No Ordre : 2752

THÈSE

Présentée

DEVANT L'UNIVERSITÉ DE RENNES 1

Pour obtenir

le grade de DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention Informatique

PAR

VU LÊ H?NH

Équipe d'accueil	Triskell
École doctorale	MATISSE
Composante universitaire	IFSIC/IRISA

TITRE DE LA THÈSE :

Test et modèle UML : stratégies de planification des tests

Soutenue le 5e Décembre 2002 devant la commission d'Examen

COMPOSITION DU JURY

M. Olivier RIDOUX, Président du jury
M. Jean-Marc JEZEQUEL, Directeur de thèse
Mme. Chantal ROBACH, Rapporteur
M. Jean-Marc GEIB, Rapporteur
M. Yves LE TRAON, Examinateur
M. Nicolas BULTEAU, Examinateur

IRISA
institut de recherche en informatique et systèmes aléatoires

Remerciements

Tout d'abord, je voudrais remercier M. Olivier Ridoux qui m'a fait l'honneur de présider le jury de ma soutenance. Je voudrais aussi remercier Mme Chantal Robach et M. Jean-Marc Geib, qui m'ont fait l'honneur d'accepter d'évaluer ma thèse.

Je voudrais remercier M. Jean-Marc Jézéquel qui a accepté de diriger ma thèse.

Mon plus grand remerciement est réservé à M. Yves Le Traon pour sa patience. C'est lui qui m'a aidé à corriger la plupart des fautes de français de ma thèse. Je voudrais remercier également Clémentine Nébut, Benoit Baudry, Loïc Helouet, Romain Sagnimorte et Simon Pickin pour l'aide qu'ils m'ont apporté.

Je voudrais remercier l'équipe d'assistance de projet : Marie-Noëlle Georgeault qui m'a aidé tout au long de mon séjour en France ; Stéphanie Lemaile et Ludivine Quoniam qui ont dû prendre des heures supplémentaires pour m'aider à envoyer à temps les copies de ma thèse aux rapporteurs.

Je voudrais remercier tous les membres de l'ancienne équipe Pampa et de la nouvelle équipe Triskell, Claude Jard, Thierry Jéron, François Pennaneac'h, Alain Le Guennec, Wai-Ming Ho, les gens qui m'ont accueilli ici, à Rennes à mon arrivée.

Enfin, je voudrais remercier la Région Bretagne et la société Softeam, pour la bourse et l'IRISA pour ses supports matériels et administratifs, sans lesquelles cette thèse n'aurait pu être rédigée.

Résumé

Dans un système à objets, la plupart des unités (souvent des classes) utilisent des services (des attributs, des méthodes) d'autres unités. Quand on teste une unité, les services utilisés par cette unité doivent être accessibles.

Il existe deux façons principales de faire le test d'intégration d'un système à objets : soit, on teste toutes les unités d'un seul coup, sans aucune étape intermédiaire (intégration « *Big-Bang* ») ; soit on intègre les unités par étapes en commençant par les unités qui « *n'utilisent aucun service des autres unités* », puis les unités qui n'utilisent que des services fournis par les unités précédemment intégrées (intégration progressive de bas en haut). L'intégration progressive est préférable à l'intégration Big-Bang car elle facilite la localisation d'erreurs.

À cause de la forte connectivité d'un système à objets, une unité peut utiliser indirectement ses propres services, à travers une chaîne d'utilisation de services. Dans ce cas, on ne peut pas déterminer quelle l'unité « *n'utilise aucun service des autres unités* » afin de la tester avant des autres. On doit donc introduire des «*bouchons* » afin d'obtenir une telle unité.

Dans cette thèse, nous définissons une manière pour trouver un ordre d'intégration des unités d'un système à objets. Cet ordre tente de minimiser le nombre de bouchons et la durée d'intégration quand le test est effectué par un groupe de plusieurs testeurs.

Resume

In an object-oriented system, most units (usually classes) use services (attributes, methods) of other units. When we test a unit, the services used by it have to be accessible.

There are two principal ways to perform the integration testing for an object-oriented system: either we test all the units of system together without any intermediate stages ("*Big-Bang*" integration); or we integrate units in stages, starting with units which *"do not use any service of other units"* and following with units which use only the services provided by the units previously integrated (bottom–up progressive integration). A progressive integration is preferred to the Big-Bang integration because it facilitates the error localisation.

Because of the strong connectivity of an object-oriented system, a unit is able to indirectly use its own services through a sequence of service uses. in this case, we cannot determine which unit *"does not use any service of other units*" in order to test it before the others. We must thus introduce "*stubs*" to obtain a such unit.

In this thesis, we define a means to find out an integration order for all the units of an object-oriented system. this order attempts to minimize the number of stubs and the integration duration when the test is carried out by a group of several testers.

Table de matière

Table des figures

Table des tableaux

Chapitre I.

Introduction

La qualité d'un produit dépend fortement de la manière dont il a été testé. Pour les logiciels, les tests sont composés de données de test, de scénarios de test et de résultats de test.

En génie logiciel, le test est l'une des procédures les plus coûteuses (avec le « *déboguage* » – recherche et élimination des anomalies de programmation grossière ou « *bogues* »). Les travaux que nous présentons dans ce mémoire ont pour but de proposer une solution afin de diminuer le coût du test d'intégration des logiciels développés avec un langage de programmation à objets.

Dans ce chapitre, nous présentons brièvement le test en général et le test des logiciels à objets en particulier. Nous insistons un peu plus sur le test d'intégration des logiciels à objets, ses contraintes et ses problèmes.

1. Le test de logiciel

Le test constitue le moyen principal de validation d'un logiciel. Le test joue un rôle à la fois dans l'obtention et dans l'évaluation de la qualité d'un logiciel. Il prend place dans le processus de développement et s'effectue par étapes successives :

i. *Le test unitaire* : une unité est la plus petite partie testable d'un programme, c'est souvent une procédure dans les programmes procéduraux ou une classe dans les programmes à objet. L'objectif du test unitaire est de révéler le plus grand nombre de fautes possibles.

ii. *Le test d'intégration* : consiste à assembler plusieurs unités et à tester la collaboration entre les unités du système. Comme dans le test unitaire, le test d'intégration essaie de révéler le plus grand nombre de fautes possibles. Il peut s'agir de fautes non détectées lors de l'étape de test unitaire ou de fautes spécifiques à l'intégration, telles que des incohérences dans les interfaces.

iii. *Le test de système* : il a pour but d'évaluer d'abord des propriétés non strictement fonctionnelles du logiciel comme sa performance, sa compatibilité par rapport à l'environnement de travail plutôt qu'à l'environnement de développement.

iv. *Le test d'acceptation* : ce test est fait avec la participation du client pour obtenir son acceptation et la livraison du logiciel.

À côté de ces étapes, il y a encore un autre type de test, c'est le test de non-régression (« *regression testing* »). Ce type de test est demandé lorsqu'on modifie un module de logiciel (ajout de traitements, optimisation du code…) et qu'on le réintègre dans le système. Ce type de test est très semblable au test d'intégration. La seule différence réside dans le fait que pour le test d'intégration, on doit tester tôt ou tard, toutes les parties du logiciel sous test. Dans le test de non-régression, on ne s'intéresse qu'aux parties ayant subi l'impact de la modification. Notons que l'on doit tester toutes les fonctions du système sous test, aussi bien après le test d'intégration (dans la phase de développement) qu'après le test non-régression (dans la phase de maintenance).

1.a. Cas de test

Quelle que soit la technique de test, tester un logiciel (entier ou partiel) consiste à l'exécuter avec des données d'entrée prédéterminées et à comparer les résultats obtenus avec ceux attendus. Pour les deux premières étapes de test (le test unitaire et le test d'intégration), on ne teste qu'une partie du logiciel et donc l'interface principale du logiciel n'est pas encore utilisée. Il faut créer et utiliser des «*pilotes de test* » (« *test driver* ») qui exécutent les « *scénarios* » de test – séries organisées d'instructions pour saisir les données entrées, provoquer l'exécution du système sous test et enfin, comparer les résultats obtenus avec les résultats attendus. On appelle un « *cas de test* » une composition d'un pilote de test, des données entrées correspondantes et des résultats attendus. Avant d'effectuer un test, il faut donc dans un premier temps avoir construit (ou généré) ces cas de test. Normalement, on les génère dès la phase de conception.

Pour le test unitaire et le test d'intégration, l'objectif est de révéler le plus grand nombre de fautes ayant des conséquences fonctionnelles. Pour cela, les pilotes de test sont exécutés avec l'intention de provoquer des défaillances. Chaque fois qu'il se produit une défaillance, les fautes à l'origine de cette exécution sont recherchées et supprimées. On attend légitimement de la suppression de ces fautes une amélioration de la qualité du logiciel. L'efficacité du travail dépend de l'aptitude des ingénieurs de test à mettre au point des cas de tests ayant une forte probabilité de révéler, par une défaillance, la présence de fautes.

On sait que tous les logiciels non-triviaux contiennent des erreurs. Si toutes les erreurs se manifestaient avec toutes les données d'entrée possibles, il ne faudrait exécuter le test qu'une fois. En général, les erreurs se cachent très «*subtilement* ». Un simple exemple de code C (adopté de [Binder 1999]) est présenté dans la Figure 1.

Dans ce morceau de code, l'instruction «$x = x + 1$ » est accidentellement changée par «$x = x - 1$ ». Supposons que le domaine d'un entier est [-32768, 32767]. Cette erreur n'est révélée qu'avec un des six cas de données entrées (voir le Tableau 1) qu'on appelle les données de bord. La probabilité de révéler cette erreur est donc de $6/65536 \approx 9{,}16.10^{-5}$, soit moins de 0,01%.

```
int seuil (int x)
   {
        // code correct : x  = x + 1;
        x = x – 1; // erreur
        return x / 30000;
    }
```

Figure 1. Erreur dans le code : la probabilité est faible pour la révéler.

Tableau 1. Les cas de test efficace pour le code dans la Figure 1.

X	Résultat attendu	Résultat obtenu
-32768	-1	Débordement
-30000	0	-1
-29999	0	-1
29999	1	0
30000	1	0
32767	Débordement	1

Une différence entre les résultats obtenus et ceux attendus pour les entrées sélectionnées constitue un indice de défaillance. Pour déterminer la différence, il faut avoir au préalable déterminé un « *oracle* » - une fonction logique qui vérifie si les résultats obtenus sont identiques à ceux attendus.

Une défaillance apporte la preuve que l'implémentation sous test est incorrecte. Autrement dit, une défaillance témoigne de la présence, dans le texte du logiciel, des fautes qui ont été introduites lors du codage ou de la conception. Il s'agit alors de corriger les fautes, après les avoir localisées, étape que l'on appelle « *diagnostic* ».

1.b. Critère d'arrêt

Lorsqu'un logiciel est soumis au test, on peut rarement effectuer un test exhaustif sur toutes les données entrées possibles pour s'assurer qu'il ne contient absolument aucune faute.

Un exemple (adopté aussi de [Binder 1999]) : Entrer trois segments de droites $a(ax_1, ay_1, ax_2, ay_2)$, $b(bx_1, by_1, bx_2, by_2)$ et $c(cx_1, cy_1, cx_2, cy_2)$, il faut vérifier si ces trois segments peuvent être les côtés d'un triangle. Supposons que les coordonnées des points soient des entiers dont les valeurs sont comprises entre 1 et n. Le nombre de combinaisons de données d'entrées possibles est de n^{12}. Le Tableau 2 montre le temps nécessaire pour passer les n cas possibles pour chaque coordonnée, avec une vitesse de vérification de 1000 cas chaque seconde. Le résultat montre qu'il est impossible d'effectuer un tel test quand n est égal à 10 car le temps de test exhaustif dépasse 30 mille ans. Il faut donc décider à un certain moment d'arrêter les tests en espérant avoir obtenu une confiance satisfaisante pour le logiciel sous test.

Tableau 2. Le temps de test avec la vitesse 1000 cas/seconde.

n	Nombre de cas possibles	Temps de test
1	1	10^{-3}s
5	244 140 625	2js 19h 49’ 0,625’’
10	10^{12}	≈ 31 700 années

1.c. Procédure générale

La Figure 2 résume le schéma global des étapes de test. Il s’agit d’une décomposition logique du test en grandes étapes (par exemple, le diagnostic et la correction peuvent avoir lieu après l’exécution d’un ensemble de cas de test).

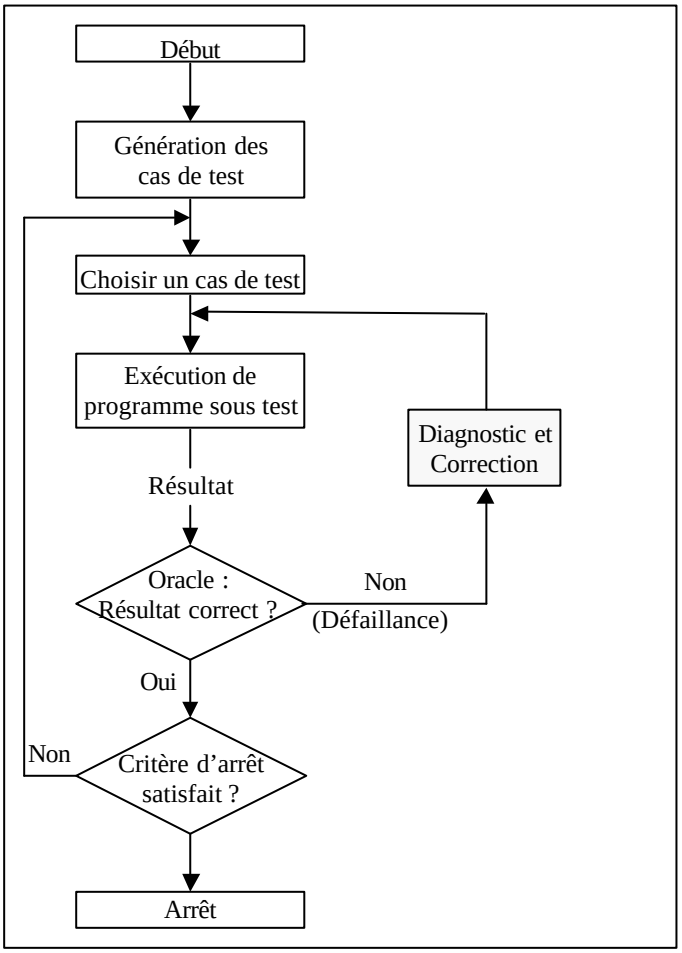

Figure 2. L’activité de test et de déboguage.

- *Génération de cas de tests* : cette étape consiste à trouver des données d'entrée, des scénarios pour exécuter les tests ainsi que les résultats attendus.
- *Exécution* : un programme de test est exécuté avec les données de test. Ce programme peut être le logiciel qu'on veut tester dans le cas du test système ou du test d'acceptation. Pour le test unitaire et le test d'intégration, ce programme de test est un pilote de test. Ce pilote doit mettre en exécution la partie de logiciel qu'on veut tester.
- *Oracle* : Un oracle est une fonction qui doit permettre de distinguer si une sortie du logiciel est correcte :
  - Si l'oracle a détecté une défaillance, il faut la corriger et ré-exécuter le même cas de test pour vérifier que la correction a effectivement éliminé la faute.
  - Si l'oracle n'a pas détecté de défaillance, on a deux directions : soit le critère d'arrêt est satisfait et la phase de test du logiciel est terminée ; soit un nouveau cas de test peut être exécuté.
- *Critères d'arrêt* : On peut rarement effectuer un test exhaustif. Dans la plupart des cas, on doit se contenter d'un test moins coûteux et il faut donc avoir un critère pour décider quand le test est satisfaisant.

En résumé, toute technique de test doit répondre à trois problèmes principaux, et peut être caractérisée en fonction de ceux-ci :

i. Problème de la génération des cas de test. Comment peut-on s'assurer que les cas de test couvrent le maximum d'erreurs avec un coût raisonnable ?

ii. Problème de l'oracle. Comment peut-on savoir si l'ensemble d'oracles est suffisamment efficace mais pas redondant ?

iii. Problème du critère d'arrêt. Comment peut-on garantir que les tests sont suffisants et que les erreurs restantes (éventuellement) ne sont pas trop « *graves* » ?

2. Les techniques de test

Il existe deux grandes catégories de tests : le test fonctionnel (aussi appelé test « *boîte noire* » ou test « *transparent* » – « *functional testing* », « *black box testing* » or « *transparent testing* ») et le test structurel (ou test «*boîte blanche* » – « *white box testing* ») [Beizer 1990 et Xanthakis 2000].

Le test fonctionnel est uniquement basé sur la connaissance de la plus formelle spécification possible du programme et des services attendus. Cette technique est intéressante car on peut développer les pilotes de test en parallèle avec l'implémentation. Si l'implémentation change et si ce changement n'implique pas le changement de la spécification, on peut réutiliser les mêmes cas de test.

Le test structurel est basé sur la connaissance de la structure interne du logiciel. Cette technique permet d'obtenir un modèle structurel du logiciel (graphe de contrôle, flot de données…). Par conséquent, elle permet de modéliser une structure qui couvre le système sous test selon certains critères, par exemple toutes les instructions, toutes

les branches… Cette modélisation facilite la conception de cas de test par la connaissance du but à atteindre et des moyens pour atteindre ce but. De plus, cette technique renforce la validité des tests puisqu'on sait quelle partie du code a été testée.

La Figure 3 montre un exemple de test structurel, basé sur un diagramme de contrôle. Cette figure nous montre qu'à partir d'un diagramme de contrôle, on peut écrire des tests en fonction du critère de couverture choisi. Pour couvrir toutes les branches, il faudra deux tests, alors que pour couvrir tous les chemins, il en faudra une troisième.

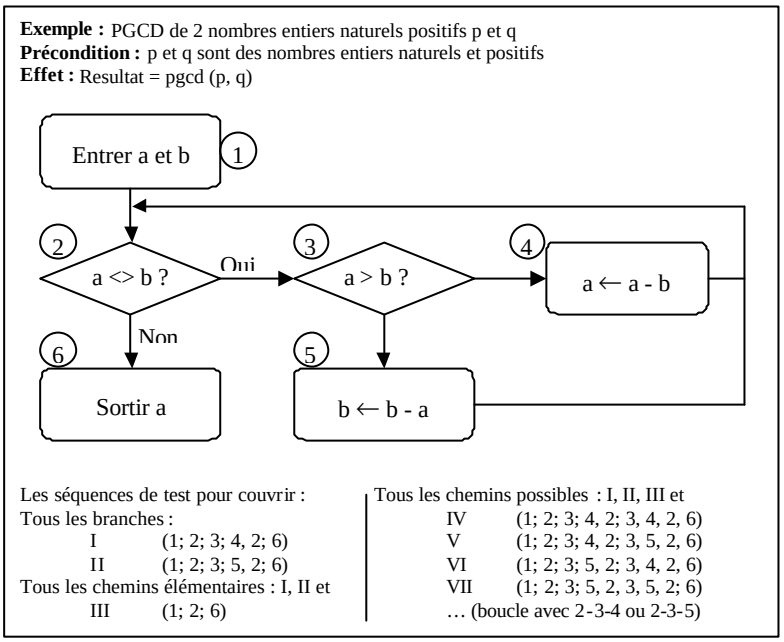

Figure 3. Un exemple pour le test structurel.

On peut utiliser une autre technique de test, le test « *boite grise* » (« *gray box testing* »). C'est une combinaison des deux techniques précédentes. D'une part, il est réutilisable et cohérent avec la conception, d'autre part, il augmente la couverture des erreurs.

3. Les difficultés du test des systèmes à objets

La technique de programmation à objets pose plusieurs problèmes que la technique de programmation procédurale ne pose pas. Ces problèmes viennent de la modularité, de l'encapsulation, de l'abstraction et du polymorphisme, conséquence de l'héritage et de la liaison dynamique.

3.a. La modularité

À cause de la modularité, une « *unité* » de logiciel à objets, qui est souvent une classe, n'est pas un sous-programme qui peut fonctionner tout seul mais il doit coopérer avec d'autres unités. Par conséquent, la plupart des tests unitaires impliquent l'utilisation d'autres unités de programme - autrement dit, un peu comme dans le test

d'intégration classique. Ainsi, le test d'intégration pour un système à objets englobe non seulement la charge du test d'intégration mais aussi celle du test unitaire. La Figure 4, montre la forte connectivité d'une partie de bibliothèque du langage Eiffel à travers son diagramme de classes UML.

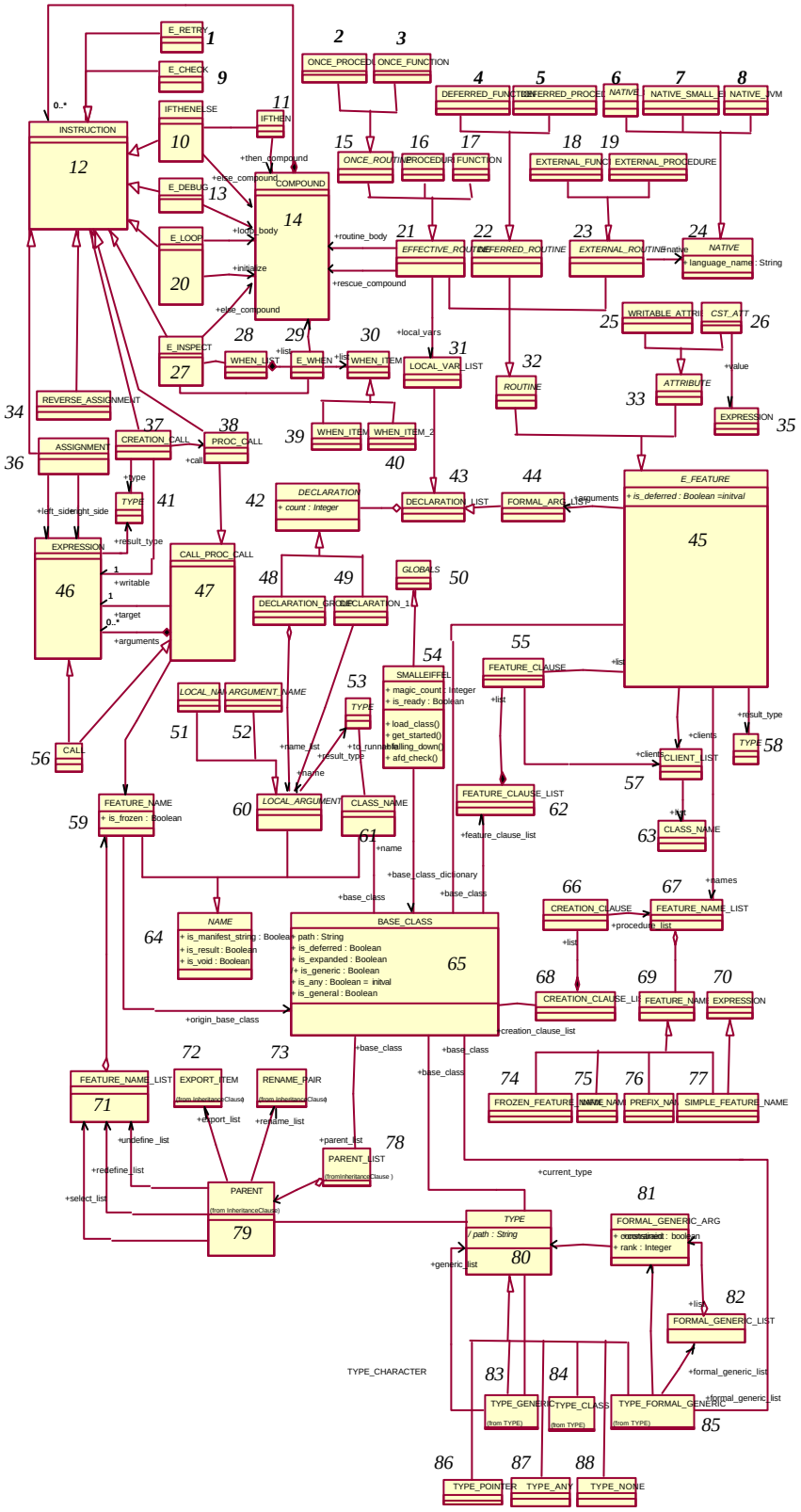

Figure 4. Une partie de bibliothèque du langage Eiffel.

3.b. L'héritage et le problème Yo-yo

L'héritage est un mécanisme de base de la programmation à objets. Il permet de réutiliser le résultat des travaux précédents et d'enrichir la fonctionnalité des classes. Les classes dérivées peuvent ajouter des attributs et des fonctions ou changer (redéfinir) le comportement des méthodes existantes. Normalement, une fonction est définie et peut-être optimisée pour un travail concret. Si l'on redéfinit le comportement de cette fonction pour un nouveau travail, rien n'assure que les deux versions fonctionnent encore bien pour le premier cas.

Un autre problème vient de l'héritage multiple et sa conséquence, l'héritage répété (voir la Figure 5). Quand la classe dérivée hérite de plusieurs classes, les classes originaires peuvent contenir les fonctions de même nom. Cela nécessite un mécanisme de contrôle afin de s'assurer que la fonction exécutée est bien celle qui a été choisie. Du point de vue du test, la « *contrôlabilité* » et la «*diagnosabilité* » du code et des fonctions testées deviennent ardues.

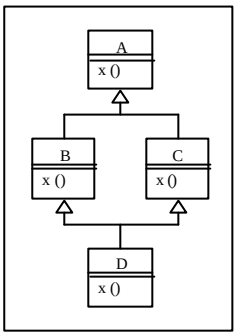

Figure 5. L'héritage multiple et l'héritage répété.

Une autre conséquence de l'héritage est le problème « *Yo-yo* ». Ce problème apparaît quand l'exécution monte et descend plusieurs fois dans une chaîne d'héritage comme le montre la Figure 6 (adaptée de [Binder 1999]). Le comportement des méthodes correspondantes est décrit dans le Tableau 3. Si le cas de test consiste en l'appel de la méthode A de la classe C5, on s'aperçoit que le code effectivement exécuté se répartit de manière complexe parmi les différentes méthodes des 5 classes de cette hiérarchie d'héritage. Ce problème rend difficile la localisation des erreurs quand les tests les révèlent.

3.c. L'encapsulation et le contrôle d'accès

Chaque classe est un ensemble d'attributs et d'opérations. Les attributs décrivent l'état d'un objet. Pour tester une classe, on doit faire changer ses attributs et voir les réactions de l'objet. Du fait du contrôle d'accès, on ne peut pas accéder directement à certains attributs et donc, on ne peut pas vérifier leurs changements. Pour passer ce contrôle, il y a des solutions comme des fonctions « *friend* » dans le langage C++. Une autre solution est la classe «*autotestable* » où on introduit non seulement le code des fonctions mais aussi le code de vérification (les tests) dans la classe (voir [Baudry 2000 #1, Baudry 2000 #2, Deveaux 2000 et LeTraon 1999]

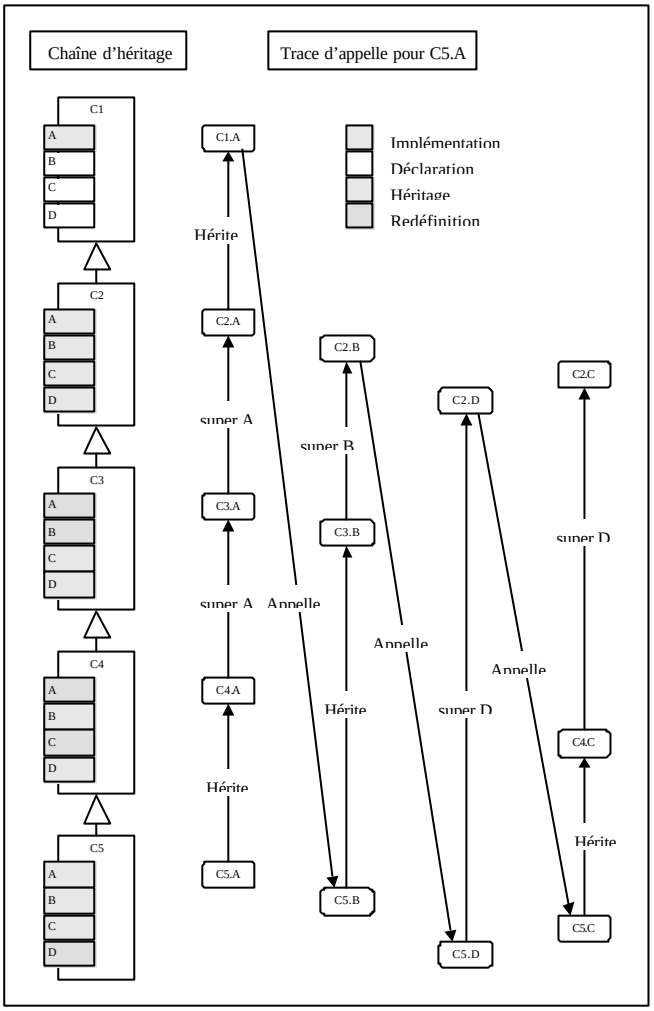

Figure 6. Le problème Yo-yo.

Tableau 3. Le comportement des méthodes présentées dans la Figure 6.

Classe	Méthode A	Méthode B	Méthode C	Méthode D
C1	Implémentation Appelle B et C	Déclaration	Déclaration	Déclaration
C2		Implémentation Appelle D	Implémentation	Implémentation. Appelle C
C3	Redéfinition. Appelle super.A	Redéfinition Appelle super.B		
C4	Redéfinition Appelle super.A		Redéfinition. Appelle super.C	
C5				Redéfinition. Appelle super.D

3.d. Le polymorphisme et l'abstraction

Le polymorphisme permet à un nom d'objet de pouvoir désigner des instances de classes différentes, issues d'une même arborescence [Beizer 1990, Binder 1999 et Xanthakis 2000]. Le polymorphisme implique un mécanisme de contrôle de l'héritage [Ponder 1994].

Une façon d'obtenir le polymorphisme est d'utiliser les abstractions. Dans les classes abstraites, on met des interfaces des opérations et on laisse les définitions de ces opérations aux classes qui implémentent ces interfaces. On ne peut pas tester une classe abstraite seule car on ne peut pas créer d'objet de cette classe. Pour la tester, il faut utiliser au moins l'une des classes qui l'implémentent. Du point de vue du test, cela signifie qu'une classe interface ou une classe abstraite ne peut être testée qu'à travers ses implémentations.

4. Le test d'intégration pour les systèmes orientés objet

Théoriquement, le test d'intégration est une étape intermédiaire entre le test unitaire et le test système. Il s'agit moins de tester la validité des fonctions principales du système et les propriétés non-fonctionnelles spécifiées dans le cahier des charges (la charge du test unitaire) que d'assembler les unités (classes ou méthodes) de manière fiable. Dans le cas d'un système à objets, comme nous l'avons vu dans le paragraphe 3.a, le test unitaire est très souvent un test d'intégration donc ce dernier prend aussi les caractéristiques du test unitaire, c'est-à-dire, vérifier la fiabilité des unités. Le test système ne peut être effectué que lorsque le test d'intégration a été effectué de manière satisfaisante et que toutes les anomalies détectées ont été corrigées.

Le test d'intégration ne se concentre pas sur les cas de test qui sont générés avant le test d'intégration. Le test d'intégration s'attache à la manière de mettre ensemble les unités et ensuite, d'exécuter l'ensemble.

Dans la phase de test d'intégration, les principales anomalies devraient porter théoriquement sur les interfaces des unités. Le test d'intégration doit détecter les erreurs portant sur la compréhension des fonctions qui devaient être réalisées et donc sur l'incohérence qui apparaît lors des tests d'intégration. Ces erreurs sont les « *fautes sémantiques* » portant sur une ambiguïté des dossiers de conception ou une incompréhension de ceux-ci (voir « *Design by Contract: The lessons of Ariane* » – [Jézéquel 1997]).

Comme on l'a présenté dans le paragraphe 3.a, le test unitaire pour un système à objets est souvent un test d'intégration. Le test d'intégration doit donc révéler aussi les erreurs que le test unitaire doit détecter. Ce sont des bogues existantes dans le corps des unités.

A l'issue du test d'intégration, on est sensé disposer d'un système s'exécutant correctement et dont « *toutes* » les fautes que les tests peuvent trouver, portant sur la cohérence de la structure, ont été révélées et corrigées. Le test d'intégration suppose la connaissance d'une structure représentant la manière dont sont dépendantes les unités du système, ce que l'on nomme l'architecture. L'exception est le cas de l'intégration « *Big-Bang* » qui consiste à tout tester ensemble d'un seul coup, sans aucune étape intermédiaire et qui est le contraire d'une intégration. Il s'agit donc de test structurel,

basé sur l'architecture du système, qui dans notre cas sera décrite en UML (diagrammes de classes).

4.a. Dépendances d'intégration

Dans les systèmes à objets, la plupart des unités utilisent les services d'autres unités. On appelle «*clients* » les unités qui demandent des services et les «*serveurs* » (ou « *fournisseur* ») les unités qui fournissent ces services. Le comportement des clients dépend des services que les serveurs fournissent. Le test d'intégration des clients dépend donc des serveurs associés. Pour intégrer un client, les services que ce client requiert doivent être accessibles, c'est-à-dire, ces services doivent exister ensemble quand on teste le client. Cette demande est appelée « *contrainte d'intégration* ». La dépendance entre client et serveur pour le test d'intégration est appelée « *dépendance d'intégration* ».

4.b. Un plan de test d'intégration – un ordonnancement

La première idée est d'utiliser le test d'intégration « *Big-Bang* » qui consiste à mettre ensemble toutes les unités et à exécuter l'ensemble. Cette technique ne marche plus quand le système est complexe, car il est alors trop difficile de localiser les erreurs. L'idée d'utiliser une intégration progressive, c'est-à-dire d'intégrer une ou quelques unités à la fois, rend le test d'intégration plus efficace pour localiser les erreurs.

Théoriquement, l'existence des services (voir le paragraphe 4.a) implique l'existence des serveurs correspondants, c'est-à-dire qu'un client ne peut pas être intégré sans ses serveurs. En pratique, on pourra utiliser un simulateur (un « *bouchon* » – « *stub* ») de serveur pour le test d'intégration (voir paragraphe 4.c) chaque fois que le serveur définitif n'est pas disponible au moment de l'intégration.

