Girard P. & Pierra G. Structures de contrôle générales en Programmation par Démonstration. Journées Francophones sur l'Ingénierie de l'Interaction Homme-Machine (IHM'95), Ed. P. Palanque, Pub. Cépaduès, Toulouse, 11-13 Octobre 1995, pp. 61-68.

Structures de contrôle générales en Programmation par Démonstration

Patrick GIRARD, Guy PIERRA
LISI / ENSMA
Site du Futuroscope, B.P. 109
86960 FUTUROSCOPE CEDEX, France
{girard,pierra}@ensma.univ-poitiers.fr

RÉSUMÉ :
La Programmation par Démonstration développe le principe selon lequel le simple fait, pour un utilisateur d'application informatique, de savoir comment effectuer une tâche à l'aide de son application, devrait être suffisant pour créer le programme capable d'accomplir cette tâche. Cependant, cette approche souffre le plus souvent d'un manque de puissance, les systèmes qui la mettent en oeuvre étant incapables de fournir toutes les structures de contrôle de la programmation usuelle. Nous décrivons dans cet article le système LIKE, capable de générer totalement interactivement les différentes structures de contrôle de la programmation structurée (Sous-Programmes avec paramètres et récursivité, Alternatives et Répétitions).

MOTS CLÉS : Programmation par Démonstration, Programmation Visuelle, Structures de contrôle.

INTRODUCTION

La Programmation sur Exemple (Example-Based Programming [5]) ou Programmation par Démonstration (PpD) [1] développe le principe selon lequel le simple fait, pour un utilisateur d'application informatique, de savoir comment effectuer une tâche à l'aide de son application, devrait être suffisant pour créer le programme capable d'accomplir cette tâche. L'absence de structures de contrôle (Sous-Programmes, Alternatives et Répétitions) a été relevée par Myers [5] comme l'un des principaux problèmes à résoudre pour rendre cette approche opérationnelle. Nombreux sont les travaux qui ont proposé des solutions partielles, souvent sous la forme de structures particulières ou simplifiées. Aucun système, tout au moins à notre connaissance, ne propose les structures de contrôle générales que constituent les répétitions basées sur les relations de récurrence ou les définitions récursives d'objets.

Cet article présente le système LIKE [2], système de PpD en Conception Assistée par Ordinateur (CAO), capable de générer, de façon totalement interactive, les différentes structures de contrôle de la programmation structurée (Sous-Programmes avec paramètres et récursivité, Alternatives et Répétitions).

La première section présente la PpD, et examine plus particulièrement les différentes structures de contrôle proposées par les différents systèmes. La deuxième section développe la notion de **contexte dynamique**, qui joue un rôle central dans la solution que nous proposons. La troisième section décrit les différentes structures de contrôle proposées, ainsi que leur gestion interne. Enfin, la quatrième section décrit les conventions de dialogue que nous avons définies pour permettre l'introduction interactive de ces différentes structures.

PROGRAMMATION PAR DÉMONSTRATION ET STRUCTURES DE CONTRÔLE

L'idée générale de la PpD consiste à généraliser la séquence d'interactions de l'utilisateur en un programme susceptible d'être réexécuté. Initialement tournée vers l'apprentissage facilité de la programmation, et donc vers les programmeurs (au moins en devenir), la PpD s'est rapidement orientée vers l'utilisateur final (*end-users* [6]), afin de lui permettre d'automatiser les tâches répétitives et fastidieuses. Les plus grandes réussites de la PpD peuvent ainsi s'observer dans le domaine de l'édition textuelle ou géométrique [1][10], mais également dans celui des macro-commandes ([3],[7],[13]). Au-delà de ces besoins, quelques tentatives d'exploration ont été tentées, dans des domaines aussi divers que la programmation iconique [1] ou géométrique (GeoNode [12]) et les générateurs d'interfaces homme-machine [1]).

L'importance des travaux réalisés a conduit nombre de Comités de Programmes de Conférences, principalement dans le domaine des Interfaces Homme-Machine, à leur consacrer des sessions spécialisées. Plusieurs états de l'art ont été publiés [4][1]. L'ouvrage d'Alan Cypher [1] décrit ainsi l'état actuel de la PpD[1], dressant un large panorama de cette dernière. Proposant une fiche récapitulative pour chacun des systèmes présentés, il approfondit essentiellement deux points : l'**inférence**, et les **structures de contrôle** (*program constructs*).

L'inférence consiste à rechercher, au travers des actions de l'utilisateur, les intentions de ce dernier, afin de permettre la généralisation des tâches

[1] Beaucoup de références citées dans cet article concernent des travaux repris dans l'ouvrage publié sous la direction de A. Cypher [1]. Pour des raisons de place, les références originales ne sont pas mentionnées ici.

effectuées. Cette inférence s'applique à la nature des objets (valeurs, arguments, contraintes), ou aux structures de programme implicitement utilisées (alternatives ou répétitions). Ainsi, dans Peridot [1], le fait de placer deux objets *au voisinage de la même horizontale* amène le système à 'inférer' qu'ils doivent être horizontalement alignés. En pratique, les règles utilisées pour de telles inférences doivent être déterministes. Elles doivent également être connues de l'utilisateur du système de façon à lui permettre de maîtriser le programme construit. Nous qualifierons donc ces règles de **conventions de dialogue**, et nous verrons, à la quatrième section, comment des conventions de dialogue supportées par des mécanismes purement algorithmiques peuvent être définies.

