UNE APPROCHE FORMELLE DE LA CONSTRUCTION DES LOGICIELS.

J.R. Abrial

Traducteur G. Laffitte

5 Septembre 1989

3mes Journées "Pratique des méthodes et outils d'aide à la conception des systèmes d'information", Nantes, 26-28 septembre 89, Liana-IUT, Textes rassemblés par H.Habrias- ISBN:2-906 082-07-4

RA.

Sommaire

UNE APPROCHE FORMELLE DE LA CONSTRUCTION DES LOGICIELS.

INTRODUCTION

La rationalisation de la CONSTRUCTION DES LOGICIELS est-elle aujourd'hui l'une des préoccupations majeures des informaticiens? Nous allons présenter ici, brièvement et de manière informelle, un certain nombre de résultats concernant l'APPROCHE AXIOMATIQUE de cette question.

Une des tendances récentes de notre discipline consiste à diminuer la distinction traditionnelle entre les deux activités principales de la construction des logiciels, à savoir la spécification et la programmation. Cette tendance prend parfois la forme de "spécifications exécutables" [14, 19] ; nous adoptons ici une approche quelque peu différente, celle de "programmes pas nécessairement exécutables".

Dans ce qui suit, notre intention est de montrer que les principaux concepts de la spécification (section 1) et ceux de la programmation (section 2) peuvent être unifiés [15, 16, 17]. Cette unification entraîne l'apparition d'une nouvelle activité appelée conception (section 3); la conception est située entre la spécification et la programmation, et son rôle est d'assurer un passage systématique de l'une à l'autre [12, 13, 16, 20].

3mes Journées "Pratique des méthodes et outils d'aide à la conception des systèmes d'information", Nantes, 26-28 septembre 89, Liana-IUT, Textes rassemblés par H.Habrias- ISBN:2-906 082-07-4

1 METHODOLOGIE DE SPECIFICATION DE LOGICIEL: MODELES

Une des idées importantes de l'approche axiomatique (orientée vers les modèles) de la spécification de logiciels consiste à séparer conceptuellement la spécification des données de celle des programmes [13] .

Ainsi, afin de spécifier tout ou partie d'un logiciel, nous adopterons la stratégie suivante: nous définirons un MODELE MATHEMATIQUE DE SES DONNEES ainsi qu'une ANIMATION de ce modèle; cette animation sera spécifiée sous la forme d'un certain nombre d'OPERATIONS dont l'impantation finale pourra modifier les données en question et fournir éventuellement certains résultats. Cette approche est représentée à la figure 1.

UN MODELE "ANIME" DE DONNEES

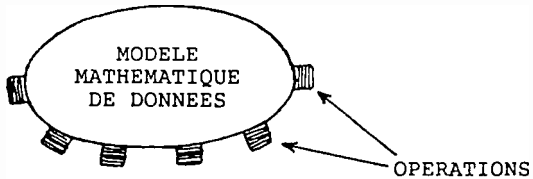

Figure 1

A ce niveau, on ne prendra en compte aucune considération d'efficacité: notre seule ligne de conduite consistera à formaliser les données de sorte que la spécification des opérations puisse être rédigée de la manière la plus "évidente" et la plus compréhensible possible.

Un logiciel ainsi spécifié sera ultérieurement utilisé par une ou plusieurs INTERFACES activant les opérations offertes par le modèle. En conséquence, le programmeur de l'une de ces interfaces aura l'ILLUSION de travailler directement au niveau d'abstraction utilisé dans le modèle. Il ne s'agit évidemment là que d'une hypothèse de travail. En pratique, les opérations effectivement activées par l'interface sont les RAFFINEMENTS ultimes des opérations spécifiées; ces raffinements sont réalisés au moyen de certains CHANGEMENTS DE VARIABLES (voir section 3.3.2) dont le rôle est d'assurer que les données peuvent être stockées dans des mémoires conventionnelles avec une parfaite efficacité.

Ensuite, lorsque le modèle est agréé par les différentes parties concernées, la construction peut être scindée en deux activités INDEPENDANTES: d'un côté, le raffinement des opérations spécifiées; de l'autre, le développement des différentes interfaces utilisant ces opérations.

Dans la suite de cette section, nous allons étudier brièvement la manière de définir des modèles mathématiques de données (section 1.1), puis nous considèrerons leur animation (section 1.2) et leur cohérence (section 1.3). Enfin (section 1.4), nous verrons la manière de construire des modèles de grande taille (contenant beaucoup de variables).

1.1 MODELES MATHEMATIQUES DE DONNEES

De manière générale, un modèle mathématique est défini, d'abord en listant ses VARIABLES, ensuite en écrivant les propriétés que ces variables doivent vérifier; l'ensemble de ces propriétés est appelé l'INVARIANT [13] du modèle. D'autre part, un modèle peut être rendu plus général au moyen de PARAMETRES GENERIQUES représentant des ensembles abstraits prédéfinis qui peuvent être utilisés dans l'invariant.

La formalisation de l'invariant d'un modèle utilise sans restriction la LOGIQUE DU PREMIER ORDRE et la THEORIE DES ENSEMBLES. Dans cette section ainsi que dans d'autres où nous aborderons les questions de notations, nous mentionnerons brièvement comment celles-ci sont définies en termes de BASES et de CONSTRUCTEURS. Par exemple, dans le cas de la logique du premier ordre appliquée à la théorie des ensembles, nous avons les éléments suivants:

BASES:	appartenance à un ensemble
CONSTRUCTEURS:	implication disjonction conjonction négation quantifications

A partir de ces concepts élémentaires, il est facile de dériver d'autres concepts tels que l'égalité ou l'inclusion ensemblistes.

Les concepts de la théorie des ensembles, quant à eux, peuvent être définis de manière analogue à partir des éléments suivants:

BASES:	paramètres génériques
CONSTRUCTEURS:	produit cartésien ensemble des parties compréhension ensembliste

Ces constructeurs d'ensembles peuvent être combinés pour former des êtres mathématiques plus élaborés tels que les relations binaires, les fonctions partielles ou totales, les suites, les arbres, etc... Nous supposons également que nous avons à notre disposition la théorie classique permettant de construire des ensembles définis par induction (comme les entiers naturels, les suites finies, les arbres finis, etc...) et de définir sur ceux-ci des fonctions (primitives) récursives. Par exemple, il est possible de définir ainsi les opérations arithmétiques, de même que les opérations classiques sur les suites et les arbres.

