
5. L'Intelligence des Environnements de Programmation

Harald WERTZ
Université Paris VIII
Département Informatique
2, rue de la Liberté
93526 S^t-Denis Cedex 02
hw@litp.ibp.fr

Résumé

Ce chapitre fait un survol des activités de recherche autour des environnements de programmation. Un environnement de programmation est un ensemble d'outils d'aide à l'exécution, à l'édition, à la documentation, à la lecture, à l'annotation, à la vérification, à l'observation et à la maintenance de programmes. Les environnements, ou leurs caractéristiques, que nous examinerons ne concernent que les aspects qui donnent une aide efficace au concepteur et/ou à l'implémenteur de programmes. Nous distinguerons les environnements de programmation proprement dits — c'est-à-dire les aides directes à la programmation — des environnements de génie logiciel, qui ont pour but de fournir des aides tout au long du processus de développement d'un projet : de l'analyse des demandes, en passant par les spécifications, à l'implémentation et la maintenance.

5.1. Historique et Motivation

Dans la mesure où tout le développement des langages informatiques et des systèmes opératoires peut être considéré comme un effort pour faciliter les tâches des programmeurs, il peut être considéré comme une amélioration continue des environnements de programmation. Au début, la programmation se faisait au niveau binaire : soit par des contacts physiques à établir, soit par des rudiments de programmation. On déterminait la suite de zéros et de uns dont l'interprétation par la machine aboutissait au calcul désiré. Très vite les techniciens et chercheurs utilisant ces machines

ont été confrontés au problème de la correction des programmes ainsi écrits. Notons que la correction de programmes suppose

- qu'on ait compris le fonctionnement effectif du programme, son déroulement et les changements des valeurs qui y sont associées,
- qu'on ait compris les lieux exacts où est produit un calcul différent de celui désiré, et
- qu'on sache par quelle modification du programme ce comportement non désiré peut être éliminé.

Notons que la lecture même des programmes (à cette époque elle se faisait encore partiellement par la consultation du plan de câblage) ne donnait aucune indication du rapport conceptuel entre le programme et la solution du problème modélisée par ce programme.

Cette difficulté à faire correspondre le texte des fragments de programmes avec des concepts faisant partie du problème, a incité, dès le début des années 50, divers chercheurs [Gill51, Wilkes51] à proposer des outils d'observation de l'exécution des programmes. C'est à cette époque également qu'on a introduit déjà au niveau matériel, des moyens d'exécuter les programmes en pas-à-pas (une manière d'exécuter le programme instruction par instruction avec l'option de pouvoir examiner le contenu des registres ou des mots mémoires choisis). Quelques unes des premières machines — tel que l'EDSAC 2 [Wilkes57, Barron63] — proposaient même un mécanisme (toujours au niveau matériel) correspondant au concept très moderne de démons¹ : elles disposaient d'un mécanisme permettant d'observer l'exécution des programmes et d'interrompre cette exécution continue à l'instant où des événements prédéfinis, relatifs au calcul ou au traitement, ont lieu, pour ensuite examiner, comme dans le pas-à-pas, des registres et des mots mémoires choisis.

C'est, bien entendu, ce problème de l'illisibilité des programmes binaires, ainsi que les problèmes liés au niveau très rudimentaire des instructions, qui a amené les informaticiens à développer des langages de programmation. Au début ce n'était qu'une représentation mnémonique des instructions de base. Ce langage s'appelait le langage assembleur. Bien qu'on soit encore loin des instructions sophistiquées des langages actuels, il est clair qu'il est plus aisé de lire un texte comme :

mov r1, r2

que l'instruction binaire correspondante² qui est :

1. Techniquement, un démon est une procédure lancée automatiquement dès qu'un événement computationnel prédéfini survient. L'instant précis de ce lancement — dans le déroulement séquentiel du programme — ne peut pas être prévu à priori.

01000100100

D'une certaine manière, cet exemple capte toute la problématique des environnements de programmation : dans tous les cas il s'agit de trouver des moyens pour rendre l'écriture, la lecture et la modification des programmes plus aisées, d'une part en permettant des représentations conceptuellement plus simples et, d'autre part, en mettant à la disposition des programmeurs des outils permettant de limiter la complexité perçue des programmes.

Rappelons qu'un des problèmes majeurs de la programmation réside dans ce que Terry Winograd [Winograd75] nomme *la barrière de complexité*. Cette barrière de complexité est, principalement, une limite de nos capacités de mémorisation : au fur et à mesure de la croissance d'un programme, le programmeur se construit, incrémentalement, une *image* mentale de l'interaction entre ses différentes parties, de l'utilisation et de l'utilité des variables, du flux de contrôle et des données. Cette image, qui peut être très détaillée pour des programmes de faible dimension, doit, parallèlement à l'agrandissement, se transformer en une image de plus en plus abstraite, ne gardant que des aspects globaux de l'interaction et de la relation entre l'implémentation et la spécification (qui peut être ou non formelle), et éliminant de plus en plus les détails locaux de l'implémentation. Cette modification de la représentation interne que le programmeur possède de son programme, est liée à des limitations inhérentes au fonctionnement cognitif humain (cf. par exemple le papier bien connu de [Miller56]).

Afin de parer à ces insuffisances humaines, les chercheurs en informatique proposent différentes solutions :

- le développement de langages de programmation de *haut niveau* pour la programmation symbolique (tels que LISP, Prolog, Smalltalk etc). Nous résumerons ce développement dans le paragraphe suivant.
- le développement de systèmes d'aide à la programmation tels que des systèmes permettant la sauvegarde de différentes *versions* d'un programme, des systèmes construisant des descriptions du flux de contrôle ou/et des données, des systèmes permettant une exécution contrôlée des programmes, des outils de traces d'exécutions, etc. La description de tels systèmes d'outils intégrés constituera la partie centrale de ce chapitre.

2. **mov** est une mnémonique pour move, synonyme de transfert, **r1** pour registre 1 et **r2** pour registre 2. Cette instruction copie le contenu du registre 1 dans le registre 2. Le code binaire peut se lire comme 010 pour le code correspondant à l'instruction de copie (ou de transfert), 001 désigne l'adresse no 1, le 0 suivant indique que c'est l'adresse d'un registre, et la séquence 0100 restante peut être interprétée comme l'adresse 2 (binaire 010) d'un registre (le dernier 0 est — comme préalablement — une indication du mode d'adressage).

- le développement de systèmes de programmation transformationnelle. Ces systèmes visent à formaliser les phases précédant la programmation proprement dite (principalement les phases de l'analyse des demandes et des spécifications) pour ensuite compiler ces formalisations dans des programmes efficaces. En dernier ressort, ces recherches ne sont rien d'autre que la reprise des recherches autour des langages de programmation de haut niveau à un niveau encore plus élevé ou plus conceptuel. Nous présentons ces recherches brièvement dans la dernière partie de ce chapitre.

5.2. Les Langages De Programmation De Haut Niveau

Les langages assembleurs, bien qu'ils aient apporté un changement significatif dans les techniques de programmation, se sont très vite révélés insuffisants pour le développement de programmes complexes. Ceci pour trois raisons principales :

- La portabilité : étant donné que l'assembleur n'est qu'une écriture mnémotechnique pour les instructions disponibles sur un ordinateur donné, chaque ordinateur possède son propre langage assembleur. Avec la multiplication des machines et des types différents de machines, il devenait de plus en plus nécessaire de pouvoir transporter un programme d'une machine vers une autre. Si le programme est écrit en langage assembleur, un tel transport de logiciel revient quasiment à réécrire complètement le programme : par définition, les programmes assembleurs ne sont pas facilement portables.
- La complexité des applications : si au début du développement de l'informatique, les personnes écrivant des programmes étaient des spécialistes de l'informatique, l'élargissement des domaines d'application de ces machines demandait très tôt des connaissances d'autres spécialités. Ecrire un programme de calcul physique nécessite de connaître à la fois la programmation, les mathématiques nécessaires à ce calcul et les particularités du domaine physique modélisé. Il semble plus simple que les informaticiens créent des langages spécialisés pour des applications particulières plutôt que d'apprendre chacun des domaines d'application possibles. Par exemple, le langage Fortran, le premier langage de haut niveau, a été créé pour permettre à des mathématiciens ou des physiciens de pouvoir eux-mêmes écrire leurs programmes.
- la complexité des programmes : comme nous l'avons dit plus haut, un programme écrit dans un langage informatique devient rapidement très complexe, d'une part à cause des interactions entre les différentes parties du programme et, d'autre part à cause de la quasi impossibilité de faire

clairement correspondre les concepts modélisés avec des blocs d'instructions. Souvent c'est une non-compréhension partielle de ces interactions qui est la source d'erreurs de programmation difficiles à détecter.

Les voies de solution choisies pour chacun de ces trois problèmes ont été diverses. Les questions de portabilité ont été résolues par le développement de langages compilés. A partir de l'instant où une machine disposait d'un compilateur pour un certain langage, on pouvait, à moindre frais y porter des logiciels écrits sur d'autres machines dans le même langage. Deux des premiers langages compilés, Fortran [FOR54] et Cobol [COB73] ont été pour longtemps les langages disponibles sur la majorité des ordinateurs.³

Notons que ces deux langages s'adressaient à une clientèle très différente : si Fortran (le FORMula TRANslator) s'adressait à des personnes ayant besoin de faire des calculs numériques, donc principalement à des mathématiciens et des physiciens, Cobol s'adressait de prime abord aux personnes voulant automatiser la gestion. Ainsi, dès les premiers langages compilés, on prenait en compte le deuxième problème évoqué ci-dessous : le problème de la complexité des applications.

Bien entendu, avec l'élargissement des domaines d'application des ordinateurs, le nombre et le type de langages de programmation s'agrandissaient à vue d'oeil. Ainsi, dès 1967, Graham Birtwistle, Ole Dahl, Bjørn Myhrhaug et Kristen Nygaard développaient le langage Simula [Birtwistle73] qui, comme son nom l'indique, est spécialisé dans les logiciels de simulation. A la même époque, les chercheurs en Intelligence Artificielle commençaient à programmer en IPL-V [Newell64], un langage de programmation symbolique qui introduisait le concept de listes, un concept repris, perfectionné et généralisé par l'équipe d'Intelligence Artificielle du MIT avec le langage LISP [McCarthy65].

Ces langages, initialement basés exclusivement sur des interprètes⁴ introduisaient non seulement les concepts de liste et de garbage collector, permettant de récupérer dynamiquement et de réutiliser la mémoire préalablement utilisée et dont le contenu n'est plus nécessaire, mais ils étaient également les premiers langages permettant — à l'intérieur même du langage — de manipuler les programmes écrits dans ce langage. Cette caractéristique de pouvoir manipuler les programmes par des programmes (pendant l'exécution même) posait les premiers jalons de tout un courant de recherches concernant le troisième problème cité ci-dessus : le problème de la complexité des programmes. Ainsi, dès 1967 apparaissaient, en LISP, les premiers pro-

3. Nous faisons volontairement abstraction ici d'un autre langage ancien qu'est Algol-60 [Naur63] et ses successeurs directs Algol-68 [Wijngaarden69] et Algol-W [Sites71]. Ces langages, qui ont contribué énormément au développement des recherches logicielles (Algol-60 était le premier langage avec une instruction de sélection du style *si-alors-sinon*) n'ont jamais réussi à s'imposer à l'extérieur de la communauté de recherche informatique.

4. Les interprètes ne traduisent pas les programmes dans un langage plus proche de la machine, comme le font les compilateurs, mais ils les *interprètent* directement.

grammes de trace d'exécution [Teitelman66], des éditeurs connaissant la structure des programmes [Teitelman65], des programmes permettant d'interrompre temporairement l'exécution des programmes (des points d'arrêt), voire même les premiers évaluateurs partiels [Lombardi64]. Bien entendu, c'est en partant de ces programmes que s'est formé le champ de recherches autour des environnements de programmation.

LISP est le langage le plus utilisé en Intelligence Artificielle, donc destiné à une programmation expérimentale et extrêmement complexe. Cette complexité a induit le développement de tout un ensemble de langages dérivés : Planner [Hewitt72] introduisait un contrôle programmatoire du backtracking⁵, Conniver [McDermott74] simplifiait l'implémentation de Planner et introduisait la possibilité de contextes multiples⁶.

A partir de LISP, Planner et Simula, Alan Kay a conçu le langage Smalltalk [Goldberg83], un langage qui non seulement intègrait la totalité des concepts utilisés dans ces langages, mais qui également a contribué énormément à l'état de l'art des environnements de programmation : par exemple, l'interface du MacIntosh est — en grande partie — une copie de l'interface de Smalltalk. Nous reviendrons sur Smalltalk dans le paragraphe sur les environnements de programmation.

Alain Colmerauer a, indépendamment de LISP et plutôt à partir des techniques de programmation de la démonstration automatique de théorèmes, conçu le langage Prolog [Colmerauer82]. Ce langage permet d'exprimer les programmes comme une suite de théorèmes logiques qu'il s'agit de démontrer.

Ainsi quatre types de programmation — et de langages de programmation — coexistent aujourd'hui : les langages impératifs (Fortran et Pascal, par exemple), les langages fonctionnels (avec LISP en tête), les langages orientés objets (Smalltalk en est l'incarnation la plus connue) et les langages logiques (du style Prolog). Les tendances actuelles sont :

- d'intégrer ces différents styles dans des langages uniques — ceci pour permettre de choisir (en fonction des problèmes et des sous problèmes à résoudre) des styles de programmation différents à l'intérieur d'un même langage. Tao [Takeuchi83] et LOGIN [Aït85] sont deux réalisations intéressantes de ces efforts.

5. Le backtracking est une technique de retour vers des situations antérieures au rencontre d'une situation de blocage.

6. En programmation IA, un contexte est principalement un espace de recherche. Des contextes multiples sont donc la coexistence de plusieurs espaces de recherche hypothétiques.

- d'intégrer, à l'intérieur de ses langages l'ensemble des outils nécessaires au développement et à la mise au point de programmes. C'est ce point que nous allons traiter dans le paragraphe suivant.

5.3. Les Systèmes d'Aide a la Programmation

Avant d'entrer dans le vif du sujet, c'est-à-dire : la description des outils intelligents disponibles à l'intérieur des systèmes d'aide à la programmation, examinons brièvement les étapes dans la construction d'un programme. Ceci nous permettra de distinguer entre les phases de la programmation et celles de sa préparation.

5.3.1. des demandes à la réalisation de programmes

Tout programme débute par la définition d'une tâche qu'il s'agit d'automatiser. Souvent, ce n'est pas l'informaticien qui définit cette tâche, mais la personne qui en est jusqu'alors responsable. C'est ici que les premiers problèmes surgissent : la personne demandeuse d'un service informatique s'exprime en des termes propres à sa problématique, qu'il s'agit — pour l'informaticien — de traduire en des termes programmatoires. Nous appelons cette première phase de traduction d'une problématique (ce qu'il faut faire) vers une autre (comment le faire) la phase d'analyse des demandes ou étude des besoins⁷ (cf. la Figure 1 sur la page suivante). Cette analyse des demandes se fait en collaboration étroite entre le demandeur et l'informaticien et aboutit à une spécification — plus ou moins informelle — des diverses tâches informatiques à réaliser.

