



BD/MSPS 

Guide de programmation

Le présent guide décrit la façon d'utiliser le MSPS en mode boîte de verre. Le mode boîte de verre permet à l'utilisateur d'ajouter des variables et des paramètres au MSPS ainsi que de modifier les algorithmes du MSPS ou d'en mettre en œuvre de nouveaux. Le compilateur C Microsoft est nécessaire pour l'utilisation en mode boîte de verre.



Statistics
Canada

Statistique
Canada

Canada

Table des matières

Introduction.....	1
Objet du mode boîte de verre.....	1
Exigences de matériel et de logiciel pour le mode boîte de verre	4
Connaissance requise de la programmation.....	4
Connaissances requises du système d'exploitation.....	4
Concepts de base de programmation (qui ne sont pas propres à un langage)5	
Connaissance du langage de programmation C	5
Exemple de Départ rapide.....	6
Preliminaires	7
Changer l'environnement du projet	8
Modifier la fonction de pilote de rechange (Adrv . cpp)	8
Modifier la fonction pour les allocations familiales (Afamod . cpp).....	9
Essai du modèle MSPS obtenu	11
Résumé.....	14
Le BD/MSPS et la structure du répertoire de la boîte de verre	14
La structure Individus/ ménages de la BDSPS	17
La structure des données du BD/MSPS	18
Présentation des pointeurs dans le BD/MSPS	19
Le bestiaire.....	19
Exemples de boucles.....	20
Références relatives à un individu	22
Résumé.....	23
Structure d'appel des fonctions du MSPS	23
Développement en boîte de verre : ajout de paramètres scalaires ordinaires	25
Procédure générale de modification en boîte de verre : une récapitulation.....	25
Créer un sous-répertoire de travail.....	26
Déterminer les fichiers à modifier	26
Copier les fichiers pertinents vers un sous-répertoire de travail.....	26
Modifier les fichiers pertinents	26
Compiler la nouvelle version.....	27
Tester la nouvelle version du modèle	27
Effectuer l'analyse prévue	27
Présentation de l'ajout des paramètres.....	27
Copier les fichiers Adrv . cpp , Mpu . h , Ampd . cpp , Afamod . cpp ,	
SPSMGL . dsw.....	28
Mise à jour du projet.....	29
Mise à jour de la description de l'algorithme dans Adrv . cpp.....	29
Modifier Mpu . h afin de définir les nouveaux paramètres.....	29
Modifier Ampd . cpp de façon à rendre les paramètres disponibles au MSPS.....	30
Modifier les fonctions qui utilisent le nouveau paramètre.....	32
Valider et faire des exécutions de production en boîte noire.....	33
Résumé/ conclusion	35
Développement en boîte de verre : ajout de paramètres inhabituels	35
pmaddent : La fonction et ses arguments.....	36

Description des paramètres scalaires	40
Paramètres RÉEL/flottant/NUMBER	40
Paramètres ENTIER/int	40
Paramètres de DRAPEAU/FLAG	40
Paramètres FRACTION	40
Paramètres OPTION	40
Paramètres EDIT-FRACTION	41
Paramètres FICTIFS/DUMMY	41
Vecteurs de paramètres définis par l'utilisateur.....	41
Ajouts aux fichiers Mpu . h , Cpu . h ou Apu . h	42
Ajouts au fichier Ampd . cpp	43
Références aux vecteurs de paramètres définis par l'utilisateur dans le code source.....	44
Spécification des valeurs de vecteur de paramètres.....	44
Résumé.....	45
Tableaux définis par l'utilisateur pour les recherches	46
Types de tableaux et fonctions de recherche	47
Présence dans les fichiers d'en-tête du MSPS	47
Présence dans les appels de pmaddent dans Ampd . cpp	48
Emploi des références au tableau dans le code utilisateur	49
Présence dans les fichiers de paramètres	49
Points clés pour l'ajout de paramètres de tableau	50
Ajout de matrices de paramètres.....	51
Présence dans Mpu . h	51
Présence dans Ampd . cpp.....	52
Référence aux éléments de la matrice dans le code de source.....	53
Présence dans les fichiers de paramètres	53
Résumé/Conclusion	54
Développement en boîte de verre : ajout de nouvelles variables.....	55
Aperçu de l'ajout de variables	55
Caractéristiques et types de variables dépendantes	56
Les fonctions vardef et stradd ainsi que leurs arguments	57
Argument «Name» de Vardef (et définition de la souche du nom la variable) :	57
Argument «Home Structure» de Vardef :	58
Argument «Variable Location» de Vardef :	58
Argument «Type-C» de vardef (C_NUM & C_INT) :	58
Argument (Type) «Utilisation» de Vardef (V_ANAL & V_Clas) :	58
Appels de Stradd pour les variables d'analyse numérique :	59
Appels de stradd pour les variables d'analyse à nombre entier :	59
Appels de stradd pour les variables de classe à chiffre entier :	60
L'exemple de supplément d'allocations familiales, prolongé	61
Modifications des fichiers de projet et Adrv . cpp.....	62
Modifications de vsu.h.....	62
Modifications du fichier vsdu . cpp	63
Modifications du fichier Afamod . cpp (ou, plus généralement, tout nouveau	

code de source de base).....	63
Identification des chaînes.....	64
Variables locales	64
Calcul et affectation des nouvelles variables de modèle	64
Compilation.....	66
Validation.....	66
Résumé/Conclusions	69
Modification des variables de données de base et de variante.....	70
Modifications qui touchent tous les systèmes fiscaux/de transfert d'un modèle.....	71
Modification typique de la croissance du revenu et de la population par les fichiers APR et API	71
Modifications exigeant une nouvelle logique pour le fichier <code>adj u . cpp</code>	72
Ajout de nouveaux paramètres d'ajustement de la base de données	73
Un exemple pratique	74
Liste de vérification pour le changement «global» des variables de base de données	78
Modifications qui touchent seulement la base ou seulement la variante	79
Mise en oeuvre des changements dans <code>Aca11 . cpp</code>	80
Un exemple pratique	82
Liste de vérification pour les changements apportés à la base de données propres au système	89

Introduction

Le *Guide de programmation* décrit la façon dont l'utilisateur peut modifier le MSPS afin de modéliser des systèmes fiscaux/de transfert ou des options de politiques qui ne peuvent pas être traitées par le BD/MSPS tel qu'il est installé à l'origine; ainsi, vous pouvez modifier la logique du système fiscal/de transfert afin d'évaluer les répercussions de la distribution statique qui serait entraînée par un projet de politique.

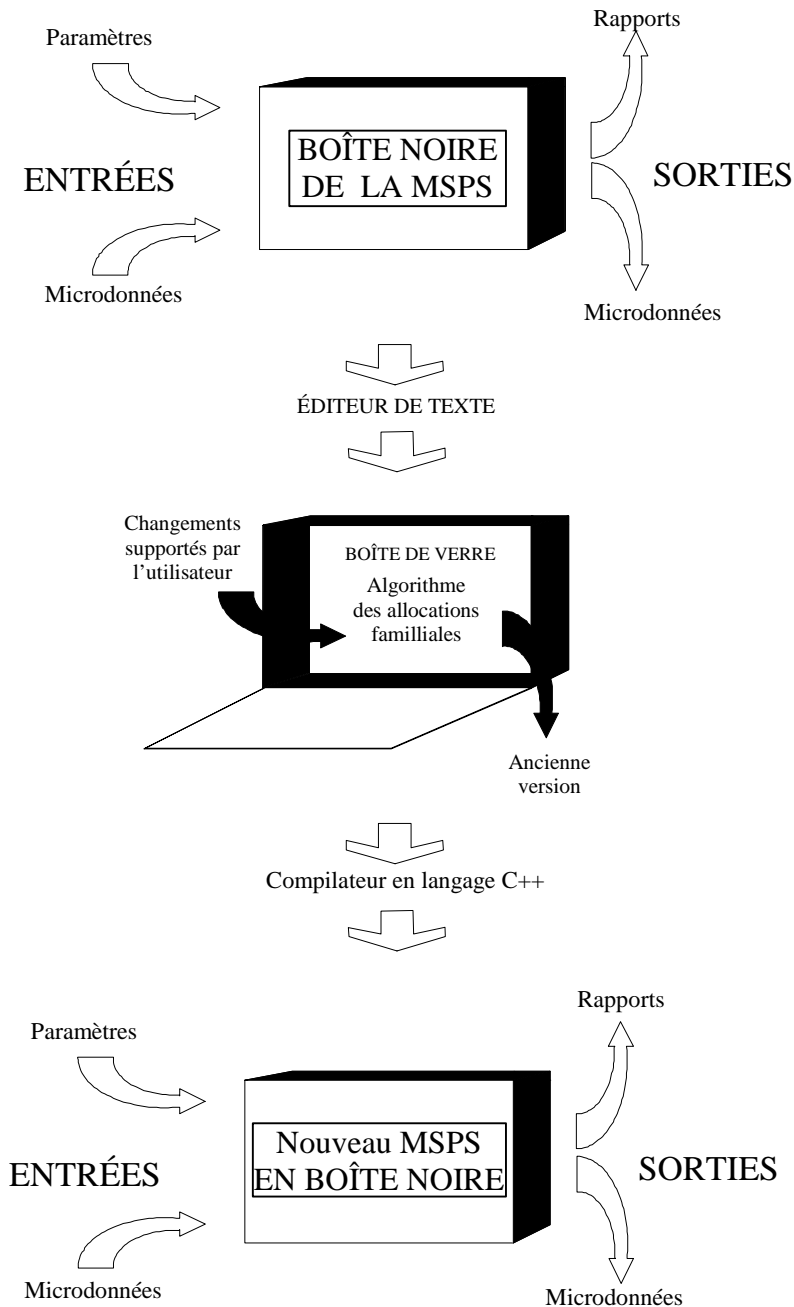
Le présent chapitre présente diverses rubriques préliminaires essentielles à la compréhension de l'utilisation du MSPS en mode boîte de verre. Voici des rubriques bien précises du chapitre :

- (1) une description du mode boîte de verre, particulièrement en regard du mode boîte noire,
- (2) les exigences de matériel et de logiciel en vue de l'utilisation du mode boîte de verre,
- (3) le niveau de connaissance de la programmation qui est nécessaire.

Les sections subséquentes du Guide expliquent en détail l'élaboration d'applications en mode boîte de verre. La section suivante décrit la marche à suivre pour un «*départ rapide*» qui vérifie la réussite de l'installation du MSPS par la mise en œuvre d'une modification simple en mode boîte de verre de la version du MSPS installée au départ. La section intitulée *Le BD/MSPS et la structure de répertoires du mode boîte de verre* explique la structure des sous-répertoires pertinente aux divers aspects des opérations effectuées en mode boîte de verre. La *Structure individus-ménages de la BDS* donne des détails essentiels sur les principales structures de données de la BDS utilisées par le MSPS. La *Structure d'appel du MSPS* décrit la structure d'appel des modules du MSPS qui constituent un modèle en particulier. La section *Développement en boîte de verre : ajout de paramètres scalaires ordinaires* traite des mécanismes utilisés pour l'ajout de paramètres de modèles définis par l'utilisateur à un modèle MSPS, et elle touche les formes les plus courantes de paramètres scalaires. La section *Développement en boîte de verre : ajout de paramètres inhabituels* poursuit avec l'ajout de genres moins courants de paramètres scalaires ainsi qu'avec l'ajout de vecteurs et de matrices de nouveaux paramètres de modèle. La section *Développement en boîte de verre : ajout de nouvelles variables* décrit l'ajout de nouvelles variables à un modèle. La section *Changer des variables de données de base et de variante* indique finalement la façon de faire la gestion des algorithmes standard et de rechange dans le contexte des opérations en mode boîte de verre.

OBJET DU MODE BOÎTE DE VERRE

Voici une vue simplifiée des systèmes participant au processus de simulation fiscale et de transfert :



Un utilisateur spécifie une série d'entrées (paramètres et données) qui sont ensuite traitées par un système d'algorithmes (la boîte noire) qui à son tour produit des sorties de système (tables et microdonnées). L'utilisateur peut créer de nombreuses simulations différentes en variant les entrées et en analysant les sorties. Il peut même déduire une partie du contenu de la boîte noire par des essais répétés. Cependant, les simulations possibles sont limitées par le contenu de la boîte noire. Si, par exemple, les règles qui régissent la taxe sur les ventes des fabricants ne sont pas comprises dans le système d'algorithmes (avec des mécanismes pour des paramètres et des données d'entrée appropriés), alors ce programme ne peut pas être

simulé sans que l'on ait à ouvrir et à modifier vraiment la boîte noire. La capacité d'examiner l'intérieur de la boîte noire et d'en modifier le contenu se compare à la boîte noire que l'on transforme en boîte de verre.

Le présent guide explique la façon dont on procède pour utiliser le MSPS en mode boîte de verre. Plus particulièrement, le terme «mode boîte de verre» désigne une méthode de modification de la version exécutable du programme MSPS pour la réalisation d'analyses qui ne sont pas possibles avec le MSPS original et non modifié. Le mode boîte de verre peut être utilisé pour l'ajout ou la modification de paramètres, de variables et d'algorithmes. L'utilisation du mode «boîte de verre» implique la modification du code source en langage C++ et la compilation d'une nouvelle version exécutable du programme. Le «mode boîte noire» désigne l'exécution subséquente d'une version exécutable, soit celle qui est expédiée par Statistique Canada, soit celle qui a été modifiée par l'utilisateur en mode «boîte de verre». C'est toujours en mode boîte noire qu'un utilisateur fait une variété de simulations pertinentes à une politique par le changement de paramètres, de variables utilisateur et d'expressions de totalisation.

Du fait des étapes supplémentaires qu'implique l'utilisation en mode boîte de verre, l'utilisateur devrait l'éviter autant que possible. Le MSPS possède un certain nombre de dispositifs qui permettent à l'analyste d'obtenir de nombreux résultats désirés sans reprogrammation. La technique la plus courante consiste pour l'utilisateur à modifier les valeurs par défaut des paramètres de programme qui commandent le MSPS. L'analyste pourrait simuler les répercussions d'une augmentation ou de l'abolition des allocations familiales en modifiant les valeurs numériques des paramètres pertinents. Dans un second exemple, l'analyste peut définir ses propres variables dans un fichier de paramètres de commande et utiliser les variables obtenues dans une vaste gamme de sorties du MSPS. Le Guide d'introduction donne un exemple détaillé dans lequel l'analyste utilise les variables définies par l'utilisateur pour simuler un crédit d'impôt pour revenu gagné. De même, l'analyste peut définir «à la volée» des variables comme des expressions et les exporter ou les entrer dans les tableaux comme s'il s'agissait de vraies variables et il peut représenter de façon commode les écarts qu'il y aura pour une variable donnée entre les systèmes fiscaux/de transfert de base et de variante. Le Guide d'utilisation des tableaux croisés donne plusieurs exemples de ce type de définition à la volée.

Le mode de boîte de verre doit être utilisé dans les conditions suivantes :

- (1) Ajout de nouveaux paramètres.
- (2) Ajout de nouvelles variables qui exigent une référence à d'autres membres de famille en particulier.
- (3) De nouvelles propositions qui sont conçues de façon à avoir des interactions avec le système fiscal/de transfert. Par exemple, des allocations imposables pour les nouveaux-nés.
- (4) De nouvelles propositions qui modifient la logique des programmes existants de façons qui n'ont pas été prévues dans les paramètres.

Si l'utilisateur a besoin de faire de tels changements dans le MSPS afin de refléter de nouveaux systèmes fiscaux/de transfert, il doit connaître à fond les techniques décrites dans le présent guide.

EXIGENCES DE MATÉRIEL ET DE LOGICIEL POUR LE MODE BOÎTE DE VERRE

Le *Guide d'installation* donne en détail les exigences relatives au matériel et au logiciel. Pour la plupart des utilisateurs, une imprimante est une nécessité pratique. L'étude qui suit suppose qu'il y en a une dans le système.

Les principaux aspects des besoins en logiciel sont les suivants :

1. L'utilisation du BD/MSPS en mode boîte de verre exige la présence de Visual C++ qui sert à compiler les instructions du code source en langage C de l'utilisateur en langage machine exigé par le MSPS.
2. Le MSPS lui-même exige un système d'exploitation compatible avec la version de Visual C++.
3. L'utilisateur doit avoir un éditeur approprié pour l'entrée ou la modification du code source en langage C, cette édition étant au centre de l'utilisation du mode boîte de verre.
4. Il est suggéré d'utiliser un **éditeur de texte efficace compatible avec le code C++**.

L'utilisateur qui s'attend à faire une grande utilisation du MSPS en mode boîte de verre désirera aussi probablement une efficacité supplémentaire fournie par les logiciels «utilitaires» comme la trousse d'outils MKS qui rend de nombreuses fonctions de style Unix disponibles à l'intérieur du système d'exploitation.

CONNAISSANCE REQUISE DE LA PROGRAMMATION

Parce que l'utilisation du MSPS en mode boîte de verre exige que l'utilisateur fasse une certaine programmation, l'utilisateur de la boîte de verre devra avoir quelques connaissances de plus que l'utilisateur habituel du mode boîte noire. La présente section énonce les genres de choses que l'utilisateur de la boîte de verre devra soit connaître, soit être prêt à apprendre.

Connaissances requises du système d'exploitation

L'utilisation du MSPS en mode boîte de verre exige que l'utilisateur soit bien à l'aise dans un certain nombre de domaines du système d'exploitation. L'utilisateur a besoin de savoir ce qui touche les unités de disque, les fichiers, les règles d'attribution de noms aux fichiers ainsi que les répertoires et sous-répertoires.

L'utilisateur devrait connaître à fond le concept de l'environnement DOS et les variables d'environnement comme la variable PATH. Le fonctionnement efficace du MSPS en mode boîte de verre exige aussi que l'utilisateur connaisse très bien un certain nombre d'instructions DOS. Les instructions DOS les plus nécessaires sont :

DIR	Affiche le contenu du répertoire
TYPE	Liste le contenu d'un fichier
MKDIR	Crée un nouveau répertoire
CHDIR	Change de répertoire courant
RMDIR	Supprime le répertoire

COPY	Copie des fichiers
XCOPY	Copie des fichiers et (ou) des répertoires
DEL	Supprime un fichier
SET	Modifie et (ou) affiche des variables d'environnement
PATH	Affiche le chemin courant

L'utilisateur qui n'est pas à l'aise avec les concepts et les instructions décrites ici évitera probablement beaucoup de frustrations en consacrant quelque temps à consulter le manuel de DOS ou à acquérir quelques aptitudes à le faire en environnement Windows avant de se lancer dans des applications réelles en boîte de verre.

Concepts de base de programmation (qui ne sont pas propres à un langage)

Le mode boîte de verre du MSPS n'est pas la place pour apprendre votre premier langage de programmation. L'utilisateur doit connaître à fond au moins un langage informatique de haut niveau (p. ex., le FORTRAN, le BASIC, le PASCAL et le SAS) avant d'utiliser la boîte de verre. Parce que les applications en boîte de verre exigent de la programmation dans un langage compilé, il est souhaitable que l'utilisateur de la boîte de verre s'attaque à la tâche une fois qu'il est à l'aise avec les concepts essentiels. L'utilisateur doit être à l'aise avec l'idée d'utiliser un éditeur de texte pour rédiger ou réviser le code source et avec l'idée d'utiliser un compilateur pour produire le fichier exécutable désiré. L'utilisateur bénéficiera d'une connaissance approfondie des notions de bibliothèques, de macros, de programmation modulaire et de validation de programme.

Il est même essentiel que l'expérience de l'utilisateur avec ces concepts soit appliquée. De préférence, avant de s'attaquer à des applications en boîte de verre du MSPS, l'utilisateur devrait toujours avoir rédigé et mis au point plusieurs programmes informatiques inhabituels, et pas nécessairement en langage C. Bien qu'il puisse arriver qu'un utilisateur apprenne à programmer en utilisant le MSPS, nous recommandons de ne pas faire ce genre de tentative. Pour l'éventuel utilisateur du MSPS qui aurait besoin d'acquérir ou de renforcer des compétences fondamentales en programmation, il existe divers ouvrages sur la programmation.

Connaissance du langage de programmation C

Parce que les applications en boîte de verre du MSPS exigent la programmation en langage C, l'utilisateur doit aussi programmer en C. Bien que la structure du MSPS fait en sorte que certaines choses comme les entrées et les sorties sont faites pour l'utilisateur, l'utilisateur éventuel sera plus efficace si les notions sont déjà comprises. L'utilisateur doit comprendre le but de la définition des constantes et de la déclaration des variables et il doit évaluer la portée de ces déclarations. Il doit comprendre les variables et les types de variables, particulièrement les variables pointeurs et les variables structurées, ainsi que la façon dont le langage C les utilise. Il doit comprendre la nature et la structure des fonctions et la variété des instructions qui les composent. Il doit connaître à fond les grands cheminement des instructions de commande du langage C (if-else, switch, while, for, do-while), ainsi que les tables d'affectations et d'opérateurs du langage C, dont les opérateurs d'incrément. Pour l'utilisateur qui a déjà travaillé avec d'autres langages de programmation et qui peuvent absorber cette information sous forme concentrée, le livre de

Kernighan et Ritchie, «The C Programming Language» est la référence standard. De même, la manuel du langage C qui est livré avec le compilateur Microsoft C Optimizing est une source très utile et fiable d'information sur le langage C et sa mise en oeuvre.

Enfin, bien sûr, l'utilisateur du MSPS doit comprendre les éléments du compilateur Microsoft C. Il lui faut aussi comprendre tout ce qui se passe et les divers messages d'erreur que le compilateur peut donner à la suite du traitement du code de l'utilisateur. L'autorité dans ces domaines est, bien sûr, l'ensemble des manuels de Microsoft livré avec le compilateur C.

Exemple de Départ rapide

Comme son titre l'indique, ce chapitre donne à l'utilisateur un départ rapide dans l'utilisation du MSPS en mode boîte de verre. Le chapitre a trois grandes fonctions. En premier lieu, il permet à l'utilisateur de vérifier l'installation du compilateur et du BD/MSPS. Si l'utilisateur peut réussir à exécuter l'exemple simple du chapitre, alors toutes les grandes parties de l'installation ont été exécutées correctement. En deuxième lieu, les exemples présentent la terminologie et les concepts essentiels de la boîte de verre. En troisième lieu, les exemples illustrent, de façon intégrée, le cheminement général des applications du mode boîte de verre.

La technique utilisée dans ce chapitre est principalement narrative. Amenant le lecteur au fil de toutes les étapes d'une application en boîte de verre simplifiée, la chapitre porte uniquement sur la méthode générale. Il décrit les détails essentiels de l'exercice, mais ne tente pas d'être exhaustif. L'illustration particulière donnée ici a été choisie pour sa simplicité; elle aborde les aspects les plus critiques des applications en boîte de verre, mais elle ne s'embourbe pas dans les exigences essentielles associées à des applications plus ambitieuses.

L'exemple modélise substantiellement un changement relativement simple apporté à un programme de transfert simple, les allocations familiales, dans le système fiscal/de transfert. Notre analyste hypothétique, intrigué par la pratique de l'Île-du Prince-Édouard, en 1970, cherche à vérifier les répercussions qu'auraient sur la distribution et sur l'agrégation les prestations supplémentaires d'allocations familiales aux grosses familles. Plus précisément, dans le système de variante, l'analyste désire accroître le montant des allocations familiales fédérales de 10 \$ par mois par enfant pour certains enfants, dans certaines familles. Lorsqu'une famille a trois enfants ou plus qui ont actuellement de 0 à 17 ans, alors elle reçoit, au cours de l'année, un montant supplémentaire égal à 120 \$ multiplié par le nombre d'enfants «supplémentaires», c'est-à-dire 120 \$ pour une famille de trois enfants, 240 \$ pour une famille de quatre enfants, etc. Supposons que la prestation supplémentaire d'allocations familiales serait versée par le gouvernement fédéral aux prestataires habituels et que la prestation serait traitée tout comme une prestation régulière des allocations familiales fédérales.

Comme l'indique l'exposé narratif, le lecteur ne devrait pas s'interroger sur les «pourquoi» de la mise en œuvre. Les sections subséquentes du Guide de programmation les expliquera tous plus à fond. Cependant, il est fortement souhaitable que l'utilisateur exécute l'exemple au complet, jusqu'au point d'effectuer réellement toutes les tâches décrites. Ce n'est que de

cette manière que le premier but de l'exercice peut être atteint, c'est-à-dire que le processus d'installation peut être vérifié.

PRÉLIMINAIRES

L'utilisateur devrait commencer en sélectionnant un sous-répertoire dans lequel il pourra travailler. C'est le répertoire du disque dur dans lequel l'utilisateur modifiera les copies des fichiers de code de source en langage C qui nous intéressent et décrira la nature du nouveau système. **Nous recommandons fortement que l'utilisateur utilise un répertoire autre que ceux qui ont été créés à l'installation du BD/MSPS pour le BD/MSPS lui même.** L'utilisateur peut créer un nouveau sous-répertoire si nécessaire. Pour les fins du présent exposé, nous supposons qu'un sous-répertoire nommé GLASSEX1 peut être utilisé comme sous-répertoire de travail.

L'utilisateur commence le processus en copiant, du sous-répertoire GLASS du BD/MSPS au sous-répertoire de travail GLASSEX1, tous les fichiers de gabarit pertinents. Les fichiers de gabarit sont les fichiers qui contiennent déjà la plus grande partie de l'information nécessaire pour une application en boîte de verre et que l'utilisateur modifiera pour créer les versions finales nécessaires à l'application. Pour cet exemple, les fichiers de gabarit pertinents sont les suivants :

1. `Adrv.cpp`, le gabarit de «pilote» de rechange qui invoque toutes les fonctions fiscales/de transfert du MSPS, dans le bon ordre. Ce gabarit, livré avec le MSPS, est en réalité une réplique de la fonction de pilote de base (**l'utilisateur devrait le copier dans son sous-répertoire de travail**).
2. `Afamod.cpp`, le gabarit d'allocations familiales de rechange qui calcule la prestation d'allocations familiales. Ce gabarit, livré comme élément du MSPS, est effectivement une réplique de la fonction `famod.cpp` du système de base qui calcule les prestations d'allocations familiales. (**L'utilisateur devrait le copier dans son sous-répertoire de travail**).
3. `SPSMGL.sln` et `SPSMGL.vcproj` font la compilation et établit les liens du nouveau modèle de l'utilisateur (**il faut copier ces fichiers de /spsm/glass à votre sous-répertoire de travail**).

Pour d'autres applications en boîte de verre, l'utilisateur peut aussi avoir besoin de copier d'autres gabarits fiscaux/de transfert et d'autres fichiers d'en-tête en langage C. Dans cet exemple, cependant, l'utilisateur n'a besoin de modifier aucun des fichiers d'en-tête parce que le modèle ne crée aucune nouvelle variable et n'utilise aucun nouveau paramètre officiel.

La procédure générale pour notre application en boîte de verre utilisée pour des fins d'illustration est tout ce qu'il y a de simple.

1. Sur des COPIES des fichiers `Adrv.cpp` et `Afamod.cpp`, nous faisons un petit nombre de changements décrits ci-dessous.
2. Nous ouvrons ensuite l'utilitaire solution `SPSMGL.sln` dans Visual Studio .net. Pour

le travail avec le nouveau modèle, le projet devrait être recompilé de façon à produire un nouveau fichier exécutable (nous supposons que l'utilisateur sait comment procéder).

CHANGER L'ENVIRONNEMENT DU PROJET

L'environnement du projet devrait être modifié si l'utilisateur désire changer en GLASSEX1.EXE le nom du fichier SPSM.exe associé au projet dans **Project: Setting: Link**.

Les nouveaux fichiers Adrv.cpp et Afamod.cpp doivent être inclus dans le projet (**Project: Add to project: Files**).

Le sous-répertoire clé \SPSM\DEFS devrait être ajouté dans **Tools: Options: Projects: VC++ Directories: Include files**, puisque les définitions touchant les applications en boîte de verre résident à cet endroit.

Le répertoire clé \SPSM\WIN32 devrait être ajouté sous **Tools: Options: Projects: VC++ Directories: Library files**, puisque les bibliothèques correspondants à la boîte de verre résident à cet endroit.

MODIFIER LA FONCTION DE PILOTE DE RECHANGE (ADRV.CPP)

Adrv.cpp contient deux genres d'information que l'utilisateur du mode boîte de verre désire modifier. Le premier genre comprend l'information d'étiquetage que le MSPS utilise dans ses rapports et ses messages d'erreur. Lorsque l'utilisateur fait les changements appropriés ici, la sortie obtenue fournit plus d'information. Le second genre comprend les appels de fonction qui font l'essentiel des calculs du système fiscal/de transfert du modèle.

L'utilisateur fait les changements d'étiquetage dans la partie du code qui ressemblent à ce qui suit et qui débute à la ligne 50 :

```
===== GLOBAL VARIABLE DEFINITIONS ===== */
/*global*/ char ALTNAME[IDSIZE+1] = "Unnamed";
/* Give global string describing version of this module */
/*global*/ char FAR Tdrv[] = "Untitled"
```

La chaîne **ALTNAME[IDSIZE+1]** fournit un nom d'identification pour le pilote de rechange; l'utilisateur remplace le paramètre fictif «Unnamed» par le nom plus significatif «FA Quick Start». Le nouveau nom ne doit pas compter plus que 20 caractères. Cet autre nom s'affichera à l'écran de bienvenue. La chaîne **Tdrv[]** donne un titre pour le pilote de rechange; l'utilisateur remplace le paramètre fictif «Untitled» par un titre plus significatif «FA Quick Start». Le nouveau titre ne doit pas avoir plus que 20 caractères. Le contenu de **TDrv** figure comme information dans le fichier de paramètres de commande comme description d'un algorithme. À la fin de ces substitutions, la «section étiquetage» révisée ressemble à ce qui suit :

```
/* ===== GLOBAL VARIABLE DEFINITIONS ===== */
/*global*/ char ALTNAME[IDSIZE+1] = "FA Quick Start";
/* Give global string describing version of this module */
/*global*/ char FAR Tdrv[] = "FA Quick Start"
```

Dans le corps du code, l'utilisateur n'a à faire qu'un changement pour indiquer que le

calcul des prestations, pour le système de variante, devrait utiliser un autre calcul pour les allocations familiales.

La partie pertinente du code, une seule ligne qui se trouve vers la ligne 125, s'affiche comme suit :

```
famod(hh); /* compute family allowances */
```

Si elle n'est pas modifiée, cette ligne invoque le calcul habituel des allocations familiales. L'utilisateur change la ligne de façon à invoquer plutôt l'autre calcul pour les allocations familiales, comme nous allons le décrire brièvement ci-dessous. Cette modification n'exige que la substitution du nouveau nom de fonction et le code source révisé est donné ci-dessous :

```
Afamod(hh); /* compute family allowances */
```

Pour cet exemple de départ rapide, ces trois changements simples constituent tout ce qu'il fallait faire pour la modification de la fonction `Adrv.cpp`.

MODIFIER LA FONCTION POUR LES ALLOCATIONS FAMILIALES (AFAMOD.CPP)

La fonction `Afamod.cpp` fait le calcul des allocations familiales pour l'autre système. Tout comme c'était le cas pour les changements du fichier `Adrv.cpp`, les changements de l'utilisateur sont de deux ordres, changement de l'étiquetage et changement du corps du programme.

Le changement de l'étiquette est tout ce qu'il y a de simple. À la ligne 54, la fonction donne un titre, `Tfa[]`, pour le module, le titre étant utilisé dans le rapport où le MSPS indique la fonction utilisée pour le calcul des impôts et des transferts. Comme avec le titre du pilote, cet titre figure comme description d'un algorithme dans le fichier de paramètres de commande. La partie pertinente du code est la suivante :

```
===== GLOBAL VARIABLE DEFINITIONS ===== */
/* Give global string describing version of this module */
/*global*/ char FAR Tfa[] = "Untitled"
```

Les changements apportés par l'utilisateur à la chaîne «Untitled» donnent un peu plus d'information. La section obtenue est donnée ci-dessous :

```
===== GLOBAL VARIABLE DEFINITIONS ===== */
/* Give global string describing version of this module */
/*global*/ char FAR Tfa[] = "FA Quick Start"
```

Le corps du changement apporté à `Afamod.cpp` est un peu plus compliqué, mais il n'est pas dramatique. L'option à examiner touche directement trois des variables calculées.

1. allocations familiales imposables (tfa),
2. allocations familiales fédérales (ffa) et
3. allocations familiales (fa).

(Bien sûr, d'autres variables du modèle, par exemple les impôts calculés, sont aussi touchés indirectement.) Lorsque le nombre d'enfants de la famille de recensement (la variable «nch») est de trois ou plus, nous désirons augmenter chacune des trois variables d'allocations familiales de 120 \$ multiplié par le nombre d'enfants «supplémentaires». Tout ce qui, par ailleurs, touche les répercussions de ce changement de politique, par exemple, les

répercussions sur l'impôt, sera pris en compte automatiquement par d'autres parties du MSPS. De toute façon, les variables dans le sous-programme sont provisoires et cessent d'exister lorsque l'exécution quitte la fonction Afamod; seuls les éléments qui ont été enregistrés dans les parties pertinentes de la structure du ménage pourront avoir une influence sur les calculs ailleurs dans le système.

Une fois bien claire la nature des changements désirés, la grande question qui se pose est l'endroit où il faut faire le changement dans la fonction Afamod.cpp. Pour des fins de clarté et de bonne logique, le changement devrait être apporté après que les trois variables aient déjà reçu les montants d'allocations familiales du «système de base», mais avant que tout calcul, comme l'affectation des montants aux variables à l'intérieur de la structure de données pour le ménage. Dans cet exemple les changements peuvent tous être faits en parallèle, au même endroit.