Au point où nous en sommes, nous pouvons dire pour résumer que toute expression formelle possédant une définition mathématique rigoureuse est "permise".

1.2 ANIMATION DU MODELE

Comme nous l'avons expliqué auparavant, pour animer un modèle de données, nous SPECIFIONS un certains nombre d'OPERATIONS (éventuellement PARAMETREES) permettant de modifier les valeurs des variables et fournissant éventuellement des résultats. De telles modifications seront plus tard réalisées physiquement au moyen de l'exécution des PROGRAMMES correspondants.

Avant d'aller plus loin, il paraît important de préciser ce que nous entendons exactement par le terme "spécification" et par le terme "programme".

Un PROGRAMME	défini sur certaines variables globales et fournissant certains résultats est la description précise de la manière dont les valeurs de ces variables sont MODIFIEES et dont les valeurs de ces résultats sont CALCULEES.
Une SPECIFICATION	(de programme) est la description précise de certaines PROPRIETES que doivent vérifier les modifications de variables et les résultats de calculs définis par ce programme.

Nous pouvons constater que ces définitions laissent une certaine marge entre un programme et une spécification; en particulier, il est clair que plusieurs programmes distincts peuvent correspondre à la même spécification. Plus précisément, nous dirons qu'un programme SATISFAIT une spécification si les modifications et les calculs qu'il définit vérifient les propriétés exprimées dans la spécification.

Cette relation de "satisfaction" ne doit en général pas être vérifiée pour toutes les valeurs possibles des variables du modèle et des paramètres de l'opération; ceci résulte du fait que, la plupart du temps, seules certaines valeurs des variables et des paramètres ont un sens pour l'opération en question. Il en résulte que la spécification d'une opération possède une caractéristique qui n'est pas partagée par le programme: celle-ci, appelée PRECONDITION, définit les valeurs des variables et des paramètres pour lesquelles l'opération a un sens.

Par analogie avec le terme précondition, nous utiliserons également dans ce qui suit celui de CONDITION DE LIAISON pour qualifier collectivement les propriétés décrites dans la spécification d'une opération. Enfin, le terme POSTCONDITION désigne une condition qui doit être vérifiée par les valeurs des variables et des résultats juste après l'exécution de l'opération.

En résumé, la spécification d'une opération est définie par:

- sa précondition,

- sa condition de liaison.

Il existe au moins trois styles différents permettant de formaliser la spécification d'une opération. Par la suite, nous utiliserons le dernier cité. Dans un souci de simplification, nous présentons ces styles pour des opérations sans paramètre et sans résultat.

LE STYLE PREDICATIF [11, 12, 13]

Soit x la variable (ou la collection des variables) du modèle. La précondition est exprimée sous la forme d'un PREDICAT DU PREMIER ORDRE en x: pre(x); la condition de liaison est exprimée sous la forme d'un prédicat du premier ordre en x et x': link(x,x'), où x' représente, par convention, les valeurs des variables après l'exécution de l'opération. Dans la suite, nous supposerons que la condition de liaison est vraie si la précondition est fausse.

LE STYLE RELATIONNEL [10]

Soit V l'ensemble auquel appartient la variable (ou la collection des variables) du modèle. La précondition est exprimée sous la forme d'un sous-ensemble de V: PRE; la condition de liaison par une RELATION BINAIRE de source et destination V: LINK. Nous supposons que le produit cartésien du complémentaire de PRE et de V est inclus dans LINK (cette supposition est équivalente à celle que nous avons faite dans le cas du style prédicatif). Il est évident que ces deux styles sont équivalents, formellement:

$$PRE = \{x \mid pre(x)\}$$

$$LINK = \{x,x' \mid link(x,x')\}$$

$$pre(x) \iff x \in PRE$$

$$link(x,x') \iff x,x' \in LINK$$

LE STYLE IMPERATIF [2, 15, 16, 17]

Dans ce cas, la précondition et la condition de liaison sont intégrées dans une notation qui ressemble à la partie INSTRUCTION d'un très petit langage de "programmation" défini au moyen des bases et des constructeurs suivants:

BASES:	substitution multiple substitution vide
CONSTRUCTEURS:	précondition garde choix borné choix non borné

Les constructeurs sont des généralisations, ainsi que des simplifications, des "commandes gardées" de Dijkstra [6]. Nous appellerons globalement ceux-ci des SUBSTITUTIONS GENERALISEES. Voici une description plus précise des notations utilisées:

x,...,y := E,...,F	Substitution multiple: x,...,y sont des variables distinctes, et E,...,F sont des expressions en nombre égal.
x:=E ||...|| y:=F	Une forme alternative de la notation précédente (nous généraliserons cette notation lors de la section 1.4).
skip	Substitution vide.
P | S	Précondition: P est un prédicat du premier ordre, et S une substitution généralisée.
P ==> S	Garde: P est un prédicat du premier ordre, et S une substitution généralisée.
S [] T	Choix borné: S et T sont deux substitutions généralisées.
@z.S	Choix non borné: z est une variable distincte de celles du modèle, et S une substitution généralisée.

Nous suivrons désormais la convention syntaxique suivante: les opérateurs binaires sont ici cités par ordre de priorité croissante:

| [] ==> || :=

Toute substitution généralisée définit un TRANSFORMATEUR DE PREDICAT (une fonction de prédicat) qui associe à toute postcondition R la PLUS FAIBLE PRECONDITION, notée [S]R, permettant de garantir que R est vraie juste après l'exécution de l'opération. Lorsque c'est le cas, on dit que S ETABLIT R. Voici les définitions (récursives) de ces transformateurs de prédicats appliqués à une postcondition R:

[x,...,y := E,...,F]R	<=>	R dont les occurences libres des variables x,...,y ont été remplacées respectivement et SIMULTANEMENT par les expressions E,...,F.
[skip]R	<=>	R
[P | S]R	<=>	P & [S]R
[P ==> S]R	<=>	P => [S]R

[S [] T]R	<=>	[S]R & [T]R
[@z.S]R	<=>	∀z.[S]R (où z n'a pas d'occurence libre dans R)

Nous allons maintenant citer certaines propriétés générales découlant de ces définitions. Ces propriétés ne seront pas utilisées directement en pratique, seulement indirectement par l'intermédiaire de leurs conséquences.