Les **structures de contrôle** supportées par les systèmes de PpD sont généralement limitées. En 1990 [5], l'absence (ou tout au moins le caractère très partiel) des structures de contrôle dans les systèmes de PpD constituait même l'un des obstacles majeurs à l'utilisation de ce mode de programmation. De nombreux travaux ont donc visé à introduire de telles structures, et nombre de systèmes en proposent aujourd'hui des versions plus ou moins complètes.

Ainsi, parmi les systèmes qui définissent l'ensemble de leurs programmes par l'exemple, les sous-programmes avec paramètres sont-ils présents dans Chimera [3], Geometer's Sketchpad [1] et AIDE [1], mais avec seulement des paramètres d'entrée (ce sont plus des 'macros' que de réels sous-programmes). La récursivité n'est explicitement fournie que dans Geometer's Sketchpad [1], mais l'absence d'expressions de contrôle dans ce système rend la définition récursive incomplète : l'utilisateur est obligé de préciser à chaque exécution le niveau de profondeur de récursivité qu'il souhaite voir appliquer. Les conditions sont assez souvent présentes (Peridot [1], Turvy [1], Metamouse [1], TELS [1]), mais se limitent toujours à des traitements de cas très particuliers (existence ou non d'un objet, satisfaction d'une contrainte géométrique). Enfin, les structures répétitives sont des itérations de collections (*set iteration* [1], consistant à appliquer une même action à des objets différents pris dans un ensemble, Peridot [1], ou dans un texte, TELS [1]), permettent l'identification automatique de séquences d'interactions répétées (EAGER [1]), Turvy [1], Chimera [3]), ou encore permettent l'instanciation de structures répétitives prédéfinies comme la création d'un nombre prédéfini d'objets (Metamouse [1], Peridot [1]). Peridot [1] est le seul système qui introduit la notion de relations de récurrence, mais ces relations sont déterminées par identification avec des classes prédéfinies. Les systèmes considérés comme les plus complets abandonnent cependant le principe de base de la PpD en demandant à l'utilisateur de textuellement modifier le programme (hors exemple) afin d'introduire les structures de contrôle (SmallStar [1], GeoNode [12], Tinker [1]).

Avant de voir, dans la section suivante, comment la notion de contexte permet de fournir un cadre général pour la réalisation des structures de contrôle en PpD, relevons les points qui font du domaine de la CAO un domaine de choix pour la PpD.

La PpD est particulièrement efficace lorsque deux conditions sont réunies : (1) les objets du domaine sont intrinsèquement graphiques, et (2) l'interface du système permet d'exprimer naturellement les relations entre objets. Le fait que ces deux conditions soient remplies dans le domaine de la CAO géométrique explique que ce domaine soit le premier dans lequel les techniques de la PpD aient atteint la maturité industrielle. Le modèle géométrique d'un objet technique est en effet constitué d'un ensemble d'entités géométriques associées à un nombre important de relations. Le dessinateur industriel, l'ingénieur mécanicien ou l'architecte connaissent ces relations, et veulent que le système interactif permettent de les définir. *Telles fenêtres doivent être alignées sur la même horizontale, tels trous équidistants, etc*. Les systèmes CAO conventionnels ont donc toujours permis l'expression de ces contraintes. Mais, jusqu'à une date récente, elles n'étaient pas ensuite mémorisées. Au cours des dernières années, une nouvelle génération de systèmes dits paramétriques [9][11] est apparue sur le marché. En mémorisant, tout au long de la construction, les relations impliquées dans l'exemple, ils produisent en fait un programme séquentiel, susceptible d'être ensuite ré-exécuté en changeant des valeurs numériques définies lors de la construction de l'exemple. Si l'interface offerte par de tels systèmes apparaît particulièrement appréciée des utilisateurs (ProEngineer® par exemple a pris en quelques années 15% de parts du marché français), ils ne permettent néanmoins que de reproduire des 'programmes' séquentiels.

LE RÔLE CENTRAL DU CONTEXTE

La principale différence entre programmation traditionnelle et PpD réside dans le fait suivant : alors que le programmeur traditionnel *décrit* des actions à effectuer sur des objets théoriques (les variables), le programmeur par démonstration *réalise* des actions sur des objets concrets possédant des valeurs. Alors que le premier effectue trois étapes (édition-compilation-test), le second alterne simplement deux phases, la phase d'**enregistrement**, au cours de laquelle le programme est enregistré, et la phase d'**exécution**. La tâche d'un système de PpD comporte essentiellement deux points :

- enregistrer ou réexécuter les actions sélectionnées par l'utilisateur
- substituer les valeurs d'exemple par des variables (enregistrement) et inversement (réexécution).

Si le premier point ne présente pas de difficulté particulière tant que l'on demeure dans l'enregistrement séquentiel, le second point nécessite

une analyse assez fine des données impliquées dans l'exemple.