La question du test d'intégration est donc une question de planification des étapes de test. Il s'agit de définir dans quel ordre les unités doivent être intégrées. La manière d'intégrer s'appelle une « *stratégie* » d'intégration. Un exemple de stratégie d'intégration est donné dans la Figure 7, la stratégie « *de bas en haut* » (« *Bottom up* »), qui part des feuilles de l'architecture. Les pilotes de test possèdent les données avec lesquelles l'unité sous test doit être exécutée en utilisant la partie du système déjà intégrée.

4.c. Premier problème – Décomposition des interdépendances

L'exemple de la Figure 7 présente le cas «*idéal* » : l'architecture est «*acyclique* » et l'intégration se fait de bas en haut. Cependant, les serveurs ne peuvent pas toujours être intégrés avant les clients. Du fait de la forte connectivité des unités des systèmes à objets, il y a des cas où les serveurs demandent, directement ou indirectement via un circuit de dépendances, les services de leurs propres clients. Dans ces cas, on a des « *interdépendances* » ; les serveurs et les clients participent à des « *groupes interdépendants* ».

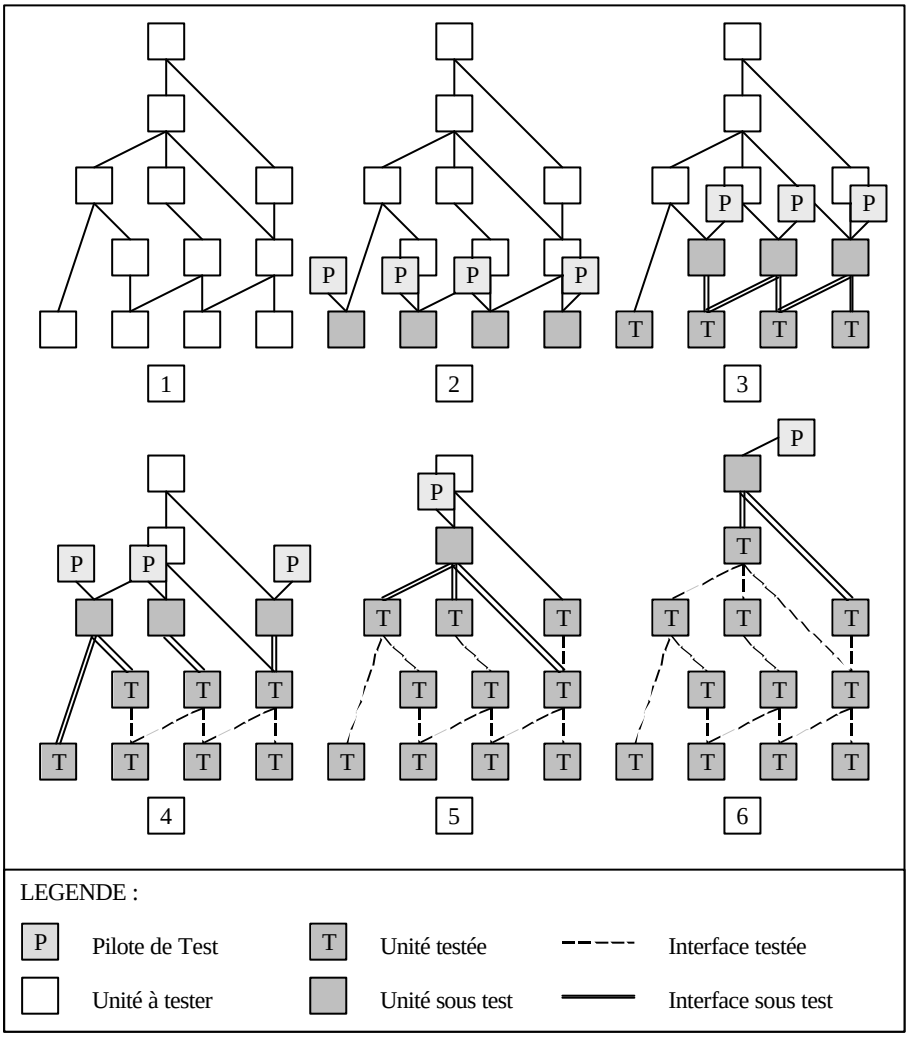

Figure 7. La stratégie « *de bas en haut* ».

Un exemple simple est présenté dans la Figure 8 qui montre une interdépendance. Cet exemple est extrait de la bibliothèque du langage Eiffel. Dans cet exemple, les unités demandent directement des services l'une à l'autre, mutuellement. En effet, la connectivité forte des systèmes à objets peut grouper une vingtaine d'unités dans un groupe interdépendant (voir l'exemple InterViews dans [Kung 1995 #2]) : ces interdépendances ne sont pas décelables facilement et un modèle est nécessaire pour les détecter automatiquement.

```
expanded class INTEGER

inherit INTEGER_REF

...
```

```
class INTEGER_REF

...

feature

  item: INTEGER;
```

Figure 8. Deux unités interdépendantes.

Dans ce cas, fréquent dans le domaine objet à l'intérieur d'un « *package* », il y a deux solutions :

i. Soit, on remplace les services d'une unité, nécessaires à l'exécution des cas de test, par les services fournis par un simulateur. Ce remplacement se fait successivement jusqu'au moment où les interdépendances sont totalement décomposées. Les simulateurs ne devront pas utiliser de service des unités du même groupe interdépendant. Les unités qui ne sont pas simulées seront intégrées avant les unités simulées. Le simulateur est souvent appelé un « *bouchon* » de test (« *stub* »).

ii. Soit, on intègre toutes les unités d'un groupe interdépendant en une seule étape (ce qui nous ramène au problème de l'intégration Big-Bang) ou plutôt on utilise une approche intermédiaire où l'intégration se fait groupe par groupe. On intègre les interdépendances qui ne sont pas trop complexes pour la localisation des erreurs et on utilise des bouchons pour décomposer les interdépendances plus complexes. On présentera cette solution dans le paragraphe 3, du Chapitre IV.

La création des bouchons est coûteuse et l'utilisation de bouchons est risquée. On ne peut pas simuler entièrement le comportement d'une unité parce que si l'on simule tous ses services, on risque de créer une nouvelle unité toute aussi complexe. En général, un bouchon est une version simplifiée de l'unité originelle qui ne doit utiliser aucun service d'autres unités du même groupe interdépendant. Notons qu'un bouchon doit fournir « *correctement* » au moins tous les services correspondants aux données de test prédéterminées.

Un bouchon « *réaliste* » est un simulateur complet de l'unité (voir la Figure 9). Il peut s'agir d'un composant ancien (moins performant mais fonctionnellement équivalent) que l'on réutilise uniquement à cette fonction.

Un bouchon est « *spécifique* » s'il renvoie des réponses prédéterminées aux appels d'un seul client (voir la Figure 9). C'est le cas le plus courant dans la pratique industrielle puisqu'on ne dispose que rarement d'un simulateur complet des services d'une unité.

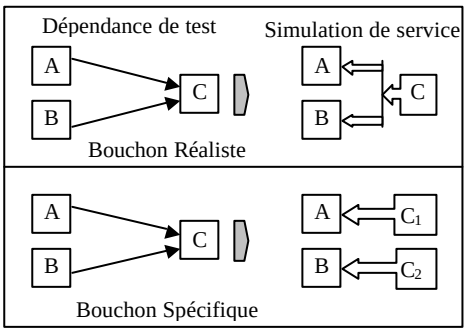

Figure 9. Un bouchon réaliste | Deux bouchons spécifiques.

Il est évident que la création d'un bouchon spécifique par un serveur est moins coûteuse et moins risquée que la création de son bouchon réaliste. Cependant, la création d'un bouchon réaliste d'un serveur peut être moins coûteuse que la création

de plusieurs bouchons spécifiques du même serveur car les services simulés peuvent partager des codes.

Un bouchon ne sert qu'à l'intégration des unités et n'est pas utilisé dans la version finale du logiciel. En plus, étant donné le coût de création des bouchons, l'ordre d'intégration doit si possible minimiser ce coût en minimisant d'abord leur nombre. Dans l'exemple de la Figure 7, aucun bouchon n'est nécessaire car il n'y a pas d'interdépendance dans l'architecture.

Dans le cas de partage des travaux de développement, on peut être aussi amené à utiliser un bouchon même dans le cas idéal qui n'a pas d'interdépendance pour peu que le code d'un serveur ne soit pas encore développé.

La question est maintenant : comment choisir l'unité à simuler ? Pour les groupes interdépendants simples et très liés comme dans la Figure 8, on peut intégrer toutes les unités en un seul coup. Pour un groupe interdépendant plus complexe, la difficulté de l'intégration Big-Bang se pose de nouveau si on intègre en même temps toutes les unités : il faut éviter une telle situation. Dans ce cas, il faut forcément créer des bouchons. Une autre question apparaît alors : comment peut-on déterminer si un groupe interdépendant est complexe ?

La façon de déterminer les unités qui doivent être simulées constitue l'un des deux problèmes majeurs du test d'intégration. De fait, la plupart des recherches sur le test d'intégration se contentent de chercher des réponses à ces deux questions : déterminer un groupe interdépendant complexe et choisir un bouchon. Une contribution de cette thèse est de proposer une autre solution qui améliore, en terme de nombre de bouchons demandés, par rapport aux solutions existantes, la décomposition des groupes interdépendants.

Un bouchon n'étant qu'un simulateur qui imite le comportement de son unité originelle, rien n'assure que les comportements du bouchon et de l'unité originelle soient identiques. Par conséquent, après avoir intégré une unité en utilisant un bouchon, on doit tester cette unité dans son environnement final avec l'unité originelle. Cette étape est appelée une étape de « *retest* ».

Par exemple, si on doit décomposer une interdépendance comme celle de la Figure 10 en utilisant le bouchon qui est le simulateur de l'unité 1. L'unité 3 est testée deux fois : une première fois en utilisant le bouchon de l'unité 1 et une deuxième fois en utilisant la vraie unité 1.

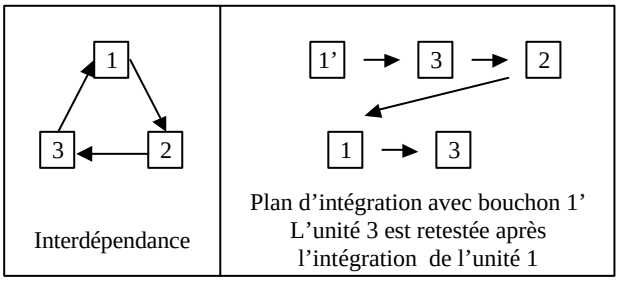

Figure 10. Le choix un bouchon : une tâche d'ordonnancement.

4.d. Deuxième problème – Parallélisation du test d'intégration

Dans la plupart des cas, le test est une tâche partagée. Il est effectué par un groupe de « *testeurs* », personnes affectées à l'équipe de validation. Comment peut-on distribuer le test aux testeurs pour que le temps du test d'intégration soit minimal ? Avec deux testeurs, pour les unités dont les demandes de services sont représentées dans la Figure 11 par des flèches, en supposant que le test de chaque unité demande une unité de temps, quel est le meilleur ordre d'intégration ? L'ordre (6-7, 4-5, 3, 2, 1) requiert 5 unités de temps ; une meilleure solution, (4-6, 3-7, 2-5, 1) demande seulement 4 unités de temps. Comment peut-on obtenir la solution efficace dans un cas général ?

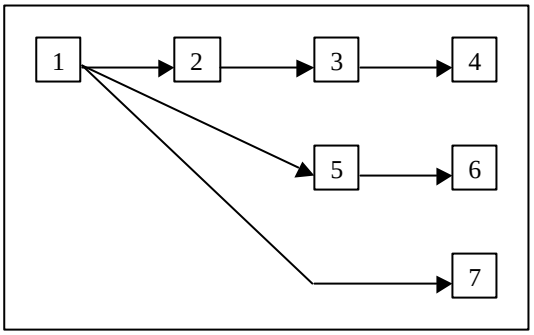

Figure 11. Comment parallélise-t-on l'intégration ?

La minimisation de la durée d'intégration est le deuxième problème posé par le test d'intégration. Notons que la minimisation du temps d'intégration n'est pas pris en compte si un seul testeur effectue l'intégration.

Il n'y a aucune solution existante qui tente de paralléliser le test d'intégration. Une autre contribution de cette thèse est d'avoir proposé un moyen de déterminer une solution efficace pour ce problème en se basant sur des algorithmes classiques.

4.e. Modélisation de la structure de dépendances

Pour mettre à profit les résultats de la théorie des graphes, il faut modéliser la structure de dépendances du système à l'aide d'un graphe comme dans la Figure 7. Les problèmes d'ordonnancement et de parallélisation sont alors bien posés et peuvent être résolus de manière efficace.

Dans le graphe modélisant la structure de dépendances, un nœud représente une unité sous test et un arc (orienté) modélise une dépendance : un arc d'un nœud A vers un nœud B signifie que l'unité A est cliente d'un ou plusieurs service(s) de l'unité B. Une unité peut être un groupe de classes, une classe ou une méthode dans une classe. Plus la granularité d'une unité est fine, plus la stratégie obtenue est précise. Il est évident que si les unités sont des méthodes, on connaît exactement l'ordre dans lequel seront testées les méthodes. Cependant, si les unités sont seulement des classes, on ne sait pas dans une classe, dans quel ordre ses méthodes seront testées.

La structure de dépendances est bien présentée par un diagramme de classes UML. Comme le modèle UML est très répandu, nous proposons un ensemble de règles pour

modéliser un diagramme de classes UML par un graphe. C'est la troisième contribution de cette thèse.

4.f. Résumé

Le problème du test d'intégration est d'ordonner les unités. Ce problème se divise en deux sous problèmes :

i. Minimisation du coût de création et d'utilisation des bouchons en réduisant d'abord leur nombre et si possible leur complexité.

ii. Minimisation du temps d'intégration en fonction du nombre de testeurs pouvant travailler en parallèle à l'intégration du système.

La question du test d'intégration revient à trouver la meilleure stratégie qui donne l'ordre d'intégration des unités selon ces deux problèmes. D'ores et déjà, il faut préciser (du fait de l'existence des interdépendances) que la minimisation du coût de simulation est un problème de type NP-complet [Gondran 1985], et que dans le pire des cas, la complexité de l'algorithme qui permet d'atteindre la solution optimale est factorielle par rapport au nombre d'unités du système sous test. Ainsi, un système de 40 classes demanderait à essayer les 40! ($\approx 8{,}2.10^{47}$) ordres d'intégration possibles afin de trouver l'ordre optimal. Ceci revient à dire qu'il faut déterminer une heuristique - une stratégie d'intégration – la plus efficace possible. C'est sur ce point que nous avons focalisé notre recherche, en utilisant l'expérience des recherches précédentes dans ce domaine.

Toutes les stratégies d'intégration existantes sont basées sur un modèle de graphe car ce modèle représente très bien les problèmes d'ordonnancement comme celui des stratégies d'intégration. Par conséquent, une stratégie de test d'intégration contient trois grandes étapes (voir la Figure 12) :

i. *Modélisation* : Construire un graphe qui représente les dépendances d'intégration du système sous test (à partir d'UML par exemple).

ii. *Décomposition des interdépendances* : Décomposer les interdépendances qui sont trop difficiles à intégrer et à tester entièrement, en interdépendances dont la complexité est maîtrisable. C'est dans cette étape qu'on résout le premier problème principal du test d'intégration (minimisation du nombre et de la complexité des bouchons).

iii. *Planification* : Trouver une organisation du test d'intégration pour minimiser le temps du test. Celle-ci devra prendre en compte le nombre de testeurs effectuant le test d'intégration.

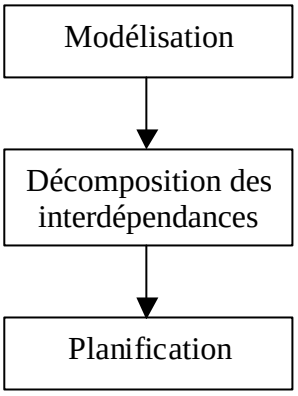

Figure 12. Trois grandes étapes de la planification du test d'intégration.

5. Notre contribution

Comme indiqué à la fin des paragraphes 4.c, 4.d et 4.e, notre contribution porte sur les trois étapes d'une stratégie de test d'intégration : modélisation, décomposition des interdépendances et planification.

Pour modéliser la structure de dépendances d'intégration, nous proposons un ensemble de règles de transformation à partir d'un diagramme de classes UML. Notre modélisation a trois avantages. Premièrement, en profitant de l'utilisation répandue et de la standardisation de UML, elle peut être utilisée pour la plupart des conceptions de projets logiciels. Deuxièmement, elle permet de capturer toutes les informations utiles pour la planification. Troisièmement, elle permet d'obtenir ces informations dès la phase de conception dans le cycle de vie du logiciel, ce qui rend l'approche prédictive.

Dans le graphe qui modélise la structure de dépendances d'intégration, les interdépendances sont identifiées par ses «*composantes fortement connexes* » (CFC). Ce travail qui se situe dans le prolongement de travaux antérieurs de l'équipe ([Jéron 1999 et LeTraon 2000]), vise à décomposer les interdépendances en décomposant ces CFCs. Pour décomposer des CFCs, nous proposons une heuristique qui comporte plusieurs critères. En se basant sur les cycles « *élémentaires* » - un cycle dont les nœuds apparaissent une seul fois, notre heuristique permet de diminuer le nombre de bouchons ; elle permet aussi de diminuer le nombre d'étapes de retest.

Nous proposons enfin une heuristique intermédiaire entre une stratégie incrémentale stricte et la stratégie Big-Bang. D'une part, notre heuristique permet de faciliter la localisation d'erreurs comme la stratégie progressive. D'autre part, elle diminue le coût de création des bouchons comme la stratégie Big-Bang. Cette heuristique cherche à éviter d'introduire des bouchons dans des ensembles de classes très cohérents d'un point de vue objet (design patterns par exemple).

Quant à la planification, nous proposons une nouvelle solution qui assure aussi la parallélisation du test d'intégration. En se basant sur la notion du chemin critique, notre parallélisation produit une solution simple. La partie droite de la Figure 13 résume notre travail.

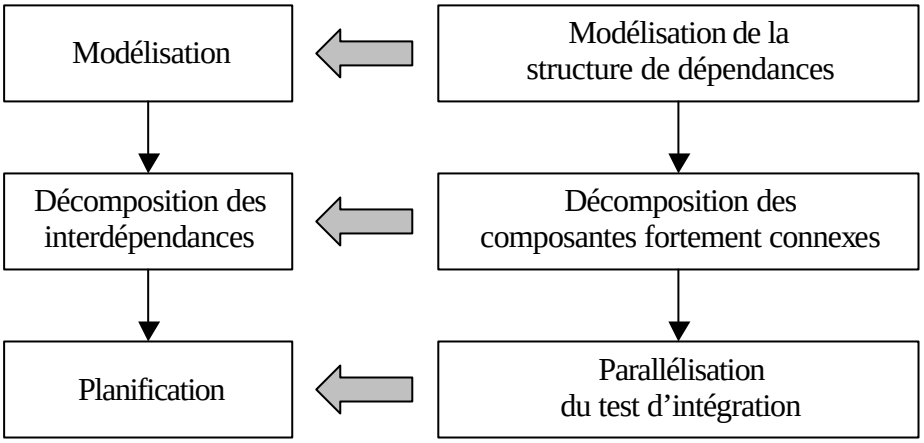

Figure 13. La stratégie « Triskell » pour planifier le test d'intégration.

Pour illustrer les avantages de notre solution, nous avons expérimenté et comparé notre approche sur six études de cas qui sont des systèmes et des bibliothèques de logiciels. Le nombre de classes et de dépendances de ces études de cas varie entre de la dizaine à des centaines. Ce sont le commutateur Télécom SMDS (« Switched Multimegabits Data Service ») avec 37 classes et 72 dépendances ; la bibliothèque Pylon du langage Eiffel avec 50 classes et 133 dépendances ; une partie du compilateur Eiffel avec 104 classes et 140 dépendances ; la bibliothèque InterView, présentée par Kung dans [Kung 1995 #2] avec 146 classes et 419 dépendances ; une partie de Java J2SE v1.31 avec 588 classes et 1935 dépendances et la bibliothèque Swing de J2EE v1.31 avec 694 classes et 3819 dépendances. Les études sur ces études de cas sont faites par notre prototype TestPlan.

Enfin, outre le prototype TestPlan, notre approche a été implantée comme un module de l'outil commercial de modélisation Objecteering de la société Softeam.

6. Plan de la thèse

Nous présentons les travaux d'autres groupes de recherches dans le Chapitre II, « État de l'art ». Ces travaux ont été publiés dans la dernière décennie. Nous choisissons de nous focaliser sur les travaux de l'équipe de David C. Kung [Kung 1995 #1, Kung 1995 #2 et Kung 1996], de Kuo Chung Tai et Fonda J. Daniels [Tai-Daniels 1999], de l'équipe de Lionel Briand [Briand 2001], de Yvan Labiche [Labiche 2000 #1 et Labiche 2000 #2]. et finalement le travail préliminaire de l'équipe [Jéron 1999 et LeTraon 2000]

Dans le Chapitre III, « Modélisation de la structure de dépendances », nous présentons un ensemble de règles pour modéliser l'architecture d'un système sous test. Le diagramme de classes d'un modèle UML est utilisé comme base de modélisation.

Le Chapitre IV, « Décomposition des composantes fortement connexes » est consacré à la décomposition des groupes d'unités interdépendantes – le problème le plus difficile du test d'intégration. Nous séparons les critères pour minimiser le coût de création des bouchons et le risque d'utilisation des bouchons.

Le Chapitre V, « Parallélisation du test d'intégration », présente nos travaux de parallélisation du test d'intégration en se basant sur le traitement de la complexité d'une unité et le chemin critique.

Dans le Chapitre VI, « Expérimentations et étude comparative des différentes stratégies », nous présentons notre expérimentation sur six études de cas de complexité différente. Nous comparons les résultats obtenus par notre solution avec ceux d'autres groupes de recherches. Ces résultats sont obtenus grâce à notre outil TestPlan dont la conception et le mode d'emploi sont présentés dans l'annexe.

Nous terminons notre mémoire par un résumé au Chapitre VII, « Conclusion et perspective ». Nous récapitulons les améliorations de notre solution par rapport aux autres stratégies existantes. Nous abordons aussi dans ce chapitre les travaux ultérieurs qu'on doit effectuer pour améliorer notre approche.

Les premiers pas de notre travail ont été présentés dans la conférence ECOOP 2001 à Budapest, Hongrie [VuLe 2001]. Cette présentation ne contient que la partie qui concerne le test progressif strict et pas la version mixte Big-Bang/incrémental. Les résultats comparatifs présentés dans cet article, montrent que notre stratégie a apporté une amélioration significative à l'état de l'art.

Chapitre II.

État de l'art

Nous décrivons dans ce chapitre les travaux publiés sur le test d'intégration. Nous commençons, dans le paragraphe 1, par les notions de base du test d'intégration qui sont nécessaires pour comprendre l'apport des algorithmes introduits dans ce mémoire. Les principales définitions (graphe, composante fortement connexe, cycle…) serviront dans le Chapitre IV, « Décomposition des composantes fortement connexes ».

Il existe de nombreux travaux dans ce domaine : [Briand 2001, Chung 1997, Harrold 1992, Jéron 1999, Jorgensen 1994, Kung 1995 #1, Kung 1995 #2, Kung 1996, Labiche 2000 #1 et Labiche 2000 #2, LeTraon 2000, McGregor 1994 #3, Mercier 1998, Müllerburg 1990, Tai-Daniels 1999, Winter 1998].

La recherche de l'équipe de David C. Kung [Kung 1995 #1, Kung 1995 #2 et Kung 1996] sur le test d'intégration est présentée dans le paragraphe 2. C'est le premier travail qui comporte toutes les trois étapes d'une stratégie de test d'intégration (modélisation, décomposition des interdépendances et planification). La solution de Kung est très simple. Cependant, cette solution laisse plusieurs problèmes à résoudre pour trouver une stratégie efficace.

Ensuite, dans le paragraphe 3, nous abordons le travail de Kuo Chung Tai et Fonda J. Daniels [Tai-Daniels 1999]. La recherche de Tai-Daniels se concentre sur la décomposition des interdépendances. Quant aux modélisation et planification, Tai-Daniels reprennent les résultats de Kung.

Un autre travail qui sera également présenté, à la suite des travaux de Tai-Daniels est celui de l'équipe de Lionel Briand [Briand 2001]. Cette recherche est présentée dans le paragraphe 4. Comme le travail de Tai-Daniels, celui de Briand porte sur la décomposition des interdépendances. Pour la modélisation, il utilise celle de Kung. Quant à la planification, il utilise aussi en partie celle de Kung.

Puis, dans le paragraphe 5, nous poursuivons avec le travail de l'équipe de Yvan Labiche [Labiche 2000 #1 et Labiche 2000 #2]. Leur travail essaie de compléter les travaux de Kung par un traitement du polymorphisme et de l'abstraction mais ne traite pas le problème de l'interdépendance.

Le travail préliminaire de l'ancienne équipe Pampa [Jéron 1999, LeTraon 2000] est présenté dans le paragraphe 6. Ce travail utilise l'algorithme de Tarjan pour

déterminer les interdépendances et le poids des unités à simuler. Il donne aussi la première parallélisation des tests.

Par ailleurs, on peut trouver les informations générales sur le test d'intégration dans l'ouvrage de Robert V. Binder [Binder 1999] : un chapitre entier est dédié au test d'intégration.

Nous allons faire quelques remarques après avoir présenté plus en détail les travaux listés ci-dessus. Ces remarques créent la base de nos propositions afin d'obtenir une stratégie plus complète et plus efficace pour le test d'intégration.

1. Unité, intégration et dépendances d'intégration

Pour présenter les recherches sur la stratégie de test d'intégration, nous rappelons dans cette partie les notions de base du test d'intégration.

Un logiciel à objets est composé de nombreuses unités.

Définition 1. *Une unité U d'un programme P est une partie testable du programme. Elle peut être une méthode dans une classe, une classe ou un groupe de classes.*

Définition 2. *Le test d'intégration est une tâche qui essaie de faire coopérer toutes les unités d'un système dans un environnement commun (environnement de test) et de localiser les erreurs si les unités ne peuvent pas coopérer correctement.*

Dans les systèmes à objets, la plupart des unités utilisent les services d'autres unités. Cette utilisation signifie qu'une unité utilise le résultat de traitement d'une autre unité pour fonctionner. Rober V. Binder [Binder 1999] présente plusieurs types d'utilisation de services entre classes.

Définition 3. *Une classe C_1 utilise un service d'une autre classe C_2 si et seulement si :*

- *La classe C_1 est dérivée de la classe C_2.*
- *La classe C_1 utilise un objet de la classe C_2 comme un attribut (la composition ou l'agrégation).*
- *La classe C_1 utilise un objet de la classe utilitaire C_2.*
- *La classe C_1 générique utilise la classe C_2 comme type concret pour sa déclaration générique.*
- *Une méthode de la classe C_1 appelle une autre méthode de la classe C_2.*
- *Une méthode de la classe C_1 utilise un objet ou un pointeur sur un objet de type C_2 comme un paramètre ou comme une variable locale.*

En général, quand une unité U_1 utilise un service d'une unité U_2, si on veut intégrer l'unité U_1, le service que l'unité U_2 fournit doit être disponible. C'est la demande d'existence de service pour l'intégration. On peut dire que l'intégration de l'unité U_2

dépend du service de l'unité U_2. En bref, pour l'intégration, l'unité U_1 dépend de l'unité U_2.

Définition 4. *Il existe une dépendance d'intégration d'une unité U_1 vis à vis d'une unité U_2 , noté « U_1 ? U_2 » si et seulement si :*

- *soit l'unitéU_1 utilise un service de l'unité U_2, noté « $U_1 > U_2$ »*

  et alors U_1 est appelée le client de cette dépendance

  et U_2 est appelée le serveur de cette dépendance.

- *soit* $\exists U_3 \in P \mid U_1 > U_3 \wedge U_3 \; ? \; U_2$

Notons que par définition, la dépendance d'intégration est une relation transitive.

A cause de la transitivité, les unités peuvent occasionnellement impliquer l'existence d'une interdépendance. C'est le cas où une unité dépend d'elle-même à travers une succession de dépendances. Les interdépendances sont souvent appelées les dépendances « cycliques » ou « cycles » car elles correspondent aux cycles dans les modèles de graphes, les modèles qui sont souvent utilisés pour planifier le test d'intégration. C'est l'existence de cycles qui rend difficile l'ordonnancement.

Définition 5. *Une interdépendance* (*IN-DEP*) *est un ensemble d'unités dont chaque unité a une dépendances vis à vis d'elle-même.*

$$IN\text{-}DEP \leftarrow \{U_1, U_2, \ldots, U_n \in P^n \mid (U_1 \; ? \; U_2) \wedge (U_2 \; ? \; U_3) \wedge \ldots \wedge (U_n \; ? \; U_1)\}$$

Une interdépendance peut exister isolément mais elle peut croiser d'autres interdépendances et créer un groupe interdépendant dont pour chaque couple d'unités, il existe une dépendance bidirectionnelle (directe ou indirecte). Dans la théorie des graphes qu'on utilise pour planifier le test d'intégration, ces groupes interdépendants correspondent à des composantes fortement connexes (CFC). Les CFCs rendent plus difficile la planification.

Définition 6. *Un groupe interdépendant* (*GIN-DEF*) *est un ensemble d'unités dans lequel il existe une dépendance bidirectionnelle* (*directe ou indirecte*) *entre chaque couple d'unités.*

$$GIN\text{-}DEP \leftarrow \{U \in P \mid \forall (U_1, U_2) \in GIN\text{-}DEP^2, (U_1 \; ? \; U_2) \wedge (U_1 \; ? \; U_2\}\}$$

2. Solution de l'équipe de David C. Kung

La solution pour le test d'intégration de l'équipe de David C. Kung [Kung 1995 #1, Kung 1995 #2 et Kung 1996] est composée de trois étapes. La première étape consiste à modéliser la structure de dépendances d'intégration d'un système en analysant le code d'implémentation du système sous test. Cette modélisation permet d'obtenir un graphe de relation des objets (ORG – « Object Relation Graph »). La deuxième étape consiste à transformer cet ORG en un ORG acyclique en essayant de minimiser le coût de création des « *bouchons* ». La dernière étape est d'ordonner la tâche d'intégration en se basant sur cet ORG acyclique.

2.a. Construction du Graphe de Relation des Objets à partir du code

Comme signalé dans le paragraphe 4.b du Chapitre I, la planification du test d'intégration est un problème d'ordonnancement qui est très bien représenté par un modèle de graphe. En 1995, Kung a proposé un ensemble de règles pour transformer les dépendances d'intégration d'un système en un graphe orienté et étiqueté, nommé le «*Graphe de relation des objets*» (ORG – Object Relation Graph). Kung a donné la définition des ORGs en se basant sur trois types de dépendances : héritage, agrégation et association. Notons que la notion d'agrégation et d'association de Kung n'est pas identique à celle que l'on retrouve dans le langage UML. Ces relations sont obtenues en analysant le code source. Les définitions sont illustrées par des exemples de code C++. Kung utilise la classe comme unité de base pour l'intégration.

Pour modéliser les dépendances, il définit un graphe orienté et un graphe orienté-étiqueté :

Définition 7. *Un graphe orienté GO = (N, A) est un graphe tel que* :

- $N = \{N_1, N_2, \ldots, N_n\}$: *ensemble fini de nœuds.*
- $A \subseteq N \times N$: *ensemble d'arcs orientés.*

Définition 8. *Un graphe orienté et étiqueté GOE = (N, E, A) est un graphe tel que* :

- $N = \{N_1, N_2, \ldots, N_n\}$: *ensemble fini de nœuds.*
- $E = \{E_1, E_2, \ldots, E_m\}$: *ensemble fini d'étiquettes des arcs.*
- $A \subseteq N \times N \times E$: *ensemble d'arcs orientés et étiquetés.*

L'ensemble de GOE est donc sous-ensemble de GO :

$$\text{GOE} \subseteq \text{GO}$$

Toutes les opérations sur un GO sont applicables sur un GOE. Dans la suite, si les opérations introduites ne nécessitent pas d'étiquettes, nous les présentons à travers un GO mais pas un GOE.

Un ORG est un graphe orienté et étiqueté dont les arcs sont étiquetés par une des trois étiquettes : « **I** » pour modéliser un héritage (« **I**nheritance »), « **Ag** » pour modéliser une **Ag**régation et « **As** » pour modéliser une **As**sociation.

Définition 9. *Un graphe de relation des objets (ORG) pour un programme à objets P est un graphe orienté et étiqueté ORG (N, E, A) où* :

- $N = \{N_1, N_2, \ldots, N_n\}$: *ensemble fini de nœuds, représentant les classes du programme P.*
- $E = \{I, Ag, As\}$: *ensemble fini d'étiquettes des arcs.*
- $A = A_I \cup A_{Ag} \cup A_{As}$: *ensemble d'arcs définis ci-dessous.*

où l'ensemble des arcs d'héritage A_I, des arcs d'agrégation A_{Ag} et des arcs d'association A_{As} sont définis comme suit :

Définition 10. *L'ensemble d'arcs d'héritage A_I Í $N \times N \times E$ est un ensemble d'arcs orientés, représentant la relation d'héritage entre les classes. Pour n'importe quel couple de classes $(C_1, C_2) \in N^2$, un arc $(C_1, C_2, I) \in A_I$ indique que la classe C_1 est une classe dérivée de la classe C_2. Dans le langage C++, $(C_1, C_2, I) \in A_I$ est défini si et seulement si une de ces déclarations apparaît dans l'entête de la classe C_1 :*

- *« class C_1 : C_2 » ou « class C_1 : private C_2 ».*
- *« class C_1 : public C_2 ».*
- *« class C_1 : protected C_2 ».*

Définition 11. *L'ensemble d'arcs d'agrégation A_{Ag} Í $N \times N \times E$ est un ensemble d'arcs orientés, représentant la relation d'agrégation entre les classes. Pour un couple de classes $(C_1, C_2) \in N^2$, un arc $(C_1, C_2, Ag) \in A_{Ag}$ indique que la classe C_1 contient un ou plusieurs objet(s) de la classe C_2. Dans le langage C++, $(C_1, C_2, Ag) \in A_{Ag}$ est défini si et seulement si une de ces conditions est satisfaite :*

- *Un objet de la classe C_2 est déclaré comme un attribut de la classe C_1. C'est une agrégation automatique.*
- *Un objet de la classe C_2 est déclaré comme un attribut statique de la classe C_1. C'est une agrégation statique.*
- *Un objet de la classe C_2 est crée dynamiquement par une fonction de la classe C_1. C'est une agrégation dynamique.*

Définition 12. *L'ensemble d'arcs d'association A_{As} Í $N \times N \times E$ est un ensemble d'arcs orientés, représentant la relation d'association entre les classes. Pour un couple de classes $(C_1, C_2) \in N^2$, un arc $(C_1, C_2, As) \in A_{As}$ indique que la classe C_1 est associée à la classe C_2 de l'une de ces trois façons :*

- *La classe C_1 utilise un attribut de la classe C_2. C'est une dépendance par données.*
- *Une fonction de la classe C_2 est invoquée par une fonction de la classe C_1. C'est un passage d'un message.*
- *Un objet de la classe C_2 est déclaré comme un paramètre formel d'une méthode de la classe C_1. C'est un passage d'un paramètre par un objet.*

Dans le langage C++, $(C_1, C_2, Ag) \in A_{Ag}$ est défini si et seulement si une des trois conditions ci-dessus est satisfaite.

