperationsTypeDescription")]
            HRESULT OperationsTypeDescription(
                  [out, retval] BSTR *pVal);
      [propget, id(8),
       helpstring("property VirtualHardwareTypeDescription")]
            HRESULT VirtualHardwareTypeDescription(
                  [out, retval] BSTR *pVal);
      [propget, id(9), helpstring("property Author")]
            HRESULT Author([out, retval] BSTR *pVal);
      [propget, id(10), helpstring("property Version")]
            HRESULT Version([out, retval] BSTR *pVal);
      [propget, id(11), helpstring("property Description")]
            HRESULT Description([out, retval] BSTR *pVal);
      [propget, id(12), helpstring("property PortMapType")]
            HRESULT PortMapType([out, retval] BSTR *pVal);
      [propget, id(13), helpstring("property PortMapTypeDescription")]
            HRESULT PortMapTypeDescription([out, retval] BSTR *pVal);
};
```

```
// IBVHProcessingHook
[
      object,
      uuid("8342C6D2-630B-419f-B608-ACA6C79166AE"),
      helpstring("IBVHProcessingHook Interface"),
      pointer_default(unique)
]
__interface IBVHProcessingHook : IUnknown
{
      [helpstring("method init")]
            HRESULT init([in] IUnknown* pInt);
      [helpstring("method chain")]
            HRESULT chain([in] IBVHProcessingHook* pInt);
      [helpstring("method link")]
            HRESULT link([in] IBVHProcessingHook* pInt);
      [helpstring("method hook")]
            HRESULT hook();
      [helpstring("method clean")]
            HRESULT clean();
};


// IBVHVisual
[
      object,
      uuid("BDD42C5B-FA14-41ec-AD63-99AA93E83708"),
      dual, helpstring("IBVHVisual Interface"),
      pointer_default(unique)
]
__interface IBVHVisual : IDispatch
{
      [id(1), helpstring("method show")]
            HRESULT show([in] VARIANT_BOOL bStatus);
      [id(2), helpstring("method enabling")]
            HRESULT enabling([in] VARIANT_BOOL bStatus);
      [id(3), helpstring("method isVisible")]
            HRESULT isVisible([out, retval] VARIANT_BOOL* pbStatus);
      [id(4), helpstring("method isEnable")]
            HRESULT isEnable([out, retval] VARIANT_BOOL* pbStatus);
};

// IBVHDisplayOutput
[
      object,
      uuid("0B476D72-66B8-4CA1-B19B-27CAD7746C49"),
      helpstring("IBVHRpmOutput"),
      pointer_default(unique)
]
__interface IBVHDisplayOutput : IUnknown
{
      HRESULT display([in] BSTR sInformation);
};
```

```
// IBVHPortsAccessor
[
	object,
	uuid("2595CBF3-C741-4d07-AA05-0D72433AD5D0"),
	dual, helpstring("IBVHPortsAccessor Interface"),
	pointer_default(unique)
]
__interface IBVHPortsAccessor : IDispatch
{
	[id(1), helpstring("method writeInteger")]
		HRESULT writeInteger([in] int iPort, [in] int iValue);
	[id(2), helpstring("method writeReal")]
		HRESULT writeReal([in] int iPort, [in] double dValue);
	[id(3), helpstring("method writeVInteger")]
		HRESULT writeVInteger([in] int iPort,
					[in, size_is(4)] int* paiVector);
	[id(4), helpstring("method writeVReal")]
		HRESULT writeVReal([in] int iPort,
					[in, size_is(4)] double* padVector);
	[id(5), helpstring("method readInteger")]
		HRESULT readInteger([in] int iPort,
					[out, retval] int* piValue);
	[id(6), helpstring("method readReal")]
		HRESULT readReal([in] int iPort,
					[out, retval] double* pdValue);
	[id(7), helpstring("method readVInteger")]
		HRESULT readVInteger([in] int iPort,
					[out, size_is(4), retval] int*
						paiVector);
	[id(8), helpstring("method readVReal")]
		HRESULT readVReal([in] int iPort,
					[out, size_is(4), retval] double*
						padVector);
};


// IBVMExceptionEvent
[
	object,
	uuid("19BD7F16-6515-4ABC-B80E-6D74A52605CA"),
	dual, helpstring("IBVMExceptionEvent Interface"),
	pointer_default(unique)
]
__interface IBVMExceptionEvent : IDispatch
{
	[id(1), helpstring("method raise")]
		HRESULT send([in] int iExceptionVector);
};


#endif
```

Fichier motions.h

```
//
// motions.h
// ----------------------------------------------------------------------
// Définition des classes de support à la gestion des mouvements et
autres
// fonctions du BVH.
//
// ----------------------------------------------------------------------
// 2002-8-22 - FWS
//           - Création
//
//


#ifndef _MOTION_H_
#define _MOTION_H_


#include <string>
#include <queue>
#include <list>



// Fichier autogénéré par les wizards XSLT de la spécification du BVH
utilisé.
#include "PortsManagerDef.h"

#include "COMInterfaces.h"

class Motion;


//////////////////////////////////////////////////////////////////////////
//
// Types et enum
//
//////////////////////////////////////////////////////////////////////////



/**/
// Énumération précisant le type de port. Cette énumération est
// utilisée par la structure SPortChange pour déterminer quel type de
// port est associé au changement.
```

```
enum TPortType { int_type = 1, real_type = 2, void_type = 16 };




/**/
// Type d'une valeur réelle utilisée par les trajectoires, par exemple
// une valeur de rayon.
//
typedef           double            TMotionScalar;



/**/
// <summary>
// Queue contenant des pointeurs sur des objets Motion.
// </summary>
// <remarks>
// Des pointeurs sont utilisés puisque Motion est une classe abstraite
// et que l'opérateur d'égalité ne pourrait être utilisé pour ajouter
// des objets à la queue. ie l'espace alloué ne serait que pour un
// objet Motion et non pour l'objet dérivé. Il ne serait donc pas
// possible de copier tous les attributs de l'objet dérivé puisque
// Motion n'a aucune idée de ce qui peut dérivé de lui. Et de toutes
// façons, une classe abstraite ne peut être créée.
// NOTE: On doit faire attention de ne pas désallouer les objets
// pointés par les pointeurs placés dans cette queue.
// </remarks>
typedef           std::queue<Motion*>          TMotionPtrQueue;

/**/
// <summary>
// Liste contenant des pointeurs sur des objets Motion.
// </summary>
// <remarks>
// Des pointeurs sont utilisés puisque Motion est une classe abstraite
// et que l'opérateur d'égalité ne pourrait être utilisé pour ajouter
// des objets au vector. ie l'espace alloué ne serait que pour un objet
// Motion et non pour l'objet dérivé. Il ne serait donc pas
// possible de copier tous les attributs de l'objet dérivé puisque
// Motion n'a aucune idée de ce qui peut dérivé de lui. Et de toutes
// façons, une classe abstraite ne peut être créée.
// NOTE: On doit faire attention de ne pas désallouer les objets
// pointés par les pointeurs placés dans ce vecteur.
// </remarks>
typedef           std::list<Motion*>           TMotionPtrList;
```

```
////////////////////////////////////////////////////////////////////////
//
// Interfaces
//
////////////////////////////////////////////////////////////////////////


/**/
// <summary>
// Interface utilisée pour permettre à MachineManager de lancer des
// exceptions à la BVM.
// </summary>
struct IBvmException {
      virtual void            raise(int vec) = 0;
};