Distinction constantes / paramètres

Généraliser un exemple revient en premier lieu à identifier parmi les objets introduits par l'utilisateur quels sont ceux qui demeureront constants pour chacune des exécutions futures, et quels sont ceux qui peuvent/doivent varier. Les premiers sont des *constantes*, alors que les seconds constituent les *paramètres* ou *arguments* du programme final. Les constantes doivent être enregistrées en l'état au sein du programme, alors que la valeur des paramètres doit être substituée par leur nom dans le programme enregistré. Lors de l'exécution du programme, les valeurs des constantes sont utilisées telles quelles, alors que de nouvelles valeurs pourront être affectées aux paramètres, valeurs sur lesquelles porteront les actions enregistrées.

Cette distinction n'est cependant pas suffisante. En effet, la programmation traditionnelle ne distingue pas constantes et paramètres (ou arguments), mais plutôt constantes et variables, ces dernières pouvant elles-mêmes être soit des paramètres, soit des variables internes. Or, une particularité des systèmes graphiques de conception vient s'ajouter : le principal but d'un processus de conception consiste à *créer* de nouveaux objets à partir des premiers.

Variables implicites

Du point de vue de la PpD, ceci revient à construire de nouveaux objets (par les commandes de création disponibles au niveau du système) à partir d'objets créés précédemment. Traduit en terme de programmation, il s'agit de *déclarer* de nouvelles variables (création) puis de leur **affecter** pour valeur dans l'exemple le résultat de l'action de création. Il faut alors, dans le programme, effectuer la même substitution valeur/nom que celle décrite ci-dessus pour les paramètres. Nous avons appelé ces objets des *variables implicites*.

Une difficulté majeure se présente alors : celle de l'identification des variables. Si, pour les paramètres, qui sont dans la pratique en nombre limité, demander à l'utilisateur de préciser leur nom parait raisonnable (il pourra en particulier leur donner un nom significatif), agir de même pour les objets créés durant l'enregistrement d'un programme reviendrait à supprimer une grosse partie des avantages de la PpD. Dans la quasi-totalité des cas (hors structures de contrôle), le dessein de l'utilisateur est pourtant clairement identifiable : il s'agit d'une référence temporelle, c'est à dire de la référence au *énième* objet créé pendant l'exemple en cours. Une simple numérotation systématique des objets créés durant la définition de l'exemple permet ainsi d'identifier chacune des variables, et d'effectuer automatiquement les substitutions requises.

Exemple

Illustrons ce mécanisme par un exemple : supposons que nous voulions créer deux cercles de même rayon (paramètre), situés sur une même horizontale, à une distance donnée (autre paramètre). Avec un système permettant de construire des points par leur coordonnées, et des cercles par centre et rayon, le script représentant la séquence interactive permettant d'obtenir cette figure pourrait être le suivant (les commandes sont représentées en gras, les valeurs numériques ou littérales sont représentées en italique, le symbole <> représente une désignation graphique d'entité à l'écran) :

Créer_Point *0* *0*
Créer_Point *0* *30*
Créer_Cercle <> *10*
Créer_Cercle <> *10*

Figure 1 : Script de l'exemple

Dans un système proposant une interface utilisateur de type menus, le résultat serait le suivant (les points sont représentés par des croix, et les symboles près de celles-ci correspondent à l'écho des pointés de désignation, marqués <> dans le script d'interaction) :

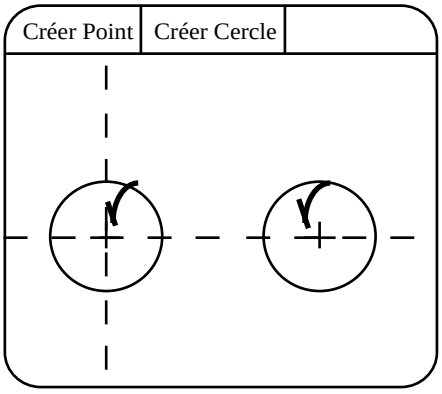

Figure 2 : Visualisation des interactions précédentes.

Généraliser cet exemple revient à identifier les paramètres, puis à déclarer automatiquement les objets créés par les commandes pour permettre d'effectuer la substitution valeur/nom. La figure 3 illustre ce programme, qui ne nécessite qu'un changement mineur dans l'interface offerte à l'utilisateur (la définition des paramètres) :

DÉBUT
Paramètre (*'Rayon'* ***)***
Paramètre (*'Distance'* ***)***
Créer_Point *0* *0* -> P1
Créer_Point *0* Distance -> P2
Créer_Cercle P1 Rayon -> C1
Créer_Cercle P2 Rayon -> C2
FIN

Figure 3 : Programme résultant

En italique gras, est représentée la commande additionnelle nécessaire pour définir les paramètres. Les flèches suivies de noms (P1, C2...) identifient la création de nouveaux objets (pour une meilleure lecture, nous avons choisi de typer les objets, et de les numéroter dans leur type), alors que les noms soulignés correspondent au résultat de la substitution automatique des valeurs par les noms des variables.

Le système LIKE

Le système LIKE, que nous avons conçu pour mettre en œuvre cette démarche, est un moniteur de PpD greffé sur un système de CAO 2D permettant de construire des modèles constitués d'entités simple (points, droites, cercles, ...) éventuellement structurées (notion de pièce). Dans le système CAO support, un soin particulier a été apporté aux diverses opérations de créations par contraintes, et à la définition d'une calculette grapho-numérique capable de calculer, par exemple, *la moitié de la distance séparant le point x du centre du cercle y*.