L'exemple n'est plus valide et sera révisé

La partie critique du code de source, tel qu'il existe avant la mise en œuvre de nos changements, ressemble à ce qui suit : (Les instructions de DEBUG illustrées ici n'ont pas trait au calcul habituel des prestations d'allocations familiales. Leur présence permet un suivi détaillé qui est fait au besoin, mais il n'est pas pertinent ici, sauf s'il indique la partie d'Afamod.cpp, vers la ligne 366, où les changements doivent être apportés aux allocations familiales.)

```
else {
    DEBUG1("%s standard FA calculation\n");
    tfa = nch * MP.STDFA;           /* taxable family allowances */
    ffa = tfa;                     /* federal part of family allowances */
}

DEBUG3("%s tfa=%.2f, ffa=%.2f\n", tfa, ffa);
```

En substance, nous désirons ajouter l'expression «(nch-2) * 120.0» à chacune des trois variables clés, allocations familiales imposables (tfa), allocations familiales fédérales (ffa), et allocations familiales (fa). En outre, ces augmentations ne conviennent que si le nombre d'enfants de 0 à 17 ans de la famille de recensement est d'au moins trois. L'instruction «if» du langage C et son opérateur «+=» fournissent une manière très commode de ce faire.

```
else {
    DEBUG1("%s standard FA calculation\n");
    tfa = nch * MP.STDFA;           /* taxable family allowances */
    ffa = tfa;                     /* federal part of family allowances */
    /* $120/yr bonus for 3rd and subsequent children <18 */
    if (nch >= 3) {
        tfa += (nch-2) * 120.0;
        ffa += (nch-2) * 120.0;
    }

    DEBUG3("%s tfa=%.2f, ffa=%.2f\n", tfa, ffa);
```

Quand les changements sont apportés à Afamod.cpp, le vrai travail que l'utilisateur a à faire est maintenant pratiquement terminé. Tous les changements pertinents à l'étiquette et au corps du code sont terminés et, si l'on suppose qu'il n'y a pas eu d'erreur pendant leur entrée, tout ce qu'il reste à faire c'est de compiler le nouveau modèle, puis de le valider. Plus

important encore, c'est le fichier exécutable obtenu à la compilation C++, dans cet exemple GLASSEX1.EXE, que l'utilisateur exécute pour analyser les répercussions du changement qui a été modélisé.

ESSAI DU MODÈLE MSPS OBTENU

Lorsque les changements ont été apportés, que le fichier résultant a été compilé et que les liens ont été établis de façon à créer le nouveau fichier exécutable, nous sommes prêts à faire l'essai du nouveau modèle. Les deux buts connexes de cette étapes sont :

1. chercher la preuve que nous avons fait la modification souhaitée, et
2. produire les sorties qui nous aideront à diagnostiquer les erreurs le cas échéant.

Une preuve explicite prend la forme de tableaux croisés d'une exécution comparative qui utilise le système fiscal/de transfert non modifié comme système de base et la forme modifiée comme système de variante. Plus loin, dans la présente section, nous donnerons des exemples de deux tableaux croisés de ce genre.

Afin de faire l'exécution comparative du nouveau modèle et d'obtenir la sortie dont nous avons besoin, nous devons modifier les paramètres de commande pour le modèle. Le *Guide des paramètres* donne une description complète des paramètres de commande du MSPS; ici, nous dressons une simple liste des valeurs des paramètres clés aux fins de notre exemple : (la partie «glassx1a» des deux noms de fichiers est un acronyme pour «Glass box example 1, version a (exemple boîte de verre 1, version a)».)

```
OUTCPR glassx1a.cpr # Name of control parameter file (out)
VARALG FA Quick Start # Name of variant algorithm
VARMETH 3 # Method of creating variant variables
BASMETH 2 # Method of creating base variables
OUTTBL glassx1a.tbl # Name of report file (out)
```

Deux tables suffiront pour la validation dans cet exemple :

1. totaliser le nombre de familles de recensement, les allocations familiales fédérales de variantes, les allocations familiales fédérales de base et l'écart qu'il y a entre elles, tout ça par nombre d'enfants de 0 à 17 ans (pour illustrer que nous donnons les nouvelles allocations familiales aux bonnes unités, et au bon montant) et
2. totaliser «delta allocations familiales» et «delta revenu imposable» par type de famille de recensement pour démontrer que nous donnons les nouvelles allocations familiales seulement aux bons genres d'unités ainsi qu'une partie de ces allocations est récupérée par le système fiscal, la fraction du montant récupéré étant plus élevée pour les familles comptant deux parents que pour les familles monoparentales.

Le paramètre XTSPEC servant à générer ces tables ressemble à ce qui suit :

```
XTSPEC
CF: cfnkids+ *
    {units,
     _imfa: L="Base Family Allowance (M)",
     imffa: L="New Family Allowance (M)",
     imffa-_imffa: L="Family Allowance Increase (M)",
```

```

(imffa-_imffa)/units: L="Average Family Allowance Increase"};
CF:  cftype+ *
      {imffa-_imffa: L="Family Allowance Increase (M)",
      immdisp-_immdisp: L="Disposable Income Increase (M)",
      (immdisp-_immdisp)/units: L="Mean Disposable Income Increase"};

```

Voici les points importants de cette demande :

1. La première table utilise "cfnkids" (nombre d'enfants de 0 à 17 ans) comme variable de commande de rang. Il faut noter que cfnkids est une variable de classe de la BDSPS, tandis que la variable «nch» utilisée ci-dessus dans les modifications apportées à l'intérieur du fichier Afamod.cpp est une variable locale qui est définie comme variable «à virgule flottante» et ne pourrait pas être utilisée ici pour les fins de totalisation, même si elle était une variable de classe.
2. Les variables totalisées utilisées dans la première table sont précisément celles qui sont décrites ci-dessus, les nombre de familles, les prestations d'allocations familiales nouvelles et anciennes et l'écart entre les deux.
3. La seconde table est simplement la totalisation, pour une autre variable de classe existante, des écarts qu'il y a dans les allocations familiales et dans le revenu imposable, avec la variable «à soulignement» référant au système de base et la variable sans soulignement référant au système de variante.

Les tables obtenues lorsque l'on exécute le nouveau modèle GLASSEX1, avec \SPSD\ba88t.cpr, sont données ci-dessous :

Table 1U: Selected Quantities for Census Families by Number of children in census family

Number of children in census family	Unit Count (000)	New Family Allowance (M)	Base Family Allowance (M)	Family Allowance Increase (M)
0	6401.6	0.0	0.0	0.0
1	1454.2	516.5	516.5	0.0
2	1430.7	1061.7	1061.7	0.0
3	612.9	850.0	776.5	73.5
4	111.9	229.5	202.6	26.8
5	36.8	83.7	70.4	13.3
6	5.3	28.4	25.8	2.5
7	0.0	0.0	0.0	0.0
8	0.0	0.0	0.0	0.0
9	0.0	0.0	0.0	0.0
All	10053.4	2769.8	2653.6	116.2

Table 2U: Selected Quantities for Census Families by Census family type

Census family type	Family Allowance Increase (M)	Disposable Income Increase (M)

With Kids, 1 Adult	16.7	15.6
With Kids, 2+ Adult	99.4	72.3
With Elderly, 1 Adult	0.0	0.0
With Elderly, 2+ Adult	0.0	0.0
Other, 1 Adult	0.0	0.0
Other, 2+ Adult	0.0	0.0
+-----+-----+		
All	116.2	87.9
+-----+-----+		

Les valeurs contenues dans les tables 1U et 2U résultent de l'exécution du nouveau modèle avec le sous-ensemble de 5 % de la BDSPPS en 1988 (ba88t.cpr) et de la demande des tables décrite ci-dessus. La première table confirme qu'il semble que nous donnions les allocations familiales supplémentaires aux bons genres de familles de recensement. Des prestations accrues, quelque 116 millions, sont indiquées seulement pour les familles de recensement ayant plus de deux enfants de 0 à 17 ans, et le montant brut est 120 \$ multiplié par le nombre d'enfants «supplémentaire» de ces familles.

La seconde table donne plus de preuves encore que les nouvelles prestations sont données seulement aux bons genres de famille de recensement et, en outre, que les nouvelles prestations sont partiellement récupérées par l'impôt. En outre, le niveau de récupération par l'impôt est plus bas pour les familles monoparentales que pour les familles qui comptent deux parents; on devait s'attendre à cette situation puisque (1) les personnes qui déclarent des prestations d'allocations familiales dans les familles où il y a deux parents tendent à avoir des revenus plus élevés et à être soumis à des taux marginaux d'impôt plus élevés, et (2) la Loi de l'impôt sur le revenu exige que le conjoint ayant le revenu net le plus élevé déclare les prestations d'allocations familiales.

Nous concluons à partir des valeurs contenues dans ces tables que les changements apportés ci-dessus ont fort probablement été réussis en regard de nos intentions.

L'essai tout juste décrit termine notre premier exemple de départ rapide. Du fait que l'orientation de l'exemple, nous n'avons peut-être pas été aussi soigneux et aussi méthodiques que nous aurions dû l'être si nous l'avions travaillé sur une application réelle. Par conséquent, nous mentionnons ici brièvement un certain nombre de choses que nous pourrions avoir choisi de faire pour la mise en œuvre de nos changements hypothétiques.

Nous pourrions avoir ajouté des commentaires de «l'histoire de la révision» dans les fichiers `Adrv.cpp` et `Afamod.cpp` afin de documenter la nature des changements que nous avons apportés et des raisons que nous avons pour faire la mise en œuvre de la façon dont nous nous y sommes pris. Cette forme de documentation est un élément de saine pratique professionnelle pour le développement et la maintenance du logiciel.

Nous pourrions avoir créé une variable à virgule flottante intermédiaire (locale) dans `Afamod.cpp` pour stocker l'augmentation des prestations d'allocations familiales de la famille. Cette augmentation, une fois calculée, aurait été affectée directement aux variables `tfa`, `ffa` et `fa` de façon que nous n'aurions pas calculé l'expression identique trois fois en parallèle. Si l'on fait exception de gains d'efficacité mineurs possibles, le code obtenu aurait probablement été légèrement plus facile à comprendre.

Nous pourrions avoir produit un paramètre de 10 \$ par mois (120 \$ par année), au cas où nous aurions désiré répéter l'analyse plus tard pour une valeur différente de la prestation d'allocations familiales supplémentaire. De même, nous aurions pu faire un paramètre du nombre d'enfants NON admissibles à la prestation supplémentaire; quelqu'un pourrait désirer connaître les répercussions du fait de restreindre les prestations supplémentaires aux familles de quatre enfants ou plus, ou de l'élargir de façon à accepter les familles ayant seulement deux enfants de 0 à 17 ans.

Nous aurions pu choisir de créer une nouvelle variable qui contiendrait seulement l'augmentation avant impôts pour la famille, de façon à rendre cette partie de la structure variable pour le ménage afin que nous puissions de façon plus commode totaliser cette variable d'«écart» dans des tableaux croisés, ou l'exporter pour analyse subséquente dans le SAS.

Nous aurions pu choisir de faire des tests plus ambitieux afin de vérifier que les changements souhaités ont été mis en œuvre. Par exemple, nous aurions pu produire une table donnant l'importance relative des changements dans les impôts sur le revenu du fédéral et du provincial afin de veiller à ce que les nouvelles prestations soient correctement prises en compte aux niveaux tant fédéral que provincial. Nous aurions pu totaliser l'ampleur du changement dans le crédit d'impôt pour enfants afin d'évaluer si les nouvelles prestations d'allocations familiales étaient correctement prises en compte dans la définition du revenu aux fins du crédit.

En général, la style de modification et le niveau de test effectués ici conviennent pour les buts restreints de ce premier exemple. Cependant, pour une application en boîte de verre plus sérieuse, l'utilisateur désirera probablement être plus méthodique quand il fera les changements nécessaires, accordant une plus grande attention aux questions de documentation, d'étiquetage, de validation et, éventuellement, d'efficacité des calculs.

RÉSUMÉ

Le chapitre a brossé une description provisoire des applications en boîte de verre dans le MSPS, illustrant ces applications d'un exemple spécifique. Les rubriques de section comprennent le changement des calculs du code dans une fonction d'allocations familiales de variantes, la modification de la fonction du pilote du MSPS qui coordonne le calcul des impôts et des transferts, et l'utilisation du compilateur C++ pour la création d'une nouvelle version du modèle. Une courte section sur la validation a illustré la production des tables qui ont permis d'évaluer la réussite du changement.

Le BD/MSPS et la structure du répertoire de la boîte de verre

Le présent chapitre donne à l'utilisateur de la boîte de verre une explication de la structure de répertoires du disque dur à l'intérieur de laquelle le MSPS fonctionne. L'information qu'on y trouve est pertinente puisqu'elle indique à l'utilisateur où se trouvent certains éléments, quels sont ceux qui ne doivent pas être touchés, quels sont ceux qui sont supposés servir comme gabarits pour les changements, quels sont ceux qui servent strictement comme exemple pour le code que l'utilisateur construira, etc.

Supposons comme suit la structure de répertoires du disque dur de l'utilisateur :

```
C:  [ Root directory ]
|--- MSC   [ Microsoft C compiler, with its own subdirectories ]
|--- SPSP  [ Data for the SPSP/M, with no subdirectories ]
|--- SPSM  [ SPSM proper, subdirectories as shown ]
|         |--- DEFS
|         |--- EXAMPLE
|         |--- GLASS
||        |--- MODEL
|         |--- WIN32
|--- GLASSEX1 [ Glass box task subdirectory 1 ]
|--- GLASSEX2 [ Glass box task subdirectory 2 ]
etc.
```

Au haut de la figure, nous voyons le répertoire racine de l'utilisateur, avec deux sous-répertoires de premier niveau MSC et SPSP. Le sous-répertoire MSC contient le compilateur de l'utilisateur, absolument nécessaire pour la création des applications en boîte de verre; MSC compte un certain nombre de sous-répertoires de niveau inférieur qui ne sont pas donnés ici. Le sous-répertoire SPSP contient toutes les données brutes du BD/MSPS ainsi qu'un certain nombre de fichiers de paramètres par défaut; il ne compte aucun sous-répertoire de niveau inférieur.

Le sous-répertoire SPSM et les sous-répertoires de niveau inférieur intéressent plus directement l'utilisateur du mode boîte de verre. Ces sous-répertoires ont été créés automatiquement pour l'utilisateur pendant l'extraction du MSPS; les noms utilisés sont les valeurs par défaut recommandées. Nous fournissons ici une brève description de chacun de ces répertoires, de leur contenu général et de leur pertinence pour les applications en boîte de verre.

Il faut d'abord faire ici un commentaire -- **L'UTILISATEUR NE DEVRAIT EN AUCUNE FAÇON MODIFIER QUOI QUE CE SOIT DES SOUS-RÉPERTOIRES DU MSPS.** (1) Les applications en boîte de verre impliqueront toujours le travail avec des **COPIES** de certains des fichiers qui se trouvent dans ces sous-répertoires. (2) Tout le travail en boîte de verre de l'utilisateur se fera dans l'un des **SOUS-RÉPERTOIRES DISTINCTS** qu'il aura créés pour contenir les fichiers de travail destinés aux applications en boîte de verre. Il peut même être utile pour l'utilisateur d'activer l'attribut «lecture seulement» pour tous les fichiers qu'il y a dans ces sous-répertoires.

DEFS Ce sous-répertoire contient un certain nombre de fichiers d'aide qui définissent les structures et les constantes utilisées partout dans le MSPS. Ce qui intéresse le plus l'utilisateur de la boîte de verre, ce sera le fichier `vs.h` qui définit la structure hiérarchique des données contenant l'information du BD/MSPS sur les ménages et les individus. Il faut cependant se rappeler que l'utilisateur n'aura jamais l'occasion de modifier cette structure. L'ajout par l'utilisateur de variables définies par l'utilisateur se fait dans une COPIE du fichier `vsu.h`.

EXEMPLE Ce sous-répertoire contient divers fichiers d'INCLUSION qui servent à spécifier les paramètres pour les exécutions d'exemple décrites dans la partie

didacticiel du document *Introduction et survol*. Bien qu'ils puissent être très utiles pour les fins de test de la réussite de l'installation du MSPS et pour l'apprentissage de l'utilisation des modèles qui ont déjà été développés, ces fichiers ne sont pas directement pertinents au développement des modèles en boîte de verre et peuvent être laissés de côté pour notre présente étude de la boîte de verre.

- GLASS** Ce sous-répertoire contient les gabarits que l'utilisateur utilisera comme point de départ pour le code qu'il rédigera afin de créer des modèles et des systèmes fiscaux/de transfert de variante. (1) Il contient le code source pour toutes les fonctions fiscales et de prestation du MSPS; l'utilisateur trouvera probablement plus efficace de créer toute nouvelle fonction en modifiant une COPIE de ces éléments. (2) Il contient des fonctions qui rendent les variables et les paramètres définis par l'utilisateur accessibles au MSPS en général, avec les fichiers d'en-tête connexes qui définissent les structures pertinentes pour contenir les paramètres et les variables définis par l'utilisateur.
- MODEL** Ce sous-répertoire contient des exemples de définition de paramètres et de variables modélisés. Les éléments qu'il y a dans ce sous-répertoire visent SEULEMENT à servir d'exemple concret pour l'utilisateur lorsqu'il commence à définir de nouvelles variables et de nouveaux paramètres pour des applications en boîte de verre. L'utilisateur n'aura jamais l'occasion de modifier le contenu de ces fichiers, ni même d'utiliser ou de modifier des copies de ces fichiers.
- WIN32** Ce sous-répertoire contient un petit nombre de «fichiers d'exécution de commande» de Windows 32 bits qui régissent la forme de structure de recouvrement que le MSPS utilise. À un niveau très général, ces éléments sont semblables à ceux qu'il y a dans LIB dans le sens que SPSMGL.dsw a besoin d'eux et sait comment les utiliser pour la compilation d'une nouvelle version du modèle. Il contient aussi certains fichiers exécutables utilisés pour la modification de la BDSPPS dans un projet.

Tout au bas de la structure de répertoires du disque dur de l'utilisateur, il y a un sous-répertoire de «travail» GLASSEX1 qui contient les applications en boîte de verre et deux sous-répertoires WINREL et WINDEBUG. L'utilisateur peut avoir autant de sous-répertoires de travail qu'il lui en faut pour les applications en boîte de verre qu'il construit. Celui-ci correspond à l'exemple de départ rapide décrit au chapitre 2. Il contient tous les fichiers que l'utilisateur crée quand il fait le double de cet exemple. Les fichiers en question sont les suivants :

ADRV.CPP
AFAMOD.CPP
FAQSTST1.CPR
FAQSTST1.TBL
SPSMGL.SLN
SPSMGL.VCPROJ
SPSMGL.NCB
SPSMGL.OPT
SPSMGL.PLG
GLASSEX1.EXE
GLASSEX1.PDB
WINREL

ADRV.CPP et AFAMOD.CPP sont les fichiers de code source en langage C++ copiés du sous-répertoire GLASS puis modifiés de façon à refléter la logique que l'on souhaitait dans le nouveau programme; leurs équivalents OBJ sont les fichiers objets produits à la sortie lorsque les fichiers ".C" sont compilés dans WINDEBUG et WINREL. GLASSEX1.EXE et GLASSEX1.pdb ont été créés par l'instruction Compile. Enfin, FAQSTST1.CPR est le fichier de paramètres de commande pour les exécutions du programme FAQSTST1, et FAQSTST1.TBL contient les tableaux croisés que l'exécution correspondante de SPSMFAQS a produits.

L'information essentielle de ce chapitre peut se résumer à ce qui suit :

1. **L'utilisateur du MSPS ne devrait en aucune façon changer quoi que ce soit dans le sous-répertoire SPSM ou dans l'un ou l'autre de ces sous-répertoires créés pendant l'installation du MSPS.** (Il faut noter cependant que certains fichiers qui peuvent être définitivement inutiles peuvent être supprimés complètement.)
2. L'utilisateur de la boîte de verre crée des sous-répertoires «de travail» distincts pour les applications en boîte de verre. Il est préférable que ces sous-répertoires ne soient pas des sous-répertoires de SPSM.
3. **L'utilisateur de la boîte de verre copie les éléments pertinents du répertoire SPSM\GLASS, qu'il utilise comme gabarits pour les modifications à apporter.** Les modifications elles-mêmes sont alors faites dans ces COPIES. Les sections suivantes du *Guide de programmation* indiquent très en détail ce que l'utilisateur doit modifier et où les gabarits pertinents se trouvent.
4. Les sous-répertoires clés \SPSM\DEFS devraient être ajoutés dans **Tools: Options: Directory**, puisque les définitions pertinentes aux applications en boîte de verre s'y trouvent.

La structure Individus/ménages de la BDSPS

Le présent chapitre vise trois grands buts, chacun étant élaboré dans une section distincte, mais ils ont tous trait au grand sujet de la structure des données du BD/MSPS et de leur utilisation.

La section suivante donne un aperçu du cadre de travail du MSPS pour le stockage des données sur les ménages, les familles et les individus. Il est essentiel que l'utilisateur de la boîte de verre connaisse bien cette structure puisqu'il cherche à y référer ou à y modifier des valeurs de variables modélisées et de variables de données existantes ainsi qu'à créer de nouvelles variables de ce genre au besoin pour une version adaptée du MSPS.

La deuxième section donne des explications sur l'utilisation des variables pointeurs comme outil majeur permettant à l'utilisateur d'accéder à des éléments particuliers des données. Elle décrit aussi les grandes règles d'affectation des noms utilisées pour les applications en boîte

de verre. Ces sujets sont pertinents tant pour l'utilisateur qui construit ses propres applications en boîte de verre que pour les personnes qui cherchent à comprendre les algorithmes standard du MSPS. La «philosophie» sous-jacente de ce développement va dans le sens du reste du présent guide - sous de nombreux aspects, il est beaucoup plus important pour l'utilisateur de la boîte de verre de savoir comment, mécaniquement, faire quelque chose de façon normalisée et robuste que de comprendre toutes les raisons découlant des concepts de structures et de techniques. En d'autres mots, l'orientation de la présente section est résolument pratique; elle se concentre beaucoup plus sur la mécanique du «comment faire» que sur les détails du «pourquoi».

La troisième section fournit un «bestiaire» des fragments de codes à utiliser pour exécuter les tâches courantes en boîte de verre, particulièrement en ce qui a trait aux structures de données. Le principe n'est pas seulement que l'utilisateur devrait pouvoir reproduire une roue existante plutôt que de la réinventer, mais aussi que la roue copiée devrait exister en outre sous une forme normalisée et ne pas exiger de mise au point. Les fragments de codes de cette section comprennent a) le traitement des familles et des individus pertinents par des instructions «for», b) la référence à d'autres membres de la famille, c) l'accès à la base de données et aux variables modélisées existantes ainsi que d) l'affectation de nouvelles valeurs à ces variables.

LA STRUCTURE DES DONNÉES DU BD/MSPS

La BDSPS est un fichier dont l'ordre est fixé. Le contenu ne peut pas être trié par l'utilisateur. Il est essentiel de comprendre l'ordre de tri de la base de données lorsque l'on tente de faire des boucles dans les ménages. La base de données regroupe les ménages en grappes qui sont triés au hasard de façon stratifiée. Chaque ménage est alors trié comme suit :

Ménage

Familles économiques

Familles de recensement

Familles nucléaires

Chefs de famille

Conjoint, le cas échéant

Enfants, des plus jeunes aux plus vieux

À l'intérieur d'un ménage, les individus sont regroupés en familles économiques. À l'intérieur d'une famille économique, les individus sont regroupés en familles de recensement. À l'intérieur de la famille de recensement, les individus sont regroupés en familles nucléaires. À l'intérieur de la famille nucléaire, le chef est toujours en premier et il est suivi du conjoint, le cas échéant. Les enfants suivent alors et sont triés selon leur âge.

Un ménage complet est chargé dans la structure de données spécifiées ci-dessus. Les boucles

peuvent alors être établies afin de permettre le traitement de l'une ou l'autre des unités d'analyse à l'intérieur d'un ménage.

Des descriptions détaillées de la substance des variables individuelles elles-mêmes du BD/MSPS se trouvent dans le *Guide des variables*. Une bonne partie des détails touchant le contenu de nombreuses structures se trouve dans *vs.h*. Les éléments clés nécessaires à la définition des variables se trouvent dans *spsm.h*. Certaines des macros permettent à l'utilisateur de faire des choses de façon symbolique afin de rendre leur signification plus claire, ou pour la constance dans la précision numérique :

```
#define LOGICAL int          /* type used to store true or false values      */
#define TRUE 1              /* manifest constants to make code more readable */
#define FALSE 0
#define NUMBER float
#define ZERO (float) 0.0
#define HALF (float) 0.5
#define ONE (float) 1.0
#define THOUSAND (float) 1000.0
#define MILLION (float) 1000000.0
```

PRÉSENTATION DES POINTEURS DANS LE BD/MSPS

La structure *uv* est une structure dont le contenu est défini par l'utilisateur, tant pour sa substance que pour les noms de variables. Un chapitre décrit la façon dont l'utilisateur crée de nouvelles variables, par exemple pour définir un nouveau programme fiscal/de transfert. L'utilisateur détermine le contenu de «uv» par le fichier d'en-tête *vsu.h*, et le fichier *vsdu.c*, mais peut modifier les valeurs des éléments définis eux-mêmes partout à l'intérieur du fichier *Adrv.cpp*. Ces capacités d'attribution et de définition constituent l'essence même des applications en boîte de verre lorsque l'utilisateur a besoin d'ajouter de nouvelles variables. Bien sûr, l'utilisateur doit prendre soin de donner les nouvelles variables et les nouveaux impôts aux bons individus de façon que le cumul fonctionne correctement dans le reste du MSPS.

Le langage C utilise beaucoup les variables pointeurs, c'est-à-dire des variables qui pointent vers un secteur particulier de la mémoire et particulièrement à une structure de données en particulier. Bien que les parties du code de source du MSPS qui ont trait aux algorithmes fiscaux/de transfert fassent une utilisation moins intense des pointeurs et de la mathématique des pointeurs que les parties fermées à l'utilisateur, l'utilisateur de la boîte de verre aura quand même à utiliser les pointeurs. Même si l'utilisation des pointeurs est essentielle, la conception du MSPS a rendu cette utilisation aussi simple que cela était possible pour les concepteurs. Divers fragments de codes et macros sont fournis de façon à rendre l'utilisation des pointeurs aussi simple et aussi mécanique que cela était possible. La section bestiaire décrit brièvement la façon dont ces pointeurs sont appliqués pour des tâches courantes en boîte de verre comme les boucles et les références. Il faut cependant noter que cette section ne vise aucunement à constituer un cours exhaustif sur l'utilisation plus générale des pointeurs à l'extérieur du MSPS.

LE BESTIAIRE

Un bestiaire est une «collection de descriptions d'animaux réels ou imaginaires». Les «animaux» recueillis et décrits ici sont réels. Il y a des fragments du code de source en langage C qui sont susceptibles d'être utiles à l'utilisateur de la boîte de verre pendant la lecture et l'écriture de codes pour les programmes fiscaux/de transfert. Les fragments de codes décrits ici sont tous compris dans le fichier `BESTIARY.C` de façon que l'utilisateur puisse copier les segments sans avoir à les retaper.

Les éléments du bestiaires sont fournis conformément à la philosophie soutenue tout au long du présent guide. Plus précisément, l'utilisateur ne devrait pas avoir à réinventer la roue, mais devrait recevoir toute l'aide possible en tirant profit de choses qui existent déjà à l'intérieur du MSPS. Comme il peut copier le code existant, en le modifiant peut-être où moment de la copie, il en tire quatre grands avantages :

1. Le code de source existant est sans aucun doute correct, et, par conséquent, il n'a pas besoin de mise au point.
2. Il y aura plus grande uniformité entre les codes de l'utilisateur et ceux qui sont livrés dans le MSPS.
3. La copie est beaucoup plus rapide que la nouvelle entrée du code.
4. L'utilisateur peut souvent faire le travail nécessaire sans avoir à comprendre tous les détails sous-jacents. En général, il s'agira d'une en-tête suivie du code lui-même et, parfois, d'une courte explication ou d'un bref commentaire.

Exemples de boucles

Une des tâches les plus courantes dans la lecture, la modification ou la rédaction de codes est l'itération à l'intérieur des unités pertinentes d'un ménage ou de l'une de ses structures. Le jeu suivant de segments de codes est probablement ce qu'il y est de plus complet en ce qui a trait à l'itération requise par l'utilisateur. Il faut noter que les segments du code de source comprennent les définitions pertinentes requises. Par exemple, dans le premier exemple ci-dessous, l'utilisateur doit déclarer le pointeur «in» du type «P_in,» et le nombre entier, «ini» de façon qu'ils puissent être utilisés dans le fonctionnement de la boucle (itération). En pratique, les définitions figurent dans le code de source avant la boucle elle-même.

```
/** * PROCESS ALL INDIVIDUALS IN HOUSEHOLD hh */

register P_in in;
int ini;

for (ini=0, in=&hh->in[0]; ini<hh->hhnin; ini++, in++) {
    DEBUG2("%s processing individual %d in household\n", ini);
    /* code here, using pointer 'in' */
}
```

Dans la boucle ci-dessus, et dans les autres qui suivent, l'instruction «for» du langage C est utilisée. Les éléments qui précèdent le point virgule initial initialisent les variables pour l'itération. La condition qu'il y a entre les deux points-virgules précise quand la boucle doit se poursuivre. Les éléments qui se trouvent à l'intérieur des parenthèses, mais après le deuxième point-virgule, spécifient l'incrémentation nécessaire pour l'itération suivante.

Aussi dans le fragment de codes, il y a un commentaire «code here». Il indique l'endroit où le code du MSPS, ou encore le code de l'utilisateur, devrait aller pour agir sur l'unité à l'intérieur de laquelle se produit le cycle d'itération. Le commentaire «code here» identifie aussi cette unité en ce qui a trait au pointeur que commande la boucle.

```

/**** PROCESS ALL INDIVIDUALS IN ECONOMIC FAMILY ef      **/

    register P_in in;
    int ini;      for (ini=0, in=ef->efin; ini<ef->efnpers; ini++, in++) {
    DEBUG2("%s processing individual %d in economic family\n", ini);
    /* code here, using pointer 'in' */
    }

/**** PROCESS ALL INDIVIDUALS IN CENSUS FAMILY cf      **/

    register P_in in;
    int ini;
    for (ini=0, in=cf->cfin; ini<cf->cfnpers; ini++, in++) {
    DEBUG2("%s processing individual %d in census family\n", ini);
    /* code here, using pointer 'in' */
    }

/**** PROCESS ALL CHILDREN (including 18+) IN CENSUS FAMILY cf      **/

    register P_in in;
    int ini;
    for (ini=0, in=cf->cfinch; ini<cf->cfnchild; ini++, in++) {
    DEBUG2("%s processing child (including 18+) %d in census family\n", ini);
    /* code here, using pointer 'in' */
    }

/**** PROCESS YOUNG CHILDREN IN CENSUS FAMILY cf      **/

    register P_in in;
    int ini;
    for (ini=0, in=cf->cfinch; ini<cf->cfnkids; ini++, in++) {
    DEBUG2("%s processing child (<18) %d in census family\n", ini);
    /* code here, using pointer 'in' */
    }

/**** PROCESS ALL INDIVIDUALS IN NUCLEAR FAMILY nf      **/

    register P_in in;
    int ini;
    for (ini=0, in=nf->nfin; ini<nf->nfnpers; ini++, in++) {
    DEBUG2("%s processing individual %d in nuclear family\n", ini);
    /* code here, using pointer 'in' */
    }

/**** PROCESS CHILDREN IN NUCLEAR FAMILY nf      **/

    register P_in in;
    int ini;
    for (ini=0, in=nf->nfinch; ini<nf->nfnkids; ini++, in++) {
    DEBUG2("%s processing child %d in nuclear family\n", ini);
    /* code here, using pointer 'in' */
    }

/**** PROCESS ALL ECONOMIC FAMILIES IN HOUSEHOLD hh      **/

```

```

P_ef ef;
int efi;
for (efi=0, ef=&hh->ef[0]; efi<hh->hhnef; efi++, ef++) {
DEBUG2("%s processing economic family %d\n", efi);
/* code here, using pointer 'ef' */

}

/** PROCESS ALL CENSUS FAMILIES IN HOUSEHOLD hh      **/

P_cf cf;
int cfi;
for (cfi=0, cf=&hh->cf[0]; cfi<hh->hhncf; cfi++, cf++) {
DEBUG2("%s processing census family %d\n", cfi);
/* code here, using pointer 'cf' */

}

/** PROCESS ALL NUCLEAR FAMILIES IN HOUSEHOLD hh      **/

P_nf nf;
int nfi;
for (nfi=0, nf=&hh->nf[0]; nfi<hh->hhnnf; nfi++, nf++) {
DEBUG2("%s processing nuclear family %d\n", nfi);
/* code here, using pointer 'nf' */

}

```

Références relatives à un individu

Une autre tâche courante en boîte de verre implique la référence à d'autres individus d'une structure ou d'une sous-structure, ou à des unités d'analyse «plus élevées» dans la structure. C'est à travers ces références que l'utilisateur peut référer aux caractéristiques comme la province de résidence d'un individu, le revenu du conjoint du membre le plus âgé de la famille de recensement (s'il y a un conjoint), ou l'âge du deuxième enfant, dans l'ordre de l'âge, vivant au sein d'une famille de recensement, à l'intérieur d'une famille économique commune.

```

/** REFERENCE SPOUSE OF INDIVIDUAL in      **/

if (in->id.idspoflg) {
P_in inspo;
inspo = in->id.idinspo;
/* code here, using pointer 'inspo' */
}

```

Notice here that there will not always exist a spouse.

```

/** REFERENCE HOUSEHOLD OF INDIVIDUAL in      **/

P_hh hh;
hh = in->id.idhh;
/* code here, using pointer 'hh' */

```

Quand il a récupéré le pointeur vers le ménage, l'utilisateur a accès aux caractéristiques du ménage, comme la province de résidence. Contrairement à la situation avec le conjoint d'un individu, le ménage existera toujours.

```

/** REFERENCE ECONOMIC FAMILY OF INDIVIDUAL in      **/
P_ef ef;

```

```
ef = in->id.idef;
/* code here, using pointer 'ef' */
```

De même, la famille économique de l'individu existera toujours et sera pertinente pour l'évaluation de l'endroit où l'individu vit dans une unité en dessous du seuil de pauvreté.

```
/** REFERENCE CENSUS FAMILY OF INDIVIDUAL in */
P_cf cf;
cf = in->id.idcf;
/* code here, using pointer 'cf' */
/** REFERENCE NUCLEAR FAMILY OF INDIVIDUAL in */
P_nf nf;
nf = in->id.idnf;
/* code here, using pointer 'nf' */
```

Ces références clés, couplées aux fragments d'itération de la section précédente, permettent à l'utilisateur de faire, de façon relativement commode, à peu près tout ce qui est susceptible d'être nécessaire pour la simulation d'un système fiscal/de transfert.