La Figure 14 présente un ORG entre les classes Véhicule, Voiture, Moteur, Pneu et Passager. Un passager doit utiliser un véhicule, un véhicule a un moteur et des pneus. La voiture est un type de véhicule.

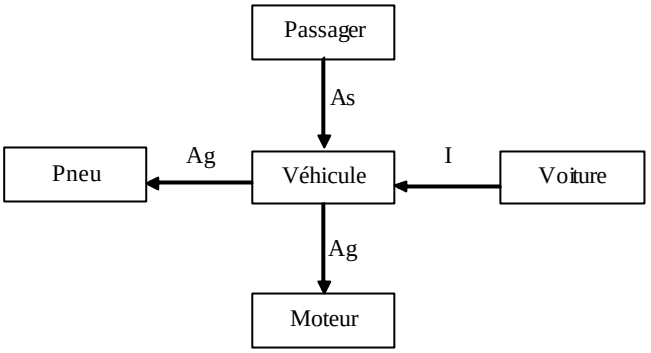

Figure 14. Un ORG.

2.b. Des unités interdépendantes aux composantes fortement connexes

Dans un graphe orienté, une suite de nœuds (ou d'arcs) successifs forme un chemin.

Définition 13. *Un chemin « C » dans un graphe orienté GO = (N, A) est une suite des nœuds :*

$$C \leftarrow (n_1\ n_2 \ldots n_m) \in N^m \mid \forall i = 1..\ m-1, \exists (n_i, n_{i+1}) \in A$$

le nœud n_1 est le début du chemin, le nœud n_m est la fin du chemin.

la note « n_1 ? n_2 » montre l'existence d'un chemin dont n_1 est le début et n_2 la fin.

Une suite de nœuds peut fermer et former un cycle :

Définition 14. *Un cycle « c » est un chemin C ($n_1\ n_2 \ldots n_m$) tel que :*

$$\exists (n_m, n_1) \in A$$

Par la définition, on a :

Propriété 1. *Pour chaque nœud d'un cycle, il existe un chemin vers lui-même :*

$$\forall n \in c, \exists (n\ ?\ n)$$

Dans la décomposition de CFCs on ne s'intéresse qu'aux cycles élémentaires – les cycles dont chaque nœud apparaît une seule fois.

Définition 15. *Un cycle élémentaire «c_e» d'un GO (N, A) est un cycle c dont chaque nœud apparaît une seule fois*

$$c_e \leftarrow \mathrm{c}\ (n_1\ n_2 \ldots n_m) \mid \forall (i, j) = 1..m,\ i \neq j \Leftrightarrow n_i \neq n_j$$

A partir de ce point, s'il n'y pas de précisions, tous les cycles abordés sont compris comme des cycles élémentaires.

Un graphe qui n'a pas de cycle est appelé un graphe acyclique. Pour un ORG acyclique, il ne faut pas créer de bouchons pour faire le test d'intégration (voir l'exemple de la Figure 7) car il n'y a pas d'interdépendance dans le système. On possède alors une intégration par les feuilles.

Définition 16. *Un graphe orienté est acyclique* (*GOA*) *s'il n'existe aucun cycle dans ce graphe.*

$$GOA = (N, A) \mid \forall n \in N, ?\ (n\ ?\quad n)$$

Par contre, un graphe qui possède au moins un cycle est appelé un graphe cyclique. Pour un ORG cyclique qui représente la structure de dépendances, il faut vérifier si l'on doit créer des bouchons pour faire le test d'intégration.

Définition 17. *Un graphe orienté est cyclique* (*GOC*) *s'il existe au moins un cycle dans ce graphe.*

$$GOC = (N, A) \mid \exists n \in N, \exists(n\ ?\quad n)$$

Avec un ORG, une interdépendance est représentée par un cycle et un groupe interdépendant est représenté par un groupe de composantes fortement connexes (CFC) qui se définit comme suit :

Définition 18. *Un groupe des composantes fortement connexes* (*CFC*) *d'un graphe orienté GO = (N, A) est un ensemble de nœuds tel que pour chaque couple de nœuds a et b de la CFC, il existe un cycle auquel ces deux nœuds participent.*

$$CFC \leftarrow \{n \in N \mid \forall(a, b) \in CFC^2, \exists c \subseteq CFC, (a, b) \in c^2\ \}$$

Pour généraliser l'application des algorithmes sur les CFCs, présentés dans ce rapport, on considère tout nœud qui n'appartient à aucune CFC de la Définition 18 comme une CFC « *triviale* ».

Définition 19. *Une CFC triviale est un nœud n qui n'appartient à aucun cycle.*

$$CFCt \leftarrow \{n\} \mid n \in N \wedge ?\ (n\ ?\quad n)$$

Dans la suite, s'il n'y a pas de précision, le terme CFC est utilisé pour les deux types de CFC, triviale et non-triviale.

2.c. Décomposition des CFCs par les arcs de type « *Association* »

Kung argue que dans les trois types de dépendances (héritage, agrégation et association), la simulation de la dépendance d'association est la moins risquée car parmi ces trois relations, l'association est la moins forte. Il estime donc que la création d'un bouchon correspondant à un service représenté par une association est le moins coûteux de tous.

Il considère qu'il y a trois niveaux de relation, correspondant aux trois types de dépendances :

i. Avec l'héritage, les unités sont construites « *de haut en bas* », de l'unité la plus générale à l'unité la plus spécifique. Les unités en bas de chaîne d'héritage utilisent toutes les données et les opérations visibles de leurs ascendants

ii. Avec l'agrégation, les unités sont construites «*de bas en haut* », de l'unité la plus simple à l'unité la plus complexe. Les unités en haut de chaîne d'agrégation peuvent utiliser toutes les données et les opérations visibles de leurs unités.

iii. Avec l'association, les unités associées n'utilisent pas entièrement les données et les opérations visibles d'autres unités associées donc cette dépendance est la plus faible des trois types de dépendance.

Kung affirme qu'une interdépendance contient au moins une dépendance d'association. Nous n'avons trouvé nul part la preuve pour cet argument. Au contraire, la Figure 8 montre le cas d'une interdépendance sans dépendance d'association après la définition de Kung (voir la Définition 10 et Définition 11).

En se basant sur ces deux arguments, Kung propose d'éliminer successivement « *n'importe quel arc d'association* » dans une CFC jusqu'au moment où il n'y a plus de CFC. Les unités successeurs de ces arcs sont simulées et deviennent les bouchons de test.

Par exemple, si on a un ORG comme celui de la Figure 15, un résultat de la méthode de Kung est décrit dans la Figure 16. Notons que l'on peut obtenir plusieurs solutions avec l'indéterminisme de la formule «*n'importe quel arc d'association* » de Kung.

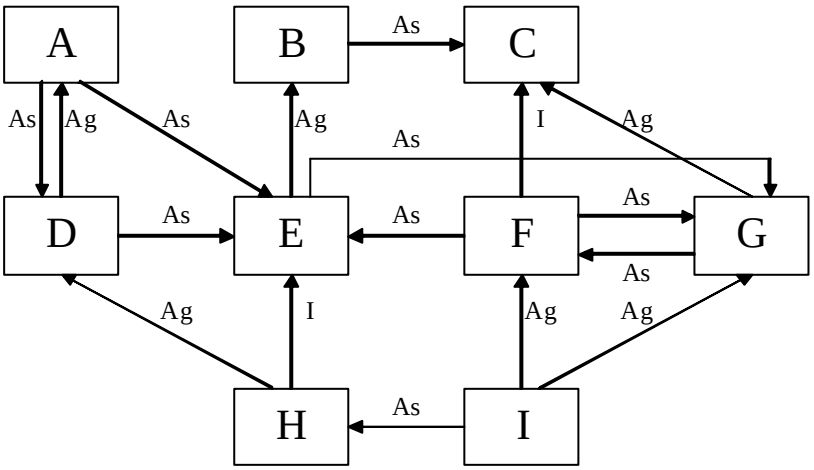

Figure 15. Un ORG pour illustrer les stratégies d'intégration.

Dans la Figure 16, les CFC non-triviales sont décrites par des rectangles gris. Les arcs d'association représentés en pointillés sont ceux qui peuvent être éliminés pour décomposer des CFCs. Si on considère que l'on crée des bouchons spécifiques seulement, on en compte trois : D pour intégrer A (notons D_A), F pour G (F_G) et G pour E (G_E). Ces bouchons apparaissent avec un double cadre.

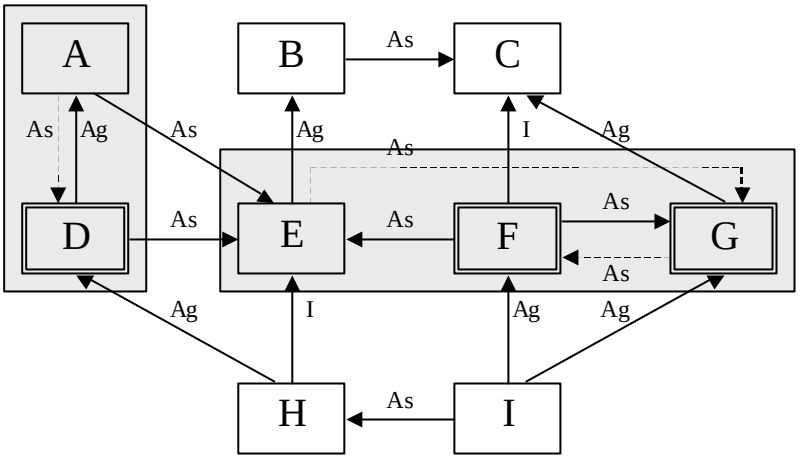

Figure 16. Kung : les bouchons pour l'ORG de la Figure 15.

Avec ces bouchons (D_A, G_E, F_G), on peut considérer que les nœuds de l'ORG de la Figure 15 créent un autre ORG qui est présenté dans la Figure 17 où les bouchons de test sont figurés explicitement. Ce nouvel ORG est acyclique puisqu'il n'a plus d'interdépendance.

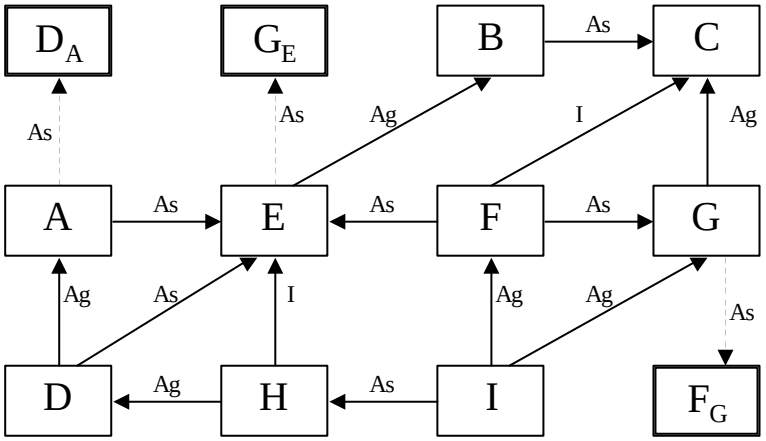

Figure 17. Kung : la décomposition les CFCs de l'ORG de la Figure 15.

2.d. Ordonnancement du test d'intégration

Kung utilise un couple de poids pour planifier le test d'intégration. Kung suppose une construction d'un graphe ORG' :

$$ORG' = (N', E, A')$$

où :

i. N' : l'ensemble de nœuds. Chaque nœud $n \in N'$ est une CFC (triviales ou non-triviales) de l'ORG du système sous test .

ii. A' : l'ensemble d'arcs. Les arcs $a \in A'$ de cet ORG' sont des arcs de l'ORG origine qui relient les CFCs, les unes aux autres.

Par sa construction, l'ORG' est un graphe acyclique car les CFCs deviennent les nœuds. On peut faire le test d'intégration pour chaque nœud de cet ORG' sans se soucier du problème d'interdépendance. Pour les intégrer, Kung utilise leur hauteur – la distance maximale entre le nœud et une feuille comme l'ordre d'intégration. Cette hauteur est calculée par l'algorithme de recherche en profondeur. Kung appelle cette hauteur le niveau majeur (NM) :

$$NM(n) \leftarrow \begin{cases} 1 & \Leftarrow S(n) = f \\ 1 + \max\left\langle NM(m) \middle| \; m \in S(n) \right\rangle & \Leftarrow S(n) \neq f \end{cases}$$

où $S(n)$ est l'ensemble des successeurs de n dans le nouvel ORG' (un nœud pouvant correspondre à une CFC) :

$$S(n) \leftarrow \{m \in N' | \; \exists e \in E \wedge \exists (n, m, e) \in A'\}$$

Autrement dit, Kung attribue un niveau majeur à chaque CFC de l'ORG originel. Les nœuds d'une CFC prennent le niveau majeur de la CFC. La Figure 18 présente le niveau majeur des CFCs de l'ORG de la Figure 15. Les indices des niveaux majeurs sont notés au coin de chaque CFC. Le Tableau 4 récapitule ces hauteurs. On obtient alors un premier ordonnancement en couches des CFCs en commençant par celles de niveau majeur 1. Il reste à ordonner les nœuds dans chaque CFC.

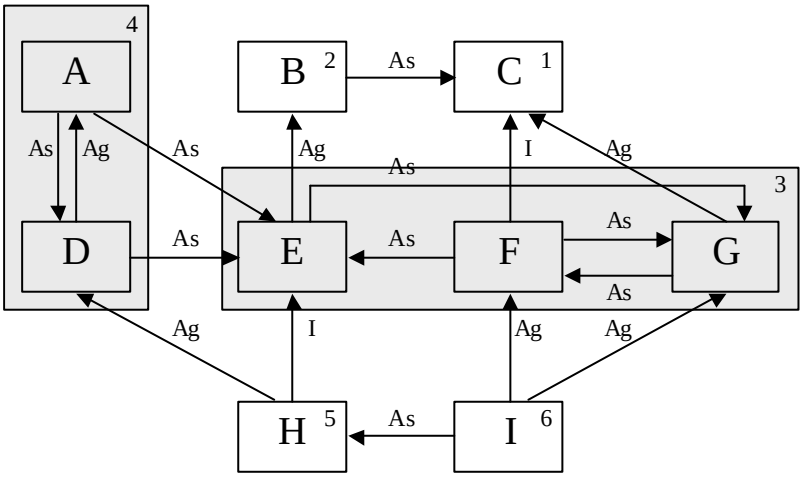

Figure 18. Kung : la hauteur des CFCs de l'ORG de la Figure 15.

Tableau 4. Kung : la hauteur des CFCs de l'ORG de la Figure 15.

CFC	Niveau majeur
A D	4
B	2
C	1
E F G	3
H	5
I	6

Après avoir décomposé les CFCs, Kung attribue à chaque nœud un autre poids qui est sa hauteur « *locale* ». Cette hauteur est calculée à l'intérieur de chaque CFC mais sans les arcs éliminés. C'est aussi un travail effectué par l'algorithme de recherche en profondeur. Kung appelle cette hauteur le niveau mineur (nm) :

$$nm(n) \leftarrow \begin{cases} 1 & \Leftarrow S(n) = f \\ 1 + \max\left\langle nm(m) \middle| \ m \in S(n) \right\rangle & \Leftarrow S(n) \neq f \end{cases}$$

Etant donné l'ensemble des arcs éliminés A_e dont on ne tient pas compte, l'ensemble $S(n)$ des successeurs de n dans la CFC de n est:

$$S(n) \leftarrow \{m \in CFC \mid n \in CFC \wedge \exists e \in E \wedge \exists (n, m, e) \in A \setminus A_e\}$$

La Figure 19 présente les deux niveaux des nœuds de l'ORG de la Figure 15. Le niveau majeur est noté avant le niveau mineur.

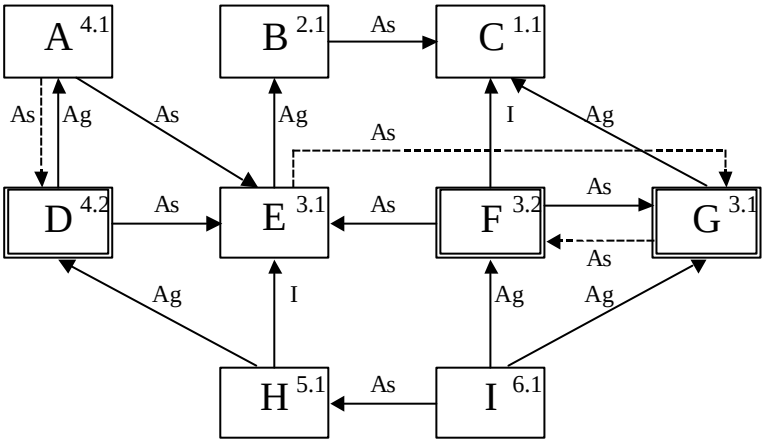

Figure 19. Kung : les couples (niveau majeur – niveau mineur) des nœuds de l'ORG de la Figure 15.

La stratégie consiste ensuite à tester les nœuds suivant l'ordre croissant des couples (NM(n), nm(n)). Avec un couple (NM(n), nm(n)), l'ordre d'intégration est :

$$(NM(n_1) < NM(n_2)) \vee ((NM(n_1) = NM(n_2) \wedge (nm(n_1) < nm(n_2)) \Rightarrow n_1 \ ? \ n_2$$

Si plusieurs nœuds ont le même niveau, Kung dit qu'on peut les tester dans n'importe quel ordre. Le Tableau 5 présente un des résultats possibles de l'idée de Kung, appliquée sur l' ORG de la Figure 15.

Tableau 5. Kung : l'ordre d'intégration correspondant à la Figure 19.

Ordre	Nœud	Niveaux	Bouchon utilisé
1	C	1.1	
2	B	2.1	
3	E	3.1	G
4	G	3.1	F
5	F	3.2	
6	A	4.1	D
7	D	4.2	
8	H	5.1	
9	I	6.1	

2.e. Résumé

Après avoir construit la structure de dépendances à partir du code du système sous test, Kung considère les CFCs non-triviales comme un seul nœud dans un nouvel ORG. Il associe à chaque nœud de ce nouvel ORG sa hauteur – la distance maximale entre le nœud et une feuille du nouvel ORG.

Cette hauteur est utilisée comme le niveau majeur pour planifier le test d'intégration. Le niveau majeur d'une CFC non-triviale est aussi associé à tous ses nœuds. L'intégration est ordonné en premier temps par l'ordre croissant de ce niveau majeur : le nœud ayant le niveau majeur plus petit est intégré avant le nœud ayant le niveau majeur plus grand.

Ensuite, les CFCs non-triviales de l'ORG originelle sont traitées. Kung élimine successivement des arcs d'association jusqu'au moment où les CFCs non-triviales disparaissent Les CFCs non-triviales sont donc décomposées.

Les nœuds d'une ancienne CFC non-triviale sont associés au deuxième attribut : sa hauteur dans la CFC. Cette hauteur est utilisée comme le niveau mineur pour planifier le test d'intégration. Entre deux nœuds ayant le même niveau majeur, le nœud ayant le niveau mineur plus petit est intégré avant le nœud ayant le niveau mineur plus grand.

2.f. Commentaire

La modélisation de Kung utilise le code du système sous test comme source de la structure de dépendances. Par conséquent, la solution de Kung ne convient pas pour la planification du test d'intégration, qui doit être faite dès la phase de conception.

À l'exception de la relation d'héritage, les dépendances modélisées par Kung n'ont pas la même signification que dans la notation UML. Les associations de Kung sont des dépendances dans l'UML. Les dépendances d'agrégation contiennent les compositions ainsi que les associations.

Le traitement de Kung n'est pas complète. Kung ne traite pas le polymorphisme, les classes abstraites, les interfaces, les classes génériques.

L'idée de Kung est très simple : il élimine successivement n'importe quel arc d'association dans une CFC pour éliminer les interdépendances. Comme Kung ne présente pas de façon de grouper les bouchons créés en éliminant les arcs d'association dans un bouchon réaliste, les bouchons créés par la stratégie de Kung sont des bouchons spécifiques. Le résultat de la décomposition des CFCs de Kung n'est pas stable car Kung ne fixe aucune critère pour choisir une dépendance d'association à simuler.

Cette décomposition des CFCs n'est qu'une solution « *locale* ». Kung a tenté de diminuer le coût de création de chaque bouchon en éliminant les arcs d'association mais il ne tient pas compte du coût total de création de bouchons. Rien n'assure qu'une solution qui est localement bonne est aussi globalement bonne.

L'argument qu'un cycle doit posséder au moins une relation d'association est trop restrictif. Un cycle peut exister sans arc d'association. D'après la Définition 11, les classes INTEGER et INTEGER_REF de la Figure 8 sont liées par une relation d'agrégation (la classe INTEGER_REF déclare un attribut de type INTEGER). Sur la Figure 8, il existe donc un cycle entre deux classes qui ne contient qu'une relation d'héritage et une relation d'agrégation, après la définition de Kung de ces deux types de dépendances.

La modélisation de Kung peut donner des arcs «doublons » entre deux nœuds. Le fait que Kung ne supprime pas les doublons peut causer la création de bouchons inutiles. Par exemple, si on a un ORG comme celui de la Figure 20, on peut choisir de simuler la dépendance d'association représentée par l'arc d'association (*a*, *b*, *As*). En réalité, cette simulation est inutile parce qu'il existe encore une autre dépendance (*a*, *b*, *I*) entre ces deux classes donc l'interdépendance existe toujours. La bonne simulation doit être celle de l'arc (*b*, *a*, *As*).

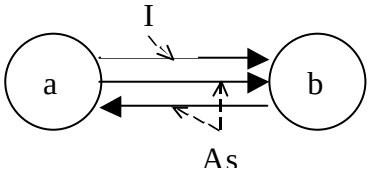

Figure 20. Kung : la simulation inutile à cause des doublons.

Kung ne prévoit pas les étapes de « *retest* » afin de tester chaque unité dans son environnement final (comme on a présenté à la fin du paragraphe 4.c, « Premier problème – Décomposition des interdépendances » du Chapitre I, « Introduction »).

Enfin, Kung donne un ordre d'intégration qui est alloué pour un seul testeur. Cependant, le test est souvent une tâche effectuée par un groupe de plusieurs testeurs. Par conséquent, il faut paralléliser le test d'intégration. Dans ces cas, la solution de Kung n'est pas une solution efficace. Nous allons présenter notre solution dans le Chapitre V, « Parallélisation du test d'intégration ».

3. Proposition de Kuo Chung Tai et Fonda J. Daniels

En 1999, Kou Chung Tai et Fonda J. Daniels [Tai-Daniels 1999] ont publié leur solution pour le test d'intégration. Pour la modélisation des dépendances du système sous test, ils reprennent l'ORG de Kung (voir le paragraphe 2.a). Ils ont proposé une autre décomposition des interdépendances. Quant à la planification, ils utilisent un mécanisme basé sur un couple de deux niveaux comme celui de Kung (voir le paragraphe 2.d) mais avec une autre stratégie de numérotation pour calculer les niveaux majeurs et les niveaux mineurs. De plus, ils ont ajouté à leur planification les étapes de retest, ce qui n'existe pas dans la solution de Kung.

3.a. Classement de l'ORG en couches

Tai-Daniels admettent l'existence, dans quelques langages de programmation à objets, des interdépendances sans dépendance d'association. Cependant, ils se restreignent aux langages de programmation à objets pour lesquels toute interdépendance est due aux dépendances d'association.

Sous cette condition, dans un premier temps, ils créent un ORG' du système sous test qui ne contient pas de dépendance d'association :

$$ORG' = (N, E', A')$$

dont :

- $E' = \{I, Ag\}$
- $A' = A \setminus A_{As}$

où A_{As} est l'ensemble des arcs d'association.

$$A_{As} \leftarrow \{(p, s, As) \in A \mid (p, s) \in N^2\}$$

Avec cette construction, l'ORG' est un graphe acyclique car les cycles existent seulement avec les dépendances d'association. Ils classent les nœuds suivant leur hauteur « $h(n)$ » dans l'ORG'. Cette hauteur est calculée par une recherche en profondeur sur l'ORG' :

$$h(n) \leftarrow \begin{cases} 1 & \Leftarrow S(n) \neq f \\ 1 + \max\left\langle h(m) \middle| \; m \in S(n) \right\rangle & \Leftarrow S(n) \neq f \end{cases}$$

où $S(n)$ est l'ensemble des successeurs de n, sans compter les successeurs des arcs d'association :

$$S(n) \leftarrow \{m \in N \mid \exists e \in E' \wedge \exists (n, m, e) \in A'\}$$

On voit que les nœuds qui ont une même hauteur «i » forme une couche $C(i)$ (voir la Figure 21) :

$$C(i) \leftarrow \{n \in N \mid h(n) = i\}$$

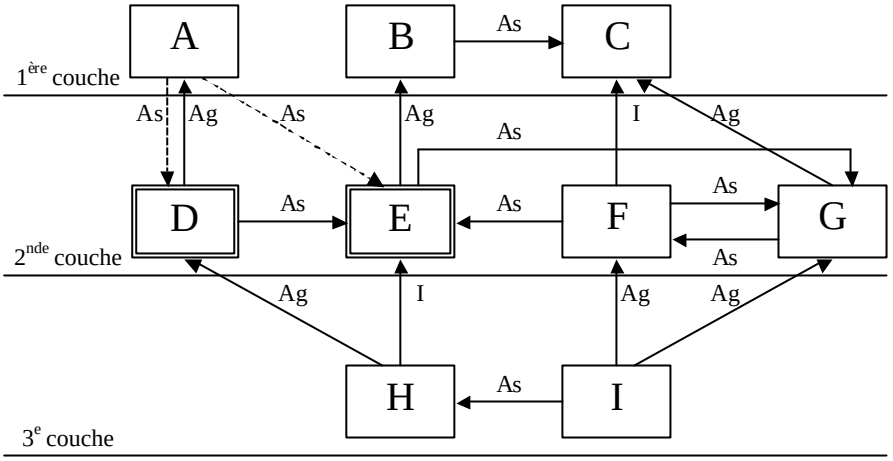

Figure 21. Tai-Daniels : les couches de l'ORG de la Figure 15.

Retournons à l'ORG origine avec tous les arcs d'associations, Tai-Daniels éliminent tous les arcs d'association inter-couches. Les successeurs de ces arcs sont simulés pour créer des bouchons (inter-couches). S'il y a des cycles dans l'ORG d'origine, les cycles inter-couches n'existent plus parce que tous les arcs d'association inter-couches sont éliminés.

Dans la Figure 21, les arcs pointillés sont éliminés et les unités successeurs sont simulées. En éliminant les arcs, Tai-Daniels créent les bouchons spécifiques : D_A et E_A.

3.b. Décomposition de CFC dans chaque couche en éliminant les arcs d'association

Maintenant, dans chaque couche C(i), s'il y a un cycle qui contient un bouchon, créé pour intégrer un nœud de la couche inférieure C(j), avec j < i, Tai-Daniels considèrent que le bouchon est réutilisé pour casser le cycle.

Autrement dit, le bouchon inter-couche créé est considéré comme un bouchon réaliste et tous les arcs entrants et participants à un cycle sont éliminés. Dans le cas du cycle (E F G), puisque le bouchon E est créé pour intégrer A (voir la Figure 22) l'arc FE est donc aussi éliminé et ce cycle est cassé.

Avec une telle approche, le nombre de bouchons spécifiques augmente mais le nombre de bouchons réalistes n'augmente pas.

S'il y a un cycle dans une couche C(i) qui ne contient pas de bouchon inter-couches (comme le cycle (F G) dans la Figure 22), ils attribuent pour chaque arc (*p, s, As*) de ce cycle un poids *P* qui est calculé comme suit :

$$P_{(p,\,s,\,As)\,\in\,A,\,(p,\,s)\,\in\,C^2(i)} \leftarrow |\{(x, p, As) \mid x \in C(i)\}| + |\{(s, y, As) \mid y \in C(i)\}|$$

L'arc avec le poids le plus grand est éliminé et l'unité successeur est simulée. Le processus est répété jusqu'au moment où les cycles n'existent plus.

Dans la Figure 22, le poids de l'arc FG est 2 (GF + GF) et le poids de l'arc GF est 3 (EG + FG + FG) : l'arc GF va être éliminé. Il y a donc quatre bouchons spécifiques (D_A, E_A, E_F et F_G) qui correspondent aux trois bouchons réalistes : D, E et F.

Le bouchon spécifique D_A correspond au simulateur spécifique de l'unité D pour l'utilisation qu'en fait le client A. Si on considère que l'on crée des bouchons spécifiques seulement, on en aura plus que dans le cas où on considère des bouchons réalistes seulement. En effet, un bouchon réaliste pour E sert aussi bien pour le client A que pour le client F.

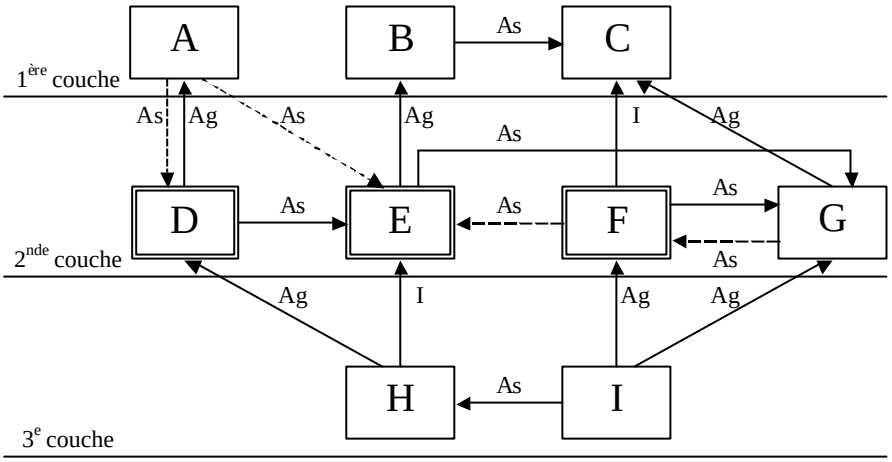

Figure 22. Tai-Daniels : les bouchons pour l'ORG de la Figure 15.

Avec ces bouchons, les nœuds de l'ORG de la Figure 15 créent un ORG acyclique dont la construction suit le même mécanisme que celui de la Figure 17 (nous ne présentons plus l'ORG acyclique résultat)

3.c. Caractéristiques de l'ordonnancement de Tai-Daniels

Pour planifier le test d'intégration, Tai-Daniels ont précisé deux critères pour assurer que toutes les dépendances sont bien testées. Ils appellent ces critères les critères de « couverture d'arcs » car les arcs modélisent les dépendances. Chaque fois qu'une dépendance est testée, ils disent que l'arc correspondant est «*couvert* ». Nous présentons ces deux critères ci-dessous :

Critère 1. *Avant le test d'intégration d'une classe C, tous les arcs qui sortent de C doivent être couverts au moins une fois. S'il existe un arc, de C à une classe C' qui n'est pas encore intégrée, un bouchon de C' doit être utilisé pour couvrir cet arc C-C'.*

Ce critère correspond à ce que nous avons appelé la demande d'existence de service pour l'intégration (voir le paragraphe 4.a du Chapitre I). Par conséquent, si un chemin existe de la classe C à la classe C' qui ne contient que des arcs d'héritage ou des arcs d'agrégation, la classe C' doit être testée avant la classe C. Avec les hypothèses de Tai-Daniels, lorsqu'il n'y a que des dépendances d'héritage et

d'agrégation, il n'y a pas d'interdépendance : l'ordre d'intégration des unités est aussi l'ordre topologique des nœuds correspondants

Critère 2. *Après avoir intégré une classe C, tous les arcs qui entrent dans la classe C doivent être couverts au moins une fois. S'il existe un arc, sortant d'une classe C' qui est déjà intégrée et entrant dans la classe C, la classe C' doit être re-testée pour recouvrir cet arc.*

Dans ce cas, pour être intégrée, la classe C' a forcément utilisé un bouchon qui est un simulateur de la classe C. Comme un bouchon n'est pas une vraie unité, rien n'assure que l'unité C', qui travaille bien avec le bouchon de la classe C, va bien coopérer avec la classe C elle-même. Par conséquent, la classe C' doit être re-testée après le test de la classe C.

Cette idée de retester une unité avec son environnement réel et plus avec des bouchons est importante : nous la reprendrons dans notre approche.

Notons que l'ordre de retest doit respecter l'ordre de test. Par exemple, supposons qu'on a trois classes : C_1, C_2 et C_3 ; la classe C_1 est intégrée avant la classe C_2 et l'intégration de ces deux classes utilise un bouchon de C_3 (Figure 23 – a). L'ordre de test est donc (…C'_3, …, C_1, …, C_2 … C_3,…). Pour simplifier, on supprime les étapes de test intermédiaires et on obtient un ordre (C'_3, C_1, C_2, C_3) comme dans la (Figure 23 – b).