/**/
// <summary>
// Interface permettant d'accéder aux coordonnées machines.
// </summary>
// <remarks>
// Le "Inverse Kinematics Engine" se trouve dans l'implémentateur de
// cette interface. Cette interface est typiquement utilisée par la
// classe Motion pour obtenir les coordonnées courantes et mettre à
// jour les changements de coordonnées.
// </remarks>
struct IMachineCoorAccessor {
      virtual double          get_x() = 0;
      virtual double          get_y() = 0;
      virtual double          get_z() = 0;
      virtual double          get_a() = 0;
      virtual double          get_b() = 0;
      virtual double          get_c() = 0;
      virtual void            set_x(double x) = 0;
      virtual void            set_y(double y) = 0;
      virtual void            set_z(double z) = 0;
      virtual void            set_a(double a) = 0;
      virtual void            set_b(double b) = 0;
      virtual void            set_c(double c) = 0;
};



/**/
// <summary>
// Accès aux ports BNCL du BVH.
// </summary>
```

```
// <remarks>
// Cette interface est implémentée par la classe qui gère les valeurs
// et les changements sur les ports et est appelée par n'impore quelle
// classe désirant modifier ou lire les ports.
// </remarks>
struct IPortsAccessor {
      virtual void          route(int iPort, int iValue) = 0;
      virtual void          route(int iPort, double dValue) = 0;
      virtual int           readInteger(int iPort) = 0;
      virtual double        readReal(int iPort) = 0;
};
```

```
/**/
// <summary>
// Interface implémentée par la classe gérant les mouvements de la
// machine.
// </summary>
// <remarks>
// Il est possible d'ajouter des trajectoires synchronisées ou des
// trajectoires simultannées. Idéalement les mouvements sont placés
// dans une queue et sont réalisés un à la suite de l'autre. Les
// mouvements simultannés sont placés dans une autre queue, ou selon
// une méthode propre à l'implémentateur de cette interface.
//
// NOTE:
// 1) L'implémentateur de cette interface DOIT faire un "delete" sur
//    les Motion après leur utilisation.
// 2) Les objets Motion doivent avoir été créés à l'aide de l'opérateur
//    new.
// </remarks>
struct IMotionController {
      virtual void addSynchroMotion(Motion* pclMotion) = 0;
      virtual void addSimulMotion(Motion* pclMotion) = 0;
      virtual bool process() = 0;
};
```

```
//////////////////////////////////////////////////////////////////////////
//
// Structures
//
//////////////////////////////////////////////////////////////////////////
```

```
/**/
// <summary>
// Union contenant la valeur d'un port.
// </summary>
// <remarks>
// Cette valeur peut être entière ou réelle.
// </remarks>
union UPortValue {
      int                 integer;
      double              real;

      __int64             value;
};
```

```
/**/
// <summary>
// Structure contenant les informations nécessaires au changement d'un
// port.
// </summary>
// <remarks>
// Cette structure est placée dans PortsChangeFifo. Le changement peut
// se faire sur un port entier ou réel.
// </remarks>
struct SPortChange {
      int                 port;
      TPortType   type;
      UPortValue  change;

      SPortChange() { port = 0; type = void_type; change.value = 0; }
      SPortChange(int iPort, int iValue) {
            port = iPort;
            type = int_type;
            change.integer = iValue;
      }
      SPortChange(int iPort, double dValue) {
            port = iPort & 0xFFFFFFFE;
            type = real_type;
            change.real = dValue;
      }

      SPortChange&        operator=(SPortChange& rstSource) {
            port = rstSource.port;
            type = rstSource.type;
            change.value = rstSource.change.value;
            return *this;
      }
```

```
};




/**/
// <summary>
// Union permettant d'accéder aux ports entier et réels plus aisément.
// </summary>
// <remarks>
// Cette union contient un pointeur sur des int et un pointeur sur des
// double.
// </remarks>
union UPorts {
	int*		integer;
	double*		real;
};



/**/
// <summary>
// Structure représentant un point machine (orientation réelle des
// axes) dans l'espace.
// </summary>
struct SMotionPoint {
	double			x, y, z;
	double			a, b, c;


	SMotionPoint() {
		x = 0.0; y = 0.0; z = 0.0; a = 0.0; b = 0.0; c = 0.0;
	}

	SMotionPoint(double x, double y, double z) {
		x = x; y = y; z = z; a = 0.0; b = 0.0; c = 0.0;
	}

	SMotionPoint(double x, double y, double z,
			double a, double b, double c) {
		x = x; y = y; z = z; a = a; b = b; c = c;
	}

	SMotionPoint(const SMotionPoint& rstSource) {
		x = rstSource.x; y = rstSource.y; z = rstSource.z;
	}

	SMotionPoint& operator=(SMotionPoint& rstSource) {
		x = rstSource.x; y = rstSource.y; z = rstSource.z;
		a = rstSource.a; b = rstSource.b; c = rstSource.c;
		return *this;
	}
```

```
	SMotionPoint operator+(SMotionPoint& rstSource) {
		return SMotionPoint(
			x + rstSource.x, y + rstSource.y, z + rstSource.z);
	}

	SMotionPoint operator-(SMotionPoint& rstSource) {
		return SMotionPoint(
			x - rstSource.x, y - rstSource.y, z - rstSource.z);
	}

	SMotionPoint operator-() {
		return SMotionPoint(-x, -y, -z);
	}
};


/**/
// <summary>
// Structure représentant un vecteur machine dans l'espace.
// </summary>
struct SMotionVector {
	double			i, j, k;

	SMotionVector() { this->i = 0.0; this->j = 0.0; this->k = 0.0; }
	SMotionVector(double i, double j, double k) {
		this->i = i; this->j = j; this->k = k;
	}

	SMotionVector& operator=(SMotionVector& rstSource) {
		this->i = rstSource.i;
		this->j = rstSource.j;
		this->k = rstSource.k;
	}
};



////////////////////////////////////////////////////////////////////////////
//
// Classes
//
////////////////////////////////////////////////////////////////////////////


/**/
// <summary>
// Classe abstraite permettant de gérer une trajectoire.
// </summary>
// <remarks>
// Cette classe est utilisée comme classe de base par toutes les
```

```
// classes implémentant un certain type de trajectoire. Les mouvements
// accessoires de la machine dérivent également de Motion pour
// faciliter le codage des routines gérant l'ensemble des trajectoires
// (par exemple MotionManager). On évite ainsi de créer une autre
// classe pour les mouvements accessoires alors que cela ne serait pas
// nécessaire et aurait de toutes façons la même interface que la
// classe Motion. Il est à noter que les attributs de cette classe sont
// "protected". Il en a été décidé ainsi puisque ces attributs sont
// directement utilisés par les classes dérivant de Motion et qu'aucun
// contrôle particulier ne doit être fait sur ceux-ci. On évite ainsi
// des appels de méthodes getxxx inutiles.
// </remarks>
class Motion {
protected:

      IMachineCoorAccessor*        pinPortsManagerM;
      __int64                                iTimeM;
      bool                               bInMotionM;


public:

      Motion(IMachineCoorAccessor* pinPortsManager) {
            pinPortsManagerM = pinPortsManager;
      }

      void              reset() { bInMotionM = false; }
      void              setTime(__int64 iTime) { iTimeM = iTime; }

      virtual bool      next() = 0;
      virtual double    getTotalDistance() = 0;
};




/**/
// <summary>
// Classe gérant les mouvements de la machine, tant des axes que des
// accessoires.
// </summary>
// <remarks>
// Cette classe implémente l'interface IMotionController. Il est
// important de noter que MotionManager fera un "delete" sur les
// pointeurs des objets Motion contenus dans la queue synchro et le
// vecteur simultanné.
// </remarks>
class MotionManager : public IMotionController {
private:

      TMotionPtrQueue         clSynchroMotionM;
      TMotionPtrList          clSimulMotionM;
```

```
public:

      MotionManager();
      ~MotionManager();


      /* Interface IMotionController */
      void        addSynchroMotion(Motion* pclMotion);
      void        addSimulMotion(Motion* pclMotion);
      bool        process();

};