La première propriété correspond à la possibilité de mettre toute substitution généralisée sous une FORME NORMALE: plus précisément, il est possible de montrer (par induction structurelle) que toute substitution généralisée S portant sur la variable (ou collection de variables) x peut être mise sous la forme suivante, où A(x), B(x,x') et C(x) sont trois prédicats du premier ordre:

$$A(x) \mid @x'.(B(x,x') ==> x := x') [] C(x) ==> skip \qquad (1)$$

On peut remarquer que cette formule utilise tous les constructeurs que nous avons définis. A partir de ce résultat, nous pouvons déduire une seconde propriété constituée par l'identité suivante:

$$S = [S]true \mid @x'.(not\ [S](x \neq x') ==> x := x') \qquad (2)$$

Finalement, notre dernière propriété montre que le style prédicatif défini ci-dessus et le présent style impératif sont EQUIVALENTS. Plus précisément, en supposant que nous ayons une précondition pre(x) et une condition de liaison link(x,x'), la substitution généralisée correspondante s'écrit:

$$pre(x) \mid @x'.(link(x,x') ==> x := x') \qquad (3)$$

Réciproquement, soit S une substitution généralisée, alors la précondition et la condition de liaison correspondantes s'écrivent:

pre(x)	<=>	[S]true	(4)
link(x,x')	<=>	not [S](x ≠ x')	

1.3 COHERENCE DU MODELE

Après avoir défini un modèle animé, on peut essayer de prouver qu'il est cohérent; ceci nécessite deux sortes de preuves.

D'abord, il faut prouver que le modèle n'est pas "vide". Pour cela, nous définissons une opération spéciale, appelée INITIALISATION, et nous devons prouver qu'elle établit l'invariant. Ceci garantit que le modèle n'est pas vide à condition que l'initialisation ne soit pas contradictoire. Une opération non contradictoire est dite "faisable"; au contraire, une opération établissant une contradiction est dite "miraculeuse" (capable d'établir N'IMPORTE QUELLE postcondition) [15, 17].

Ensuite, il faut prouver que chaque opération CONSERVE L'INVARIANT sous l'hypothèse de sa précondition. Soit I(x) l'invariant du modèle et soit une opération définie avec la précondition pre(x) et la condition de liaison link(x,x') ; l'obligation de preuve de conservation d'invariant peut alors être définie de la manière suivante:

$$I(x) \;\&\; pre(x) \;\Rightarrow\; \forall x'.(link(x,x') \Rightarrow I(x')) \qquad (5)$$

On peut montrer, en utilisant les formules (2) et (4), que cette condition est équivalent à la suivante, rédigée en style impératif:

$$I(x) \;\&\; [S]true \Rightarrow [S]I(x)$$

En résumé, un modèle animé est constitué des éléments suivants:

- ses paramètres génériques,
- ses variables,
- son invariant,
- son initialisation,
- ses opérations.

1.4 CONSTRUCTION DE LA SPECIFICATION D'UN GRAND MODELE

En rédigeant les spécifications de systèmes réels, nous nous trouvons tôt ou tard confrontés au problème de la TAILLE de nos modèles mathématiques ainsi qu'à la taille des preuves de cohérence correspondantes. Nous avons évidemment besoin d'un mécanisme de structuration nous permettant de CONSTRUIRE DE GRANDS MODELES à partir d'autres plus petits et également de construire des PREUVES DE GRANDE TAILLE à partir d'autres plus petites. Le principal problème est le nombre de variables: un modèle contenant jusqu'à 7 variables est compréhensible, tandis qu'un modèle contenant plus de 7 variables a de bonnes chances de ne pas l'être.

Nous allons montrer ici comment il est possible de construire des spécifications de grande taille à partir d'autres plus petites. La démarche proposée consiste à étudier D'ABORD comment simplifier les preuves et à en déduire ensuite une technique permettant de construire de grands modèles.

Comme nous l'avons vu à la section précédente, la forme générale de l'obligation de preuve de préservation d'invariant, pour un modèle dont l'invariant est I(x) et pour une opération dont la précondition est pre(x) et la condition de liaison link(x,x'), est la suivante:

$$I(x) \;\&\; pre(x) \;\Rightarrow\; \forall x'.(link(x,x') \Rightarrow I(x'))$$

Supposons maintenant que la collection des variables du modèle soit constituée de deux parties DISJOINTES x et y; en outre, supposons que l'invariant puisse être mis sous la forme I1(x) & I2(y), où y n'a pas d'occurence libre dans I1(x) et où x n'a pas d'occurence libre dans I2(y); supposons enfin que l'opération considérée possède une précondition de la forme pre1(x) & pre2(y) et une condition de liaison de la forme link1(x,x') & link2(y,y'). L'obligation de preuve de conservation d'invariant devient alors:

$$I1(x) \;\&\; I2(y) \;\&\; pre1(x) \;\&\; pre2(y) \Rightarrow \forall x',y'.(link1(x,x') \;\&\; link2(y,y') \Rightarrow I1(x') \;\&\; I2(y'))$$

Mais, lorsque l'invariant, la précondition et la condition de liaison ont les formes spéciales énoncées ci-dessus, on peut démontrer (en utilisant les règles du Calcul des Prédicats) que la grande preuve ci-dessus se ramène aux deux preuves plus simples suivantes:

```
I1(x) & pre1(x) => ∀x'.(link1(x,x') => I1(x'))
&
I2(y) & pre2(y) => ∀y'.(link2(y,y') => I2(y'))
```

Ce simple résultat nous donne la clé de la démarche à utiliser pour structurer de grandes spécifications: l'idée est de JUXTAPOSER deux (ou plusieurs) modèles animés possédant des collections de variables disjointes. Par "juxtaposition" de deux modèles, nous entendons la concaténation de leurs variables et la conjonction de leurs invariants (on peut remarquer que, lors de la juxtaposition, on peut aussi, en même temps, ajouter un invariant de "collage"). En ce qui concerne les opérations, nous pouvons soit les prendre telles qu'elles apparaissent dans leur modèle respectif, soit renforcer leur précondition, soit en combiner plusieurs au moyen de "gardes" et de "choix bornés", soit finalement, et c'est là le plus important et le plus intéressant, en combiner deux (appartenant à des modèles différents) en faisant respectivement la CONJONCTION DE LEURS PRECONDITIONS ET DE LEURS CONDITIONS DE LIAISON.