RÉSUMÉ

La première partie du chapitre décrivait la structure des données utilisées dans le BD/MSPS. Cette partie indique aussi les macros de fonction et les constantes manifestes les plus importantes que l'utilisateur pourra trouver dans le code de source du MSPS. Les parties suivantes décrivaient le rôle des variables pointeurs dans le MSPS et établissaient les caractéristiques des principaux types de pointeurs utilisés. Cette partie se termine par un bestiaire des fragments de code utilisés pour les tâches les plus courantes en mode boîte de verre, l'itération dans les unités d'individus et de familles et la référence au conjoint d'un individu ou aux unités d'analyse qui le contiennent.

Le chapitre suivant érige sur cette base en décrivant la façon dont le MSPS traite les ménages quand il s'agit de calculer les impôts et les transferts. Cette description sert ensuite de base aux derniers chapitres qui indiquent la façon dont on ajoute des variables et des paramètres définis par l'utilisateur dans le cours de la modification de la logique du système fiscal/de transfert.

Structure d'appel des fonctions du MSPS

Le calcul des impôts et des transferts d'argent pour un ménage est commandé par une fonction dont la seule tâche consiste à appeler toutes les autres fonctions individuelles d'algorithmes fiscaux/de transfert. L'ordre des appels est critique pour la simulation étant donné les besoins en information des fonctions fiscales/de transfert. Par exemple, il faut connaître le revenu net avant que le SRG ne puisse être calculé. La liste suivante donne les fonctions appelées par `drv` et par `adrv`, dans l'ordre où ils sont appelés.

Fonction	Description
ui(hh)	Calcul des prestations d'assurance-chômage
famod(hh)	Calcul des allocations familiales
oas(hh)	Calcul de la sécurité de vieillesse
dem(hh)	Calcul des nouvelles subventions démographiques

txinet(hh)	Calcul du revenu net
gis(hh)	Calcul du supplément de revenu garanti pour les personnes âgées
gist(hh)	Calcul du supplément provincial de revenu pour les personnes âgées
samod(hh)	Calcul de l'aide sociale
txitax(hh)	Calcul du revenu imposable
txhstr(hh)	Calcul des déductions pour enfants et conjoint
txcalc(hh)	Calcul de l'impôt fédéral
txctc(hh)	Calcul du crédit d'impôt pour enfants
txfstc(hh)	Calcul du crédit fédéral sur la taxe de vente
txprov(hh)	Calcul des crédits des impôts provinciaux
gai(hh)	Calcul de nouveaux crédits remboursables, de nouvelles garanties
memo1(hh)	Calcul du revenu disponible, etc.
ctmod(hh)	Calcul des taxes à la consommation et affectation aux personnes
memo2(hh)	Calcul du revenu consommable, etc.
cceopt(hh, drv)	Valeur zéro pour les frais de garde d'enfants pour les jeunes enfants, si cela se révèle optimal
classu(hh)	Calcul des variables d'établissement de rapports définies par l'utilisateur (dans /glassbox)

L'ordre d'appel des fonctions d'éléments de `drv` reflète l'ordre logique entre eux.

- Les premières fonctions, `ui`, `famod` et `oas`, simulent des programmes dont les prestations sont déterminées par des facteurs autres que le revenu et, par conséquent, elles sont appelées en premier.
- `dem` est un sous-programme pour les applications boîte de verre qui exigent que des calculs se produisent avant l'entrée dans les programmes de système fiscal.
- `txinet` calcule le revenu net avant certains transferts.
- `gis` calcule les transferts aux personnes âgées.
- `samod` calcule l'aide sociale ou les transferts de revenus garantis.
- Les impôts fédéraux et provinciaux sont calculés ensuite, dans les quatre fonctions suivantes, avec le préfixe `tx` (`txitax`, `txhstr`, `txcalc` et `txprov`).
- `gist`, `txctc` et `txfstc` calculent les programmes de transfert en fonction du revenu.
- `gai` est un autre sous-programme qui est destiné aux utilisateurs de la boîte de verre qui désirent simuler des options pour lesquelles il faut de l'information sur tous les impôts sur le revenu et tous les transferts d'argent personnels. Par exemple, l'utilisateur peut utiliser cette fonction pour simuler un programme de supplément de revenu.
- Les fonctions `memo1` et `memo2` créent des variables agrégées pour les fins de rapport.
- Dans la fonction `ctmod`, la taxe d'accise et la taxe de vente sont calculées par l'application des taux actuels de taxe de vente basés sur les entrées et les sorties de dépenses des familles observées.
- `cceopt` optimise le revenu en optimisant le crédit pour frais de garde d'enfants et le crédit pour enfants.
- `classu` est un sous-programme qui permet à l'utilisateur du mode boîte de verre de calculer et d'affecter les valeurs à des variables nouvelles ou redéfinies.

Les fonctions appelées par `drv` appellent d'autres fonctions et sous-fonctions afin qu'elles

exécutent leurs calculs. La page suivante contient une liste complète des noms des fonctions et des sous-fonctions avec une courte description, dans l'ordre où elles sont appelées par `drv`. Veuillez consulter le *Guide des algorithmes* à la fonction voulue pour obtenir une description plus détaillée. Les sous-fonctions peuvent se retrouver dans la liste qu'il y a sous la fonction qui les appellent. Par conséquent, pour une compréhension complète du calcul du revenu net, il faudrait consulter les deux fonctions `txinet` et `txcea`.

Les noms de fonction sont imprimés en caractères minuscules et gras, en courrier (p. ex., `txinet`, `txcalc`) et correspondent à un seul fichier de code de source en langage C (p. ex., `tixnet.cpp`, `txcalc.cpp`). Les sous-fonctions sont définies à l'intérieur de la fonction (fichier) qui les appellent et sont données en courrier minuscule (p. ex., `uisqz`, `gissub`). L'exemple suivant est un appel à une sous-fonction `uiclm()` dans `ui.cpp` où `uiclm` est défini dans une section de `ui.cpp`.

```
valid_claim = uiclm(in, &in->id.ucl, in->id.ucl.ucyl, &in-  
>im.ubl,  
                  hh->hd.hdprov, hh->hd.hdurb, wctb);
```

Développement en boîte de verre : ajout de paramètres scalaires ordinaires

Comme son titre le laisse entendre, ce chapitre explique à l'utilisateur de la boîte de verre la mécanique des tâches de programmation associées à l'ajout de paramètres scalaires ordinaires pendant le développement d'applications en boîte de verre. Sur le plan de la structure, ce chapitre donne l'information dans un exemple pratique détaillé. La première section revoit la procédure générale utilisée pour le développement d'applications en boîte de verre, décrivant les étapes qui sont à la base de toute modification de modèle, que ce soit par la modification de codes, ou encore par l'ajout de paramètres ou de variables. La deuxième section poursuit avec plusieurs mesures préliminaires à prendre avant l'ajout de paramètres. Elle décrit aussi la nature de l'exemple à utiliser, et il s'agit d'un prolongement de l'exemple de supplément d'allocations familiales utilisé dans le chapitre de Départ rapide du Guide. Les autres sections utilisent ensuite l'exemple pour expliquer en détail les étapes de l'ajout des genres les plus courants de paramètres scalaires à un modèle. Enfin, la dernière section résume les points essentiels relativement à l'ajout de ces paramètres des formes les plus courantes à un modèle.

PROCÉDURE GÉNÉRALE DE MODIFICATION EN BOÎTE DE VERRE : UNE RÉCAPITULATION

La section précédente a déjà décrit la procédure générale de développement d'applications en boîte de verre, ce qui comprend la logique qui régit ces étapes. Nous résumons ici les points clés de façon condensée.

- Créer un sous-répertoire de travail
- Déterminer les fichiers à modifier

- Copier les fichiers pertinents dans le sous-répertoire de travail
- Modifier les fichiers pertinents
- Compiler la nouvelle version
- Tester la nouvelle version du modèle
- Faire les analyses prévues

Créer un sous-répertoire de travail

L'utilisateur crée un nouveau «sous-répertoire de travail» pour recevoir les fichiers pertinents à la nouvelle application en boîte de verre. Il modifie les fichiers qu'il y a dans le sous-répertoire de travail en ne touchant à aucun autre fichier du BD/MSPS.

Déterminer les fichiers à modifier

L'utilisateur détermine les fichiers qu'il y a dans `c:\spsm\glass` pour lesquels des variantes devront être créées. Par exemple, dans l'exemple de Départ rapide, nous avons déterminé qu'il s'agissait des fichiers `Afamod.cpp`, `Adrv.cpp` et `SPSMGL.dsw`. L'exemple donné dans ce chapitre indique comment d'autres fichiers, par exemple `Mpu.h` et `Ampd.cpp`, ont aussi trait à l'ajout de paramètres à une application en boîte de verre. Une section expliquera comment d'autres fichiers encore, `Vsu.h` et `Vsdu.cpp`, peuvent intéresser l'utilisateur qui désire ajouter des variables à un modèle. De toute évidence, les fichiers de fonction fiscale/de transfert qui utilisent les nouveaux paramètres doivent aussi être modifiés. L'utilisateur peut parfois trouver plus efficace d'utiliser comme gabarit le fichier déjà développé dans une application antérieure plutôt que de revenir aux fichiers de gabarit du sous-répertoire glass.

Copier les fichiers pertinents vers un sous-répertoire de travail

L'utilisateur copie tous les fichiers désignés comme pertinents vers le sous-répertoire de travail. L'utilisateur travaillera seulement avec des copies, sans toucher les originaux.

Modifier les fichiers pertinents

L'utilisateur apporte les modifications appropriées dans chacun des fichiers indiqués comme pertinents. Nous recommandons que les changements soient apportés dans l'ordre suivant :

1. Inclure tous les fichiers pertinents dans le projet et changer le nom du fichier de sortie dans Project: Setting: Link.
2. Modifier le fichier `Adrv.cpp`, si nécessaire.
3. Modifier les fichiers `Mpu.h` et `Ampd.cpp`, le cas échéant, pour ajouter de nouveaux paramètres au modèle.
4. Modifier le fichier `Vsu.h` et `Vsdu.cpp`, le cas échéant, pour ajouter toute nouvelle variable de sortie au modèle.

5. Modifier les fichiers du code de source pour ajouter la nouvelle logique souhaitée au code du système fiscal/de transfert.

Nous suivrons cet ordre dans les exemples que nous présentons ici et dans les sections suivantes.

Compiler la nouvelle version

L'utilisateur devrait activer la fonction Debugging dans Build: Set Active Configuration puis faire la mise au point du projet. Lorsque le programme a été corrigé correctement, le nouveau modèle devrait être compilé.

Tester la nouvelle version du modèle

L'utilisateur teste la nouvelle version par un ensemble d'analyses de validation conçu de façon à révéler tout problème qu'il peut y avoir dans la logique qui a été ajoutée ou modifiée. Cette étape peut exiger le retour à une version antérieure si l'on veut corriger les lacunes qui sont décelées.

Effectuer l'analyse prévue

Enfin, quand la validation est complète, l'utilisateur peut procéder à des «exécutions de production» du nouveau code exécutable afin de simuler les conséquences du changement qui a été modélisé.

PRÉSENTATION DE L'AJOUT DES PARAMÈTRES

La présente section poursuit avec quelques éléments préliminaires critiques pour la procédure d'ajout de paramètres scalaires habituels. En premier lieu, elle illustre la raison pour laquelle l'utilisateur peut désirer ajouter un ou plusieurs paramètres à un modèle. En outre, elle décrit la substance des nouveaux paramètres que nous utilisons pour illustrer l'ajout de paramètres ordinaires.

Comme on l'a noté à la fin de l'exemple de Départ rapide, notre analyste hypothétique a pris quelques raccourcis pour faire des choses qui pourraient être faites différemment dans un exercice d'élaboration de politiques du monde réel, particulièrement si le nouveau modèle visait à être utilisé de façon répétitive ou par de nombreux analystes. Un de ces raccourcis consistait à «implanter» l'augmentation de 120 \$ par année aux allocations familiales directement dans la fonction `Afamod.cpp`. Certes, cela aurait été acceptable si l'utilisateur n'aurait jamais voulu tenter d'autres valeurs d'augmentation, mais cela n'est pas particulièrement efficient si l'on était intéressé à évaluer les répercussions d'autres valeurs. L'utilisateur devrait modifier de nouveau le code puis recompiler le modèle pour chaque valeur distincte dont il voudrait faire l'examen; l'utilisateur pourrait, par exemple, chercher à vérifier s'il est vrai que les répercussions sont habituellement proportionnelles au montant de l'augmentation et pourraient désirer tenter plusieurs valeurs pour mener à bien son enquête. Avec des paramètres appropriés ajoutés au modèle, aucune modification supplémentaire n'est nécessaire et l'utilisateur peut étudier les effets de plusieurs valeurs, sans devoir recompiler le modèle, simplement en fournissant de nouvelles valeurs de paramètres au modèle modifié.

Par conséquent, les diverses sections du présent chapitre décrivent les étapes nécessaires à l'ajout de paramètres au modèle, en nettoyant l'exemple de Départ rapide par une illustration en particulier. Ce chapitre se limite aux formes les plus couramment utilisées de paramètres scalaires. Nous croyons que les genres d'ajouts décrits combleront peut-être 80 % des besoins d'ajouts de paramètres que peut éprouver l'utilisateur de la boîte de verre. Nous gardons pour les dernières sections la définition plus ésotérique de paramètre scalaire ainsi que des vecteurs et matrices de paramètres. Quel que soit le genre des nouveaux paramètres, une fois qu'ils sont ajoutés à un modèle, ils sont accessibles pour toutes les fonctions appelées par `Adrv.cpp`; elles ne sont pas limitées à la fonction pour un seul programme de transfert unique.

Essentiellement, nous ajouterons trois paramètres à une variante du modèle utilisé dans le Départ rapide. Les trois ajouts correspondent aux trois formes les plus courantes de paramètres que l'utilisateur de la boîte de verre aura l'occasion d'utiliser.

1. Le premier paramètre, une valeur scalaire «à virgule flottante» ou «réelle», fournira la valeur de l'augmentation d'allocations familiales remise au titre de certains enfants; il éliminera la valeur de 120 \$ implantée directement. Nous appellerons ce paramètre `fasuppc` (supplément d'allocations familiales par enfant).
2. Le deuxième paramètre, une valeur entière scalaire, indiquera le nombre d'enfants à partir duquel le supplément commence à être versé; il élimine la valeur «3» implantée directement dans l'exemple de Départ rapide. Nous appellerons ce paramètre `fasupfec` (premier enfant admissible aux allocations familiales).
3. Le troisième paramètre, une variable de «drapeau» qui est effectivement un commutateur booléen, indiquera si l'on doit porter attention aux deux premiers paramètres. À ce chapitre, sa fonction est parallèle à celle de nombreuses variables de drapeau utilisées partout à l'intérieur du MSPS. Lorsqu'il est «activé», il permet le calcul du supplément; lorsqu'il est «désactivé», le modèle calcule les allocations familiales, mais ne tient pas compte du supplément. Nous appellerons ce paramètre `fasupflg` (drapeau de supplément d'allocations familiales).

Notre description suppose que l'utilisateur a choisi d'utiliser `\glassex2` comme répertoire de travail, en le créant au besoin.

COPIER LES FICHIERS `ADRV.CPP`, `MPU.H`, `AMPD.CPP`, `AFAMOD.CPP`, `SPSMGL.DSW`

L'utilisateur copie dans le nouveau sous-répertoire de travail tous les fichiers qui doivent être modifiés. De même, l'utilisateur qui désire modifier `Adrv.cpp` pour mettre à jour la description utilisée dans les fichiers de code (ici, seulement `Afamod.cpp`) qui sont modifiés. Par conséquent, il faut copier `Adrv.cpp`.

Deux autres fichiers, `Mpu.h` et `Ampd.cpp`, sont toujours pertinents lorsque l'utilisateur désire ajouter un nouveau paramètre de modèle. `Mpu.h` (paramètres de modèle, utilisateur) est un fichier d'en-tête de langage C qui définit la nature du nouveau paramètre, `Ampd.cpp` (autre définition de paramètre de modèle) contient les appels de fonctions qui font connaître

les paramètres de l'utilisateur au reste du MSPS de façon qu'ils puissent, par exemple, être l'objet de références par leur nom aux fins de changement de valeurs «à la volée» lorsque l'utilisateur exécute le fichier exécutable du MSPS.

L'utilisateur doit copier ses fichiers `Mpu.h` et `Ampd.cpp` depuis le sous-répertoire `glass` ou quelque source équivalente. Si, par exemple, l'utilisateur a déjà, quelque part, modifié ces fichiers afin de définir d'autres paramètres et désire conserver les modifications déjà faites, il peut copier les gabarits pour `Mpu.h` et `Ampd.cpp` depuis le sous-répertoire où ils existent déjà. Quand nous parlons de «gabarits», nous parlons de fichiers existants, ou d'éléments de texte ou de code, qui servent de point de départ commode pour l'exécution de modifications souhaitées. Ainsi, cela n'aurait pas de sens du tout si l'utilisateur devait entrer, à partir de rien, une version entièrement nouvelle des fichiers pertinents. Dans cet exemple, nous supposons que ce sont les premiers paramètres ajoutés et nous copierons les gabarits du répertoire `glass`.

Enfin, bien sûr, l'utilisateur doit copier les fonctions fiscales/de transfert essentielles ou les conditions qui utiliseront le nouveau paramètre. Pour nos fins, la seule fonction essentiellement pertinente est la fonction `Afamod.cpp`. Plutôt que de la copier du répertoire `glass` puis de devoir par la suite la reprendre au début, nous la copierons de `glassex1` de façon que notre travail soit en partie fini; il suffira de trouver l'endroit où il faudrait faire l'affectation de l'augmentation.

L'utilisateur devra copier SPSMGL.dsw qui décrit l'environnement du projet.

MISE À JOUR DU PROJET

Tous les fichiers nécessaires devraient être inclus dans le projet et le nom du fichier exécutable à la sortie devrait être modifié dans Project: Setting: Link to `glassex2.exe`.

MISE À JOUR DE LA DESCRIPTION DE L'ALGORITHME DANS ADRV.CPP

Rappelez-vous que, de l'exemple de Départ rapide, les variables globales `altname[]` et `Tdrv[]` ont reçu de nouvelles valeurs qui servent à indiquer et à documenter la nature du changement apporté. Ici, avec une nouvelle version du modèle que l'on crée, une substitution correspondante est nécessaire. Les deux substitutions, qui consistent exclusivement dans le contenu de deux chaînes, produisent le code suivant :

```
===== GLOBAL VARIABLE DEFINITIONS ===== */
/*global*/ char ALTNAME[IDSIZE+1] = "Parameterized FA Supplement";
/* Give global string describing version of this module */
/*global*/ char FAR Tdrv[] = "Parameterized FA Supplement"
```

À ce moment, nous pouvons faire la compilation de mise au point afin de vérifier nos modifications. Un contrôle comme celui-là aide l'utilisateur à trouver des erreurs de syntaxe pendant que la nature de la modification est encore fraîche à sa mémoire. Pour ce faire, il suffit de sélectionner le projet `Win32Debug` dans Project: Set Active Project et de lancer Build:Start Debug. Si la compilation et les liens sont nécessaires, C++ vous le fera savoir.

MODIFIER MPU.H AFIN DE DÉFINIR LES NOUVEAUX PARAMÈTRES

L'utilisateur a ensuite besoin de modifier le fichier `Mpu.h` afin de définir le type des

nouveaux paramètres. Lorsque la modification est apportée dans la version boîte de verre de Mpu.h, la ligne qui contient la chaîne "UMDUMMY" est remplacée par la définition des nouveaux paramètres. Le nom "UMDUMMY" renvoie à «paramètre de modèle utilisateur fictif». Nous appelons le premier paramètre nouveau FASUPPC pour indiquer qu'il s'agit du montant du supplément d'allocations familiales par enfant admissible. Avant le changement, la ligne indiquée (vers la ligne 62) ressemble à ce qui suit :

```
int UMDUMMY;          /* dummy entry                                */
```

Parce que, comme l'indique l'étiquette, cette entrée est seulement un paramètre fictif, de façon que le MSPS ait quelque chose à manipuler si l'utilisateur n'a pas défini de paramètres utilisateur, nous supprimons complètement cette ligne. Nous la remplaçons par les lignes :

```
NUMBER FASUPPC;      /* Family Allowance Supplement per Child */
int     FASUPFEC;     /* FA Supplement, First Eligible Child */
int     FASUPFLAG;    /* FA Supplement, Activation Flag      */
```

À la première ligne, «NUMBER» est une macro utilisée par le MSPS pour assurer la portabilité d'un ordinateur à l'autre; elle correspond au type «à virgule flottante». FASUPPC est le nom du nouveau paramètre. La règle du MSPS est que ces noms de paramètres sont mis en majuscules. Les deux autres paramètres sont des chiffres entiers, naturellement. Pour des fins de lisibilité, nous avons aussi ajouté des commentaires à la droite afin d'indiquer la nature des valeurs de paramètre.

Ces simples ajouts mettent fin à notre modification de Mpu.h. Habituellement, si nous ajoutons de nouveaux paramètres à un ensemble qui contiendrait des paramètres utilisateurs déjà en place, nous n'ajouterions que les nouvelles définitions au bas de la liste existante dans Mpu.h, tout comme le paramètres FASUPFEC et FASUPFLAG suivent ici le paramètre FASUPPC.

Le MSPS attribue de l'espace pour au plus 500 nouveaux paramètres, suffisamment pour les applications habituelles de l'utilisateur du mode boîte de verre. Il est possible même d'avoir plus de paramètres si certains d'entre eux sont du type «int», plus petit. **Toute tentative de dépasser cette limite entraînera un message d'erreur à la compilation qui fera ressortir le problème.**

MODIFIER AMPD.CPP DE FAÇON À RENDRE LES PARAMÈTRES DISPONIBLES AU MSPS

L'utilisateur a aussi besoin de modifier le fichier Ampd.cpp pour rendre le nouveau paramètre visible partout à l'intérieur des parties du MSPS qui peuvent avoir besoin d'y référer. Le MSPS fournit une fonction «pmaddent» (module de paramètres, ajout d'une entrée) pour effectuer cette tâche. L'utilisateur appelle cette fonction une fois pour chaque nouveau paramètre, juste avant l'instruction "DEBUG_OFF (Ampd)", près de la fin du fichier Ampd.cpp, vers la ligne 138.

Si l'utilisateur travaille avec une copie du fichier Ampd.cpp qui contient déjà l'appel de pmaddent pour d'autres paramètres, ces autres appels peuvent être utilisés comme gabarits. Dans notre exemple, cependant, puisqu'il n'y a, pour l'instant, aucun autre paramètre ajouté, nous copions le gabarit pmaddent du fichier C:\SPSM\MODEL\Mpd1.cpp (fichier 1 de

définition de paramètres de modèle). Pour notre premier paramètre, FASUPPC, nous reconnaissons que ce paramètre du type NOMBRE devrait être très semblable au paramètre STDFA qui se trouve vers la ligne 252. Nous copions simplement cet appel de pmaddent et faisons les substitutions appropriées. Cette pratique, qui consiste à copier quelque chose à peu près semblable qui existe déjà et fonctionne, puis à le modifier, est une pratique standard en développement en boîte de verre. Cet appel, tel qu'il est copié, ressemble à ce qui suit :

```
pmaddent(pcp, "STDFA", (char *)&MP.STDFA, NULL, P_SCL, C_NUM, 0, 0, NULL, 0);
```

Nous la modifions pour nos fins en changeant les deux références à STDFA de façon que la ligne renvoie à notre nouveau paramètre. Le remplacement de «STDFA» par «FASUP» et de «(char *)&MP.STDFA" by "(char *)&MP.UM.FASUPPC», parce que le nouveau paramètre est un élément de la structure UM (modèle utilisateur) qui se trouve dans la structure MP (paramètres de modèle), nous donne le résultat suivant :

```
pmaddent(pcp, "FASUPPC", (char *)&MP.UM.FASUPPC, NULL, P_SCL, C_NUM, 0, 0, NULL, 0);
```

Pour l'instant, nous conservons tous les autres arguments de la fonction sans nous soucier de ce qu'ils représentent. Dans la mesure où nous avons choisi un gabarit approprié pour nos emprunts, il n'y a pas de problème. Plus tard, nous étudierons la signification de chacun des arguments de pmaddent de façon à pouvoir juger plus facilement des sources appropriées de gabarits pmaddent, et à les récupérer plus efficacement à la suite de tout choix inapproprié.

Nous choisissons UIWAITWKS (le nombre entier de semaines dans la période de carence de l'assurance-chômage) comme gabarit pour nos paramètres entiers précisant la «position» du premier enfant de la famille auquel le supplément sera accordé. De même, nous choisissons un paramètre de drapeau existant qui servira de gabarit pour notre drapeau de nouveau supplément d'allocations familiales; FAFLAG, qui détermine si les allocations familiales doivent être calculées, semble un bon choix. Avant nos modifications, ces deux appels ressemblent à ce qui suit :

```
pmaddent(pcp, "UIWAITWKS", (char *)&MP.UIWAITWKS, NULL, P_SCL, C_INT, 0, 0, NULL, 0);
pmaddent(pcp, "FAFLAG", (char *)&MP.FAFLAG, NULL, P_SCL, C_INT, E_FLAG, 0, NULL, 0);
```

Comme avec le paramètre FASPPC ci-dessus, nous modifions chacun de ces gabarits à deux endroits, remplaçant le nom du paramètre et sa relation avec la structure MP. Les appels de pmaddent modifiés ressemblent à ce qui suit :

```
pmaddent(pcp, "FASUPFEC", (char *)&MP.UM.FASUPFEC, NULL, P_SCL, C_INT, 0, 0, NULL, 0);
pmaddent(pcp, "FASUPFLAG", (char *)&MP.UM.FASUPFLAG, NULL, P_SCL, C_INT, E_FLAG, 0, NULL, 0);
```

Ces simples ajouts terminent la modification du fichier Ampd.cpp pour ce qui est de rendre les VALEURS des nouveaux paramètres disponibles partout dans le MSPS, du moins une fois que nous aurons attribué ces valeurs d'une façon quelconque. Plus tard dans la présente section, nous réglons le cas de certains des mécanismes qui permettent à l'utilisateur de faire les affectations. Cependant, il nous faut encore donner des étiquettes claires pour les paramètres de façon que le MSPS puisse les utiliser pour donner une documentation significative des paramètres de modèle, le cas échéant.

Encore une fois, la conception du MSPS facilite le travail. Il y a une fonction toute faite,

`stradd`, pour faire la mise en œuvre de l'étiquetage. Juste après les instructions `pmaddent`, nous insérons trois lignes pour appeler cette fonction, `stradd` --

```
stradd("FASUPPC", "Family Allowance Supplement per Child");
stradd("FASUPFEC", "FA Supplement, First Child Payable");
stradd("FASUPFLAG", "FA Supplement, Activation Flag");
```

La fonction `stradd` (ajout de chaîne), lorsqu'elle est exécutée, «lie» la chaîne de descripteur aux paramètres de façon que le descripteur se retrouve automatiquement dans toute l'étiquetage et toute la documentation pertinente du MSPS. Étant donné que les arguments de fonction (**`stradd`**) sont aussi simples, c'est-à-dire qu'il y a une chaîne identifiant le nom d'un nouveau paramètre, et une seconde chaîne fournissant la description associée, nous n'avons pas besoin de recourir à un gabarit.

L'élément final de cette étape, la compilation partielle de la fonction `Ampd.cpp`, est facultatif, mais nous le recommandons car il peut mener à un développement ordonné des applications en boîte de verre. Ce type de compilation partielle permet à l'utilisateur de demander au compilateur de vérifier les erreurs de syntaxe, tandis que la nature de la modification est encore fraîche à notre esprit. L'opération ne vérifie pas que le code de source modifié correspond au reste au MSPS. Il faut noter que l'on doit avoir modifié tous les fichiers d'en-tête pertinents, ici le fichier d'en-tête `Mpu.h`, d'abord pour que la compilation avec `Debug` fonctionne.

MODIFIER LES FONCTIONS QUI UTILISENT LE NOUVEAU PARAMÈTRE

Pour terminer la modification de la programmation qui est nécessaire à l'ajout de paramètres, nous devons modifier la fonction `Afamod.cpp` de façon à lui faire utiliser les nouveaux paramètres symboliques plutôt que les valeurs implantées directement dans l'exemple de Départ rapide. Nous commençons pas modifier l'étiquette définie pour cette fonction; plus précisément, nous modifions le code définissant l'étiquette de façon qu'elle se lise comme suit :

```
/*global*/ char FAR Tfa[] = "Afamod.cpp Parameterized"
```

Cette étiquette étant fournie, le MSPS peut l'utiliser chaque fois qu'il a l'occasion d'utiliser la description de fonction dans sa documentation.

Essentiellement, les modifications à apporter à la fonction `Afamod.cpp` sont faciles à faire.

Où l'exemple de Départ rapide utilisait «120.0», nous entrons la représentation symbolique "MP.UM.FASUPPC". Les règles d'affectation des noms, exactement identiques à celles qui sont utilisées pour l'appel de la fonction «`pmaddent`» dans la modification apportée ci-dessus à l'`Ampd.cpp`, reflètent l'emplacement de FASUP dans la sous-structure UM (modèle utilisateur) de la structure MP (paramètre de modèle) que le MSPS utilise pour stocker tous les paramètres de modèle.

Où l'exemple de Départ rapide utilisait 3 pour représenter le nombre d'enfants requis dans la

famille pour le versement du supplément, nous insérons MP.UM.FASUPFEC. Toutes les formules pertinentes sont ajustées en conséquence.

Nous faisons le calcul du supplément et nous l'additionnons aux variables `fa`, `tfa` et `ffa` selon la valeur de la nouvelle variable de drapeau, `fasupflag`.

Donc, le code de source de l'exemple de Départ rapide qui ressemblait à ce qui suit :

```
/* $120/yr bonus for 3rd and subsequent children <18 */
if (nch >= 3) {
  tfa += (nch-2) * 120.0;
  ffa += (nch-2) * 120.0;
}
```

devient, dans la réincarnation de `glassex2` :

```
/* Conditionally add a Family Allowance bonus for the
"FASUPFECth" and subsequent children <18 in the unit */
if ((MP.UM.FASUPFLAG == 1) & (nch >= MP.UM.FASUPFEC)) {
  tfa += (nch-MP.UM.FASUPFEC+1) * MP.UM.FASUPPC;
  ffa += (nch-MP.UM.FASUPFEC+1) * MP.UM.FASUPPC;
}
```

La logique sous-jacente demeure inchangée, mais elle est maintenant spécifiée par l'intermédiaire de paramètres. En outre, nous avons modifié le commentaire de façon à indiquer la généralisation des paramètres symboliques. En rédigeant le code de source de cette façon, nous avons supposé que l'utilisateur du modèle fournira seulement des valeurs raisonnables pour les paramètres. Par exemple, nous croyons ici qu'aucun utilisateur ne fournira pas par inadvertance une valeur zéro (0) pour MP.UM.FASUPFEC et ne créera pas intentionnellement un supplément d'allocations familiales pour les familles n'ayant pas d'enfants âgés de 0 à 17 ans. Plus tard, nous montrerons comment l'utilisateur peut se servir des fonctions de vérification-correction du MSPS pour faire en sorte que les valeurs des paramètres soient raisonnables.

Encore une fois, nous exécutons une compilation avec Debug pour rattraper les erreurs de syntaxe avant la compilation du nouveau modèle.

VALIDER ET FAIRE DES EXÉCUTIONS DE PRODUCTION EN BOÎTE NOIRE

Comme avec l'exemple de Départ rapide, nous avons encore besoin de vérifier si nous obtenons des résultats raisonnables. Comme l'exécution du MSPS ne coûte pratiquement rien et n'exige pas tellement de temps, deux exécutions de validation particulières s'imposent immédiatement.

1. La première exécution se fait avec le paramètre FASUPPC à la valeur zéro, utilisant les mêmes tables qui ont été générées dans l'exemple de Départ rapide. Pour cette exécution, nous donnons la valeur 3 au paramètre FASUPFEC et la valeur 1 au paramètre FASUPFLAG. Nous nous attendons à ce qu'il n'y ait aucune différence entre les systèmes de base et de variante parce que la valeur zéro pour le paramètre fait en sorte que le changement est nul.
2. Nous modifions le premier test de façon à fournir une valeur de 120.0 pour le paramètre FASUPPC, laissant les paramètres FASUPFEC et FASUPFLAG aux valeurs 3 et 1

respectivement. Encore une fois, nous demandons à la sortie les tables du Départ rapide afin d'observer les mêmes résultats que nous avons obtenu à l'origine avec l'exemple Départ rapide quand la valeur 120.0 était implantée directement.