L'environnement de PpD fourni dans LIKE, quoique logiquement 'greffé' sur le système CAO lui-même, s'intègre totalement dans le dialogue du système complet, et ne rajoute qu'un nombre minimal de commandes. Des règles strictes contrôlées par LIKE filtrent l'accessibilité aux objets visibles sur l'écran. Ces règles peuvent s'énoncer comme suit :

- Les paramètres du programme doivent être définis par le programmeur et sont désignables ;
- Toute autre valeur numérique ou graphique (x,y correspondant à une position graphique) introduite dans l'exemple est considérée comme constante ;
- Tout objet graphique créé au cours de l'enregistrement du programme reçoit un nom implicite (numérotation) qui permettra au système LIKE de l'identifier ; cet objet est ensuite désignable ;
- Toute référence (désignation graphique) à un objet créé précédemment est remplacée dans le programme enregistré par l'identifiant de cet objet ;
- En corollaire, toute désignation d'objet non créé au cours du programme est rejetée ;
- L'association des deux ensembles de variables décrits ci-dessus constitue le **contexte dynamique** du programme, formé du **contexte explicite** (paramètres) et du **contexte implicite** (variables à nom implicite).

Ces quelques règles permettent d'obtenir un système d'utilisation très simple, ne possédant que peu de commandes spécifiques de programmation, et dont le comportement est entièrement prévisible (pas d'inférence). Nous allons voir dans la prochaine section comment peuvent être définies les structures de contrôle dans un tel système.

STRUCTURES DE CONTRÔLE DANS LIKE

Les structures de contrôle (au sens large) présentent une incompatibilité majeure avec la gestion du contexte implicite que nous avons proposée ci-dessus. En effet, un exécution correcte, selon les règles précédentes, suppose une substitution nom/valeur en phase d'exécution équivalente à la substitution valeur/nom de la phase d'enregistrement. Cette équivalence est implicitement basée sur le fait que les objets sont temporellement créés dans le même ordre. Le quatrième objet créé lors de l'enregistrement du programme est supposé être le même objet lors de toute exécution. Ceci est évident pour tout enregistrement purement séquentiel, en l'absence d'échec d'une commande.

Mais dans le cas de l'introduction des structures de contrôle, il n'en est pas de même. Quel sens peut avoir la désignation d'un objet durant la branche ALORS d'une structure alternative si le programme est passé lors de l'exécution par la branche SINON ? On pourrait envisager de le substituer par l'objet de rang équivalent créé dans la branche SINON, mais existe-t-il ? La numérotation automatique des variables est basée sur le principe d'un nombre constant de créations d'objets à chaque exécution. La création d'un nombre différent d'objets entre les deux branches d'une alternative créerait ainsi un décalage dans la numérotation des variables créées après l'alternative. Pour résoudre ce problème, Tinker [1] fait coïncider la fin du programme avec la fin de la structure alternative !

Ce problème est encore plus crucial dès lors qu'une structure répétitive est créée. Sauf cas particulier, une structure répétitive réalise un nombre d'itérations différent à chaque exécution. La numérotation ultérieure des variables est donc naturellement affectée. Quant à la désignation d'objets construits au cours de l'itération, quel sens a-t-elle ? Que signifie la désignation d'un objet construit au cours du troisième tour d'une itération en comptant quatre lors de la phase d'enregistrement, lorsque l'exécution de la même structure compte sept itérations ? S'agit-il d'un objet du troisième tour (constante temporelle stricte) ? ou du sixième tour (avant-dernier tour) ? Pis encore, qu'en est-il dans le cas où la structure ne comprend que deux tours ?

Nous verrons ci-dessous comment le principe général des sous-programmes peut être appliqué aux autres structures pour fournir une solution cohérente à ces problèmes.

Sous-Programmes

La gestion des sous-programmes telle que les langages modernes tels PASCAL ou Ada la réalisent s'adapte parfaitement aux besoins de la PpD. Les paramètres du sous-programme constituent l'interface d'échange avec le programme appelant. Les autres variables du sous-programme représentent des variables locales à ce dernier, et sont inaccessibles de l'extérieur de celui-ci. La réentrance du code

suppose en revanche que ces mêmes variables locales soient recréées à chaque appel.

En fait, cette assimilation est un peu rapide. Assimiler les seuls paramètres, tels que nous les avons définis dans la section précédente, à l'interface d'un sous-programme est très réducteur. En effet, seuls des paramètres d'entrée ont, pour l'instant, été introduits. Or, les langages classiques proposent également des paramètres de sortie, ou d'entrée-sortie, dont la définition permet au programme appelant d'exploiter les résultats du sous-programme. Dans le cas de la PpD, une solution naturelle peut être envisagée : les variables internes du sous-programme, essentiellement graphiques, représentent en fait **globalement** le paramètre de sortie implicite du sous-programme. Elles constituent un ensemble auquel l'utilisateur pourra ensuite accéder, par exemple pour globalement leur faire subir une transformation géométrique ou une duplication. Ces différentes observations nous conduisent aux règles suivantes, qui gouvernent la gestion des sous-programmes dans LIKE :

- Tout programme peut être considéré comme un sous-programme ; ses paramètres constituent les paramètres d'entrée du sous-programme.
- L'ensemble des objets créés au cours d'un sous-programme (contexte implicite) est intégré comme un tout dans le contexte implicite du programme appelant. Il ne pourra ensuite être manipulé dans celui-ci que globalement.
- À chaque exécution du sous-programme, un nouveau contexte est créé (réentrance).