Il faut remarquer que cette dernière combinaison d'opérations (conjonction des préconditions et des conditions de liaison) est en fait une GENERALISATION DE L'OPERATEUR || DE PARALLELISME que nous avons introduit précédemment (section 1.2) en tant que "sucre syntaxique" pour la substitution multiple. Nous allons donner une définition plus formelle de cette remarque informelle. Soient S et T deux substitutions généralisées travaillant sur des variables distinctes (respectivement x et y) et mises sous forme normale (1):

```
S = A(x) | @x'.(B(x,x') ==> x := x') [] C(x) ==> skip

T = P(y) | @y'.(Q(y,y') ==> y := y') [] R(y) ==> skip
```

Leur combinaison "multiple" (ou parallèle) S || T est définie comme suit:


```
S || T = A(x) & P(y) |
         @x',y'.(B(x,x') & Q(y,y') ==> x,y := x',y') []
         @x'.(B(x,x') & R(y) ==> x := x')            []
         @y'.(Q(y,y') & C(x) ==> y := y')            []
         C(x) & R(y) ==> skip
```

Comme nous pouvons le voir d'après la formule (4), la précondition de S || T est bien la conjonction des préconditions de S et T, à savoir A(x) & P(y). On peut également, en utilisant (4), calculer la condition de liaison de S || T, soit:

```
(B(x,x') & Q(y,y'))              ou
(B(x,x') & R(y) & y=y')          ou
(Q(y,y') & C(x) & x=x')          ou
(C(x) & R(y) & x=x' & y=y')
```

qui peut s'écrire sous la forme suivante:

```
(B(x,x') ou (C(x) & x=x')) & (Q(y,y') ou (R(y) & y=y'))
```

Cette dernière formule exprime exactement la conjonction des conditions de liaison de S et T prises individuellement: nous pouvons en conclure que la définition de l'opérateur || que nous avons proposée possède bien la propriété désirée. Cependant, cette définition n'est pas très utile en pratique car elle nécessite de mettre les deux substitutions sous forme normale pour pouvoir les juxtaposer au moyen de l'opérateur ||. Heureusement, à partir de cette définition très générale, il est possible de déduire les propriétés suivantes qui sont conformes à l'intuition et d'un emploi commode:

```
S || T            =  T || S
S || (T || U)     =  (S || T) || U
S || skip         =  S
x := E || y := F  =  x,y := E,F
S || (P | T)      =  P | (S || T)
S || (P ==> T)    =  P ==> (S || T)   (si [S]true est vrai)
S || (T [] U)     =  (S || T) [] (S || U)
```

S || @z.T = @z.(S || T) (si z n'a pas d'occurence libre dans T ni dans S)

La forme de ces lois indique clairement que l'opérateur || est une COMMODITE D'ECRITURE et qu'il peut toujours être éliminé: en d'autres termes, après avoir juxtaposé deux opérations au moyen de cet opérateur, on peut toujours éliminer toutes les occurences de ce dernier.

Une autre propriété très importante de cet opérateur "multiple" est la suivante: soient S et T deux substitutions généralisées travaillant sur deux collections de variables disjointes x et y respectivement, et soient deux prédicats I(x) et J(y) tels que y n'ait pas d'occurence libre dans I(x) et que x n'ait pas d'occurence libre dans J(y), nous avons alors:

[S]I(x) & [T]J(y) => [S || T](I(x) & J(y))

Nous pouvons alors voir que la preuve de la condition "complexe" [S || T](I(x) & J(y)) peut être ramenée à deux preuves plus simples, à savoir [S]I(x) et [T]J(y); ceci est dû au fait que S et T travaillent sur des collections de variables DISJOINTES et que la condition que l'on doit établir peut être mise sous forme de la conjonction de deux prédicats ne contenant chacun qu'une seule de ces collections de variables. Il faut remarquer finalement que nous avons les propriétés triviales suivantes:

[S]I => [P ==> S]I

[S]I & [T]I => [S [] T]I

Tous ces résultats nous montrent que, quelle que soit la manière dont nous combinons les opérations des modèles de base, lors de la juxtaposition de ceux-ci, les preuves de conservation d'invariant faites individuellement pour chaque modèle sont SUFFISANTES pour assurer que les nouvelles opérations conservent le nouvel invariant général, à une exception près: l'invariant de "collage" pour lequel toutes les preuves sont inévitablement à faire en totalité. Nous sommes arrivés au but que nous nous étions fixé: n'utiliser qu'un SEUL MECANISME pour simplifier la construction de grands modèles et en même temps simplifier leurs preuves de cohérence. La figure 2 décrit ce mécanisme:

UN MODELE CONSTRUIT A PARTIR DE DEUX AUTRES

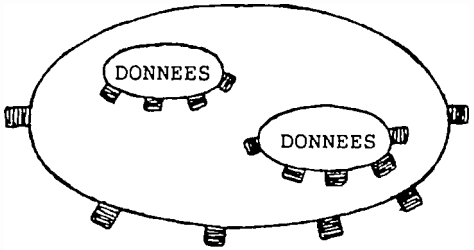

Figure 2

Il faut remarquer que les opérations du modèle externe peuvent "lire" directement les données des modèles internes; cependant, elles ne peuvent pas les modifier directement: ceci ne peut être accompli qu'indirectement au moyen des opérations offertes par les modèles internes (comme nous l'avons déjà expliqué, en procédant de la sorte, les preuves de conservation d'invariant faites sur chaque modèle interne assurent la conservation de l'invariant du modèle externe). Nous avons ici une application laxiste du PRINCIPE D'ENCAPSULATION.

Nous pouvons finalement remarquer que les modèles internes peuvent eux-mêmes avoir été construits en utilisant la même approche.

En fait, ces modèles internes ne sont rien d'autre que des ARTIFICES utilisés pour construire commodément de grands modèles. On peut observer que la COMPREHENSION progressive de notre futur système va de son coeur vers ses parties les plus externes: le système est appréhendé en allant DANS LE SENS INVERSE DE CELUI DU FLUX D'INFORMATIONS qui y entrent. Il faut bien se méfier du fait que ceci ne signifie pas que la future CONSTRUCTION du système doive suivre nécessairement la même démarche (voir section 3.3.3).

2 METHODOLOGIE DE CONSTRUCTION DE LOGICIEL: MODULES

Nous tournons maintenant notre attention vers une autre activité, à savoir la CONSTRUCTION DES LOGICIELS. Dans cette section, notre première intention est d'analyser les principaux concepts mis en oeuvre dans la programmation de logiciels de grande taille; ensuite, nous envisagerons les différentes manières de mettre ces concepts en pratique.