3. Nous modifions FASUPFEC de façon à lui donner la valeur 2, en nous attendant à ce que cela augmente considérablement le coût de l'option hypothétique, puisqu'il y a relativement beaucoup de familles comptant deux enfants. Les tables spécifiques nous permettent de vérifier facilement, du moins pour le montant brut du supplément, si les montants de supplément appropriés ont été calculés pour chacun des types de famille, par nombre d'enfants.
4. Enfin, nous ajoutons un quatrième test pour désactiver le supplément par le paramètre FASUPFLAG. En faisant ce test de validation, nous laissons les paramètres FASUPPC et FASUPFEC à 120.0 et 2 respectivement de façon que nous puissions nous assurer que tout effet est causé par la désactivation du paramètre de drapeau à zéro. Comme dans le cas de la première exécution de validation décrite ci-dessus, nous nous attendons à ce qu'il n'y ait aucune différence entre les allocations familiales de base et les allocations familiales en option, le calcul du supplément ayant été supprimé.

Pour effectuer les tests de validation, il ne reste qu'à affecter les valeurs désirées aux nouveaux paramètres. La conception du MSPS facilite cette opération. Si nous exécutons simplement le nouveau modèle sans nous préoccuper de préciser la valeur de paramètre nécessaire, le MSPS note l'omission et nous permet de fournir «à la volée» la valeur par la fonction de modification de paramètres. Ou, pour obtenir une fonction équivalente, nous pourrions avoir placé une entrée appropriée dans le fichier MPR (paramètres de modèle), puisque ces fichiers contiennent habituellement les paramètres de modèle, que les paramètres soient définis par l'utilisateur ou qu'ils soient intégrés dans le MSPS livré. De même, le nouveau fichier de paramètres pourrait avoir été spécifié dans un fichier MPI (inclusion de paramètres de modèle). La description précise de ces deux dernières méthodes se trouve dans le Guide d'utilisation.

En faisant les tests ci-dessus, nous voyons que notre changement, l'ajout des trois nouveaux paramètres, a été fait correctement étant donné que tous les ensembles de sorties sont ceux qui étaient prévus. Les résultats du troisième test, quand nous déplaçons la valeur de FASUPFEC (premier enfant admissible) sont particulièrement marqués. Nous pouvons à cet endroit vérifier que les montants appropriés de prestations supplémentaires sont ajoutés aux familles selon le nombre d'enfants de 0 à 17 ans. Maintenant que les changements apportés au modèle sont validés, nous pouvons faire l'ensemble pertinent d'exécutions de production. Par exemple, un client pourrait nous demander d'utiliser la valeur 60.0 pour le paramètre FASUPPC afin de vérifier ses attentes selon lesquelles le nombre de familles touchées serait le même qu'avec la valeur de 120.0 et que les coûts, agrégés et moyens par famille touchée, seraient seulement la moitié de ceux qu'il y aurait eu avec la valeur 120.0. De même, nous pouvons substituer une valeur beaucoup plus grande, disons 5000.0, pour vérifier notre attente selon laquelle, avec un transfert de cette envergure, la partie du supplément récupérée par le système fiscal s'accroîtrait passablement puisque certaines familles se retrouveraient à des taux d'imposition supérieurs.

RÉSUMÉ/CONCLUSION

Il convient de conclure en soulignant, mais sans toutefois revenir sur le sujet, les points clés concernant l'ajout de paramètres scalaires ordinaires à un modèle. En notant ces points, on suppose que l'analyste travaille avec des COPIES des fichiers pertinents et qu'il exécute toutes les modifications dans le sous-répertoire de travail consacré à l'analyse en cours. Nous supposons aussi que l'utilisateur a fait la mise à jour du projet de façon à inclure tous les fichiers de code de source pertinents. Du point de vue technique, nous supposons que l'utilisateur récupérera souvent des éléments de codes existants semblables comme gabarit et qu'il les modifiera au besoin.

1. Modifier le fichier d'en-tête `Mpu.h`, en ajoutant une instruction pour chaque paramètre nouveau. L'instruction indique le nom du paramètre et son type, NUMBER étant utilisé pour les valeurs à virgule flottante.
2. Modifier le fichier de code de source `Ampd.cpp` en ajoutant deux instructions pour chaque paramètre nouveau.
 - Ajouter un appel de "pmaddent" pour chaque paramètre de façon que le MSPS puisse rendre sa valeur disponible à toutes les fonctions appelées par `Adrv.cpp`. La pratique normale consiste à copier l'appel à partir d'un appel existant puis à le modifier à deux endroits -- le nom du paramètre et son adresse.
 - Ajouter un appel de `stradd` pour chaque paramètre de façon que le MSPS lie l'étiquette du paramètre à ce nouveau paramètre.
3. Modifier les fonctions essentielles pertinentes de façon à utiliser les nouveaux paramètres en modifiant l'étiquette ainsi que la logique interne de la fonction.
4. Mettre au point et compiler le nouveau programme. Faire les «exécutions de production» nécessaires avec le modèle, puis interpréter les résultats.

Développement en boîte de verre : ajout de paramètres inhabituels

Le présent chapitre décrit plus en détail les arguments de la fonction `pmaddent` et l'utilisation de cette fonction lorsque l'utilisateur ajoute des paramètres matriciels, vectoriels et scalaires à des applications en boîte de verre. Pour ce faire, il érige sur la base établie à la section précédente (Ajout des paramètres scalaires ordinaires), en élaborant de nouvelles considérations pour des paramètres scalaires inhabituels, pour des tables vectorielles et de recherche ainsi que pour des matrices. Enfin, la dernière section résume les points essentiels en ce qui a trait à l'ajout de ces paramètres moins courants à un modèle.

La première section du chapitre présente l'ensemble des arguments pour la fonction clé `pmaddent`, en décrivant les principales caractéristiques de chacun d'eux. La section suivante présente ensuite une liste des types de paramètres scalaires que l'utilisateur peut désirer ajouter. Pour chaque type, elle indique brièvement l'objet du type en particulier, décrivant les arguments clés de `pmaddent` pour ce type et indiquant un gabarit `pmaddent` approprié à

utiliser lorsque l'on crée un paramètre de ce type. Suivent des sections qui donnent des considérations spéciales ayant trait à l'ajout de vecteurs de paramètres, puis de paramètres de recherche et de matrices de paramètre.

PMADDENT : LA FONCTION ET SES ARGUMENTS

Rappelez-vous que, dans la section portant sur la description de l'ajout de paramètres ordinaires, l'aspect le plus compliqué de l'ajout de nouveaux paramètres à un modèle réside dans la modification du fichier `Ampd.cpp`, puisque les changements à `Mpu.h` consistent en des définitions très simples de types de paramètres. Parmi les changements à apporter au fichier `Ampd.cpp`, le seul grand défi, et aucunement un défi particulièrement difficile à relever, provient de l'appel de la fonction `pmaddent`. Nous avons noté que l'utilisateur de la boîte de verre peut habituellement contourner la complexité de cette fonction en choisissant simplement un gabarit d'appel «approprié», copié d'un paramètre «suffisamment similaire» déjà défini. Dans cette section, nous expliquons plus en détail le sens des divers arguments de `pmaddent`, de façon que l'utilisateur de la boîte de verre puisse utiliser `pmaddent` en toute confiance, même s'il n'a pas de gabarit évident à copier et à modifier.

Notre point de départ pour la description des arguments de `pmaddent` est le commentaire explicatif qui se trouve dans le fichier `Ampd.c` lui-même (vers la ligne 150 de la version BOÎTE DE VERRE). Nous prendrons chacun des dix arguments dans l'ordre. Nous soulignons cependant que l'utilisateur devrait n'avoir que très peu d'occasions d'avoir à utiliser cette information. La plupart du temps, le paramètre à ajouter sera bien compris et pourra utiliser un gabarit de paramètre à peu près semblable qui sera facile à identifier. Dans tous ces cas, l'utilisateur devrait simplement modifier le gabarit pertinent et poursuivre la modélisation, en laissant les dédales de `pmaddent` aux personnes qui exécutent des tâches inhabituelles.

Voici un résumé des arguments de `pmaddent` dans `Ampd.cpp`.

```
/**
 * pmaddent(
 *   pcp,                <= parameter chain being extended (leave as is)
 *   "XXXXX",            <= name by which the parameter will be known
 *   (char *)&MP.UM.XXXXX, <= address of the parameter
 *   Format,              <= printing information for the parameter
 *   Agg_Type,            <= Aggregate type (scalar, vector, etc.)
 *   C_Type,              <= C-type (integer, number, string)
 *   Edit,                <= Edits to be performed
 *   Row_max,             <= Maximum number of rows, or option edit limit.
 *   Rows_addr,           <= Address of int holding current number of rows
 *   Limit <= Number of columns *
 **)
```

Le premier argument (`pcp`) est particulièrement simple : l'utilisateur entre TOUJOURS la variable `pcp`. L'argument identifie la chaîne de paramètres spécifique que l'utilisateur prolonge. Bien que le MSPS emploie d'autres chaînes de paramètres dans ses opérations, l'utilisateur peut ajouter des paramètres SEULEMENT à la chaîne `pcp`.

Le deuxième argument, représenté par la chaîne fictive "XXXXX" dans le commentaire, est le nom de l'utilisateur du paramètre. Le nom sera le même que l'utilisateur a employé par la définition `Mpu.h`. L'utilisateur devrait prendre soin de choisir des mnémoniques

raisonnables pour ces noms, par exemple FASUPFLAG, le nom que nous avons utilisé précédemment. La règle du MSPS est que ces noms devraient commencer par une lettre majuscule et contenir seulement des majuscules et des chiffres.

Le troisième argument, représenté par la chaîne fictive (char *)&MP.UM.XXXXXX est l'adresse du paramètre. La partie initiale («cast» dans le langage C) de l'argument, «(char *)», est invariable. De même, la partie «MP.UM» est invariable parce que les paramètres de l'utilisateur sont toujours ajoutés à la structure «paramètre de modèle, modèle utilisateur». La partie 'XXXXXX' représente le nom du paramètre utilisateur; elle prend la valeur de la chaîne utilisée comme deuxième argument, sans les guillemets. Enfin, conformément à la manière dont le langage C traite les adresses de variables, la perluète (&) est présente si le paramètre est scalaire et elle est habituellement absente s'il ne l'est pas (c.-à-d. qu'elle est absente si le paramètre est un vecteur, un paramètre de recherche de table ou une matrice). Ce dispositif courant du langage C de référence spécifique au premier élément d'un tableau est repris plus loin, sous une rubrique particulière. Aux fins spéciales du paramètre «FICTIF», décrit ci-dessous, ce troisième argument prend la valeur «NULL».

Le quatrième argument, représenté dans la description ci-dessus par «Format», est une chaîne. Il contient l'information sur la façon dont le MSPS devrait donner la valeur du paramètre lorsqu'il en fait la documentation. Habituellement, l'utilisateur utilise le format prédéfini «NULL» pour indiquer que le MSPS doit imprimer le paramètre de la façon qui lui semble correcte. Un autre format prédéfini, «F_FRACT», contient la chaîne «8.5» et convient particulièrement pour l'impression de la valeur d'une fraction. L'utilisateur peut aussi entrer une chaîne explicite pour l'argument; par exemple, l'utilisation de «8.0» précise que la valeur devrait occuper huit caractères et qu'elle ne devrait pas comprendre de partie fractionnaire. Un argument «7.2» spécifierait une chaîne de sept caractères, avec deux chiffres après la virgule décimale. Lorsque cela convient, par exemple pour les paramètres du style Recherche de table, l'argument peut comprendre de multiples indicateurs de format, par exemple «8.0 8.2 8.2». Le format prédéfini F_LKTUR, utilisé pour les paramètres de type P_LKPXY, fournit un exemple concret de cette utilisation.

Le cinquième argument, représenté dans la description ci-dessus par «Agg_Type», indique le type de paramètre. Cet argument reflète un choix forcé entre les six valeurs entières de 0 à 5. Chacune des six valeurs a une contrepartie mnémonique que l'utilisateur peut employer, pour des fins de clarté, à la place de la valeur numérique elle-même. Les six valeurs, leur contrepartie mnémonique, et leur interprétation, sont comme suit :

La valeur 0, représentée par la mnémonique P_SCL, est la valeur la plus courante. Elle est utilisée pour un paramètre qui a une valeur scalaire (entier, flottant, fraction, etc.).

La valeur 1, représentée par la mnémonique P_VCT, est utilisée lorsque le paramètre est un vecteur. D'autres renseignements clés au sujet du vecteur, par exemple le nombre d'éléments qu'il contient, sont donnés par d'autres arguments de pmaddent.

Les valeurs 2 et 3, représentées par les mnémoniques P_LKPXY et P_LKPSL, sont utilisées à l'intérieur du MSPS pour deux genres spéciaux de tableaux dans lesquels les recherches sont effectuées, une en format X-Y et l'autre en format plage-pente. Dans le cas où

l'utilisateur désire créer des paramètres de ces types, les paramètres GISST et FTX fournissent des exemples opérationnels. Ces deux types de paramètres définissent des tableaux qui correspondent aux fonctions LKUP1 et LKUP2 respectivement; les fonctions LKUP1 et LKUP2 elles-mêmes sont documentées dans le Guide des algorithmes. L'utilisation de tableaux dans le MSPS est documentée plus en détail dans ce chapitre. La valeur 4, représentée par la mnémonique P_TBL, est utilisée lorsque le paramètre est une matrice bidimensionnelle (table). D'autres renseignements clés sur la matrice, par exemple, le nombre de rangs et de colonnes, sont donnés dans d'autres arguments de pmaddent. La matrice des taxes à la consommation, CTTXRM, fournit un bon exemple.

La valeur 5, représentée par la mnémonique P_DUMMY, ne sera habituellement pas utilisée par l'utilisateur de la boîte de verre. Ce type de paramètre correspond à une entrée fictive utilisée pour retenir un nom d'une chaîne d'en-tête pour des fins de documentation.

Le sixième argument, représenté dans la description ci-dessus par «C_Type», indique le type de paramètre. Il y a trois entrées possibles pour cet argument. La valeur C_INT convient lorsque la valeur du paramètre est de façon inhérente un chiffre entier, c.-à-d. qu'il consiste en un nombre sans partie fractionnaire, et qu'il a une valeur se situant entre les valeurs des chiffres entiers à l'intérieur des limites du langage C. L'utilisateur emploie une valeur de C_INT pour cet argument lorsque l'entrée dans Mpu.h pour le paramètre utilisé dans une déclaration «int». Les paramètres qui sont des «drapeaux» ou des «options» sont habituellement des chiffres entiers.

La valeur C_NUM convient quand la valeur du paramètre peut avoir une partie fractionnaire ou quand elle est trop grande pour être stockée comme nombre entier. L'utilisateur utilise une valeur de C_NUM pour cet argument lorsque l'entrée du paramètre dans Mpu.h utilisait une déclaration «NUMBER».

La valeur C_STR est utilisée lorsque la valeur de paramètre est une entrée fictive utilisée pour une chaîne d'en-tête. L'utilisateur de la boîte de verre ne donne habituellement pas l'occasion d'utiliser C_STR.

Le septième argument, représenté dans la description ci-dessus par «edit», indique que les fonctions de vérification seront imposées sur la valeur du paramètre. L'activation du contrôle de vérification forcera la valeur du paramètre à respecter les diverses contraintes qui peuvent être appropriées. En outre, il peut restreindre la capacité de l'utilisateur de modifier les valeurs de paramètre au moment de l'exécution par la fonction de modification des paramètres du MSPS. L'argument de pmaddent régissant ces contrôles de vérification est une valeur de nombre entier. Habituellement, l'utilisateur choisit une valeur en entrant un élément d'un ensemble de valeurs mnémoniques prédéfinies (décrites ci-dessous).

Voici les codes et leur interprétation :

E_NONE (valeur 0) indique qu'il n'y a aucune contrôle de vérification exécuté sur ce paramètre.

E_FIXL (valeur 1) s'applique seulement lorsque le paramètre est un vecteur, une table de

recherche ou un tableau (et, par conséquent, a un nombre maximum connu de rangs). Le code de vérification empêche l'utilisateur de tenter de modifier le nombre réel de rangs de la valeur maximale. La mnémonique ici indique que la limite de nombre de rangs est considérée comme fixe.

E_FLAG (valeur 2) indique que le paramètre est un drapeau. Selon les règles du MSPS, cela signifie que le paramètre est traité comme une variable binaire (définie comme chiffre entier) qui doit prendre soit la valeur 0 (zéro), soit la valeur 1 (un).

E_FRCT (valeur 4) indique que le paramètre est une valeur fractionnaire qui doit se situer à l'intérieur des limites 0,0 et 1,0.

E_NOCH (valeur 8) indique que l'utilisateur n'a pas le droit de modifier la valeur du paramètre avec l'éditeur de paramètres du MSPS. Ce contrôle de vérification peut s'appliquer à n'importe quel type de paramètre, C_INT, C_NUM ou C_STR.

E_OPT (valeur 16) indique que le paramètre est un type spécial d'«option», correspondant à un choix forcé (nombre entier) de valeur se situant entre 1 et le nombre maximal optionnel permis. Le nombre maximal lui-même est fourni, pour les paramètres d'options, par le huitième argument de pmaddent.

Si des codes multiples s'imposent, l'utilisateur peut simplement additionner des valeurs d'éléments pertinents ensemble. Par exemple, une valeur de 12 indique un paramètre qui doit être fractionnaire et que l'utilisateur n'a pas la permission de faire les modifications à la volée au moment de l'exécution.

Le huitième argument, représenté dans la description ci-dessus par «Row_max», indique le nombre maximal de rangs pour certains types de paramètres (P_VEC, P_LKPXY, P_LKPSL, ou P_TBL). (Il faut cependant noter la souplesse du MSPS du fait que le nombre réel de rangs utilisés pour une applications en particulier peut être inférieur à cette valeur maximale.) Pour les autres types de paramètres (P_SCL et P_DUMMY) cet argument devrait prendre la valeur 0 (zéro), sauf pour les paramètres d'OPTION, où il indique le nombre de valeurs d'options légitimes. (Une valeur N pour un paramètre d'OPTION indique que les valeurs légitimes se situent entre 1 et N inclusivement.) Comme les paramètres scalaires (P_SCL) sont la norme, cet argument prendra le plus souvent la valeur 0.

Le neuvième argument, représenté dans la description ci-dessus par «Rows_addr», contient l'adresse de la variable, en chiffre entier, correspondant au nombre réel (courant) de rangs pour certains genres de paramètres, P_VEC, P_LKPXY, P_LKPSL et P_TBL. Lorsque le nombre de rangs n'est pas pertinent, par exemple pour un paramètre scalaire ou un paramètre fictif, l'utilisateur entre la valeur «NULL» pour cet argument; par conséquent, cet argument prend habituellement la valeur «NULL».

Le dixième et dernier argument de pmaddent, représenté dans la description ci-dessus par «Limit», indique, pour les paramètres de type P_TBL, le nombre de colonnes de la table. Contrairement à la souplesse fournie par les rangs, où le nombre réel de rangs peut être plus petit que le nombre maximum, le MSPS exige que le nombre réel de colonnes soit fixé à l'avance. Pour tous les autres types de paramètres, cet argument prend la valeur 0 (zéro).

DESCRIPTION DES PARAMÈTRES SCALAIRES

Maintenant que la description des arguments de `pmaddent` est terminée, nous passons d'abord aux genres de paramètres scalaires que l'utilisateur peut désirer ajouter. La discussion porte ici sur eux dans l'ordre grossièrement descendant de leur fréquence d'utilisation prévue. Pour chacun des types, la description indique 1) la nature générale du paramètre, 2) les principaux arguments de `pmaddent`, et 3) un gabarit approprié de `pmaddent`. Même si ce chapitre traite principalement de types de paramètres plus spécialisés, nous avons, pour des fins d'exhaustivité, inclus à la section des paramètres scalaires, des cas de types de paramètres plus courants déjà décrits dans la section précédente du présent guide de programmation.

Paramètres RÉEL/flottant/NUMBER

L'analyste utilise ce type de paramètre lorsqu'il a besoin de fournir une valeur réelle, par exemple certaines garanties de programme exprimées en dollars et en cents. La définition dans `Mpu.h` utilisera la spécification `NUMBER`. Dans l'appel de `pmaddent`, l'argument clé est l'entrée `C_NUM` pour `C_Type`. Un gabarit approprié est --

```
pmaddent(pcp, "STDFA", (char *)&MP.STDFA, NULL, P_SCL, C_NUM, 0, 0, NULL, 0);
```

Paramètres ENTIER/int

L'analyste utilise ce paramètre lorsqu'il a besoin de fournir une valeur qui est, de façon inhérente, un chiffre entier, par exemple le nombre habituel de semaines de la période de carence aux fins de l'assurance-chômage. La définition dans `Mpu.h` utilisera la spécification `int` dans l'appel de `pmaddent`, l'argument clé est l'entrée `C_INT` pour `C_Type`. Un gabarit approprié est --

```
pmaddent(pcp, "UIWAITWKS", (char *)&MP.UIWAITWKS, NULL, P_SCL, C_INT, 0, 0, NULL, 0);
```

Paramètres de DRAPEAU/FLAG

L'analyste utilise ce type de paramètre lorsqu'il désire fournir une valeur de «commutation», par exemple un indicateur qui précisera si certains autres calculs doivent être exécutés ou non. La définition de `mpu.h` utilisera la spécification `int` pour un paramètre de ce genre. Dans l'appel de `pmaddent`, les arguments clés sont l'entrée `C_INT` pour le `C_Type` et l'entrée `E_FLAG` pour `edit`. Un gabarit approprié est --

```
pmaddent(pcp, "FAFLAG", (char *)&MP.FAFLAG, NULL, P_SCL, C_INT, E_FLAG, 0, NULL, 0);
```

Paramètres FRACTION

L'analyste utilise ce type de paramètre lorsqu'il désire fournir une valeur qui est, de façon inhérente, une fraction et, par conséquent, plus restreinte en valeur qu'un nombre à virgule. Les taux d'imposition et les taux de cotisation sont de bons exemples de ce type de paramètre. La définition dans `Mpu.h` utilise la spécification `NUMBER` dans ces paramètres. Dans l'appel de `pmaddent`, les arguments clés sont l'entrée `C_NUM` pour `C_Type` et l'entrée `F_FRACT` pour `Format`. Dans l'appel de gabarit que nous suggérons pour ce type de paramètre, l'utilisateur a choisi de NE PAS exiger le contrôle de vérification qui restreindra la valeur entre zéro et l'unité; le gabarit lui-même est --

```
pmaddent(pcp, "UIBASRATE", (char *)&MP.UIBASRATE, F_FRACT, P_SCL, C_NUM, 0, 0, NULL, 0);
```

Paramètres OPTION

L'analyste utilise ce type de paramètre lorsque le paramètre reflète un choix forcé entre un nombre fixe et restreint de choix possibles; une valeur numérique est utilisée pour indiquer une sélection nominale ou qualitative. Comme exemple d'une distinction qualitative de ce genre, on pourrait considérer un paramètre qui indique si les cotisations au RPC/RRQ doivent être traitées comme 1) une déduction aux fins de calcul du revenu imposable, ou 2) un crédit non remboursable aux fins de calcul des impôts, ou 3) un crédit d'impôt remboursable aux fins de l'impôt fédéral sur le revenu, mais non aux fins de l'impôt provincial sur le revenu. La définition dans Mpu.h pour un paramètre de DRAPEAU utilise une spécification int. Dans l'appel de pmaddent, les arguments clés sont l'entrée C_INT pour C_TYPE, l'entrée E_OPT pour Edit et l'entrée numérique donnant le nombre de catégories légitimes pour l'argument Row-max. Un gabarit approprié est --

```
pmaddent(pcp, "MDCROPT", (char *)&MP.MDCROPT, NULL, P_SCL, C_INT, E_OPT, 2, NULL, 0);
```

Paramètres EDIT-FRACTION

L'analyste utilise ce type de paramètre lorsqu'il est souhaitable de restreindre toute valeur fournie par l'utilisateur dans l'intervalle se situant entre zéro et l'unité. Par exemple, le paramètre pourrait représenter un taux de réduction des taxes qui serait considéré comme déraisonnable s'il correspondait à un taux inférieur à 0 % ou supérieur à 100 %. La définition utilisée dans Mpu.h pour un paramètre de fraction modifiable utilise une spécification NUMBER. Dans l'appel de pmaddent, les arguments clés sont l'entrée C_NUM pour C_Type et l'entrée E_FRCT pour Edit. L'utilisateur pourrait désirer préciser aussi une spécification de Format de F_FRACT. Un gabarit approprié est --

```
pmaddent(pcp, "CHATR1", (char *)&MP.CHATR1, NULL, P_SCL, C_NUM, E_FRCT, 0, NULL, 0);
```

Paramètres FICTIFS/DUMMY

L'utilisateur ne précise habituellement pas de paramètres FICTIFS, qui sont prévus pour la transmission de l'information sur l'étiquetage et la répartition en sections lorsque les configurations de paramètre sont documentées. Un gabarit représentatif est -

```
pmaddent(pcp, "2.3.1", NULL, NULL, P_DUMMY, C_STR, 0, 0, NULL, 0);
```

Pour tous les types de paramètres scalaires, l'utilisateur a le choix entre des mécanismes permettant de leur fournir les valeurs :

1. spécification par inclusion du paramètre dans le fichier de paramètres (fichiers MPR, CPR et APR),
2. spécification par présence dans un fichier de paramètres d'inclusion supplémentaire (fichiers MPI, CPI et API), et
3. spécification par la fonction de modification dynamique des paramètres. (Il faut cependant noter que la capacité d'utiliser la troisième option peut être restreinte par l'entrée pmaddent du paramètre pour l'argument Edit.) Cette méthode est automatique si l'utilisateur choisit de ne pas préciser de valeur; si l'argument Edit le permet, le MSPS affichera un message pour demander une valeur.

VECTEURS DE PARAMÈTRES DÉFINIS PAR L'UTILISATEUR

Les parties précédentes du présent chapitre portaient principalement sur le paramètres

scalaires, en partie du fait qu'ils constituent les types les plus courants, et en partie parce qu'ils sont les plus faciles à décrire. Cependant, le MSPS offre aussi à l'utilisateur la possibilité de créer des vecteurs de paramètres. Ces vecteurs sont plus pertinents lorsque l'utilisateur désire créer un ensemble de paramètres connexes avec les membres de l'ensemble se retrouvant dans un ordre «indexable» naturel dans une seule dimension.

Supposons, par exemple le cas d'une liste modélisée dans un certain programme de supplément au logement que l'on veut proposer. Pour chaque grosseur de famille jusqu'à dix, ce programme hypothétique a une limite de revenu au-delà de laquelle une famille devient catégoriquement inadmissible aux prestations. Malheureusement, ces limites, bien qu'augmentant avec la grosseur de la famille, ne sont pas liées à la grosseur de la famille de façon uniforme ou facile à calculer. Plutôt, l'utilisateur désire avoir dix paramètres différents, correspondant aux grosseurs de famille de un à dix et plus, pour représenter les seuils de prestation. Il est beaucoup plus logique d'avoir un vecteur de paramètres, indexé sur la grosseur de famille, que d'élaborer un code qui traite chacune des dix possibilités comme un cas distinct et développé indépendamment.

Dans cette section, nous définissons les points clés que l'utilisateur doit comprendre pour définir les vecteurs de paramètres utilisateur pour les modèles du MSPS. Nos commentaires ci-dessus au sujet de l'ajout de paramètres en général vaut toujours (ordre des changements à apporter aux fichiers, utilisation des mnémoniques, validation, etc.), mais nous nous concentrons sur les aspects propres à l'utilisation efficace des vecteurs de paramètres définis par l'utilisateur.

Ajouts aux fichiers `Mpu.h`, `Cpu.h` ou `Apu.h`

Tout comme l'utilisateur déclare des paramètres scalaires dans `Mpu.h` (ou `Cpu.h` ou `Apu.h`), il doit aussi déclarer tout vecteur de paramètres définis par l'utilisateur dans ces fichiers. Les déclarations de valeur scalaire et de valeur vectorielle se ressemblent beaucoup, sauf que les déclarations de vecteur indiquent, par une expression entre crochets, la longueur du vecteur. Le MSPS traite les vecteurs de paramètres comme des vecteurs de colonne; par conséquent, la longueur du vecteur est le nombre de rangs.

Pour notre exemple de programme de subvention au logement, supposons que l'utilisateur a déclaré une constante manifeste `HPPYCOMR` (programme de logement hypothétique, nombre maximum de rangs de seuil de revenu). L'utilisateur lui a attribué la valeur dix parce qu'il y aura un seuil distinct pour chaque grosseur de famille, jusqu'à dix et plus. La définition serait faite par une instruction de la forme --

```
#define HPPYCOMR 10 /* maximum # of number of rows in the HPPYCO vector */
```

Il faut consulter le fichier `Mp.h` dans le sous-répertoire `SPSM\DEFS` (commençant vers la ligne 40) pour obtenir des illustrations de vecteurs de paramètres qui font partie du MSPS en boîte noire, plutôt que d'être définis par l'utilisateur.

Le vecteur lui-même doit être nommé `HPPYCO`, avec la valeur de la *i*ème entrée correspondant au seuil pour une famille de grosseur *i*+1. (Il faut se rappeler que le langage C débute tous les vecteurs avec l'entrée zéro.) L'entrée `Mpu.h` pour le nouveau vecteur ressemble sensiblement à --

```
NUMBER HHPYCO[HHPYCOMR]; /* Hypothetical Housing Program Income Cutoffs */
```

Bien qu'il soit possible d'entrer la longueur directement dans la déclaration, par exemple en utilisant quelque chose comme `HHPYCO[10]`, nous vous encourageons fortement de ne pas le faire. Nous recommandons plutôt l'approche de la constante manifeste décrite ci-dessus. La raison de notre recommandation découle du besoin, dans l'appel correspondant de `pmaddent` dans `Ampd.cpp`, d'une entrée pour le nombre maximum de rangs. L'utilisation d'une constante manifeste donnée aux deux endroits élimine la possibilité qu'une révision ultérieure mène à l'utilisation d'une valeur dans `Mpu.h` tandis qu'une autre sera utilisée dans `Ampd.c`. Si l'utilisateur créait un écart entre la valeur dans `Mpu.h` (ou `Apu.h` ou `Cpu.h`) et la valeur dans `Ampd.c`, les erreurs qui surviendraient pourraient poser de très grandes difficultés quand il s'agirait de trouver leur cause.

Rappelez-vous que le nombre réel de rangs présents dans un vecteur (colonne) pour une exécution donnée du MSPS peut différer du nombre maximum possible pour ce paramètre (lui être inférieur). Par conséquent, l'utilisateur doit aussi déclarer, dans le même fichier d'en-tête, une variable dans laquelle le MSPS stockera le nombre réel de rangs utilisés (une valeur qui peut varier d'une exécution à l'autre d'une version exécutable donnée en mode boîte de verre). L'utilisateur fournit une variable dans laquelle le MSPS stockera le nombre réel de rangs par une déclaration supplémentaire dans le fichier d'en-tête. Selon la règle du MSPS, selon laquelle ces variables longues sont nommées selon le nom du paramètre avec un suffixe de «rangs», le fichier `Mpu.h` devrait aussi contenir une déclaration de la forme suivante --

```
int HPPYCOrows; /* number of rows in HPPYCO */
```

Le fichier `mp.h`, dans le sous-répertoire `SPSM\DEFS`, fournit de nombreux exemples dans sa section portant sur les limites de tableaux (vers la ligne 580). Plus loin, l'appel de `HPPYCO` par `pmaddent` dans le `Ampd.cpp` référera à l'adresse de la variable `HPPYCOrows`.

Ajouts au fichier `Ampd.cpp`

Pour que le MSPS puisse rendre les valeurs du nouveau vecteur de paramètres disponibles au code final de l'utilisateur, l'utilisateur doit établir les liens appropriés par un appel de `pmaddent`, tout comme avec des paramètres scalaires. L'appel devrait ressembler à ce qui suit :

```
pmaddent(pcp, "HHPYCO", (char *)MP.UM.HHPYCO, NULL, P_VCT, C_NUM, E_NONE,
HHPYCOMR, &MP.UM.HHPYCOrows, 0);
```

ou

```
pmaddent(pcp, "HHPYCO", (char *)&MP.UM.HHPYCO[0], NULL, P_VCT, C_NUM, E_NONE,
HHPYCOMR, &MP.UM.HHPYCOrows, 0);
```

Dans le premier appel donné en exemple, le troisième argument n'utilise pas la `&` parce que la référence vise le nouveau vecteur de paramètres; le langage C traite une référence de ce genre comme l'adresse du premier élément. Dans le deuxième exemple, l'utilisateur a choisi de référer plus explicitement à l'adresse du premier élément en incluant la `&` et l'index `[0]`. Les fichiers `MpdX.cpp` du sous-répertoire `SPSM\MODEL` contiennent des exemples des deux types de référence.

Trois autres arguments de `pmaddent` méritent des commentaires particuliers pour notre description des points particuliers des vecteurs de paramètres définis par l'utilisateur. L'argument `Agg_Type` (n° 5) prend nécessairement la valeur `P_VCT`. L'argument `Row-max` (n° 8) est la constante manifeste créée dans `Mpu.h` pour spécifier le nombre maximum de rangs; dans notre exemple de programme de subvention au logement, cela correspond à l'entrée `HHPYCOMR`. Enfin, l'entrée `Rows-addr` (n° 9) correspond au nom de la variable déclarée qui doit contenir le nombre réel de rangs, précédé de la perluette; dans notre programme de subvention au logement donné en exemple, cela correspond à l'entrée `&MP.UM.HHPYCOrows`.