Ces règles, implémentées dans LIKE, ont prouvé, malgré leur caractère restrictif en ce qui concerne les paramètres de sortie, leur utilisabilité. Cependant, la règle suivante peut être ajoutée :

- Les paramètres de sortie du sous-programme, en nombre obligatoirement fixe, sont insérés dans le contexte du programme appelant, permettant une manipulation individuelle dans ce dernier.

Notons que, lorsqu'un sous-programme est appelé à partir d'un autre (sous-) programme, ses paramètres effectifs sont introduits par l'intermédiaire de la calculette grapho-numérique, sous la forme d'une expression appartenant au contexte englobant. Il s'agit donc d'un réel mécanisme de sous-programme. Les règles de gestion du contexte que nous avons données ci-dessus permettent d'établir un cadre de réalisation pour toutes les structures de contrôle, comme nous allons le détailler dans la section suivante.

Alternatives, répétitions et contexte

Les alternatives et les répétitions peuvent être globalement considérées comme des sous-programmes simplifiés. Comme ces derniers, leurs variables implicites constituent un ensemble d'objets dont la manipulation individuelle n'a pas vraiment de sens, mais dont la manipulation globale peut/doit être autorisée.

À chaque structure, doit donc être associé un contexte propre (contexte implicite). Toutes les actions internes à la structure peuvent se référer non seulement aux variables de ce contexte, mais également à celles du contexte englobant (celui de la séquence dans laquelle est définie la structure de contrôle), qui définissent implicitement ses paramètres d'entrée. La gestion du contexte dynamique d'un programme (en cours d'exécution) sous la forme d'une pile de contextes, avec insertion globale de chaque contexte dans son contexte englobant, en fin d'exécution, permet ainsi de répondre aisément aux règles d'accessibilité des variables.

Ainsi, une alternative simple (SI-ALORS-SINON) aura-t-elle un contexte unique, constitué de deux sous-contextes, l'un pour la branche ALORS, l'autre pour la branche SINON. L'expression de contrôle de l'alternative pourra se référer exclusivement aux variables du contexte englobant. Pour une répétition, le contexte sera divisé en sous-contextes, un pour chaque tour de la répétition. Le contexte du tour courant sera accessible à l'expression de contrôle de la répétition. Enfin, pour permettre l'imbrication des structures répétitives, tout comme la réentrance pour les sous-programmes, les contextes implicites des alternatives et des répétitions sont créés au début de chaque exécution, puis fermés lors de l'achèvement de chaque structure, et inclus dans le contexte englobant.

Exemple

La figure 4 représente le texte schématisé d'un programme contenant une alternative et une répétition. Pour simplifier, nous n'avons représenté que les actions donnant lieu à création d'objet (qui sont repérés par leur numéro dans le contexte auquel ils appartiennent), et identifié les divers contextes qui sont associés au programme. L'indentation montre la structuration des contextes (le contexte de la répétition est en fait multiple, ce qui est symbolisé ici par l'indice i) :

Début ***Contexte principal (CP)***
Inst. 1 -> CP1
Inst. 2 -> CP2
SI Condition
ALORS ***Contexte Alors (CA)***
Inst 3 -> CA1
RÉPÉTER ***Contexte Répétition (CR$_i$)***
Inst 4 -> CR$_i$1
Inst 5 -> CR$_i$2
Inst 6 -> CR$_i$3
FIN RÉPÉTER -> CA2
SINON ***Contexte Sinon (CS)***
Inst 7 -> CS1
Inst 8 -> CS2
FIN SI -> CP3
Inst. 9 -> CP4

Fin

Figure 4 : Programme structuré

La figure 5 représente le contexte dynamique de ce même programme au cours de l'exécution du troisième tour de la répétition. La pile de contextes matérialise les possibilités d'accès au sein des variables des différents contextes (seul le contexte grisé est inaccessible ici).

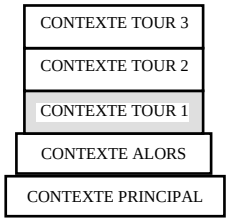

Figure 5 : Contexte dynamique du même programme.

Enfin, la figure 6 représente ce même contexte, à la fin d'une exécution ayant permis d'exécuter la branche ALORS, et de répéter quatre fois l'itération.

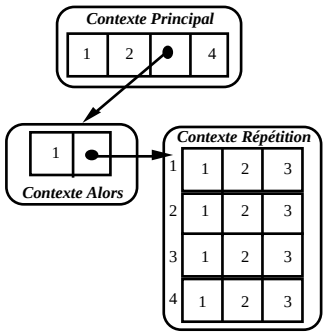

Figure 6 : Contexte en fin d'exécution

Récursivité

Compte tenu des principes que nous avons décrits ci-dessus, la prise en compte de la récursivité résulte simplement de l'application des règles définies (principalement celles concernant la réentrance), et de la définition de l'alternative qui termine la récursivité.