Le concept le plus important utilisé dans la programmation de logiciels de grande taille est, à notre avis, celui de MODULE. C'est un concept qui reçoit des noms variés dans différents contextes de programmation: par exemple, on l'appelle une "classe" (SIMULA), un "module" (MODULA), un "package" (ADA), etc... Ce concept est important parce qu'il permet de découper un logiciel de grande taille en morceaux indépendants possédant des interfaces bien définies. Ce concept classique de programmation sera ici défini simplement au moyen des éléments suivants: ses VARIABLES, son INITIALISATION et ses SERVICES.

Un système logiciel de grande taille est généralement composé de plusieurs modules, parfois disposés l'un à côté de l'autre, parfois imbriqués, le plus fréquemment organisés en structures mixtes. L' "utilisateur" d'un module n'est pas autorisé à "lire" ou "modifier" directement les données internes de celui-ci; il ne peut le faire que par l'intermédiaire des services offerts par le module à condition, bien entendu, que le module ait été initialisé auparavant. C'est ce que l'on appelle le PRINCIPE D'ENCAPSULATION (strict).

Chaque service peut avoir deux effets distincts, soit modifier les variables internes du module, soit renvoyer un ou plusieurs résultats (certains services peuvent combiner les deux effets).

Après avoir choisi un langage de programmation, nous pouvons alors construire les différents module d'un système logiciel en DECLARANT leurs variables et en PROGRAMMANT leur initialisation et leurs services. Nous allons maintenant étudier les caractéristiques correspondant à la déclaration et à la programmation. Il faut remarquer que nous n'avons pris en considération que les mécanismes dont l'expérience montre qu'ils sont suffisants en pratique: leur nombre est très restreint, et ils sont présents dans la plupart des langages de programmation impératifs. En fait, nous croyons fermement que le choix d'un langage de programmation est bien moins important que la plupart des gens ne veulent bien le penser (et le dire); nous pensons également que ce choix doit être basé sur des critères de

simplicité et d'efficacité, plutôt que sur la "sophistication" comme c'est trop souvent le cas. On peut aussi s'astreindre à n'utiliser qu'une partie restreinte d'un langage de programmation existant.

De manière traditionnelle, les déclarations définissent les variables d'un programme en leur assignant un TYPE qui peut être construit d'après les bases et constructeurs suivants:

BASES:	entiers réels caractères
CONSTRUCTEURS:	tableau sur un intervalle d'entiers fichier

Il est assez significatif de remarquer que nous n'avons pas de constructeur de type "structure" ("record"); il est encore plus significatif de remarquer que nous n'avons pas de constructeur de type "pointeur"!

D'après la terminologie du langage de programmation PASCAL, les services d'un module sont soit des PROCEDURES (services avec effets secondaires et aucun résultat), soit des FONCTIONS (services avec résultat(s) et effets secondaires éventuels). Ces services peuvent être paramétrés, les paramètres effectifs étant passés PAR VALEUR. Il faut remarquer l'absence de passage de paramètres "par référence" ou "par nom".

Les programmes correspondant à chaque procédure ou fonction sont écrits au moyen d'un répertoire d'INSTRUCTIONS construit à partir des bases et constructeurs suivants:

BASES:	instruction vide affectation dans une variable de type simple affectation dans une case de tableau écriture dans un fichier lecture depuis un fichier
CONSTRUCTEURS:	conditionnelle simple (IF) conditionnelle multiple (CASE) séquence (;) boucle simple (WHILE)

Les deux instructions d'affectation nécessitent la définition d'EXPRESSIONS TYPEES, ce qui est réalisé au moyen des bases et constructeurs suivants:

BASES: constante entière
constante réelle
variable simple

CONSTRUCTEURS: élément de tableau
opérateurs arithmétiques biaisés
(à cause des débordements possibles)
taille de fichier
conversion de type
(de réel vers entier et vice-versa)

Enfin, les constructeurs d'instructions (sauf la séquence) nécessitent la définition d'EXPRESSIONS BOOLEENNES construites à partir des bases et constructeurs suivants:

BASES: comparaisons de valeurs d'expressions ayant même type de base

CONSTRUCTEURS: conjonction booléenne
disjonction booléenne
négation booléenne

3 SYNTHESE DES MODELES ET DES MODULES

Nous avons l'intention de montrer dans cette section comment il est possible d'unifier les notions de MODELES et de MODULES. Pour cela, nous étudions d'abord leurs différences (section 3.1) de manière à les unifier (section 3.2), enfin nous examinons le passage systématique de l'un à l'autre (section 3.3).

3.1 ANALOGIES ET DIFFERENCES

Il est clair qu'il existe des ressemblances entre les modèles et les modules: ils ont tous des "variables"; ils ont tous une initialisation; les invariants des modèles et les déclarations des modules sont par nature similaires; l'instruction d'affectation dans une variable simple ressemble fortement à la substitution simple (ce qui a permis d'unifier leurs notations); enfin, les opérations des modèles et les services des modules sont très proches les uns des autres.

Cependant, certaines notions présentes dans les modèles n'ont pas d'équivalent dans les modules; ce sont: les paramètres génériques, les préconditions, les gardes, et enfin les choix bornés et non bornés. Dans les modèles, nous avons en outre à notre disposition la logique du premier ordre et la théorie des ensembles dans toute leur généralité; il est évident que de telles possibilités n'existent pas dans les langages de programmation.

Réciproquement, deux concepts de programmation n'ont pas d'équivalent dans les modèles: la séquence et la boucle. On peut noter que les instructions conditionnelles simple et multiple peuvent être formalisées en termes de gardes et de choix bornés.

Pour ce qui est des "types", il est clair qu'ils peuvent être formalisés dans le cadre de la théorie des ensembles. Par exemple, le type "entier" peut être formalisé par un sous-ensemble des entiers naturels muni d'une arithmétique biaisée (à cause des dépassements de capacité). Le type "réel" peut être formalisé par un sous-ensemble des nombres RATIONNELS (fractions d'entiers) muni lui aussi d'une arithmétique biaisée. Le constructeur "tableau sur un intervalle d'entiers" peut être formalisé en par une fonction totale construite sur cet intervalle. Enfin, les "fichiers" peuvent être simplement formalisés par des suites finies.

Les expressions typées utilisées en programmation peuvent être formalisées par les expressions correspondantes dans la théorie des ensembles. Enfin, les expressions booléennes peuvent être formalisées au moyen de prédicats de la logique des propositions (pas de quantification).

3.2 UNIFICATION DES DEUX CONCEPTS: MACHINES ABSTRAITES

Comme nous l'avons vu dans la section précédente, les modules sont presque des modèles: seulement "presque", car les modèles n'ont ni séquence ni boucle. Ajoutons conceptuellement ces deux caractéristiques aux modèles.