Dans ce cas, comme la classe C_1 est testée avant la classe C_2, il est possible que la classe C_2 utilise (directement ou indirectement) un service de la classe C_1 (la flèche pointillée dans la Figure 23 – a) Par conséquent, quand les deux classes C_1 et C_2 doivent être re-testées après avoir testé la classe C_3, la classe C_1 doit être re-testée avant la classe C_2 (Figure 23 – c).

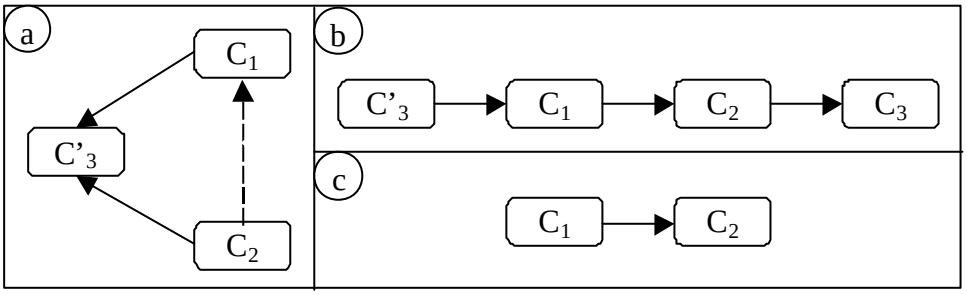

Figure 23. Tai-Daniels : l'ordre de retest avec un bouchon commun

a – utilisation de bouchon et dépendance éventuelle;
b – ordre de test ; c – ordre de retest.

Un autre cas est celui de deux clients avec deux serveurs différents : les classes C_1 et C_2 sont des clients des classes C_3 et C_4, respectivement (Figure 24 - a), C_1 est intégrée avant C_2, l'intégration de ces deux classes utilise des bouchons de C_3 et C_4 et l'intégration de ces deux dernières classes sont faites en même temps (Figure 24 - b) Pour la même cause que l'exemple précédant, quand les deux classes C_1 et C_2 doivent être re-testées après avoir intégré en même temps les deux classes C_3 et C_4, la classe C_1 doit être re-testée avant la classe C_2 (Figure 24 - c).

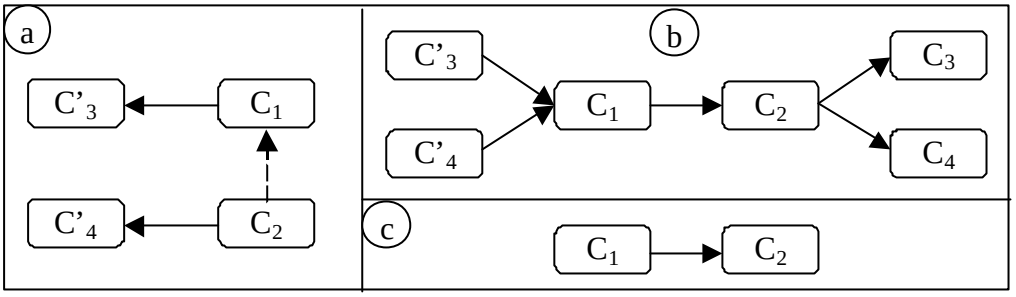

Figure 24. Tai-Daniels : l'ordre de retest avec deux bouchons séparés

a – utilisation de bouchon et dépendance éventuelle;
b – ordre de test ; c – ordre de retest.

3.d. Ordonnancement du test d'intégration

En profitant du même mécanisme d'ordonnancement que Kung, qui utilise des couples (niveau majeur, niveau mineur), Tai-Daniels utilisent les indices des couches comme le « niveau majeur ».

$$NM\,(n) \leftarrow h\,(n)$$

Le niveau mineur est la hauteur locale de chaque nœud dans son niveau majeur après avoir cassé les cycles :

$$nm(n) \leftarrow \begin{cases} 1 & \Leftarrow S(n) = f \\ 1 + \max\left\langle nm(m) \middle|\ m \in S(n) \right\rangle & \Leftarrow S(n) \neq f \end{cases}$$

où $S(n)$ est l'ensemble de successeurs de n dans sa couche :

$$S\,(n) \leftarrow \{m \in C(i) \mid h\,(n) = i \wedge \exists (n, m, As) \in A\}$$

Pour appliquer leurs propositions sur les caractéristiques d'un plan de test d'intégration (voir le paragraphe 3.c), ils ont ajouté à ce mécanisme les étapes de re-test. Quant aux nœuds qui ont le même couple (niveau majeur, niveau mineur) Tai-Daniels disent qu'on peut les intégrer en même temps. La Figure 25 présente les couples (niveau majeur, niveau mineur) pour les nœuds de l'ORG de la Figure 15. Le Tableau 6 donne l'ordre d'intégration, correspondant à la Figure 25, en expliquant les étapes de retest.

3.e. Résumé

Pour minimiser les risques d'utilisation et le coût de création des bouchons, Tai-Daniels décomposent l'ORG en couches et éliminent les arcs d'association inter-couches, ainsi que les arcs d'association ayant les plus « fortes probabilités » – après eux – de participer à la création des CFCs. Les bouchons créés pour éliminer des arcs inter-couches sont considérés comme des bouchons réalistes.

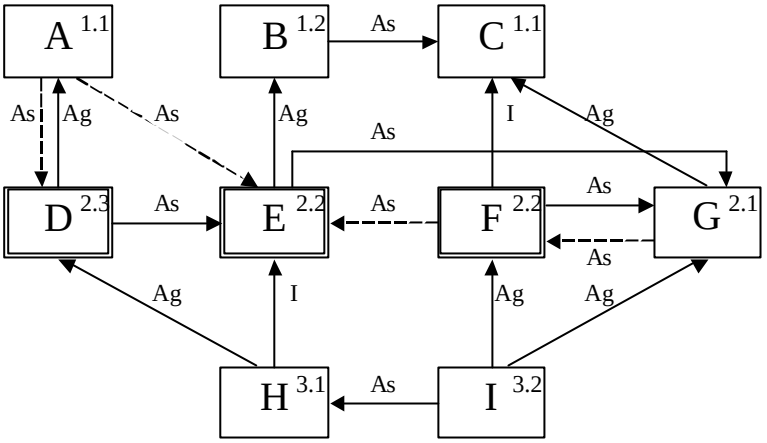

Figure 25. Tai-Daniels : les couples (niveau majeur – niveau mineur) des nœuds de l'ORG de la Figure 15.

Tableau 6. Tai-Daniels : l'ordre d'intégration correspondant à la Figure 25.

Ordre	Nœuds	Niveau	Bouchon	Retester
1	A	1.1	D, E	
2	C	1.1		
3	B	1.2		
4	G	2.1	F	
5	F	2.2	E	
6				G
7	E	2.2		
8				A
9				F
10	D	2.3		
11				A
12	H	3.1		
13	I	3.2		

Tai-Daniels introduisent les étapes de retests pour les nœuds qui sont testés en utilisant des bouchons. Ces étapes de retest sont effectuées après le test des unités originelles des bouchons utilisés. Nous reprendrons cette idée dans notre approche (voir le paragraphe 4 du Chapitre IV, « Décomposition des composantes fortement connexes »)

Pour planifier l'intégration, ils utilisent un même mécanisme que Kung mais au lieu de l'ordre des CFCs comme le niveau majeur, ils prennent l'ordre des couches.

3.f. Commentaire

En se basant sur la modélisation de Kung, Tai-Daniels reprennent tous ses désavantages : pas de traitement dynamique, pas de traitement des doublons, la modélisation n'est pas applicable dès la phase de conception.

En utilisant leur ordonnancement en couches, la solution de Tai-Daniels réclame trop de bouchons même s'ils ont précisé que le nombre de bouchons doit être minimum. La Figure 26 illustre cela : l'unité D est simulée inutilement (pour l'ordre d'intégration {A, B, C, D}) alors que les unités peuvent être intégrées sans bouchon (par l'ordre d'intégration {B, D, A, C}).

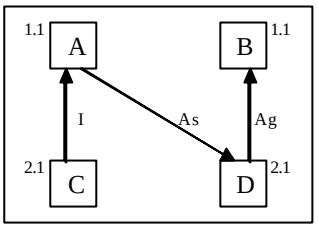

Figure 26. Tai-Daniels créent des bouchons inutiles.

Même s'ils ont parlé de l'intégration de deux classes « en même temps », ils ne précisent pas comment on peut paralléliser le test d'intégration.

4. Amélioration de la sélection des arcs à éliminer avec un poids : la proposition de l'équipe de Lionel Briand

En 2001, l'équipe de Lionel Briand [Briand 2001] a proposé une nouvelle solution basée sur les travaux de Kung et Tai-Daniels. Il propose d'associer un poids à chaque arc d'association basé sur la formule de Tai-Daniels (voir le paragraphe 3.b).

4.a. La probabilité de participer à la création des CFCs

Briand a changé deux éléments de cette formule. Premièrement, il prend la définition de niveau majeur proposée par Kung (voir le paragraphe 2.d) :

Avec

$$ORG' = (N', E, A')$$

où :

i. N' : l'ensemble des CFCs (triviales et non-triviales) de l'ORG du système sous test.

ii. A' : l'ensemble d'arcs de l'ORG origine qui relient les CFCs, les unes aux autres.

Le niveau majeur est la hauteur de chaque CFC dans le nouvel ORG' :

$$NM(n) \leftarrow \begin{cases} 1 & \Leftarrow S(n) = f \\ 1 + \max\langle NM(m) |\ m \in S(n)\rangle & \Leftarrow S(n) \neq f \end{cases}$$

où $S(n)$ est l'ensemble de successeurs de n dans le nouvel ORG', inclues les CFCs entières :

$$S(n) \leftarrow \{m \in N' | \exists e \in E \wedge \exists (n, m, e) \in A'\}$$

Deuxièmement, au lieu d'utiliser la somme de deux nombres d'arcs dans chaque niveau majeur, il utilise leur produit. Son argument est que cette multiplication correspond à la possibilité qu'un arc participe au plus grand nombre de cycles. Le poids associé à un arc d'association dans une CFC s'exprime donc de la manière suivante :

$$P_{(p, s, As) \in A, (p, s) \in CFC}{}^{2} \leftarrow |\{(x, p, As) \mid x \in CFC\}| + |\{(s, y, As) \mid y \in CFC\}|$$

Le Tableau 7 présente le poids des arcs d'association dans les CFCs de l'ORG de la Figure 15.

Tableau 7. Briand : le poids des arcs d'association des CFCs de la Figure 15.

Arc	CFC	Nombre d'arcs entrés	Nombre d'arcs sortis	Poids
AD	AD	1	1	1
EG	EFG	1	1	1
FE	EFG	1	1	1
FG	EFG	1	1	1
GF	EFG	2	2	4

L'arc avec le plus grand poids va être éliminé et l'unité successeur est simulée. Les poids seront mis à jour et la procédure va continuer jusqu'au moment où les cycles n'existent plus. Dans la Figure 27, l'arc AD va être éliminé pour décomposer la CFC {A, D}, l'arc GF va être éliminé pour décomposer la CFC {E, F, G}. Il y a donc deux bouchons spécifiques (D_A et F_G) qui correspondent (si on ne considère que ceux-ci) aux deux bouchons réalistes, D et F.

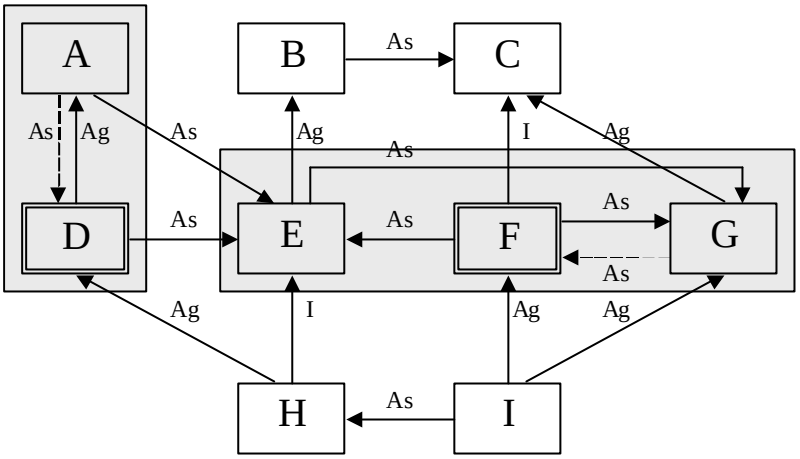

Figure 27. Briand : les bouchons sélectionnés pour l'ORG de la Figure 15.

Briand et al. ont repris aussi le mécanisme d'ordonnancement de Kung. Le niveau majeur de chaque CFC est le même que celui de Kung. Le niveau mineur est la hauteur interne de chaque nœud dans sa propre CFC. La Figure 28 présente les couples (niveau majeur – niveau mineur) de chaque nœud de l'ORG de la Figure 15 et le Tableau 8 montre l'ordre d'intégration correspondant.

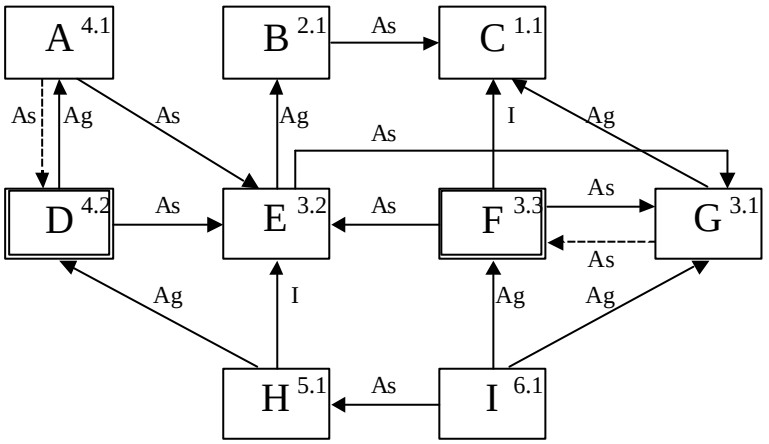

Figure 28. Briand : les couples (niveau majeur – niveau mineur) des nœuds de l'ORG de la Figure 15.

4.b. Résumé

Briand a amélioré la solution de Kung en associant un poids à chaque arc d'association dans une CFC afin de mieux choisir un bouchon. Ce poids, selon Briand, représente mieux la probabilité de participer à la création des CFCs par rapport à celui de Tai-Daniels.

Tableau 8. Briand : l'ordre d'intégration correspondant à la Figure 28.

Ordre	Nœud	Niveaux	Bouchon utilisé
1	C	1.1	
2	B	2.1	
3	G	3.1	F
4	E	3.2	
5	F	3.3	
6	A	4.1	D
7	D	4.2	
8	H	5.1	
9	I	6.1	

4.c. Commentaire

En utilisant les poids, la solution de Briand est plus stable que celle de Kung. Briand montre quel arc on va éliminer. Quant à Kung, il ne le précise pas. En plus, la complexité de l'algorithme pour décomposer des CFCs que Briand utilise est seulement linéaire donc, pour une planification qui n'est pas très exigeante, on peut utiliser la décomposition de Briand. Cependant, il faut y ajouter le traitement dynamique, le traitement des doublons, la parallélisation et les étapes de retest.

5. Traitement des liaisons dynamiques et de l'abstraction – Complément du travail de Yvan Labiche

Récemment, Yvan Labiche a publié avec ses collègues les résultats de leur recherche sur le test d'intégration [Labiche 2000 #2]. Ce travail est une partie de sa thèse [Labiche 2000 #1]. Il a utilisé l'ORG de Kung, appelé aussi diagramme de relation des objets (ORD – « Object Relation Diagram ») comme Tai-Daniels. Il ne traite pas le problème d'interdépendance, ce qui réduit considérablement l'application directe de cette approche. Ils présupposent donc l'application préalable d'un algorithme approprié. Le point d'entrée de leur solution est un ORG acyclique. Par exemple, reprenons l'ORG de la Figure 27 et y ajoutons les bouchons D_A (le bouchon de D pour intégrer A) et F_G (le bouchon de F pour intégrer G), on obtient un ORG acyclique dans la Figure 29. Le but du travail de cette équipe n'est pas d'abord de minimiser les bouchons de test, mais de faire un plan d'intégration précis qui spécifie pour chaque étape, l'environnement nécessaire à l'intégration (en tenant compte par exemple du polymorphisme).

Labiche a analysé les limitations du travail de Kung et ils ont proposé une solution pour dépasser ces limitations.

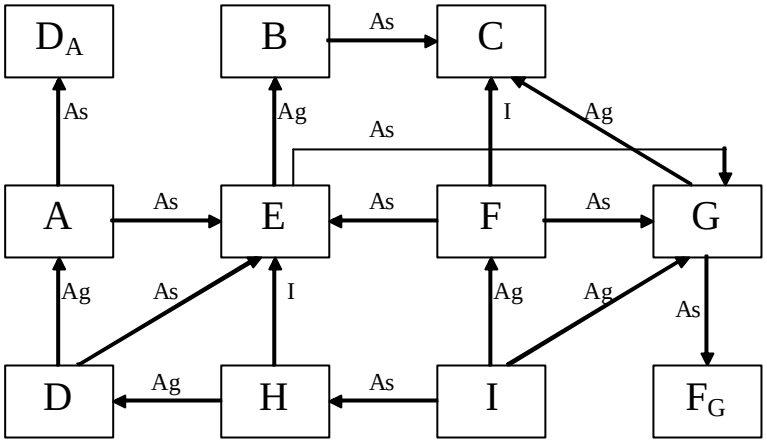

Figure 29. Un ORG acyclique – point de départ de la solution de Labiche

5.a. Les défauts du travail de l'équipe de David C. Kung

En analysant le travail de Kung, Labiche a trouvé trois limitations particulières :

i. *La traitement dynamique* : Kung ne traite pas la relation dynamique entre les classes. Par exemple, dans la Figure 29, la classe A est associée à la classe E, qui est la classe mère de la classe H. Donc, par le polymorphisme, A peut être dynamiquement associée à H. Par conséquent, A doit être intégrée en utilisant H.

ii. *L'information utile* : L'ordre d'intégration de Kung montre la classe à intégrer à chaque étape mais il ne donne pas les informations sur les classes desquelles la classe sous test dépend. Ces informations aident dans la construction des cas de test dans la phase de conception.

iii. *L'abstraction* : Quelques étapes dans l'ordre d'intégration de Kung sont irréalisables dans le cas où les classes sous test sont des classes abstraites ou des interfaces. Par conséquent, si on ne tient pas compte du polymorphisme, l'intégration des classes qui demandent des services des classes abstraites est aussi irréalisable.

5.b. Prise en compte du polymorphisme

Pour tenir compte du polymorphisme, Labiche utilise pour chaque classe X, deux ensembles de classes et une fonction booléenne associée à cette classe X.

- $D_1(X)$: l'ensemble de classes desquelles dépend la classe X statiquement. La dépendance statique existe entre deux classes C_1 et C_2 si et seulement s'il y a un chemin direct entre ces deux classes. Supposons Rs est la relation binaire, associée aux arcs orientés de l'ORG.

$$Rs \leftarrow \{(C_1, C_2) \mid \exists e \in E \wedge \exists (C_1, C_2, e) \in A\}$$

L'ensemble $D_1(X)$ est :

$$D_1(X) \leftarrow \{C_k \mid (X, C_k) \in Rs^+\}$$

où Rs^+ est la fermeture transitive de Rs.

- $D_2(X)$: l'ensemble de classes desquelles la classe X dépend statiquement ou dynamiquement. La dépendance dynamique existe entre deux classes C_1 et C_2 si et seulement s'il y a une troisième classe C_3, qui est le serveur de la classe C_1 et en même temps, est l'ancêtre de la classe C_2. Supposons Ru définit la relation d'utilisation, Ri définit la relation d'héritage et Rd représente les dépendances dynamiques :

$$Ru \leftarrow \{(C_1, C_2) \mid \exists e \in \{Ag, As\} \wedge \exists (C_1, C_2, e) \in A\}$$

$$Ri \leftarrow \{(C_1, C_2) \mid \exists (C_1, C_2, I) \in A\}$$

$$Rd \leftarrow Ru \; o \; Ri$$

où l'opération de composition « **o** » pour deux graphes orientés est définie comme suit :

$$GO\,(N, A) = GO_1\,(N_1, A_1)\; o\; GO_2\,(N_2, A_2) \Leftrightarrow$$

$$N = N_1 \cup N_2$$

$$A = \{(x, y) \mid \exists z \in N, (x, z) \in A_1 \wedge (z, y) \in A_2\}$$

L'ensemble $D_2(X)$ est :

$$D_2(X) \leftarrow \{C_k \mid (X, C_k) \in (Rs \cup Rd)^+\}$$

où $(Rs \cup Rd)^+$ est la fermeture transitive de $(Rs \cup Rd)$.

- $B_d(X)$: la fonction booléenne, indique si la classe X dépend dynamiquement d'au moins une classe de $D_2(X)$.

$$B_d(X) \leftarrow \begin{cases} 0 \Leftarrow \forall (C_1, C_2) \in (\{X\} \cup D_2(X)), (C_1, C_2) \notin Rd \\ 1 \Leftarrow \exists (C_1, C_2) \in (\{X\} \cup D_2(X)) \mid (C_1, C_2) \in Rd \end{cases}$$

La Figure 30 correspond à la Figure 29, enrichie par le traitement du polymorphisme. Dans cette figure, les flèches en pointillés sont des dépendances, ajoutées par le polymorphisme. Le Tableau 9 présente les ensembles $D_1(X)$, $D_2(X)$ et la fonction $B_d(X)$ pour les classes dans la Figure 30.

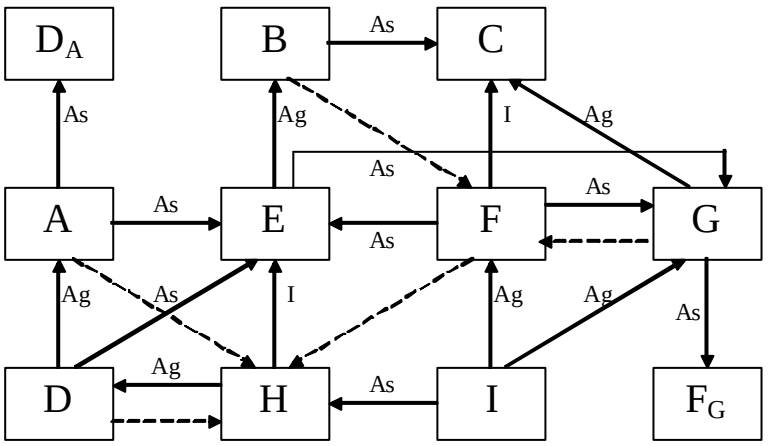

Figure 30. Labiche : prise en compte du polymorphisme pour l'ORG de la Figure 29.

Tableau 9. Labiche : $D_1(X)$, $D_2(X)$ et $B_d(X)$ pour les classes de l'ORG de la Figure 30.

Classe X	$D_1(X)$	$D_2(X)$	$B_d(X)$
A	(B, C, D_A, E, F_G, G)	(A, B, C, D, D_A, E, F, F_G, G, H)	1
B	(C)	(A, B, C, D, D_A, E, F, F_G, G, H)	1
C	ϕ	ϕ	0
D	(A, B, C, D_A, E, F_G, G)	(A, B, C, D, D_A, E, F, F_G, G, H)	1
D_A	ϕ	ϕ	0
E	(B, C, F_G, G)	(A, B, C, D, D_A, E, F, F_G, G, H)	1
F	(B, C, E, F_G, G)	(A, B, C, D, D_A, E, F, F_G, G, H)	1
F_G	ϕ	ϕ	0
G	(C, F_G)	(A, B, C, D, D_A, E, F, F_G, G, H)	1
H	(A, B, C, D, D_A, E, F_G, G)	(A, B, C, D, D_A, E, F, F_G, G, H)	1
I	(A, B, C, D, D_A, E, F, F_G, G, H)	(A, B, C, D, D_A, E, F, F_G, G, H)	1

A partir des ensembles $D_1(X)$, $D_2(X)$ et de la fonction $B_d(X)$, Labiche construit des triplets de test T (T.cible, T.besoin, T.type) où :

- T.cible : l'ensemble de classes sous test.
- T.besoin : l'ensemble de classes qu'on doit utiliser pour intégrer T.cible, y compris les classes sous test, T.cible $\subseteq$ T.besoin.
- T.type : Sta (pour statique) ou Dyn (pour dynamique) :

$$(T.besoin = T.cible \cup \bigcup_{X \in T.cible} D_1(X) \Rightarrow T.type = Sta$$

$$(T.besoin = T.cible \cup \bigcup_{X \in T.cible} D_2(X)) \wedge (\forall X \in T.cible, B_d(X) = 1) \Rightarrow T.type = Dyn$$

Le Tableau 10 présente les triplets de test pour les nœuds de l'ORG de la Figure 30. Pour le test statique, T.cible est une seule classe. Dans le cas du test dynamique, une CFC peut apparaître à cause des arcs ajoutés pour prendre en compte le polymorphisme. Les unités de la CFC ainsi créée doivent être testées ensemble. C'est le cas des unités A, B, D, E, F, G, H de l'ORG de la Figure 30.

Tableau 10. Labiche : les triplex de test pour les nœuds de l'ORG de la Figure 30.

T.cible	T.besoin	T.type
(A)	(A, B, C, D_A, E, F_G, G)	Sta
(B)	(B, C)	Sta
(C)	(C)	Sta
(D)	(A, B, C, D, D_A, E, F_G, G)	Sta
($D_{A)}$	($D_{A)}$	Sta
(E)	(B, C, E, F_G, G)	Sta
(F)	(B, C, E, F, F_G, G)	Sta
(F_G)	(F_G)	Sta
(G)	(C, F_G, G)	Sta
(H)	(A, B, C, D, D_A, E, F_G, G, H)	Sta
(I)	(A, B, C, D, D_A, E, F, F_G, G, H, I)	Sta
(A, B, D, E, F, G, H)	(A, B, C, D, D_A, E, F, F_G, G, H)	Dyn
(I)	(A, B, C, D, D_A, E, F, F_G, G, H, I)	Dyn

A partir du Tableau 10, Labiche construit l'ordre partiel pour intégrer les unités. Cet ordre est défini comme suit :

Définition 20. *L'étape de test d'intégration M* (*M.cible, M.besoin, M.type*) *précède l'étape N* (*N.cible, N.besoin, N.type*), *noté* :

$$M \,?\, N$$

si et seulement si :

$$(M.type = N.type \wedge M.besoin \subset N.besoin) \vee$$

$$((M.type = Sta \wedge N.type = Dyn) \wedge M.besoin \subseteq N.besoin)$$

La Figure 31 présente l'ordre partiel pour intégrer les nœuds de l'ORG de la Figure 30.

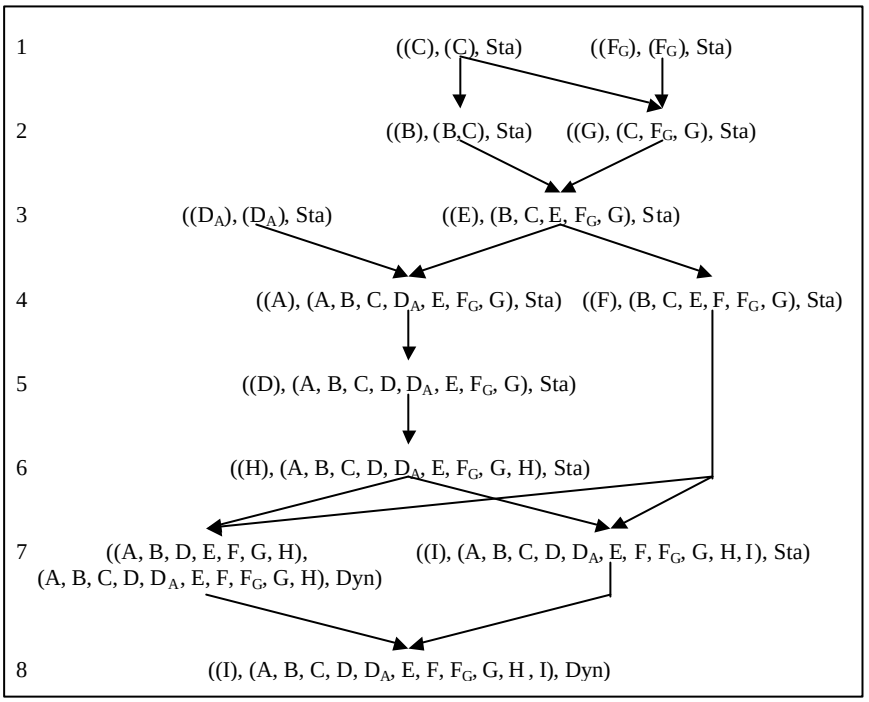

Figure 31. Labiche : l'ordre partiel pour intégrer les nœuds de l'ORG de la Figure 30.

5.c. Élargissement du système de notation

Dans la Figure 31, on connaît les classes utilisées pour intégrer un ensemble de classes (T.besoin) et le type du test d'intégration (T.type). Cependant, on ne connaît ni la relation entre les classes, ni quelle classe doit créer l'objet, ni même quelle classe introduit le polymorphisme... On peut trouver toutes ces informations sur un ORG. Pour ajouter ces informations à l'ordre partiel du test d'intégration, Labiche utilise un système de notation pour chaque étape du test d'intégration :

i. Dans un triplet du test d'intégration T, on ne garde que l'ensemble T.besoin.

ii. Les classes de T.cible sont marquées « # ».

iii. Une classe qui joue un rôle de parent et qui n'est pas instanciée est entourée par des parenthèses « ( ) ».

iv. Une classe qui joue un rôle à la fois de parent et de serveur est notée avec un plus « + » si T.type = Sta et avec une étoile « * » si T.type = Dyn.

La Figure 32 présente l'ordre partiel dans la Figure 31 avec ce système de notation. Analysons l'étape (B, C+, E, F#, F_G, G) de la ligne 4. La classe sous test de cette étape est la classe F (F#). La classe C est notée avec un signe «+ » car cette étape est une étape de test statique, la classe C est l'ancêtre de la classe F et le serveur de la classe B.

Pour l'étape (A#, B#, C*, D#, D_A, E#*, F#, F_G, G#, H#) de la ligne 7, les classes sous test sont A, B, D, E, F, G et H (A#, B#, D#, E#*, F#, G#, H#). La classe C est

notée avec une étoile «* » car elle joue des rôles comme dans l'étape (B, C+, E, F#, F_G, G) mais l'étape (A#, B#, C*, D#, D_A, E#*, F#, F_G, G#, H#) est une étape de test dynamique. La classe E est une des classes sous test, l'ancêtre de la classe H et un des serveurs de la classe A : elle est donc notée avec un dièse « # » et une étoile « * ».

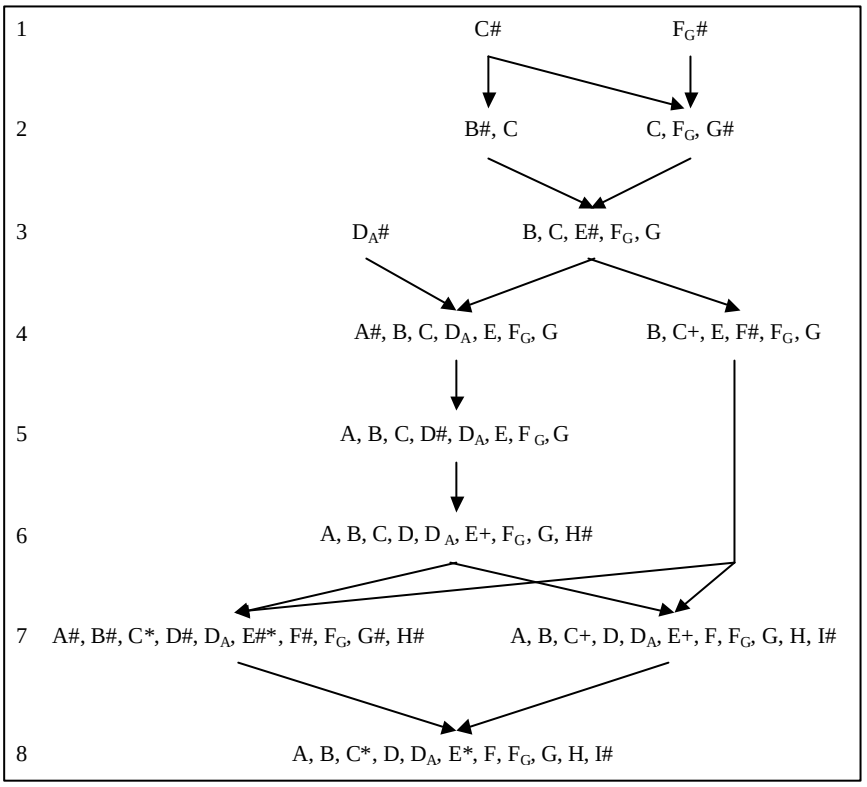

Figure 32. Labiche : le nouveau système de notation pour le graphe partiel dans la Figure 31.

5.d. Prise en compte de l'abstraction

Supposons que les deux classes dérivées (les classes C et E) sont des classes abstraites. L'intégration indépendante de ces classes abstraites devient impossible car elles ne sont pas instanciables. Il faut donc les grouper avec au moins une de leurs classes dérivées. Par conséquent, l'ordre d'intégration va changer.

Dans le graphe, l'ordre du test d'intégration est changé en utilisant les règles suivantes :

i. À cause de l'abstraction, les classes abstraites ne sont pas instanciables donc les étapes correspondantes dans le graphe partiel de l'ordre du test d'intégration sont supprimées en reliant respectivement leurs prédécesseurs à leurs successeurs. L'intégration de ces classes abstraites est attribuée à l'étape faisable la plus proche.

ii. Si une classe abstraite joue un rôle à la fois de serveur et de parent, elle perd son rôle de serveur donc elle est entourée par des parenthèses. Son rôle de serveur est dédié aux classes qui réalisent les interfaces de cette classe abstraite.

On présente dans l'Annexe I, « Prise en compte de l'abstraction dans l'approche de Yvan Labiche », les étapes d'application des règles dont le résultat est donné dans la Figure 33.