/**/
// <summary>
// Fifo contenant les changements de ports.
// </summary>
// <remarks>
// Les changements en écriture des ports ne se répercutent pas
// automatiquement sur les ports. Les changements sont mis dans un
// queue et sont ensuite extraits par une thread différente.
// </remarks>
class PortsChangesFifo {
private:

      std::queue<SPortChange>       clChangeQueueM;


public:

      PortsChangesFifo();
      ~PortsChangesFifo();

      void              addInteger(int iPort, int iValue);
      void              addReal(int iPort, double dValue);
      SPortChange&      getCurrent();
      void              pop();
      int               size();
      bool              isEmpty();
      void              clear();
};
```

```
/**/
// <summary>
// Classe de gestion de la machine.
// </summary>
// <newpara>
// Cette classe est à la base du contrôle de la machine. La méthode
// MachineManager::process() est appelée par le créateur de
// MachineManager et à son tour MachineManager::process() s'occupe
// d'appeler IMotionController::process() pour effectuer les
// trajectoires.
// </newpara>
// <newpara>
// L'interface IMachineCoorAccessor est implémentée par cette classe et
// permet à une classe utilisatrice d'accéder aux valeurs des
// coordonnées machines des six axes possibles. L'interface
// IPortsAccessor est utilisée par une classe utilisatrice pour écrire
// dans des ports ou bien lire sur des ports. Les écritures de ports
// passent par le PortsChangesFifo. La méthode process() extrait des
// valeurs de ports de ce Fifo à mesure qu'elle est appelée et selon
// les besoins (ie si aucune trajectoire synchro n'est en court).
// </newpara>
// </remarks>
class MachineManager : public PortsManager, public
IMachineCoorAccessor, public IPortsAccessor {
private:

      PortsChangesFifo        clChangesM;
      IMotionController*            pinMotionControllerM;
      IBvmException*                pinBvmM;

      // Position courante.
      SMotionPoint                  stPositionM;
      // Direction de coupe.
      SMotionVector                 stDirectionM;
      bool                          bNeedNewPosM;

      // Variables used for temperature control.
      double                        dInitialVoltageM;

      IBVHDisplayOutput*            pOutputM;


private:

      bool              extract();        // AUTOGENERATED
      void              computepos();     // Cinématique inverse.


public:
```

```
MachineManager(IMotionController* pinMotionController,
               IBvmException* pinBvm);
~MachineManager();

bool                process();


void                setDisplayOutput(IBVHDisplayOutput* pOutput);


// Interface IMachineCoorAccessor
double        get_x();
double        get_y();
double        get_z();
double        get_a();
double        get_b();
double        get_c();
void          set_x(double x);
void          set_y(double y);
void          set_z(double z);
void          set_a(double a);
void          set_b(double b);
void          set_c(double c);


// Interface IPortsAccessor
void          route(int iPort, int iValue);
void          route(int iPort, double dValue);
int           readInteger(int iPort);
double        readReal(int iPort);



////
//// Abstract virtual methods inherited from PortsManager class.
////


// Port Group = Machine Control
int validate_MC_ERROR(int val); // WRITE VALIDATION: Error
                                   Code Read.
int validate_MC_ERRORADR(int val); // WRITE VALIDATION: Error
                                      Port Address Read.
int validate_MC_ERRORRESET(int val); // WRITE VALIDATION: Error
                                        Port Reset.

// Port Group = Spindle Control
int validate_TC_SPINDLE(double val); // WRITE VALIDATION: Spindle
                                        rotation speed.

// Port Group = TemperatureControl
int validate_TC_ACTIVATE(int val);  // WRITE VALIDATION:
```

```
                                        Temperature control
                                        activation port.

};
```

Fichier PortsManagerDef.h

```
//
// PortsManagerDef.h
// ------------------------------------------------------------------
// Definition of PortsManager class.
// THIS CLASS IS AUTOGENERATED BY bvh_code_generation.xsl
// ON AN XML DOCUMENT DESCRIBING A BNCL VIRTUAL HARDWARE
// AND WITH bvh_classdef_extraction.xsl EXECUTED ON THE
// XML RESULT OF THE FIRST XSL.
//
// This document can change with each execution of the
// automatic code generation.
//
// ------------------------------------------------------------------
//
//


#define            DOUBLE_INFINITY
      (double)(__int64)0x7FF0000000000000



//////////////////////////////////////////////////////////
//// Port possible errors codes
//////////////////////////////////////////////////////////

/*
Value that can be read at the error port when everything is working
according to instructions.
*/
#define ERR_OK 0x00000000 // Everything is fine

/*
Linear feed or spindle rotation speed entered is invalid. This can be
due to a value greater than the max possible speed. BVH must trigger
this error if speed are in excess of what the machine is capable of.

*/
#define ERR_INVALIDSPEED 0x80000005 // Invalid speed
/*
```

```
Global error code used when a general error has been encountered.
*/
#define ERR_ERROR 0xFFFFFFFF // Global error

/*
An attempt has been made to access a port that does not exist on this
BVH specification.
*/
#define ERR_INVALIDADR 0xFFFFFFFE // Invalid port address




////////////////////////////////////////////////////
//// Port address by ID
////////////////////////////////////////////////////


// Port Group = Machine Control
#define BNCLPORT_MC_ERROR (0x00000000 + 0x0) // Error Code Read.
#define BNCLPORT_MC_ERRORADR (0x00000000 + 0x1) // Error Port Address
                                                   Read.
#define BNCLPORT_MC_ERRORRESET (0x00000000 + 0x2) // Error Port Reset.

// Port Group = Spindle Control
#define BNCLPORT_TC_SPINDLE (0x00000300 + 0x0) // Spindle rotation
                                                 speed.

// Port Group = TemperatureControl
#define BNCLPORT_TC_ACTIVATE (0x00000400 + 0x0) // Temperature control
                                                   activation port.


////////////////////////////////////////////////////
//// Exception vectors
////////////////////////////////////////////////////


#define EXCP_ERROR (64) // General error reporting interruption



////////////////////////////////////////////////////
//// PortsManager class
////////////////////////////////////////////////////

class PortsManager {
private:
```

```
    union {
          int                 integer;
          double              real;
          __int64             value;
    } ports[5];

public:

    PortsManager() {
          for(int i = 0;i < 5;i++) {
                ports[i].value = 0x0000000000000000;
          }
    }

    void throwError(int iError) {
          set_MC_ERROR(iError);
          set_MC_ERRORRESET(1);
    }


    //// Port Group = Machine Control,
    //// Group Base Address = 0x00000000

    // READ: Error Code Read.
    int get_MC_ERROR() { return ports[0].integer; }
    // WRITE: Error Code Read.
    void set_MC_ERROR(int iValue) { ports[0].integer = iValue; }
    // WRITE VALIDATION: Error Code Read.
    virtual int validate_MC_ERROR(int val) = 0;
    // READ: Error Port Address Read.
    int get_MC_ERRORADR() { return ports[1].integer; }
    // WRITE: Error Port Address Read.
    void set_MC_ERRORADR(int iValue) { ports[1].integer = iValue; }
    // WRITE VALIDATION: Error Port Address Read.
    virtual int validate_MC_ERRORADR(int val) = 0;
    // READ: Error Port Reset.
    int get_MC_ERRORRESET() { return ports[2].integer; }
    // WRITE: Error Port Reset.
    void set_MC_ERRORRESET(int iValue) { ports[2].integer = iValue; }
    // WRITE VALIDATION: Error Port Reset.
    virtual int validate_MC_ERRORRESET(int val) = 0;

    //// Port Group = Spindle Control,
    //// Group Base Address = 0x00000300

    // READ: Spindle rotation speed.
    double get_TC_SPINDLE() { return ports[3].real; }
    // WRITE: Spindle rotation speed.
    void set_TC_SPINDLE(double dValue) { ports[3].real = dValue; }
    // WRITE VALIDATION: Spindle rotation speed.
    virtual int validate_TC_SPINDLE(double val) = 0;
```