Il faut noter que d'autres capacités activées par les arguments de `pmaddent` demeurent accessibles à l'utilisateur. Par conséquent, l'utilisateur emploie `C_Type` pour indiquer si la variable a une valeur à virgule flottante ou une valeur entière. L'utilisateur emploie l'argument `Format` pour préciser, s'il le désire, un format pour chacune des valeurs individuelles du vecteur. Et l'utilisateur emploie l'argument `Edit` pour imposer tout contrôle de vérification pertinent.

Tout comme c'est le cas avec les paramètres scalaires, l'utilisateur désire aussi modifier le fichier `Ampd.cpp` pour ajouter l'appel de `stradd` pour chaque nouveau vecteur de paramètres définis par l'utilisateur. Cet ajout fera en sorte que, lorsque le MSPS documente les nouveau paramètres définis par l'utilisateur, la description textuelle de l'utilisateur touchant les paramètres fera partie de cette documentation.

Références aux vecteurs de paramètres définis par l'utilisateur dans le code source

Quand l'utilisateur a fini de modifier le fichier d'en-tête et le fichier `Ampd.cpp` qui sont nécessaires pour rendre le vecteur de paramètres accessibles aux fonctions de base, il reste à référer aux valeurs de paramètre pertinentes dans ces fonctions de base. Pour poursuivre avec notre exemple hypothétique de programme de subvention au logement, supposons que l'utilisateur rend accessible une variable à chiffre entier, `HHPFS` (grosueur de famille pour le programme hypothétique de subvention au logement), qui donne la grosueur de famille définie par les règlements que l'on prévoit appliquer pour régir le programme. Supposons aussi que l'utilisateur est absolument certain que la valeur de `HHPFS` se situera à l'intérieur de la plage 1 à 9 inclusivement. Pour référer au seuil de revenu pertinent pour les prestations du programme hypothétique, l'utilisateur, se rappelant que le langage C numérote toujours les éléments d'un vecteur en commençant à zéro, emploierait une expression de la forme suivante :

```
MP.UM.HHPYCO[HHPFS-1]
```

Spécification des valeurs de vecteur de paramètres

Afin que le nouveau code de l'utilisateur puisse vraiment faire quelque chose, les valeurs des éléments du vecteur doivent être mises à la disposition du MSPS de façon qu'il puisse, à son tour, les rendre accessibles au code de l'utilisateur. Habituellement, l'utilisateur précise ces valeurs dans un fichier «.MPR» ou «.MPI» (ou à leur équivalent ".CPR", ".CPI", ".APR" ou ".API"). Le vecteur `UIREPUER`, précisant le taux de chômage régional dans la mesure où il s'applique aux exigences d'admissibilité à l'A.-C. pour les gens qui se retrouvent encore une fois dans la situation de chômage, fournit un bon exemple.


```

UIREPUER      5      # Regional unemployment rate
6.0
7.0
8.0
9.0
11.5

```

Le format est clair. La première ligne contient le nom du paramètre et est suivie du nombre RÉEL d'éléments à utiliser; il faudrait ajouter un commentaire d'information facultatif pour indiquer de façon évidente la nature du paramètre à l'intention de tout lecteur du fichier. Des lignes successives précisent, à raison d'une valeur par ligne, les valeurs du vecteur. Il est important que le nombre d'entrées d'éléments ne dépasse pas la valeur du nombre maximum de rangs précisé dans l'entrée pmaddent et que le nombre de lignes supplémentaires du fichier de paramètres soit égal au nombre donné à la première ligne des paramètres; le MSPS vérifie que ces exigences sont respectées.

Pour poursuivre avec notre programme hypothétique de prestations au logement, l'utilisateur peut entrer, dans le fichier ".MPR" ou ".MPI", quelque chose comme ce qui suit :

```

HHPYCO      10      # Income cutoffs for housing program, by family size
5000.0
6120.0
7250.0
8400.0
9500.0
10600.0
11600.0
12500.0
13300.0
13900.0

```

Résumé

Les facteurs clés de l'ajout de vecteurs de paramètres utilisateur à un modèle boîte de verre du MSPS peuvent se résumer ainsi :

1. Faire les modifications appropriées au fichier d'en-tête (p. ex., Mpu.h).
 - Utiliser une constante manifeste pour la longueur maximale du vecteur, p. ex.,
 - #define HHPYCOMR 10 /* Nombre maximum de rangs pour HHPYCO */
 - Déclarer le vecteur lui-même,
 - NUMBER HHPYCO[HHPYCOMR]; /* commentaire */
 - Déclarer une variable pour recevoir la longueur réelle de ce vecteur, p. ex.
 - int HPPYCOrows; /* Nombre réel de rangs dans HPPYCO */
2. Apporter les changements appropriés au fichier Ampd.cpp; rappelez-vous des avantages d'une compilation partielle.
 - Insérer un appel approprié de pmaddent, habituellement par la modification d'une copie d'un appel existant.
 - Entrer un appel de stradd de façon que le MSPS puisse étiqueter les nouveaux paramètres au besoin.
3. Rédiger le code de source en langage C qui utilise les paramètres. Rappelez-vous la règle du C selon laquelle les vecteurs commencent à numéroté l'élément à zéro. La

compilation de mise au point est souvent utile ici aussi.

4. Fournir la valeur pour les éléments du vecteur par une entrée multilignes dans le fichier de paramètres approprié.
5. Ne pas oublier la nécessité de validation et d'essai si vous voulez être certain qu'un nouveau code fait ce qu'il devrait faire.

TABLEAUX DÉFINIS PAR L'UTILISATEUR POUR LES RECHERCHES

Les paramètres, sous forme de tableaux, sont utilisés principalement lorsqu'on a besoin d'exécuter une certaine forme de recherche, par exemple pour une valeur donnée en x, trouver la valeur correspondante en y. La présente section emploie, comme exemples, deux tableaux déjà présents dans le MSPS et un nouveau tableau hypothétique défini par l'utilisateur qui doit être ajouté comme paramètre. Ensemble, les trois exemples couvrent les trois grandes formes de paramètres de tableau dont un utilisateur du mode boîte de verre pourrait habituellement avoir besoin.

Le premier des exemples de tableau existant touche les impôts fédéraux -- pour un revenu imposable donné, calculer l'impôt correspondant à partir du tableau/barème d'imposition.

Le deuxième exemple de tableau existant a trait au taux de participation à un programme -- en supposant que la décision de faire ou non une demande de prestation dans le cadre d'un programme est censée dépendre de la prestation qui pourrait être demandée (plus élevée serait la prestation qui serait reçue, plus il serait probablement qu'une unité fasse une demande de cette prestation), étant donné l'avantage possible pour une unité, rechercher la probabilité que l'unité fasse une demande de prestation (ou de participation au programme).

Le troisième exemple, le nouveau paramètre, porte sur un supplément de revenu entièrement hypothétique basé très libéralement sur un crédit d'impôt sur le revenu gagné aux États-Unis, mais appliqué à des gains individuels. Dans ce cas, les programmes hypothétiques de supplément de revenu donnent une prestation au revenu initial, jusqu'à 10 000 \$ par année, au taux de 15 %, ne donnent pas de prestations aux gains de 10 000 \$ à 15 000 \$, puis, au-delà de 15 000 \$, réduit la prestation donnée auparavant au taux de 10 % des gains au-dessus de 15 000 \$, de façon qu'il n'y ait pas de prestations à verser à des personnes gagnant 30 000 \$ ou plus. Le nouveau paramètre décrira la prestation payable comme fonction des gains de l'individu. Les paires de coordonnées pertinentes sont donc (0, 0), (10000, 1500), (15000, 1500) et (30000, 0).

Pour ce qui est de leur spécification comme paramètres du MSPS, les tableaux sont très semblables à des vecteurs. La grande exception est que le tableau a un nombre fixe de colonnes, trois, plutôt que la simple colonne d'un vecteur. (Pour ce qui est de leur utilisation, les tableaux emploient les fonctions lkup1 et lkup2 du MSPS.) Par conséquent, avec les exceptions relativement mineures mises en évidence dans la présente section, on ajoute un tableau à une application boîte de verre à peu près comme on ajouterait un vecteur de paramètres. Par conséquent, les prescriptions axées sur les vecteurs concernant les noms mnémoniques, les étiquetages stradd, les compilations partielles, les validations, etc. ne sont pas reprises ici.

Types de tableaux et fonctions de recherche

Il est essentiel de faire immédiatement une distinction entre deux choses pour l'utilisation efficace des tableaux dans le MSPS.

La première chose à distinguer touche le type de tableaux. L'utilisateur fait le choix de type par le cinquième argument de l'appel de pmaddent.

Si l'argument est P_LKPXY, alors les recherches dans le tableau sont faites dans le format X-Y, avec la première colonne (la valeur x) du tableau et la seconde colonne (valeur y); la valeur de pente de la troisième colonne (la pente des segments successifs de tableau) est présente, mais elle n'est pas prise en compte (cette information étant redondante du fait qu'elle est calculée à partir de la paire X-Y). Si le cinquième argument de pmaddent est P_LKPSL, alors les recherches dans le tableau se feront dans le format de la pente, l'information qu'il y a dans la première colonne (valeur x) et la troisième colonne (pentes) étant utilisée en plus de la première valeur de la deuxième colonne (valeur y). Les autres valeurs de la deuxième colonne ne sont pas prises en compte dans ce qu'elles sont redondantes du fait qu'elles pourraient être calculées avec le reste de l'information contenue dans le tableau.

Le deuxième point à distinguer est de déterminer si l'utilisateur désire ou non appliquer l'interpolation dans le calcul lorsqu'il fait la recherche connexe à l'intérieur du tableau. Lorsque l'interpolation est désirée (lorsque la valeur désirée pourrait se situer ENTRE les entrées de la colonne des valeurs y), l'utilisateur invoque la fonction lkup1 de la bibliothèque d'algorithmes du MSPS. Lorsque l'interpolation n'est pas désirée, l'utilisateur invoque la fonction sœur lkup2. Le Guide des algorithmes fait une description précise de ces deux algorithmes.

Présence dans les fichiers d'en-tête du MSPS

Tout comme avec les vecteurs de paramètres, les paramètres définis par l'utilisateur qui sont des tableaux exigent certaines entrées dans le fichier d'en-tête approprié (Mpu.h, Cpu.h, ou Apu.h).

Une de ces entrées est (habituellement) une constante manifeste servant à définir la longueur maximale du tableau. Le tableau le barème d'impôt fédéral (FTX) utilise la longueur maximale (FTXMAX). Le tableau de participation des célibataires prestataires de pension de retraite touchant le SRG (GISST) utilise GISSTMAX. Pour notre tableau de supplément des gains, ESS, nous utiliserons ESSMAX. Les définitions correspondantes (dans Mp.h pour FTXMAX et GISSTMAX, et dans Mpu.h pour ESSMAX) sont les suivantes :

```
#define FTXMAX    15      /* maximum of number of rows in FTX table      */
#define GISSTMAX  8       /* maximum of number of elements in GISST table */
```

et

```
#define ESSMAX    5       /* maximum number of rows in ESS schedule      */
```

La deuxième est une variable dans laquelle le MSPS stocke le nombre réel de rangs utilisés dans le tableau pour une exécution donnée. Il doit, bien sûr, être inférieur ou égal au nombre maximal. Selon les règles du MSPS, les définitions des variables dans Mp.h qui doivent

contenir le nombre réel d'éléments sont les suivantes :

```
int  GISSTrows;          /* number of rows in GISST table */
int  FTXrows;            /* number of rows in FTX          */
```

Dans `mpu.h`, nous expliquerons cette règle et définirons une variable `ESSrows` pour le nombre réel de rangs dans ESS --

```
int  ESSrows;            /* number of rows in ESS schedule */
```

`Mp.h` (pour les tableaux de FTX et GISST) et `Mpu.h` (pour le tableau ESS) ont aussi besoin de contenir la définition des tableaux eux-mêmes. Habituellement, ils sont faits avec des constantes manifestes définies auparavant. Le MSPS fournit une constante, `LKP_COLS`, qui indique clairement son rôle dans la définition du nombre de colonnes pour les tableaux de recherche. Les définitions elles-mêmes sont simples :

```
NUMBER FTX[FTXMAX][LKP_COLS]; /* Federal tax table [taxable income,basic federal
tax] */
NUMBER GISST[GISSTMAX][LKP_COLS]; /* GIS take-up rate: single pensioner by benefit
level [benefit,rate] */
NUMBER ESS[ESSMAX][LKP_COLS]; /* Earnings supplement schedule [earnings, benefit
level] */
```

Présence dans les appels de `pmaddent` dans `Ampd.cpp`

L'utilisateur qui définit des paramètres de tableau doit modifier le fichier `Ampd.cpp`, en ajoutant les appels de `pmaddent`, pour permettre au MSPS de mettre le paramètre à la disposition du code source réel. Nous commençons par examiner les entrées pertinentes de `pmaddent` du MSPS afin d'y trouver les tableaux FTX et GISST.

L'exemple FTX, tiré du fichier `Mpd2.cpp`, ressemble à ce qui suit :

```
pmaddent(pcp, "FTX", (char *)&MP.FTX[0][0], NULL, P_LKPSL, C_NUM, 0,
FTXMAX, &MP.FTXrows, 0);
```

Il faut noter que le troisième argument indique clairement que le tableau a des rangs et des colonnes et que le cinquième argument indique qu'il s'agit d'un tableau axé sur les pentes; le huitième et le neuvième arguments utilisent les entrées de constante manifeste et de nombre réel de rangs définies dans `Mp.h`.

L'exemple du GISST, tiré du fichier `Mpd1.cpp` ressemble à ce qui suit :

```
pmaddent(pcp, "GISST", (char *)&MP.GISST[0][0], F_LKTUR, P_LKPXY, C_NUM, E_F
```

Ici, le cinquième argument indique qu'il s'agit d'un tableau du type X-Y. Encore une fois, le huitième et le neuvième arguments utilisent les éléments définis dans le tableau, dans le fichier `mp.h`.

Pour le programme hypothétique de supplément de gains, nous ajouterions au fichier `Ampd.cpp` un appel de `pmaddent` (probablement copiée d'un appel existant modifié au besoin) qui ressemble à ce qui suit :

```
pmaddent(pcp, "ESS", (char *)&MP.UM.ESS[0][0], NULL, P_LKPXY, C_NUM, 0, ESSMAX,
&MP.UM.ESSrows, 0);
```

Les forts parallèles qu'il y a avec le tableau GISST existant devraient être évidents. Il faut cependant noter les grandes différences qui touchent le tableau de paramètres défini par l'utilisateur : le qualificatif UM dans le troisième et le neuvième arguments, la constante définie par l'utilisateur (nombre maximum de rangs) et l'adresse de variable (nombre réel de rangs) pour les huitième et neuvième arguments de pmaddent.

Emploi des références au tableau dans le code utilisateur

Les applications en boîte de verre qui utilisent les tableaux font référence à ceux-ci presque exclusivement par deux fonctions de recherche du MSPS, lkup1 et lkup2. Cela simplifie beaucoup les expressions du code de source utilisant les paramètres. Nos trois exemples illustrent la nature de ces références.

La fonction ATXCALC.CPP du sous-répertoire GLASS sert pour le calcul de l'impôt fédéral sur le revenu. Ce calcul exige, pour un individu, la recherche de l'impôt de cet individu comme fonction de son revenu imposable. L'utilisateur choisit d'appliquer ou non l'interpolation (en choisissant entre lkup1 et lkup2), fournit le tableau, le nombre de rangs et la valeur x pertinente, et la fonction de recherche fait automatiquement le reste. Ici, l'utilisateur désire l'interpolation, appliquée dans un tableau. Le code de source pertinent ressemble à ce qui suit :

```
if (isnzero(in->im.imitax)) {
/* calculate federal tax */
in->im.imfedtax = (NUMBER) lkup1(MP.FTX, MP.FTXrows, in->im.imitax);
DEBUG2("%s fedtax =%.2f\n", in->im.imfedtax);
}
```

La fonction AGIS.CPP du sous-répertoire GLASS calcule les prestations de SRG. Ce calcul exige la recherche, comme fonction des prestations possibles qui seraient payables, de la probabilité que l'unité adhérerait au programme de prestations (c.-à-d. qu'il en fera la demande). Ici, l'utilisateur choisit de ne pas invoquer l'interpolation - le taux de participation désiré est celui du dernier rang dans lequel la prestation possible est au moins aussi grande que la valeur x du rang. L'utilisateur fournit le tableau, le nombre réel de rangs et les prestations de SRG possibles, et la fonction de recherche retourne la probabilité de participation. (Encore une fois, le tableau lui-même se trouve à la sous-section suivante.) L'expression servant à vérifier la probabilité de participation ressemble à ce qui suit

```
lkup2(MP.GISST, MP.GISSTrows, (double) gis))
```

Pour l'illustration du supplément de gains, supposons que l'utilisateur a attribué la définition appropriée de gains pour un individu à une variable (double) nommée iearn. Alors, l'expression de recherche du supplément de gains correspondant pour l'individu serait --

```
lkup1(MP.UM.ESS, MP.UM.ESSrows, iearn)
```

Il faut noter la nécessité du qualificatif UM indiquant que ESS est un tableau défini par l'utilisateur.

Présence dans les fichiers de paramètres

Comme avec tout autre paramètre, l'utilisateur a la responsabilité de définir les paramètres de tableau dans le fichier de paramètres approprié (.MPR/I, .CPR/I ou .APR/I). En parallèle à la spécification d'un vecteur de paramètres, la première ligne fournit le nom du

paramètre et le nombre de rangs, avec un commentaire identifiant le paramètre. Les rangs suivants du tableau sont la valeur x, la valeur y, le triplet de la pente. Il est probable que la seule caractéristique qui n'est pas évidente est que les articles redondants (ceux qui ne seront pas utilisés pour le calcul) sont insérés entre parenthèses.

Le tableau FTX axé sur la pente décrit l'impôt à payer (avant la réforme fiscale) comme fonction du revenu imposable --

FTX	10	#	Federal	tax
			table	
	0	0	0.060	
	1238	(74)	0.160	
	2476	(272)	0.170	
	4952	(693)	0.180	
	7428	(1139)	0.190	
	12380	(2080)	0.200	
	17332	(3070)	0.230	
	22284	(4209)	0.250	
	34664	(7304)	0.300	
	59424	(14732)	0.340	

Le tableau GISST du type X-Y décrit les probabilités de participation comme fonction du montant des prestations de SRG disponibles. L'utilisation de la fonction lkup2 avec le tableau signifie que les taux de participation sont modélisés comme ayant une croissance abrupte aux niveaux de prestations clés.

GISST	5	#	GIS take-up rate: single pensioner by benefit
		level	
	0	0.365	(0.0009)
	169	0.510	(0.0006)
	419	0.660	(0.0003)
	919	0.820	(0.0001)
	3169	1.000	(0.0001)

Le tableau ESS de type X-Y décrit la prestation de supplément de gains comme fonction des gains de l'individu; il est utilisé avec la fonction lkup1 parce que l'interpolation est désirée.

ESS	4	#	Hypothetical earnings supplement schedule
	0	0	(0.15)
	10000	1500	(0.00)
	15000	1500	(-0.10)
	30000	0	(0.00)

Points clés pour l'ajout de paramètres de tableau

La plupart des points clés des paramètres de tableau sont identiques à ceux des paramètres de vecteur.

1. Modifier le fichier d'en-tête pertinent de façon à inclure une constante manifeste pour le

nombre maximum de rangs, une variable à chiffre entier pour stocker le nombre réel de rangs, et la définition du tableau lui-même.

2. Modifier le fichier `Ampd.cpp` de façon à inclure les appels de `pmaddent` et de `stradd` appropriés, généralement copiés de quelque part ailleurs, puis modifiés.
3. Fournir le tableau dans un fichier de paramètres ou un fichier d'inclusion de paramètres approprié et ne pas oublier de valider l'ajout.

Deux autres points clés sont propres aux paramètres de tableau.

1. Assurez-vous, dans le fichier de paramètres, que la colonne des valeurs `x` du tableau contient des valeurs qui sont strictement en ordre ascendant.
2. N'oubliez pas de «marquer» les valeurs redondantes dans le tableau en les mettant entre parenthèses.

AJOUT DE MATRICES DE PARAMÈTRES

Pour certaines fins spécialisées touchant des groupes de paramètres, même des vecteurs ou des tableaux de paramètres ne sont pas suffisamment commodes. Par exemple, plutôt que d'avoir plusieurs vecteurs d'égale longueur en parallèle, il peut être beaucoup plus efficace de faire une recherche dans une matrice de valeurs. La conception du MSPS permet la définition et l'utilisation de ces matrices, bien que le programme en limite à deux le nombre de dimensions (rangs et colonnes). Cette section décrira l'utilisation des matrices de paramètres par deux exemples, un tiré de la version boîte noire du MSPS, et l'autre exigeant la spécification d'une nouvelle matrice de paramètres définie par l'utilisateur. Étant donné l'étroite relation qu'il y a entre les vecteurs de paramètres et les matrices de paramètres, il n'y a pas de division marquée en particulier pour la présente section.

L'illustration en boîte noire utilise la matrice `CTPRST` propre aux capacités de taxe à la consommation du MSPS. Ce paramètre fournit une grande matrice de facteurs (40 éléments de consommation (rangs) par 10 provinces (colonnes)) touchant le calcul des taxes de vente provinciales.

Le second exemple, dans lequel l'utilisateur ajoute au MSPS une nouvelle matrice de paramètres, exige une matrice de niveaux seuil de revenu pour une mesure hypothétique et expérimentale de la pauvreté. Afin de faciliter la classification des familles comme étant ou non pauvres, l'utilisateur désire avoir une matrice qui fournit les seuils pertinents comme fonction de variables à nombre entier spécifiant les structures de famille (rangs) et la grosseur du lieu de résidence (colonnes). Par conséquent, l'entrée de la matrice (3,2) contiendra le seuil de pauvreté pour une famille dont l'index de structure est trois et dont l'index de la grosseur du lieu de résidence est deux. L'utilisateur a choisi de nommer sa matrice `EPMCO` (seuils expérimentaux de mesure de la pauvreté). Pour les fins de cet exemple, nous supposerons que l'utilisateur choisit une mesure définie en fonction de 18 structures de familles (impliquant, disons, des combinaisons de nombre et d'âges des membres de la famille) et quatre catégories de grosseurs de lieux de résidence.

Présence dans `Mpu.h`

Prenons d'abord l'exemple de la matrice de la boîte noire. Nous ne sommes pas surpris de trouver une information d'en-tête pertinente pour CTPRST dans le fichier Mp.h du sous-répertoire DEFS. Par conséquent, il y a une définition de nombre entier, pour définir le nombre réel de rangs (éléments de consommation) CTNUMCOM, de la façon suivante :

```
int CTNUMCOM; /* number of rows for commodity dimension parms */
```

Il y a en outre une définition pour la matrice elle-même --

```
NUMBER CTPRST[NUMCOM][NUMREG]; /* Provincial retail sales tax [com x prov] */
```

Cependant, Mp.h ne contient pas de constantes manifestes pour les dimensions de la matrice (NUMCOM et NUMREG) puisqu'elles sont si étroitement liées à la conception de la fonction de taxe à la consommation dans le MSPS qu'elles ont été définies ailleurs de façon que le module de taxe à la consommation puisse utiliser les constantes de façon plus commode.

En revenant à notre matrice expérimentale de seuils de mesure de la pauvreté, nous savons que nous devons fournir l'information «de définition» pertinente au MSPS par des entrées dans le fichier Mpu.h. Les besoins spécifiques sont (1) les constantes manifestes pour les dimensions, (2) une variable pour le nombre réel de rangs, et (3) la matrice elle-même. Les lignes correspondant à ces articles dans Mpu.h pourraient ressembler à ce qui suit :

```
#define EPMFAMMAX 18 /* maximum of number of family structures (rows) for EPMCO matrix */
```

```
#define EPMSIZE 4 /* number of size of place of residence categories for EPMCO matrix */
```

```
int EPMCOrows; /* number of rows for EPMCO matrix */
```

```
NUMBER EPMCO[EPMFAMMAX][EPMSIZMAX]; /* experimental poverty measure cutoffs [fam x size] */
```

Présence dans Ampd.cpp

En parallèle avec le besoin de vecteurs de paramètres, le MSPS a besoin pour chaque matrice de paramètres un appel de pmaddent de façon que la valeur de paramètres puisse être mise à la disposition du code de source de l'utilisateur.

Pour notre exemple en boîte noire, cet appel, qui se trouve dans le fichier Mpd4.cpp, ressemble à ce qui suit : (il y a, bien sûr, un appel de stradd correspondant.)

```
pmaddent(pcp, "CTPRST", (char *)MP.CTPRST, NULL, P_TBL, C_NUM, E_FIXL,
NUMCOM, &MP.CTNUMCOM, NUMREG);
```

Les seuls arguments qui nous intéressent en particulier à ce moment-ci sont l'entrée P_TBL pour le cinquième argument (Agg_Type) et l'entrée NUMREG pour l'argument final (nombre de colonnes). Les huitième et neuvième entrées (maximum et adresse du nombre réel de rangs) sont exactement telles que nous nous serions attendu qu'elles soient étant donné les descriptions précédentes des vecteurs de tableau.

Si nous étudions notre exemple de mesure de la pauvreté en boîte de verre, nous reconnaissons qu'il est nécessaire d'ajouter un appel de pmaddent dans le fichier Ampd.cpp afin de permettre au MSPS de donner au code de source de l'utilisateur l'accès à la matrice de paramètres. Cet appel pourrait bien ressembler à ce qui suit :

```
pmaddent(pcp, "EPMCO", (char *)MP.UM.EPMCO, NULL, P_TBL, C_NUM, E_NONE, EPMFAMMAX, &MP.UM.EPMCOrows,
```


EPMSIZE);

On suppose que l'utilisateur ajoutera aussi au fichier `Ampd.cpp` un appel de `stradd` afin de permettre au MSPS de produire l'information documentaire appropriée.

Référence aux éléments de la matrice dans le code de source

La référence aux éléments de la matrice de paramètres est facile. Si on suppose que la variable `i` contient la catégorie élément de consommation (nombre entier) et la variable `j` le code de la province (nombre entier), alors le facteur de récupération correspondant à cette combinaison est --

```
MP.CTPRST[i][j]
```

De même, si la variable à nombre entier `fstruct` contient le code de structure de famille et la variable à nombre entier `sizecode` fournit la catégorie de la grosseur du lieu de résidence, alors le seuil expérimental de mesure de la pauvreté pour la combinaison structure/grosseur est donné par --

```
MP.UM.EPMCO[fstruct][sizecode]
```

Le facteur principal à prendre en compte dans ces références est la règle du langage C que chaque dimension commence avec un élément zéro; par exemple, notre tableau de 18 sur 4 utilise des indices qui vont de 0 à 17 et de 0 à 3 respectivement. Un utilisateur doit décider du compromis approprié entre l'utilisation de nombres entiers «naturel et positif» comme indice dans les matrices, et l'économie d'un bloc de mémoire fixe disponible pour les paramètres utilisateur (y compris toute variable d'adresse de rang nécessaire).

Présence dans les fichiers de paramètres

Comme c'est le cas avec toute autre forme de paramètre, l'utilisateur doit fournir des valeurs pour les paramètres. Habituellement, cela se fait par des entrées dans les fichiers de paramètres ou les fichiers d'inclusion de paramètres appropriés (p. ex., `.MPR`, `.MP`, `.CPR`, `.CPI`, `.APR` ou `.API`). Pour les matrices de paramètres, une entrée dans le fichier de paramètres comprend une première ligne qui précise le nom du paramètre et le nombre réel de rangs, ainsi qu'un commentaire de documentation, habituellement. Les lignes suivantes du paramètre fournissent alors les rangs de la matrice. Dans notre exemple, nous fournissons seulement la première ligne, d'identification, puis la première des lignes des valeurs numériques.

Pour l'exemple en boîte noire --

```
CTPRST      40      # Provincial retail sales tax
0.01326 0.01326 0.01326 0.01326 0.01316 0.01406 0.02242 0.00626 0.00010 0.00550
0.15257 0.15257 0.15257 0.15257 0.13057 0.24354 0.15684 0.13914 0.00013 0.29100
0.17538 0.17538 0.17538 0.17538 0.16338 0.22635 0.13837 0.08953 0.00010 0.00605
0.08125 0.08125 0.08125 0.08125 0.08424 0.07750 0.06300 0.08521 0.00009 0.07406
0.08029 0.08029 0.08029 0.08029 0.07239 0.06953 0.05715 0.07306 0.00010 0.06512
0.08293 0.08293 0.08293 0.08293 0.06684 0.05282 0.05581 0.00305 0.00008 0.06866
0.00296 0.00296 0.00296 0.00296 0.00359 0.00197 0.00130 0.00171 0.00001 0.00141
0.00997 0.00997 0.00997 0.00997 0.00934 0.00753 0.01018 0.01073 0.00024 0.01057
0.00886 0.00886 0.00886 0.00886 0.01140 0.01421 0.00969 0.00879 0.00022 0.01017
0.08363 0.08363 0.08363 0.08363 0.06777 0.00206 0.02368 0.04331 0.00004 0.00662
0.08283 0.08283 0.08283 0.08283 0.35376 0.00201 0.02646 0.00544 0.00004 0.02263
0.09406 0.09406 0.09406 0.09406 0.06143 0.00733 0.01685 0.01645 0.00064 0.02582
0.08515 0.08515 0.08515 0.08515 0.07698 0.09175 0.07097 0.06762 0.00011 0.08368
0.08160 0.08160 0.08160 0.08160 0.09371 0.08702 0.06739 0.06646 0.00008 0.07739
```

```

0.08086 0.08086 0.08086 0.08086 0.08141 0.08654 0.06925 0.06538 0.00009 0.07740
0.08238 0.08238 0.08238 0.08238 0.08320 0.08203 0.06751 0.05395 0.00011 0.07746
0.08331 0.08331 0.08331 0.08331 0.09420 0.01711 0.07477 0.01461 0.00009 0.01935
0.00067 0.00067 0.00067 0.00067 0.00054 0.00464 0.00740 0.00678 0.00006 0.00690
0.05967 0.05967 0.05967 0.05967 0.05408 0.04822 0.02270 0.01925 0.00017 0.01865
0.00821 0.00821 0.00821 0.00821 0.01031 0.00618 0.00623 0.00397 0.00011 0.00738
0.00043 0.00043 0.00043 0.00043 0.00034 0.00124 0.00145 0.00173 0.00002 0.00059
0.01581 0.01581 0.01581 0.01581 0.00875 0.10256 0.01323 0.00799 0.00025 0.01145
0.02112 0.02112 0.02112 0.02112 0.02389 0.04246 0.03516 0.00786 0.00013 0.01465
0.07207 0.07207 0.07207 0.07207 0.06970 0.08270 0.07019 0.04924 0.00005 0.10050
0.07667 0.07667 0.07667 0.07667 0.07584 0.08081 0.06841 0.03319 0.00014 0.04053
0.14145 0.14145 0.14145 0.14145 0.14506 0.01002 0.00841 0.00897 0.00012 0.01248
0.04574 0.04574 0.04574 0.04574 0.04843 0.08112 0.03185 0.02851 0.00021 0.02790
0.03739 0.03739 0.03739 0.03739 0.04921 0.01000 0.02035 0.01185 0.00019 0.01653
0.08336 0.08336 0.08336 0.08336 0.08897 0.07353 0.06346 0.06354 0.00003 0.04449
0.07581 0.07581 0.07581 0.07581 0.08182 0.07966 0.05424 0.06289 0.00007 0.07054
0.07746 0.07746 0.07746 0.07746 0.08965 0.04561 0.05949 0.03563 0.00009 0.04247
0.04765 0.04765 0.04765 0.04765 0.04967 0.02692 0.02058 0.02111 0.00016 0.01419
0.00489 0.00489 0.00489 0.00489 0.00411 0.00745 0.00795 0.00733 0.00017 0.00929
0.08402 0.08402 0.08402 0.08402 0.11465 0.08444 0.06428 0.06551 0.00008 0.07433
0.07875 0.07875 0.07875 0.07875 0.07826 0.08018 0.07052 0.06623 0.00015 0.07777
0.04826 0.04826 0.04826 0.04826 0.04245 0.00867 0.00918 0.00758 0.00008 0.01028
0.06598 0.06598 0.06598 0.06598 0.07010 0.05898 0.07703 0.01556 0.00707 0.02343
0.02430 0.02430 0.02430 0.02430 0.02547 0.02539 0.00705 0.00708 0.00018 0.01004
0.01002 0.01002 0.01002 0.01002 0.01255 0.00805 0.00822 0.00735 0.00029 0.01300
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000

```

Pour notre exemple de mesure de la pauvreté -

```

EPMCO      18      # Experimental poverty measure cutoffs
              5600.0
              6210.0
              6530.0
              7050.0

```

RÉSUMÉ/CONCLUSION

Il est convenu de conclure en soulignant, mais sans les reprendre en détail, les points clés en général pertinents à l'ajout de paramètres scalaires inhabituels et de paramètres non scalaires à un modèle. En notant ces points, nous supposons que l'analyste suit les procédures générales décrites pour les paramètres scalaires. Par exemple, on suppose que l'analyste travaille avec des COPIES de tous les fichiers pertinents et qu'il apporte toutes les modifications dans le sous-répertoire de travail réservé à l'analyse en cours. Nous supposons aussi que l'utilisateur a fait la mise à jour de l'environnement de projet et qu'il utilise les listes de vérification appropriées fournies pour les paramètres inhabituels.