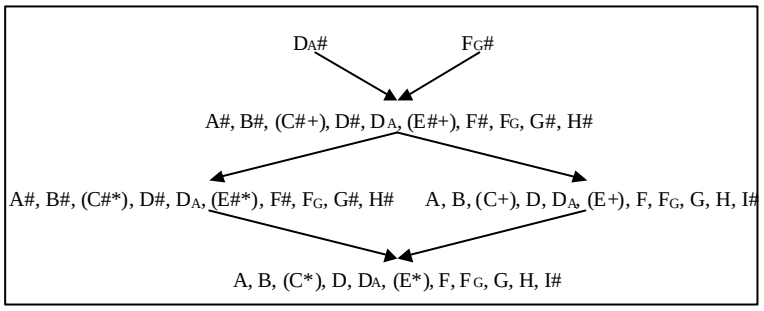

Figure 33. Labiche : prise en compte l'abstraction pour le graphe partiel dans la Figure 32.

5.e. Résumé

Labiche a essayé d'ajouter le traitement dynamique à la planification du test d'intégration. Il a ajouté à l'ORG les arcs, dus au polymorphisme. Il regroupe le test d'intégration d'une classe abstraite avec le test d'intégration de ses implémentations. Pour le test d'intégration de chaque classe, il propose un système de notation qui permet de désigner toutes les classes nécessaires.

5.f. Commentaire

Le fait que l'abstraction ne soit prise en compte qu'au dernier moment force parfois l'intégration de plusieurs unités en même temps (voir la Figure 33). Ici, la difficulté de l'intégration Big-Bang réapparaît. De plus, ce travail ne permet pas de traiter les interdépendances.

6. La solution de Pampa

En 1999 et en 2000, l'équipe Pampa (IRISA) a publié sa solution [Jéron 1999 et LeTraon 2000] qui se compose aussi de trois parties : la modélisation, la décomposition et la parallélisation. La modélisation de Pampa prend un diagramme de classes UML en entrée. Pampa appelle la structure de dépendances obtenue « *le graphe de dépendances de test* » (Test Dependency Graph - GDT). La décomposition et la parallélisation sont faites sur ce GDT.

6.a. Le graphe de dépendances de test

La modélisation de Pampa se base sur un diagramme de classes UML. Un nœud dans un GDT modélise une classe ou (si le niveau de détail du diagramme de classes UML d'entrée en permet) une méthode (voir la Figure 34).

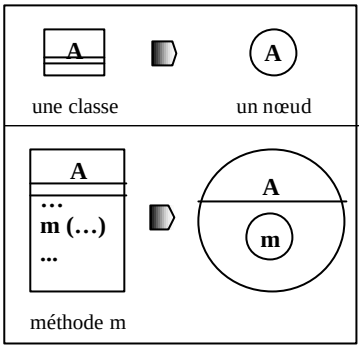

Figure 34. Pampa : des nœuds dans un GDT

La relation entre les unités est classifiée en trois catégories : classe-classe, méthode-classe et méthode-méthode. Les règles de modélisation de ces trois types de relations sont données dans la Figure 35 et la Figure 36. Notons que Pampa ne différencie pas les types de relation entre deux classes comme Kung.

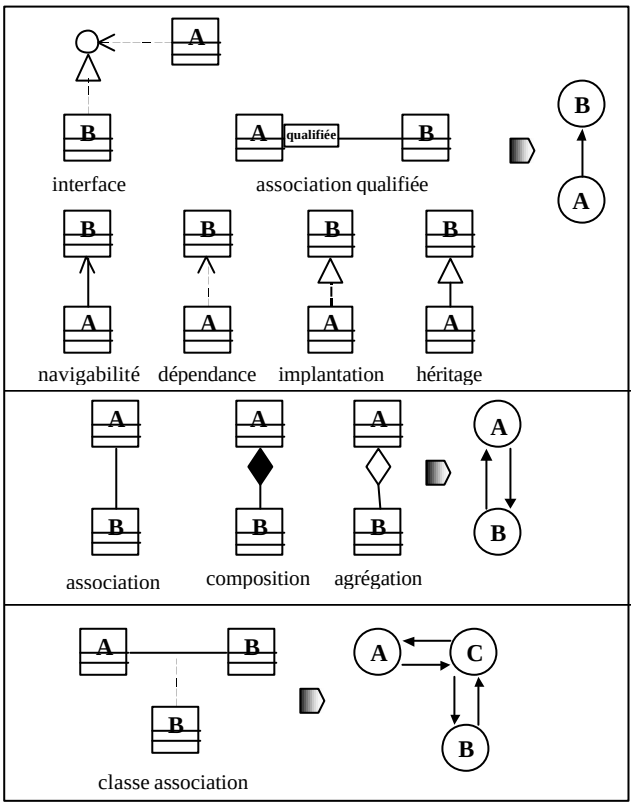

Figure 35. Pampa : la modélisation de la relation classe-classe.

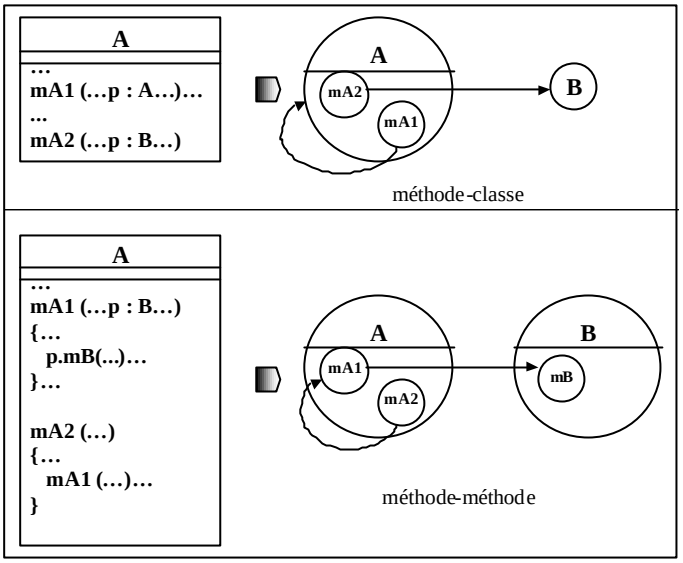

Figure 36. Pampa : la modélisation de la relation méthode-méthode et méthode-classe.

La Figure 37 donne un exemple de la modélisation de Pampa. Après avoir modélisé les méthodes et les classes, les nœuds qui modélisent les méthodes sont encadrés par les nœuds des classes correspondants donc les algorithmes de graphe ne peuvent pas y appliquer. Il faut alors transformer le GDT résultat en un GDT dont les nœuds sont totalement séparés.

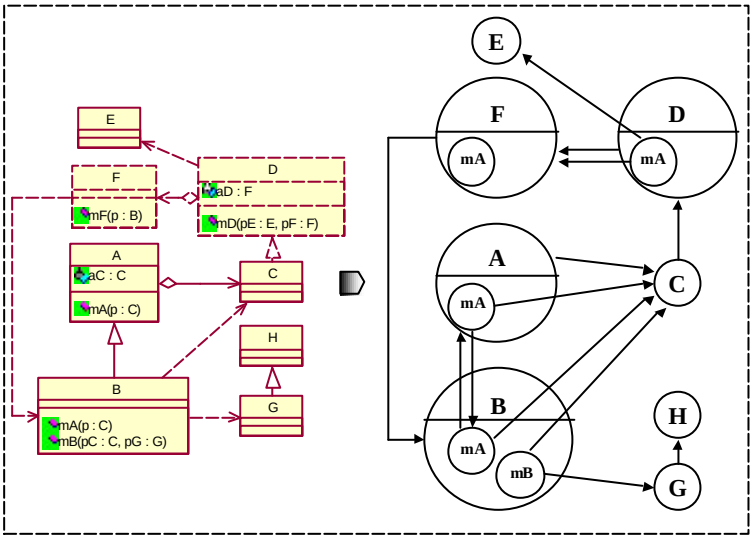

Figure 37. Pampa : un exemple de modélisation.

Il y a deux façons pour décomposer les nœuds encadrés : soit on supprime les nœuds et les relations aux méthodes (ce qui rend inutile la modélisation des méthode) ; soit on considère qu'une classe dépend de toutes ses méthodes et on ajoute cette dépendance dans le GDT (voir la Figure 38). Après avoir décomposer des nœuds encadrés, Pampa applique l'algorithme de Tarjan pour déterminer les CFCs et pour obtenir le poids de chaque nœuds dans les CFCs.

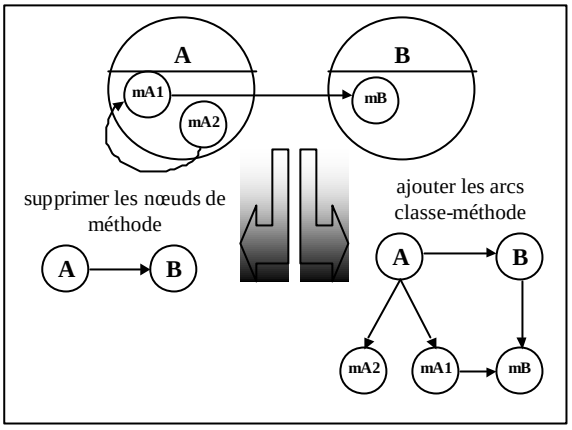

Figure 38. Pampa : Standardisation du GDT.

6.b. La décomposition des CFCs avec le nombre des « fronts ».

Au début, l'algorithme de Tarjan pour déterminer les CFCs différencie trois types d'arcs dans une CFC : les branches (« *tree arcs* »), les arcs croisés (« *cross arcs* ») et les arcs de retour (« *front arcs* »).

Par exemple, la Figure 39 présente ces trois types d'arcs en utilisant l'algorithme de Tarjan avec l'ordre d'accès alphabétique, c'est-à-dire, quand on accède aux successeurs d'un nœuds, ces successeurs sont ordonnés en ordre alphabétique. Par exemple, les successeur de l'unité F sont ordonnés comme suit : C, E G. Notons que le type d'un arc change quand l'ordre d'accès change.

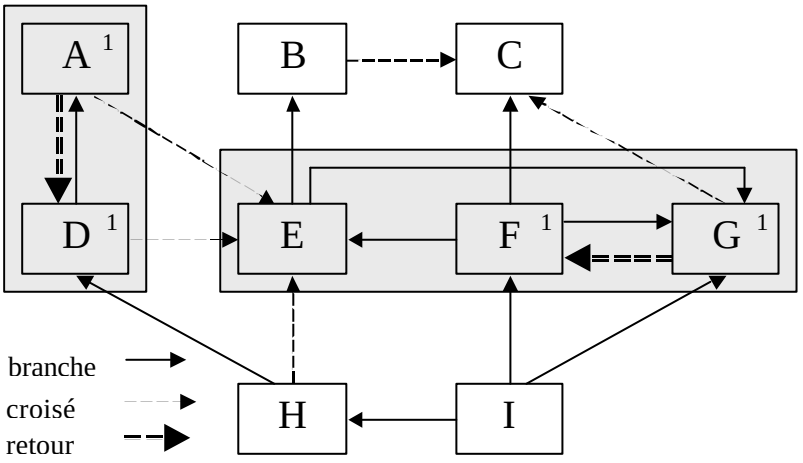

Figure 39. Pampa : les trois types d'arcs de l'algorithme de Tarjan

i. les branches (« *tree arcs* ») : une branche connecte un nœud visité à un nœuds non-visité en utilisant l'algorithme de Tarjan. Les branches créent un arbre (ou une forêt) couvrant du GDT.

ii. les arcs croisés (« *cross arcs* ») : un arc croisé connecte un nœud visité à un autre nœud visité d'une autre branche. Par exemple, les arcs BC, GC, HE, AE et DE dans la Figure 39.

iii. les arcs de retour (« *front arcs* ») : un arc retour ferme un cycle comme les arcs GF et AD dans la Figure 39.

Pour décomposer les CFCs, la solution préliminaire de Pampa propose de simuler le nœud qui a le nombre maximum d'arcs de retour en considérant que ce nombre représente le nombre de cycles où ce nœud participe.

Par exemple, en ordre d'accès alphabétique, le nombre d'arcs de retour est présenté en haut à droit des nœuds A, D, F et G de la Figure 39. Les nœuds A (ou D) et F (ou G) sont simulés, les arcs DA (AD) et GF (FG) sont cassés. Il faut donc deux bouchons réalistes.

6.c. Parallélisation

L'équipe Pampa a proposé le premier travail sur la parallélisation. Il estime que le nombre d'étapes d'intégration minimum (`#étapes_min`) ne descend pas sous la limite :

$$\#\text{étapes\_min} \geq \max(A, B)$$

ou

A = `#nœuds div #testeurs + 1`:

`#nœuds` : le nombre total de nœuds dans le GDT

`div` : opérateur de division des entiers

`#testeurs` : le nombre de testeurs qui effectuent le test

L : est la longueur maximale des chemins d'une racine à une feuille

L'équipe Pampa attribue pour chaque nœud ***n*** du GDT un poids ***P(n)*** qui est la longueur maximale des chemins d'une racine à ce nœud. Ce travail est effectué par un algorithme de recherche en profondeur :

$$P(n)=\begin{cases} 1 & \Leftarrow PS(n)=f \\ \max\langle P(m)\rangle+1, m\in PS(n) & \Leftarrow PS(n\neq f) \end{cases}$$

où $PS(n)$ est l'ensemble de prédécesseurs de n :

$$PS(n) \leftarrow \{m \in N \mid \exists(m, n) \in A\}$$

L'ordonnancement va suivre l'ordre décroissant des poids. La Figure 40 présente des poids pour les nœuds dans la Figure 39 et le Tableau 11 présente l'ordre d'intégration correspondant avec deux testeurs.

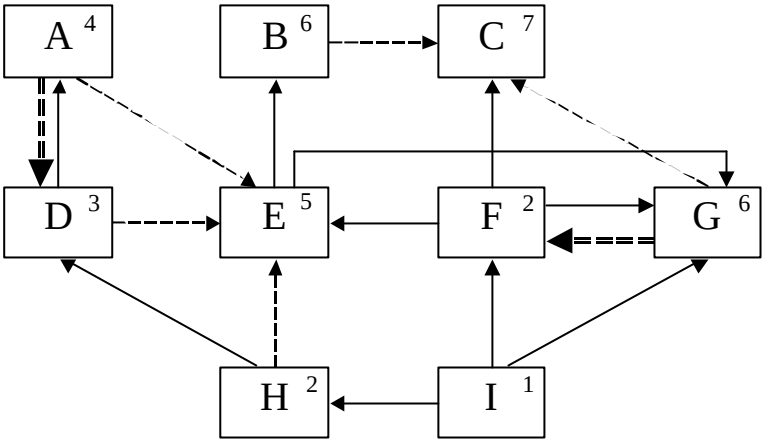

Figure 40. Pampa : Le poids des nœuds.

Tableau 11. Pampa : Ordre de test avec deux testeurs.

Etape	1er testeur	2nb testeur
1	C	
2	B	G
3	E	
4	A	F
5	D	
6	H	
7	I	

6.d. Résumé

La solution initiale de l'équipe est une solution « complète » qui traite toutes les trois parties d'une stratégie de planification des test d'intégration : la modélisation, la décomposition et la parallélisation.

La modélisation est basé sur le diagramme de classes UML. Cette modélisation considère une classe ainsi qu'une méthode comme une unité si le niveau de détail du diagramme de classes UML le permet. Cette modélisation ne traite pas les relations dynamiques, les classes génériques ainsi que les interfaces.

La décomposition attribue à chaque nœud un poids qui est le nombre d'arcs retour. Cette procédure utilise l'algorithme de Tarjan qui permet déterminer efficacement les CFCs

La parallélisation se base sur la longueur des branches du GDT : La feuille de la branche la plus longue est testée d'abord.

6.e. Commentaire

La modélisation de Pampa a choisi un bon point de départ : cette modélisation utilise le diagramme de classes UML comme entrée, ce qui permet d'obtenir un plan d'intégration avant la phase de développement. Cependant, on doit ajouter dans cette modélisation le traitement dynamique et compléter le traitement statique (les classes génériques).

La décomposition de Pampa n'est pas stable. Avec les différents ordres d'accès, le nombre d'arcs de retour change. Il faut donc trouver un autre type de poids. Un autre problème de cette décomposition est le suivant : quand on parle du nombre d'arcs de retour, on ne précise pas le nœud simulé. Par exemple, pour la CFC E-F-G du GDT de la Figure 39, F et G ont le même nombre d'arcs de retours ; si on simule G, la CFC existe encore.

La parallélisation de Pampa est simple, mais Pampa, comme Kung, ne prévoient pas les étapes de retest.

7. Résumé

Dans ce chapitre, nous avons présenté les recherches sur le test d'intégration. Parmi ces travaux, la recherche de Kung [Kung 1995 #1, Kung 1995 #2 et Kung 1996] est le seul travail « complet », car il comporte les trois étapes du test d'intégration (modélisation, décomposition des groupes interdépendants et planification). Cependant plusieurs points dans sa solution doivent être améliorés, en particulier l'algorithmique pour identifier et décomposer les interdépendances..

Les autres travaux présentés dans ce chapitre se concentrent sur l'amélioration d'un ou plusieurs points du travail de Kung. Cependant, il n'existe pas de stratégie complète, qui assemble tous les avantages et qui évite tous les inconvénients des recherches existantes.

Dans la recherche de Kung, on peut utiliser la relation entre le niveau de risque d'utilisation de bouchons et la nature de la dépendance simulée. La modélisation de Kung est basée sur le code du système, qui n'existe pas encore avant la phase d'implémentation dans le cycle de vie d'un logiciel. Par conséquent, cette modélisation convient plus à la maintenance qu'au test d'intégration. Pour le test d'intégration, une modélisation à partir d'un modèle de conception est plus pertinente. La décomposition des groupes interdépendants de Kung n'est pas efficace en minimisation du nombre de bouchons. Il faut améliorer les critères de sélection de bouchon. La planification de Kung ne permet pas non plus la parallélisation. Il faut avoir une autre organisation.

Le travail de Tai-Daniels [Tai-Daniels 1999] nous présente la solution exacte pour couvrir l'utilisation des bouchons. Par contre, leur solution pour la décomposition des groupes interdépendants n'est pas efficace pour minimiser le nombre de bouchons. Nous retiendrons de ce travail surtout l'introduction d'étapes de retest.

Le travail de Briand [Briand 2001] porte seulement sur la décomposition des interdépendances. Le résultat obtenu est meilleur que ceux de Tai-Daniels ou de Kung, mais on peut augmenter l'efficacité de Briand (voir le Chapitre VI, « Expérimentations et étude comparative des différentes stratégies »).

Le travail de Labiche [Labiche 2000 #1 et Labiche 2000 #2] nous montre les inconvénients du travail de Kung et nous propose une façon de traiter le polymorphisme et l'abstraction. Cependant, pour éviter les groupes interdépendants qui contiennent des dépendances implicites apparaissant à cause du polymorphisme et de l'abstraction, on devrait appliquer l'approche proposée avant (pas après) la décomposition des groupes interdépendants.

La solution préliminaire d'IRISA nous présente un bon algorithme pour déterminer les CFCs et pour paralléliser les tests ainsi qu'une bonne entrée pour la modélisation. C'est sur cette solution, je construits ma stratégie de planification des test d'intégration.

Chapitre III.

Modélisation de la structure de dépendances

Puisqu'une stratégie de test d'intégration est un ordonnancement et que le problème d'ordonnancement est bien résolu en utilisant la théorie des graphes, nous allons modéliser la structure de dépendances du système sous test par un graphe.

UML est un langage graphique qui permet de représenter la structure et le comportement d'un système à objets. Ce langage graphique est très répandu et devient un moyen standard pour représenter un système informatique. Pour profiter de ce modèle, nous l'utilisons comme une source pour extraire les dépendances d'intégration.

Dans ce chapitre, nous présentons notre ensemble de règles de modélisation pour obtenir un graphe qui décrit les dépendances du système sous test, à partir d'un modèle UML.

Dans le paragraphe 1, nous définirons le graphe qui modélise la structure de dépendances du système sous test. Les nœuds de ce graphe modélisent les classes. Les arcs de ce graphe représentent les dépendances entre les classes du système sous test.

Les règles de modélisation de dépendance à partir d'un modèle UML sont présentées dans le paragraphe 2. Le paragraphe 3 présente dans le détail une façon de simplifier le graphe obtenu par élimination des arcs/dépendances redondant(e)s.

Nous améliorons la modélisation dans le paragraphe 4 pour obtenir plus d'informations si le niveau de granularité de conception le permet. Cette amélioration permet d'extraire les dépendances entre méthodes, et aussi entre méthodes et classes. Cette amélioration nous permet de préciser exactement à quel moment une méthode doit être testée.

Un exemple est présenté dans le paragraphe 5. Le diagramme de classes UML de cet exemple est celui de la bibliothèque de graphe de notre outil TestPlan qui est conçu pour mettre en application nos idées.

1. Graphe de dépendances de test

Une solution pour le test d'intégration est un ordonnancement qui est bien résolu par un modèle de graphe. Pour profiter des résultats de théorie des graphes, nous

utilisons un graphe orienté, nommé le « *graphe de dépendances de test* » - GDT (« Test Dependency Graph ») pour modéliser les dépendances d'intégration.

En profitant du résultat de recherche de Kung sur la relation entre les risques d'utilisation des bouchons et la nature des dépendances, ce GDT est un graphe dont les arcs sont étiquetés par des informations sur la catégorie des dépendances.

Définition 21. *Un graphe de dépendances de test* (*GDT*) *d'un programme à objets P est un graphe orienté, étiqueté et connexe GDT* := (*N, E, A*) *où* :

- *N* : *l'ensemble fini des nœuds, n* Î *N représente une unité sous test.*
- *E* = {*H, C, A*} : *l'ensemble d'étiquettes, représente le type de dépendance entre les nœuds : « H » pour Héritage, « C » pour Composition et « A » pour Association.*
- *A* Í *N* × *N* × *E* : *l'ensemble d'arcs orientés, a* (*p, s, e*) Î *A représente une dépendance directe de test avec le type e, entre le prédécesseur p et le successeur s. Entre chaque couple de nœuds x et y, il y a au plus un arc de x à y et il y a au plus un arc de y à x.*

2. Transformation d'un diagramme de classes UML en graphe de dépendances de test

Nous donnons dans cette section les principales règles de transformation d'un diagramme de classes UML vers un GDT. Les notations UML utilisées dans cette partie sont les notations standards d'UML 1.4 [UML 2001]. On suppose que le diagramme est un diagramme de conception (détaillé et proche de l'implémentation). En effet, il serait illusoire d'espérer faire un plan d'intégration à partir d'un modèle peu raffiné (d'analyse par exemple).

2.a. Unité de test

Avec la modularité, la classe constitue l'unité par excellence dans un système à objets. Chaque classe est modélisée par un nœud dans le GDT (voir la Figure 41).

Modélisation 1. *Dans le GDT, une classe est modélisée par un nœud.*

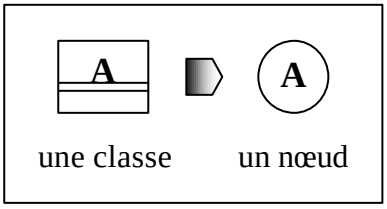

Figure 41. Modélisation de la classe.

2.b. Dépendance de type Héritage/Généralisation

Une classe concrète A, dérivée directe d'une classe concrète B, hérite des méthodes et attributs de la classe B. Par conséquent, l'intégration de la classe dérivée A demande que les méthodes et les attributs hérités existent. Autrement dit, la classe A dépend de la classe B pour le test d'intégration (voir la Figure 42).

Modélisation 2. *Dans le GDT, une dépendance d'héritage dans un diagramme de classes UML entre une classe concrète « A » et son ancêtre direct concret « B » est modélisée par un arc, étiqueté par « H », du nœud A vers le nœud B.*

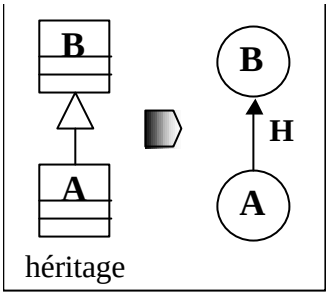

Figure 42. Modélisation de l'héritage.

On considérera le cas de l'héritage de classes abstraites en 2.g, « Polymorphisme/Utilisation d'une interface/Abstraction ».

2.c. Dépendance de type Composition/Agrégation

L'intégration d'une classe A qui est un composé/agrégat d'objets de classe B implique l'existence des services de B. Donc, la classe A dépend de la classe B pour le test d'intégration. Normalement, une association de type composition/agrégation est représentée avec une navigabilité explicite. Cette relation est modélisée par un arc unique entre deux nœuds (voir la Figure 45).

Modélisation 3. *Dans le GDT, une dépendance de composition/agrégation avec la navigabilité explicite, dans un diagramme de classes UML entre une classe « A » et une classe « B » est modélisée par un arc, étiqueté par « C », du nœud A vers le nœud B.*

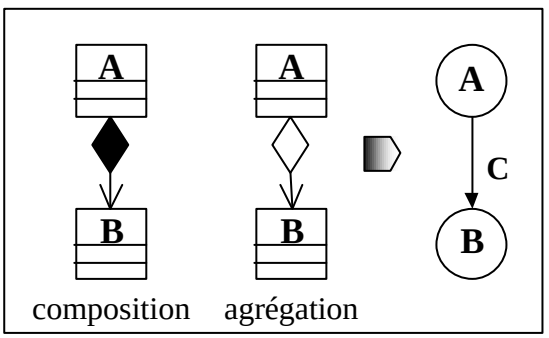

Figure 43. Modélisation de la composition et de l'agrégation – navigabilité explicite.

Si la navigabilité d'une relation composition/agrégation est implicite, cette relation est modélisée par deux arcs entre deux nœuds (voir la Figure 44), ce qui correspond à une modélisation prudente. En effet, la relation est rarement implanté de manière bidirectionnelle. On choisit donc de modéliser plus de dépendances que de risquer d'en omettre.

Modélisation 4. *Dans le GDT, une dépendance de composition/agrégation avec la navigabilité implicite, dans un diagramme de classes UML entre une classe « A » et une classe « B » est modélisée par un arc, étiqueté par « C », du nœud A vers le nœud B et un arc, étiqueté par « A », du nœud B vers le nœud A.*

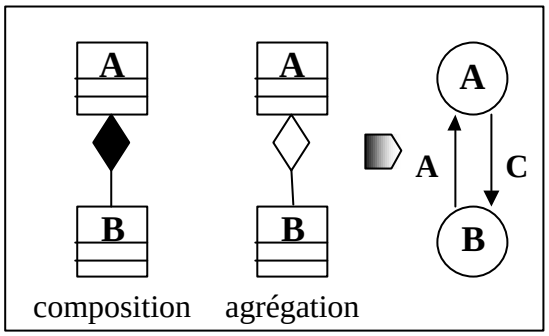

Figure 44. Modélisation de la composition et de l'agrégation – navigabilité implicite.

2.d. Dépendance de type Association

(i) Associations binaires

La modélisation d'une relation d'association dépend de la navigabilité. La navigabilité indique que la classe navigante utilise un (ou plusieurs) objet(s) de la classe naviguée. Par conséquent, l'intégration de la classe navigante exige que la classe naviguée existe. La classe navigante dépend donc de la classe naviguée pour le test d'intégration. La navigabilité peut être décrite implicitement ou explicitement comme le montre la Figure 45.

Modélisation 5. *Dans le GDT, une dépendance d'association dans un diagramme de classes UML, navigable explicitement d'une classe «A » vers une classe « B », est modélisée par un arc, étiqueté par « A », du nœud A vers le nœud B.*

Modélisation 6. *Dans le GDT, une dépendance d'association dans un diagramme de classes UML, navigable implicitement entre deux classes « A » et « B », est modélisée par un arc, étiqueté par « A », du nœud A vers le nœud B et un arc, étiqueté par « A », du nœud B vers le nœud A.*

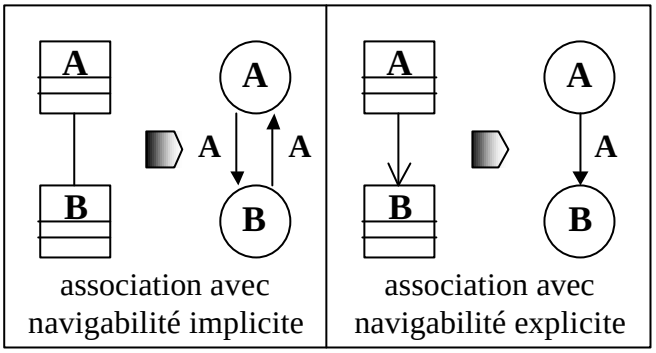

Figure 45. Modélisation de l'association.

(ii) Association n-aires et classe d'association

Nous ne traitons pas les associations n-aires et les classes d'association. A notre avis, on ne peut pas implémenter directement ces associations et ces classes. On doit les transformer en associations binaires et en classes ordinaires. Il y a plusieurs façon pour les transformer qu'on peut trouver dans [Fowler 1999] ou dans [Muller 1997]. Après la transformation, la modélisation de ces dépendances revient à des modélisations d'associations binaires. De plus, ce type de représentation ne devrait plus apparaître sur un diagramme de conception détaillée dont nous supposons disposer.

2.e. Dépendance transitoire – « *dependency* »

Pour une stratégie d'intégration progressive, l'intégration d'une classe A qui dépend de manière transitoire (« *dependency* ») d'une classe B implique que la classe B soit présente. Donc, la classe A dépend de la classe B pour le test d'intégration (voir la Figure 46).

Modélisation 7. *Dans le GDT, une dépendance transitoire (« dependency ») dans un diagramme de classes UML entre une classe A et une classe B est modélisée par un arc, étiqueté par « A », du nœud A vers le nœud B.*

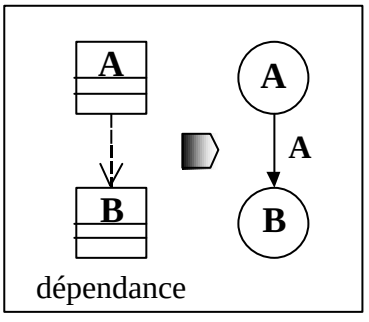

Figure 46. Modélisation de la dépendance transitoire (« *dependency* »).

2.f. Classe générique/classe attachée

Une classe générique ne demande aucun service à la classe attachée (« bound class ») mais l'intégration de la classe générique exige que la classe attachée soit présente [Overbeck 1994]. On peut donc ordonner l'intégration de ces deux classes par un arc dans un GDT (voir la Figure 47).

Dans un diagramme de classes UML, on a trouvé deux façons de présentation de la relation entre une classe générique et une classe attachée : soit elle est implicite soit elle est explicite [UML 2001].

Modélisation 8. *Dans le GDT, la relation dans un diagramme de classes UML entre une classe générique A et sa classe attachée B est modélisée par un arc, étiqueté par « A », du nœud A vers le nœud B.*

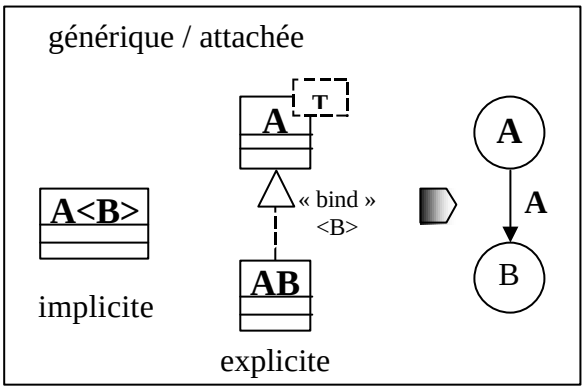

Figure 47. Modélisation de la relation entre la classe générique et la classe attachée.

2.g. Polymorphisme/Utilisation d'une interface/Abstraction

Le polymorphisme introduit des dépendances implicites qu'on doit prendre en compte (voir la Figure 48), après la construction d'un GDT préliminaire à l'aide des règles ci-dessus.

Modélisation 9. *Dans le GDT, la dépendance implicite due au polymorphisme entre une classe A qui dépend de la classe B, et la classe C, héritière de la classe B, est modélisée par un arc supplémentaire du nœud A vers le nœud C. Cet arc a la même étiquette que l'arc entre le nœud A et le nœud B.*

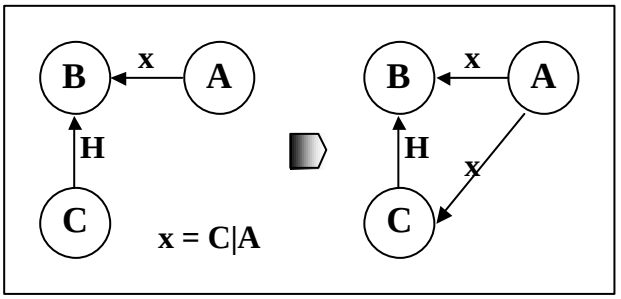

Figure 48. Modélisation : le traitement du polymorphisme.

Un cas spécial du polymorphisme et de l'abstraction est l'utilisation d'une interface. Cette utilisation est modélisée comme pour le polymorphisme : la classe d'interface est ensuite supprimée du GDT, ainsi que ses relations (voir la Figure 49).

Modélisation 10. *Dans le GDT, la dépendance implicite due au polymorphisme dans un diagramme de classes UML entre une classe A qui dépend d'une interface et une classe B qui implémente cette interface est modélisée par un arc, étiqueté par « A », du nœud A vers le nœud B.*

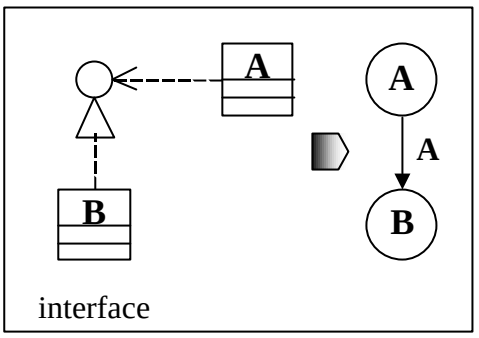

Figure 49. Modélisation : le traitement de l'utilisation d'une interface.

Les classes abstraites, comme les interfaces, sont des classes qu'on ne peut pas intégrer isolément. Par conséquent, il faut supprimer les nœuds qui modélisent ces classes et grouper le test d'intégration de ces classes avec toutes les classes qui implémentent ces classes abstraites. Pour garder les dépendances des classes abstraites, ces dépendances sont récupérées par les implémentations comme montre la Figure 50. Dans cette figure, on utilise les nœuds fictifs des classes A, B, C et D (marqué par un bord pointillé). Après avoir récupéré les dépendances des classes abstraites, ces nœuds fictifs sont fusionnés comme montre la Figure 51.