C'est à partir de cette spécification informelle que l'informaticien va commencer à construire son programme. Dans la Figure 1, nous distinguons deux voies possibles :

- soit l'informaticien procède d'abord à la réalisation d'un *prototype*⁸ pour commencer à implémenter le programme seulement après validation du prototype,
- soit il ignore cette étape de prototypage et commence directement à développer son programme final.

Bien qu'intuitivement il semble évident que la construction d'un prototype, qui peut être exécuté et validé en interaction avec le demandeur, est préférable à la construction immédiate du programme final, ce n'est que très récemment, avec l'avènement de langages de très haut niveau, adaptés au prototypage, que cette voie a pu être

7. Dans la littérature anglo-saxonne cette phase s'appelle la phase de *requirements analysis*.

8. Un *prototype* est un programme qui modélise le comportement du programme voulu. Principalement, c'est une implémentation — dans un langage de haut niveau — qui fait abstraction des détails techniques et des questions d'efficacité et qui n'essaie de capturer que les activités principales du programme prévu. C'est une sorte de programme *croquis*.

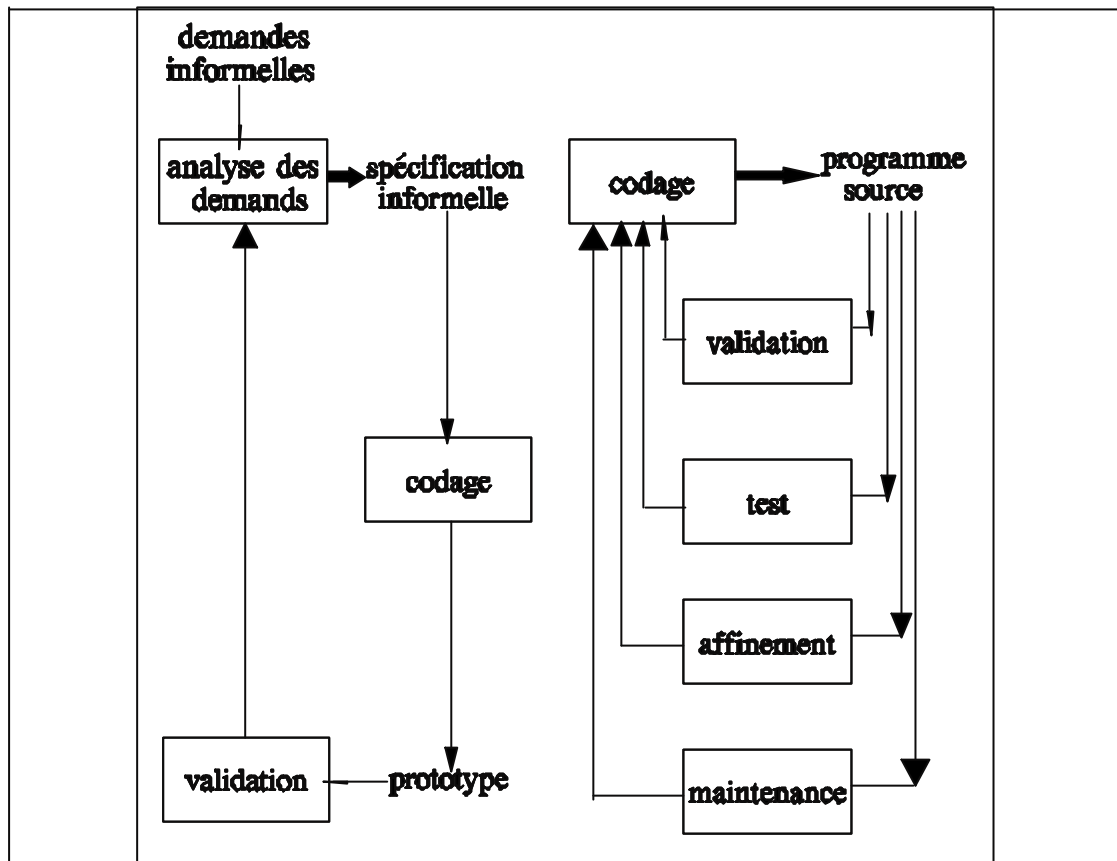


Figure 1 : Développement traditionnel de programmes

suivie. Antérieurement, les coûts du travail et du temps nécessaires à la construction d'un tel prototype étaient trop élevés.

Si un prototype a été construit — c'est-à-dire si l'on a programmé une première version au brouillon du projet — il s'agit ensuite de valider le comportement de ce prototype avec le (ou les) utilisateur(s) final(aux) du programme. Notons que cette phase de validation est extrêmement importante : souvent, la spécification informelle est erronée ou incomplète, ne capturant pas ou seulement partiellement les caractéristiques demandées au programme. La construction d'un prototype aide à économiser des efforts de modification fastidieuses du programme définitif. Les aides disponibles au niveau de la construction (du codage) du prototype et au niveau de sa validation sont les mêmes que celles disponibles pour la construction et la validation du programme définitif. Nous y reviendrons par la suite dans le paragraphe sur la programmation transformationnelle.

Néanmoins, la plupart du temps, après la spécification débute immédiatement la construction du programme définitif. Ce programme — après avoir été testé — doit également être validé par rapport à son comportement et son efficacité. Si une divergence entre la demande et le comportement du programme existe, il faut modifier le programme, le tester et le re-valider ensuite. Sur la Figure 1, qui représente les diverses phases dans la vie d'un programme, ces activités successives sont représentées

par les boucles *codage* \Rightarrow *programme source* \Rightarrow *validation / test / affinement*⁹ / *maintenance* \Rightarrow *codage*.

Examinons, à travers un environnement de programmation type, les divers outils d'aide qui sont actuellement mis à la disposition du programmeur. Nous y retrouverons les boucles que nous venons de mentionner.

bVLISP [Wertz83, Wertz84], l'environnement de programmation expérimental à travers lequel nous exposerons les capacités des outils d'aide à la programmation, est un environnement intégré dans un interprète VLISP [Greussay77]. Le choix d'un langage évolué tel que LISP, comme langage de base pour un environnement de programmation s'impose par le fait que LISP possède, dans sa version de base, déjà tout un ensemble de caractéristiques extrêmement puissantes pour la construction d'environnements. Notons particulièrement que les programmes LISP sont représentés sous forme de listes qui constituent la structure de données principale de LISP. Ceci permet l'accès et la modification de programmes LISP en LISP même, et économise l'effort de construction de parseurs¹⁰ et de décompilateurs¹¹ nécessaires si la représentation des programmes exécutables diverge de leur représentation externe.¹²

5.3.2. L'édition et la documentation de programmes

Mais revenons vers l'activité de programmation. La première chose à faire est d'entrer le programme. Ceci se fait en général avec un éditeur, un programme spécial pour écrire du texte dans un fichier. En bVLISP, l'éditeur est intégré dans l'interprète : toute interaction avec la machine se fait à travers cet éditeur, sans jamais sortir de LISP. L'édition n'est pas organisée autour du texte (c'est-à-dire autour des caractères, des lignes et des paragraphes) mais autour des structures de base de LISP, donc autour des listes, des atomes et des structures spécifiques des diverses commandes et fonctions disponibles. Pour chaque fonction LISP existe un *canevas* qu'il s'agit de remplir. Ainsi, l'édition devient principalement une composition et un remplissage de structures ou de canevas. Par exemple, au lieu de taper linéairement le fragment de programme ci-dessous :

(DE EXEMPLE (VAR) (CONS VAR (CDR VAR)))

le programmeur procède selon le scénario ci-dessous :

-
9. Dans la littérature inspirée par le vocabulaire anglophone, l'affinement s'appelle souvent "tuning".
 10. Un *parseur* est un programme traduisant la représentation externe (donc sous forme d'une chaîne de caractères) en une forme intermédiaire plus facile à traiter par le compilateur.
 11. Un *décompilateur* est un programme traduisant la représentation interne d'un programme en la représentation externe. C'est l'activité inverse du parseur et du compilateur.
 12. Les caractéristiques importantes de LISP sont, entre autres, que la gestion de la mémoire est dynamique, les objets créés par l'utilisateur ont le même statut que les objets primitifs, le traitement de conditions exceptionnelles est programmable et — finalement — LISP est un langage non-typé.

<p>(□)</p> <p>(DE <□nom><paramètres><corps>)</p> <p>(DE EXEMPLE (<□variables>) <corps>)</p> <p>(DE EXEMPLE (<variables>) (CONS <□objet> <liste>))</p> <p>(DE EXEMPLE (<variables>) (CONS <objet> (CDR <□liste>)))</p>	<p>Dès que le programmeur tape une parenthèse ouvrante, l'éditeur insère une parenthèse fermante et place le curseur entre les deux.</p> <p>Après que le programmeur ait tapé DE, l'éditeur insère des mnémoniques à l'endroit des arguments de cette fonction : ici, c'est le canevas de la fonction DE. Le programmeur peut se déplacer d'un mnémonique à l'autre et les remplacer par ses arguments. Le curseur est placé de telle sorte que la suite de la frappe remplace le mnémonique <nom>.</p> <p>Après avoir donné le nom de la fonction, le programmeur a entré une parenthèse ouvrante à l'endroit des paramètres. L'éditeur répond par une parenthèse fermante et le mnémonique adéquat.</p> <p>Le programmeur a choisit de ne pas encore remplir le canevas des variables, mais de commencer immédiatement à écrire le corps de la fonction. Pour cela il lui suffisait de pointer vers le mnémonique <corps>, de taper la parenthèse ouvrante (ce qui lui donnait le canevas général d'un appel de fonction (<fonction><arguments>)) et de remplacer le mnémonique <fonction> par l'appel de la fonction CONS.</p> <p>et ainsi de suite ...</p>
---	---

Les aides que des éditeurs de ce type — communément appelés *éditeurs structuraux* — apportent, sont, premièrement, que l'écriture en terme des structures du langage de programmation utilisé excluent toute erreur de syntaxe et, deuxièmement, que les mnémoniques données aux endroits des arguments donnent un rappel permanent du type des arguments attendus. Ce deuxième point, bien que n'apportant qu'une aide de niveau relativement bas, prévient les erreurs du type *oubli d'un argument* et *inversion des arguments*, des erreurs très communes, même pour le programmeur confirmé. Les endroits ou les mnémoniques n'ont pas encore été remplacés sont mémorisés par l'éditeur et celui-ci peut, en réponse à une demande correspondante de l'utilisateur, le placer à l'endroit suivant non encore instancié. Ceci est un

rudiment d'aide de l'éditeur pour rappeler au programmeur tous les endroits où le programme n'est pas encore terminé, une sorte de liste d'endroits où le programme nécessite encore des développements¹³. Notons également, qu'au fur et à mesure des nouvelles définitions entrées par le programmeur, l'éditeur construit de nouveaux canevas qui seront montrés au moment voulu. Ainsi, après la frappe de la fonction **EXEMPLE** ci-dessous, la frappe d'un appel de cette fonction livre automatiquement le canevas :

(EXEMPLE <liste>)

qui contient deux informations :

- premièrement, la fonction **EXEMPLE** possède *un* paramètre, et
- deuxièmement, ce paramètre doit recevoir un argument qui est du type liste.

Cette deuxième information a été dérivée du fait que l'argument **VAR** se trouve comme argument de la fonction **CDR** qui — elle — attend toujours une liste en argument. Bien entendu, si à un instant donné l'utilisateur appelle une fonction **FOO** non encore définie, le canevas montré sera :

(FOO <argument(s)>)

Mais, si l'argument est instanciée comme :

(FOO (EXEMPLE <liste>))

l'éditeur en dérive que la fonction **FOO** doit avoir un seul argument et que ce seul et unique argument peut être une liste, puisque le résultat d'un appel de la fonction **EXEMPLE**, étant lui-même le résultat d'un appel de la fonction **CONS**, doit être une liste. Ce qui lui fait générer le nouveau canevas :

(FOO <liste?>)

où le point d'interrogation exprime principalement le fait qu'avec les informations actuellement disponibles on sait que l'argument *peut* être une liste.

Notons que, si la définition ultérieure de la fonction **FOO** confirme qu'elle ne possède qu'un seul paramètre et que la valeur de ce paramètre doit être une liste, ce canevas devient définitif. Si, par contre, pendant la définition de cette fonction des

13. Ceci correspond alors à ce que les anglophones appellent une *outstanding task list*, celle-ci étant néanmoins générée par l'éditeur.

divergences apparaissent, comme par exemple si elle est définie avec deux paramètres, l'éditeur ouvre d'abord une fenêtre de dialogue demandant à l'utilisateur de confirmer cette divergence, et si l'utilisateur répond par l'affirmative, l'éditeur met à jour le canevas et marque les endroits où **FOO** est appelée pour permettre à l'utilisateur de se positionner ensuite sur ces endroits afin de corriger la ou les erreur(s).

Ainsi, au fur et à mesure de l'écriture d'un programme, l'éditeur construit une description des différentes fonctions construites, de leurs arguments, leurs résultats et leurs interactions. Il utilise constamment ces descriptions pour vérifier la correction des programmes entrés par l'utilisateur, pour lui montrer les endroits non encore complétés, pour mettre à jour les descriptions périmées et pour montrer des endroits qui se trouvent en contradiction avec une nouvelle description.

Bien entendu, cette description peut être utilisée de manière explicite par l'utilisateur. Elle constitue sa fiche de rappel, le micro-manuel spécialisé de son application, qu'il peut interroger interactivement dès qu'un problème se présente. Bref, c'est le noyau d'une documentation du programme, construit automatiquement par l'éditeur. La manière de visualiser cette documentation peut varier : soit l'utilisateur veut juste voir un graphe du flux de contrôle de son programme, il verra alors un arbre comme à la Figure 2 ci-dessous, soit elle sera donnée sous forme textuelle, soit elle sera utilisée, de manière implicite, par le module de documentation.

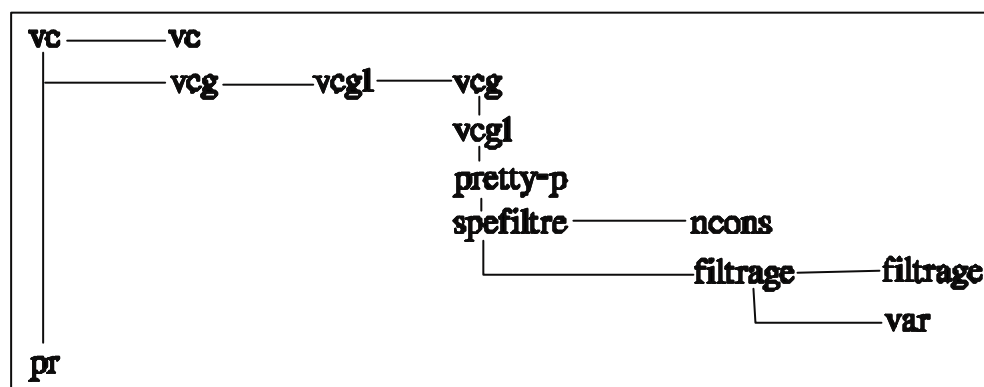


Figure 2 : Exemple d'arbre de flux de contrôle

L'arbre ci-dessus est non seulement une représentation graphique du flux de contrôle, donc de la structure du programme, mais il peut être également utilisé comme moyen de lire le programme : il suffit de pointer vers une des fonctions pour sélectionner, dans un menu, l'affichage de la définition de la fonction, l'affichage de sa description compactée, l'affichage des variables locales et globales qu'elle consulte et/ou modifie, pour demander ensuite un arbre du flux de données des valeurs de ces variables.