1. Nous recommandons la «copie» comme façon générale de procéder à ce travail. Tout au long du chapitre, nous avons fourni des exemples concrets des éléments qu'un utilisateur pourrait utiliser comme gabarit. L'utilisateur devrait rarement avoir besoin d'employer tout le matériel dans `Mpu.h/Cpu.h/Apu.h` (définitions, constantes manifestes pour le nombre maximum de rangs et le nombre réel de rangs) et `Ampd.cpp` (`pmaddent` et `stradd`).
2. L'utilisateur avancé peut désirer connaître les «services» spéciaux auxquels il peut accéder par les arguments de `pmaddent` : la capacité de préciser les formats d'impression, de faire le contrôle de vérification et le nombre maximum de rangs permis ou les options.
3. Les vecteurs peuvent parfois être beaucoup plus efficaces qu'un certain nombre de paramètres scalaires nommés individuellement. Le MSPS offre cette capacité, bien que

l'utilisateur doit fournir de l'information supplémentaire dans l'appel de `pmaddent` et être certain de fournir une autre variable pour le nombre de rangs pertinents, ainsi qu'une constante pour les dimensions. Nous avons offert de nombreux gabarits possibles pour faciliter la méthode de copie.

4. Sous de nombreux aspects, le tableau est un cas spécial de vecteurs, applicable lorsque l'on a besoin de faire une recherche de valeurs `y` comme fonction de valeurs `x`, avec une relation fixe.
5. Les matrices (bidimensionnelles) sont aussi possibles. Il est nécessaire de fournir de l'information supplémentaire, le nombre de colonnes, mais la méthode de la matrice peut être considérablement plus efficace que la manipulation de multiples vecteurs parallèles. Encore une fois, la méthode de copie et modification est recommandée.

Développement en boîte de verre : ajout de nouvelles variables

Le présent chapitre décrit la façon dont on ajoute des variables dépendantes définies par l'utilisateur à une application en boîte de verre du MSPS. Il illustre donc la façon de relever les défis comme ceux qui se posent dans l'exemple de Départ rapide, lorsque l'utilisateur aimerait avoir eu une variable distincte pour l'hypothétique supplément d'allocations familiales. La disponibilité de variables dépendantes définies par l'utilisateur est encore plus importante si l'utilisateur modélise quelque nouveau programme, par exemple un supplément de gains qui ne pourraient pas être combinés de façon commode dans une variable dépendante d'un modèle existant.

Du point de vue de la structure, ce chapitre traite de toutes les grandes étapes et tous les grands problèmes entourant l'ajout de nouvelles variables dépendantes dans un modèle. Il y a aussi dans ce chapitre un aperçu du processus et une section qui définit les grands types de variables que l'utilisateur peut désirer ajouter. Suit un exemple de la fonction critique `vardef` qui établit les liens entre le code source de l'utilisateur et le reste du MSPS et qui décrit aussi comment utiliser la fonction `stradd` pour rendre l'étiquetage des nouvelles variables accessible partout à l'intérieur du MSPS. Nous présentons ensuite un exemple en prolongeant l'exemple de supplément d'allocations familiales utilisé dans les exemples précédents définissant les nouvelles variables qui seront disponibles dans plusieurs fonctions de sortie du MSPS. Il y a ensuite des exemples de modification du code de source que l'utilisateur doit faire et des descriptions de compilation et de validation du modèle obtenu.

APERÇU DE L'AJOUT DE VARIABLES

Dans les grandes lignes, les principales étapes de l'ajout de nouvelles variables sont les suivantes :

1. Décider des nouvelles variables dépendantes nécessaires, choisir les noms appropriés et des descriptions pour elles, et copier tous les fichiers de code de source et d'en-tête pertinents vers un sous-répertoire dans lequel le nouveau modèle sera construit.
2. Apporter les changements pertinents à l'environnement de projet (en identifiant tous les fichiers de code de source appropriés associés aux nouvelles variables dépendantes) et

mettre à jour `Adrv.cpp` (en fournissant les chaînes de texte de documentation).

3. Apporter les changements nécessaires à `vsu.h` et `vsdu.cpp` pour rendre les nouvelles variables dépendantes accessibles partout à l'intérieur du modèle MSPS qui sera créé.
4. Fournir le nouveau code de source (dans des modules nouveaux ou existants) pour le calcul des valeurs des nouvelles variables dépendantes.
5. Compiler le nouveau modèle et le valider afin d'en vérifier la bonne qualité.

Les points ci-dessus sont, bien sûr, un simple aperçu. La section portant sur l'ajout de paramètres et la section de récapitulation fournissent une description beaucoup plus complète du processus de création de modèles dans son ensemble. Le présent chapitre, cependant, porte essentiellement sur les détails particulièrement pertinents à l'ajout de nouvelles variables dépendantes.

CARACTÉRISTIQUES ET TYPES DE VARIABLES DÉPENDANTES

Le MSPS donne à l'utilisateur la possibilité de créer trois types distincts de variables dépendantes définies par l'utilisateur. Il s'agit de trois types de variables scalaires. Le MSPS ne permet pas des variables dépendantes de vecteurs et de matrices. Les types de variables sont les suivants :

1. Analyse numérique -- Il s'agit du type le plus courant de variables dépendantes définies par l'utilisateur. Elle consiste en une valeur numérique (à virgule flottante) qui sera utilisée comme variable d'analyse, par exemple, totalisée comme une entrée de cellule dans le paramètre de commande XTSPEC. Un bon exemple de ce type de variable dépendante est la valeur de quelque prestation nouvelle qui est fonction du revenu et qui pourra être versée à une famille.
2. Analyse à nombre entier -- Moins fréquemment utilisé, ce type de variable dépendante consiste en une valeur entière (int) qui sera utilisée comme variable d'analyse. La principale utilisation de ce type de variable est l'exportation en format SAS, où une variable entière occupe moins de caractères qu'une variable d'analyse numérique. Des exemples de ce type de variable pourraient être les nombres minimum et maximum de semaines qu'une famille pourrait passer sans revenu gagné au cours de l'année (tel que cela été déduit des variables de la population active pour les membres de la famille, p. ex., les semaines sans travail et de recherche de travail).
3. Variable de classe à nombre entier -- ce type de variable dépendante consiste en une valeur entière (int) qui sera utilisée comme variable de classe, p. ex., pour définir les catégories d'une variable de classe dans le paramètre XTSPEC. Ce type de variable est particulièrement intéressant lorsque sa valeur représente les catégories purement nominales, p. ex., une classe de famille par types.

Quelques autres caractéristiques des variables dépendantes définies par l'utilisateur, individuellement et collectivement, auront une importance considérable pour l'utilisateur du mode boîte de verre --

En premier lieu, toutes les variables dépendantes définies par l'utilisateur sont définies au niveau de l'individu. Par conséquent, l'utilisateur doit prendre soin d'attribuer des valeurs aux individus «appropriés» de façon que, lorsque l'unité d'analyse est à un niveau supérieur, disons au niveau de la famille de recensement, les algorithmes de cumul du MSPS donnent les résultats désirés.

En second lieu, l'espace alloué pour ces variables peut recevoir environ 50 variables. Le dépassement de cette limite peut entraîner des erreurs obscures qu'il sera difficile de retrouver.

LES FONCTIONS VARDEF ET STRADD AINSI QUE LEURS ARGUMENTS

Les fonctions vardef et stradd sont absolument essentielles à la capacité de créer de nouvelles variables définies par l'utilisateur et de faire en sorte qu'elles soient utilisées correctement partout dans le reste du MSPS. C'est seulement par l'intermédiaire de l'information communiquée par les appels de ces fonctions que le reste du MSPS connaît la nature des nouvelles variables et du texte de documentation qui les accompagne. Cette section documente en premier la fonction vardef puis passe à la fonction stradd.

La fonction vardef joue le même rôle général pour les variables définies par l'utilisateur que pmaddent joue pour les paramètres définis par l'utilisateur. Il y aura un appel de vardef pour chaque variable que l'utilisateur définit. vardef définit les caractéristiques de la nouvelle variable de façon que le MSPS puisse la lier dans le cadre de travail des variables utilisées par la base de données propre du BD/MSPS, les variables d'analyse et de classe. Les appels de vardef sont toujours faits dans la fonction vsdu.c. La courte description suivante des arguments de la fonction se trouve vers la ligne 100 de cette fonction --

```
*   vardef("_uvew",          <= the name of the variable, quoted, with '_'
*       IN,                  <= home structure (leave at 'IN')
*       im.uv.ew,            <= variable location (always in im.uv)
*       C_INT,               <= C-type (C_INT or C_NUM)
*       V_CLAS               <= type of variable (V_CLAS or V_ANAL)
*       );
```

Nous décrirons la nature des arguments de vardef un à la fois, dans l'ordre. Les sections subséquentes de ce chapitre fournissent des illustrations précises de l'utilisation des deux fonctions vardef et stradd.

Argument «Name» de Vardef (et définition de la souche du nom la variable) :

Le premier argument donne le nom de la variable comme chaîne de texte entre guillemets doubles. L'utilisateur doit toujours inclure un caractère de soulignement comme premier caractère après les guillemets doubles d'ouverture puis les caractères «uv» comme deuxième et troisième caractère pour indiquer le statut «variable utilisateur». Le reste du nom, c'est-à-dire tout ce qu'il y a après le préfixe «_uv», est connu comme la souche de nom de la variable. En termes généraux, la partie souche devrait être aussi informative et mnémonique que possible.

Pour les variables qui ne seront pas exportées à l'extérieur du MSPS, il n'y a aucune limite réelle au nombre de caractères de la souche de nom. Cependant, les variables qui doivent être exportées vers d'autres progiciels sont soumis à certaines contraintes. Par exemple, si la

variable clé doit être exportée vers le SAS, alors la souche ne doit pas dépasser six caractères. Si elle doit être exportée vers le module MAPSIT EXAMINE, alors la partie souche ne devrait pas dépasser dix caractères.

Argument «Home Structure» de Vardef :

Le deuxième argument indique que la structure dans laquelle la nouvelle variable réside. Parce que les variables définies par l'utilisateur sont TOUJOURS définies au niveau de l'individu, l'utilisateur devrait toujours entrer cet argument comme un «IN» (sans guillemets).

Argument «Variable Location» de Vardef :

Le troisième argument indique l'endroit où se trouve la variable (par rapport aux structures de données du MSPS). L'emplacement est spécifié par trois éléments, dont deux sont invariables. Précisément, la première partie de l'emplacement est TOUJOURS égale à «im.uv» (sans guillemets). Cette information indique au MSPS que la nouvelle variable est à l'intérieur de la partie variable utilisateur (uv) de la structure im (variables de modèle du niveau individu). La partie finale de la spécification d'emplacement est la souche de nom de la nouvelle variable, telle qu'elle est définie ci-dessus pour le premier argument.

Argument «Type-C» de vardef (C_NUM & C_INT) :

Le quatrième argument précise le type de variable dans le langage C. Il prend l'une ou l'autre de deux valeurs. Les variables d'analyse numérique utilisent l'entrée «C_NUM» (sans les guillemets). Les variables de classe et d'analyse à nombre entier utilisent la valeur «C_INT» (sans les guillemets).

Argument (Type) «Utilisation» de Vardef (V_ANAL & V_Clas) :

Le cinquième et dernier argument précise si le MSPS doit traiter la variable comme variable d'analyse (totalisable) ou comme variable de classe (de catégorie). Il prendra une des deux valeurs. Les variables d'analyse numérique et à nombre entier utiliseront toutes deux l'entrée «V_ANAL» (sans les guillemets). Les variables de classe à nombre entier utiliseront «V_CLAS» (sans les guillemets) form.

La combinaison des quatrième et cinquième entrées indique au MSPS combien d'octets de mémoire il doit affecter aux variables, un point important à prendre en compte étant donné la limite de 200 octets par variable définie par l'utilisateur. Comme on l'a noté ci-dessus, les besoins sont de six octets pour une variable d'analyse numérique, trois octets pour une variable d'analyse à nombre entier et un octet pour une variable de classe à nombre entier.

Nous avons déjà vu des applications simples de la fonction stradd quand nous avons étudié la documentation des paramètres utilisateur. La même fonction a le même but ici, mais d'une façon plus sophistiquée, puisqu'elle est utilisée pour définir tant une courte description des variables utilisateur elle-même, que, dans le cas spécial des variables de classe à nombre entier et les variables d'analyse à nombre entier, la plage des valeurs et les étiquettes textuelles associées à des valeurs particulières des variables. Le fichier vsdu.cpp contient, vers la ligne 110, une documentation abrégée pour la description de variables et l'étiquette de valeur.

```

*   stradd("uvew",          <= the name of the variable, quoted
*       "Region"           <= a printing label for the variable
*   );
**   stradd("ew",           <= the stem name of the variable, quoted
*       "\tEast\tWest"    <= string containing a label for each valid
*       );                level, preceded by a tab '\t' character.

```

Comme ce fut le cas avec la fonction `vardef` ci-dessus, nous poursuivrons l'analyse des arguments dans l'ordre. Cela se complique ici du fait que le NOMBRE D'APPELS de `stradd` et la structure des arguments de `stradd` dépendent du type de variables pour lesquelles `stradd` est utilisé. Cependant, le nombre d'arguments de `stradd` est toujours de deux. En accordant plus d'importance à la clarté plutôt qu'à la brièveté, nous décrirons chacun des trois types (analyse numérique, analyse à chiffre entier et classe à chiffre entier) individuellement.

Appels de Stradd pour les variables d'analyse numérique :

Les variables d'analyse numérique exigent seulement un appel de la fonction `stradd`. Le premier argument précise le nom de la variable. Il est identique à celui qui est utilisé pour le premier argument de `vardef`, SAUF QUE LE CARACTÈRE DE SOULIGNEMENT DU DÉBUT QUI EST HABITUELLEMENT PRÉSENT À CET ENDROIT EST OMIS DANS CE CAS-CI.

Le deuxième argument pour une variable d'analyse numérique est la chaîne (entre guillemets) que le MSPS utilise lorsqu'il a besoin d'imprimer une description de la variable.

Par exemple --

```
stradd("uvnewben", "New Hypothetical Benefit");
```

Appels de stradd pour les variables d'analyse à nombre entier :

L'ajout de variable d'analyse à nombre entier exige deux appels distincts de `stradd`. Le premier appel définit l'étiquette pour la variable dans son ensemble. Le second appel définit, par un ensemble d'étiquettes pour les valeurs entières individuelles, la plage des valeurs de la variable.

Dans le premier appel (étiquette de variable), le premier argument précise le nom de la variable. Il est identique à celui qui est utilisé pour le premier argument de `vardef`, SAUF QUE LE CARACTÈRE DE SOULIGNEMENT DU DÉBUT QUI EST PRÉSENT À CET ENDROIT EST OMIS DANS CE CAS-CI.

Dans le premier appel (étiquette de variable), le second argument est la chaîne (entre guillemets) que le MSPS utilise lorsqu'il a besoin d'une description de la variable dans son ensemble, par exemple dans la documentation d'une table.

Dans le second appel (étiquette de valeur), le premier argument est la souche de nom pour la variable; le caractère de soulignement et la chaîne de début «uv» ne devraient pas être présents.

Dans le second appel (étiquettes de valeur), le second argument est une chaîne entre guillemets qui indique au MSPS combien de catégories sont pertinentes. La chaîne est formée de la répétition du gabarit «tx», où x varie toujours de 0 au «nombre total de

catégories moins un». Par conséquent, pour une variable ayant quatre catégories, le second argument prendrait la forme «t0\t1\t2\t3». La notation /t est la façon dont le langage C indique normalement un caractère de tabulation.

Par exemple --

```
stradd("uvnputpp", "Number persons unemployed 2+ periods");
stradd("nputpp", "\t0\t1\t2\t3\t4");
```

Appels de stradd pour les variables de classe à chiffre entier :

L'appel de stradd pour des variables de classe à chiffre entier est identique à celui des variables d'analyse à chiffre entier, À UNE EXCEPTION CRITIQUE PRÈS. Dans le second appel (étiquette de valeur), le second argument est une chaîne entre guillemets qui fournit les étiquettes textuelles pour les diverses catégories de la variable. Essentiellement, les nombreuses étiquettes fournies par l'utilisateur correspondent aux chiffres entiers de 0 au «nombre de catégories moins un» d'entrées du second appel de stradd pour une variable d'analyse à nombre entier. Donc, par exemple, les étiquettes pour la documentation de «région» pourraient ressembler à ce qui suit :

```
\tAtlantic\tQuebec\tOntario\tPrairies\tBritish Columbia
```

Ces étiquettes, qui peuvent contenir des espaces vides encastrés (puisque le caractère de tabulation sert de délimiteur) se retrouveraient dans les étiquettes lorsque l'utilisateur utiliserait la capacité de tableaux croisés du MSPS ou exporterait la nouvelle variable vers un fichier SAS.

Par exemple --

```
stradd("uvfamcat", "Nominal Family Income Category");
stradd("famcat", "\tVery Poor\tPoor\tNear Poor\tNon-Poor\tRich");
```

Au-delà des définitions descriptives des arguments de vardef et de stradd, qui se trouvent vers les lignes 100 à 115 de la fonction vsdu.cpp, vsdu.cpp contiennent aussi des combinaisons de gabarits des appels de vardef et de stradd pour les trois types des nouvelles variables. Comme c'est habituellement le cas avec le MSPS, l'utilisateur trouve commode de modifier des copies de ces gabarits lorsqu'il définit de nouvelles variables. Ces gabarits se retrouvent vers les lignes 125 à 145 de vsdu.cpp.

```
* -----
* A numeric variable:
* -----
vardef("_xxxxxxx", IN, im.uv.xxxxxxxx, C_NUM, V_ANAL);
stradd("xxxxxxx", "Variable label");

* -----
* An integer analysis variable, with values 0 through 4:
* -----
vardef("_yyxxxxxx", IN, im.uv.yyxxxxxx, C_INT, V_ANAL);
stradd("yyxxxxxx", "Variable label");
stradd("xxxxxx", "\t0\t1\t2\t3\t4");
* -----
* An integer class variable, with values 0 through 4:
* -----
vardef("_yyxxxxxx", IN, im.uv.yyxxxxxx, C_INT, V_CLAS);
stradd("yyxxxxxx", "Variable label");
stradd("xxxxxx", "\tLABEL0\tLABEL1\tLABEL2\tLABEL3\tLABEL4");
```


L'EXEMPLE DE SUPPLÉMENT D'ALLOCATIONS FAMILIALES, PROLONGÉ

Bien que l'explication ci-dessus de l'ajout de variables définies par l'utilisateur soit complète du point de vue de la définition, il est utile de voir de quoi les opérations subséquentes ont l'air en pratique. Dans la présente section, nous résumons l'exemple concret que le reste des sections expliqueront plus en détail. Essentiellement, l'exemple est un prolongement de l'exemple des allocations familiales présentées comme Départ rapide et amélioré par la suite par l'ajout de paramètre définis par l'utilisateur.

Notre objectif explicite consiste à fournir un exemple perfectionné qui donne des illustrations concrètes des trois types de variables définies par l'utilisateur et de le faire sans inonder le lecteur de toutes les tâches courantes qui seraient inévitablement associées à la préparation d'un exemple entièrement nouveau. Pour atteindre cet objectif, nous n'avons pas hésité à sacrifier quelque peu le réalisme (quant aux pratiques et à la motivation institutionnelle) au profit d'un exemple précis et clair.

Nous prolongeons l'exemple du supplément d'allocations familiales en ajoutant trois variables définies par l'utilisateur, comme suit :

1. Une variable d'analyse numérique : la nouvelle variable est le montant brut des prestations d'allocations familiales supplémentaires reçues; nous l'appellerons «uvfasup» (variable utilisateur, supplément d'allocations familiales). Nous affectons cette variable aux parents qui déclarent la prestation d'allocations familiales aux fins d'impôt.
2. Une variable d'analyse à nombre entier : la nouvelle variable est le nombre d'enfants au titre desquels la prestation supplémentaire est payable. Nous appellerons la variable «uvncfasup» (variable utilisateur, nombre d'enfants pour le supplément d'allocations familiales). Nous affectons aussi cette variable à la personne déclarant les allocations familiales aux fins d'impôt. Ce type de variable trouve sa plus grande utilisation lorsqu'elle est exportée en format SAS parce qu'elle prend moins d'espace que la variable d'analyse numérique. La variable serait aussi utile comme variable totalisée pour le compte du nombre de ces enfants.
3. Une variable de classe à chiffre entier : la nouvelle variable établit la catégorie de famille selon le nombre d'enfants au titre desquels le supplément est payable; nous nommons la variable «uvfclfasup» (variable utilisateur, classification de la famille aux fins du supplément d'allocations familiales). Nous l'utiliserons principalement comme variable de catégorie pour les tables visant à valider notre prolongement du code de supplément d'allocations familiales. Nous affecterons cette variable au chef de famille nominal. Il faut noter que cette variable de classe est très semblable à la variable d'analyse à chiffre entier, mais qu'elle peut être utilisée directement comme variable de rang ou de colonne dans un tableau croisé, tandis que la variable d'analyse à nombre entier ne le peut pas.

Au moment d'apporter les changements et de rédiger le code nécessaire à la mise en œuvre de ces nouvelles variables définies par l'utilisateur, nous supposons que les fichiers pertinents (`Adrv.cpp`, `vsu.h`, `vsdu.cpp`, `Afamod.cpp`, `SPSMGL.dsw`, etc.) ont été COPIÉS dans un nouveau sous-répertoire approprié; nous supposerons aussi qu'il

s'appelle GLASSEX3, puisqu'il s'agit de notre troisième exemple en boîte de verre.

MODIFICATIONS DES FICHIERS DE PROJET ET `ADRV.CPP`

Commençons par inclure tous les fichiers pertinents dans le projet et changer le nom du fichier exécutable dans Project: Setting: Links to `glassex3.exe`.

Les modifications à apporter au fichier `Adrv.cpp` sont simples et ne comprennent en tout que a) la mise à jour de courtes descriptions textuelles pour le modèle et b) l'indication que `Afamod` (plutôt que `famod`) doit être utilisé pour le calcul des allocations familiales.

Des deux descriptions, le MSPS affiche la première à son écran d'ouverture, pour indiquer à l'utilisateur la nature du nouveau système. Le MSPS produit la seconde description comme élément de la documentation de «`.CPR`» (paramètre de commande) qu'il produit lorsqu'il exécute le modèle. Rappelez-vous que la mise en place du texte (à l'écran et dans le fichier de sortie) empêche l'utilisateur de reproduire des descriptions plus longues que 20 caractères. Après l'ajout des nouvelles descriptions, la partie pertinente du fichier `Adrv.cpp` (vers la ligne 35) ressemble à ce qui suit :

```
===== GLOBAL VARIABLE DEFINITIONS ===== */
/*global*/ char ALTNAME[IDSIZE+1] = "FA Suppl New Vars Ex";
/* Give global string describing version of this module */
/*global*/ char FAR Tdrv[] = "FA Suppl New Vars Ex"
#ifdef MSC
" [ " __TIMESTAMP__ " ] "
#endif
;
```

La ligne modifiée (vers la ligne 106) pour indiquer que le pilote de rechange utilise `Afamod.cpp`, plutôt que `famod.cpp`, ressemble à --

```
Afamod(hh);          /* compute family allowances
```

Enfin, il faut compiler une version Debug dans Build:Start:Debug. Les liens requis et les compilations sont identifiés.

MODIFICATIONS DE `VSU.H`

Le fichier `vsu.h` sert à définir la structure en langage C qui contient les variables définies par l'utilisateur. La partie pertinente du fichier, copiée du sous-répertoire `SPSM\GLASS`, ressemble à ce qui suit :

```
typedef struct uv_ {
    NUMBER    uvdummy;          /* dummy variable */
} uv_;
```

Nous remplaçons la ligne `uvdummy` par trois lignes qui définissent nos nouvelles variables, `uvifasup`, `uvncfasup` et `uvfclfasup`. Ces nouvelles lignes indiquent le type des nouvelles variables. Après les changements, la nouvelle partie de `vsu.h` ressemble à ce qui suit :

```
typedef struct uv_ {
    NUMBER    uvfasup;          /* Family Allowance supplement payable */
    int        uvncfasup;       /* Number Children for FA supplement */
}
```

```

    int      uvfclfasup; /* Family Class (Qualifying Children) for FA suppl */
} uv_;

```

Il faut noter les règles d'attribution de nom au fichier que l'on a utilisé ici. L'instruction `typedef` exige que les variables soient précédées du préfixe `uv`, mais N'UTILISE PAS le caractère de soulignement de début qui est utilisé dans les instructions `vardef` se retrouvant plus tard dans les modifications apportées à `vsdu.cpp`.

Il n'est pas toujours nécessaire de modifier, comme nous l'avons fait ici, la version GLASS de `vsu.h`. S'il y a déjà une version (définie par l'utilisateur) de `vsu.h` contenant des variables définies par l'utilisateur qui peuvent être conservées, il suffit de faire une copie du fichier existant et de le modifier au besoin. Rappelez-vous cependant qu'il y a une limite globale de 200 octets par individu pour les variables définies par l'utilisateur.

MODIFICATIONS DU FICHIER `VSDU.CPP`

Les modifications nécessaires à la copie de `vsdu.cpp` consistent dans les appels de `vardef` et de `stradd` pour permettre au MSPS d'accéder aux nouvelles variables et à leur documentation. Étant donné la simplicité de ces appels, nous utilisons les gabarits d'exemple du début du fichier. Nous ferons ces appels à la fin du fichier `vsdu.cpp`, juste avant l'instruction «`DEBUG_OFF("vsdu");`». Les ajouts sont donnés ci-dessous :

```

/* uvfasup: (Analysis) Family Allowance supplement payable */
vardef("_uvfasup", IN, im.uv.uvfasup, C_NUM, V_ANAL);
stradd("uvfasup", "Family Allowance Supplement");
/* unvcfasup: (Analysis) number of children for whom supplement paid */
vardef("_unvcfasup", IN, im.uv.unvcfasup, C_INT, V_ANAL);
stradd("unvcfasup", "# Children for FA Supplement");
stradd("ncfasup", "\t0\t1\t2\t3\t4\t5\t6\t7");
/* uvfclfasup: (Class) Family class by number of children for FA suppl. */
vardef("_uvfclfasup", IN, im.uv.uvfclfasup, C_INT, V_CLAS);
stradd("uvfclfasup", "Family Class for FA Supplement");
stradd("fclfasup", "\t0 Ch\t1 Ch\t2 Ch\t3 Ch\t4 Ch \t5 Ch\t6 Ch\t7 Ch");

```

Il faut noter le second appel de `stradd` pour chacune des deux variables nombre entier et l'omission du préfixe `uv` dans cet appel (le second) qui définit le nombre de cas (variable d'analyse à nombre entier) ou les étiquettes de catégorie (variable de classe à nombre entier).

MODIFICATIONS DU FICHIER `AFAMOD.CPP` (OU, PLUS GÉNÉRALEMENT, TOUT NOUVEAU CODE DE SOURCE DE BASE)

Les tâches ci-dessous préparaient notre tâche centrale, la révision du fichier `Afamod.cpp` afin qu'il reflète le nouveau calcul des allocations familiales, comprenant le supplément possible à la famille. Nous utilisons le fichier `Afamod.cpp` ici, mais, de façon plus générale, à cette étape, l'utilisateur est prêt à rédiger ou à modifier le code de source nécessaire pour apporter les changements désirés au calcul des variables du MSPS, quel que soit le module qui soit visé par ces modifications. Nous illustrerons les changements à notre exemple d'allocations familiales une partie à la fois, illustrant chaque partie en montrant ce dont a l'air le fichier `Afamod.cpp` non modifié puis en montrant comment nous l'avons modifié en ajoutant nos variables. Les références pour lesquelles il y a des numéros de ligne renvoient à la version originale du fichier `Afamod.cpp` qui se trouve dans le sous-répertoire `SPSM\GLASS`.

Identification des chaînes

La documentation est importante. À mesure que nous avançons dans le fichier Afamod.cpp pour faire nos modifications, nous faisons d'abord la mise à jour de la description. Où la version GLASS du fichier Afamod.cpp donne la description fictive (vers la ligne 39) --

```
/*global*/ char FAR Tfa[] = "Untitled"
```

nous entrons une description plus informative :

```
/*global*/ char FAR Tfa[] = "New Vars Version"
```

Variables locales

Les variables intermédiaires (locales) peuvent être très utiles. Où la version GLASS du fichier Afamod.cpp définit et initialise ces variables locales (version la ligne 131), nous ajoutons les nouvelles lignes données ci-dessous. L'initialisation à zéro des variables de nombre à virgule flottante fait en sorte que l'on ne se retrouvera pas avec quelque étrange problème qui pourrait se manifester seulement sur quelques ordinateurs non standard.

```
/* user-defined intermediate (local) variables in support of glass box example 3
(user-defined SPSM variables) [using the "stem names" for two of the SPSM variables
being created] */
    NUMBER fasup = ZERO; /* amount of new FA supplement */
    int    ncfasup;      /* number of children for whom supplement payable */
```

Calcul et affectation des nouvelles variables de modèle

Vous êtes maintenant prêt à calculer les nouvelles variables et à les affecter aux variables MSPS définies par l'utilisateur approprié. Pour notre exemple Afamod.cpp, nous cherchons à calculer le montant du supplément possible. Nous le faisons immédiatement après que les allocations familiales fédérales et imposables ont été définies dans la version SPSM\GLASS du fichier Afamod.cpp, juste avant que ces valeurs aient été affectées à la sortie du sous-programme Afamod. Cette condition se produit vers la ligne 358. Le code source original pertinent ressemble à ce qui suit --

```
        else {
            DEBUG1("%s standard FA calculation\n");
            tfa = nch * MP.STDFA; /* taxable family allowances
*/
            ffa = tfa;           /* federal part of family
allowances*/
        }

        DEBUG3("%s tfa=%.2f, ffa=%.2f\n", tfa, ffa);
```

Dans le nouveau code que nous ajoutons, nous prenons soin de faire en sorte qu'une valeur appropriée soit calculée pour nos variables intermédiaires, quel que soit l'aspect de la famille nucléaire, et que les variables d'allocations familiales fédérales et imposables soient mises à jour si le supplément est pertinent. Il faut noter que nous conservons la structure de paramètres développée à la section 6 du présent guide.

```
/* Conditionally apply the Family Allowance bonus for the
* "FASUPFECTh" and subsequent children <18 in the unit,
```

```

        * including any necessary updates to taxable and federal FA */

if ((MP.UM.FASUPFLAG == 1) && (nch >= MP.UM.FASUPFEC)) {
    ncfasup = (nch-MP.UM.FASUPFEC+1);
    fasup = ncfasup * MP.UM.FASUPPC;
    tfa += fasup;
    ffa += fasup;
}
else {
    ncfasup = 0;
    fasup = ZERO;
}

```

Dans notre exemple de supplément d'allocations familiales, il est logique d'attribuer la valeur de classe de la famille au chef de la famille nucléaire. Nous le faisons à l'endroit où la version SPSM\GLASS du fichier Afamod.cpp (vers la ligne 368) attribue d'autres valeurs au membre le plus âgé. Ce code original du fichier Afamod.cpp ressemble à ce qui suit --

```

/**
 * Associate the taxable amount of family allowances, and the number of
 * family allowance children, with the eldest in the nuclear family.
 * The function txinet will reassign to the spouse if necessary.
 */

nf->nfineld->im.imtfa = tfa;
nf->nfineld->im.imqtfa = qtfa;
nf->nfineld->im.imnfach = (NUMBER) nch;

```

Après notre ajout, le code modifié ressemble à ce qui suit --

```

/**
 * Associate the taxable amount of family allowances, and the number of
 * family allowance children, with the eldest in the nuclear family.
 * The function txinet will reassign to the spouse if necessary.
 */
nf->nfineld->im.imtfa = tfa;
nf->nfineld->im.imqtfa = qtfa;
nf->nfineld->im.imnfach = (NUMBER) nch;
/* assign family classification by number of supplement children to the
nuclear family head */
nf->nfin->im.uv.uvfclfasup = ncfasup;

```

Enfin, bien sûr, nous devons veiller à ce que les variables pour le supplément et le nombre d'enfants recevant un supplément soient attribués à la mère, si possible (ou, en l'absence de la mère, au chef de la famille nucléaire). Le code SPSM\GLASS Afamod.cpp original pertinent ressemble à ce qui suit --

```

/* assign FA to mother if present */
if (nf->nfspoflg && (nf->nfinspo->id.idsex == FEMALE)) {
    DEBUG1("%s spouse is the mother\n");
    in = nf->nfinspo;
}

else {
    DEBUG1("%s head receives FA\n");
    in = nf->nfineld;
}

```

Nos modifications à cet endroit sont minimales. Nous ajoutons seulement de nouvelles lignes pour affecter le montant du supplément et le nombre d'enfants recevant le supplément. Il faut noter que nous affectons les valeurs des variables intermédiaires aux variables définies par l'utilisateur (entièrement qualifiées) que nous avons définies dans `vsu.h` et `vsdu.cpp` ci-dessus. La version modifiée du code de source ressemble à ce qui suit :

```
/* assign FA and the supplement, and # Fa supplement children to the mother when
she is present */
    if (nf->nfspoflg && (nf->nfinspo->id.idsex == FEMALE)) {
        DEBUG1("%s spouse is the mother\n");
        in = nf->nfinspo;
    }
    else {
        DEBUG1("%s head receives FA\n");
        in = nf->nfineld;
    }

    in->im.imffa = ffa;
    in->im.impfa = pfa;
    in->im.imqaafa = qaafa; /* Quebec Availability Supplement */
    in->im.imqnbfa = qnbfa; /* Quebec Newborn Allowance */
    in->im.uv.uvfasup = fasup; /* assign new supplement */
    in->im.uv.uvncfasup = ncfasup; /* assign # of children */
```

Compilation

Nous devrions faire la mise au point du modèle et vérifier s'il travaille correctement, puis compiler le nouveau modèle `GLASSEX3.EXE`.