DÉFINITION INTERACTIVE DES STRUCTURES DE CONTRÔLE

Enregistrer par démonstration un programme séquentiel est une activité simple qui nécessite seulement l'identification des paramètres. Lorsque l'environnement de programmation, comme c'est le cas pour LIKE, permet la mise au point et la modification des programmes enregistrés, il faut de plus assurer une gestion fine du contexte, pour ajuster correctement la numérotation automatique des variables, lorsque l'utilisateur insère ou supprime une opération constructive. Il faut vérifier la cohérence du programme résultant de la modification. Ainsi, lorsque l'on supprime une action générant un objet, toutes les actions ultérieures utilisant cet objet deviennent incohérentes.

Permettre à l'utilisateur de construire interactivement des structures de contrôle présente des difficultés supplémentaires, tant au plan du système de PpD que de celui des conventions de dialogue à définir. Nous allons maintenant étudier ces différents points.

Sous-Programmes

La définition interactive des sous-programmes s'apparente tout à fait à celle des programmes. La seule différence concerne la définition éventuelle de paramètres de sortie. Définir *a priori* ces derniers constitue une tâche très abstraite, qui ne peut s'appuyer sur le déroulement de l'exemple. En revanche, définir *a posteriori* les seuls paramètres de sortie est une méthode plus accessible à l'utilisateur final : il suffit d'introduire une seule commande supplémentaire, **objet_accessible**. Lorsque l'utilisateur active cette commande et désigne un objet créé par le sous-programme, celui-ci est automatiquement et implicitement introduit dans la liste des paramètres de sortie du sous-programme, et est donc référencé directement dans le contexte englobant.

À l'usage, il s'est révélé que les besoins majoritaires dans le cadre de notre système CAO se résumaient, d'une part, à manipuler globalement les résultats du sous-programme, et d'autre part, à positionner globalement le résultat de son exécution. Pour faciliter cette deuxième opération, nous avons adjoint à la commande d'appel de sous-programme, qui permet de choisir le sous-programme voulu, puis de définir ses paramètres effectifs à l'aide d'expressions sur les constantes et les variables du programme en cours, la possibilité d'établir la transformation de positionnement.

Notons que l'appel de sous-programme suppose de la part du système de PpD un changement de mode. En effet, l'appel de sous-programme en phase d'enregistrement engendre le passage en mode exécution, pour permettre l'exécution de l'appel lui-même, c'est-à-dire celle du sous-programme. Le système de PpD revient au mode enregistrement dès la fin de cette même exécution. Cette alternance des modes enregistrement et exécution est requise également pour les alternatives, et surtout les répétitions.

Alternatives et expressions

La définition interactive de structures alternatives pose deux types de problèmes, ceux relatifs à la description et à l'évaluation de l'expression de contrôle de la structure, et ceux relatifs à la description du contenu de chaque branche. Introduire une expression de contrôle requiert de disposer d'un ensemble d'opérateurs permettant de définir cette expression.

Or, définir un langage de commande textuel incluant ces opérateurs irait à l'encontre des objectifs

de simplicité et de transparence pour l'utilisateur de la PpD. Pour résoudre ce problème, certains systèmes infèrent de plusieurs exemples la nature de l'expression permettant de différencier les cas (Turvy [1], Tinker [1]). Cependant, leur pouvoir d'expression demeure extrêmement limité. Dans le domaine de la CAO, un outil interactif puissant répond exactement à ce besoin d'expressions : tous les systèmes CAO proposent en effet une calculette grapho-numérique permettant de définir des expressions complexes, quoique généralement à résultat purement numérique. La transformation de cet outil, pour lui faire calculer des valeurs booléennes, constitue une solution simple et naturelle pour l'utilisateur pour la définition des expressions nécessaires aux alternatives.

La définition interactive proprement dite de l'alternative demande quant à elle un effort supplémentaire à l'utilisateur. Quatre commandes supplémentaires sont introduites : SI, ALORS, SINON et FIN SI. L'utilisateur ouvre la structure par la commande SI, et définit l'expression par la calculette grapho-numérique. Le système évalue ensuite l'expression et ouvre la branche correspondant à la valeur de l'expression. À l'issue de la définition de la première branche, l'utilisateur peut soit forcer l'ouverture de l'autre branche (les deux branches sont alors définies sur le même exemple, les résultats apparaissent tous deux sur l'écran, mais ne sont désignables que globalement), soit fermer la structure (FIN SI), modifier les paramètres du programme en cours, et réexécuter le programme jusqu'au SI qui, en fonction de la valeur de l'expression, ouvre alors l'autre branche. Deux exemples différents peuvent ainsi être utilisés pour définir les deux branches. Ces deux modalités permettent une grande souplesse d'utilisation, et se sont révélées tout aussi utiles.

Répétitions

La définition interactive des structures répétitives consiste à définir, outre les actions à effectuer à chaque tour et la condition d'arrêt de la répétition, la relation de récurrence qui unit les objets de deux itérations successives. Si les itérations de collection utilisent une relation de récurrence implicite ('prendre l'objet suivant dans la collection, et lui appliquer les mêmes actions' comme le 'Graphical search and replace' ou le 'Constraint-based search and replace' de Kurlander [3], ou encore Eager [1]), les itérations générales (itérations de récurrence) nécessitent la définition de cette relation. Ceci peut se faire par déduction, comme dans les 'Constraints fom Multiple snapshots' de Kurlander [3], mais les procédés utilisés sont, de l'avis même des auteurs, toujours limités [1]. L'autre solution consiste à permettre la définition explicite sur l'exemple de cette relation.