Insistons sur cet aspect documentaire : l'éditeur construit une description du programme en cours de développement. Cette description fait partie intégrante de la documentation déjà existante dans le système. L'utilisateur peut à tout moment inter-

roger la machine sur les sujets traités dans le manuel du langage (qui se trouve en-ligne) *et* sur les sujets dont la documentation a été construite automatiquement. Ainsi, par exemple, si à un instant donné l'utilisateur ne se rappelle plus la définition d'une fonction, la simple frappe de la touche "?" lui ouvre une fenêtre de documentation où il peut voir sa définition et quelques exemples saillants de son utilisation — si la fonction est une fonction standard prédéfinie — ou en voir une brève description indiquant ses paramètres et leur type, les endroits où cette fonction est appelée et les fonctions qui sont appelées par elle — si c'est une fonction utilisateur. Si l'utilisateur a accompagné sa fonction d'un ensemble d'exemples d'utilisation, ces exemples seront également montrés.¹⁴

Mais avant d'entrer plus en détail dans l'aspect documentaire, revenons vers le graphique de l'arbre du flux de contrôle. Cet arbre peut être également utilisé pour une trace de l'exécution du programme. Dans ce dernier cas, parallèlement à l'exécution du programme, seront alors montrées dans ce graphique les fonctions actives à chaque instant. Le film de l'exécution en cours donne une très bonne visualisation du comportement dynamique du programme. Naturellement, rien n'interdit d'afficher en même temps, dans des fenêtres séparées, les valeurs que prennent diverses variables choisies ou le contenu de la pile ou d'une autre structure complexe choisie. Cette utilisation multiple des outils d'un environnement de programmation est une des caractéristiques principales de tout environnement de programmation : toute information, toute représentation développée dans *un* outil doit pouvoir être exploitée par et dans d'autres outils — voire même par les programmes utilisateurs.

Mais continuons avec la description de notre éditeur *intégré* dans un langage et revenons vers notre programme exemple :

(DE EXEMPLE (VAR) (CONS VAR (CDR VAR)))

Supposons que le programmeur ait changé cette définition en celle ci-dessous :

**(DE EXEMPLE (VAR)
(IF VAR (CONS VAR (EXEMPLE (CDR VAR))) NIL))**

Ce changement a pu être obtenu en enlevant le sous-arbre correspondant au corps de la fonction, en y insérant un nouveau sous-arbre correspondant à l'appel de la fonction de sélection **IF**, en instanciant ensuite chacun de ses arguments (le deuxième pour l'ancien corps) et en modifiant l'ancien corps de manière telle qu'il contienne l'appel récursif de la fonction **EXEMPLE**.

14. Nous verrons par la suite comment enrichir la définition d'une fonction avec des informations éventuellement utilisables par des outils de l'environnement de programmation ou par l'utilisateur lui-même.

Pendant toutes ces modifications, l'éditeur augmente la documentation et vérifie la consistance du programme. Par exemple, si l'utilisateur avait écrit :

(IF VAR (CONS (1+ VAR) (EXEMPLE (CDR VAR))) NIL))

à l'instant de l'insertion de l'arbre **(1+ VAR)**, l'éditeur aurait donné l'indication d'une contradiction entre le type de la variable **VAR** (une liste) et le type attendu de la fonction d'incrémentation **1+** (un nombre).

Après cette modification, la description de la fonction **EXEMPLE** contient au moins les informations suivantes :

EXEMPLE	fonctions appelées :	EXEMPLE
	nombre de paramètres :	1 (VAL)
	type de résultat :	NIL ou <i>liste</i> ($\mathbf{L}^{VAL-init}$) par accumulation
VAL	type :	<i>liste</i> ($\mathbf{L}^{VAL-init}$)
	modifications :	raccourcissement par CDR
	rôles :	arrêt de la récursion <i>et</i> source

Cette fenêtre dit qu'**EXEMPLE** est une fonction récursive (elle s'appelle elle-même), qu'elle a un paramètre appelé **VAR** et qu'elle retourne soit **NIL**, soit une liste possédant le même nombre d'éléments que l'argument initial (ce qui est exprimé par l'expression ' $\mathbf{L}^{VAL-init}$ '). Le paramètre **VAL** prendra des listes comme valeur, est utilisé pour le test d'arrêt de la récursion et se trouve raccourci par les appels récursifs successifs.

Toutes ces informations correspondent à des questions que le programmeur ou un autre utilisateur peuvent poser sur ce programme. Elles ont été dérivées par un processus de méta-évaluation¹⁵ se déroulant en arrière plan dès que la définition d'une fonction est (temporairement) terminée.

Nous n'avons pas encore montré que l'éditeur garde toutes les *versions* successives d'une fonction (ou d'un programme). La deuxième écriture de notre fonction

15. La méta-évaluation est une manière d'exécuter des programmes sur des données symboliques construites au fur et à mesure de cette exécution. L'information sur l'égalité de la longueur initiale de la liste **VAL** et de la longueur du résultat a été dérivée par un tel processus utilisant une induction sur la liste **VAL**, les opérations **CONS** successives et les opérations **CDR** successives dans l'appel récursif. Nous y reviendrons par la suite.

EXEMPLE en est alors la deuxième *version*. Nous pouvons toujours revenir vers des versions antérieures, aussi bien pour l'édition que pour l'exécution. Le développement d'un programme peut alors devenir l'experimentation simultanée de plusieurs implémentations alternatives, chacune avec ses propres performances, comportements et raisons d'existence. La lecture d'un programme devient l'exploration raisonnée de cet arbre des programmes possibles. Bien entendu, chacune de ces versions partage les portions de code communes avec d'autres versions et ne se distingue que localement. En réalité, une version est *une* lecture (ou *un* parcours) particulier du programme. Chacune des versions possède naturellement sa propre documentation.

Dans la représentation interne, toutes les versions et leur documentation ne sont qu'un seul et unique programme possédant des branches multiples. L'utilisateur, quand il exécute, lit ou édite un programme, ne voit qu'une seule de ces versions. Cette coexistence — dans la représentation interne — de multiples versions permet une généralisation très puissante : les *annotations actives* de programmes, fragments de code, fonctions ou variables. Le paragraphe suivant les examinera plus en détail.

5.3.3. Les annotations actives et la documentation de programmes

Étant donné que toutes les versions coexistent dans une unique représentation interne complexe et que la vision d'une version n'en est qu'un parcours particulier, rien n'interdit d'enrichir cette représentation interne par d'autres informations dont l'accès ne sera également qu'une particularité du parcours.

La possibilité de pouvoir adjoindre à la représentation interne du programme d'autres informations — concernant des aspects particuliers du programme (ou de la version actuelle du programme) — est intensivement utilisée dans notre environnement de programmation pour pouvoir adjoindre aux fonctions des exemples de leur utilisation, des commentaires arbitraires qui seront accessibles pendant l'exécution du programme,¹⁶ des informations d'indexage¹⁷ du programme, des commentaires téléologiques,¹⁸ des assertions sur des états devant toujours être vérifiés à certains endroits du programme, des modules aidant à l'observation du déroulement du programme ainsi que des démons lançant des activités particulières à l'avènement d'événements bien définis.

16. Normalement, les commentaires n'existent que dans le fichier contenant le code source du programme — les lecteurs ou parseurs du langage les ignorent pendant l'analyse.

17. L'*indexage* est l'ensemble des informations dérivées automatiquement par l'éditeur. Ce sont ces informations documentaires que nous avons décrites dans le paragraphe précédent.

18. Un *commentaire téléologique* est un commentaire décrivant le sens d'un programme ou d'un fragment de programme.

Avant d'entrer dans une description de l'utilité de tels mécanismes, ré-examinons la représentation interne actuelle de la fonction **EXEMPLE** :

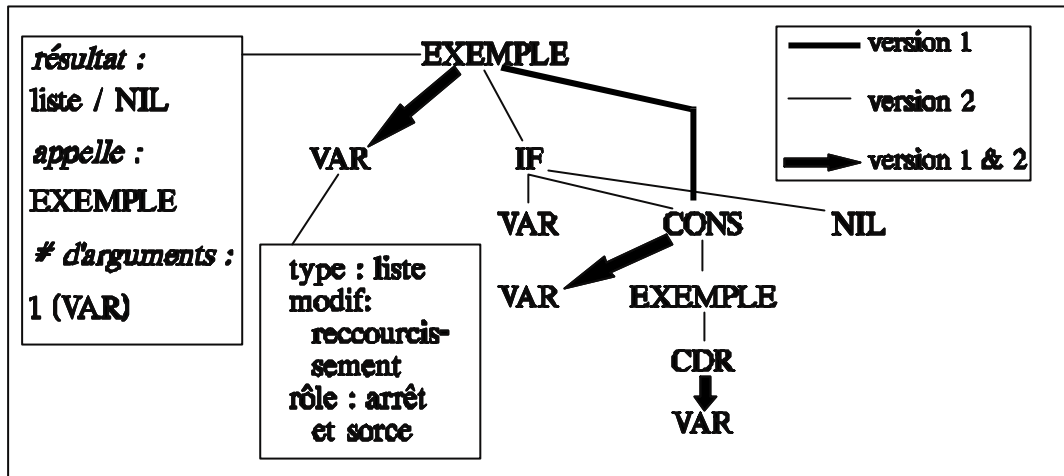


Figure 3: Représentation interne du programme

De même que les informations de parcours déterminant la version sont accrochées aux nœuds de l'arbre et que les descriptions de la fonction et de la variable sont accrochées aux noms des objets décrits, nous pouvons accrocher d'autres informations — et nous pouvons leur associer un comportement d'évaluation.

Les commentaires, par exemple, seront accrochés tels quels aux nœuds qu'ils commentent. Le comportement d'évaluation qui leur est associé sera de les ignorer pendant l'évaluation de la branche correspondante et — si une impression de la branche est demandée — de les entourer par les signes début et fin de commentaires et de les imprimer en utilisant un jeu de caractères spéciaux.

Les commentaires téléologiques seront associés aux éléments décrits — en général des fonctions ou des variables. Pendant l'évaluation et l'impression, ils seront traités comme des commentaires ordinaires, mais nous y associerons une fonction particulière qui permet leur consultation.

Une fois acquise la possibilité d'associer des commentaires aux fonctions, variables et autres objets du programme, et de les garder accessibles, nous avons envie de généraliser cette possibilité en entrant non seulement des commentaires destinés à des lecteurs humains mais également d'y associer des commentaires pour la machine — l'interprète du langage ou tout autre outil ou package disponible. Ces commentaires seront donc des sortes de descriptions indiquant à la machine quel type d'observation, de mesure, de collecte de données, etc, est à effectuer pendant l'exécution ou l'édition du programme annoté.

Historiquement, nous avons commencé à construire un tel outil pour lancer automatiquement un ensemble de tests pour chaque fonction nouvellement éditée. Nous avons alors une base de données contenant pour tout un ensemble de fonctions des exemples d'appels et de résultats correspondants, et chaque fois qu'on sortait de

l'édition d'un fichier contenant du code source LISP, un processus était lancé qui activait chacun des exemples d'appels pour les fonctions contenues dans ce fichier.

Dans l'environnement **bVLISP**, la même possibilité existe : après chaque modification d'une fonction, l'éditeur lance les exemples d'appels qui lui sont associés (il n'y a plus de recherche puisque les exemples sont directement associés à la fonction) et compare les résultats calculés avec ceux donnés dans l'exemple. S'ils sont identiques l'utilisateur ne s'aperçoit de rien; par contre, s'ils sont différents, l'éditeur informe l'utilisateur de la différence et — si possible — lui en indique également les raisons.

Ce mécanisme expose deux principes des environnements de programmation intégrés :

- toute information doit se trouver à l'endroit où cette information est utilisée, et
- une information produite par un outil (ou donnée telle quelle) doit pouvoir être utilisée par plusieurs autres outils de cet environnement.¹⁹

Nous avons déjà vu au paragraphe précédent que les exemples font partie de la documentation d'une fonction. Ici, nous voyons qu'ils peuvent être utilisés dynamiquement, après chaque édition, afin de vérifier que les intentions exprimées à travers eux ne sont pas violées.

L'ajout d'un ensemble d'exemples à notre fonction **EXEMPLE** se fait soit avec l'éditeur, soit en appelant de manière explicite la fonction **AJOUTE-EXEMPLE** comme ci-dessous :

```
(AJOUTE-EXEMPLE EXEMPLE
  ( ('(A B C) => ((A B C)(B C)(C)))
    ( () => ( ) ) ) )
```

où nous avons ajouté deux exemples d'entrée/sortie, qui seront alors ajoutés à l'ensemble des informations déjà associées à cette fonction.

Les exemples sont sûrement fort utiles, tant pour le lecteur du programme, à qui ils indiquent — de manière intuitive — ce que la fonction est supposée faire, que pour le programmeur, qui sera averti dès qu'une de ses modifications ultérieures changera le comportement décrit dans ces exemples.

19. Nous sommes convaincus qu'une excellente heuristique pour l'évaluation d'un outil d'un environnement de programmation est de mesurer l'utilisabilité des informations produites pour d'autres outils. Si les informations sont trop spécifiques et ne peuvent être utilisées ailleurs, l'outil en question n'est pas une bonne solution

Les exemples d'appels sont des annotations concernant l'éditeur. Nous pouvons également avoir des annotations actives concernant l'évaluateur — le mécanisme d'interprétation du programme. Les *assertions* en sont un exemple. Ce sont des expressions associées à un programme ou fragment de programme exprimant des conditions qui doivent toujours être satisfaites quand l'exécution passe à l'endroit auquel l'annotation a été associée. A notre fonction **EXEMPLE** nous pourrions, par exemple, associer une *assertion d'entrée* disant que l'argument doit toujours être une liste et une *assertion de sortie* disant que le résultat doit toujours être de même longueur que l'argument. Voici ces deux annotations :

assertion-d'entrée : (LISTP VAR)
assertion-de sortie : (= (LENGTH VAR) (LENGTH résultatFinal²⁰))

Comme on peut voir, les assertions sont des expressions LISP ordinaires, pouvant utiliser toute fonction LISP standard ou utilisateur définie. Une *assertion d'entrée* est une expression LISP qui est évaluée *avant* que le fragment de programme correspondant soit évalué, une *assertion de sortie* est une expression LISP qui est évaluée *après* que le fragment de programme correspondant ait été évalué. Bien entendu, il peut y avoir autant d'assertions qu'on veut et elles sont toujours associées à des versions précises. Ainsi, deux versions différentes d'un programme peuvent soit partager des assertions, soit contenir des assertions différentes.