Modélisation 11. *Dans le GDT, une dépendance entre une classe abstraite et une autre classe est récupérée par toutes les implémentations de la classe abstraite.*

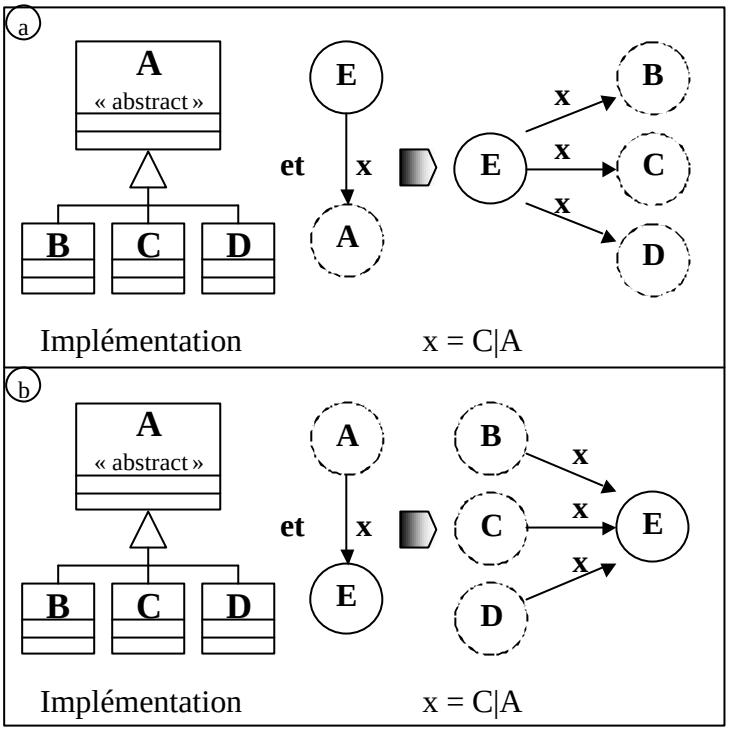

Figure 50. Modélisation : récupération des dépendances d'une classe abstraite.

Modélisation 12. *Dans le GDT, le nœud, modélisant une classe abstraite, est groupé avec tous les nœuds, modélisant les classes qui implémentent cette classe abstraite.*

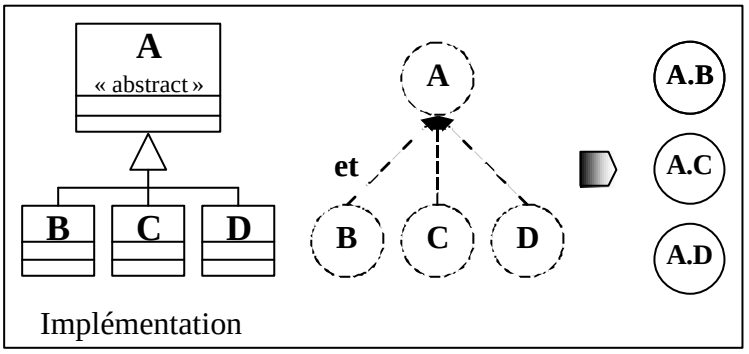

Figure 51. Modélisation : regroupement d'une classe abstraite avec ses implémentations.

Notons que pour un héritage d'une classe concrète, on ne doit pas regrouper cette classe avec les classes de sa descendance. On ne doit donc pas non plus récupérer ses dépendances.

3. Réduction des arcs – le problème des doublons

Par la modélisation, entre une classe A et une classe B peuvent exister plusieurs arcs. On peut considérer que chaque arc représente un appel à une méthode ou un attribut du serveur.

La modélisation a pour but de faciliter la décomposition des interdépendances (qui sont des CFCs du GDT) et la planification de test. Pour décomposer une CFC, il faut enlever quelques arcs de la CFC et simuler l'unité successeur de cet arc par un bouchon.

En terme de coût de création des bouchons (voir le paragraphe 2.b du Chapitre IV, « Décomposition des composantes fortement connexes »), les arcs qui entrent dans un nœud peuvent représenter le nombre de bouchons spécifiques ou le nombre de méthodes et d'attributs d'un bouchon réaliste, éventuellement créé(s) pour décomposer une CFC. L'élimination des doublons fait perdre cette information.

Pour simplifier la planification du test d'intégration, on considère qu'un bouchon spécifique simule tous les services que son unité d'intégration originelle peut offrir à un client d'un type donné. Dans ce cas, tous les arcs d'un nœud A vers un nœud B se regroupent éventuellement dans un seul bouchon si on doit simuler les services, représentés par ces arcs. Par conséquent, on peut éliminer les doublons pour le calcul des bouchons

En termes de risque d'utilisations de bouchons (voir le paragraphe 2.c du Chapitre IV, « Décomposition des composantes fortement connexes »), plus nombreux sont les arcs entre un nœud et un bouchon, plus le nombre d'utilisations est grand, et finalement plus le risque pris en introduisant des bouchons est grand. L'élimination des doublons fait perdre l'estimation sur le risque d'utilisations des bouchons.

L'existence, par exemple, de deux associations de A vers B signifie que A utilise B selon deux rôles différents. On pourrait penser que le nombre de bouchons (spécifiques) est de deux également : un par rôle que B joue vis à vis de A. Cependant, nous ne tiendrons pas compte des doublons car nous considérons que le nombre de rôles ne révèle ni le type, ni le nombre d'appels de A vers B.

Lors d'un choix entre plusieurs arcs équivalents, on choisira d'abord un arc simple pour créer un simulateur, plutôt qu'un arc qui correspondait, dans le modèle initial, à un doublon. On pourrait donc affiner l'approche en tenant compte d'une telle pondération.

Pour la planification des tests, les doublons d'un nœud A vers un nœud B ont une même signification : le nœud B ne doit pas être intégré avant le nœud A donc on a une autre raison pour éliminer des doublons, qui n'ajoutent pas d'informations primordiales.

Pour éliminer les arcs doublons, on va garder les arcs suivant l'ordre de leur étiquette : H, C, A (voir la Figure 52), c'est-à-dire, dans l'ordre décroissant de la force du lien qui lie deux unités A et B. On ne mémorise pas le nombre d'arcs de chaque type.

Modélisation 13. *Dans le GDT, s'il y a plusieurs arcs d'un nœud A à un nœud B, on va garder un seul arc. Les règles d'élimination sont* :

i. S'il existe un arc, étiqueté par H, tous les autres arcs sont supprimés.

ii. Sinon et s'il existe un arc étiqueté par C, tous les autres arcs sont supprimés.

iii. Sinon et s'il existe un arc étiqueté par A, tous les autres arcs sont supprimés.

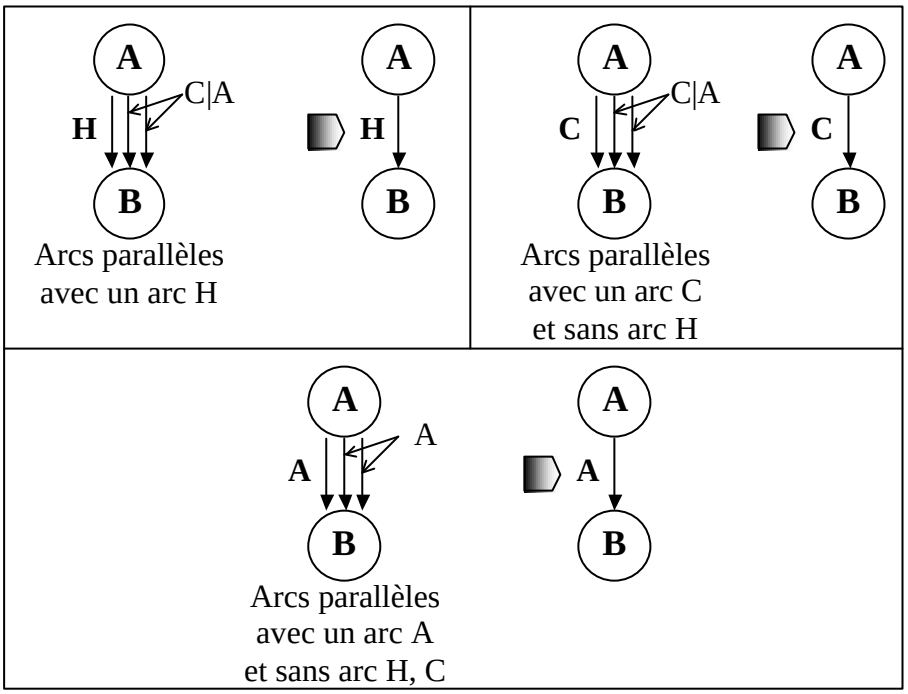

Figure 52. Modélisation : l'élimination d'arcs parallèles.

Un autre cas où un arc doit être éliminé est celui des arcs auto-dépendants dont le prédécesseur et le successeur est le même (voir la Figure 53). Ces arcs ne participent ni à la planification, ni à la décomposition de CFCs.

Définition 22. *Un arc auto-dépendant est un arc qui a le même nœud comme prédécesseur et comme successeur.*

Modélisation 14. *Dans le GDT, les arcs auto-dépendants sont éliminés.*

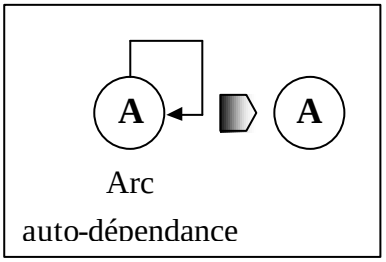

Figure 53. Modélisation : l'élimination d'arcs auto-dépendants.

3.a. Les cardinalités sur les associations/agrégations/compositions

Nous avons choisi, dans un premier temps, de ne pas tenir compte de la cardinalité des relations. En effet, l'information que nous souhaitons capturer est celle de dépendance d'une classe vis-à-vis d'une autre.

On peut – dans une approche où l'on tient compte de la complexité des bouchons – considérer cette cardinalité pour donner un poids aux arcs (d'autant plus grand que la cardinalité est élevée). Toutefois, nous considérons que ce facteur a une influence négligeable comparé à la nature des relations (héritage / composition / agrégation/ association) ou à l'existence de doublons qui définissent des relations client-serveur distinctes donc pouvant nécessiter la création de bouchons distincts.

4. Amélioration

Dans un modèle UML, à côté des classes, on peut obtenir un modèle plus détaillé en utilisant la méthode comme unité. Cette utilisation n'est possible que si le niveau de granularité du modèle le permet (par exemple, un modèle issu de la rétro-ingénierie). Dans ce cas, le niveau de précision d'une stratégie de test d'intégration dépend du choix entre une intégration restreinte aux classes ou bien allant jusqu'aux méthodes.

L'utilisation d'une méthode comme unité de test facilite la création des bouchons et diminue les risques d'utilisation parce que la simulation d'une méthode est beaucoup moins coûteuse que la simulation d'une classe. Un autre point fort de la modélisation des méthodes est que les CFCs qui existent au niveau de la modélisation de classes peuvent disparaître. En effet, si A a une association vers B et vice-versa, il se peut que seules certaines méthodes de A utilisent B et que l'utilisation que B a de A ne fasse pas appel à celles-ci. Le cycle (A B) disparaît alors, si on considère les dépendances au niveau de la méthode. Le point faible de la modélisation des méthodes est que le nombre d'unités dans le GDT est beaucoup plus grand et risque de produire des plans de test plus complexes. Il faut donc des algorithmes de complexité d'exécution raisonnable.

Pour profiter des avantages de la modélisation de méthodes, en plus des trois types de dépendance que Kung retient, nous proposons de tenir compte d'un autre type de dépendance, la dépendance sur une méthode.

4.a. Modélisation des méthodes

Chaque méthode dans une classe est considérée comme une unité et modélisée par un nœud dans le GDT. Une classe est considérée alors comme testée si toutes ses méthodes sont testées et intégrées. Avec cette considération, on peut dire qu'une classe dépend de toutes ses méthodes (voir la Figure 54).

Modélisation 15. *Dans le GDT, une méthode «m » dans une classe «A » est modélisée par un nœud « A.m » ; la dépendance entre la classe A et sa méthode m est modélisée par un arc, étiqueté par « M », du nœud A vers le nœud A.m.*

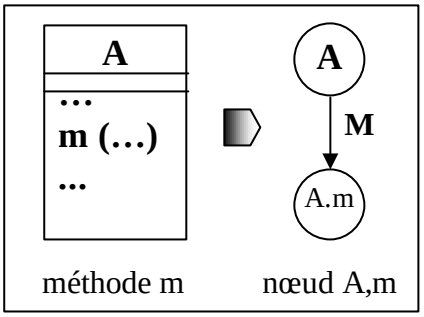

Figure 54. Modélisation de la relation classe vers méthode.

Avec ce type de dépendance, l'ensemble d'étiquettes du GDT est augmenté :

$$E = \{H, C, A, M\}$$

4.b. Paramètre des méthodes – la dépendance méthode-classe

Depuis un modèle UML, on peut extraire aussi les dépendances entre une méthode et une classe. Cette dépendance est représentée par une déclaration de la classe comme un paramètre formel d'une méthode (voir la Figure 55).

Modélisation 16. *Dans un diagramme de classes UML si une méthode « m »* (*d'une classe « X »*) *qui déclare un objet d'une classe « A » comme un paramètre formel, cette relation est modélisée dans le GDT par un arc, étiqueté par A, du nœud Xm vers le nœud A.*

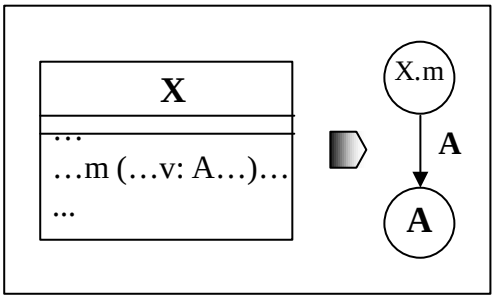

Figure 55. Modélisation de la dépendance méthode-classe.

En effet, dans un modèle UML, la dépendance entre une méthode et une classe est un cas spécial de la relation «dépendance transitoire » comme le montre la Figure 56. On retrouve donc une étiquette connue.

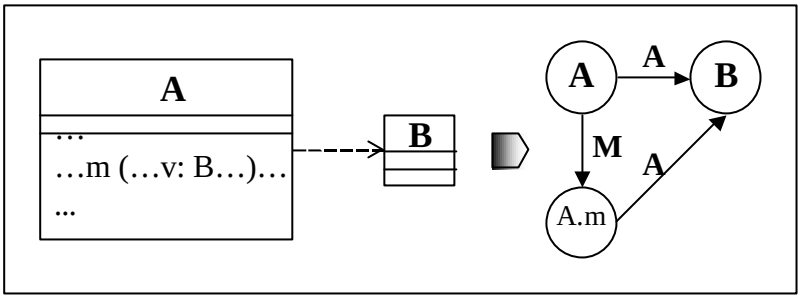

Figure 56. La modélisation d'une dépendance méthode-classe détaille une dépendance transitoire.

4.c. Extension du traitement du polymorphisme/de l'interface/des doublons aux méthodes

Le traitement du polymorphisme (voir la Modélisation 9) est étendu pour inclure la modélisation des dépendances impliquant des méthodes (voir la Figure 57). L'utilisation d'interface, ainsi que des doublons, subit le même traitement (voir la Figure 58).

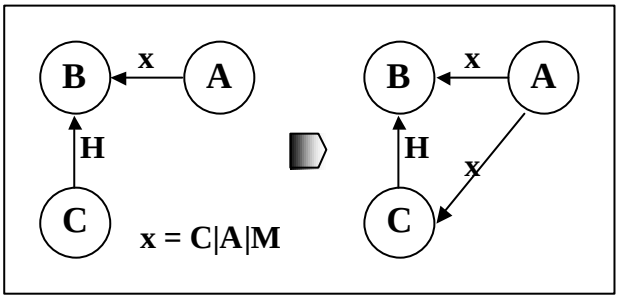

Figure 57. Extension du traitement du polymorphisme aux méthodes.

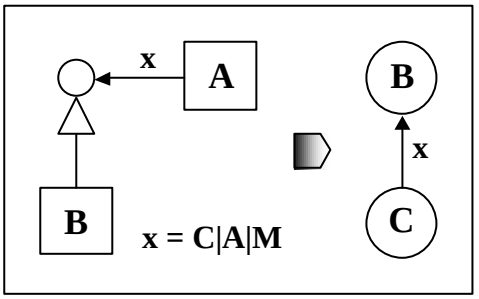

Figure 58. Extension du traitement de l'utilisation d'interface aux méthodes.

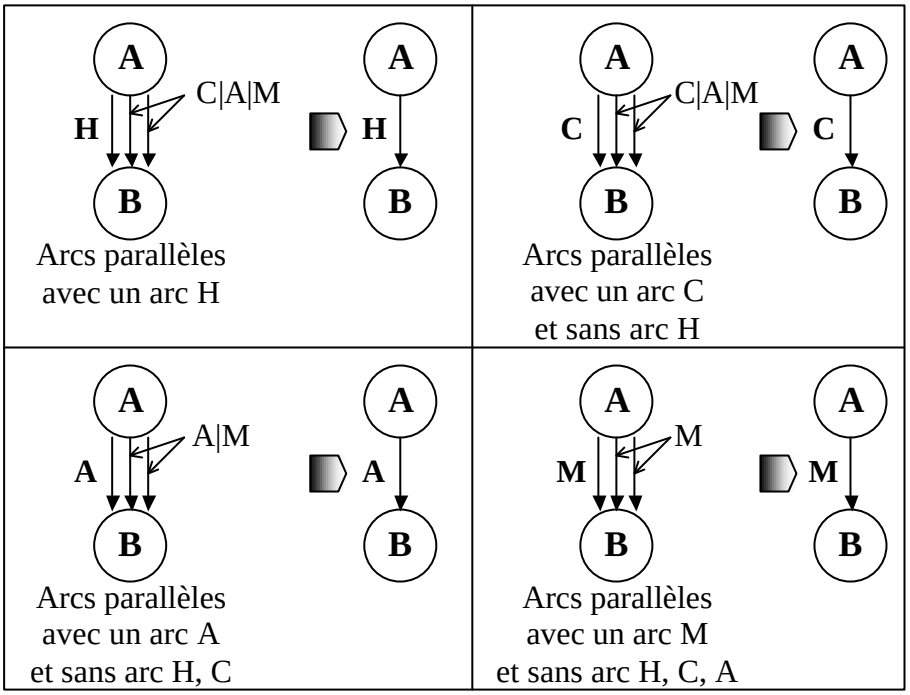

Figure 59. Extension de l'élimination des doublons aux méthodes.

5. Exemple

On va appliquer les règles de modélisation, présentées ci-dessus au diagramme de classes UML du package Graph de notre outil de test d'intégration, TestPlan. Le diagramme de classes de ce package est présenté dans la Figure 60. Le GDT correspondant est présenté dans la Figure 61.

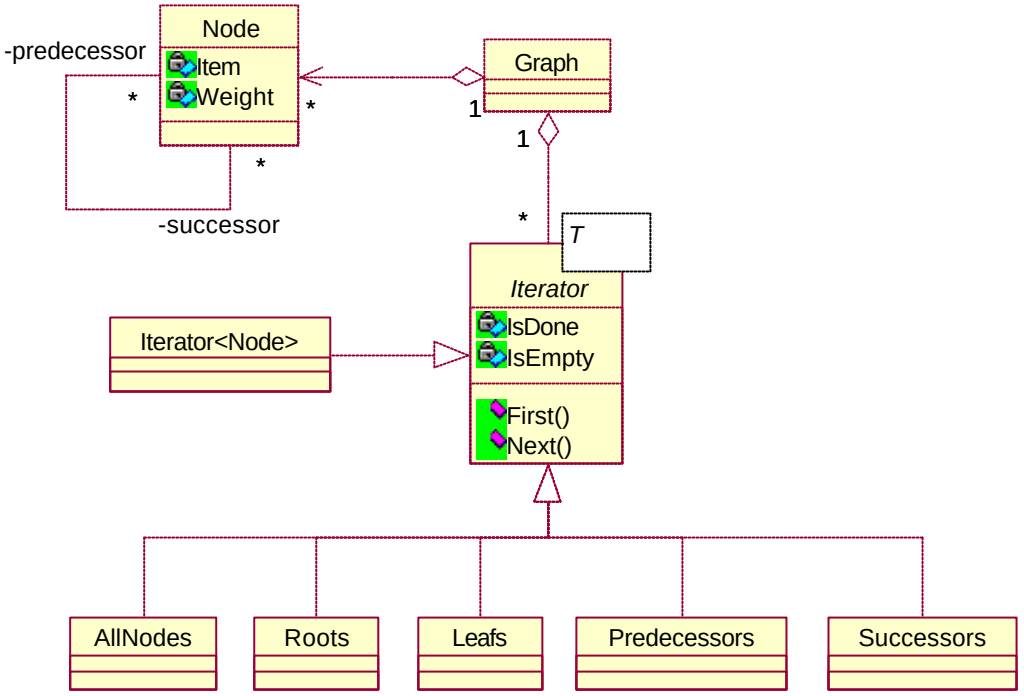

Figure 60. Package *Graph* de l'outil *TestPlan*.

Notons que la classe abstraite *Iterator* s'est groupée avec ses implémentations, les classe *AllNodes*, *Roots*, *Leafs*, *Predecessors* et *Successors*. Dans cet exemple, on voit une CFC de 6 unités : *Graph*, *Iterator-AllNodes*, *Iterator-Roots*, *Iterator-Leafs*, *Iterator-Predecessors* et *Iterator-Successors*

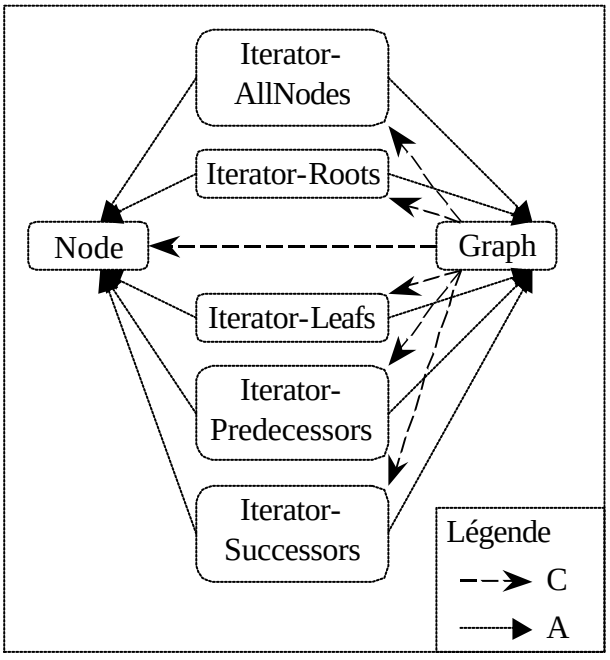

Figure 61. Modélisation - Le package Modélisation de TestPlan : le GDT.

6. Résumé

Dans ce chapitre, nous avons présenté un ensemble de règles de modélisation pour transformer une structure de dépendances en graphe de dépendance de test. La structure de dépendances peut être extraite à partir d'un diagramme de classes UML.

Cette modélisation donne un graphe de dépendances de test. Les unités sont modélisées par les nœuds. Les unités abstraites (comme les classes abstraites, les interfaces ou les méthodes virtuelles) sont assemblées avec leurs implémentations. Dans ce graphe, les dépendances explicites ainsi que les dépendances implicites, du fait du polymorphisme, sont modélisées par des arcs. Ce graphe ne garde qu'un arc pour chaque relation d'un nœud vers un autre.

Ce modèle a l'avantage d'intégrer les dépendances polymorphes ainsi que le suggérait fortement l'approche de Labiche. Il tire parti des catégories de Kung et considère le même classement en difficulté à créer les simulateurs (héritage plus complexe que composition, lui-même plus complexe qu'association).

Enfin, ce graphe est prévu pour travailler au niveau de détail de la méthode, afin de permettre – par une planification plus précise – de diminuer le nombre de bouchons de test.

Chapitre IV.

Décomposition des composantes fortement connexes

La stratégie Big-Bang ne permet pas de localiser les erreurs qui éventuellement apparaissent quand on fait le test. Il faut donc utiliser une stratégie progressive qui intègre les unités, l'une après l'autre.

La modularité d'un système à objets entraîne une forte connectivité des graphes de dépendances de test (GDTs) correspondants. Cette forte connectivité empêche l'application d'une stratégie d'intégration progressive à cause de l'existence de composantes fortement connexes (CFC). Les GDTs qui contiennent ces cycles sont appelés les GDTs cycliques. Pour transformer un GDT cyclique en un graphe acyclique sur lequel un ordonnancement progressif est applicable, on doit décomposer ces composantes fortement connexes.

La décomposition de CFCs est faite en introduisant des bouchons. A cause des risques d'utilisation et du coût de création des bouchons, la décomposition de CFCs doit minimiser d'abord le nombre des bouchons.

Parmi les trois grandes étapes du test d'intégration, l'optimisation de la décomposition des composantes fortement connexes est le problème le plus difficile (NP-complet [Gondran 1985]).

Dans ce chapitre, nous présentons notre heuristique pour choisir des bouchons afin de satisfaire les exigences de la décomposition de CFCs pour l'intégration : minimisation du coût de création et d'utilisation des bouchons. Cette heuristique n'élimine pas les arcs de type Héritage car les CFCs n'existent plus quand toutes les dépendances de trois types Méthode, Association et Composition sont simulées (une hiérarchie d'héritage étant forcément acyclique). Cette heuristique se base sur les définitions de cycle élémentaire et de composante fortement connexe – deux notions de base introduites dans le paragraphe 2.b du Chapitre II, « État de l'art ».

Le paragraphe 1 décrit comment on peut trouver les CFCs en utilisant l'algorithme de Tarjan. Cet algorithme est illustré par quatre petits exemples. Le paragraphe 2 présente notre heuristique pour décomposer les CFCs. On utilise le nombre de cycles élémentaires auxquels un nœud ou un arc participe comme le critère principal pour minimiser le nombre de bouchons. On va aussi utiliser le degré d'entrée (le nombre d'arcs entrants) d'un nœud comme un critère supplémentaire pour minimiser le nombre de retests. Cette utilisation est aussi présentée dans le paragraphe 2. La

minimisation de nombre de bouchons est renforcée par la notion de CFC cohérente. Cette notion est présentée dans le paragraphe 3. Enfin, le paragraphe 4 résume les travaux présentés dans ce chapitre par une procédure complète.

1. Détection des composantes fortement connexes

En 1972, Robert E. Tarjan [Tarjan 1972] a adapté l'algorithme de recherche en profondeur (DFS - « Depth First Search ») pour détecter et classer les CFCs. Le texte de l'algorithme de Tarjan tel que nous l'utilisons est présenté dans la Figure 62.

Tarjan utilise une pile (« Stack ») comme dans l'algorithme de recherche en profondeur classique mais il y ajoute une méthode d'accès par l'indice. Pour chaque nœud utilisé, Tarjan associe trois attributs :

i. La position du nœud dans la pile.

ii. La position du prédécesseur du nœud dans la pile.

iii. L'identificateur de la CFC à laquelle chaque nœud appartient. Chaque fois qu'un nœud est empilé, cet attribut prend la taille de la pile. La valeur de cet attribut va changer via l'algorithme de Tarjan.

La Figure 62 présente l'algorithme de Tarjan. Dans le texte de l'algorithme, on utilise les objets dont les méthodes et les attributs sont présentés dans le Tableau 12.

Tableau 12. Décomposition des CFCs : les attributs et les méthodes des objets dans l'algorithme de Tarjan

Méthode/Attribut	Classe	Signification
TousNoeuds	Graphe	Rendre l'ensemble des nœuds du graphe
S(n)	Graphe	Rendre l'ensemble des successeurs du nœud n
Élément	Ensemble	Rendre un élément quelconque
Élément (i)	Pile	Rendre le i-ème élément, à compter de la racine de pile
Empiler (n)	Pile	Empiler le nœud n au sommet de pile
Dépiler	Pile	Dépiler
Sommet	Pile	Rendre le nœud au sommet de pile
IdCFC	Nœud	Identificateur de la CFC à laquelle chaque nœud appartient.
Position	Nœud	Position de chaque nœud dans la pile
PositionPrédécesseur	Nœud	Position du prédécesseur de chaque nœud dans la pile
S	Nœud	Ensemble des successeurs du nœud

```
1.   Tarjan est
2.   Entrées
3.      G : un graphe orienté
4.   Variables Locales
5.      N : L'ensemble de nœuds du graphe
6.      p, s : Deux nœuds qui possèdent les attributs : IdCFC
7.                    Position et PositionPrédécesseur
8.      TP : Une pile qui accepte l'accès par indice
9.      CFCCompteur : Le nombre de CFCs sortantes
10.     CFC : un CFC
11.  Sortie
12.     CFCs : L'ensemble de CFCs non-triviales
13.  Début
14.     N ← G.TousNoeuds
15.     Quand N ≠ ∅ Faire
16.        CFC ← ∅
17.        p ← N.Element ; TP.Empiler (p) ; p.S ← G.S(p) ∩ N
18.        p.Position ← TP.Taille ; p.IdCFC ← p.Position
19.        p.PositionPrédécesseur ← 0
20.        Quand TP ≠ ∅  Faire
21.           Quand p.S ≠ ∅ Faire
22.              s ← p.S.Element ; p.S ← p.S \ {s}
23.              Si s ∈ TP Faire                          // Un cycle
24.                 Si s.IdCFC < p.IdCFC Faire
25.                    p.IdCFC ← s.IdCFC                // même CFC
26.                 FinSi
27.              SiNon                                    // descendre
28.                 p ← s ; TP.Empiler (p) ; p.S ← G.S(p) ∩ N
29.                 s.Position ← TP.Taille ; s.IdCFC ← s.Position
30.                 s.PositionPrédécesseur ← p.Position
31.              FinSi
32.           FinQuand
33.           Si p.Position = p.IdCFC Faire
34.              // Racine d'un CFC => Sortir un CFC
35.              CFCCompteur ← CFCCompteur + 1
36.              Quand TP.Taille >= p.Position Faire
37.                 s ← TP.Sommet ; s.IdCFC ← CFCCompteur
38.                 CFC ← CFC ∪ {s} ; TP.Dépiler ; N ← N \ {p}
39.              FinQuand
40.              Si CFC.Taille > 1 Faire          // CFC non-trivial
41.                 CFCs ← CFCs ∪ {CFC}
42.              FinSi
43.           SiNon                            // Reculer dans une CFC
44.              Si s.IdCFC < p.IdCFC Faire
45.                 p.IdCFC ← s.IdCFC                 // même CFC
46.              FinSi
47.           FinSi
48.           Si TP.Taille > 0 Faire                   // Reculer
49.              s ← p
50.              p ← TP.Element (p.PositionPrédécesseur)
51.           FinSi
52.        FinQuand
53.     FinQuand
54.  FinTarjan
```

Figure 62. Décomposition des CFCs : l'algorithme de Tarjan.

Comme dans l'algorithme DFS classique, l'algorithme de Tarjan commence par prendre un nœud (le nœud courant). Chaque fois qu'on progresse dans le parcours du graphe, on doit visiter un des successeurs du nœud courant et celui-ci devient le nœud courant (la phase de descente). Si l'on n'a plus de successeur à visiter, on doit reculer au prédécesseur du nœud courant (phase de recul).

Chaque fois qu'on progresse dans le parcours du graphe, si on trouve un nœud **n** qui existe dans la pile, on sait qu'il existe un cycle. On appelle ce nœud la racine du cycle.

Pour les nœuds qui n'appartiennent à aucun cycle, le comportement de l'algorithme de Tarjan est identique à l'algorithme DFS classique, il les dépile immédiatement en reculant.

Quant aux nœuds qui appartiennent à une CFC, l'idée de Tarjan est de ne pas les dépiler immédiatement. Puisqu'un nœud peut participer à plusieurs cycles, Tarjan va chercher la racine la plus profonde de la CFC. Le dépilage n'est effectué que quand le recul atteint cette racine. Tous les nœuds, de cette racine au sommet de la pile, participent à la CFC car tous les nœuds qui n'appartiennent à aucun cycle sont dépilés.

Pour signaler si un nœud appartient à une CFC, Tarjan utilise un identificateur de CFC (IdCFC) et la position du nœud dans la pile (Position). Au moment d'empiler, cette IdCFC prend la valeur de Position. Si ce nœud appartient à une CFC, cette IdCFC va prendre l'IdCFC de la racine la plus profonde parmi les racines des cycles auxquels il participe. Par conséquent, un nœud dont l'IdCFC a la même valeur que sa position dans la pile est un des deux catégories : soit il est la racine la plus profonde d'une CFC car l'IdCFC des autres nœuds de la CFC a changé ; soit il ne participe à aucune CFC car son IdCFC ne change pas. Le dépilage est effectué quand un recul trouve un de ces deux types de nœuds.

Puisque les nœuds ne sont pas tous dépilés immédiatement à chaque étape de recul, la relation (prédécesseur – successeur) entre un nœud et le nœud au dessus dans la pile n'est plus garantie. Pour trouver le prédécesseur dans la phase de recul, Tarjan utilise le troisième attribut, la position du prédécesseur de nœud dans la pile (PositionPrédécesseur). Il faut aussi une pile qui accepte l'accès par l'indice.

Nous présentons quatre exemples dans l'Annexe II, « Illustration de l'algorithme de Tarjan ». Ces exemples commence par un GDT simple qui ne contient qu'une seule CFC et la CFC ne contient qu'un seul cycle. Les GDTs suivants ajoutent des branches au premier GDT pour créer des CFCs plus en plus complexe afin d' illustrer les comportements différents de l'algorithme de Tarjan face aux CFCs.

2. Décomposition des CFCs

Pour planifier l'ordre d'intégration, il faut décomposer les CFCs. Les CFCs sont dues aux interdépendances, c'est-à-dire, lorsqu'une unité dans une CFC a une dépendance, directe ou indirecte, avec toutes les autres unités de la CFC. Pour décomposer une CFCs, il faut donc casser une de ces dépendances et pour cela, introduire autant de bouchons qu'il y a d'arcs à supprimer.