Illustrons ce point à l'aide de l'exemple représenté sur la figure 7. Il s'agit "d'empiler" des cercles d'un rayon divisé par deux à chaque étage, jusqu'à atteindre un rayon minimum. Le programme peut se résumer à une répétition, pour faire une suite de cercles empilés, puis à une symétrie de l'ensemble. Un pas quelconque de la répétition (sauf le premier) peut se définir ainsi : *construire un premier cercle tangent au cercle du tour précédent et à l'axe central, de rayon égal à la moitié de celui du cercle du tour précédent, puis créer un cercle de centre identique au dernier créé, et de rayon égal à la moitié de son rayon.*

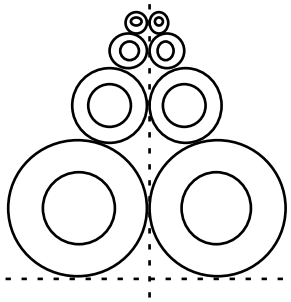

Figure 7 : Dessin souhaité

Cette description appelle trois remarques : (1) les objets d'une branche se définissent par rapport aux objets du tour précédent ou du tour courant ; c'est le propre des itérations de récurrence ; (2) la séquence d'actions qui constitue le premier tour est identique à celle des autres tours ; la seule différence est que, en l'absence de tour précédent, les références se font à des objets du contexte englobant : le premier cercle est tangent à deux droites préexistantes ; (3) les relations de récurrence peuvent être entièrement définies en réexécutant une deuxième fois les actions du premier tour, et en demandant seulement à l'utilisateur, pour chacun des objets désignés lors du premier tour, si le même objet doit être utilisé pour le tour suivant (c'est le cas de la droite verticale) ou si un objet **construit au premier tour** doit remplacer l'objet du contexte englobant (c'est le cas de la droite horizontale, remplacée par le premier cercle construit lors du premier tour).

Ces trois remarques induisent l'interface utilisateur et les conventions de dialogue que nous avons définies dans le système LIKE. Deux commandes supplémentaires sont à la disposition de l'utilisateur (RÉPÉTER et JUSQU'À) Après avoir sélectionné la commande RÉPÉTER, l'utilisateur définit le premier tour. Pour ce faire, il accède aux objets du contexte englobant (qui peut être multiple, voir figure 5). Il sélectionne ensuite la commande JUSQU'À, sans introduire l'expression de contrôle, qui ne le sera qu'à l'issue du second tour.

Le système passe alors automatiquement en mode exécution, pour définir le deuxième tour. Tout en exécutant automatiquement les commandes, il met en évidence (clignotement) les objets du contexte englobant désignés (ici, la droite horizontale, dans un premier temps). L'utilisateur peut alors seulement (1) désigner la même droite, qui sera alors utilisée dans

tous les tours, ou (2) désigner un objet du premier tour (ici le premier cercle créé) ; une relation de récurrence est alors créée, reportée de tour en tour. De même, les expressions utilisées qui mettent en jeu des objets du contexte englobant (ici le paramètre 'rayon') sont mises en évidence ; l'utilisateur peut alors (1) confirmer leur utilisation à l'identique pour tous les tours, ou (2) définir, à l'aide de la calculette grapho-numérique, une nouvelle expression pouvant mettre en jeu à la fois des éléments du contexte englobant, ou des deux tours de la répétition (dans l'exemple, Rayon_de (Premier cercle du tour précédent) / 2). Lors de cette exécution du deuxième tour, les références à des objets internes au premier tour sont automatiquement transformées en références au tour courant (par exemple, le centre du premier cercle créé dans le tour courant).

Lorsque tous les objets ont été ainsi redésignés, le système affiche le message 'JUSQU'À ?' et l'utilisateur introduit l'expression de contrôle, à l'aide de la calculette grapho-numérique, accédant pour ce faire à la fois aux objets des deux tours de la répétition et ceux du contexte englobant. À la fin de cette définition, le système exécute toutes les itérations suivantes, les références utilisées dans les relations de récurrence étant mises à jour de tour en tour.

La figure 8 ci-dessous montre le contenu du programme finalement enregistré. Chaque désignation d'objet ou expression dans la répétition apparaît entre crochets, un symbole '#' précédant les entrées effectuées par l'utilisateur pendant le second tour (chaque référence à un objet de la répétition est suivie d'une lettre permettant de différencier son appartenance au tour précédent, P, ou au tour courant, C).