Notons la différence entre l'annotation par les exemples et celle par les assertions : les exemples sont vérifiés par l'éditeur *après* chaque modification; les assertions sont vérifiées par l'évaluateur *chaque fois* que le contrôle passe à l'endroit où ces assertions se trouvent. Ainsi, les assertions expriment des faits dont la vérification dépend des arguments bien précis avec lesquels le programme est exécuté, tandis que les exemples expriment des faits qui peuvent être vérifiés indépendamment des exécutions particulières. Autrement dit — en référence à la Figure 1 (page 8) — les annotations de versions et les exemples sont des outils utilisés pendant le codage du programme, les assertions sont des outils pour le test et l'affinement du programme.

Si le test d'un prédicat n'est pas satisfait à un instant donné, la machine affiche — dans la fenêtre des assertions — le texte de l'assertion et les données actuelles pour lesquelles ce test n'est pas satisfait.

Bien entendu, l'utilisateur peut lui-même construire ses propres annotations pour l'évaluateur. Dans ce cas, il doit non seulement indiquer le prédicat à tester, mais également l'action à entreprendre dans le cas de non-vérification de ce prédicat. Ces annotations s'appellent soit des *activités d'entrée*, si le prédicat doit être testé *avant* l'exécution du fragment de programme correspondant, soit des *activités de sortie*, si le prédicat est testé *après* l'exécution du code correspondant. Un exemple simple de

20. **résultatFinal** est une variable désignant le résultat de l'évaluation de l'expression à laquelle l'assertion est associée.

telles activités est l'impression des arguments à l'entrée de la fonction et l'impression du résultat à la sortie de la fonction. Ceci — une sorte d'outil élémentaire de trace de la fonction — peut être obtenu grâce aux deux activités ci-dessous :

activité-d'entrée (T (PRINT VAR “—>” ‘EXEMPLE))
activité-de-sortie : (T (PRINT ‘EXEMPLE “—>” résultatFinal))

où l'expression “**T**” est le prédicat à tester (ici, **T** étant synonyme de true, le prédicat est toujours vrai) et l'appel de **PRINT** est l'action à entreprendre si le test est satisfait.

Notons que le package standard de trace de **bVLISP** génère des activités d'entrée/sortie similaires à celles-ci.

Toutes ces activités supplémentaires, les assertions et les activités d'entrée/sortie, font partie intégrante de la représentation interne du programme et sont donc toujours accessibles — pour la lecture par l'utilisateur ou pour une utilisation par les programmes. Néanmoins, elles peuvent être inhibées sur des portions de programmes considérées comme (temporairement) finies. Ainsi, sans modifier le programme et sans perdre des informations qui peuvent se révéler très utiles ultérieurement, l'utilisateur peut focaliser l'attention du système sur *ces* régions qui lui semblent encore critiques ou qui sont encore en cours de développement.

Examinons un instant la représentation interne actuelle de notre fonction **EXEMPLE** (cf. Figure 4).

Dans cette représentation nous faisons apparaître l'ensemble des annotations évoquées dans ce paragraphe. Bien que toutes les annotations concernent uniquement la deuxième version du programme, on peut voir que la représentation d'un programme peut vite devenir très complexe. C'est pourquoi, pour ne pas surcharger l'utilisateur, l'ensemble de ces descriptions est normalement caché et n'est visualisé que pendant des opérations où elles interviennent directement où sur demande de l'utilisateur.

Sur la Figure 4, nous avons ajouté quelques annotations supplémentaires, principalement pour montrer la généralité de ces mécanismes. Tout d'abord nous avons rajouté à la fonction **EXEMPLE** un commentaire téléologique indiquant la raison (telle que perçue par le programmeur) de cette fonction. Ce commentaire ainsi que le nom et la liste des variables sont montrés si l'on procède à une lecture compacte du programme. Nous avons également ajouté une annotation à la constante **NIL**, indiquant que **NIL** est élément neutre de la fonction **CONS**. Ce commentaire n'existe ici que pour montrer que, effectivement, tout objet LISP peut être annoté — même une constante.

Les variables peuvent être annotées au niveau global — l'annotation sera alors active en permanence, indépendamment de l'endroit particulier où l'exécution se trouve — ou à l'intérieur d'un lieu particulier — par exemple : seulement à l'inté-

rieur d'un ensemble de fonctions ou seulement dans une ou plusieurs branches particulières de la représentation interne. Par défaut, la validité d'une annotation de variable est considérée globale. Nous utilisons cette caractéristique pour tracer les valeurs

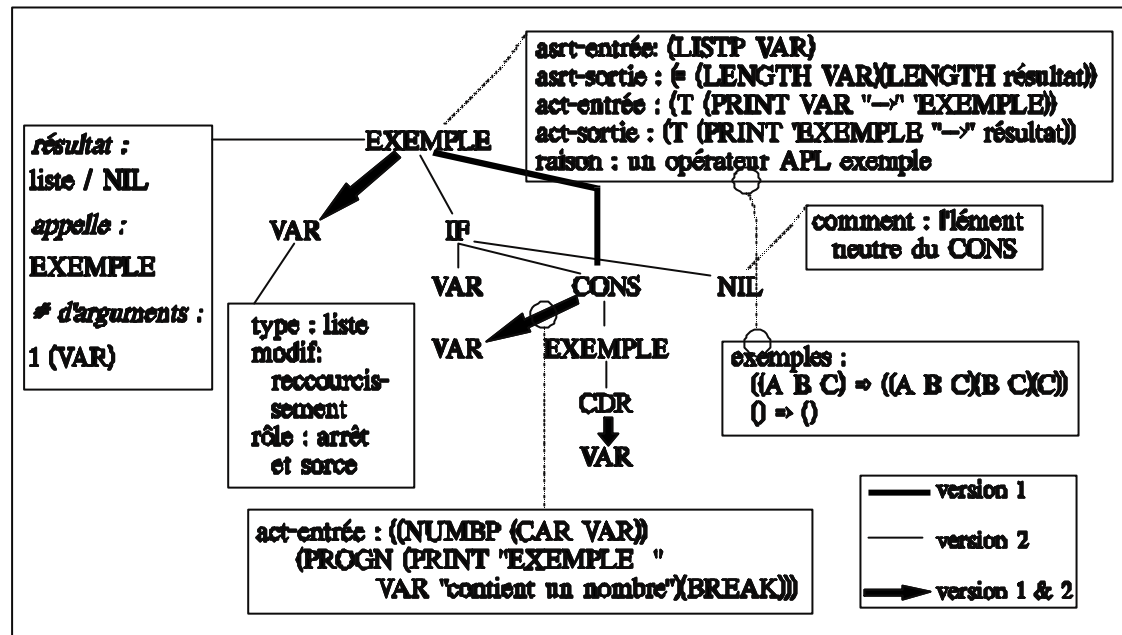


Figure 4 Représentation interne enrichie du programme

de variables. Notons néanmoins que, puisque cette annotation est écrite dans un style identique aux autres annotations, on peut en plus restreindre l'activation en correspondance avec la satisfaction d'un test arbitraire, comme par exemple : seulement si sa valeur est à l'intérieur d'un certain intervalle, ou si la dernière modification de sa valeur a été réalisée dans un ensemble de fonctions bien précise, etc.

Ainsi, pour toute annotation, on peut fixer le grain d'application à une taille quelconque, depuis un grain d'application globale jusqu'à un grain tellement particulier que l'activation n'a lieu que dans des cas exceptionnels. L'aide apportée à la mise au point de programme par un tel mécanisme d'activités supplémentaires, soumis à des conditions de lancement très finement réglables, est inestimable.²¹

La généralisation de ce mécanisme d'annotation est montrée dans l'annotation de la branche gauche (celle qui mène vers l'évaluation de la variable **VAR**) de la fonction

21. Notons également que de tels mécanismes d'annotation généralisent le concept de variables actives [Stefik86] à toute structure de données et de programme et ouvrent une possibilité de discipline de programmation inexplorée : avec des tels mécanismes, on peut construire un programme où l'algorithme de calcul est indépendant des interfaces utilisateurs nécessaires.

CONS. Là, nous avons ajouté une activité d'entrée stipulant que si un des éléments de la variable **VAR** est un nombre, nous voulons un message indiquant ce fait, pour nous trouver ensuite dans une boucle d'inspection — ce qui sera le résultat d'un appel de la fonction **BREAK**. Nous y reviendrons dans le paragraphe suivant.

Pour conclure ce paragraphe, insistons de nouveau sur l'importance documentaire que prennent ces annotations dans notre environnement de programmation. Chacune de ces annotations est à la fois un programme qui sera activé automatiquement — chaque fois que c'est nécessaire — et une aide cruciale pour le lecteur du programme. Elles lui apprennent les raisons, les limitations et les conditions particulières qui ont amené le programmeur à écrire le programme tel qu'il l'a réalisé. L'ensemble de ces annotations, versions et commentaires, permet au lecteur ultérieur de *comprendre* les choix pris et les raisons des diverses particularités d'implémentation, ce qui l'aide pour des modifications et des opérations de maintenance éventuelles. Insistons également sur le fait que toutes ces annotations sont accessibles à l'environnement *et* aux programmes utilisateurs : chacune peut devenir une partie active pendant la lecture, l'édition et/ou l'exécution du programme.

5.3.4. L'évaluation dans un environnement de programmation

La programmation, suivant la Figure 1 (page 8), est une boucle infinie du codage vers le test, la validation, l'affinement et la maintenance, avec des retours vers le codage. Jusqu'à maintenant nous avons principalement vu des outils qui aident pendant les processus de lecture et de codage du programme. Dans ce paragraphe nous examinerons quelques-uns des services qu'un environnement de programmation tel que **bVLISP** offre pendant l'exécution d'un programme.

Bien entendu, comme tout évaluateur, celui de l'environnement de programmation doit pouvoir normalement exécuter des programmes — de manière aussi efficace et correcte que tout autre évaluateur. C'est le minimum absolu ! C'est également un des rôles de notre évaluateur — mais ce n'est qu'un parmi d'autres. Puisque la plupart du temps où l'on utilise un tel environnement on se trouve encore dans une phase de développement ou d'exploration, le programme contient encore des erreurs, des inconsistances non détectées par l'éditeur ou des parties non encore terminées, donc incomplètes et c'est à la rencontre de ces situations qu'un environnement de programmation doit aider l'utilisateur — au lieu de le punir avec un message d'erreur qui avorte le calcul.

L'évaluateur de **bVLISP** distingue entre différents contextes :

- le contexte syntaxique qui détermine les versions du programme à évaluer et
- les contextes dynamiques qui déterminent les différents modes d'évaluation : l'évaluation ordinaire et l'évaluation attentive.

Concernant les contextes syntaxiques, notons que si le programme est composé d'une seule fonction **bVLISP**, le contexte est bien entendu une des multiples versions existantes de cette fonction. Par contre, si le programme est constitué de tout un ensemble de fonctions, le contexte syntaxique détermine le *système* qui doit être évalué. Un *système* est principalement une description de l'ensemble de fonctions et d'instructions d'initialisation composant le programme et, pour chacune de ces composantes, la version composant le système actuellement valide. C'est donc une généralisation de la notion de *version*, puisque nous avons des versions non seulement au niveau des fonctions individuelles mais également au niveau des compositions de ces fonctions. Ceci nous permet de pouvoir explorer de manière interactive les interactions entre diverses versions des mêmes fonctions ou/et des fonctions ne faisant partie que de certains systèmes.

Mais examinons un peu les contextes dynamiques : l'évaluation ordinaire représente juste l'évaluation normale *sans* la moindre activité supplémentaire — inutile de la décrire plus en détail.

L'évaluation *attentive* permet le lancement de tout un ensemble d'activités supplémentaires pendant l'évaluation. Nous en avons déjà mentionné deux : la possibilité de visualiser le déroulement du programme dans la fenêtre montrant l'arbre des appels de fonctions et la possibilité de tracer le contenu de variables dans des fenêtres associées à ces variables.

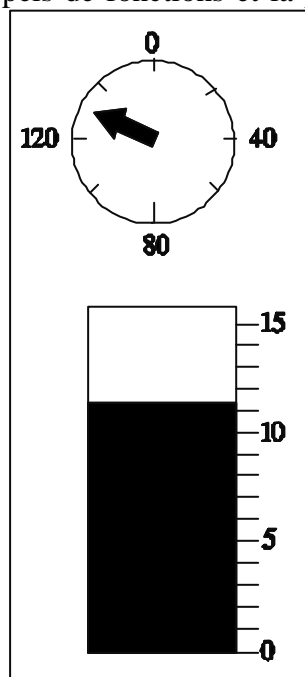


Figure 5 : representations analogues

Cette trace peut être simplement l'affichage du contenu d'une variable, mais elle peut être faite également de manière analogique comme dans la Figure 5 ci-contre, où, dans la première représentation, les variations de l'aiguille indiquent les variations de la valeur et, dans la deuxième représentation, les changements de hauteur de la colonne signalent les changements correspondants de la variable dont cette structure est la trace. Si de telles traces sont actives, chaque changement de valeurs sera visualisé dans la structure associée.

Bien entendu, l'activation de telles traces peut être limitée à certaines régions du programme — si elles sont utilisées pour localiser des erreurs dont on connaît déjà la région d'origine — ou peut être accompagnée d'autres formes de trace plus courantes, telles que des traces d'appels de fonctions par exemple.²²

22. ou encore : ces traces peuvent faire partie de l'interface utilisateur. On utilisera alors les ressources de l'environnement dans le programme lui-même.

Notons que dans notre environnement de programmation, *toutes* les activités supplémentaires peuvent cohabiter entre elles. Des activités supplémentaires peuvent même s'imbriquer, de manière telle qu'une activité supplémentaire peut en activer une autre — comme, par exemple, le lancement de la trace d'une fonction peut activer la trace du package de trace lui-même. Ceci est possible d'une part à cause de la localité des annotations (comme nous l'avons dit plus haut : toute information se trouve à l'endroit auquel elle réfère ou à l'endroit qui utilise cette information) et, d'autre part, par le fait que les annotations peuvent être annotées elles-mêmes, et ceci de manière récursive : il peut bien exister une annotation active d'une annotation active d'une annotation active, ... et ainsi de suite.... Chacune de ces annotations s'active — de manière complètement indépendante des autres — à l'instant où l'évaluateur passe à l'endroit annoté.

Une autre manière d'évaluer des fragments de programmes bien définis est le mode *pas à pas*. Ce mode, utile principalement si l'on cherche une erreur à l'intérieur d'une région bien définie du programme, permet d'entrer après exécution de chaque instruction dans une boucle d'inspection, pendant laquelle on peut examiner l'état actuel du calcul, pour ensuite continuer l'exécution.