De tels modèles étendus sont appelés des MACHINES ABSTRAITES.

L'introduction de la séquence et de la boucle dans les modèles entraîne certaines difficultés théoriques, notamment dans le cas de la boucle. Les "instructions" de nos modèles ainsi étendus (machines abstraites) n'ont aucune raison d'être continues en général à cause du non-déterminisme non borné [7]. Il en résulte que les techniques classiques permettant de définir le transformateur de prédicat associé à une boucle ne sont plus utilisables [6]. Il faut le définir d'une autre manière [3, 5, 8]. Le seul résultat non conforme à l'intuition est le suivant: certaines boucles peuvent "se terminer" après un nombre d'itérations qui ne peut pas être borné par un nombre calculable à l'avance en fonction des valeurs initiales des variables. C'est notamment le cas lorsque le "corps" de la boucle contient du non-déterminisme non borné. Pour prouver la cohérence de la théorie, il faut aussi montrer que les propriétés des substitutions généralisées (forme normale, etc..) sont également valables pour la séquence et la boucle, mais ceci sortirait du cadre du présent exposé. Les définitions formelles de la séquence et de la boucle sont données dans l'Appendice.

3.3 DES MODELES AUX MODULES

Dans la section précédente, nous avons vu comment les concepts des modèles et des modules peuvent être unifiés dans le cadre des machines abstraites. Dans ce qui suit, nous considèrerons encore des modèles et des modules, mais dans le sens restreint suivant:

Un MODELE est une machine abstraite dont la spécification des opérations ne contient ni séquence ni boucle.

Un MODULE est une machine abstraite possèdant les restrictions suivantes: d'abord, son invariant implique que chaque variable appartient à un ensemble qui se trouve être la formalisation d'un "type"; ensuite, ses opérations ne contiennent que des "instructions" de programmation comportant des prédicats qui se trouvent être des formalisations d' expressions booléennes.

Dans cette section, nous allons voir comment nous pouvons passer d'un modèle à un module; cette activité est appelée CONCEPTION (section 3.3.1). Nous allons aussi étudier les modalités pratiques de cette activité; on peut mettre en oeuvre ces modalités soit en travaillant sur les données (section 3.3.2), soit en travaillant sur les opérations (section 3.3.3).

3.3.1 CONCEPTION

La conception est un processus réalisé au moyen d'un nombre fini d'étapes. A chaque étape est associée une machine abstraite. Nous partons d'un modèle et arrivons finalement à un module, après avoir éventuellement rencontré au passage certaines machines abstraites intermédiaires. La figure 3 illustre ce processus:

Figure 3

Deux machines abstraites qui se suivent dans la séquence présentée ci-dessus sont reliées par une relation d'ordre partiel appelée RAFFINEMENT; cette relation généralise aux machines abstraites la notion de "satisfaction" présentée informellement à la section 1.2.1. Deux machines reliées par cette relation de raffinement ont évidemment des OPERATIONS SIMILAIRES; nous entendons par là: même nom, même sorte (ou absence) de paramètres, même sorte (ou absence) de résultats.

De manière informelle, nous dirons qu'une machine abstraite M est raffinée par une machine abstraite N si toute INTERFACE travaillant avec ses propres variables et appelant les opérations de la machine M (au préalable initialisée) obtient des RESULTATS COMPARABLES à ceux obtenus en appelant les opérations correspondantes de la machine N (au préalable

initialisée). Nous entendons par "résultats comparables" que les résultats produits de manière non-déterministe par N sont aussi des résultats produits de la même manière par M. La réciproque est en général fausse: M peut produire certains résultats que N est incapable de produire. Ceci est dû au fait que N peut être MOINS NON-DETERMINISTE que M.

Cette définition très générale [10] n'est cependant pas d'un grand intérêt pratique. Les sections suivantes ont pour but de montrer comment il est possible d'améliorer cet état de choses.

3.3.2 RAFFINEMENT PAR LES DONNEES

Pour raffiner une machine abstraite, nous proposons la démarche suivante: d'abord, nous procédons à un CHANGEMENT DE VARIABLE défini au moyen d'une condition, appelée "condition d'abstraction" [13], reliant les anciennes et les nouvelles variables; ensuite, nous proposons une nouvelle machine abstraite ne travaillant qu'avec les nouvelles variables; enfin, nous vérifions, comme il sera expliqué plus loin, que la nouvelle machine raffine l'ancienne.

Pendant très longtemps [13], on a cru que la condition d'abstraction devait être fonctionnelle et "surjective" des nouvelles variables vers les anciennes; en d'autres termes, toutes les valeurs possibles des variables de l'ancienne machine devaient pouvoir être définies fonctionnellement à partir des variables de la nouvelle. Cette condition est certainement nécessaire dans le cas d'un "changement de variable" en mathématiques (par exemple pour calculer la valeur d'une intégrale). Récemment [9, 10,15], il est devenu clair que cette contrainte pouvait être abandonnée au risque, évidemment, d'une certaine "perte d'information" dans la nouvelle machine; mais cela ne présente aucun inconvénient si l'information en question n'est pas utilisée dans la définition des opérations de l'ancienne machine.

Plus précisément, soit M une machine abstraite possédant les éléments suivants:

variable	x
invariant	I(x)
opération	S (avec la précondition P(x))

Supposons d'autre part que cette machine soit "cohérente" (section 1.3) en d'autres termes, qu'elle conserve l'invariant sous l'hypothèse de la précondition; soit formellement en style impératif:

$$I(x) \;\&\; P(x) \;\Rightarrow\; [S]I(x)$$

Nous proposons les éléments suivants:

variable	y
condition d'abstraction	J(x,y)
opération	T

La preuve du raffinement de l'opération S par T consiste à montrer que, sous la triple hypothèse de l'invariant, de la précondition de S et de la condition d'abstraction, la nouvelle opération T établit que l'ancienne opération S N'ETABLIT PAS LA NEGATION de la condition d'abstraction [9, 18], soit:

$$I(x) \;\&\; P(x) \;\&\; J(x,y) \;\Rightarrow\; [T]\mathrm{not}[S]\mathrm{not}(J(x,y))$$

La double négation garantit que la nouvelle opération T est moins non-déterministe que l'ancienne opération correspondante S. Si cette preuve peut être établie pour tout couple (S,T) d'opérations correspondantes, et si une preuve analogue peut l'être pour les initialisations, nous pouvons alors montrer [4] que N est une machine abstraite qui raffine M suivant la définition informelle donnée à la section 3.1. La nouvelle machine N peut à son tour être raffinée de manière analogue, et ainsi de suite.