2.a. Bouchon

L'idée générale pour décomposer une CFC est de remplacer un service (une dépendance), fourni par une unité U, par un autre service, dit «identique », fourni par un simulateur S qui, quant à lui, n'utilise aucun service fourni par les autres unités de la même CFC. Ce simulateur est souvent appelé « bouchon ».

Les CFCs peuvent être décomposées en utilisant des bouchons spécifiques ou réalistes (voir la Figure 63) :

i. Bouchon spécifique : Un bouchon qui simule seulement les services d'un seul client. On peut dire qu'un bouchon spécifique est un bouchon « ciblé ».

ii. Bouchon réaliste : Un bouchon qui simule les services de tous les clients. On peut dire qu'un bouchon spécifique est un bouchon « universel ».

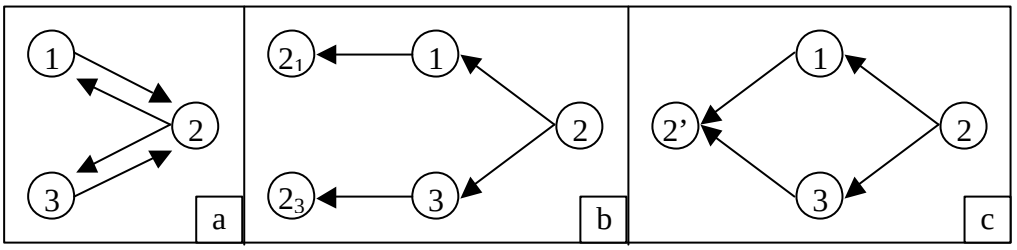

Figure 63. Décomposition des CFCs : Bouchon.

a – CFC ; b – bouchon spécifique ; c – bouchon réaliste.

En général, les bouchons sont les versions simplifiées des unités originelles. Il y a trois raisons pour que les services qu'un bouchon fournit ne soient pas souvent identiques à ceux que l'unité originelle peut offrir :

i. La restriction d'utilisation des services : un bouchon ne doit pas utiliser les services d'autres unités de la même CFC.

ii. La complexité : un bouchon est aussi une unité. Si on crée un bouchon qui peut offrir tout ce qu'une unité originelle peut offrir, on risque de créer une unité aussi complexe que l'originelle.

iii. La demande de cas de test : Le test est rarement un test exhaustif. Les cas de test sont un sous-ensemble des cas possibles. Les bouchons ne doivent pas simuler tous les services que l'unité originelle peut offrir mais ils fournissent seulement les services que les cas de test demandent.

Les bouchons ne sont pas les « vraies » unités. L'utilisation de bouchons est donc risquée et coûteuse : d'une part, les bouchons sont aussi des morceaux de code donc ils peuvent porter aussi des erreurs ; d'autre part, les bouchons sont utilisés dans la phase de test mais pas dans le produit final. Il faut fortement minimiser le coût de création et d'utilisation de bouchons.

2.b. Minimiser le coût de création de bouchon

Le coût de simulation d'un service dépend de la complexité du service : plus le service est complexe, plus on doit dépenser de temps, de travail pour le simuler. Cependant, on n'a pas souvent suffisamment d'information dans la phase de conception pour estimer la complexité d'un service et par conséquent, le coût de sa simulation.

Un bouchon peut simuler un ou plusieurs services. A cause de l'indéterminisme du coût de simulation d'un service, le coût de création des bouchons est aussi souvent difficile à déterminer.

Pour simplifier le problème, dans un premier temps, on fera la supposition que le coût de création de tous les bouchons est identique et on remplace le coût de création total de bouchons par le nombre de bouchons.

Ensuite, pour profiter des résultats de la recherche de Kung sur les types de dépendances, on considère que le coût de création de tous les bouchons spécifiques d'une dépendance de type quelconque est identique ; entre deux bouchons spécifiques de deux types de dépendances différentes, le coût de création est augmenté par l'ordre de type de dépendances : Méthode, Association, Composition mais la différence est négligeable par rapport au coût (en général) de création des bouchons. Comme on l'a dit au début de ce chapitre, on ne s'intéresse pas au type Héritage car les CFCs n'existent plus quand toutes les dépendances de trois types Méthode, Association et Composition sont simulés.

Hypothèse 1. *Supposons que le coût de création de tous les bouchons spécifiques d'un type de dépendance soit identique.*

Hypothèse 2. *Supposons que le coût de création d'un bouchon spécifique d'une dépendance de type Méthode soit inférieur à celui d'un bouchon spécifique d'une dépendance de type Association.*

Hypothèse 3. *Supposons que le coût de création d'un bouchon spécifique d'une dépendance de type Association soit inférieur à celui d'un bouchon spécifique d'une dépendance de type Composition.*

Hypothèse 4. *Supposons que la différence de coût de création entre deux bouchons spécifiques de deux dépendances de types différents soit négligeable par rapport au coût de création du bouchon lui-même.*

En bref, pour choisir un bouchon spécifique, on va utiliser dans un premier temps un critère qui ne tient pas compte du type de dépendances (Hypothèse 1 et Hypothèse 4). Ensuite, si l'utilisation de ce critère ne permet pas de choisir, on sélectionne d'abord les bouchons spécifiques de type Méthode. S'il n'y a pas de bouchon spécifique de type Méthode, on va choisir ceux de type Association. Enfin, s'il n'y a pas de bouchon spécifique de type Association, on va choisir ceux de type Composition (Hypothèse 2 et Hypothèse 3).

Quant aux bouchons réalistes, on considère que chaque bouchon réaliste est une combinaison de plusieurs bouchons spécifiques. Cependant, dans un bouchon réaliste, des services fournis aux clients peuvent partager des services internes. Par conséquent, on ne peut pas estimer le coût de création d'un bouchon réaliste par le

nombre de bouchons spécifiques combinés et on doit retourner à la supposition de l'approximation du coût de création de bouchons comme on a fait avec les bouchons spécifiques (Hypothèse 1 et Hypothèse 4), c'est-à-dire, on considère que le coût de création de tous les bouchons réalistes est identique.

Hypothèse 5. *Supposons que le coût de création de tous les bouchons réalistes soit identique.*

Comme pour un bouchon spécifique, pour choisir un bouchon réaliste, on va utiliser dans un premier temps un critère qui ne tient pas compte du type de dépendances. Ensuite, pour profiter des résultats de la recherche de Kung sur les types de dépendances, on va utiliser le nombre de bouchons spécifiques combinés de chaque type de dépendances comme un critère supplémentaire : on choisit d'abord les bouchons réalistes dont le nombre de bouchons spécifiques combinés de type Méthode est le plus grand, ensuite, de type Association et enfin, de type Composition. Ici, la réduction des doublons (voir le paragraphe 3 du Chapitre III, « Modélisation de la structure de dépendances ») peut changer le résultat de notre solution.

Avec les hypothèses 1, 4 et 5, la minimisation du coût de création des bouchons devient la minimisation du nombre de bouchons. Cependant, les algorithmes proposés peuvent, dans certaine mesure, être adaptés pour gérer des poids associés aux arcs (complexité de la relation entre unité) ou aux nœuds (complexité intrinsèque de l'unité). En effet, le type de dépendance est une sorte de poids associé aux arcs.

(i) Bouchon spécifique

Nous considérons que les cycles sont les éléments de base, construits une CFC. Par conséquent, les arcs participant au nombre maximal de cycles ont la probabilité maximale de participer à une CFC. Pour minimiser le coût de création des bouchons spécifiques, utilisés pour couper les dépendances cycliques, nous proposons d'éliminer successivement les arcs de type Méthode, Association ou Composition (pas d'arc de type Héritage) qui appartiennent au plus grand nombre de cycles jusqu'au moment où la CFC est cassée ; les unités successeurs de ces arcs sont simulées et deviennent les bouchons spécifiques. Parmi les arcs ayant le nombre maximal de cycles auxquels ils participent, on va les éliminer dans l'ordre des types de dépendances correspondantes : Méthode, Association, Composition.

Critère 3. *Pour minimiser le nombre de bouchons spécifiques, un bouchon spécifique b_s est un simulateur d'une unité d'intégration, représentée par le successeur s d'un arc a(p, s, e) qui satisfait les conditions suivantes :*

$$
\begin{aligned}
b_s \leftarrow\ & s \in CFC \mid \exists a(p, s, e) \in A,\ p \in CFC,\ e \in E \setminus \{I\}, \\
& \forall a'(p', s', e') \in A,\ (p', s') \in CFC^2,\ e' \in E \setminus \{I\}, \\
& p \neq p',\ s \neq s', \\
& (a.nbCycles > a'.nbCycles) \vee \\
& ((a.nbCycles = a'.nbCycles) \wedge \\
& ((e = M) \vee \\
& ((e = A) \wedge (e' \neq M)) \vee \\
& ((e = C) \wedge (e' \neq M) \wedge (e' \neq A))))
\end{aligned}
$$

où l'attribut « *nbCycles* » renvoie le nombre de cycles élémentaires auxquels chaque arc participe.

(ii) Bouchon réaliste

Comme pour un bouchon spécifique, Nous considérons que les nœuds participant au nombre maximal de cycles ont la probabilité maximale de casser des CFCs. Pour minimiser le coût de création des bouchons réalistes, nous proposons de simuler successivement les nœuds qui appartiennent au plus grand nombre de cycles jusqu'au moment où la CFC est cassée ; parmi les nœuds participant au nombre maximal de cycles, on va simuler d'abord le nœud ayant le nombre maximal de cycles auxquels les arcs entrants participent. On traite prioritairement les arcs de type Méthode et ensuite ceux de type Association.

Critère 4. *Pour minimiser le nombre de bouchons réalistes, un bouchon réaliste b_r est un simulateur d'une unité d'intégration, représentée par le nœud n qui satisfait les conditions suivantes :*

$$
\begin{aligned}
b_r \leftarrow\ & n \in CFC \mid \forall m \in CFC,\ m \neq n, \\
& (n.nbCycles > m.nbCycles) \vee \\
& ((n.nbCycles = m.nbCycles) \wedge \\
& ((n.nbCycleM > m.nbCycleM) \vee \\
& ((n.nbCycleM = m.nbCycleM) \wedge \\
& (n.nbCycleA \geq m.nbCycleA))))
\end{aligned}
$$

où l'attribut « nbCycleM » rend le nombre de cycles élémentaires auxquels tous les arcs entrants de type Méthode participent. On appelle « cycles Méthode » ces cycles

et l'attribut « nbCycleA » rend le nombre de « cycles Association » – cycles auxquels tous les arcs entrants de type Association participent

et l'attribut « nbCycles » rend le nombre total de cycles sans « cycle Héritage ».

On ne compare pas le nombre de cycles Composition car :

$$n.nbCycles = n.nbCycleM + n.nbCycleA + n.nbCycleC$$

Si l'on utilise la comparaison du nombre de cycles Composition, d'après les hypothèses 2 et 3, cette comparaison devrait être introduite à la fin de l'expression suivante :

$$(n.nbCycles = m.nbCycles) \wedge$$
$$(n.nbCycleM = m.nbCycleM) \wedge$$
$$(n.nbCycleA = m.nbCycleA)$$

Ce qui implique :

$$(n.nbCycleC = m.nbCycleC)$$

(iii) Compter les cycles élémentaires

Pour obtenir le nombre de cycles élémentaires de chaque nœud, on ajoute à chaque nœud un compteur de cycles. Ce compteur est mis à jour par l'algorithme « *CompterNbCycles* » (voir la Figure 64). Cet algorithme se base sur l'algorithme de recherche en profondeur. Chaque fois qu'on descend et qu'on trouve un nœud qui existe dans la pile (ligne 18), on détecte un cycle. On appelle ce nœud «*la racine du cycle* ». On va augmenter le compteur (lignes 21).

Le changement par rapport à l'algorithme de recherche en profondeur est le traitement des nœuds exploités (signalés par l'attribut « *EstExploité* »). Ces nœuds sont des nœuds qu'on a dépilé au moins une fois (ligne 29). Le dépilage est fait en reculant (ligne 30). Pour l'algorithme de recherche en profondeur classique, les nœuds exploités sont ignorés dans les prochaines descentes mais notre algorithme ne les ignore pas car il doit trouver tous les chemins possibles dans un graphe. Cependant, ces nœuds sont ignorés quand on vérifie leur existence dans la pile (ligne 19) puisque tous les cycles dont ce nœud est la racine sont détectés auparavant.

Propriété 2. *Quand un nœud est dépilé, tous les cycles dont ce nœud est la racine sont détectés.*

Preuve :

Un nœud dépilé est marqué par l'attribut EstExploité (ligne 29). Avant d'être dépilé, ce nœud doit se situer au sommet de la pile ***P*** de l'algorithme.

La détection des cycles se base sur la pile ***P***. Si on trouve un cycle ***c(p s …)*** dans la pile ***P*** dont ***p*** est la racine, le nœud ***s*** doit être aussi dans la pile ***P*** et le nœud ***p*** est le nœud le plus profond du ***c*** dans ***P***:

$$\exists c(p\ s\ \ldots) \subseteq P \Rightarrow s \in P$$

Cependant, ***p*** est le sommet de la pile de l'algorithme dont tous ses successeurs ne sont pas dans ***P***.

$$\forall s \in S(p), s \notin P$$

où $S(p)$ est l'ensemble de successeurs de ***p*** dans la CFC ::

$$S(p) \leftarrow \{s \in CFC \mid \exists e \in E, (p, s, e) \in A\}$$

Donc, on ne peut plus trouver un nouveau cycle dont ***p*** soit la racine quand ***p*** est dépilé.

Par conséquent, chaque cycle est détecté une seule fois. Comme aucun nœud n'est éliminé dans l'algorithme *CompterNbCycles*, tous les chemins possibles sont visités. Cet algorithme détecte alors tous les cycles. Les cycles détectés sont les cycles élémentaires car il est compté immédiatement quand l'arc qui ferme le cycle est détecté (ligne 18).

Propriété 3. *L'algorithme CompterNbCycles associe à chaque nœud de la CFC traitée le nombre de cycles élémentaires auxquels appartient ce nœud.*

Dans la Figure 64, nous présentons la version de l'algorithme qui compte le nombre de cycles pour les nœuds. L'algorithme pour les arcs ne change que la partie qui augmente le compteur (ligne 21 de la Figure 64).

Dans cet algorithme, on utilise une pile qui comprend un accès normal au sommet de la pile (opérateur « *Sommet* ») et un accès par index (opération « *Elément* »).

(iv) La complexité théorique et le coût réel de l'algorithme

Notre algorithme doit visiter tous les arcs. Théoriquement, la complexité de cet algorithme dans le pire de cas est O(n!) où «n » est le nombre de nœuds. Ce cas est celui d'un graphe complet où chaque nœud est relié à tous les autres.

Dans les études de cas, le nombre de nœuds (la taille) d'une CFC n'est jamais trop grand. La plus grande CFC que nous avons rencontrée est une CFC de 18 nœuds (SmallEiffel). Le nombre d'arcs pour chaque nœud n'est pas grand non plus. Grâce à une taille faible et une connectivité qui n'est pas trop dense, notre algorithme ne demande pas trop de temps à exécuter. Par exemple, la partie Java de 588 nœuds, 1935 connections qui forment 80 cycles demande moins de 2 secondes sur un UltraSparc 512Mo.

L'intérêt de l'algorithme global est d'exploiter dans un premier temps l'algorithme de Tarjan qui est en coût linéaire et détermine les CFCs. Dans un second temps, puisque notre expérience montre que les CFCs sont de taille et de connectivité raisonnable pour faire ce calcul, on estime le nombre exact de cycles auxquels participe chaque arc ou chaque nœud. Ainsi, pour chaque CFC on a une solution efficace. Par ailleurs, une alternative consiste d'abord à appliquer le calcul coûteux d'estimer le nombre de nœuds et d'arcs de la CFC. Si elle est trop complexe, on applique une heuristique en coût linéaire, analogue à l'algorithme de Bourdoncle [Bourdoncle 1993]

```
1.   CompterNbCycles est
2.   Entrées
3.      CFC : une CFC non-triviale              // est aussi un GDT
4.   Variables Locales
5.      N : L'nsemble des nœuds du graphe
6.      n, p, s : Nœuds qui possèdent d'un compteur de cycles
7.      P : Une pile
8.   Sortie
9.      Mettre à jour le compteur de cycles de chaque nœud de la CFC
10.  Début
11.     Pourtous s ∈ CFC Faire s.EstExploité ← FAUX FinPourTous
12.     N ← CFC.TousNoeuds
13.     Quand N ≠ ∅ Faire
14.        p ← N.Element ; p.S ← G.S(p) ; P.Empiler (p) ;
15.        Quand P ≠ ∅  Faire
16.           Quand p.S ≠ ∅ Faire
17.              s ← p.S.Element ; p.S ← p.S \ {s}
18.              Si s ∈ P Faire                          // Un cycle
19.                 Si ¬(s.EstExploité) Faire            // nouveau
20.                    Pourtous n ∈ P de s au Sommet Faire
21.                       s.nbCycle ← s.nbCycle + 1
22.                    FinPourtous
23.                 FinSi
24.              SiNon                                   // descendre
25.                 p ← s ; p.S ← G.S(p) ; P.Empiler (p)
26.              FinSi
27.           FinQuand
28.           Si P.Taille > 0 Faire                      // Reculer
29.              s ← p ; s.EstExploité ← VRAI
30.              P.Dépiler ; p ← P.Sommet
31.           FinSi
32.        FinQuand
33.     FinQuand
34.  FinCompterNbCycles
```

Figure 64. Décomposition des CFCs : l'algorithme *CompterNbCycles*.

(v) Exemple

Par exemple, pour la CFC de la Figure 65, notre algorithme a trouvé 8 cycles qui sont présentés dans le Tableau 13. La trace d'exécution de notre algorithme sur cet exemple ainsi que les améliorations nécessaires de notre algorithme sont présentées dans l'Annexe III, « Illustration et amélioration de l'algorithme pour compter le nombre de cycles ».

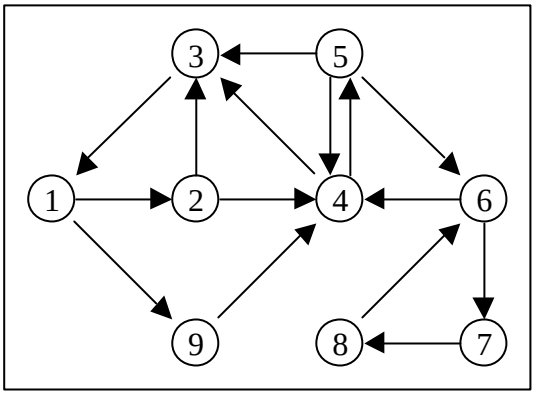

Figure 65. Décomposition des CFCs : l'illustration de l'algorithme de recherche des cycles d'une CFC.

Tableau 13. Décomposition des CFCs : les cycles de chaque nœud du GDT de la Figure 65.

Nœud	Cycles	Nœud	Cycles	Nœud	Cycles
1	(1 2 3), (1 2 4 3), (1 2 4 5 3), (1 9 4 3), (1 9 4 5 3)	4	(1 2 4 3), (1 2 4 5 3), (1 9 4 3), (1 9 4 5 3), (4 5), (4 5 6)	7	(6 7 8)
2	(1 2 3), (1 2 4 3), (1 2 4 5 3)	5	(1 2 4 5 3), (1 9 4 5 3), (4 5), (4 5 6)	8	(6 7 8)
3	(1 2 3), (1 2 4 3), (1 2 4 5 3), (1 9 4 3), (1 9 4 5 3)	6	(4 5 6, (6 7 8)	9	(1 9 4 3), (1 9 4 5 3)

2.c. Minimiser le risque d'utilisation de bouchons

On considère que le risque d'utilisation d'un bouchon dépend de sa complexité et du nombre d'utilisations, c'est-à-dire, du nombre d'appels au bouchon. La minimisation de la complexité des bouchons est résolue dans le paragraphe 2.b. Comme la complexité d'un bouchon, son nombre d'utilisations n'est pas non plus détectable à la phase de conception. Par conséquent, on doit se contenter de la supposition selon laquelle le nombre d'utilisations est égal pour tous les bouchons spécifiques. Avec cette hypothèse, la minimisation de risque d'utilisation des bouchons spécifiques est considérée comme résolue (en même temps que la minimisation de la complexité des bouchons).

Hypothèse 6. *Supposons que le nombre d'utilisations soit égal pour tous les bouchons spécifiques.*

Pour les bouchons réalistes, nous supposons que le nombre d'utilisations d'un bouchon est proportionnel au nombre de bouchons spécifiques combinés, c'est-à-dire, au nombre d'arcs entrants du nœud qui représente l'unité originelle du bouchon dans le GDT.

Hypothèse 7. *Supposons que le nombre d'utilisations d'un bouchon réaliste soit proportionnel au nombre de bouchons spécifiques combinés.*

Pour minimiser le nombre d'utilisations d'un bouchon réaliste, nous allons choisir le nœud dont le nombre d'arcs entrants (de trois types Méthode, Association et Composition) est minimum.

En profitant du résultat de la recherche de David C. Kung sur le type de dépendance (voir le paragraphe 2.c du Chapitre II), si on doit choisir entre deux bouchons réalistes dont le nombre de bouchons spécifiques combinés est égal, nous proposons de choisir le bouchon réaliste ayant le plus grand nombre de bouchons spécifiques combinés de type Méthode car d'après les hypothèses 2 et 3, ce type de bouchon spécifique est le moins complexe ; ensuite, on va choisir le bouchon réaliste ayant le plus grand nombre de bouchons spécifiques combinés de type Association. Comme pour la minimisation de la complexité des bouchons, on ne compare pas le nombre de bouchons spécifiques de type Composition.

Critère 5. *Un bouchon réaliste b_r est un simulateur d'une unité d'intégration, représentée par le nœud n qui satisfait les conditions suivantes :*

$$
\begin{aligned}
b_r \leftarrow\ & n \in CFC \mid \forall m \in CFC,\ m \neq n, \\
& (n.nbEntrants < m.nbEntrants) \vee \\
& ((n.nbEntrants = m.nbEntrants) \wedge \\
& \ ((n.nbEntrantM > m.nbEntrantM) \vee \\
& \ \ ((n.nbEntrantM = m.nbEntrantM) \wedge \\
& \ \ \ (n.nbEntrantA \geq m.nbEntrantA))))
\end{aligned}
$$

où les attributs « *nbEntrantM* » et « *nbEntrantA* » renvoient le nombre d'arcs entrants de type Méthode et de type Association respectivement,

et l'attribut « *nbEntrants* » renvoie le nombre d'arcs entrants sans les arcs de type Héritage.

On a maintenant deux critères pour choisir les bouchons réalistes (Critère 4 et Critère 5), comment va-t-on ordonner leur utilisation ? On considère que la minimisation du coût de création des bouchons est plus importante que la minimisation le nombre d'utilisations de bouchons car elle (la première) participe aussi à la minimisation du risque d'utilisation des bouchons. Un bouchon réaliste b_r est donc choisi par le Critère 6 :

Critère 6. *Un bouchon réaliste b_r est un simulateur d'une unité d'intégration, représentée par le nœud qui satisfait la condition suivante :*

$$
\begin{aligned}
b_r \leftarrow\ & n \in CFC \mid \forall m \in CFC,\ m \neq n, \\
& (n.nbCycles > m.nbCycles) \vee \\
& ((n.nbCycles = m.nbCycles) \wedge \\
& \ ((n.nbCycleM > m.nbCycleM) \vee \\
& \ \ ((n.nbCycleM = m.nbCycleM) \wedge \\
& \ \ \ ((n.nbCycleA > m.nbCycleA) \vee \\
& \ \ \ \ ((n.nbCycleA = m.nbCycleA) \wedge \\
& \ \ \ \ \ ((n.nbEntrants < m.nbEntrants) \vee \\
& \ \ \ \ \ \ ((n.nbEntrants = m.nbEntrants) \wedge \\
& \ \ \ \ \ \ \ ((n.nbEntrantM > m.nbEntrantM) \vee \\
& \ \ \ \ \ \ \ \ ((n.nbEntrantM = m.nbEntrantM) \wedge \\
& \ \ \ \ \ \ \ \ \ (n.nbEntrantA \geq m.nbEntrantA)))))))))
\end{aligned}
$$

2.d. Minimisation de nombre d'étapes de retest

Après avoir intégré le nœud d'origine d'un bouchon, on doit retester tous les nœuds qui ont utilisé ce bouchon pour intégrer comme proposé par Tai-Daniels (voir le Critère 2 du paragraphe 3.c du Chapitre II). Le nombre de retests de chaque bouchon est égal au nombre d'unités, testés avec ce bouchon. Pour minimiser le nombre de retests, il faut choisir les nœuds dont le nombre total de clients est minimum. Ce problème est aussi classé NP-complet.

Dans le paragraphe 2.c, les bouchons sont choisis en minimisant le nombre de dépendances simulées. Cette minimisation minimise au mieux le nombre de retests.

3. Une stratégie intermédiaire entre l'intégration progressive et l'intégration Big-Bang

Par notre expérience, l'intégration n'est pas trop dangereuse quand les unités ne sont pas trop complexes. On constate aussi que la décomposition des CFCs n'est pas nécessaire quand les unités sont très cohérentes comme dans les design patterns (voir la Figure 66 qui présente une instance du design pattern State) qu'on utilise fréquemment dans la conception de logiciel aujourd'hui. Avec cette expérience, on propose d'intégrer entièrement les CFCs dont les nœuds sont cohérents et ne sont pas trop complexes comme un nœud unique.

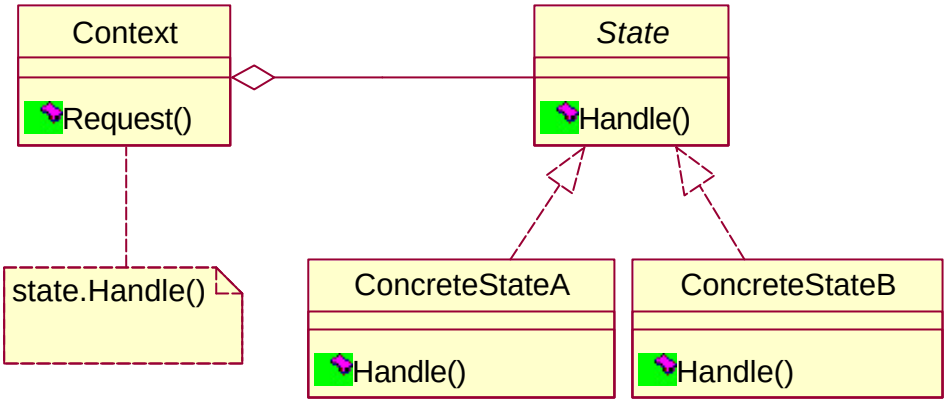

Figure 66. Design pattern State – interdépendance entre les classes.

L'avantage de ce type d'intégration est d'éviter de créer des bouchons dans les CFCs dont on connaît bien la structure et le comportement et qui correspondent à des sortes de « sous-systèmes » cohérents. Par conséquent, l'utilisation de la notion de CFC « cohérente » permet de diminuer considérablement le coût de création des bouchons (voir le Chapitre VI, « Expérimentations et étude comparative des différentes stratégies »).

On peut représenter la notion « trop complexe » par un seuil qu'on appelle le seuil de complexité maîtrisable. La valeur de ce seuil est très instable. Il dépend de la compétence du testeur. Comme on n'a pas d'information sur la compétence de testeur, on doit fixer une valeur comme le seuil maîtrisable de complexité.

Hypothèse 8. *Supposons que chaque testeur peut maîtriser l'intégration directe d'un sous-système cohérent dont le nombre d'unités est inférieur à un seuil S.*

Comme on l'a présenté au début du paragraphe 2.b, on n'a pas suffisamment d'information pour déterminer la complexité d'une unité et par conséquent, celle d'un groupe d'unités. On se contente de l'hypothèse que la complexité de tous les nœuds est égale (voir Hypothèse 5 du paragraphe 2.b). Avec cette hypothèse, un groupe d'unités qui est « trop » complexe est un groupe dont le nombre d'unités dépasse le seuil choisi.

Critère 7. *Pour une CFC, si le nombre d'unités est inférieur au seuil de complexité maîtrisable S, on va intégrer ensemble toutes les unités de cette CFC. Par contre, si le nombre d'unités est supérieur à ce seuil maîtrisable, on va décomposer la CFC en plusieurs CFCs dont le nombre d'unité est inférieur à ce nombre maîtrisable.*

La meilleure façon pour minimiser le nombre de bouchons est de diviser une grande CFC en petites CFCs dont la taille est plus ou moins «équivalente » (comme les deux CFCs 1-2-3-4 et 5-6-7 de la Figure 67) en supprimant un des «ponts » qui les relient (comme l'arcs 7-4 où 3-5 dans la Figure 67). La complexité de ces petites CFCs ne doit pas dépasser le seuil maîtrisable.

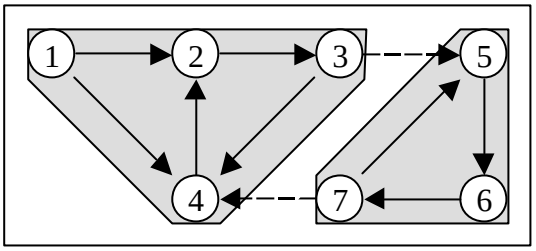

Figure 67. Décomposition des CFCs : les sous-CFCs équivalentes et les ponts.

La détection des ponts et des CFCs équivalentes est aussi un problème NP-complet. En renonçant à les détecter exactement, on propose de simuler un nœud (ou un arc) qui participe au cycle dont le nombre de nœuds est le plus grand car ce bouchon va décomposer la plus grande sous-CFC, ce qui permet de diminuer le plus vite possible la taille des CFCs jusqu'à la taille d'une CFC maîtrisable**.**

Critère 8. *Un bouchon spécifique b_s est un simulateur d'une unité d'intégration, représentée par le successeur s d'un arc* (*p, s, e*) *qui participe au cycle C dont le nombre de nœuds N_C est le plus grand.*

$$b_s \leftarrow s \in CFC \mid \exists C \in CFC, \exists (p, s, e) \in A,$$
$$(p, s) \in C^2, e \in E, \forall C' \in CFC,$$
$$N_C \geq N_{C'}$$

Critère 9. *Un bouchon réaliste b_r est un simulateur d'une unité d'intégration, représentée par le nœud n qui participe au cycle C dont le nombre de nœuds N_C est le plus grand.*

$$b_s \leftarrow n \in CFC \mid \exists C \in CFC, n \in C, \forall C' \in CFC,$$
$$N_C \geq N_{C'}$$

Comment ordonnons–nous l'utilisation du Critère 8 (choisir un bouchon spécifique en utilisant la taille maximale des cycles d'un arc) et du Critère 3 (choisir un bouchon spécifique en utilisant le nombre maximal de cycles d'un arc), ainsi que l'utilisation du Critère 9 (choisir un bouchon réaliste en utilisant la taille maximale des cycles d'un nœuds) et du Critère 6 (critère combiné du Critère 4 – choisir un bouchon réaliste en utilisant le nombre maximal de cycles d'un nœud et du Critère 5 - choisir un bouchon réaliste en utilisant le nombre minimal d'arcs entrants d'un nœud) ?

On sait que le Critère 9 est utilisé pour minimiser le coût de création des bouchons comme le Critère 4 donc il va aussi précéder le Critère 5.

En effet, on ne sait pas parmi les deux choix : diminuer le plus vite la taille d'une CFC à une taille maîtrisable en utilisant le Critère 8 et le Critère 9 (pour stratégie mixe) ou casser le plus grand nombre de cycles en utilisant le Critère 3 et le Critère 6 (pour stratégie progressive pure), quel est le meilleur choix. Puisqu'un bouchon qui participe au nombre maximal de cycles peut aussi diminuer la taille de la CFC et le bouchon qui participe au plus grand cycle ne casse qu'un seul cycle (le plus grand), on propose d'appliquer Critère 3 avant Critère 8 et Critère 4 avant Critère 9. Les critères finaux pour choisir un bouchon sont donc :

Critère 10. *Un bouchon spécifique b_s est un simulateur d'une unité d'intégration, représentée par le successeur s d'un arc $a(p, s, e)$ qui satisfait la condition suivante :*

$$
\begin{aligned}
b_s \leftarrow\ & s \in CFC \mid a(p, s, e) \in A,\ (p, s) \in C^2,\ C \in CFC,\\
& e \in E \setminus \{I\},\\
& \forall a'(p', s', e') \in A,\ (p', s') \in C'^2,\ C' \in CFC,\\
& e' \in E \setminus \{I\},\ p \neq p',\ s \neq s',\\
& (a.nbCycle > a'.nbCycle) \vee\\
& ((a.nbCycle = a'.nbCycle) \wedge\\
& \ ((N_C > N_{C'}) \vee\\
& \ \ ((N_C = N_{C'}) \wedge\\
& \ \ \ ((e = M) \vee\\
& \ \ \ \ ((e = A) \wedge (e' \neq M)) \vee\\
& \ \ \ \ ((e = C) \wedge (e' \neq M) \wedge (e' \neq A))))))
\end{aligned}
$$

Critère 11. *Un bouchon réaliste b_r est un simulateur d'une unité d'intégration, représentée par le nœud n qui satisfait la condition suivante :*

$$
\begin{aligned}
b_r \leftarrow\ & n \in C,\ C \in CFC \mid \forall m \in C',\ C' \in CFC,\ m \neq n,\\
& (n.nbCycle > m.nbCycle) \vee\\
& ((n.nbCycle = m.nbCycle) \wedge\\
& \ ((n.nbCycleM > m.nbCycleM) \vee\\
& \ \ ((n.nbCycleM = m.nbCycleM) \wedge\\
& \ \ \ ((n.nbCycleA > m.nbCycleA) \vee\\
& \ \ \ \ ((n.nbCycleA = m.nbCycleA) \wedge\\
& \ \ \ \ \ ((N_C > N_{C'}) \vee\\
& \ \ \ \ \ \ ((N_C = N_{C'}) \wedge\\
& \ \ \ \ \ \ \ ((n.nbEntrants < m.nbEntrants) \vee\\
& \ \ \ \ \ \ \ \ ((n.nbEntrants = m.nbEntrants) \wedge\\
& \ \ \ \ \ \ \ \ \ ((n.nbEntrantM > m.nbEntrantM) \vee\\
& \ \ \ \ \ \ \ \ \ \ ((n.nbEntrantM = m.nbEntrantM) \wedge\\
& \ \ \ \ \ \ \ \ \ \ \ (n.nbEntrantA \geq m.nbEntrantA)\\
& \ \ \ \ \ \ \ \ \ \ ))))))))))
\end{aligned}
$$

4. Procédure finale

En général, une procédure pour décomposer des CFCs comporte deux étapes principales (voir la Figure 68) : Déterminer les CFCs non-triviales et décomposer les CFCs dont la complexité dépasse un seuil (voir le paragraphe 3).