Une boucle d'inspection — dans laquelle on peut également entrer par définition programmée des lieux d'activation — permet de consulter ou/et de modifier les valeurs de toutes les variables (ou structures de données) définies, ainsi que de visualiser l'état de la pile système. S'il se trouve dans une telle boucle, l'utilisateur peut remonter dans la pile (pour retrouver des états antérieurs du calcul), modifier et éditer des fonctions ou des structures et continuer le calcul à l'endroit où il est entré dans la boucle (ou à tout endroit antérieur) en prenant en compte les modifications éventuelles.

Dans **bVLISP**, l'utilisateur peut également programmer des événements qui déterminent l'instant à partir duquel une action spécifique supplémentaire doit être déclenchée. Par exemple, l'utilisateur peut indiquer à l'évaluateur d'interrompre le calcul et d'entrer dans une boucle d'inspection quand une certaine variable prend une valeur comprise dans un intervalle donné, ou quand l'évaluateur entre dans une fonction FOO venant de la fonction BAR, qui elle-même a été appelée par la fonction FOOBAR. Les instants d'activation de tels événements programmables sont déterminés par des prédicats d'une complexité arbitraire, de même que les activités associées à ces événements. Ce sont des sortes d'interruptions logicielles programmables qui activent un ensemble de démons — un démon par événement²³ — observant l'évaluation du programme et l'interrompant sous contrôle de leur prédicat de lancement associé. Ces démons sont extrêmement utiles pendant la recherche d'erreurs dues à des incorrections dans le flux de données.

23. Ces démons sont toujours bien localisés. Ainsi, si le démon est associé à une variable, **c** n'est que pendant l'accès à cette variable qu'il est actif, et si le démon est associé à un parcours (comme dans l'exemple donné) il n'y aura un démon que dans la fonction **FOO**.

Bien entendu, pendant toutes ces activités l'évaluateur continue à vérifier les assertions que le programmeur a accrochées au programme. De plus, l'évaluateur continue — comme l'éditeur — de collectionner les informations supplémentaires qui ne peuvent pas être dérivées par l'analyse statique de l'éditeur. Si le programme contient par exemple des appels de fonctions calculés, l'évaluateur va, à leur activation, ajouter ces fonctions à la liste des fonctions appelées (cf. page 14). Ainsi, l'évaluateur complète et élargit la documentation construite par l'éditeur. Sur demande, l'évaluateur peut même construire un arbre des appels de fonctions pour une exécution particulière, ce qui permet — pendant les phases de test — d'identifier les chemins d'exécution possibles qui n'ont pas encore été testés.

Regardons un instant un scénario d'appel de notre fonction **EXEMPLE** :

**(DE EXEMPLE (VAR)
(IF VAR (CONS VAR (EXEMPLE (CDR VAR))) NIL))**

avec toutes ses annotations (cf. page 20) :

? (EXEMPLE '(A B C))

(A B C) —> EXEMPLE

(B C) —> EXEMPLE

(C) —> EXEMPLE

l'assertion d'entrée de EXEMPLE :

(LISTP VAR)

n'est pas satisfaite pour l'appel :

(EXEMPLE NIL)

NIL —> EXEMPLE

EXEMPLE —> NIL

EXEMPLE —> (C)

EXEMPLE —> (B C)

EXEMPLE —> (A B C)

= (A B C)

On voit bien l'impression de l'activité d'entrée que nous avons associée à la fonction, c'est l'impression des traces d'entrée du style "*valeur* —> **EXEMPLE**", l'impression de la non satisfaction de l'assertion d'entrée à l'appel récursif avec **NIL** comme argument, ensuite la dernière trace d'entrée suivie des impressions dues aux assertions de sortie, pour finalement terminer sur le résultat de cet appel. En réalité, ces impressions ne sont pas séquentielles, mais chaque type d'annotation livre — par défaut — ses affichages dans une fenêtre associée à ce type d'annotation. Rien n'ex-

clut que l'utilisateur définisse ses propres fenêtres d'affichage associées à ses annotations personnelles.

Notons que l'activité d'entrée associée à l'évaluation de la variable **VAR** n'a jamais été activée — l'argument de la fonction ne contenait aucun nombre. Rappelons que cet exemple de programme avec ses annotations est très réduit : aucune interaction entre différentes fonctions, des assertions et des activités d'entrée/sortie très simples. Le lecteur intéressé trouvera des exemples plus sophistiqués dans [Wertz84].

L'utilité d'un environnement de programmation apparaît tout particulièrement lors de la gestion des erreurs. En bVLISP, toute une batterie d'options est disponible si l'évaluateur essaye de calculer la valeur d'une expression erronée. Dans ce cas l'utilisateur peut :

- entrer dans une boucle d'inspection afin d'essayer de trouver interactivement la source de l'erreur;
- entrer dans l'édition de la fonction ou de l'expression dont l'évaluation a fait apparaître l'erreur²⁴;
- donner une valeur, si l'erreur est due à l'évaluation d'une variable non définie ou à l'appel d'une fonction non ou pas encore définie;
- donner le nom d'une fonction dont le résultat de l'appel (avec les arguments actuels) remplacera la valeur de l'expression erronée;
- remplacer, à la volée, l'expression erronée par une autre;
- continuer l'évaluation de manière symbolique (nous reviendrons vers cette possibilité);
- demander à l'environnement de tracer la source de l'erreur — ce sera principalement une remontée dans le flux de la donnée ayant produit l'erreur, afin de trouver l'endroit où cette valeur a été produite;
- demander à la machine de corriger automatiquement l'erreur²⁵;
- et, bien entendu, il peut abandonner le calcul.

Examinons juste deux de ces possibilités : la recherche automatique de la source de l'erreur et l'évaluation symbolique. Ces deux capacités font bien ressortir l'interac-

24. En général, le lieu où une erreur se manifeste n'est pas le même que celui qui a produit l'erreur.

25. Ce mécanisme est décrit en détail dans [Wertz85].

tion entre les différentes parties de l'environnement ainsi que l'utilisation multiple des outils.

La recherche automatique des sources d'erreurs utilise l'ensemble des représentations disponibles. Ce problème se présente quand l'évaluateur détecte une erreur dans une fonction due, par exemple, à une différence entre le type ou la valeur d'un argument attendu et celui qu'il a reçu. Une telle erreur peut se produire si une fonction, qui attend une liste, reçoit en argument un nombre. L'erreur n'est évidemment pas dans cette fonction — elle a été spécifiée pour traiter des listes et non pas des nombres — ni nécessairement dans l'appel de la fonction, puisque l'argument donné n'est pas obligatoirement produit dans la fonction qui lance l'appel. L'erreur se situe souvent à l'endroit où une des valeurs de l'expression livrant l'argument erroné a été produite. Si un tel cas se présente, notre environnement remonte dans le flux des données, c'est-à-dire retrace les chemins qui ont produit la donnée erronée pour, s'il trouve un lieu bien déterminé, le livrer à l'utilisateur. Naturellement, pour retracer une valeur — en remontant dans le sens inverse de sa production — la machine utilise les informations disponibles sur le flux de contrôle de cette exécution particulière qui s'est soldée par une erreur.

Dans le cas où la recherche dans le flux des données guidée par le flux de contrôle n'aboutit pas à déterminer le lieu exact ayant produit cette donnée erronée, le système consulte les annotations de l'éditeur pour livrer à l'utilisateur les lieux récemment modifiés qui se trouvent le long du flux de contrôle parcouru pendant l'exécution en tant que candidats possibles à la production de l'erreur. Si le système a tracé les chemins déjà testés et si un des chemins de l'exécution n'a pas encore été testé, c'est alors ce chemin qui sera considéré en priorité. Notons que, non seulement cette démarche utilise des informations venant d'une grande variété d'outils (tels que l'éditeur, la documentation produite pendant l'exécution, les annotations d'indexage, l'analyse du flux de contrôle et des données) mais, également, elle correspond tout à fait à la démarche qu'entreprend un programmeur humain quand il recherche une erreur.

Plus haut nous avons dit qu'un des choix à la disposition de l'utilisateur de **bVLISP** à la rencontre d'une erreur est de continuer l'exécution de manière symbolique. Bien entendu, l'utilisateur peut, de son propre choix, décider de procéder à une exécution symbolique sans être préalablement tombé sur une erreur.

L'exécution symbolique est une sorte de *d'eroulement d'un programme à la main automatisé*, c'est une manière d'exécuter un programme sur des données symboliques. Par exemple, pour évaluer l'expression LISP :

(CONS 1 L)

l'évaluateur doit connaître la valeur de la variable **L**. Si cette variable est liée à la liste (2 3 4) à l'instant de cet appel de la fonction **CONS**, le résultat sera alors la nouvelle liste :

(1 2 3 4)

Par contre, si la variable n'est pas liée à une valeur à l'instant de l'évaluation de cet appel de **CONS**, l'évaluateur doit arrêter son traitement puisqu'il n'a aucun moyen de connaître la valeur de cette variable.

Si, maintenant, nous utilisons un évaluateur symbolique sur cet appel de **CONS** — toujours en supposant que la variable **L** ne soit pas liée à une valeur — l'évaluateur doit *connaître* la fonction **CONS** suffisamment pour pouvoir déterminer une valeur symbolique. *Connaître une fonction* veut dire : l'évaluateur doit, au minimum, connaître le type des arguments, l'existence des valeurs jouant le rôle d'élément neutre pour la fonction (si une telle valeur existe) et il doit connaître l'effet de la fonction, c'est-à-dire : son résultat.

Pour la fonction **CONS**, l'évaluateur sait que c'est une fonction à deux arguments, que le premier argument peut être d'un type quelconque et que le deuxième est, en général, une liste, et que l'effet d'un appel de cette fonction sera de construire une nouvelle liste (le résultat est donc du type liste) dont le premier élément sera la valeur du premier argument (le résultat sera donc une liste avec au moins un élément) et dont le reste de la liste est constitué par la valeur du deuxième argument.

Pour revenir vers notre appel (**CONS 1 L**), l'évaluateur symbolique livrera en résultat l'information stipulant que le résultat est une liste de longueur supérieure ou égale à 1 et dont le premier élément est le nombre **1**. Le résultat d'une évaluation symbolique de la fonction **EXEMPLE** est donné à la page 14. Cette information sera ensuite prise comme valeur symbolique de l'argument de la fonction qui utilise cette expression. Si nous avons, par exemple, l'appel :

(FOO (CONS 1 L))

l'évaluateur peut — même en l'absence de toute connaissance sur la fonction **FOO** — déterminer que **FOO** possède un paramètre et que dans cet appel la valeur de ce paramètre sera le résultat de l'appel du **CONS** qui vient juste d'être examiné.

Dans cet exemple nous voyons bien qu'un évaluateur symbolique utilise des valeurs concrètes, si elles sont fournies, et construit, utilise et affine des valeurs symboliques s'il manque des valeurs concrètes. Plus nous avons de valeurs concrètes, plus une évaluation symbolique ressemble soit à une évaluation normale (si toutes les valeurs sont données), soit à une évaluation partielle (si l'évaluateur est incapable d'opérer sur des valeurs symboliques), soit à une sorte d'évaluation algébrique si, effectivement, l'évaluateur symbolique connaît l'ensemble des fonctions de base, leurs effets et leurs lois de composition (et, éventuellement, de simplification). L'évaluateur symbolique est utilisé également par le système de correction automatique d'erreurs **PHENARÉTÉ** [Wertz85] intégré dans notre environnement de programmation **bVLISP**.

Notre évaluateur symbolique connaît, bien sûr, la fonction **CONS**. Voici, l'ensemble des connaissances de cette fonction :

```

CONS {argument1 argument2}
  [et [pas-de-contrainte-sur-type argument1]
    [usual-type argument2 'LISTP]]
  {do:
    [méta-évalue argument1] -> arg1
    [méta-évalue argument2] -> arg2
  ensuite:
    test: {type(arg2)=LISTP} ->
          {type(arg2)!=LISTP} ->
          {complain-type arg2 argument2 LISTP}
    action:
          {construit-nouvelle-liste-avec arg1 arg2}
  
```

Notons que ces connaissances sont utilisées à tant par l'évaluateur symbolique (ou, si nous préférons le nommer, par le méta-évaluateur [Goossens79]) et par l'éditeur.

L'éditeur y trouve les informations sur le nombre et le type des arguments. C'est en raison de la spécification :

```

[et [pas-de-contrainte-sur-type argument1]
  [usual-type argument2 'LISTP]]
  
```

que l'éditeur peut donner le canevas :

```
(CONS <objet> <liste>)
```

si l'utilisateur le demande, et qu'il peut immédiatement interrompre celui-ci si une contradiction entre cette spécification et l'appel courant survient.

La deuxième partie de cette description de **CONS** donne sa sémantique, dans une représentation qui est utilisable par le méta-évaluateur.

L'instruction *complain* dans la spécification de la fonction **CONS** joue des rôles différents selon que l'évaluateur symbolique a été activé pour une correction d'erreurs ou non. S'il s'agit d'une correction d'erreurs, son rôle est alors de changer le

code source de manière à éliminer les incorrections. S'il s'agit seulement d'une exécution symbolique, son rôle est alors d'annoter l'expression symbolique résultante de façon à indiquer ces incorrections.

Les grandes vertus de cette manière d'évaluer des programmes sont :

- *une seule* exécution capte l'ensemble des exécutions concrètes possibles. Ainsi, par exemple, si, pendant une exécution symbolique il faut évaluer une sélection (un appel de **IF** ou de **COND** en LISP), soit les valeurs symboliques à l'entrée de cette sélection déterminent une seule branche possible — et l'évaluateur est alors sûr d'avoir détecté une incorrection, soit l'évaluateur symbolique continue avec les deux branches possibles (la branche *vraie* et *fausse* dans le cas d'un appel de **IF**) et le résultat est la disjonction des deux expressions symboliques calculées en suivant les deux branches possibles.
- des programmes inachevés, où manquent encore des définitions, des modules ou des initialisations, peuvent être exécutés. Le résultat des activations de modules ou fonctions non encore définis sera alors l'expression de l'appel fonctionnel de ce module avec les arguments symboliques de l'activation. Par exemple, si l'évaluateur trouve à un certain instant, un appel de la fonction non encore définie **FOO**, comme ci-dessous :

(FOO X)

et si la valeur symbolique de **X** est $L_0^{val-init}$, ceci n'interrompt pas le calcul et le résultat de l'évaluation symbolique de cet appel sera :

(FOO $L_0^{val-init}$)

De plus, l'environnement saura que l'argument de la fonction **FOO** doit être du type de $L_0^{val-init}$. Ceci permet de vérifier la consistance de programmes non terminés, puisque la machine pourra vérifier la consistance des types d'arguments des différents appels de **FOO** qu'elle rencontre — même si **FOO** n'est pas encore définie !

- finalement, l'exécution symbolique livre en résultat des expressions symboliques qui expriment, en termes de données symboliques et de leurs types, le calcul que le programme implémente.

5.3.5. Conclusions provisoires

Nous arrêtons ici la description des capacités de notre environnement de programmation **bVLISP**. Bien sûr, tous les environnements ne possèdent pas les mêmes outils (la correction automatique de programmes, par exemple, est unique à

bVLISP), mais ils ont tous des éditeurs connaissant la structure des programmes, ils permettent tous — d’une manière ou d’une autre — de garder différentes versions d’un programme et possèdent tous des outils sophistiqués pour la détection et le traitement des erreurs²⁶.