```
    //// Port Group = TemperatureControl,
    //// Group Base Address = 0x00000400

    // READ: Temperature control activation port.
    int get_TC_ACTIVATE() { return ports[4].integer; }
    // WRITE: Temperature control activation port.
    void set_TC_ACTIVATE(int iValue) { ports[4].integer = iValue; }
    // WRITE VALIDATION: Temperature control activation port.
    virtual int         validate_TC_ACTIVATE(int val) = 0;
};


#endif
```

ANNEXE 5

FICHIER DE DESCRIPTION DU BVH THERMOCOUPLEREADER

Cette annexe contient le fichier de description des ports contrôlés par le BVH *ThermocoupleReader*. Ce fichier est utilisé par l'utilitaire *bvh_code_generator* pour générer une partie du code d'implémentation d'un BVH. Pour plus d'informations, voir le Chapitre 3 et plus particulièrement la Figure 17.

Fichier ThermocoupleReaderBVH.xml

```
<?xml version="1.0"?>
<!DOCTYPE bnclvirtualhardware SYSTEM "bnclvirtualhardware.dtd">

<bnclvirtualhardware>

<informations>
	<help>
		This specification defines the virtual hardware used for
		events reactivity testing of the BNCL architecture. It read
		a thermocouple temperature sensor and send back a #65
		interrupt if temperature exceed a certain level.
		The level is not configurable.
	</help>
	<version virtualhardware="1.0.0" bnclspec="1.0.0"/>
	<author>Etienne Fortin</author>
	<organisation></organisation>
	<support>
		<name>Etienne Fortin</name>
		<phone></phone>
		<email>bnclspec@yahoo.com</email>
	</support>
	<log>
		<entry
			author="Etienne Fortin"
			date="2002-08-22">
			Creation from PerformanceCounterBVH
		</entry>
	</log>
	<!-- DA0C6514-886D-45c6-968A-F3D87D677351
			Machining industry. -->
	<!-- D91EC20D-1BB1-490d-90FD-ED210EDD59AD
			3 axis numerical control machining. -->
	<target
		industry="DA0C6514-886D-45c6-968A-F3D87D677351"
		type="D91EC20D-1BB1-490d-90FD-ED210EDD59AD"/>
</informations>

<portsdescription>
	<errorport
		portid="MC_ERROR"
```

```
        adrportid="MC_ERRORADR"
        resetportid="MC_ERRORRESET"/>
<portgroup
        name="Machine Control"
        baseadr="00000000">
        <help>
              Reset the thermocouple reader. All counters are reset
              to zero.
        </help>
        <port
              id="MC_ERROR"
              datatype="integer"
              valuetype="state"
              readonly="yes"
              desc="Error Code Read">
        </port>
        <port
              id="MC_ERRORADR"
              datatype="integer"
              valuetype="state"
              readonly="yes"
              desc="Error Port Address Read">
        </port>
        <port
              id="MC_ERRORRESET"
              datatype="integer"
              valuetype="state"
              desc="Error Port Reset">
              <help>
                    Reset the error port to normal condition. This
                    reset indicates to the BVH that the error has
                    been taken care of.
              </help>
              <valuedesc
                    value="0"
                    desc="Reset error port to normal state."/>
              <valuedesc
                    value="1"
                    desc="An error is pending."/>
        </port>
</portgroup>

<portgroup
        name="Spindle Control"
        baseadr="00000300">
        <port
              id="TC_SPINDLE"
              #datatype="real"
              valuetype="scale"
              desc="Spindle rotation speed">
        </port>
</portgroup>
```

```
<portgroup
      name="TemperatureControl"
      baseadr="00000400">
      <help>
            Ports that control the behavior of the temperature
            control.
      </help>


      <port
            id="TC_ACTIVATE"
            datatype="integer"
            valuetype="state"
            desc="Temperature control activation port">
            <help>
                  Activate the temperature control. A big change
                  in temperature will trigger a #65 exception to
                  the BVM.
            </help>
            <valuedesc
                  value="0"
                  desc="Deactivate temperature control"/>
            <valuedesc
                  value="1"
                  desc="Activate temperature control"/>
      </port>
</portgroup>
</portsdescription>


<errors>
      <error
            id="ERR_OK"
            value="00000000"
            desc="Everything is fine">
            <help>
                  Value that can be read at the error port when
                  everything is working according to instructions.
            </help>
      </error>
      <error
            id="ERR_INVALIDSPEED"
            value="80000005"
            desc="Invalid speed">
            <help>
                  Linear feed or spindle rotation speed entered is
                  invalid. This can be due to a value greater
                  than the max possible speed. BVH must trigger this
                  error if speed are in excess of what the machine is
                  capable of.
            </help>
      </error>
      <error
```

```
            id="ERR_ERROR"
            value="FFFFFFFF"
            desc="Global error">
            <help>
                  Global error code used when a general error has been
                  encountered.
            </help>
      </error>
      <error
            id="ERR_INVALIDADR"
            value="FFFFFFFE"
            desc="Invalid port address">
            <help>
                  An attempt has been made to access a port that does
                  not exist on this BVH specification.
            </help>
      </error>
</errors>


<exceptions>
      <errorexcp excpid="EXCP_ERROR"/>
      <exception
            id="EXCP_ERROR"
            vector="64"
            desc="General error reporting interruption"/>
</exceptions>


<helplinks>
      <figures>
      </figures>
</helplinks>

</bnclvirtualhardware>
```

ANNEXE 6

SOURCE DU MODULE BNCL UTILISÉ AVEC LE THERMOCOUPLE

Cette annexe contient le fichier source BNCL utilisé pour la réalisation du test d'extensibilité de l'architecture BNCL, et le module BNCL exécutable résultant. Ce test est présenté au Chapitre 6.

Fichier BvhThermocoupleTest1.basm

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<fvm:source xmlns:fvm="urn:fvm:source">

      <fvm:informations>
            <fvm:module
                  uuid="{D5658875-48FB-415c-BE03-670D811F83DB}"
                  architecture="{4E992097-B795-4824-BF3D-
                  45B4FBE1F959}"/>
            <fvm:version>1.0.0.6</fvm:version>
            <fvm:author>
                  TEST
            </fvm:author>
            <fvm:organisation>
                  TEST
            </fvm:organisation>
            <fvm:support>
                  <fvm:name>TEST</fvm:name>
                  <fvm:phone>1-450-649-9774</fvm:phone>
                  <fvm:email>etienne.fortin@sympatico.ca</fvm:email>
            </fvm:support>
            <fvm:log>
            </fvm:log>
      </fvm:informations>

      <fvm:staticblock>
            <fvm:data>
                  <fvm:dataentry
                        name="MESSAGE_0"
                        datatype="string"
                        value="WARNING : Temperature exceeded maximum
                        allowed. Lowering spindle speed."/>
            </fvm:data>
            <fvm:code>
                  ret;
            </fvm:code>
      </fvm:staticblock>

      <fvm:methodtable>
            <fvm:method name="reader1">
                  <fvm:return passing="register" datatype="void"/>
                  <fvm:code>
```

```
                    sav.i;

                    ld.i gr8, 1024;
                    out.i gr8, 1;

                    ld.i gr8, 768;
                    out.f gr8, 3000.0;

              loop1:
                    jmp @loop1;

                    rst.i;
                    ret;
              </fvm:code>
        </fvm:method>
        <fvm:method name="temperatureTrigger">
              <fvm:return passing="register" datatype="void"/>
              <fvm:code>
                    sav.i;

                    /* Print Message */

                    ld.is gr11, smr;

                    ld.is gr24, gpr;
                    add.i gr24, gr24, $MESSAGE_0;
                    mcall gr11, 3;

                    ld.i gr8, 768;
                    out.f gr8, 1500.0;

                    rst.i;
                    ret;
              </fvm:code>
        </fvm:method>
  </fvm:methodtable>

  <fvm:interfaces>
  </fvm:interfaces>
</fvm:source>
```