VALIDATION

Quand la compilation est terminée et que le fichier `GLASSEX3.EXE` existe, l'utilisateur peut le valider pour vérifier si la logique fonctionne tel que prévu. Comme la validation a été illustrée avec passablement de détails à la section 6, nous incluons ici seulement un ensemble représentatif de sorties de tableaux croisés. Dans les opérations de tous les jours, l'utilisateur veillera à l'exactitude du modèle avant de procéder à une exécution de production des tableaux désirés.

La mini-validation consiste ici en un ensemble de tables pour une configuration à un seul paramètre. Elle utilise la version 1986 du MSPS et modélise le système fiscal/de transfert existant en 1986. L'utilisateur configure le fichier de paramètres de commande de façon qu'il utilise `C:\SPSD\BA86.MPR` comme fichier de modèle du système de base. Le système de variante, celui qui utilise la nouvelle logique pour les allocations familiales, s'appelle ici `GLASSX3A.MPR`. Il demande une prestation de 120 \$ par année pour le second enfant et les enfants subséquents de 0 à 17 ans, dans la famille nucléaire. Le XTSPEC pertinent ressemble à ce qui suit :

```
XTSPEC
      NF: uvfc1fasup+ *
          {units,
            imffa: L="New Family Allowance",
            _imffa: L="Base Family Allowance",
            uvfasup: L="New FA Supplement"};
      NF: nfnkids+ *
          {units,
            imffa: L="New Family Allowance",
```

```

_imffa: L="Base Family Allowance",
_imffa-_imffa: L="Family Allowance Increase"};
NF: nftype+ *
{uvfasup: L="New FA Supplement",
immdisp-_immdisp: L="Disposable Income Increase"}

```

La première spécification de table illustre l'utilisation des variables définies par l'utilisateur comme variables de classe et d'analyse. Il faut noter que l'utilisation est la même que si les variables avaient fait partie du MSPS original, même jusqu'à la possibilité d'utiliser le qualificatif «+» pour indiquer l'agrégation dans la dimension des variables de catégorie.

La similarité entre les deux premières tables est intentionnelle; elle illustre que l'on peut utiliser les variables créées pour afficher l'information qui est disponible de façon moins commode à partir de variables du MSPS. En premier, par exemple, l'utilisateur n'a pas à établir une différence entre deux variables pour voir les répercussions du supplément d'allocations familiales avant impôt. En deuxième lieu, l'utilisation de la variable uvfaclfasup plutôt que de la variable nfnkids permet à l'utilisateur de passer rapidement à travers des unités de famille nucléaire qui n'ont pas d'enfants. La troisième table confirme ensuite que le supplément est pris en compte par le reste du système fiscal/de transfert, de façon que, dans l'agrégation, les gains en revenu des familles sont inférieurs aux montants bruts du supplément accordé. Les tables qui sont obtenues, modifiées très légèrement, ressemblent à ce qui suit :

```

SPSD/M (Database 4.00)
Wed Sep 27 08:34:51 1989
Base Description: 1986 actual
[Driver: Version 4.00: 82-89, File: c:\spsd\ba86.mpr]
Variant Description: 1986 actual
[Driver: FA Suppl New Vars Ex, File: glassx3a.mpr]
Sample: 0.0495
AGENAME='Standard adjustment'

```

Table 1U: Selected Quantities for Nuclear Families by Family Class for FA Supplement

Family Class for FA Supplement	Unit Count (000)	New Family Allowance (M)	Base Family Allowance (M)	New FA Supplement (M)
0 Ch	10621.5	564.1	564.1	0.0
1 Ch	1196.5	1020.3	876.7	143.6
2 Ch	521.8	758.2	633.0	125.2
3 Ch	81.1	160.3	131.1	29.2
4 Ch	14.6	34.6	27.6	7.0
5 Ch	1.5	4.3	3.4	0.9
6 Ch	0.0	0.0	0.0	0.0
7 Ch	0.0	0.0	0.0	0.0
All	12437.1	2541.7	2235.8	305.9

Table 2U: Selected Quantities for Nuclear Families by Number of children in nuclear family

Number of children in nuclear family	Unit Count (000)	New Family Allowance (M)	Base Family Allowance (M)	Family Allowance Increase (M)
0	9042.2	0.0	0.0	0.0
1	1579.4	564.1	564.1	0.0
2	1196.5	1020.3	876.7	143.6
3	521.8	758.2	633.0	125.2
4	81.1	160.3	131.1	29.2
5	14.6	34.6	27.6	7.0
6	1.5	4.3	3.4	0.9
7	0.0	0.0	0.0	0.0
8	0.0	0.0	0.0	0.0
9	0.0	0.0	0.0	0.0

Table 3U: Selected Quantities for Nuclear Families by Nuclear family type

Nuclear family type	New FA Supplement (M)	Disposable Income Increase (M)
With Kids, 1 Adult	22.2	17.8
With Kids, 2+ Adult	283.7	179.3
With Elderly, 1 Adult	0.0	0.0
With Elderly, 2+ Adult	0.0	0.0
Other, 1 Adult	0.0	0.0
Other, 2+ Adult	0.0	0.0
All	305.9	197.0

Il faut noter que la première et la deuxième tables sont exactement semblables, sauf que la première est légèrement plus compacte (avec moins de lignes), relativement mieux étiquetées et légèrement plus facile à préciser dans XTSPEC. Fondamentalement, cependant, les deux sont comparables; les rangs de «1 Ch» à «7 Ch» de la première table contiennent exactement la même information que les rangs «2» à «8» de la deuxième table. La similitude du contenu est exactement ce à quoi on s'attendait pour une option qui donne le supplément au deuxième enfant et aux enfants subséquents. Le premier et le deuxième enfant de la deuxième table, qui fait la totalisation des familles non admissibles à tout supplément, tient en une seule ligne dans la première table.

La troisième table indique qu'une certaine partie des allocations familiales est recouvrée, puisque l'augmentation du revenu disponible est inférieure au montant complet du nouveau supplément. En outre, la fraction «recouvrée» par les réactions à d'autres programmes du système fiscal/de transfert est, comme on s'y attendait, plus grande pour les unités comptant deux parents que pour les unités monoparentales.

Une fois la validation terminée, l'utilisateur procède à la production des tables désirées et

d'autres sorties.

RÉSUMÉ/CONCLUSIONS

Nous résumons les points clés de ce chapitre en donnant une liste des principaux éléments requis pour l'ajout de nouvelles variables définies par l'utilisateur à un modèle MSPS.

1. Planifier les modifications désirées «sur papier». Choisissez les noms des nouvelles variables et élaborer la logique selon laquelle elles seront calculées. Vérifiez quels sont les fichiers de code source essentiels qui seront touchés en particulier (p. ex., `Afamod.cpp`). Choisissez un sous-répertoire pour un nouveau modèle et créez-le au besoin.
2. Copier tous les fichiers pertinents dans le sous-répertoire où le travail se fera.
 - Les fichiers `SPSMGL.dsw`, `Adrv.cpp`, `vsu.h` et `vsdu.cpp` seront toujours nécessaires, avec les fichiers essentiels pertinents, p. ex., le fichier `Afamod.cpp` de notre exemple.
 - Les fichiers `mpu.h` et `ampd.cpp` peuvent aussi être nécessaires selon que les paramètres doivent être ajoutés au même moment.
3. Faire la mise à jour du projet et modifier le nom du fichier de sortie.
4. Faire la mise à jour du fichier `Adrv.cpp`.
 - Insérez de brèves descriptions pertinentes des deux arguments sous forme de deux chaînes de documentation (`ALTNAME` et `Tdrv`).
 - Modifiez les appels de fonction de façon qu'elles visent les deux versions nouvelles des fonctions de calcul d'impôt/de transfert, p. ex., `Afamod(hh)` plutôt que `famod(hh)`.
5. Faire la mise à jour de `vsu.h`. À l'intérieur de la structure `'uv_'` indiquez le type et le nom des nouvelles variables définies par l'utilisateur. N'oubliez pas d'utiliser le préfixe `'uv'` sans utiliser le caractère de soulignement.
6. Faire la mise à jour du fichier `vsdu.cpp`.
 - Pour chaque nouvelle variable, fournissez un appel de fonction `vardef` afin de définir la nature de la variable au MSPS.
 - Pour chaque nouvelle variable définie par l'utilisateur, faites l'appel de `stradd` afin de fournir une description de variable (chaîne de texte) pour la variable.
 - Pour chaque variable à nombre entier, que ce soit d'analyse ou de classification, faites l'appel de `stradd` une seconde fois (en utilisant juste la souche de nom) pour fournir une liste des étiquettes pour les valeurs entières de la variable. Rappelez-vous que, pour les variables d'analyse, elle indique seulement le nombre de catégories (de 0 à `n`), tandis que, pour les variables définies par l'utilisateur, de classe à nombre entier, l'étiquette est constituée du texte choisi par l'utilisateur.
7. Faire les modifications nécessaires au sous-programme fiscal/de transfert de base. Songez à utiliser des variables intermédiaires pour simplifier les choses. Prenez soin de

faire les initialisations appropriées et d'attribuer les valeurs calculées à un individu approprié.

8. Compiler le nouveau modèle. N'oubliez pas de le valider avant de l'utiliser pour tout travail de production sérieux.

Modification des variables de données de base et de variante

Le présent chapitre décrit la façon dont les utilisateurs peuvent, lorsque cela convient, modifier les valeurs dans la base de données du BD/MSPS, pour l'analyse de choix de politiques. Ces modifications contrastent avec les modifications aux variables dépendantes, aux paramètres et à la logique du modèle décrits dans les chapitres précédents. Ici, nous étudions la modification des données utilisées comme entrées par les algorithmes fiscaux/de transfert plutôt que de la logique de ces algorithmes. Les genres de changements étudiés ici sont temporaires. Ils affectent la valeur «vue» par le modèle de l'utilisateur dans une utilisation en particulier, mais elles ne touchent pas les valeurs réellement stockées dans la BDSPS elle-même.

Habituellement, mais pas toujours, les modifications de la base de données par l'utilisateur toucheront des montants exprimés en dollars, que ce soit des éléments de revenu ou de déduction. L'utilisateur peut désirer accroître ou réduire le revenu d'une source en particulier, par exemple la réduction du revenu d'intérêts, afin de refléter une hypothèse de chute des taux d'intérêt. Cependant, l'utilisateur peut aussi désirer modifier une variable autre que celle du revenu, par exemple, la variable de la fréquentation scolaire pour les enfants plus âgés dans certaines familles.

Pour les modèles du MSPS qui simulent deux systèmes fiscaux/de transfert (base et variante), une distinction importante réside dans le fait que les changements affectent les valeurs telles qu'elles sont «vues» par le modèle entier de l'utilisateur, ou un seul des systèmes (base ou variante) du modèle. Cette distinction est si importante que nous avons organisé la structure du chapitre en fonction d'elle. Il faut cependant noter que la distinction ne vaut rien pour les modèles qui simulent seulement un système fiscal/de transfert. La procédure recommandée ici encourage l'utilisateur à appliquer l'approche du système unique quand cela est possible.

La section suivante décrit la façon de modifier les données juste après que le MSPS les a lues pour l'exécution d'un modèle. Les changements qui y sont discutés auront naturellement un effet sur TOUS les systèmes fiscal/de transfert qu'il y a dans le modèle. La section décrit deux cas auxiliaires -- dans le premier cas auxiliaire, l'utilisateur modifie les données par l'intermédiaire de la fonction de vieillissement des données intégrée au MSPS. Dans le second cas auxiliaire, plus exigeant, l'utilisateur produit sa propre logique de vieillissement. Ce second cas auxiliaire peut exiger la définition de nouveaux paramètres de vieillissement de données pour le modèle. La première section indique où et comment faire les modifications pour un système unique et fournit un exemple pratique détaillé.

La section suivante, par contre, décrit les changements qui touchent seulement un système (base ou variante) à l'intérieur d'une exécution du MSPS. Elle explique comment l'utilisation de la fonction «fichier de résultats» du MSPS peut souvent faire de ce cas le

«système unique» le plus simple comme on l'a décrit ci-dessus. Cependant, pour les cas dans lesquels l'approche du fichier de résultats est impossible ou ne convient pas, cette section comprend aussi une description de l'endroit et de la manière dont les changements nécessaires sont apportés. Elle se termine par un exemple pratique de la façon de mettre en oeuvre des modifications de bases de données propres au système.

MODIFICATIONS QUI TOUCHENT TOUS LES SYSTÈMES FISCAUX/DE TRANSFERT D'UN MODÈLE

Cette section décrit la façon de faire des modifications aux données qui touchent tous les systèmes fiscaux/de transfert d'un modèle du MSPS. La méthode convient dans les deux cas où le modèle ne compte qu'un système fiscal/de transfert et quand le modèle a deux systèmes, mais que l'utilisateur désire que les changements apportés aux données touchent les deux systèmes.

La présente section examine les fonctions de vieillissement des données intégrées au MSPS. Selon cette méthode, l'utilisateur affecte des valeurs à des paramètres de vieillissement existants par des fichiers API (inclusion de paramètres de vieillissement).

Cette partie est suivie de l'ajout d'algorithmes d'ajustement de nouvelles données. Pour ce type de vieillissement, l'utilisateur définit la nouvelle logique de vieillissement dans le fichier `adju.cpp` et définit probablement de nouveaux paramètres par des modifications aux fichiers `apu.h` et `apdu.cpp`. L'utilisateur peut aussi désirer définir de nouvelles variables indépendantes pour aider à la validation du modèle.

Enfin, un exemple pratique détaillé pour ce second cas auxiliaire est présenté à la suite et il est suivi d'une liste de vérification touchant ce type de modification de vieillissement «global» des données.

Modification typique de la croissance du revenu et de la population par les fichiers APR et API

La conception du BD/MSPS prévoit déjà les besoins de l'utilisateur en ce qui a trait au vieillissement typique des données. Le sous-répertoire `\SPSD` contient un certain nombre de fichiers dont le nom a la forme `BAxx_yy.APR` qui indique au MSPS de faire vieillir les données, autres que la structure démographique sous-jacente, depuis l'année `XX` jusqu'à l'année `YY`. Par conséquent, le fichier `BA86_88.APR` contient les paramètres de vieillissement qui permettent de faire vieillir les variables non démographiques de la BDSPS de 1986 à 1988. Le niveau de détail de ce vieillissement est considérable. Chacun de ces fichiers contient plus de 600 paramètres qui sont utilisés par les algorithmes de vieillissement intégrés au MSPS.

Si la substance des paramètres de ces fichiers est acceptable pour les besoins de l'utilisateur, alors le vieillissement des données est simple. L'utilisateur entre le nom du fichier qui est «le plus proche de ses besoins» comme paramètre `INAPR` du fichier de paramètres de commande. Toutes les modifications nécessaires à la valeur de ces paramètres sont alors apportées par l'intermédiaire d'un fichier «API» (inclusion de paramètres de vieillissement).

Le *Guide des paramètres* fournit une description exacte de ces paramètres. Cependant, il convient d'expliquer dans les grandes lignes le contrôle étendu qu'ils assurent.

Certains paramètres précisent comment les revenus imputés/convertis doivent être traités (c.-à-d. non pris en compte ou adoptés selon l'une des deux méthodes de synthèse). De très nombreux paramètres régissent l'«élimination» des taxes à la consommation des dépenses de la famille.

Un autre ensemble de paramètre fournit les seuils de pauvreté pour les familles. Il permet à l'utilisateur de préciser un ensemble de «seuils de pauvreté» pour les familles économiques, les seuils particuliers variant selon la grosseur de la famille et la grandeur du lieu de résidence. Probablement des plus utiles pour l'utilisateur habituel, cependant, il y a le vaste ensemble de facteurs de croissance pour les variables de la BDSPP contenant des valeurs exprimées en dollars : revenus, réductions et dépenses. Pratiquement toute variable de ce genre a son propre facteur de croissance.

Le BD/MSPS fournit le vieillissement démographique commode de sa population sous-jacente. Le fichier «.WGT» du répertoire SPSP donne à l'utilisateur la possibilité d'ajuster la base de population sur tout l'intervalle de 1984 à 1991.

Modifications exigeant une nouvelle logique pour le fichier `adju.cpp`

La souplesse permise par les paramètres de vieillissement (".APR" et ".API") et les fichiers (".WGT") de vieillissement de la population suffisent souvent pour combler les besoins de l'utilisateur. Cependant, dans certains cas, l'utilisateur peut désirer un contrôle plus direct sur les données à utiliser pour une simulation, ou avoir besoin de ce contrôle plus direct. Quelques exemples indiquent dans quelle mesure cela est possible. Le lecteur devrait comprendre que les exemples donnés ici sont axés plus directement sur l'atteinte plus rapide de cette pratique que sur le maintien d'un réalisme strict des politiques.

1. L'utilisateur pourrait augmenter le niveau d'éducation moyen en ajustant la variable «idedlev» pour certains individus, ce qui entraînerait peut-être une distribution des niveaux d'éducation atteints se situant dans l'esprit de quelque prévision exogène.
2. L'utilisateur pourrait désirer accroître quelque montant de revenu ou de transfert avec un facteur qui est fonction des caractéristiques de l'unité. Par exemple, en se basant sur l'hypothèse que le portefeuille des investisseurs diffère en fonction de l'âge et du revenu de l'investisseur, un utilisateur pourrait ne pas désirer modéliser l'effet d'une augmentation des taux d'intérêt en haussant le revenu d'intérêt de chacun avec la même proportion. Plutôt, un facteur plus petit pourrait être appliqué aux individus que l'on croirait susceptibles d'être conservateurs et (ou) dont le portefeuille se renouvellerait plus lentement. Ce genre d'hypothèse traiterait ces familles comme ne pouvant pas bénéficier aussi rapidement des taux d'intérêt plus élevés.
3. Un utilisateur pourrait désirer modéliser une participation plus grande de la population active en changeant le tableau pertinent de variables de la population active pour les individus de la BDSPP (variables de semaine de travail, de gains d'emploi rémunéré, d'assurance-chômage, etc.). Les changements dans une si grande variété de variables

connexes ne se feraient qu'après une planification considérable et exhaustive.

4. À l'extrême, un utilisateur expérimenté et bien informé du MSPS pourrait même modifier la structure du ménage ou de la famille de la BDSPS, modélisant une hausse spectaculaire des naissances en ajoutant des «enfants synthétiques» aux familles appropriées de la base de données.

La fonction `adju.cpp`, qui se trouve dans le sous-répertoire `\SPSM\GLASS`, est le moyen par lequel l'utilisateur peut ajouter une nouvelle logique du vieillissement des données aux modèles du MSPS. La fonction `adju.cpp` est appelée immédiatement après que le MSPS a lu chaque ménage et avant le calcul de toute variable de transfert ou pour mémoire. L'utilisateur peut indiquer la logique de son propre changement immédiatement après l'invocation «`adj(hh)`» que le MSPS utilise pour effectuer son propre vieillissement des données, c'est-à-dire son application intégrée des paramètres de croissance de revenu spécifiés dans les fichiers «`.APR`» et «`.API`».

Pour la mise en oeuvre d'une nouvelle logique de vieillissement de données, l'utilisateur peut avoir besoin de définir de nouvelles variables intermédiaires (y compris des variables de compteur, de pointeur, etc.) et (ou) de définir de nouveaux paramètres de vieillissement de données personnalisés. La sous-section suivante décrit la méthode générale d'ajout de tels paramètres de vieillissement de données, avec les changements spécifiques connexes développés dans l'exemple pratique qui le suit.

Ajout de nouveaux paramètres d'ajustement de la base de données

L'ajout de nouveaux paramètres de base de donnée définis par l'utilisateur s'apparente étroitement à celui de nouveaux paramètres de modèle comme le décrivent les chapitres précédents. Cependant, il y a quelques légères différences.

1) Les modèles MSPS ont un seul fichier de paramètre de vieillissement (extension `".APR"`); ils peuvent avoir un ou deux fichiers de paramètres de modèle (extension `".MPR"`), selon qu'ils ont un ou deux systèmes de transfert. 2) Par conséquent, l'utilisateur fournit les valeurs des paramètres de vieillissement définis par l'utilisateur dans les fichiers `".API"` (inclusion de paramètres de vieillissement) qui modifient les fichiers `".APR"` standard, plutôt que par les fichiers `".MPI"` (inclusion de paramètres de modèle) qui modifient les fichiers `".MPR"` standard. 3) De nouveaux paramètres de vieillissement sont définis dans le fichier `apu.h` (en-tête) plutôt que dans le fichier d'en-tête `mpu.h` utilisé pour les paramètres de modèle. 4) De même, les appels de fonction qui rendent les paramètres disponibles au reste du modèle se produisent dans `apdu.cpp`, plutôt que dans le fichier `ampd.cpp` utilisé pour les paramètres de modèle. Cependant, les structures des appels pertinents de `pmaddent` et de `stradd` sont exactement identiques. Il faut par conséquent noter que certains arguments de ces fonctions diffèrent selon qu'il s'agit de paramètres de vieillissement ou de paramètres de modèle. L'exemple pratique souligne ces différences. 5) Enfin, les changements apportés à la logique elle-même sont définis dans `adju.cpp`, plutôt que les fonctions fiscales/de transfert individuelles, comme `Afamod.cpp`, qui touchent les modifications dans la logique de calcul des transferts d'un modèle.

Nous notons en passant que les paramètres de commande du MSPS suivent une structure parallèle similaire, mais, même pour les applications en boîte de verre, l'utilisateur n'a pas besoin de définir les nouveaux paramètres de commande. Plutôt, il lui suffit de modifier les valeurs des paramètres de commande existants.

Un exemple pratique

Notre utilisateur hypothétique, cherchant à refléter une réponse à un certain changement dans le traitement de l'impôt sur le revenu par le fédéral, désire accroître les cotisations aux REER dans un modèle. Il désire que la croissance s'applique soit à un système simple à analyser, soit aux deux systèmes, base et variante, d'un modèle comparatif. Cependant, cet utilisateur ne désire pas supposer que les cotisations de chacun croissent au même rythme et désire simuler une croissance non proportionnelle au revenu. L'objet principal du modèle est supposé se trouver ailleurs à l'intérieur du système fiscal/de transfert. C'est-à-dire que l'utilisateur ne porte pas un intérêt particulier aux répercussions des augmentations des REER en eux-mêmes. L'utilisateur désire plutôt obtenir tout simplement de meilleures représentations des montants de déductions à utiliser dans tous les calculs pour les systèmes de transfert pertinents.

Pour préciser plus encore l'exemple, supposons que l'utilisateur désire accroître les cotisations existantes de $x\%$ pour chaque tranche (entière ou partielle) de 10 000 \$ de gains d'emploi rémunéré et de gains d'emploi autonomes sur un montant de base initial de 20 000 \$. Par conséquent, un individu ayant 45 000 \$ de gains verrait ses cotisations au REER s'accroître d'un facteur $(1,0 + 3x)$, où x est le nouveau paramètre défini par l'utilisateur. On obtient une croissance additionnée et composée avec la croissance induite par le paramètre de croissance standard des cotisations aux REER, GFRRSP.

Dans une possibilité QUI N'EST PAS développée ici, l'utilisateur pourrait aussi avoir induit la présence des cotisations au REER pour des individus qui ont déclaré 0 \$ comme cotisations. L'exemple développé plus tard, à la section 9.2, illustre ce genre de synthèse des montants exprimés en dollars.

Dans le reste de la présente sous-section, nous suivons pas à pas la méthode utilisée pour la mise en oeuvre de cette croissance conditionnelle (au-delà de la croissance mise en oeuvre par le paramètre de croissance GFRRSP). Nous supposons que l'utilisateur a créé le sous-répertoire GLASSEX4 pour cet exemple et y a COPIÉ tous les fichiers pertinents (SPSMGL.dsw, apu.h, apdu.cpp et adju.cpp, en plus des fichiers de paramètres MSPS pertinents à l'exécution du nouveau modèle). Dans ce répertoire, l'utilisateur crée un fichier « .API » pour fournir une valeur pour le nouveau paramètre défini par l'utilisateur.

Parce que le processus d'ajout de paramètres pour les paramètres de vieillissement est si étroitement semblable à la procédure décrite dans les chapitres précédents, et pour les paramètres de modèle, notre commentaire sur ces modifications sera conservé au minimum. L'utilisateur est supposé avoir modifié le projet de façon à inclure tous les fichiers pertinents et avoir modifié le nom de sortie de la compilation de façon à lui donner le nom GLASSEX4.EXE

Nous incluons la documentation de vieillissement dans la chaîne appropriée définie dans

adju.cpp tel qu'on le décrit ci-dessous.

(A) Modifications du fichier apu.h

Nous commençons par définir un paramètre défini par l'utilisateur pour la structure de croissance des cotisations au REER définies par l'utilisateur, facteur «x» de la description ci-dessus. Comme moyen mnémorique, UDGFRSP (facteur de croissance défini par l'utilisateur, cotisations au REER) semble convenir. Le MSPS fournit jusqu'à 100 octets de paramètres de vieillissement définis par l'utilisateur, avec l'affectation indépendante de 600 octets alloués pour tous les paramètres de modèle que l'utilisateur peut désirer définir.

Les ajouts à apu.h indiquent le genre de paramètre que l'on est à définir. Ils vont juste avant la spécification de prototype de fonction, remplaçant la valeur fictive du paramètre de vieillissement utilisateur UADUMMY, dans le code apu.h.

```
typedef struct UA_ {  
int UADUMMY;      /* dummy entry */  
}  
UA_;
```

Dans notre exemple, nous remplaçons la ligne UADUMMY par --

```
NUMBER UDGFRSP;      /* User-defined growth factor for RRSP Contr. */
```

(B) Modifications du fichier apdu.cpp

Dans la fonction apdu.cpp, nous ajoutons des appels des fonctions pmaddent et stradd afin de donner un accès plus large du MSPS aux valeurs de ce nouveau paramètre. Les détails de ces fonctions se retrouvent dans les chapitres précédents. Nous faisons les ajouts à la fin de la fonction apdu.cpp, juste avant la déclaration --

```
DEBUG_OFF("apdu");
```

Nos deux appels ressemblent à ce qui suit :

```
pmaddent(pap, "UDGFRSP", (char *)&AP.UA.UDGFRSP, NULL, P_SCL, C_NUM, 0, 0, NULL, 0);
```

et

```
stradd("UDGFRSP", "User-defined growth factor for RRSP Contr.");
```

Le texte d'explication au début de la fonction apdu.cpp décrit les arguments AXÉS SUR LE VIEILLISSEMENT pour pmaddent et stradd. Il fournit aussi le gabarit pour ce que nous allons faire ici (un paramètre scalaire).

Il y a deux différences essentielles dans l'utilisation de pmaddent par rapport à la définition des nouveaux paramètres de modèle. 1) Le premier argument est pap plutôt que pcp. 2) Le troisième argument diffère dans ce que le nouveau paramètre réside dans la structure UA (vieillissement par l'utilisateur) au sein de la structure AP (paramètre de vieillissement) du MSPS. Ceci contraste avec la référence «&MP.UM» utilisée pour les paramètres de modèle définis par l'utilisateur (modèle utilisateur avec paramètres de modèle).

(C) Modifications du fichier adju.cpp

La première modification fait la mise à jour de la chaîne de texte de documentation ayant

trait au vieillissement des données. La fonction originale SPSM\GLASS définit cette chaîne vers la ligne 43 comme étant

```
/*global*/ char AGENAME[IDSIZE+1] = "Unnamed";
```

Nous la modifions de façon qu'elle ressemble à ce qui suit --

```
/*global*/ char AGENAME[IDSIZE+1] = "RRSP Contr(Earnings)";
```

Avec la valeur de paramètre disponible partout à l'intérieur du MSPS, nous faisons l'ajout au code de source de façon à mettre en oeuvre la croissance des cotisations au REER. La première chose dont nous avons besoin est un ensemble de quelques variables locales qui nous aideront à passer d'un individu à l'autre à l'intérieur du ménage analysé et à peut-être affecter les cotisations modifiées au REER. Par conséquent, nous ajoutons les quatre déclarations suivantes à la fonction `adju.cpp`, en les insérant juste après l'accolade d'ouverture de la fonction.

```
NUMBER earn;          /* total paid and self-employment earnings */
int group;             /* number of UDGFRRSP multiples to use */
register P_in in;      /* pointer to data for current person */
int ini;               /* persons processed */
```

Pour les affectations de vieillissement elles-mêmes, l'endroit approprié se trouve près de la toute fin de la fonction `adju.cpp`, à l'intérieur du segment de code --

```
    DEBUG_ON("adju");
    /* Just call the standard adjustment algorithm */
    adj(hh);
    DEBUG_OFF("adju");
```

Notre ajout est placé juste entre les instructions `adj(hh);` et `DEBUG_OFF("adju");`.

```
/* Grow RRSP contributions as a function of total earnings */

for (ini=0, in=&hh->in[0]; ini<hh->hhnin; in++, ini++) {
    if (in->id.idrrsp == (NUMBER)0.0) {
        continue;
    }
    earn = in->id.idiemp + in->id.idisefm + in->id.idisenf;
    if (earn <= (NUMBER)20000.0) {
        continue;
    }
    group = (int)(ONE+(earn-(NUMBER)20000.0)/(NUMBER)10000.0);
    in->id.idrrsp*=(ONE+AP.UA.UDGFRRSP*(float)group);
}
```

Le nouveau code, précédé d'une en-tête contenant un commentaire explicatif, est réparti en éléments qui sont relativement simples.

- (1) La partie commande de l'instruction «for» a été copiée, entièrement, depuis la fonction `memol.cpp` (calculant les totaux pour les individus) dans le sous-répertoire SPSM\GLASS. Elle passe par tous les individus du ménage. Les variables locales définies auparavant sont utilisées dans cette opération.
- (2) La croissance par multiplication des cotisations au REER n'est pas significative s'il n'y a rien au départ. Par conséquent, l'instruction «if-continue» des trois lignes suivantes bondit par-dessus le reste des quatre instructions, si l'individu n'a aucune cotisation à des REER. Le type «NUMBER», ici et plus loin, indique les intentions de l'utilisateur

relativement aux types de variables; il empêche les avertissements du compilateur.

- (3) Si les cotisations au REER sont positives, la ligne suivante calcule les gains de l'individu d'un emploi rémunéré et d'un emploi autonome agricole et non agricole. Si le total ne dépasse pas 20 000 \$, le reste de l'instruction `for` est contourné; une autre instruction «if-continue» remplit sa fonction.
- (4) L'affectation à la variable «de groupe» calcule le nombre de multiples d'UDGFRRSP pertinents pour la croissance. L'instruction finale dans le corps de la boucle applique la croissance par une affectation multiplicatrice. Ces deux instructions seront exécutées seulement si une certaine croissance est appropriée. Les types (int) et (NUMBER) qu'ils contiennent indiquent l'intention explicite de l'utilisateur quant aux conversions de types de variables; ils servent à empêcher les avertissements inutiles pendant l'étape de la compilation.

(D) Compilation du modèle amélioré

Il faut faire la mise au point du modèle avant la compilation de `GLASSEX4.EXE`. C'est seulement après cette opération que le modèle peut être exécuté pour des essais de validation et le travail de production.

(E) Entrée d'une valeur au paramètre

Pour toute exécution particulière du modèle, l'utilisateur doit fournir une valeur pour le nouveau paramètre, par exemple une valeur de 0,01. Habituellement, l'utilisateur le fera «à la volée» pendant l'exécution du nouveau modèle, ou par un fichier «.API» (inclusion de paramètre de vieillissement) qui modifiera le contenu du fichier `APR` spécifié dans le fichier de commande du modèle (" .CPR"). Dans notre exemple, le fichier ".API" consisterait dans la seule ligne --

```
UDGFRRSP      0.01
```

si aucun paramètre de vieillissement existant n'a à être modifié.

(F) Validation du modèle

Avant d'utiliser le modèle à des fins sérieuses, l'utilisateur désirera le valider pour s'assurer qu'il fonctionne de la façon prévue. Nous ne procéderons pas ici à une telle validation détaillée pour des raisons d'espace; habituellement, on générerait quelques tables sélectionnées pour des exécutions différentes, en vérifiant que le modèle produit les résultats attendus. Par exemple, en laissant à zéro le facteur UDGFRRSP, on laisserait inchangé le montant total du REER. De même, une petite valeur, disons 0,01, aurait un effet faible ou nul sur les unités à faible revenu, mais aurait un effet plus grand sur les unités à revenu plus élevé. Une table, définie au niveau de l'individu, qui indiquerait l'augmentation de la variable de cotisations au REER comme fonction des gains de l'individu contribuerait beaucoup à décider si l'algorithme donne le bon montant d'augmentation du REER. Elle pourrait être générée si l'on utilisait un fichier de résultats basé sur la base de données non modifiée et si on comparait le nombre d'individus et les montants des cotisations au REER à leurs équivalents de ces variables après le nouveau vieillissement des cotisations au REER.

Lorsque l'on utilise l'échantillon de 5 % de la BDSPPS, avec la population de 1986, les paramètres de vieillissement, et les paramètres de modèle, on obtient les résultats sommaires suivants pour un facteur UDGFRSP de 0,01 :

	Croissance avant	Croissance après	Écart
Cotisations au REER (M \$)	11 134,3	11 329,2	194,9
Impôt sur le revenu fédéral (M \$)	41 173,3	41 118,0	55,3
Impôt sur le revenu provincial (M \$)	24 190,6	24,160,5	30,1

Le total des cotisations au REER s'est accru d'environ 1,75 % et les impôts sur le revenu fédéral et provincial ont connu une chute correspondante un peu inférieure au montant des nouvelles cotisations au REER.

Liste de vérification pour le changement «global» des variables de base de données

(A) Vérifier si les fonctions existantes du MSPS suffisent à la mise en oeuvre du vieillissement souhaité des données, ce qui ferait qu'une nouvelle logique ne serait pas nécessaire.