3.3.3 DECOMPOSITION DES OPERATIONS

L'expérience montre que la technique de raffinement que nous avons vue à la section précédente ne peut pas être appliquée sur un grand nombre de niveaux. Ce phénomène peut avoir plusieurs causes:

- il est difficile de raffiner une machine abstraite qui n'est pas un modèle; en conséquence, l'introduction d'une séquence ou d'une boucle dans une opération de la nouvelle machine "bloque" toute possibilité de raffinement ultérieur,

- lorsque le nombre de niveaux s'accroît, la complexité de l'invariant et des opérations augmente également, ce qui fait que nous nous trouvons rapidement confontés à une situation difficile à maîtriser.

Nous devons alors utiliser une nouvelle technique, celle du RAFFINEMENT PAR PARTIES, qui sera exposée informellement dans le reste de cette section.

Il est possible de montrer [4] que chaque instruction de notre petit répertoire est monotone par rapport à la relation de raffinement; d'un point de vue pratique, ceci signifie que le raffinement d'une instruction composée peut être accompli en raffinant chacune des instructions qui le compose. Ceci nous donne une solution capable de mettre un frein à la complexité dangereusement croissante évoquée plus haut: à un certain niveau, plutôt que de raffiner directement une machine complexe, nous raffinons les "parties" élémentaires de ses opérations. De plus, nous regroupons ces parties de telle manière qu'elles forment les opérations d'une (ou plusieurs) machine(s). Nous dirons que l'ancienne machine est IMPLANTEE sur les nouvelles. Ces dernières peuvent être soit totalement nouvelles, soit DEJA EXISTANTES. On peut noter que la seule chose qui nous intéresse pour l'instant dans ces nouvelles machines est leur SPECIFICATION (modèle). La figure 4 illustre notre propos:

UNE MACHINE IMPLANTEE
SUR LES MODELES DE
DEUX AUTRES

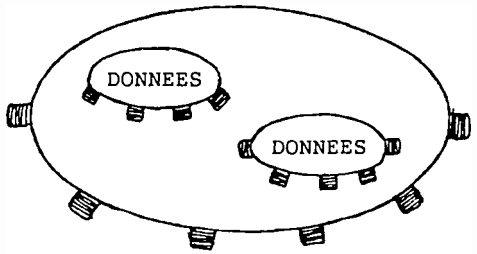

Figure 4

Comme nous pouvons le remarquer, cette figure est la même que celle de la figure 2 (section 1.4). Mais, contrairement à ce qui avait été proposé dans le cas de la construction de grands modèles lors de la JUXTAPOSITION de plus petits, le PRINCIPE D'ENCAPSULATION doit maintenant être appliqué strictement: cette fois-ci, une opération de la machine externe ne peut pas "lire"

directement les données des modèles internes; ceci ne peut être accompli qu'au moyen des opérations offertes par ces modèles. Cette contrainte vient du fait que nous voulons pouvoir ultérieurement RAFFINER les modèles internes. Il faut également noter que cette décomposition n'a aucune raison d'être la même que celle utilisée lors de la juxtaposition: la COMPREHENSION progressive d'un système est distincte de sa CONSTRUCTION progressive.

Nous nous retrouvons désormais confrontés à un nouveau problème, à savoir le raffinement des modèles internes. Il faut remarquer que, s'il s'agit d'un modèle déjà étudié, nous pouvons utiliser un raffinement existant. Sinon, nous pouvons utiliser la technique de la section précédente jusqu'à ce que nous arrivions de nouveau à une situation "bloquée" ou dangereusement complexe, auquel cas nous utiliserons l'approche développée dans cette section, etc...

Cette technique correspond à une méthode bien connue consistant à organiser un vaste système logiciel en COUCHES. Mais il est également bien connu que cette méthode possède un SERIEUX INCONVENIENT: les systèmes programmés de cette façon sont notoirement lents à cause du temps passé dans la traversée des différentes couches. Un "remède" bien connu à cette difficulté consiste à "tricher" en ne respectant pas l'opacité des couches: le résultat en est une violation du principe d'encapsulation. Par exemple, pour atteindre directement une opération située dans une couche "profonde", on "saute" par dessus plusieurs couches sans payer le prix des opérations intermédiaires dont le rôle peut consister seulement à passer la main à la couche suivante. Cette sorte de manoeuvre est une pratique dangereuse susceptible d'introduire progressivement le chaos dans un système originellement bien conçu. En effet, on perd ainsi tous les avantages de la structure en couches:

- en ce qui concerne la mise au point du système: elle ne peut plus se faire couche par couche,

- en ce qui concerne la maintenance, le tableau est encore plus dramatique: la modification d'une couche intermédiaire risque de rendre incorrectes toutes les tricheries consistant à "sauter" par dessus une de ses opérations.

Dans le cas présent, NOUS N'AVONS PAS A FAIRE FACE A CETTE DIFFICULTE, car les couches que nous considérons ne sont pas des couches de programmes, mais des COUCHES DE MACHINES ABSTRAITES. En fait, les seules opérations effectivement programmées sont celles constituées d'instructions de programmation et d'appels à d'autres opérations; très souvent, les opérations peuvent être "programmées" sous forme de MACRO-INSTRUCTIONS, ce qui permet d'éliminer lors de l'exécution le temps inutile évoqué ci-dessus tout en préservant la structure en couches du logiciel.

4 CONCLUSION

Dans cette courte synthèse, nous avons montré que la théorie de la construction de logiciel existe et peut être appliquée pratiquement.

Cette théorie possède ses propres techniques qui sont totalement différentes de celles de la théorie de la programmation: nous n'avons parlé ni de récursivité ni d'algorithmique, par contre nous avons parlé de preuves (de cohérence et de raffinement).

La conception d'un système logiciel pourrait bien n'être rien d'autre que la preuve de la validité de sa construction.

REMERCIEMENTS

Ce travail a été financé par British Petroleum. Je voudrais remercier Ib Sorensen (de BP) et les membres de son équipe pour les discussions nombreuses et très utiles que nous avons eues. Les apports techniques du Programming Research Group de l'Université d'Oxford ont également été importants, en particulier ceux de Paul Gardiner. Finalement, je voudrais remercier Guy Laffitte et Jean Contensou de l'INSEE.