Fichier BvhThermocoupleTest1.bncl


```
<fvm:module xmlns:fvm="urn:fvm:module">

      <fvm:informations>
            <fvm:module
                  architecture="{4E992097-B795-4824-BF3D-45B4FBE1F959}"
                  uuid="{D5658875-48FB-415c-BE03-670D811F83DB}"/>
            <fvm:version>1.0.0.6</fvm:version>
            <fvm:author>TEST</fvm:author>
            <fvm:organisation>TEST</fvm:organisation>
            <fvm:support>
                  <fvm:name>TEST</fvm:name>
                  <fvm:phone>1-450-649-9774</fvm:phone>
                  <fvm:email>etienne.fortin@sympatico.ca</fvm:email>
            </fvm:support>
            <fvm:log>
                  <fvm:entry
                        author="Assembler"
                        date="2002-9-27"
                        version="1.0.0.6">
                        Assembled from BvhThermocoupleTest1.basm.
                  </fvm:entry>
            </fvm:log>
      </fvm:informations>

      <fvm:staticblock>
            <fvm:data>
                  f1I+CrCM70KidPyKG1IwVFdBUk5JTkcgOiBUZW1wZXJhdHVyZSBle
                  GN1ZWR1ZCBtYXhpbXVtIGFsbG93ZWQuIExvd2VyaW5nIHNwaW5kbG
                  Ugc3B1ZWQuAAAAAAA=
            </fvm:data>
            <fvm:code>
                  DAAAAA==
            </fvm:code>
      </fvm:staticblock>

      <fvm:methodtable>
            <fvm:method name="reader1">
                  <fvm:return passing="register" datatype="void"/>
                  <fvm:code>
                        AwAAABAAgCAABAAAgQCAIAEAAAAQAIAgAAMAAIMAgIAAAAA
                        AAHCnQAgAACAAAAAABAAAAAwAAAA=
                  </fvm:code>
            </fvm:method>
            <fvm:method name="temperatureTrigger">
                  <fvm:return passing="register" datatype="void"/>
                  <fvm:code>
                        AwAAAB4AtgAegIABKACMIRAAAAALALAgAwAAABAAgCAAAwA
                        AgwCAgAAAAAAAcJdABAAAAAwAAAA=
                  </fvm:code>
            </fvm:method>
```

```
    </fvm:methodtable>

    <fvm:interfaces>
    </fvm:interfaces>

</fvm:module>
```

ANNEXE 7

PROGRAMME ÉCRIT EN CODES-G PARAMÉTRIQUES POUR L'USINAGE DE LA CAME

Cette annexe contient le programme d'usinage d'une came écrit en Custom Macro B pour un contrôleur Fanuc 16M. Ce programme permet de modifier certains paramètres, mais impose un nombre de quatre segments de came et une équation bien précise associée à chacun de ces segments.

```
O0001
N1 (POUR VS50M)
N2 (Programme paramétrique)
N3 (15-OCT-2002)
N4 G21
N5 G40 G49 G50
N6 G64 G69 G80
N7 G90 G15 G94
N8 G501
N9 G251
N10 M79
N11 M69
N12 G54
N13 G08 P0
N14 G05 P0
N15 G08 P1 Q1.
N17 G54 G90
N18 G08 P0
N19 G91 G28 Z0.000

N20 G28 X0.000 Y0.000
N21 G28 A0.000 B0.000
N22 T1 M6
N23 T0
N24 S8000 M3
N25 M8
N26 G08 P1 Q1.
N27 G90
N28 G00 A0.000 B0.000
N30 (*** VARIABLES DESCRIPTION ***)
```

```
N33 ($PI = PI 3.14159265)
N33 ($THETA = ANGLE RADS)
N33 ($DTHETA = DELTA ANGLE)
N35 ($B1 = BETA1)
N36 ($B2 = BETA2)
N37 ($B3 = BETA3)
N38 ($B4 = BETA4)
N39 ($L1 = L1)
N40 ($L2 = L2)
N41 ($L3 = L3)
N42 ($L4 = L4)
N43 ($R0 = R0)
N44 ($RR = RR)
N45 (#100 = TOTAL CONTOURING TIME)
N46 (#101 = ZONE 1 MACHINING TIME)
N47 (#102 = ZONE 2 MACHINING TIME)
N48 (#103 = ZONE 3 MACHINING TIME)
N49 (#104 = ZONE 4 MACHINING TIME)
N45 (*** VARIABLE VALUE ASSIGNATION ***)
N50 $PI = 3.14159265
N51 $THETA = 0
N51 $DTHETA = 0.01
N52 $B1 = 3 * $PI / 4
N53 $B2 = $PI / 3
N54 $B3 = 11 * $PI / 24
N55 $B4 = $B3
N56 $L1 = 30.
N57 $L2 = 20.
N58 $L3 = 20.
N59 $L4 = 70.
N60 $R0 = 30.
N61 $RR = 4.7625
N70 (*** COMPUTING APPROACH POINT X AND Y ***)
N71 $THETA = 0
N100 $S1 = $L1 * [$THETA / $B1 - 1 / $PI * SIN[ 180 * $THETA / $B1]]
N101 $V1 = $L1 / $B1 * [1 - COS[ 180 * $THETA / $B1]]
```

```
N102 $R = $R0 + $S1
N103 $X1 = $R * COS[180 * $THETA / $PI]
N104 $Y1 = $R * SIN[180 * $THETA / $PI]
N105 $THETA = $THETA + $DTHETA
N150 X[$X1] Y[$Y1]
N151 G43 Z12.000 H1
N152 Z1.250
N153 G94 G01 Z-5.000 F80
N154 G17
N97 #3001 = 0
N98 (*** CAM ZONE 1 ***)
N99 WHILE [$THETA] < [$B1] DO1
N100 $S1 = $L1 * [$THETA / $B1 - 1 / $PI * SIN[ 180 * $THETA / $B1]]
N101 $V1 = $L1 / $B1 * [1 - COS[ 180 * $THETA / $B1]]
N102 $R = $R0 + $S1
N103 $X1 = $R * COS[180 * $THETA / $PI]
N104 $Y2 = $R * SIN[180 * $THETA / $PI]
N105 $THETA = $THETA + $DTHETA
N106 G01 X[$X1] Y[$Y1]
N107 END1
N108 #101 = #3001
N197 (*** CAM ZONE 2 ***)
N198 $THETA = $B1
N199 WHILE [$THETA] < [$B1 + $B2] DO2
N200 $NU = $THETA - $B1
N200 $S2 = $L1 + $L2 * $NU / $B2
N201 $V2 = $L2 / $B2
N202 $R = $R0 + $S2
N203 $X2 = $R * COS[180 * $THETA / $PI]
N204 $Y2 = $R * SIN[180 * $THETA / $PI]
N205 $THETA = $THETA + $DTHETA
N206 G01 X[$X2] Y[$Y2]
N205 END2
N206 #102 = #3001
N297 (*** CAM ZONE 3 ***)
N298 $THETA = $B1 + $B2
```

```
N299 WHILE [$THETA] < [$B1 + $B2 + $B3] DO3
N300 $NU = $THETA - $B1 - $B2
N301 $S3 = $L1 + $L2 + $L3 * SIN[180 * $NU / [2 * $B3]]
N302 $V3 = $PI * $L3 / [2 * $B3] * COS[180 * $NU / [2 * $B3]]
N303 $R = $R0 + $S3
N304 $X3 = $R * COS[180 * $THETA / $PI]
N305 $Y3 = $R * SIN[180 * $THETA / $PI]
N306 $THETA = $THETA + $DTHETA
N307 G01 X[$X3] Y[$Y3]
N308 END3
N309 #103 = #3001
N397 (*** CAM ZONE 4 ***)
N398 $THETA = $B1 + $B2 + $B3
N399 WHILE [$THETA] < [2 * $PI] DO4
N400 $NU = $THETA - $B1 - $B2 - $B3
N401 $S4 = $L4 * [1 - 2.63415 * [$NU * $NU /[$B4 * $B4]] + 2.78055 *
[$NU * $NU * $NU * $NU * $NU / [$B4 * $B4 * $B4 * $B4 * $B4]] + 3.17060
* [$NU * $NU * $NU * $NU * $NU * $NU / [$B4 * $B4 * $B4 * $B4 * $B4 *
$B4]] -6.87795 * [$NU * $NU * $NU * $NU * $NU * $NU * $NU / [$B4 * $B4
* $B4 * $B4 * $B4 * $B4 * $B4]] +2.56095 * [$NU * $NU * $NU * $NU *
$NU * $NU * $NU * $NU / [$B4 * $B4 * $B4 * $B4 * $B4 * $B4 * $B4 *
$B4]]]
N402 $V4 = $L4 / $B4 * [ 13.90275 * [$NU * $NU * $NU * $NU / [$B4 * $B4
* $B4 * $B4]] - 5.26830 * [$NU / $B4] + 19.02360 * [$NU * $NU * $NU *
$NU * $NU / [$B4 * $B4 * $B4 * $B4 * $B4]] - 43.14565 * [$NU * $NU *
$NU * $NU * $NU * $NU / [$B4 * $B4 * $B4 * $B4 * $B4 * $B4]] + 20.48760
* [$NU * $NU * $NU * $NU * $NU * $NU * $NU / [$B4 * $B4 * $B4 * $B4 *
$B4 * $B4 * $B4]]]
N403 $R = $R0 + $S4
N404 $X4 = $R * COS[180 * $THETA / $PI]
N405 $Y4 = $R * SIN[180 * $THETA / $PI]
N406 $THETA = $THETA + $DTHETA
N407 G01 X[$X4] Y[$Y4]
N408 END 4
N409 #100 = #3001
N410 #104 = #3001
```

```
N411 #104 = #104 - #103
N412 #103 = #103 - #102
N413 #102 = #102 - #101
N500 (*** RETRACT AND FINISH ***)
N129 Z12.000
N130 G49
N131 G08 P0
N132 G91 G28 Z0.000 M9
N133 G28 X0.000 Y0.000
N134 G28 A0.000 B0.000
N135 T0 M6
N136 T0
N137 G90 G94
N138 M30
```