L’étude des environnements *intégrés* est relativement récente (cf., par exemple, [Degano83]) et existe principalement dans des langages issus des recherches autour de l’Intelligence Artificielle, tels que LISP et SmallTalk. Les environnements plus traditionnels, tels que le système UNIX [Kernighan84], ne constituent qu’une collection *indépendante* d’outils.

Néanmoins, ce bref examen de quelques outils puissants d’un environnement de programmation avancé a montré les forces et les faiblesses de tels systèmes. L’utilisation d’une base de données commune à *tous* les outils offre, à des coûts relativement faibles, une possibilité d’enrichissement des environnements tout à fait exceptionnelle. La maintenance de versions permet, sans crainte de perdre le code utile, un style de programmation très exploratoire : le programmeur *ose* tester différents algorithmes, les comparer et choisir le meilleur, il a toujours la possibilité de revenir vers des états antérieurs de son programme et il peut même intégrer dans des états antérieurs des parties de versions venant d’un futur relatif.

Le traitement d’erreur est la partie la plus sophistiquée dans la plupart des environnements de programmation. Rappelons-nous que c’est la recherche des erreurs, souvent très pénible et prenant beaucoup de temps, qui a le plus inspiré les constructeurs de tels environnements. Grâce à ces possibilités, le programmeur n’est plus puni par l’occurrence d’une erreur, mais les outils mis à sa disposition lui permettent d’explorer interactivement le programme pour mieux le comprendre et l’améliorer encore plus.

Que l’ensemble de tels outils soit extrêmement utile durant les phases de maintenance ultérieures, pour des lectures *actives*, donc avec le soutien de la machine, nous semble évident. Les versions et leurs annotations nous livrent une description savante de l’historique du développement du programme, explicitant les choix de conception et d’implémentation; les diverses représentations coexistantes du programme et de son déroulement permettent de l’explorer aisément et/ou d’évaluer — visuellement — ses performances.

La seule faiblesse qui subsiste — mais qui est de taille — est que ce type d’environnement laisse encore trop de travail à l’utilisateur : bien qu’il le soutienne dans l’ensemble du processus de la construction d’un programme, de la conception à la maintenance à l’affinement local, la connection entre ces diverses facettes de la programmation n’apparaît qu’en tant qu’annotation, éventuellement active, certes, mais laissant à l’utilisateur la tâche d’aller d’une conception, d’une représentation, d’un

26. Dans la littérature anglophone l’ensemble de ces outils s’appelle les outils de *error-recovery*.

niveau d'abstraction vers un(e) autre. L'environnement ne fait qu'enregistrer intelligemment les divers pas. Il soutient et assiste le programmeur, mais le laisse seul s'il s'agit d'aller de la modification d'un niveau d'abstraction vers la modification correspondante d'un autre niveau d'abstraction du programme.

Dans le reste de ce chapitre nous décrirons, à travers un système de prototypage, perspective d'environnement capable d'apporter une aide dans le processus de transformation du prototype en une implémentation réelle.

5.4. Une Perspective : Prototypage et Programmation Transformationnelle

Les environnements que nous avons considérés jusqu'à maintenant centrent leur aide sur de l'activité de codage — sur de la programmation. Ils aident à documenter le programme, à garder ses différentes versions, à tester et à vérifier sa consistance, ainsi qu'à observer son exécution et à le lire de manière active. Néanmoins, nous savons que pendant la construction d'un programme (ou d'un système logiciel) la phase de codage n'est qu'une phase parmi d'autres. Il y a aussi la phase d'analyse, la phase de décomposition du projet en sous-projets plus aisément traitables, les phases de validation de la conception par rapport aux demandes et — finalement — la longue phase de maintenance²⁷ qui suit la construction initiale du système.

Dans ce chapitre nous allons examiner d'autres métaphores — et pratiques — de conception de programmes. Nous allons centrer notre exposé sur la possibilité de construire des prototypes. Ces prototypes guideront ensuite toutes les opérations d'implémentation, de validation et de maintenance.

Avant cela, résumons la métaphore de construction de programmes décrite dans le chapitre précédent et graphiquement représentée dans la Figure 1 (page 8). Dans ce contexte, la spécification est *informelle*, la plupart du temps c'est un *cahier des charges* rédigé en langage naturel. Ce n'est qu'exceptionnellement que l'équipe chargée de la réalisation du programme construit un prototype, qui est alors construit *manuellement* et qui ne sert qu'à valider son comportement par rapport aux intentions exprimées dans le cahier des charges. C'est donc principalement un outil de clarification du cahier des charge. Après cette validation et cette clarification, le prototype n'est plus utilisé et l'équipe (ou le programmeur) procède — encore de manière manuelle — au codage aboutissant à un code source qui sera ensuite testé. La mise au point et toute la maintenance se feront ensuite sur ce code source. Notons, qu'en général,

27. On estime qu'en général la maintenance prend entre 60% et 70% du temps des programmeurs travaillant sur un grand programme. Le temps de codage pour la création initiale d'un tel programme est estimé au maximum être égal à 10% du temps total.

toutes les décisions de conception sont perdues ou n'existent qu'indirectement, comme en bVLISP où c'est la lecture des versions et de leurs annotations qui permet éventuellement de les retrouver. Notons également que tout travail ultérieur sur le programme se fera de manière *locale*, c'est-à-dire qu'il faut retrouver, dans le code source, les endroits implémentant le comportement qu'on veut corriger ou modifier et ce sera une modification de ce code qui réalisera le changement envisagé.

La Figure 6 ci-dessous donne une représentation d'une autre méthodologie de conception et de réalisation de programmes. Ici, à partir des demandes — exprimées de manière informelle — et d'une analyse de ces demandes, l'équipe chargée de la réalisation du programme développe une spécification formelle sous forme de *prototype*. Dans cette méthodologie, le prototype — un programme écrit dans un langage de très haut niveau — *est* donc la spécification. Ensuite, comme tout à l'heure, le prototype sera validé par rapport aux intentions. Si des divergences apparaissent, le prototype (et donc la spécification) sera modifié et mis à jour. Ceci nous garantit qu'à la fin de la boucle *prototype* \Rightarrow *validation* \Rightarrow *modification du prototype* \Rightarrow *nouvelle validation* \Rightarrow ... nous avons une spécification correcte et complète du programme.

Jusqu'ici, il n'y a pas trop de différences avec la méthodologie précédente, excepté que la construction d'un prototype est du moins standard. Les différences fondamentales viennent par la suite, puisque cette méthode suppose l'existence d'un ensemble de règles de transformation²⁸ permettant de transformer le prototype en code source du programme final. Ainsi, la spécification (ou le prototype) *devient* l'implémentation. Dans les rares environnements de programmation transformationnelle (c'est ainsi que cette méthodologie est nommée) qui existent actuellement [Balzer87, Barstow79], le choix d'une règle se fait interactivement. Par contre, l'application de la règle choisie se fait automatiquement. L'implémentation se fait alors de manière semi-automatique avec — comme pour les systèmes de transformation de programmes [Arsac77, Burstall77, Cheatham79] — la garantie que l'application d'une règle ne peut pas transformer le sens (la sémantique) du fragment de prototype ou de

28. Ces règles de transformation sont similaires aux règles des systèmes de transformation de programmes. Ces systèmes [Partsch83] décrivent comment transformer un programme en un autre programme équivalent — donc de même sémantique, mais itératif au lieu de récursif (et vice versa) [Arsac77, Burstall77] ou plus efficace [Kant83]. Ce type de règles peut être utilisé dans la programmation transformationnelle également, mais ultérieurement, pour optimiser mécaniquement le code des programmes obtenus par l'application répétée d'un premier ensemble de règles. Ce premier ensemble exprime des connaissances pour passer d'une description de très haut niveau (le prototype) vers une description de plus bas niveau (le code d'un programme exécutable). Des règles de ce style ne sont pas non plus nouvelles : déjà en 1975, Samet [Samet75] utilisait ce type de règles pour vérifier que le code généré par un compilateur était correct; Barstow utilisait ce type de règles dans son système de synthèse semi-automatique de programmes LISP [Barstow79]. Ce qui est nouveau, c'est l'intégration de tous ces systèmes de règles dans un seul et unique système.

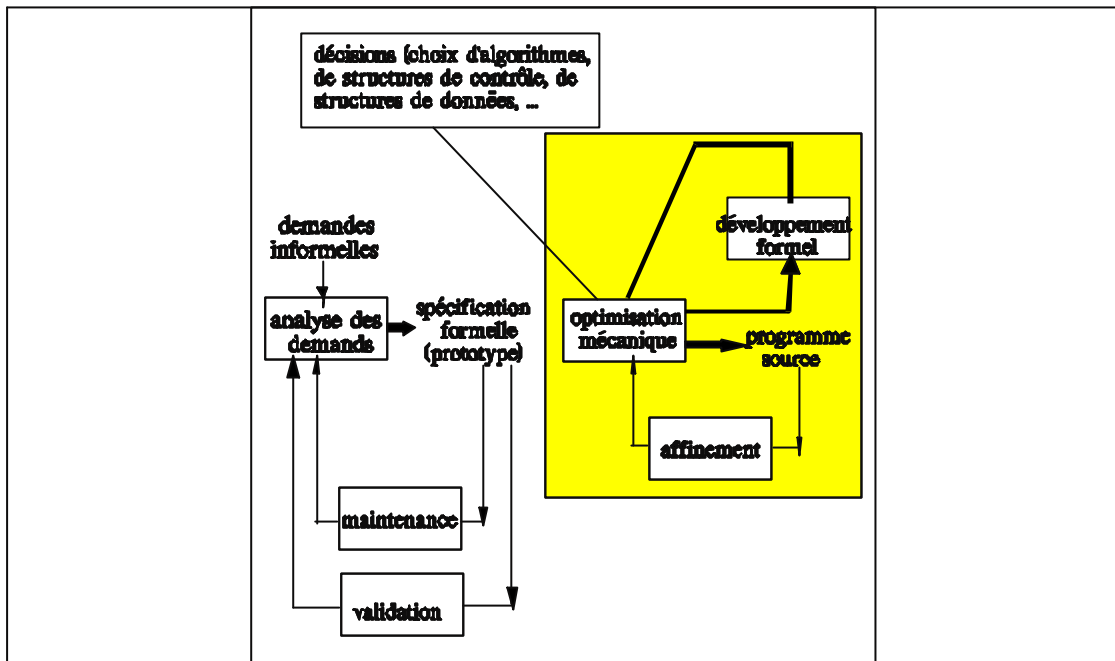


Figure 6 : Développement transformationnel de programmes

programme transformé, puisque l'ensemble des règles est vérifié par rapport à la préservation de la correction sémantique.

Étant donné que le prototype a été validé et que les transformations ne modifient en rien la sémantique du programme, la phase de *test du code* peut alors être éliminée.

La suite des règles appliquées constitue la documentation sur le choix des algorithmes, des structures de données et de contrôle, et, d'une manière générale, elle sert à documenter chacune des décisions de conception du programme. Cette documentation peut être construite automatiquement, simplement en sauvegardant ces règles. Ceci permet également de régénérer un programme à partir du prototype en ré-applicant, dans le même ordre, les mêmes règles. Cette possibilité est à la base d'une autre caractéristique fondamentale de cette méthodologie de construction de programmes : le fait que toute opération de maintenance peut se faire au niveau des spécifications formelles, donc du prototype. L'adaptation du programme correspondant s'obtient en *répétant* l'application des règles déjà déterminées aux endroits non modifiés et par le choix interactif de règles pour les parties modifiées. C'est le même processus que celui qui permet la construction originale du programme; ce processus étant accéléré néanmoins par le fait qu'il ne faut choisir de nouvelles règles que pour la transformation des parties nouvelles.

Après avoir traduit complètement le prototype en programme source dans un langage adéquat, on peut appliquer un deuxième ensemble de règles de transformation pour affiner localement le programme généré.

Bien sûr, aucun environnement de programmation transformationnelle complet n'existe aujourd'hui. Actuellement, nous avons des systèmes de prototypage [Balzer81, Krief89], des systèmes limités de transformation de prototypes vers des programmes exécutables [Balzer85], tout un ensemble de systèmes de transformation de code source [Arsac77, Burstall68, Standish76] et quelques systèmes attaquant le difficile problème du choix des algorithmes et/ou de structures de données [Kant83, Katz81].

Dans le paragraphe suivant, nous allons brièvement présenter un système de prototypage expérimental qui expose bien le très haut niveau d'abstraction de tels systèmes ainsi que leurs capacités principales : animation, vérification de consistance et évaluation symbolique dirigée par des points de vue.

5.4.1. M.PV.C. : un système de prototypage

Le système de prototypage M.PV.C. que nous allons présenter dans ce paragraphe, a été développé par Philippe Krief pour avoir un outil, puissant et aisé à utiliser, qui fournit une aide durant la phase de conception et de prototypage d'un programme. Ce système est un des résultats du projet ECCAO²⁹, un projet conjoint de l'Université Paris 8, de l'Université Paris 13 et de la Société Thomson. Le but initial était de livrer aux concepteurs de Thomson un outil de spécification interactive permettant de visualiser le flux des données, de vérifier la consistance et de générer automatiquement des spécifications dans d'autres formalismes que celui utilisé pour la spécification originale³⁰.

Un tel outil s'avèrait nécessaire, puisque jusqu'à cet instant, les concepteurs de cet environnement industriel procédaient d'une manière entièrement manuelle. En effet, tout ce qu'ils avaient à leur disponibilité c'était un programme permettant de *dessiner* des graphes SADT. A l'aide de ce programme ils dessinaient alors les graphiques des flux de données et des flux de contrôle de leurs projets qu'ils commentaient en langage naturel. La validation et la vérification de ces spécifications se faisaient ensuite durant des rencontres entre l'équipe des spécifieurs et les demandeurs, où l'on essayait de lire ensemble ces spécifications pour déterminer leur correction. Bien entendu, avec l'augmentation de la taille des projets, cette manière de validation se révélait de plus en plus inadéquate : la complexité d'interaction entre diverses parties de la spécification ne pouvait plus être traitée manuellement, le risque d'oubli de parties considérées (temporairement) comme secondaires devenait trop grand et, au dessus de tout cela, les commentaires en langage naturel révélaient de plus en plus l'ambiguïté inhérente à sa libre utilisation et ne donnaient plus aux utilisateurs ultérieurs de ces spécifications les renseignements qu'ils étaient censés donner.

29. ECCAO est un acronyme pour **É**laboration des **C**ahiers des **C**harges **A**ssistée par **O**rdinateur

30. Ce projet a été réalisé en Smalltalk dont il constitue une des plus grandes applications industrielles.

Le projet ECCAO consistait à construire un prototype pour l'élaboration interactive de cahiers des charges. Il se composait d'une représentation interne de connaissances du domaine d'application sous forme d'un réseau sémantique, d'un analyseur d'énoncés en langage naturel traduisant les commentaires en un enrichissement de ce réseau sémantique et d'un moteur d'activation permettant d'utiliser ces connaissances pendant la validation et la vérification automatique des spécifications.