Références bibliographiques

[1] Abrial J.R. 1988 Formal Introduction to Set Notations

[2] Abrial J.R. 1988 Abstract Machines: Basic Concepts

[3] Abrial J.R. 1988 Abstract Machines: Programming Concepts

[4] Abrial J.R. 1988 Abstract Machines: Refinement

[5] Boom H.J. 1982 A weaker precondition for loops. ACM TOPLAS 4

[6] Dijkstra E.W. 1976 A discipline of programming (Prentice-Hall)

[7] Dijkstra E.W. 1982 The equivalence of bounded nondeterminacy and continuity. In Selected writings on computing: a personal perspective (Springer-Verlag)

[8] Dijkstra E.W. 1986 A simple fixpoint argument without the restriction to continuity. University of Austin, Texas

[9] Gries D. 1985 A new notion of encapsulation. SIGPLAN Notices July 1985

[10] He J. et al. 1986 Data refinement refined. PRG Oxford University

[11] Hehner E.C.R. 1984 Predicative programming part 1. Comm. ACM 27

[12] Hoare C.A.R. 1985 Programs are predicates. In Mathematical Logic and Programming Languages (Prentice-Hall)

[13] Jones C.B. 1986 Systematic software development using VDM (Prentice-Hall)

[14] Kowalski R. 1985 The relation between logic programming and logic specification. In Mathematical Logic and Programming Languages (Prentice-Hall)

[15] Morgan C. 1986 The specification statement. PRG Oxford University

[16] Morris J.M. 1986 A theoretical basis for stepwise refinement and the programming calculus. University College, Dublin

[17] Nelson J. 1987 A generalization of Dijkstra's calculus. DEC Systems Research Center, Palo Alto, California

[18] Robinson K. 1986 From specifications to programs. PRG Oxford University

[19] Turner D.A. 1985 Functional programs as executable specifications. In Mathematical Logic and Programming Languages (Prentice-Hall)

[20] Wirth N. 1971 Program development by stepwise refinement. Comm. ACM 14

APPENDICE: Définitions formelles de la séquence et de la boucle

LA SEQUENCE

Soient S et T deux substitutions généralisées et R une postcondition, nous avons la définition triviale suivante:

$$[S\ ;\ T]R \quad <=> \quad [S][T]R \qquad (1)$$

LA BOUCLE

Nous allons utiliser à la fois le style "relationnel" et le style "impératif". Soit S une substitution généralisée et x la variable d'état, dont les valeurs possibles sont supposées appartenir à un ensemble V. Nous utilisons les notations mathématiques suivantes:

inc inclusion ensembliste

inter(E) intersection généralisée de l'ensemble NON VIDE E

Nous définissons alors l'ensemble WLF(S) comme suit:

$$WLF(S) = inter(\{s \mid (s\ inc\ V)\ \&\ (\{x \mid [S](x \in s)\}\ inc\ s)\})$$

Nous pouvons remarquer que cette définition est licite: en effet, l'ensemble dont on prend l'intersection généralisée n'est pas vide, car il possède au moins V comme élément.

D'autre part, nous ADMETTONS que la fonction qui, à une partie s de V, fait correspondre la partie de V définie par la formule $\{x \mid [S](x \in s)\}$ est monotone pour la relation d'inclusion. Nous pouvons vérifier que c'est bien le cas pour toutes les constructions que nous avons introduites jusqu'à présent, il restera à le montrer pour la boucle que nous sommes précisément en train de construire maintenant: c'est l'"hypothèse d'induction structurelle". Nous pouvons alors appliquer le

théorème de TARSKI nous disant que WLF(S) est le plus petit point fixe de la fonction citée ci-dessus; ceci nous donne:

$$WLF(S) = \{x \mid [S](x \in WLF(S))\}$$

D'après l'axiome d'extensionnalité, il vient:

$$x \in WLF(S) \iff [S](x \in WLF(S))$$

Si nous définissons le prédicat wlf(S) comme $(x \in WLF(S))$, nous obtenons:

$$wlf(S) \iff [S]wlf(S) \qquad (2)$$

Nous définissons alors la construction S* au moyen du transformateur de prédicat suivant:

$$[S^*]R \iff wlf(R \mid S) \qquad (3)$$

(cette définition est dûe à Paul Gardiner)

Il est ALORS possible de prouver que, sous l'hypothèse d'induction structurelle, la fonction qui, à une partie s de V, fait correspondre la partie de V définie par $\{x \mid [S^*](x \in s)\}$ est également monotone pour la relation d'inclusion. Ceci résulte du fait que WLF(S) est un plus petit point fixe.

Enfin, soient P un prédicat et S une substitution généralisée, nous allons définir les constructions de boucle "WHILE P DO S END" et conditionnelle "IF P THEN S END" comme suit:

$$\text{WHILE P DO S END} = ((P \Longrightarrow S)^* \; ; \; not(P) \Longrightarrow skip) \qquad (4)$$

$$\text{IF P THEN S END} = (P \Longrightarrow S \; [] \; not(P) \Longrightarrow skip) \qquad (5)$$

Nous allons vérifier que la construction de boucle possède bien la propiété de "dépliage" bien connue:


```
WHILE P DO S END  =  IF P THEN
                        S ;
                        WHILE P DO S END
                     END
```

Pour montrer que ces deux substitutions sont égales, il suffit de montrer qu'elles ont la même plus faible précondition pour une postcondition R quelconque. Voici la démonstration formelle:

```
    [WHILE P DO S END]R
<=> [(P ==> S)* ; not(P) ==> skip]R                    d'après (4)
<=> [(P ==> S)*][not(P) ==> skip]R                     d'après (1)
<=> [(P ==> S)*](not(P) => [skip]R)
<=> [(P ==> S)*](not(P) => R)
<=> wlf((not(P) => R) | P ==> S)                       d'après (3)
<=> [(not(P) => R) | P ==> S]wlf((not(P) => R) | P ==> S)
                                                       d'après (2)
<=> [(not(P) => R) | P ==> S][WHILE P DO S END]R
<=> (not(P) => R) & [P ==> S][WHILE P DO S END]R
<=> (not(P) => R) & (P => [S][WHILE P DO S END]R)
<=> (P => [S][WHILE P DO S END]R) & (not(P) => R)
<=> (P => [S][WHILE P DO S END]R) & (not(P) => [skip]R)
<=> (P => [S ; WHILE P DO S END]R) & (not(P) => [skip]R)
<=> [P ==> (S ; WHILE P DO S END)]R & [not(P) ==> skip]R
<=> [IF P THEN (S ; WHILE P DO S END) END]R        d'après (5)
```

C.Q.F.D.