ANNEXE 8

PROGRAMME ÉCRIT EN LANGAGE C POUR L'USINAGE DE LA CAME

Cette annexe contient les fichiers source en langage C utilisés lors du test d'usinage de la came. Ces fichiers sont compilés à l'aide du compilateur C pour l'architecture BNCL et sont ensuite assemblés en modules exécutables à l'aide de l'assembleur BNCL.

Fichier bvh_macros.h

```
#define          BNCL_START_DEVICE \
                       asm("push.i gr8;");      \
                       asm("ld.i gr8, 0;");     \
                       asm("out.i gr8, -1;");   \
                       asm("pop.i gr8;");

#define          BNCL_STOP_DEVICE  \
                       asm("push.i gr8;");      \
                       asm("ld.i gr8, 1;");     \
                       asm("out.i gr8, -1;");   \
                       asm("pop.i gr8;");

#define          BNCL_OUTPUT_POINT(x, y, z)     \
                       asm("push.i gr8;");      \
                       asm("push.f fr8;");      \
                       asm("ld.i gr8, 256;");   \
                       asm("out.f gr8, %0;" : : "f" (x)); \
                       asm("add.i gr8, gr8, 2;");     \
                       asm("out.f gr8, %0;" : : "f" (y)); \
                       asm("add.i gr8, gr8, 2;");     \
                       asm("out.f gr8, %0;" : : "f" (z)); \
                       asm("add.i gr8, gr8, 2;");     \
                       asm("out.f gr8, 1.0;"); \
                       asm("pop.f fr8;");       \
                       asm("pop.i gr8;");

#define          BNCL_OUTPUT_STRING(str) \
                       asm("ld.i gr24, %0;" : : "r" (str));       \
                       asm("scall 3;");  \
                       asm("scall 0;");

#define          BNCL_OUTPUT_FLOAT(fl)    \
                       asm("ld.f fr24, %0;" : : "f" (fl)); \
                       asm("scall 2;");  \
                       asm("scall 0;");

#define          BNCL_OUTPUT_INTEGER(in) \
                       asm("ld.i gr24, %0;" : : "r" (in)); \
                       asm("scall 1;");  \
                       asm("scall 0;");

#define          BNCL_OUTPUT_ENDLINE      \
```

```
                        asm("scall 0;");

#define            BNCL_WAIT_FOR_INTERRUPT \
                        asm("scall 1;");
```

Fichier came.h

```
/* Header file defining available came profile computation
   functions. This file is used by the came machining cycle.

   Copyright (C) 2002 Etienne Fortin

   2002-09-27 - Etienne Fortin
           - Creation. For now, only functions needed to compute test
             scenarios of the BNCL Architecture are availables.
*/

#ifndef __CAME__H__
#define __CAME__H__


typedef float (*dispFunc)(float, float, float, float);
typedef float (*speedFunc)(float, float, float, float);


/* Function used to machine a came profile. */
int gencame(float b[], float Lp[], float L[], float R0, float Rr,
             dispFunc fct[], speedFunc dfct[], int num, float inc,
float err);



/* Functions used to compute came profile. */

/* Type 1 cycloid */
float came_C1(float A, float b, float Lp, float L);

//float came_C2(float A, float b, float Lp, float L);

//float came_C3(float A, float b, float Lp, float L);

//float came_C6(float A, float b, float Lp, float L);

//float came_H1(float A, float b, float Lp, float L);

/* Type 2 harmonic */
float came_H2(float A, float b, float Lp, float L);
```

```
//float came_H5(float A, float b, float Lp, float L);

//float came_H6(float A, float b, float Lp, float L);

/* Type 2 polynomial */
float came_P2(float A, float b, float Lp, float L);

//float came_S1(float A, float b, float Lp, float L);

/* Constant speed */
float came_V1(float A, float b, float Lp, float L);


/* Functions used to compute roller displacement speed. */

/* Type 1 cycloid */
float came_V_C1(float A, float b, float Lp, float L);

//float came_V_C2(float A, float b, float Lp, float L);

//float came_V_C3(float A, float b, float Lp, float L);

//float came_V_C6(float A, float b, float Lp, float L);

//float came_V_H1(float A, float b, float Lp, float L);

/* Type 2 harmonic */
float came_V_H2(float A, float b, float Lp, float L);

//float came_V_H5(float A, float b, float Lp, float L);

//float came_V_H6(float A, float b, float Lp, float L);

/* Type 2 polynomial */
float came_V_P2(float A, float b, float Lp, float L);

//float came_V_S1(float A, float b, float Lp, float L);

/* Constant speed */
float came_V_V1(float A, float b, float Lp, float L);


#endif /* __CAME__H__ */
```

Fichier math.h

```
/* Header file for common mathematical functions.
```

```
   Copyright (C) 2002 Etienne Fortin

   2002-09-27 - Etienne Fortin
            - Creation
*/


#ifndef __MATH_H__


/* Constants */

extern float pi;


/* Trigonometric functions */

/* Sine */
float sin(float a);
/* Cosine */
float cos(float a);



/* Power functions. */
float pow(float a, float b);



#endif /* __MATH_H__ */
```