Le vieillissement désiré de la population peut-il être mis en oeuvre par le choix de fichiers de pondération de cas existants? Si c'est le cas, il suffit de spécifier le fichier de pondération de cas pertinent (avec l'extension ".WGT") par le paramètre de commande INPWGT (pondération à l'entrée). Utilisez un fichier ".CPI" pour fournir la valeur INPWGT souhaitée, ou entrez-la à la volée en répondant aux messages guides du modèle.

L'ajustement des valeurs des données peut-il être fait par la modification des valeurs des paramètres de vieillissement de données du MSPS, de concert avec l'algorithme de vieillissement des données normal du MSPS (adj(hh))? Si c'est le cas, fournissez les valeurs de paramètre de vieillissement pertinents du MSPS par un fichier ".API". Spécifiez-les au MSPS soit en direct, soit par un fichier de traitement par lots utilisé pour coordonner l'exécution du modèle.

(B) Si les ajustements désirés à apporter aux données ne peuvent être faits par les procédures intégrées de vieillissement des données, alors il faut une nouvelle logique. Les étapes d'ajout de cette nouvelle logique de vieillissement des données sont les suivantes :

1. Copier tous les fichiers pertinents vers un nouveau répertoire créé pour l'analyse. Les fichiers \SPSM\GLASS\adju.cpp, SPSMGL.dsw sont toujours pertinents. Les fichiers \SPSM\GLASS apu.h et \SPSM\GLASS\apdu.cpp seront pertinents lorsqu'il faudra de nouveaux paramètres des vieillissement.
2. Modifier l'environnement du projet de façon à inclure tous les fichiers pertinents et changer le nom du modèle compilé. Modifiez apu.h si de nouveaux paramètres de vieillissement de données sont définis.
3. Modifier apdu.cpp si de nouveaux paramètres de vieillissement de données sont définis. Les changements consisteront dans l'ajout de nouveaux appels de pmaddent et de stradd, de façon que la substance des nouveaux paramètres soit disponible partout à

l'intérieur du MSPS. Faites la mise au point du modèle.

4. Modifier `adju.cpp`. Modifiez d'abord la chaîne de texte de documentation de la fonction, `AGENAME[IDSIZ+1]`. Mettez ensuite en oeuvre la nouvelle logique du vieillissement des données. Cette étape implique souvent la déclaration de variables locales utile ainsi que le passage par tous les individus ou toutes les familles du ménage.
5. Compiler et valider le modèle avant de l'utiliser pour des exécutions de production. La totalisation parallèle d'individus et de montants pertinents avant et après les modifications touchant le vieillissement des données sont recommandés.
6. Faire des exécutions de production en utilisant la nouvelle logique de vieillissement après sa validation.

MODIFICATIONS QUI TOUCHENT SEULEMENT LA BASE OU SEULEMENT LA VARIANTE

La construction d'un modèle dans lequel le vieillissement des données diffère entre le système de base et le système de variante est en elle-même plus compliquée que la construction d'un modèle dans lequel les deux systèmes sont traités de façon identique. Lorsque cela est possible, l'utilisateur devrait éviter ce genre de complication. La capacité du MSPS d'utiliser les fichiers de résultats (extension ".MRS") fournit le principal mécanisme pour éviter le vieillissement des données soumis à des conditions de système.

La méthode fondamentale consiste à diviser le problème en deux volets, un pour chaque système. Ensuite, à l'intérieur de chacun de ces systèmes, un seul algorithme de vieillissement de données s'applique tout comme les méthodes décrites plus haut dans le présent chapitre. L'utilisateur crée d'abord un fichier de résultats pour l'un des deux systèmes, choisissant les variables nécessaires pour les totalisations propres au système et pour toute comparaison qui doit être faite. Dans la création de ce premier système, l'utilisateur applique les hypothèses de vieillissement de données qui conviennent pour ce système. Par la suite, le second système est simulé, avec le vieillissement de données de rechange approprié qui doit lui être appliqué. Le fichier de résultats est lu, en parallèle avec le traitement du second système, de façon que les deux systèmes, avec leurs hypothèses de vieillissement de données différentes, sont disponibles simultanément pour toutes les comparaisons nécessaires. La publication *Introduction et survol* fournit une illustration de l'utilisation des fichiers de résultats.

Le reste de la présente section convient lorsque la méthode des fichiers de résultats est considérée comme ne convenant pas ou n'étant pas appropriée pour la tâche à accomplir. Quelques exemples illustreront ces circonstances.

1. L'utilisateur peut accorder une grande importance au fait d'avoir un modèle qui est complet en lui-même et qui, quand il a été validé, est relativement facile à utiliser en direct.
2. L'application prévue du modèle peut impliquer l'analyse de sensibilité qui exigerait plusieurs fichiers MRS, avec une possibilité inhérente que la confusion s'installe. Elle

pourrait exiger, par exemple, l'étude des répercussions de la modification du vieillissement d'une variable en particulier, avec diverses autres variables modifiées à répétition en parallèle entre les systèmes de base et de variante.

3. L'application prévue pourrait impliquer des comparaisons compliquées exigeant de gros fichiers .MRS (ou un grand nombre d'entre eux simultanément) quand le stockage sur disque est un problème.

Nous croyons cependant que les situations de ce genre, bien qu'elles puissent se produire à l'occasion, seront l'exception plutôt que la règle. Nous encourageons donc l'utilisateur à tenter d'éviter les modèles de systèmes parallèles dans lesquels le vieillissement des données diffère entre les deux systèmes.

Dans ses grandes lignes, la méthode à étudier pour modifier les données propres à un système est semblable à celle qui est utilisée pour modifier la LOGIQUE FISCALE/DE TRANSFERT d'un système. Tout nouveau paramètre de vieillissement de données propre à un système est ajouté, par les fichiers `mpu.h` et `mpdu.cpp`, comme paramètre de modèle et non comme paramètre d'ajustement de données comme tel. Comme on l'a décrit ci-dessous, l'utilisateur peut désirer ajouter de nouvelles variables dépendantes de modèle pour assurer le suivi des changements apportés. Bien que, si de nouveaux paramètres et de nouvelles variables dépendantes ne sont pas nécessaires, la procédure s'applique également aux modèles de base et de variante du MSPS, nous expliquerons la procédure par rapport à la situation plus courante des modèles de variante.

La méthode axée sur le MODÈLE tout juste résumée est permise par la conception du MSPS. Comme il n'y a qu'un fichier ".APR", ces paramètres affectent inévitablement le vieillissement de données pour tous les systèmes à l'intérieur du modèle. Par contre, les changements apportés par les fichiers ".MPI", et par les fonctions `Adrv.cpp` et `drv.cpp` propres au système, s'appliquent seulement à un système fiscal/de transfert désigné. L'utilisateur peut tirer avantage de cette spécificité au système pour mettre en oeuvre des ajustements de données propres au système.

La clé aux changements du vieillissement des données propres au système réside dans la modification apportée au fichier `Acall.cpp`. Essentiellement, l'utilisateur «intercepte» l'enregistrement de données du ménage juste avant qu'il ne soit utilisé par la fonction dans cette procédure, il effectue les changements désirés et, par la suite, rétablit l'enregistrement de données à son état original juste avant que l'exécution ne quitte la procédure. La section suivante explore plus en profondeur ces étapes axées sur `Acall.cpp`.

Mise en oeuvre des changements dans `Acall.cpp`

La présente section porte presque exclusivement sur les détails des changements apportés à l'intérieur d'`Acall.cpp`. Du fait de la similitude de l'ajustement des données spécifiques propres au système avec les genres de révision de systèmes fiscaux/de transfert décrits auparavant dans le *Guide de programmation*, certaines rubriques ne sont pas reprises ici. Particulièrement, l'utilisateur devrait ajouter tout nouveau paramètre et toute nouvelle variable dépendante nécessaire en utilisant la méthode expliquée dans les sections

précédentes. Par exemple, un utilisateur pourrait désirer ajouter une nouvelle variable de modèle pour indiquer si la valeur originale de la base de données pour une variable a été modifiée par les ajustements propres au système.

Nous étudierons les changements requis dans l'ordre où le lecteur devrait les rencontrer en lisant le code source `Acall.cpp`. Plus tard, un exemple pratique fournit une application concrète des changements.

(A) Déclarer de nouvelles variables locales (à `Acall.cpp`)

Il faut se rappeler que la procédure générale exige que l'utilisateur sauvegarde les valeurs des variables à ajuster. La sauvegarde permet de rétablir les valeurs encore une fois après avoir quitté `Acall.cpp`. Par conséquent, l'utilisateur doit inclure dans `Acall.cpp` des déclarations locales appropriées pour fournir la sauvegarde nécessaire. Habituellement, les variables à ajuster seront définies au niveau de l'individu. Par conséquent, les nouvelles variables devraient être en général définies comme des vecteurs de longueur `MAXIND`. (`MAXIND` est le nombre maximal d'individus dans une famille.) L'utilisateur peut aussi désirer définir d'autres variables de travail locales. Habituellement, l'utilisateur déclare ces variables juste avant d'ouvrir l'accolade de cette fonction, vers la ligne 99 de la version non modifiée du fichier `Acall.cpp`

(B) Sauvegarder les valeurs à modifier

Comme première chose à faire à l'intérieur de la partie exécutable du fichier `Acall.cpp`, l'utilisateur devrait sauvegarder les valeurs originales des variables qui seront modifiées. Si cela est fait, aucune des autres fonctions invoquées à l'intérieur `Acall.cpp` ne peut modifier la valeur d'abord ou utiliser la valeur non modifiée. Habituellement, la sauvegarde est effectuée par une instruction «for» qui passe par tous les individus d'un ménage et les copie, un à la fois, dans les éléments de vecteur déclarés à l'étape (A). Un des éléments du bestiaire fournit les commandes de passage pas-à-pas pertinents. L'utilisateur fait cela vers la ligne 90 du code non modifiée, juste après l'instruction -

```
DEBUG_ON("Acall");
```

(C) Modifier les valeurs de la base de données

Immédiatement après la sauvegarde des valeurs et avant que le pointeur de ménage ne soit passé à l'une ou l'autre des fonctions fiscales/de transfert ou de cumul, l'utilisateur devrait apporter les modifications nécessaires aux valeurs des variables pertinentes. Ces changements constitueront le gros de la «vraie programmation», c'est-à-dire la logique qui ne peut nécessairement pas être adaptée de façon commode ailleurs dans le MSPS.

(D) Utiliser les valeurs maintenant ajustées

Cette étape est la plus facile de toutes, puisqu'elle ne requiert aucun effort spécial de la part de l'utilisateur. Elle consiste à RETENIR les appels des diverses fonctions fiscales/de transfert et pour mémoire. Comme les valeurs des variables pertinentes ont déjà été ajustées à ce point, toutes ces fonctions exécuteront leurs calculs en utilisant les ménages ajustés.

(E) Remettre en place les valeurs originales

L'étape finale consiste à restaurer les valeurs originales aux variables qui ont été ajustées. Cela se fait habituellement vers la ligne 99 de la version non modifiée du fichier `Acall.cpp`, juste avant que le contrôle ne quitte la fonction, c'est-à-dire juste avant l'instruction --

```
DEBUG_OFF("Acall");
```

L'exécution du remplacement est importante du point de vue de la globalité du code, de sa maintenabilité et de sa réutilisabilité. L'utilisateur programme le changement sans savoir si le système programmé sera un système de base ou de variante. En remettant les choses comme elles étaient, l'utilisateur peut minimiser la possibilité d'effets secondaires non désirés ailleurs dans le modèle. En outre, cette procédure réduit au minimum la possibilité d'effets secondaires non désirés si jamais les nouveaux ajustements sont utilisés encore dans un autre modèle.

Un exemple pratique

(A) La substance à modéliser

Nous commençons par une description de la logique essentielle utilisée dans l'exemple. Il sera évident que les mêmes buts de vieillissement de données pourraient avoir été atteints par l'utilisation des techniques d'«évitement» décrites ci-dessus; cependant, comme notre objectif de documentation ici est l'illustration de techniques de vieillissement de données propres au système, nous considérons que ces techniques d'évitement ne conviennent pas pour nos buts immédiats.

Supposons que quelque analyse exogène touchant de nouveaux besoins de déclaration de revenu laissent croire que des individus déclareront plus de revenus d'emploi autonome. Plus précisément, supposons que 5 % de ces individus (1) ne déclarant pas plus de 100 \$ de revenu d'emploi autonome (agricole et non agricole combinés) et (2) qui sont âgés dans les deux cas de plus de 25 ans et de moins de 60 ans et (3) qui ont en outre une demie année ou plus sans travail et en recherche de travail, ont vraiment des revenus d'emploi autonome non agricole qui n'ont pas été déclarés auparavant, mais qui seront à l'avenir déclarés. En outre, supposons les montants du «nouveau» revenu d'emploi autonome pour ces personnes devraient être distribués, croit-on, uniformément entre zéro et 4 000 \$ par année.

L'utilisateur cherche à estimer les impôts sur le revenu supplémentaires pouvant être perçus de ces personnes et aussi à évaluer les répercussions de ce revenu «découvert» sur la réduction du taux de pauvreté comme il est mesuré avec la fonction du seuil de la pauvreté. Pour effectuer cette étude, l'utilisateur prévoit, dans le système fiscal/de transfert de variante imputer les montants appropriés de ces nouveaux revenus à des personnes sélectionnées au hasard qui satisfont ces conditions.

(B) Nouveaux paramètres et variables pertinents

Selon les pratiques MSPS recommandées qui cherchent à éviter les valeurs implantées directement dans un modèle, l'utilisateur établit comme suit les nouveaux paramètres de

vieillesse définis par l'utilisateur :

Paramètre	Description :	Valeur :
NSEFLAG	«Drapeau de nouveau revenu d'emploi autonome»	1
NSEAMT	«Montant de base du nouveau revenu d'emploi autonome»	100,0 0,05
NSEFRC	«Fraction du nouveau revenu d'emploi autonome»	26
NSEWKS	«Exigence de semaines de nouvel emploi autonome»	25
NSEMINAGE	«Âge minimum du nouvel emploi autonome»	60
NSEMAXAGE	«Âge maximum de nouvel emploi autonome»	4 000,0
NSEMAXINC	«Nouveau revenu maximum du nouvel emploi autonome»	

De même, l'utilisateur définit de nouvelles variables qui permettront le compte commode du nombre de personnes admissibles et du nombre pour lesquels les nouveaux revenus sont synthétisés. Il sera aussi utile d'avoir une nouvelle variable supplémentaire pour les montants de revenu synthétisé.

Variable : Description :

uvnseef	«Nouvel emploi autonome admissible»
uvnsesf	«Nouvel emploi autonome reçu»
uvnseamt	«Montant du nouvel emploi autonome»

(C) Préparation en vue de l'analyse

L'utilisateur commence par créer un nouveau sous-répertoire pour l'analyse, GLASSEX5. Il copie les fichiers de gabarit requis : SPSMGL.dsw (pour commander la compilation), mpu.h et Ampd.cpp (pour rendre les nouveaux paramètres disponibles), vsu.h et vsdu.cpp (pour rendre les nouvelles variables disponibles) et Acall.cpp (pour mettre en oeuvre les nouveaux ajustements de base de données propres au système).

Nous examinons les changements dans l'ordre où l'utilisateur serait encouragé à les apporter.

(D) Changements à apporter au projet

Tous les fichiers pertinents devraient être inclus dans le projet et le nom du modèle devrait être changé et devenir GLASSEX5.EXE.

(E) Changements au fichier mpu.h

L'utilisateur fournit les déclarations pour tous les paramètres nouveaux décrits ci-dessus.

```
int     NSEFLAG;      /* New Self-Employment Income Flag */
NUMBER NSEAMT;        /* New Self-Employment 'Trivial Amount' */
NUMBER NSEFRC;        /* New Self-Employment Fraction */
NUMBER NSEWKS;        /* New Self-Employment Weeks Requirement */
NUMBER NSEMINAGE;     /* New Self-Employment Minimum Age */
NUMBER NSEMAXAGE;     /* New Self-Employment Maximum Age */
NUMBER NSEMAXINC;     /* New Self-Employment Maximum New Income */
```

(F) Changements au fichier Ampd.cpp

L'utilisateur modifie le fichier Ampd.c en fournissant les appels de pmaddent et de stradd pour tous les nouveaux paramètres. Les nouveaux appels de pmaddent seraient les suivants :

```
pmaddent(pcp, "NSEFLAG", (char *)&MP.UM.NSEFLAG, NULL, P_SCL, C_INT, E_FLAG, 0, NULL, 0);
pmaddent(pcp, "NSEAMT", (char *)&MP.UM.NSEAMT, NULL, P_SCL, C_NUM, 0, 0, NULL, 0);
pmaddent(pcp, "NSEFRC", (char *)&MP.UM.NSEFRC, NULL, P_SCL, C_NUM, E_FRCT, 0, NULL, 0);
pmaddent(pcp, "NSEWKS", (char *)&MP.UM.NSEWKS, NULL, P_SCL, C_NUM, 0, 0, NULL, 0);
pmaddent(pcp, "NSEMINAGE", (char *)&MP.UM.NSEMINAGE, NULL, P_SCL, C_NUM, 0, 0, NULL, 0);
pmaddent(pcp, "NSEMAXAGE", (char *)&MP.UM.NSEMAXAGE, NULL, P_SCL, C_NUM, 0, 0, NULL, 0);
pmaddent(pcp, "NSEMAXINC", (char *)&MP.UM.NSEMAXINC, NULL, P_SCL, C_NUM, 0, 0, NULL, 0);
```

Les appels de stradd connexes ressembleraient à ce qui suit :

```
stradd("NSEFLAG", "New Self-Employment Income Flag");
stradd("NSEAMT", "New Self-Employment 'Trivial Amount'");
stradd("NSEFRC", "New Self-Employment Fraction");
stradd("NSEWKS", "New Self-Employment Weeks Requirement");
stradd("NSEMINAGE", "New Self-Employment Minimum Age");
stradd("NSEMAXAGE", "New Self-Employment Maximum Age");
stradd("NSEMAXINC", "New Self-Employment Maximum New Income");
```

(G) Changements au fichier vsu.h

Dans ce fichier, l'utilisateur déclare les nouvelles variables qui contribueront à une totalisation et à une validation plus commode des individus pour lesquels le nouveau revenu est considéré ou réellement synthétisé.

```
int      uvnseef; /* Eligible for New Self-Empl Synthesis */
int      uvnseef; /* Received New Self-Empl Income */
NUMBER   uvnseamt; /* New Self-Empl Amount */
```

(H) Changements au fichier vsdu.cpp

Dans vsdu.c, l'utilisateur appelle vardef et stradd pour rendre les nouvelles variables disponibles partout à l'intérieur du modèle. Comme il a été indiqué ci-dessus, il y a des variables de classe à étudier pour les sorties de tableaux croisés et une valeur à virgule flottante NUMBER pour le montant du revenu d'emploi autonome synthétisé.

```
/* uvnseef: (Class) Flag: Individual eligible for NSE synthesis? */
vardef("_uvnseef", IN, im.uv.uvnseef, C_INT, V_CLAS);
stradd("uvnseef", "Eligibility for Synth Self-Empl");
stradd("nseef", "\tNot Eligible\tEligible");

/* uvnseef: (Class) Flag: Individual Got Synth. NSE? */
vardef("_uvnseef", IN, im.uv.uvnseef, C_INT, V_CLAS);
stradd("uvnseef", "Synth Self-Empl Receipt");
stradd("nseef", "\tNo Receipt\tReceipt");

/* uvnseamt: (Analysis) NUMBER: Amount of synthesized NSE */
vardef("_uvnseamt", IN, im.uv.uvnseamt, C_NUM, V_ANAL);
stradd("uvnseamt", "Synth Self-Empl Amount");
```

(I) Changements au fichier Acall.cpp

(i) Les modifications commencent par la déclaration de nouvelles variables essentielles au processus d'ajustement des données. Nous utilisons la notation standard du MSPS pour le pointeur vers un individu et pour le nombre de personnes soumises au traitement (pour les règles d'arrêt à l'intérieur des ménages). En outre, il y a un vecteur déclaré pour contenir les valeurs originales du revenu d'emploi autonome non agricole des individus.

```
register P_in in;          /* pointer to data for current person */
int ini;                  /* persons processed */
NUMBER orignfse[20];      /* original non-farm self-empl income */
```

(ii) Les modifications se poursuivent avec le code servant à stocker le revenu d'emploi autonome non agricole existant de façon qu'il puisse être par la suite rétabli à son état original. Nous utilisons un des éléments standard du bestiaire, passant pas-à-pas d'un individu à l'autre à l'intérieur du ménage, pour mettre en oeuvre cet archivage.

```
/* Archive original database values for non-farm self-employment */

for (ini=0, in=&hh->in[0]; ini<hh->hhnin; in++, ini++) {
    orignfse[ini]=in->id.idisnf;
}
```

Une version légèrement plus efficace de ce code rendrait l'exécution de l'instruction de sauvegarde conditionnelle à l'attribution de la valeur au paramètre NSEFLAG pour activer la fonction de synthèse. La version est ici plus simple et légèrement plus sûre.

(iii) Mettre en oeuvre le revenu d'emploi autonome augmenté sous conditions

[Le travail dans l'utilisation des variables pseudo-aléatoires existantes tant pour le choix des nouvelles personnes à déclarer des revenus d'emploi autonome (non agricoles) que pour le montant qu'elles doivent déclarer. Expliquer comme cela est essentiel à la reproductibilité étant donné la sélection des sous-ensembles de données.]

```
/* Selectively synthesize non-farm self-employment income */

for (ini=0, in=&hh->in[0]; ini<hh->hhnin; in++, ini++) {
    in->im.uv.uvnseef=0; /* assign values to new vars */
    in->im.uv.uvnseesf=0;
    in->im.uv.uvnseamt=(NUMBER)0.0;
    if (MP.UM.NSEFLAG==0) {
        continue; /* don't synthesize if facility is off */
    }

    if ( ((in->id.idisefm+in->id.idisnf)>MP.UM.NSEAMT) ||
        (in->id.idnage<MP.UM.NSEMINAGE) ||
        (in->id.idnage>MP.UM.NSEMAXAGE) ||
        (in->id.idlyun<(int)MP.UM.NSEWKS) ) {
        continue; /* ignore ineligible individuals */
    }

    in->im.uv.uvnseef=1; /* mark indiv. as potentially eligible */

    if (in->id.idrand[2]>MP.UM.NSEFRC) {
        continue; /* individual was not selected to get income */
    }

    in->im.uv.uvnseesf=1; /* mark indiv. as recipient */ in->im.uv.uvnseamt=in-
    >id.idrand[3]*MP.UM.NSEMAXINC; /*synthesize amt */ in->id.idisnf+=in-
    >im.uv.uvnseamt; /* add syn amt to non-farm self-empl */
}
```

Le code ci-dessus, bien que d'une certaine longueur, est simple. L'intérieur de la boucle de parcours des individus, on fait ce qui suit :

On attribue des valeurs par défaut aux nouvelles variables définies par l'utilisateur.

On n'exécute pas le reste des instructions de la boucle si la fonction n'est pas activée.

On n'exécute pas le reste des instructions de la boucle si l'individu ne respecte pas les conditions d'admissibilité pour la synthèse du nouveau revenu d'emploi autonome.

On marque l'individu comme pouvant être admissible en vue de la synthèse; puis on n'exécute pas le reste des instructions de la boucle si l'individu n'est pas «choisi» pour recevoir le revenu.

Si l'exécution atteint cette étape, on marque l'individu comme bénéficiant du revenu synthétisé et on impute le montant, en additionnant le nouveau montant à la variable d'emploi autonome non agricole de la personne.

Une fois la boucle exécutée, la synthèse du nouveau revenu d'emploi autonome non agricole est complète pour tous les membres du ménage. À ce point, l'instruction «régulière» du fichier `Adrv.cpp` se poursuit, calculant les montants d'impôt/de transfert et les divers articles pour mémoire.

(iv) Enfin, après que le ménage ajusté a été traité par toutes les fonctions fiscales/de transfert et pour mémoire, le nouveau code rétablit les valeurs originales de revenu d'emploi autonome non agricole.

```
/* Restore original database values for non-farm self-employment */  
  
for (ini=0, in=&hh->in[0]; ini<hh->hhnin; in++, ini++) {  
in->id.idisenf=orignfse[ini]; }
```

Une version légèrement plus efficace de ce nouveau code ferait en sorte que l'exécution des instructions de rétablissement des données originales soit conditionnelle à l'attribution de la valeur 1 au paramètre `NSEFLAG` qui active la fonction de synthèse. La version ici est plus simple et légèrement plus sûre.

(J) Les nouveaux fichiers MPI et CPI

Il reste encore à fournir les valeurs aux divers paramètres de façon que le MSPS, pendant une exécution en particulier, puisse mettre en oeuvre les ajustements désirés. Un «fichier d'inclusion» de paramètres (extension ".MPI") avec les entrées suivantes remplit cette fonction.

<code>NSEFLAG</code>	1
<code>NSEAMT</code>	100.0
<code>NSEFRC</code>	0.05
<code>NSEWKS</code>	26.0
<code>NSEMINAGE</code>	25.0
<code>NSEMAXAGE</code>	60.0
<code>NSEMAXINC</code>	4000.0

De même, il est nécessaire de veiller à ce que des séries indépendantes pertinentes de variables pseudo-aléatoires soient générées pour servir d'entrée aux choix «aléatoires» de bénéficiaires de revenus synthétiques et des montants associés de revenus synthétisés.

(K) Compilation et validation du modèle

Quand il a terminé toutes les modifications du code de source, l'utilisateur devrait d'abord faire la mise au point du modèle, puis le compiler dans le fichier exécutable désiré, GLASSEX5. Nous concluons cet exemple pratique en expliquant un ensemble très rapide et obscur de tables de validation. Pour une application sérieuse, l'utilisateur devrait normalement faire une validation beaucoup plus rigoureuse des changements. Il faut se rappeler aussi que ce genre d'ajustement de données propre au système pourrait avoir été fait plus facilement avec des fichiers de résultats (".MRS"). Avec ce mécanisme, une logique d'attribution de revenu équivalente aurait été appliquée par l'intermédiaire du fichier adju.cpp et les paramètres pertinents auraient été fournis par un fichier API.

Supposons, pour les fins de cette validation rapide, que la source exogène de l'utilisateur a déjà indiqué grossièrement combien d'individus devraient se retrouver avec ce nouveau revenu d'emploi autonome, peut-être comme une fonction de quelque variable de politique pertinente.

L'utilisateur voudra d'abord totaliser les nombres d'individus selon les valeurs des deux variables de classe définies par l'utilisateur, uvnsef et uvnsef. Les entrées de cette table peuvent alors être comparées à la source exogène afin de confirmer (1) que les nombres d'individus admissibles correspondent à ceux qui sont spécifiés par la «source exogène», (2) qu'une proportion appropriée de ces individus se sont vu imputer un nouveau revenu d'emploi autonome.

Ensuite, l'utilisateur voudrait vérifier que le montant moyen du revenu du nouvel emploi autonome imputé est approprié (c.-à-d. la moitié de la valeur du paramètre NSEMAXINC 4 000 \$). Il serait logique de totaliser le montant total du nouveau revenu imputé de façon que ce montant puisse être comparé aux augmentations d'impôt sur le revenu du fédéral et du provincial. Par conséquent, l'utilisateur peut vérifier que la proportion appropriée de nouveau revenu revient au gouvernement comme impôt sur le revenu.

Même pour les exécutions de validation, il semble logique de chercher le niveau de changement dans les incidences des unités au-dessous du seuil de pauvreté. Étant donné les conditions relativement serrées d'admissibilité à la réception du revenu synthétisé et de la portion relativement faible de population admissible choisie pour recevoir le nouveau revenu d'emploi autonome, l'utilisateur devrait s'attendre seulement à de petits changements de cette incidence.

Ici, nous montrons la première partie de cette validation, vérifiant les montants du nouveau revenu d'emploi autonome. Nous utilisons le BD/MSPS de 1986 avec l'échantillon de 5 %. Les changements dans le «taux de pauvreté», qui ne sont pas montrés ici, seraient calculés à l'aide des variables du MSPS «efpovthr» (seuil de pauvreté) et «impovinc» (revenu pour fins

de comparaison avec le seuil de pauvreté pertinent). La validation est effectuée de façon plus commode par des tableaux croisés. Les paramètres de commande pertinents, entrés par le fichier ".CPI", sont les suivants :

```
XTFLAG      1
XTSPEC
IN: { units }
* uvnseef
* uvnseef;
IN: { uvnseamt,
      uvnseamt/units }
* uvnseef;
IN: { uvnseamt,
      imtxf-_imtxf,
      imtxp-_imtxp }
* uvnseef
```

Les tables obtenues ressemblent à ce qui suit --

Table 1U: Unit Count (000) for Individuals by Eligibility for Synth Self-Empl and Synth Self-Empl Receipt

Synth Self-Empl Receipt

Eligibility for Synth Self-Empl	No Receipt	Receipt
Not Eligible	23351.7	0.0
Eligible	809.6	47.2

Table 2U: Selected Quantities for Individuals by Synth Self-Empl Receipt

Synth Self-Empl Receipt

Quantity	No Receipt	Receipt
Synth Self-Empl Amount (M)	0.0	92.5
uvnseamt/units	0	1962

Table 3U: Selected Quantities for Individuals by Synth Self-Empl Receipt

Synth Self-Empl Receipt

Quantity	No Receipt	Receipt
Synth Self-Empl Amount (M)	0.0	92.5
imtxf-_imtxf (M)	1.0	12.9
imtxp-_imtxp (M)	0.5	9.5

Pour ce qui est de la substance de ces tables, nous supposons que les 809,6 milliers de personnes de la table 1U correspondent raisonnablement bien avec la «source de données exogène» hypothétique. Puisque 47,2 milliers de ces personnes reçoivent un certain nouveau revenu d'emploi autonome, l'objectif de 5 % a été grossièrement atteint. On suppose que la proportion serait plus proche de 5 % si nous utilisions toute la BDSPS.

La table 2U confirme que notre nouvel algorithme attribue le nouveau revenu d'emploi autonome seulement à des personnes admissibles à le recevoir. Le montant total du nouveau revenu et les montants moyens associés confirment que les montants attendus d'un nouveau

revenu sont synthétisés (en gros 2 000 \$ par personne choisie).

La table 3U indique la partie du nouveau revenu, un peu plus que le quart, qui est retenue par le système fiscal. Comme on s'y attendait, la plus grande partie de ces montant récupérés l'est directement des individus qui l'ont reçue, bien qu'il y ait certaines des personnes qui ne l'ont pas reçue, principalement du fait que certains individus qui l'ont reçue se retrouvent comme ayant moins de valeur à cause des exemptions personnelles entraînées par leur nouveau revenu. Il est clair que, avec un revenu de moins de 100 M \$ distribués dans l'ensemble du secteur personnel, nous ne nous attendons pas à de grandes répercussions sur la proportion de la population en dessous du seuil de pauvreté.

Enfin, quand l'utilisateur est certain de l'exactitude des procédures d'ajustement, il exécute la BDSPS complète dans le modèle dans une ou plusieurs exécutions de production. Pour atteindre les buts de l'exemple décrit au début de la présente section, la sortie devrait inclure les totaux d'impôt sur le revenu du fédéral et du provincial ainsi que le nombre de familles au-dessus et au-dessous des seuils de pauvreté, ces sorties étant produites tant avec que sans la synthèse du nouveau revenu d'emploi autonome non agricole. Habituellement, l'utilisateur inclurait aussi la ventilation de ces variables par variable de classe pertinente comme type de famille.

Liste de vérification pour les changements apportés à la base de données propres au système

- (A) Créer un nouveau sous-répertoire pour l'analyse. Copiez dans ce répertoire les gabarits de tous les fichiers qui sont nécessaires pour l'analyse. Les éléments qui sont probablement nécessaires comprennent `SPSMGL.dsw`, `mpu.h`, `Ampd.cpp`, `vsu.h`, `vsdu.cpp`, `Acall.cpp` et un fichier de commande ("`.CPR`"). L'utilisateur créera aussi, dans le même sous-répertoire, d'autres fichiers nécessaires à l'analyse pour lesquels il n'y a aucun gabarit évident, par exemple le fichier "`.MPI`" qui fournira les valeurs aux paramètres de vieillissement des données propres au système, ou un fichier de traitement par lots pour commander le traitement du MSPS.
- (B) Changer l'environnement du projet de façon à inclure tous les fichiers pertinents et changer le nom du fichier de sortie exécutable.
- (C) Modifier les fichiers `mpu.h` et `Ampd.cpp` de façon à déclarer tout nouveau paramètre d'ajustement de données propre au système et à les rendre disponibles pour le reste du MSPS, par des appels de `pmaddent` et de `stradd`.
- (D) Modifier les fichiers `vsu.h` et `vsdu.cpp` au besoin afin de déclarer toute nouvelle variable de modèle propre au système et à les rendre disponibles au reste du MSPS par les appels de `vardef` et de `stradd`.
- (E) Modifier le fichier `Acall.cpp` afin de sauvegarder les valeurs originales des variables à ajuster, à effectuer les changements puis, après le traitement du ménage, à rétablir les valeurs originales avant de quitter la procédure. Ces étapes exigent habituellement la définition de VECTEURS locaux de valeurs dont les dimensions correspondent aux nombres possibles d'individus dans un ménage.

- (F) Compiler le nouveau modèle et corriger tout problème décrit par le compilateur.
- (G) Fournir les valeurs pour les nouveaux paramètres d'ajustement de données propres au système par des fichiers ".MPI". Lorsque le vieillissement dépend de l'utilisation de variables pseudo-aléatoires, il faut fournir un fichier ".CPI" dans lequel on a apporté les modifications nécessaires au paramètre SEED. Le modèle aura accès à ces valeurs de paramètres de modèle et de commande au moment de l'exécution du modèle soit en direct, soit par un fichier de traitement par lots du MSPS.
- (H) Valider soigneusement le modèle, puis faire des exécutions de production.