Le système M.PV.C. est ce moteur qui permet de parcourir une spécification et d'appliquer tout un ensemble de fonctionnalités pendant ce parcours. La conception du système M.PV.C. a été — comme son nom l'indique — influencée par une méthodologie Smalltalk appelée M.V.C., pour **M**odèle, **V**ue, **C**ontrôleur [Mével87]. M.V.C. s'utilise pour l'implémentation de programmes interactifs : le modèle est la structure sur laquelle on travaille, la vue est l'interface de sortie qui donne la représentation externe de cette structure, et le contrôleur est l'interface d'entrée permettant d'interagir avec l'objet ou sa représentation. Les dépendances entre ces trois entités peuvent être représentées graphiquement comme sur la Figure 7 ci-contre.

L'utilité de ce schéma consiste dans le fait qu'il permet de bien dissocier les différentes phases conceptuelles de l'implémentation de systèmes interactifs : le programmeur peut construire le modèle indépendamment des interfaces utilisateurs et il peut construire l'interface de sortie indépendamment de l'interface d'entrée. M.V.C. et les outils permettant son implémentation font partie de l'environnement standard de Smalltalk.

Le système M.PV.C. est une généralisation de cette méthodologie : dans un premier temps le programmeur élabore un **M**odèle décrivant les caractéristiques et les fonctionnalités des divers objets constituant son prototype. Dans un second temps, il décrit le ou les mécanismes généraux de parcours de son prototype — c'est la partie **C**ontrôleur. Dans un troisième temps, il élabore autant d'interprétations qu'il lui est nécessaire et les combine en fonction de ses besoins, ce sont alors les différents **P**oints de **V**ues qu'il peut avoir de ses objets [Krief89]. Ces trois entités — modèle, points de vues, contrôleur — s'organisent comme sur la Figure 8 ci-contre. Comme on peut voir, l'organisation d'un système M.PV.C. est considérablement plus simple que celle d'un système M.V.C. Si dans le premier modèle, M.V.C., la vue et le contrôleur doivent connaître les deux autres parties — le modèle ne connaît que sa vue — dans le système M.PV.C. le modèle est indépendant à la fois du contrôleur et du point de vue. Le point de vue connaît son modèle (ou les objets de son modèle); le contrôleur connaît le modèle ainsi que le point de vue qu'il doit activer. Cette organisation met en valeur le fait que plusieurs algorithmes peuvent parcourir un *même* modèle de la *même* façon. C'est-à-dire que le Contrôleur, donc le séquençement des étapes de balayage du modèle, est identique pour chacun des algorithmes. Seule l'activité locale à chacune de ces étapes, c'est-à-dire le Point de Vue, est spécifique à l'algorithme.

Le système M.PV.C. a été utilisé pour le développement et l'analyse de spécifications SADT [Ross77], PAWS [IRA84], State-Charts [Harel84] ainsi que pour le développement d'un système de réseaux neuronaux. Durant ces développements,

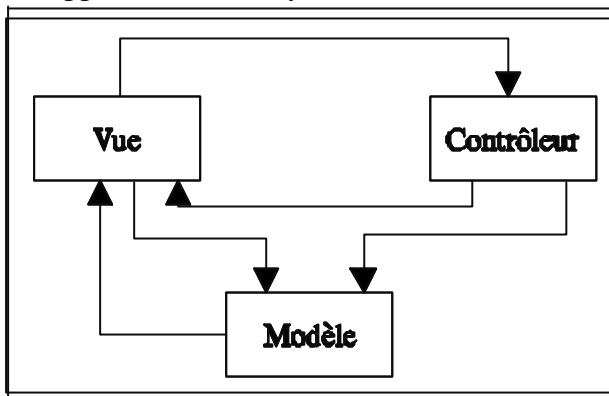


Figure 7 : schema M.V.C.

différents algorithmes ont dû être mis en œuvre : l'évaluation symbolique [Goossens80, Howden77, Wertz85] des flux de données, la trace de l'évaluation d'un graphe, l'animation graphique d'un parcours et l'évaluation partielle [Futamura71, Lombardi64] de spécifications à des fins de conversion d'un modèle de spécification vers un autre.

Comment se met en œuvre un tel système ? Tout d'abord, le programmeur décrit, dans le *modèle*, la structure complexe sur laquelle le système travaille. Pour une application SADT³¹ par exemple, le modèle sera l'implémentation des objets composant une spécification SADT :

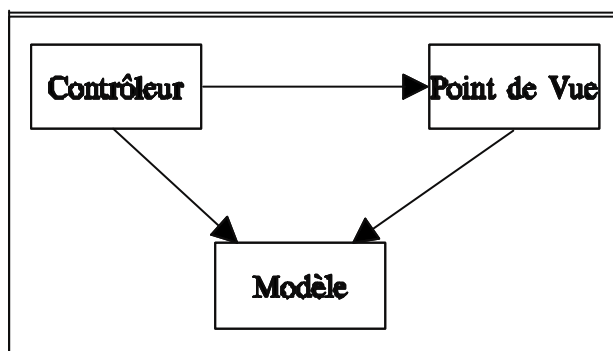


Figure 8 : schema M.PV.C.

principalement la description des boîtes et de leurs entrées, sorties et mécanismes, les connexions existantes entre différentes boîtes, ainsi que la description de leurs rôles, limitations et fonctionnements. L'implémentation du modèle se fera indépendamment des couches contrôleur et

point de vue. Le modèle représente par la suite le noyau du système M.PV.C. : toute activité de celui-ci se fera en consultant ce modèle.

Une fois l'implémentation du modèle terminée (rappelons qu'elle se fait sans référence au point de vue et au contrôleur) le concepteur du système procède à l'écriture de la partie contrôleur. Le contrôleur décrit, comme nous l'avons dit plus haut, le séquençement du parcours — c'est la partie *flux de contrôle* de l'algorithme parcourant le modèle. Puisque plusieurs algorithmes peuvent parcourir une spécification de manière identique, ce séquençement est indépendant des activités à entreprendre à

31. Une spécification SADT est un ensemble de graphes, où chaque nœud est une boîte qui décrit soit une activité, soit une donnée. Les arcs des graphes décrivent soit le flux des données (pour le premier type de boîte), soit des flux de contrôle (pour le deuxième type de boîtes). Il existe également des arcs exprimant des contraintes sur les mécanismes à utiliser. Tout objet SADT est — en général — accompagné d'une description de son rôle, de ses limitations et de ses contraintes.

chaque étape du parcours, qui seront décrites par le point de vue. Afin d'organiser le parcours, le contrôleur utilise des *règles de séquençement* attachées à chaque étape.

C'est l'écriture d'un ensemble de telles règles qui détermine le comportement du contrôleur pour un modèle donné. Chaque règle est composée de trois parties : un test d'activation, une activation du point de vue et une ou plusieurs continuations. Ces règles sont attachées à la classe des objets constituant le modèle³², c'est l'activation d'une règle par le contrôleur qui sera appliquée à une instance de la classe. Pour toute règle, la partie *test d'activation* détermine l'applicabilité de la règle même. Pendant un parcours d'une spécification SADT par exemple, le *test d'activation* d'une règle associée à une boîte qui pourrait être la vérification que toutes les flèches entrantes ont été activées préalablement. Si le test est négatif, le contrôleur essaye d'appliquer la règle suivante, sinon le point de vue associé au contrôleur est activé. Cette activation s'effectue en demandant au point de vue d'exécuter une activité précise correspondant à la règle courante (et décrite dans la partie activation de la règle) dans le contexte de l'objet actuellement balayé [Krief89, page 5]. Cette activité s'appelle la *sémantique évaluatoire* attachée à l'étape. Dans le cas d'une animation d'un diagramme SADT, la *sémantique évaluatoire* attachée à une flèche consistera, par exemple, à faire parcourir une icône, représentant le type de donnée transmis par cette flèche, le long de la flèche. Dans le cas d'une évaluation symbolique, la *sémantique évaluatoire* associée à une flèche consistera à vérifier la cohérence des données reçues par rapport aux données attendues.

La dernière partie d'une règle détermine les étapes suivantes du contrôleur, c'est-à-dire l'ensemble des règles suivantes qu'il faudra appliquer, ainsi que les objets du modèle pour lesquels elles devront s'appliquer. Cette partie est décrite par un ensemble de continuations que Krief nomme des *liens de séquençement*.

Le programmeur peut décrire le contrôleur soit en le décrivant directement dans un langage de description de règles, soit en le décrivant à l'aide d'un *éditeur de séquençement*. Cet éditeur permet de dessiner interactivement le graphe des séquençements et d'y attacher les spécifications fonctionnelles décrivant les tests d'activation et les activations du point de vue. Un tel graphe (représenté à la Figure 9) peut ensuite être compilé dans le langage de description de règles. Bien entendu, le système permet également de *décompiler* un ensemble de règles en leur graphe des séquençements.

Finalement, le programmeur doit définir la *sémantique évaluatoire* attachée à chacune des étapes décrites par le contrôleur activant ce point de vue. Concrètement, la

32. Puisque le système est écrit dans un langage orienté objet, Smalltalk en l'occurrence, il repose sur l'organisation hiérarchique des classes. Une classe décrit la structure générale d'un objet et les questions auxquelles l'objet peut répondre. Une instance est un représentant particulier d'une classe, qui a la même organisation générale que toute autre instance de la classe, mais qui se distingue par ses valeurs propres. Clairement, le *modèle* définit les classes et leurs instances, le *contrôleur* organise le parcours de ces instances et le *point de vue* décrit ce qu'il faut faire à chaque étape.

dent. Ceci pourra représenter le premier pas vers un *système interactif de transformation de prototypes*.

En conclusion, nous pensons que le système M.PV.C fournit non seulement un outil de prototypage très puissant et très agréable à utiliser, mais il donne également la méthodologie nécessaire à la transformation du système en un système de programmation transformationnelle. La décomposition de systèmes en un modèle, la représentation du monde sur lequel le programme ultérieur travaille, les points de vue, qui donnent diverses sémantiques pour chacun des objets du modèle, et les contrôleurs, qui livrent les parties algorithmiques du programme, sont valides pour tout programme — en particulier pour les programmes nécessaires à la réalisation d'un tel système de programmation transformationnelle.

5.5. Conclusion

Dans ce chapitre nous avons donné, à travers l'étude de deux systèmes construits à l'Université Paris 8, un bref survol des capacités des environnements de programmation intelligents. Nous avons choisi ces deux systèmes, **bVLISP** et **M.PV.C.**, puisque d'une part ils sont effectivement implémentés, tournent et sont utilisés, et que, d'autre part, ils exposent les caractéristiques les plus avancées des systèmes de ce type. Rappelons quelques unes de ces caractéristiques :

- Tout d'abord, un environnement de programmation est un système *intégrant* un grand nombre d'outils. bVLISP, par exemple, possède des outils de lecture, d'édition, d'exécution, de documentation, de vérification, d'animation, d'annotation et de maintenance de programmes. bVLISP n'est pas une *collection* de tels outils, mais — comme tout bon environnement de programmation — intègre les divers outils et modules de manière telle qu'ils puissent se cross-fertiliser mutuellement : les connaissances déduites par *un* outil et les représentations développées pour *un* outil sont, et peuvent être, utilisées par *tous* les autres .
- Un environnement de programmation doit intégrer divers domaines de connaissances : des connaissances syntaxiques et sémantiques du langage de programmation utilisé, des connaissances algorithmiques et des connaissances sur le domaine d'application du programme. bVLISP, par exemple, utilise des connaissances du langage de programmation durant les phases d'édition et d'analyse des programmes et utilise des connaissances algorithmiques lors de la correction automatique des programmes. M.PV.C. utilise des connaissances du domaine d'application pour la construction du modèle et pour la vérification de la consistance.
- Un environnement de programmation doit permettre d'avoir plusieurs points de vue des objets sur lesquels il travaille. Ainsi, en bVLISP, un

programme peut être visualisé sous forme d'un arbre syntaxique, sous forme d'un arbre du flux des données et/ou de contrôle ou sous des formes textuelles à des degrés de détail variés. De même, bVLISP laisse coexister les différentes versions du développement du programme — chacune d'elles pouvant être visualisée selon les diverses manières décrites ci-dessus. M.PV.C. permet d'associer à un modèle plusieurs contrôleurs et/ou plusieurs sémantiques évaluatoires. Ou encore, M.PV.C. permet d'entrer et de visualiser les règles de séquençement aussi bien par l'édition interactive d'un graphe de séquençement que par l'édition textuelle de ces règles. Ce sont aussi des points de vue multiples sur un même objet.

- Un environnement de programmation doit combiner des aides *locales* et des aides *globales*. Les aides locales sont des outils qui n'ont qu'une portée localisée : ce sont, par exemple, les traces d'une fonction, l'insertion d'une assertion à l'intérieur d'une seule fonction, l'édition d'une fonction, etc. En bVLISP, ces aides locales cohabitent avec des aides plus globales, telles qu'un module permettant la trace d'exécution d'un ensemble de fonctions interdépendantes (en montrant sur l'arbre de contrôle, en parallèle à l'exécution du programme, l'avancement du calcul) ou la possibilité d'assertions non-locales comme, par exemple, des assertions concernant la conjonction d'un ensemble d'évènements, ou encore l'édition du programme à l'aide de l'arbre de dépendance des différentes fonctions qui est constamment mis à jour par le système (plus précisément : à chaque modification d'une fonction, le système calcule la portée de cette modification en construisant un graphe des dépendances. L'utilisateur peut ensuite *voyager* le long de ce graphe en apportant, si nécessaire, des modifications). Dans le système M.PV.C. le niveau local des aides est offert par l'environnement Smalltalk (le langage dans lequel ce système est écrit) et le niveau global est représenté justement par les outils du système lui-même.

Pour résumer ces quatre points : un environnement de programmation doit savoir donner des aides au concepteur/programmeur à des niveaux divers concernant l'implémentation, les descriptions (descriptions de l'implémentation et descriptions du domaine d'application) et l'exécution.

Nous avons volontairement fait abstraction des interfaces standard des environnements de programmation. Nous n'avons donc pas abordé des questions de multi-fenêtrage, des outils de pointage tels que des souris, des tablettes ou des écrans *tactiles*. L'ensemble de ces outils semble suffisamment connu depuis l'avènement populaire de micro-ordinateurs tel que le Macintosh d'Apple ou le système Windows de Microsoft. Nous pensons néanmoins avoir couvert l'ensemble des problèmes et des caractéristiques des environnements de programmation.

Comme Barstow [Barstow88], nous conclurons ce chapitre par l'énoncé de quelques sujets de recherche, concernant les environnements de programmation, qui n'ont pas assez été abordés jusqu'à maintenant, mais pour lesquels les connaissances accumulées semblent aujourd'hui suffisamment évoluées pour pouvoir *oser* les attaquer.