Fichier came.c

```
/* Implementation file for the came profile computation
   functions.

   Copyright (C) 2002 Etienne Fortin

   2002-09-27 - Etienne Fortin
            - Creation.
*/


#include "bvh_macros.h"

#include "came.h"
#include "math.h"
```

```
/* ==================================================================
*/
/*                         Generation function
*/
/* ==================================================================
*/



/* Generates points of a came.
      b     Array of betas. Size = num.
      Lp    Array of precedent length. Size = num.
      L     Array of length. Size = num.
      fct   Array of functions of each portion of the came. Size = num.
      dfct  Array of derivatives of functions of each portion of the
            came. Size = num.
      num   Number of portions of the came.
      inc   Increment to use, in radians, for points generation.
      err   Tolerable error. The error is the distance between two
            points and not the "cordal error".*/
int gencame(float b[], float Lp[], float L[], float R0, float Rr,
             dispFunc fct[], speedFunc dfct[], int num, float inc,
float err) {
      float a = 0;        // Current absolute angle.
      float A = 0;   // Angle relative to start of current segment.
      float j = 0;        // Index of first point.
      float d[num];       // Angles limits.
      float br[2];        // Lower and upper bound of a came segment.

      float s;            // Current displacement of roller.
      float ds;           // Current speed of roller.


      // Total number of segment.
      int    sz = num;

      float x, y; // Coordinate of point.

      int          cnt = 0;    // Point counter.


      // Create absolute angles limits for each
      // segment of the came.
      d[0] = b[0];
      for(int i = 0;i < sz-1;i++) {
            d[i+1] = d[i] + b[i+1];
      }
```

```
	// Main Process loop.
	while(a < d[sz-1]) {
		//BNCL_OUTPUT_STRING("LOOP!")
		br[0] = 0;
		br[1] = 0;

		// Compute displacement and speed according to
		// current segment.
		for(int i = 0;i < sz;i++) {
			br[0] = br[1];
			br[1] = d[i];
			if(a >= br[0] && a < br[1]) {
				A = a - br[0];
				s = fct[i](A, b[i], Lp[i], L[i]);
				ds = dfct[i](A, b[i], Lp[i], L[i]);
			}
		}

		x = (R0 + s) * cos(a);
		y = (R0 + s) * sin(a);

		BNCL_OUTPUT_POINT(x, y, 0.0)

		a += inc;
		cnt++;

	}

	return cnt;
}




/* ===============================================================
*/
/*                            Profiles
*/
/* ===============================================================
*/


/**/
float came_C1(float A, float b, float Lp, float L) {
	float s = Lp + L*(A/b - 1/pi*sin(pi*A/b));
	return s;
}


/**/
float came_H2(float A, float b, float Lp, float L) {
	float s = Lp + L*(sin(pi*A/(2*b)));
```

```
	return s;
}


/**/
float came_P2(float A, float b, float Lp, float L) {
	float s = Lp + L*(1 - 2.63415*pow(A/b, 2) + 2.78055*pow(A/b, 5) +
3.17060*pow(A/b, 6) - 6.87795*pow(A/b, 7) + 2.56095*pow(A/b, 8));

	return s;
}


/**/
float came_V1(float A, float b, float Lp, float L) {
	float s = Lp + L*(A/b);
	return s;
}



/* ===========================================================
*/
/*                          Speeds
*/
/* ===========================================================
*/

/**/
float came_V_C1(float A, float b, float Lp, float L) {
	float v = L/b*(1 - cos(pi*A/b));;
	return v;
}


/**/
float came_V_H2(float A, float b, float Lp, float L) {
	float v = pi*L/(2*b)*(cos(pi*A/(2*b)));
	return v;
}


/**/
float came_V_P2(float A, float b, float Lp, float L) {
	float v = L/b*(-5.26830*(A/b) + 13.90275*pow(A/b, 4) +
19.02360*pow(A/b, 5) - 48.14565*pow(A/b, 6) + 20.48760*pow(A/b, 7));
	return v;
}


/**/
float came_V_V1(float A, float b, float Lp, float L) {
	float v = L/b;
```

```
        return v;
}
```

Fichier math.c

```
/* Implementation file for common mathematical functions.

   Copyright (C) 2002 Etienne Fortin

   2002-09-27 - Etienne Fortin
              - Creation.
*/

#include "math.h"


/* ================================================= */
/*                    Constants                      */
/* ================================================= */

float pi = 3.1415927;


/* ================================================= */
/*            Trigonometric functions                */
/* ================================================= */


/**/
float sin(float a) {
        float r;

        asm("sin.f %0, %1;" : "=f" (r) : "f" (a));

        return r;
}

/**/
float cos(float a) {
        float r;
```

```
        asm("cos.f %0, %1;" : "=f" (r) : "f" (a));

        return r;
}



/* ============================================== */
/*                 Power functions                */
/* ============================================== */

/**/
float pow(float a, float b) {
        float r;

        asm("pow.f %0, %1, %2;" : "=f" (r) : "f" (a), "f" (b));

        return r;
}
```

Fichier main.c

```
#include "bvh_macros.h"
#include "math.h"
#include "came.h"



/**/
void came_test1() {
        BNCL_OUTPUT_STRING("START came_test1")

        float           b[4]  = { 3*pi/4, pi/3, 11*pi/24, 11*pi/24 };
        float           Lp[4] = { 0, 30, 50, 0 };
        float           L[4]  = { 30, 20, 20, 70 };
        float           R0 = 30;
        float           Rr = 4.7625;


        dispFunc        fct[4]  = { &came_C1, &came_V1, &came_H2,
&came_P2 };
        speedFunc       dfct[4] = { &came_V_C1, &came_V_V1, &came_V_H2,
&came_V_P2 };

        int                     num = 4;
        float           inc = 0.001;
        float           err = 0.01;
```

```
    /*void gencame(float b[], float Lp[], float L[], float R0, float
Rr,
            dispFunc fct[], speedFunc dfct[], int num, float inc,
float err);*/

    BNCL_START_DEVICE
   int cnt = gencame(b, Lp, L, R0, Rr, fct, dfct, num, inc, err);
    BNCL_STOP_DEVICE
    BNCL_WAIT_FOR_INTERRUPT



    BNCL_OUTPUT_ENDLINE
    BNCL_OUTPUT_INTEGER(cnt)

   BNCL_OUTPUT_STRING("END came_test1")
}
```

BIBLIOGRAPHIE

1 Schofield, S., Wright, P. (1998). Open Architecture Controllers for Machine Tools, Part 1: Design Principles. *Journal of Manufacturing Science and Engineering*, Volume 120, 417-424

2 Mori, M., Yamazaki, K. et al. (2001) A study on development of an open servo system for intelligent control of a CNC machine tool. *CIRP Annals - Manufacturing Technology,* Volume 50 numéro 1, 247-250

3 Okafor, A.C., Ertekin, Yalcin M. (2000) Derivation of machine tool error models and error compensation procedure for three axes vertical machining center using rigid body kinematics. *International Journal of Machine Tools & Manufacture,* Volume 40, 1199-1213

4 Farouki, Rida T., Manjunathaiah, Jairam, Yuan, Guo-Feng (1999) G codes for the specification of Pythagorean-hodograph tool paths and associated feedrate functions on open-architecture CNC machines. *International Journal of Machine Tools & Manufacture,* Volume 39, 123-142

5 Mize, Christopher D., Ziegertb, John C. (2000) Neural network thermal error compensation of a machining center. *Precision Engineering*, Volume 24, 338-346

6 Lin, Rong-Shine (2000) Real-time surface interpolator for 3-D parametric surface machining on 3-axis machine tools. *International Journal of Machine Tools & Manufacture,* Volume 40, 1513-1526

7 Klein, George (1985). An Approach to Motion Control System Standardization. *Proceedings of the International MOTORCON Conference 1985*, 119-126

8 Grossman, David D. (1986). Opportunities of Research on Numerical Control Machining. *Communications of the ACM (Association for Computing Machinery)*, 29(6), 515-522

9 Ulmer Jr., Bernard C., Kurfess, Thomas R. (1998). Integration of an Open Architecture Controller with a Diamond Turning Machine. *Mechatronics*, 349-361

10 Rober, Stephen J., Shin, Yung C. (1995). Modeling and Control of CNC Machines Using a PC-Based Open Architecture Controller. *Mechatronics*, 5(4), 401-420

11 Weck, Manfred, Kahmen, Andreas (2001) Open Real-Time Interfaces for Monitoring Applications within NC-Control Systems. *Proceedings of the 2001 IEEE International Conference on Robotics & Automation*, 199-204.