Début
Paramètre *(Rayon)*
Paramètre *(Min)*
Horizontale (0) -> CP 1
Verticale (0) -> CP 2
RÉPÉTER
CercleTangent à [*CP1 # CR1P*] [*CP2 # CP2*]
de rayon
[*Rayon #* **Rayon_De** (*CR 1C*) / **2**] -> CR 1
Cercle de Centre Centre_De [*CR 1C*]
de rayon Rayon_De [*CR 1C*] / **2** -> CR 2
JUSQU'À Rayon_De[*CR2C*] < *Min* -> CP 3
Symétrie par rapport à *CP 2* **de** *CP 3* -> CP 4
Fin

Figure 8 : Programme final

CONCLUSION ET PERSPECTIVES

Nous avons présenté dans ce papier la Programmation par Démonstration, et montré comment la CAO constituait un domaine d'application privilégié pour ce mode de programmation. Après avoir relevé le manque de complétude des systèmes actuels de PpD, principalement en ce qui concerne les structures de contrôle, nous avons démontré le rôle essentiel de la gestion du contexte pour résoudre ce problème. Puis, nous avons présenté le système LIKE, système de PpD qui présente la caractéristique de supporter les cas les plus généraux de structures de contrôle de la programmation structurée ou de la programmation fonctionnelle, tout en ne mettant en œuvre aucun mécanisme d'inférence. C'est en particulier, à notre connaissance, le premier système qui propose une définition par démonstration des répétitions basées sur une relation de récurrence quelconque.

Ce système s'appuie sur une gestion automatique du contexte dynamique du programme implicite. Ce contexte associe à chaque objet créé dans l'exemple une variable dont l'identificateur est l'ordre de création de l'objet. Sous-programmes et structures de contrôle sont associés à une variable unique, mais dont le contenu, accessible globalement, est structuré. Il est également basé sur l'utilisation d'une calculette grapho-numérique permettant de définir des expressions numériques ou booléennes. Le programme dispose d'une visualisation textuelle, analogue à celle présentée dans les exemples, mais l'essentiel du développement est effectué graphiquement, le système permettant d'exécuter le programme pas à pas, et de le modifier par ajout ou suppression de séquences constructives.

Le domaine d'application du système est la CAO. C'est un domaine dans lequel les objets possèdent une représentation graphique intrinsèque, et dans lequel les utilisateurs sont habitués à exprimer graphiquement des relations entre objets. Au prix d'un très faible entraînement, le système est donc parfaitement exploitable pour un utilisateur ordinaire. Dans d'autres domaines moins favorables, les principes et les conventions de dialogue que nous avons définies doivent pouvoir s'appliquer de façon analogue. Ils risquent cependant d'apparaître moins intuitifs pour les utilisateurs.

Développé dans le cadre d'un contrat, le système LIKE a été évalué par la société contractante, et a démontré l'intérêt de la démarche proposée. Son utilisation sur un exemple complexe de pièce faisant intervenir alternatives et répétitions, en parallèle avec un développement classique en FORTRAN, a abouti à un gain d'un facteur 5 par rapport à ce dernier.

Cependant, ce système ignore la sémantique des commandes qu'il enregistre, afin d'assurer une indépendance maximum avec le système sur lequel il est greffé. Ceci présente l'inconvénient d'enregistrer un programme qui ne peut s'exécuter que sur le système ayant servi à le construire. Afin d'assurer la portabilité des programmes ainsi construits par démonstration, le système LIKE a donné naissance à un autre système, le système EBP [8], qui permet de générer un programme neutre, en FORTRAN, s'appuyant sur une Interface de Programmation

d'Application (API) normalisée, et dont la pré-industrialisation est prévue dans le cadre du projet ESPRIT PLUS (Parts Library Usage and Supply, Project N° 8984).

BIBLIOGRAPHIE

1. CYPHER A. : Watch what I do, Programming by Demonstration, *MIT Press, 1993, 635p.*

2. GIRARD P. : Environnement de programmation pour non-programmeurs et paramétrage en conception assistée par ordinateur : le système LIKE, *Thèse de l'Univ. de Poitiers, 1992, 195p.*

3. KURLANDER D. : Graphical Editing by Example, *Ph.D Thesis, Columbia University, 1993.*

4. MYERS B. : Report on the end-user programming working group, *in Languages for developing user interfaces,Myers Ed., Jones and Bartlett, Boston, 1992, pp. 343-366.*

5. MYERS B. : Taxonomies of Visual Programming and Program Visualization, *J. of Visual Lang. and Comp., 1, 1990, pp. 97-123.*

6. NARDI B. A. : A small matter of programming, perspectives on end user computing, *MIT Press, Cambridge, Massachusetts, 1993, 157p.*

7. OLSEN D. R., DANCE J. R. : Macros by Example in a Graphical UIMS, *IEEE Comput. Graphics and Appl., Jan. 1988, pp. 68-78.*

8. POTIER, J.C., GIRARD, P., PIERRA, G., BESNARD F., *Rev. d'Automatique et de Productique Appliquée (RAPA), 8, 2-3, 1995, pp. 229-234.*

9. ROLLER D., SCHONEK F., VERROUST A. : Dimension-driven geometry in CAD : a survey, *in Theory on practice of geometric Modeling, Springer Verlag, 1990, pp. 509-523.*

10.SASSIN M. Creating User-Intended Programs with Programming by Demonstration, *Proc. IEEE Symp. on Visual Languages, Saint-Louis, Missouri, 4-7 Oct. 1994, pp. 153-160.*

11.SOLANO L., BRUNET P., Constructive constraint-based model for parametric CAD systems, *CAD, 26, 8, 1994, pp. 614-621.*

12.VAN EMMERIK M. : Interactive design of parametrized 3D models by direct manipulation, *PhD Thesis, Delft University, NETHERLAND, 1990, 141p.*

13.WILDE N., WYSIWYC (What You See Is What You Compute) Spreadsheet, *IEEE Symp. on Vis. Lang., 1993, pp 72-76*.