Tout d'abord il y a le problème de l'automatisation des connaissances du domaine d'application. Ces connaissances sont nécessaires au programmeur pendant la quasi-totalité de la vie d'un programme : elles sont nécessaires pour comprendre la spécification, elles interviennent pendant la conception et l'implémentation du programme, elles dirigent — au moins en partie — les phases de test et de validation du logiciel, et — finalement — elles sont à la base des évolutions ultérieures et de la maintenance des logiciels. Nous pensons que la création de quelques systèmes spécifiques pour deux ou trois domaines d'application et qui fonctionnent effectivement devrait apporter suffisamment de techniques pour pouvoir aborder le problème fondamental, qui est ici : comment représenter des connaissances du domaine d'application d'une manière indépendante de ce domaine et comment le faire de manière telle qu'un concepteur/programmeur puisse effectivement les utiliser ?

Le deuxième sujet de recherche concerne les techniques du contrôle de la recherche. Ce problème est particulièrement crucial pour le paradigme de la programmation transformationnelle. De tels systèmes utilisent un si grand nombre de règles et l'espace de recherche est tellement énorme qu'il faut absolument trouver des techniques qui réduisent fortement le nombre de règles pouvant s'appliquer à un instant donné et dans une situation donnée. Ces techniques devraient, probablement, utiliser des connaissances sur la représentation des données, les différentes manières de contrôler le flux de contrôle, donc des connaissances algorithmiques, ainsi que des connaissances sur l'efficacité des divers opérateurs disponibles. Comme un programmeur humain, un système de programmation transformationnelle doit n'activer des connaissances qu'à l'instant où elles sont demandées. L'organisation des différentes bases de connaissances, telle qu'elle est pratiquée dans les recherches sur les systèmes-experts distribués, pourra livrer les premières indications sur des solutions possibles.

Le dernier sujet est : comment représenter l'historique de la conception d'un système très grand et très complexe (disons : un système d'une taille équivalent au système Multics par exemple) ? Comment le représenter de manière à ce qu'il soit utilisable par le programmeur, par le système et par un lecteur ultérieur ? Comment le représenter pour qu'il mette en relation *tous* les lieux affectés par une décision ? Puisqu'une décision à un niveau peut engendrer tout un ensemble de contraintes de décisions à des niveaux très séparés, comment montrer leurs interactions ? Actuellement nous n'avons aucune idée sur la façon de répondre, non seulement à cause des masses d'information, de descriptions et de connaissances qui sont à traiter avec des programmes très complexes et de taille très élevée, mais également parce que nous ne savons pas encore comment intégrer — et lier entre elles — des connaissances de domaines différents.

Peut-être que des représentations analogiques, sous forme de scènes visuelles connues (comme des paysages, par exemple), comme celles décrites dans des romans de science fiction [Vinge81, Gibson87], qui forment des sortes de réalités virtuelles à l'intérieur desquelles le programmeur ou le concepteur se déplace, indiquent une voie de solution. Pour l'instant, de toute façon, aucun système intégrant de telles descriptions du développement n'existe pour des programmes de taille très élevée et il reste encore (heureusement) beaucoup de travail de recherche, de conception et de développement à faire.

Remerciements

Les Figures 1 et 6 ont été reproduites d'après des notes prises lors d'une conférence de Bob Balzer. Le paragraphe sur le système M.PV.C. repose largement sur le papier de Philippe Krief [Krief89], qui m'a beaucoup appris sur Smalltalk et qui a grandement influencé ma compréhension et mes idées sur les environnements de programmation. Par ailleurs, je tiens à remercier les équipes de Thomson et de l'Université Paris 13 qui ont participé au projet ECAO. La société Thomson a financé, en partie, les recherches aboutissant au système M.PV.C. La relecture attentive de Florence Caroff-Krief et de Marie-Noëlle Rozin a contribué à améliorer la présentation de ce chapitre.

5.6. Bibliographie

- [Aït85] H. Aït Kaci, *LOGIN: A logic programming language with built-in inheritance*, MCC Technical Report n° AI-068-85, 1985
- [Arsac77] J. Arsac, *La construction de programmes structurés*, Dunod Informatique, Paris, 1977
- [Balzer81] R. Balzer, *Transformationnel programming: An example*, IEEE Transactions on Software Engineering, Vol. 7, n° 1, pp. 3-14
- [Balzer85] R. Balzer, *A 15 year perspective on automatic programming*, IEEE transactions on Software Engineering, Vol. 11, n° 11, pp. 1257-1268, 1985
- [Balzer87] R. Balzer, *Living in the next-generation operating system*, IEEE Software, Vol. 4, n° 6, pp. 77-85, 1987
- [Barron63] D. W. Barron et D. F. Hartley, *Techniques for program error diagnosis on EDSAC 2*, Computer Journal, n° 6, Avril 1963
- [Barstow79] D. Barstow, *An experiment in knowledge-based automatic programming*, Artificial Intelligence, Vol. 12, n° 2, pp. 73-119, 1979

- [Barstow88] D. Barstow, *Artificial Intelligence and Software Engineering*, dans : Exploring Artificial Intelligence, ed.: H. Shrobe, Morgan Kaufmann Publishers, San Mateo, CA, 1988, pp.641-670
- [Birtwistle73] G. M. Birtwistle, O. J. Dahl, B. Myhrhaug, K. Nygaard, *Simula Begin*, Petrocelli/Charter, New-York, 1973
- [Borgida85] A. Borgida, S. Greenspan, J. Mylopoulos, *Knowledge representation as the basis for requirements specification*, IEEE Computer, Vol. 18, n^o 4, pp. 82-91, 1985
- [Burstall68] R. Burstall, *Proving Properties of Programs by Structural Induction*, Computing Journal, Vol.12, n^o 1, pp. 41-48, 1968
- [Burstall77] R. Burstall et J. Darlington, *A transformation system for developing recursive programs*, Journal of the ACM, Vol 24, n^o 1, pp. 44-67, 1977
- [Brooks75] F. P. Brooks, *The mythical Man-Month*, Addison-Wesley, Reading, MA, 1975
- [Cheatham79] T. Cheatham, T. Townley et G. Holloway, *A system for program refinement*, 4th Int. Conf. on Software Engineering, Munich, pp. 53-62, 1979
- [COB73] *American National Standard COBOL*, ANS 73, New York, 1973
- [Colmerauer82] A. Colmerauer, *Prolog II: Manuel de référence et modèle théorique*, Groupe Intelligence Artificielle, Université d'Aix-Marseille II, 19823
- [Degano83] P. Degano & E. Sandewall, *Integrated Interactive Computing Systems*, North-Holland, Amsterdam, 1983
- [Fickas85] S. Fickas, *Automating the transformational development of software*, IEEE Transactions on Software Engineering, Vol. 11, n^o 11, pp. 1268-1277, 1985
- [Futamara71] Y. Futamara, *Partial evaluation of computer programs: an approach to a compiler-compiler*, Journal of the Institute of Electronics and Communications Engineers, 1971
- [FOR54] *Preliminary report: Specifications for the IBM mathematical FORMula TRANslating System: FORTRAN*, IBM Corp., Progress Res. Group. Appl. Sci. Div., 1954
- [Gibson87] W. Gibson, *Count Zero*, Grafton Books, London, 1987

- [Gill51] S. Gill, *The Diagnosis of Mistakes in Programs on the EDSAC*, pp. 538-554 dans *Proc. of the Royal Society of London*, Vol 206A, 1951
- [Goldberg83] A. Goldberg et D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983
- [Goldberg83a] A. Goldberg et D. Robson, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, MA, 1983
- [Goldstine63] H. H. Goldstine et J. von Neumann, *Planning and coding problems for an electronic computing instrument*, pp.80-235 dans *John von Neumann, Collected Works*, Vol. V, ed. A. H. Traub, Pergammon Press, London, 1963
- [Goossens79] D. Goossens, *Meta-Interpretation of Recursive List-Processing Programs*, dans *Proc. 6th Int. Joint Conference on Artificial Intelligence*, Vol. 2, Tokyo, Japan, 1979
- [Greussay77] P. Greussay, *Contribution à la définition interprétative et à l'implémentation de λ -langages*, Thèse d'État, Université Paris 7, Novembre 1977
- [Greenspan84] S. J. Greenspan, *Requirements Modeling: A Knowledge representation Approach to Software Requirements Definition*, Ph.D. Thesis, University of Toronto, Canada, Mars 1984
- [Harel84] D. Harel, *Statecharts: A Visual Approach to Complex Systems*, Dept. of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel, Février 1984
- [Hewitt72] C. Hewitt, *Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot*, AI-TR 258, M.I.T. Artificial Intelligence Laboratory, Cambridge, PhD thesis, 1972
- [Howden77] W. E. Howden, *Symbolic Testing and the DISECT Symbolic Evaluation Program*, IEEE Transactions on Software Engineering, Vol. 3, n° 4, pp. 266-278, 1977
- [IRA84] Information Research Associates, *Performances, Analyst's Workbench System (PAWS), Users Manual*, Austin, Texas, 1984
- [Kant83] E. Kant, *On the efficient synthesis of efficient programs*, Artificial Intelligence Journal, Vol. 20, n° 3, pp. 253-306, 1983
- [Katz81] S. Katz et R. Zimmerman, *An advisory system for developing data representations*, dans *Proc. 7th International Joint Conference on Artificial Intelligence*, Vancouver, Canada, pp. 1030-1036, 1981

- [Kernighan84] B. W. Kernighan et R. Pike, *The UNIX programming Environment*, Prentice-Hall, Englewood Cliffs, NJ, 1984
- [King76] J. C. King, *Symbolic Execution and Program Testing*, Communications of the ACM, Vol. 19, no 7, 1976
- [Krief89a] Ph. Krief, *M.PV.C. - Une méthode générale pour l'implémentation de systèmes actifs de spécification, d'interprétation et de simulation, dans un environnement Objet*, Rapport Technique, Dépt. Informatique, Université Paris 8, Juillet 1989
- [Krief89b] Ph. Krief, thèse, Université Paris 8, à paraître 1990
- [Lombardi64] L. A. Lombardi et B. Raphael, *LISP a language for an incremental Computer*, dans *The Programming Language LISP: its Operation and Applications*, ed. E. C. Berkeley & D. G. Bobrow, M.I.T.-Press, Cambridge, 1964
- [McCarthy65] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart et M. I. Levin, *LISP 1.5 Programmer's Manual*, M.I.T. Press, 1965
- [McDermott74] D. V. McDermott et G.J. Sussman, *The CONNIVER Reference Manual*, AI-Memo 259a, M.I.T. Artificial Intelligence Laboratory, Cambridge, 1974
- [Mével87] A. Mével, T. Gueguen, *Smalltalk-80*, Eyrolles, Paris, 1987
- [Miller56] G. A. Miller, *The Magical Number Seven, Plus or Minus Two : Some Limits on Our Capacity for Processing Information*, Psychological review, (63) pp. 81-97, 1956
- [Naur63] P. Naur et al., *Revised Report on the Algorithmic Language ALOGOL 60*, Communications of the ACM, May 1960, pp. 299-314, 1960
- [Newell64] A. Newell, F. M. Tonge, E. A. Feigenbaum, B. F. Green Jr. et G. H. Mealy, *Information Processing Language-V Manual*, 2nd edition, Prentice-Hall, Englewood Cliff, NJ, 1964
- [Partsch83] H. Partsch et R. Steinbrüggen, *Program Transformation Systems*, Computing Surveys, Vol. 15, n° 3, pp. 199-236, 1983
- [Reif77] J. H. Reif et H. R. Lewis, *Symbolic Evaluation and the Global Value Graph*, 4th ACM Symposium on Principles of Programming Languages, Los Angeles, CA, pp. 104-118, 1977

- [Ross77] D.T. Ross, *Structured Analysis: A Language of Communication Ideas*, IEEE Transactions on Software Engineering, Vol. SE-3, n° 1, 1977
- [Samet75] H. Samet, *Automatically Proving the Correctness of Translations Involving Optimized Code*, AIM-256, Artificial Intelligence Laboratory, Stanford University, Stanford, CA, 1975
- [Sestoft88] P. Sestoft, H. Sondergaard, *A Bibliography on Partial Evaluation*, SIGPLAN Notices, Vol.23, n° 2, pp.19-27, 1988
- [Sites71] R. L. Sites, *Algol W reference manual*, Technical Report STAN-CS-71-230, Computer Science Department, Stanford University, Palo Alto, Ca, Août 1971
- [Standish76] Standish et al., *The Irvine Program Transformation Catalogue*, Dept. of Computer Science, University of California at Irvine, Irvine, CA, 1976
- [Stefik86] M. Stefik, D.G. Bobrow et K. Kahn, *Integrating access-oriented programming into a multi-paradigm environment*, IEEE Software, 1986
- [Takeuchi83] I. Takeuchi, H. Okuno et N. Ohsato, *TAO: A harmonic mean of LISP, Prolog and Smalltalk*, SIGPLAN Notices, Vol. 18, n° 79, 1983
- [Teitelman65] W. Teitelman, *Edit and break functions for LISP*, M.I.T. Artificial Intelligence Laboratory, AI-Memo n° 84, Juillet 1965
- [Teitelman66] W. Teitelman, *PILOT: A step towards man-computer symbiosis*, M.I.T. Artificial Intelligence Laboratory, Ph.D. Thesis, Septembre 1966
- [Teitelman78] W. Teitelman, *INTERLISP Reference Manual*, Xerox Palo Alto Research Center, Palo Alto, October 1978
- [Vinge81] V. Vinge, *True Names*, Baen Books, New York, 1981
- [Wertz83] H. Wertz, *An Integrated, Interactive and Incremental Programming Environment for the Development of Complex Systems*, dans [Degano83] pp. 235-250, 1983
- [Wertz84] H. Wertz, *Étude, Réalisation et Évaluation d'un Environnement de Programmation utilisant des Représentation Multiples pour le Développement continu de logiciels très évolué*, Thèse d'État, Université Paris 8, Rapport LITP RT 84-16, Avril 1984
- [Wertz85] H. Wertz, *Intelligence Artificielle : Application à l'analyse de programmes*, Masson, Paris, 1987

- [Wertz89] H. Wertz, *(Common)LISP, une introduction à la programmation*, Masson, Paris, 1989
- [Wilkes51] M. V. Wilkes, D. J. Wheeler et S. Gills, *The Preparation of Programs for an Electronic Digital Computer*, Addison-Wesley, Reading, MA, 1951
- [Wilkes57] M. V. Wilkes, D. J. Wheeler et S. Gills, *The Preparation of Programs for an Electronic Digital Computer (Second Edition)*, Addison-Wesley, Reading, MA, 1957
- [Wijngaarden69] A. Van Wijngaarden (ed.), B. J. Mailloux, J. E. L. Peck et C. H. A. Koster, *Report on the Algorithmic Language ALOGOL 68*, Numer. Math. 14, pp. 79-218, 1969
- [Winograd75] T. Winograd, *Breaking the Complexity Barrier (Again)*, dans *Interactive Programming Environments*, ed. Barstow, Shrobe & Sandewall, McGraw-Hill, New York, pp. 3-18, 1975