12 Altintas, Y., Peng, J. (1990) Design and analysis of a modular CNC system. *Computers in Industry*, 13(4), 305-316.

13 Yamazaki, Kazuo, Hanaki, Yoshimaromori, Masahiko Tezuka, Kasusaka (1997) Autonomously Proficient CNC Controller For High-Performance Machine Tools Based On An Open Architecture Concept. *CIRP Annals – Manufacturing Technology*, 47(1), 275-278

14 Yellowley, Ian, Oldknow, Kevin, Ardekani, Ramin (1999). A Critical Evaluation of Open Architecture CNC Design. *SPIE Conference on Sensors and Controls for Intelligent Machining September 1999*, Volume 3832, 6-16

15 Engels, Daniel W., Sarma, Sanjay E. (1999). Towards an Open Architecture Specification Language for Machine Control. *SPIE Conference on Sensors and Controls for Intelligent Machining September 1999*, Volume 3832, 17-25

16 Proctor, Fred, Shackleford, Will, Yang, Charles (1995). Simulation and Implementation of an Open Architecture Controller. *Proceedings of SPIE octobre 1995*, Volume 2596, 196-204

17 Yamazaki, Kazuo (1996). Open Architecture CNC Controller in the USA. *Proceedings of the 7th IMEC Conference*, Tokyo, 204-218

18 Seethaler, R.J., Yellowley, I. (2000) Process Control And Dynamic Process Planning. *International Journal of Machine Tools and Manufacture*, 40(2), 239-257

19 Wikipedia, the free encyclopedia. *Micro Channel Architecture*, [En Ligne] http://www.wikipedia.org/wiki/Micro_Channel_Architecture (consulté le 7 novembre 2002).

20 Satoru, Fujita, Yoshida, Toshiro (1996). OSE : Open System Environment for Controller – Development of an Open Architecture CNC with OSEC Specification. *Proceedings of the 7th IMEC Conference*, Tokyo.

21 Sarma, S.E., Schofield, S., Stori, J., MacFarlane, J., Wright, P.K. (1996). Rapid Product Design from Detail Design. *Computer Aided Design*, 28(5), 383-392.

22 Conklin, Edward K., Rather, Elizabeth D. (1998). *Forth Programmer's Handbook*. Forth Inc.

23 Bottorf, Jan (1999). *The Argument Permutation (ASP) Virtual Machine Instruction Set*. Paradigm Matrix Inc.

24 International Standardization Organisation (ISO) (2001). *Industrial automation systems and integration : Physical device control*, Projet de norme ISO14649-1.

25 Williams, T.J. (1973). CAM and NC Software: Needs for and Benefits from Generalized and Multi-Industry Standardized Languages. *Proceedings of the IFIP/IFAC International Conference on Programming Languages for Machine Tools (PROLAMAT) 73*, Budapest, 1-29.

26 Instrument Society of America (1972). *Industrial Computer System Fortran Procedures for Executive Functions and Process Input-Output*. Standard S61.1, Pittsburg: ISA

27 Lambda, The Programming Languages Weblog (2002). *Critiques*, [En Ligne], http://lambda.weblogs.com/newsItems/viewDepartment$critiques (consulté le 8 novembre 2002)

28 Springer, A. (1979) *A comparison of language C and Pascal.* Rapport technique G320-2128, IBM

29 Thimberly, Harold (1999) Critique of Java, *Software – Pratice and Experience*, 29(5), 457-478

30 Weck, M., Wolf, Jochen, Kiritsis, Dimitris (2001) The STEP compliant NC programming interface. *Proceedings of the 2001 IMS (Intelligent Manufacturing Systems) Forum*, Ascona

31 STEP-NC Consortium. *Need for a New Data Interface for NC Programming*, [En Ligne], http://www.step-nc.org/html/introduction_goal.htm (consulté le 29 avril 2002)

32 Fanuc (1995). *FANUC Series 16-MB, FANUC Series 18-MB, FANUC Series 160-MB, FANUC Series 180-MB, Manuel de l'utilisateur*. Fanuc LTD.

33 Lynch, Mike (1997). *Parametric Programming for Computer Numerical Control Machine-Tools and Touch Probes*. Michigan : Society of Manufacturing Engineer

34 Dijkstra, Edsger W. (1968). GOTO Statement Considered Harmful. *Communications of the ACM (Association for Computing Machinery)*, 11(3), 147-148

35 Lindholm, Tim, Yellin, Frank (1999). *The Java Virtual Machine Specification* (2e éd.). Sun Microsystems Inc.

36 Brown, P.J. (1972). Levels of Language for Portable Software. *Communications of*

the ACM (Association for Computing Machinery), 15(12), 1059-1062

37 Patterson, David A., Lew, Karl, Tuck, Richard (1979). Towards an Efficient, Machine-Independant Language for Microprogramming, *IEEE Transactions on Computers 1979*, 22-35

38 Gough, K. John, Corney, Diane (2000). Evaluating the Java Virtual Machine as a Target for Languages Other Than Java, *Lectures Notes in Computer Science*, 1897, 278-290

39 Stroustrup, Bjarne (1999) *C++* (3e Éd.). New Jersey : Murray Hill

40 Krutsch, Kenneth F., Cargo, David S., Howlett, Virginia (2001) *Java Custom UI Components* (1er Éd.). Wrox Press

41 Intel Corporation (1999) *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*. Intel Corporation

42 Perl Mongers. *Perl 6 Internals*, [En Ligne]. http://dev.perl.org/perl6/talks/tpc5-internals/perl 6 internals.html (consulté le 31 avril 2002)

43 ECMA (2001). *Common Language Infrastructure*, Norme ECMA335

44 Hammond, Mark (2002) *Python for .NET: Lessons Learned*, ActiveState Tool Corporation

45 Stob, Verity (2002) Caught in .NET, *Dr. Dobb's Journal September 2002*

46 Mabie H., H., Ocvrik W., F. (1978) *Mechanisms and dynamics of machinery* (3e éd.). New-York : John Wiley & Sons

47 Rogerson, Dale (1997) *Inside COM* (1er éd.). Microsoft Press.

48 Ray, Erik T. (2001) *Learning XML* (1er éd.). O'Reilly & Associates.

49 Cagle, Kurt, Corning, Michael, Diamond, Jason & al. (2001) *Professional XSL* (1er éd.). Wrox Press Inc.

50 Schwartz, Randal L., Phoenix, Tom (2001) *Learning Perl* (3e éd.). O'Reilly & Associates.

51 Red Hat. *Cygwin* [En ligne]. http://www.cygwin.com (consulté le 17 décembre 2002).

52 Griffith, Arthur (2002) *GCC: The Complete Reference* (1er éd.). McGraw-Hill Osborne Media.

53 STEP-NC Consortium. *Overview of the current projects related to STEP-NC*, [En Ligne], http://www.step-nc.org/html/projects.htm (consulté le 17 décembre 2002).

54 Lozano, Jerry, Baker, Art (2000) *The Windows 2000 Device Driver Book : A Guide For Programmers* (3e éd.). Prentice Hall PTR