Ensuite, si une CFC est simple, on ne la décompose pas. On la laisse et on intègre ensemble tous les nœuds de cette CFC. Par contre, si une CFC est complexe, on va essayer de la décomposer en plusieurs CFCs simples.

Pour décomposer une CFC complexe, on doit choisir un arc à simuler (les bouchons sont alors spécifiques) ou plusieurs arcs entrants d'un nœud (avec les bouchons réalistes). Ce travail va être répété jusqu'au moment où il n'existe plus de CFC dont la complexité dépasse le seuil choisi.

Un bouchon spécifique doit satisfaire le Critère 10. Quant à un bouchon réaliste, il doit satisfaire le Critère 11.

Les expérimentations du Chapitre VI, «Expérimentations et étude comparative des différentes stratégies», montrent l'influence du seuil sur le nombre de bouchons qu'il faut créer. Ceci permet de calibrer le seuil raisonnable dans le domaine [2, 5] car au delà de 8, les CFCs deviennent rares jusqu'à des seuils trop importants pour être intégrés d'un seul coup (supérieur à 14). Ces valeurs sont données à titre indicatif à partir des six expérimentations que nous avons faites.

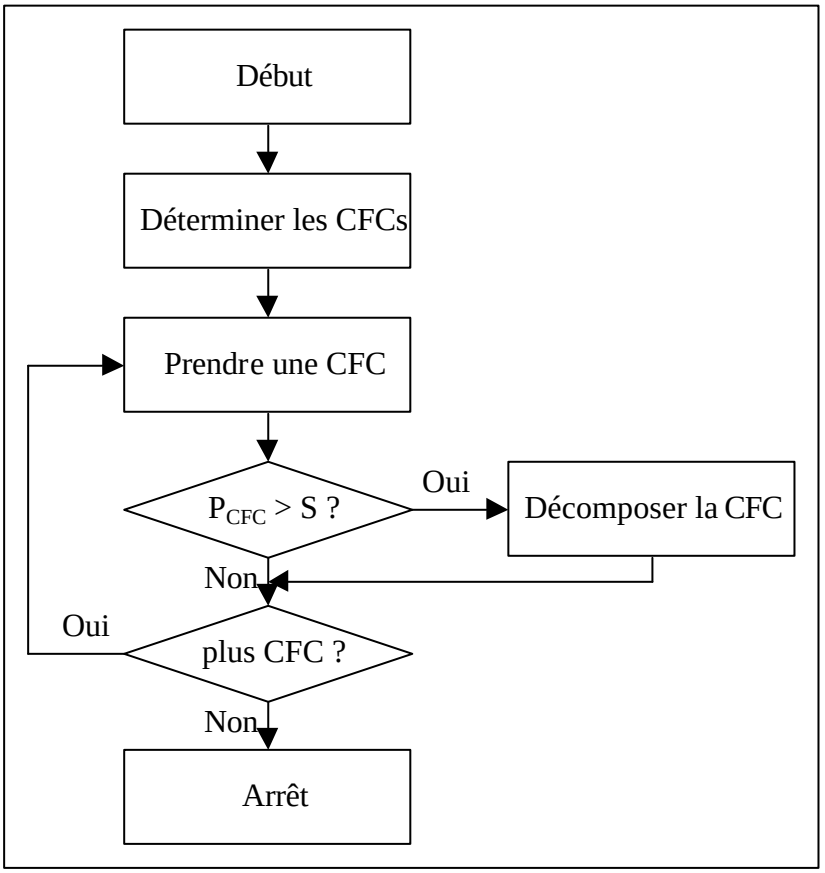

Figure 68. Décomposition des CFCs : la procédure générale.

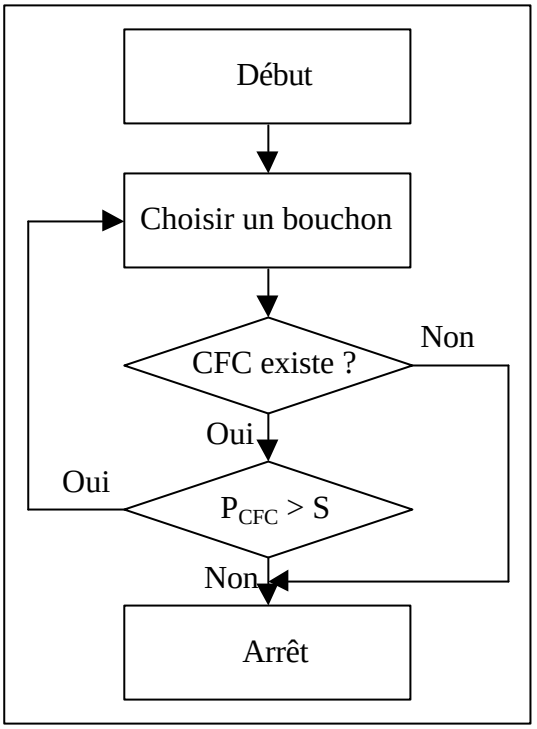

Figure 69. Décomposition des CFCs : la décomposition d'une CFC.

5. Exemple

On va appliquer ces critères pour trouver des bouchons réalistes afin de décomposer les CFCs de notre outil TestPlan. Les diagrammes de classes de cet outil sont présentés dans l'Annexe IV, « La structure de l'outil TestPlan ». Puisque les noms de classes sont longs, nous faisons la numérotation de tous les noms de classes. Le Tableau 14 présente cette numérotation.

Tableau 14. Décomposition de CFCs de l'outil TestPlan : la numérotation de classes.

Classe	N°	Classe	N°
AllNodes	1	Predecessors	10
Graph	2	Roots	11
GraphBuilder	3	SCC	12
Iterator	4	SCCNode	13
Leafs	5	Successors	14
Node	6	TarjanGraph	15
ParallelisationGraph	7	TarjanGraphBuilder	16
ParallelisationGraphBuilder	8	TarjanGraphNode	17
ParallelisationNode	9	TestPlan	18

Le Tableau 15 présente des relations entres les classes de l'outil TestPlan. Dans ce tableau, les dépendances directes et statiques (pas à cause du polymorphisme) entre le nœud identifié par la ligne et le nœud identifié par la colonne sont marquées par les lettres « H », « C » et « A » qui correspondent aux types de ces dépendances ; les dépendances directes et dynamique (dues au polymorphisme) sont marquées par les cellules hachurées ; les autres dépendances de fermeture transitive de ces relations sont marquées par la lettre « x ».

On trouve dans ce tableau une seul CFC de taille 6, composé des nœuds Graph (2) et les Itérators (4.1, 4.5, 4.10, 4.11 et 4.14). L'interdépendance entre ces nœuds est représentée par les cellules treillissées.

Tableau 15. Décomposition de CFCs – l'outil TestPlan : Fermeture transitive.

	2	3.8	3.16	4.1	4.5	4.10	4.11	4.14	6	7	9	12	13	15	17	18
2				C	C	C	C	C	C		C		C		C	
3.8	x			x	x	x	x	x	x	A	A		x		x	
3.16	x			x	x	x	x	x	x		x		x	A	A	
4.1	A				x	x	x	x	A		A		A		A	
4.5	A			x		x	x	x	A		A		A		A	
4.10	A			x	x		x	x	A		A		A		A	
4.11	A			x	x	x		x	A		A		A		A	
4.14	A			x	x	x	x		A		A		A		A	
6																
7	H			x	x	x	x	x	x		C		x		x	
9									H							
12	H			x	x	x	x	x	x	A	x		C		x	
13									H							
15	H			x	x	x	x	x	x		x	A	x		C	
17									H							
18			A	x	x	x	x	x	x	A	x		x	A	x	

Le Tableau 16 présente les paramètres des nœuds dans la CFC : la 1ère colonne présente les nœuds ; la 2nde présente le nombre de cycles auxquels chaque nœud participe ; la 3e présente le nombre de cycles auxquels tous les arcs entrants de chaque nœud de type d'association appartiennent et le 4e colonne – le nombre de cycles auxquels tous les arcs entrants de chaque nœud de type de composition appartiennent. Puisque avec ces trois paramètres, on peut choisir un bouchon, nous ne présentons pas la valeur des autres attributs (le nombre d'arcs entrants de chaque type de dépendance, la taille maximale des cycles auxquels chaque nœud participe).

Tableau 16. Décomposition de CFCs – l'outil TestPlan : les paramètres des nœuds dans la CFC.

Nœud	nbCycles	nbCycleA	nbCycleC
2	5	5	0
4.1	1	0	1
4.5	1	0	1
4.10	1	0	1
4.11	1	0	1
4.14	1	0	1

La taille de la CFC trouvée est 6. Si la taille maîtrisable est inférieure à 7, on doit créer des bouchons. Si on fixe la taille maîtrisable de CFC à 5, la classe Graph est simulée (les cellules de la ligne du nœud 2 du Tableau 16 sont hachurées). Dans notre exemple, la CFC trouvée est totalement décomposée : la Figure 70 présente le GDT obtenu après la phase de décomposition des CFCs. Puisque les types de dépendances ne sont plus utilisés par la suite de notre stratégie, nous ne les présentons pas dans le GDT de la Figure 70.

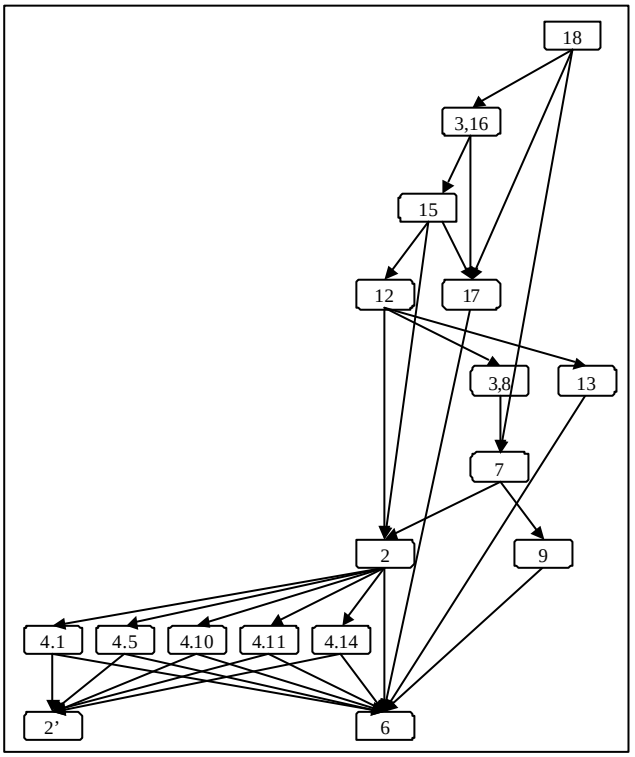

Figure 70. Décomposition de CFCs de l'outil TestPlan : le GDT résultat quand le seuil maîtrisable est inférieur à 7.

6. Résumé

Dans ce chapitre, nous avons abordé la décomposition de CFCs. Nous avons proposé une heuristique pour obtenir une bonne solution pour la minimisation du coût de création de bouchon et du risque d'utilisation de bouchon. Cette heuristique utilise trois paramètres : le nombre de cycles auxquels chaque nœud participe, le nombre de services que chaque nœud fournit et la taille maximale des cycles auxquels chaque nœud participe.

Notre solution (comme les solutions existantes) n'est pas la solution optimale. Nous devons utiliser plusieurs simplifications (les hypothèses des paragraphes 2.b, 2.c et 3). Pour améliorer notre solution, nous devrions chercher un mécanisme pour connaître plus précisément la complexité de chaque unité.

Chapitre V.

Parallélisation du test d'intégration

La décomposition des CFCs donne un graphe de dépendances de test (GDT) acyclique qui permet d'ordonner le test d'intégration. Comme la stratégie d'intégration choisie n'est par la stratégie Big-Bang (voir le paragraphe 3 du Chapitre IV), le test d'intégration est effectué par étapes. Un testeur ne teste qu'une seule unité – ou un groupe d'unités de complexité maximale fixée – à un moment précis.

Le test est une tâche qui est souvent faite par un groupe de plusieurs testeurs et non nécessairement par un seul testeur. Donc, en général, il faut répartir le travail de test. La parallélisation est un problème classique. L'algorithme pour résoudre ce problème est aussi classé NP-complet [Gondran 1985].

Dans ce chapitre, nous présentons notre parallélisation de la tâche d'intégration. Cette parallélisation se base aussi sur la notion du chemin critique.

La notion du chemin critique est reprise dans le paragraphe 1. Ensuite, les équations pour calculer les chemins critiques sont présentées dans le paragraphe 2. La procédure pour partager le test d'intégration est présentée dans le paragraphe 3. Un exemple est donné dans le paragraphe4.

La parallélisation est effectuée aussi par étapes. A chaque étape de parallélisation, un nœud ***n*** du GDT est alloué à un testeur ***t***. Cette allocation signifie que le testeur ***t*** doit tester le nœud ***n***. La séquence de nœuds alloués à un testeur définit le travail entier que ce testeur doit effectuer durant l'intégration. Chaque nœud de cette séquence est associé à un attribut entier, indiquant le décalage en nombre d'étapes entre le début de son intégration et le début de l'intégration du système sous-test entier. Il existe de nombreux travaux de recherche dans le domaine de la parallélisation [Chaudhuri 1992, Ishida 1994, Kumar 1994, Leighton 1992, Oleinick 1982, Zomaya 1996], particulièrement pour les calcules parallèles dans les systèmes répartis. Nous présentons ici notre solution, qui s'est avérée efficace sur les études de cas traitées.

1. Chemin critique

La notion du chemin critique se base sur la notion de la durée d'intégration. Pour chaque unité d'intégration, il faut dépenser une « unité » de temps pour terminer son test d'intégration. Notons que comme pour la complexité d'une unité d'intégration, sa

durée d'intégration est aussi une valeur indéterminable précisément avant que le test soit terminé : il faudra donc en faire l'approximation à priori.

Pour un chemin composé de plusieurs nœuds, la durée d'intégration d'un chemin est :

Définition 23. *La durée d'intégration D_C d'un chemin C est égale à la somme des durées d'intégration de tous ses nœuds.*

$$D_C = \sum_{n_i \in C} D_{n_i} .$$

où D_n est la durée d'intégration individuelle du nœud « n ».

Parmi les chemins, il y a un ou plusieurs chemin(s) dont la durée d'intégration est supérieure à celle des autres. Ces chemins sont appelés chemins critiques.

Définition 24. *Soit E l'ensemble de chemins d'un G (N, A):*

$$E_C \leftarrow \{C_i\,(n_{i,1}, n_{i,2}, \ldots, n_{i,j}, \ldots, n_{i,m}) \mid n_{i,j} \in N,\ i = 1..n,\ j = 1..m,$$

$$\forall i = 1..n,\ \forall\, j = 1..m-1,\ \exists (n_{i,j}, n_{i,j+1}) \in A\}$$

Un chemin critique (C_C) d'un graphe GDT est un chemin dont la durée d'intégration est la plus grande.

$$C_C \leftarrow C \in E_C \mid\ D_{C_C} = \max_{C \in E_C} \left(\langle D_C \rangle \right)$$

Le test d'intégration est parallélisé par des branches du GDT. Une branche est définie comme :

Définition 25. *Une branche d'un GDT (N, E, A) est un chemin dont le début est une racine du GDT et la fin est une feuille du GDT.*

$$B \leftarrow C(n_1, n_2, \ldots, n_m) \mid \forall (a, b) \in N^2,\ \forall e \in E,\ ?\ (a, n_1, e) \in A \wedge\ ?\ (n_m, b, e) \in A$$

où n_1 est le début du chemin C et n_m est la fin du chemin C.

On voit que :

Propriété 4. *Un chemin critique est une branche du graphe.*

Si un chemin critique C_C n'est pas une branche, on peut ajouter des arcs au début ou à la fin de C_C pour obtenir une branche B. En ajoutant des nœuds à C_C, la durée d'intégration de C_C augmente, ce qui veut dire que C_C n'est pas encore un chemin critique.

On a une parallélisation « *totale* » si chaque testeur peut faire le test d'intégration sur une branche. C'est le cas où le nombre de testeurs est égal au nombre de branches

Propriété 5. *Le nombre suffisant de testeurs pour paralléliser totalement le test d'intégration est égal au nombre de branches du GDT.*

Par la définition du chemin critique (Définition 24), on a :

Propriété 6. *Dans une parallélisation totale, la durée de travail de chaque testeur est inférieure ou égale à la durée d'intégration du chemin critique*

Avec la parallélisation, les testeurs qui s'occupent les chemins critiques terminent leurs travaux après les autres. Le test d'intégration termine donc quand les testeurs qui s'occupent des chemins critiques terminent leurs travaux. Par conséquent, on a :

Propriété 7. *Dans une parallélisation totale, la durée d'intégration du système sous test est égale à celle du chemin critique*

Si le nombre de testeurs est supérieur au nombre de branches, les testeurs redondants ne doivent rien faire et la durée d'intégration ne diminue pas. Elle est aussi égale à la durée d'intégration du chemin critique. Donc :

Propriété 8. *Si le nombre de testeurs est supérieur au nombre de branches, la durée d'intégration du système sous test est égale à celle du chemin critique.*

Dans une parallélisation totale, on ne peut pas (et donc on ne doit pas) minimiser la durée d'intégration car elle est déjà la durée minimale – c'est la durée d'intégration d'un chemin critique. Par contre, si le nombre de testeurs est inférieur au nombre de branches, la durée d'intégration dépend du plan d'intégration (comme exemple de la Figure 11 dans le paragraphe 4.d du Chapitre I, « Introduction »).

Comme la parallélisation est un problème NP-complet [Gondran 1985], nous proposons d'utiliser pour heuristique la durée d'intégration des chemins critiques comme critère pour allouer les unité aux testeurs. On sélectionne donc toujours la feuille d'un chemin critique pour allouer à un testeur, on supprime cette feuille du graphe et on itère.

Critère 12. *Pour paralléliser le test d'intégration, on sélectionne toujours la feuille d'un chemin critique pour l'allouer à un testeur.*

Dans une parallélisation non–totale de $\boldsymbol{n}$ testeurs sur un GDT de $\boldsymbol{m}$ branches ($\boldsymbol{n} < \boldsymbol{m}$), on ne peut allouer que n branches çà la fois. On peut donc traiter seulement que n branches de la même manière qu'une parallélisation totale. On appelle ces branches des « *branches d'intégration parallèle* ». Les unités des $(m - n)$ branches restantes doivent être distribuées à n testeur de la manière la plus efficace possible. On appelle ces branches des « *branches d'intégration non-parallèle* ».

A une étape donnée, l'idée de notre heuristique est de profiter de l'écart entre la durée d'intégration du (ou des) chemin(s) critique(s) et des chemins non-critiques : les branches d'intégration non-parallèle les plus longues seront testées par les testeurs non alloués à des chemins critiques. Ils permettent de réduire les durées d'intégration

sur les branches restantes (non-critiques) et d'éviter de devoir se retrouver avec plus de chemins critiques à une étape ultérieure que de testeurs. Par exemple, si on a 2 testeurs et que le GDT du système sous test a 2 branches dont la durée d'intégration est présentée dans la Figure 71, le 1er testeur va tester le chemin critique (le 1er chemin) et le chemin ayant la durée d'intégration minimale (le 4e chemin) et le 2nd testeur – le 2nd et le 3e.

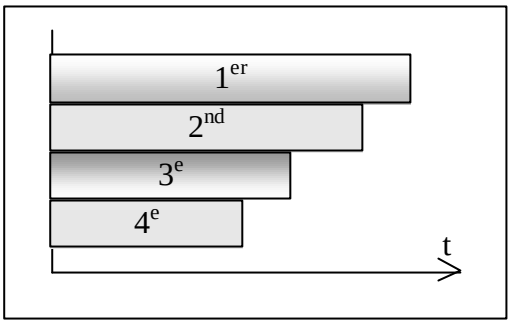

Figure 71. Parallélisation par le chemin critique.

En général, on a :

Propriété 9. *La durée d'intégration minimale* D_{min} *est supérieure ou égale à la plus grande de ces deux valeurs : la durée d'intégration moyenne de chaque testeur et la durée d'intégration d'un chemin critique :*

$$D_{min} \geq \max\left\langle \frac{D}{n}, D_{Cc} \right\rangle$$

où

- D : la durée d'intégration du système sous test quand on le teste par un seul testeur. D est égal à la somme des durées d'intégration de chaque unité ***u*** du système sous test S :

  $$D = \sum_{u \in S} D_u$$

  si on considère que toutes les unités sont de complexité équivalente, D est le nombre d'unités de système sous test.

  $$D = |S|$$

- n : nombre de testeurs qui effectuent la parallélisation (notons que ce nombre n'est utilisé que quand il est inférieur au nombre de branches).

- D_{Cc} : la durée d'intégration d'un chemin critique du GDT.

Par exemple, si on a un GDT de 34 nœud et 8 testeurs, et si chaque nœud prend une unité de temps à être intégré, il faudra au minimum 34/8 étapes pour faire l'intégration. Si, cependant, il existe dans ce GDT un chemin critique de durée 10, il faudra dans le meilleur des cas 10 étapes, car un chemin critique constitue une borne minimale incompressible. On utilisera ce minorant pour comparer l'efficacité de notre heuristique pour la parallélisation au Chapitre VI, « Expérimentations et étude comparative des différentes stratégies ».

2. Recherche des chemins critiques

Comme on l'a présenté, on va choisir tout d'abord la feuille d'un chemin critique. Puisqu'un chemin critique est trouvé en comparant la durée cumulée d'intégration D_C – la durée d'intégration du chemin C, de sa racine jusqu'à sa feuille – on propose d'associer cette durée cumulée à chaque nœud. Les feuilles des chemins critiques du GDT sont des feuilles ayant la plus grande durée cumulée. Cette durée D_C est calculée simplement par un algorithme de recherche en profondeur en se basant sur l'équation suivante :

$$D_C(n) \leftarrow \begin{cases} D_I(n) & \Leftarrow P(n) \neq f \\ D_I(n) + \max\left\langle D_C(m) \middle| \ m \in P(n) \right\rangle & \Leftarrow P(n) \neq f \end{cases}$$

où D_I est la durée d'intégration de chaque nœud et $P(n)$ est l'ensemble de prédécesseurs de n :

$$P(n) \leftarrow \{m \in N \mid \exists (m, n, e) \in A, e \in E\}$$

La Figure 72 présente un GDT. La D_I (en unités de temps, par exemple des heures) de chaque nœud est présentée au dessus de chaque nœud et la D_C de chaque nœud en dessous.

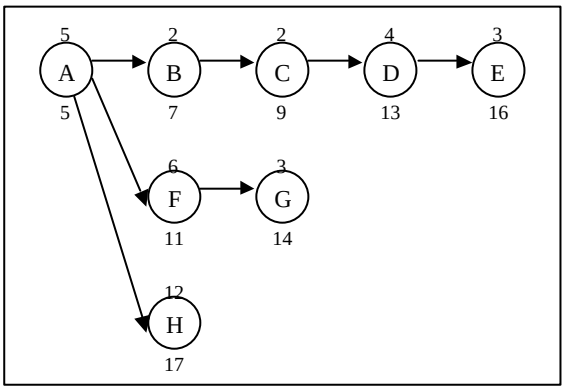

Figure 72. Parallélisation : la durée d'intégration cumulée des nœuds.

En pratique, on ne peut pas déterminer la durée d'intégration avant de l'effectuer. Si on considère que cette valeur est identique pour tous les unités, une feuille d'un chemin critique sera la feuille la plus profonde. On calcule la profondeur p de chaque nœud n par l'équation :

$$p(n) \leftarrow \begin{cases} 1 & \Leftarrow P(n) \neq f \\ 1 + \max\left\langle p(m) \middle| \ m \in P(n) \right\rangle & \Leftarrow P(n) \neq f \end{cases}$$

où $P(n)$ est l'ensemble de prédécesseurs de n :

$$P(n) \leftarrow \{m \in N \mid \exists (m, n, e) \in A, e \in E\}$$

3. Procédure de parallélisation

La procédure de parallélisation est présentée dans la Figure 73. Après avoir calculé les durées d'intégration cumulées, on va allouer les nœuds aux testeurs.

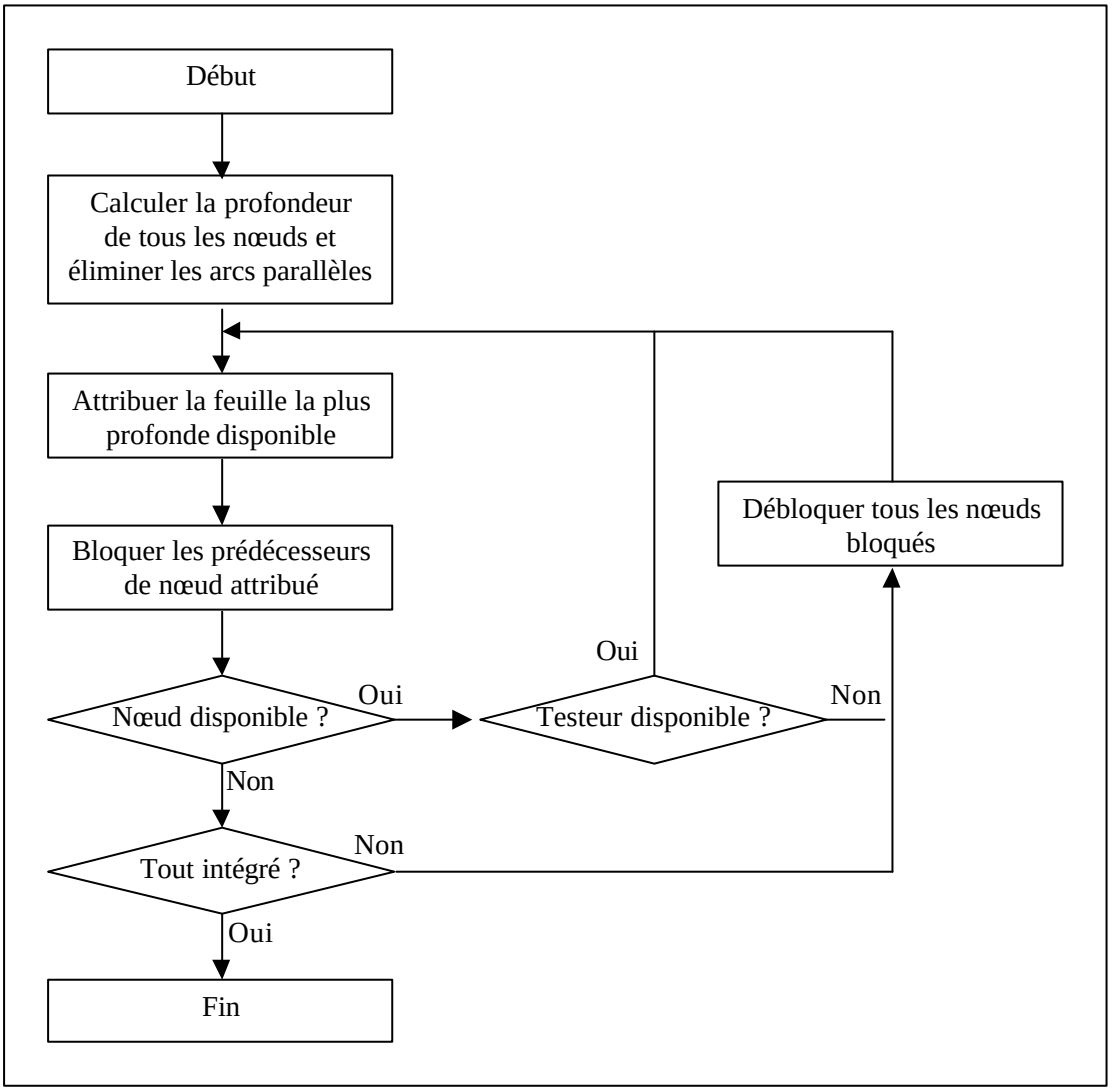

Figure 73. La procédure de parallélisation.

A cause de parallélisation, dans chaque itération d'allocation, on doit faire les traitements suivants :

i. Détermination de la feuille d'un chemin critique disponible,

ii. Blocage et déblocage des prédécesseurs,

iii. Elimination les arcs transitifs.

3.a. Déterminer la feuille d'un chemin critique disponible

A la première itération de l'allocation, on ne doit pas faire cette détermination. On peut choisir n'importe quelle feuille d'un chemin critique pour allouer au premier testeur.

Lors les itérations suivantes, en vue du blocage du prédécesseur des nœuds alloués (voir le paragraphe 3.b), on doit déterminer la feuille d'un chemin critique disponible. Par exemple, pour le GDT de la Figure 72, après l'allocation du nœud E, le nœud D doit être bloqué. Par conséquent, le chemin critique disponible maintenant est donc A-F-G mais pas A-B-C-D.

3.b. Bloquer et débloquer des prédécesseurs

Si on a plusieurs testeurs, après avoir alloué un nœud à un testeur, on va allouer d'autres nœuds aux testeurs restants. A cause de la dépendance d'intégration, un nœud et ses prédécesseurs ne peuvent pas être testés en parallèle. On doit donc bloquer les prédécesseurs des nœuds qui sont en train d'être testés. Ce blocage est suspendu quand le test du nœud concerné se termine. Par exemple, avec le GDT de la Figure 74, si on a deux testeurs, après avoir alloué E, le nœud D doit être bloqué.

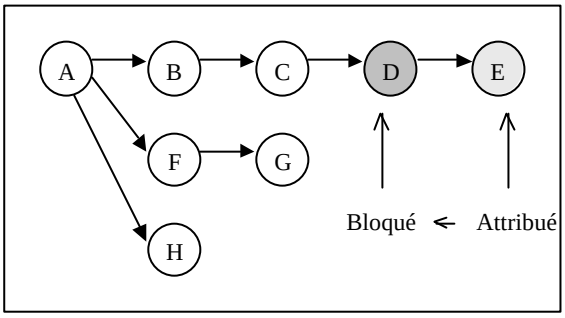

Figure 74. Parallélisation : Blocage du prédécesseur après allocation de son successeur.

3.c. Eliminer les arcs transitifs

Les arcs transitifs ne participent pas à aucun chemin critique mais il participe à tous les déblocages. Par exemple, dans la Figure 75, le chemin critique est toujours le chemin dont le nombre de nœud est le plus grand. C'est le chemin 1-2-3. Quand on alloue le nœud 3 à un testeur, tous ses prédécesseurs, les nœuds 1 et 2, sont bloqués. On voit que le blocage du nœud 2 entraîne le blocage du nœud 1 car le nœud 1 est aussi le prédécesseur du nœud 2. Le blocage du nœud 1, évoqué par l'allocation du nœud 3, est donc inutile. Quand on débloque les prédécesseurs, le déblocage du nœud 3 est aussi inutile.

Pour éliminer les blocages et les déblocages inutiles, il faut éliminer ces arcs parallèles. Cette élimination devrait être faite en même temps qu'on fait la recherche de chemin critique.

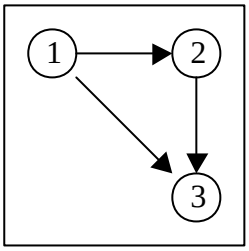

Figure 75. Parallélisation : Elimination des arcs transitifs.

3.d. Exemple

En utilisant la D_I et D_C de chaque nœud comme présente la Figure 72, avec deux testeurs, l'ordre d'intégration sera celui de la Figure 76. Par contre, si on ne connaît pas la durée d'intégration de chaque unité et qu'on la considère comme identique pour tous les unités, l'ordre d'intégration va changer comme le présente la Figure 77.

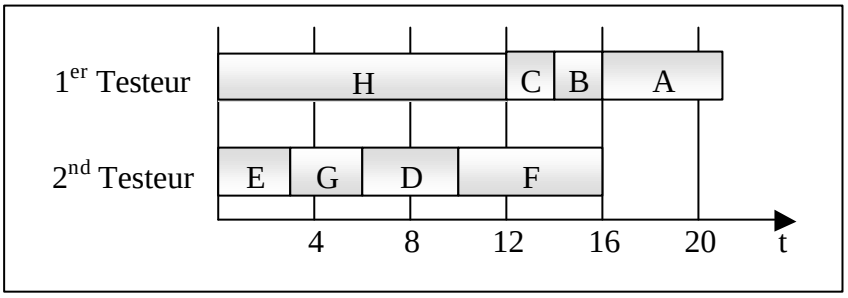

Figure 76. Parallélisation de l'intégration du GDT de la Figure 72.

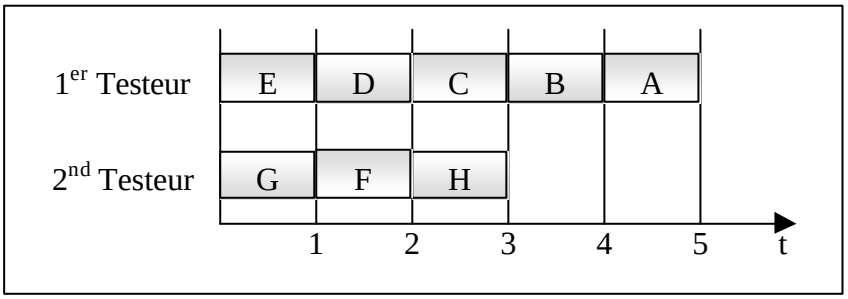

Figure 77. Parallélisation de l'intégration du GDT de la Figure 72, sans l'utilisation des durées d'intégration individuelle.

4. Parallélisation d'intégration de l'outil TestPlan

On va appliquer cet algorithme à l'exemple de l'outil TestPlan. Supposons que la durée d'intégration individuelle de chaque classe de l'outil TestPlan est identique. La Figure 78 présente le résultat obtenu sur la Figure 70. Dans la Figure 78, les arcs pointillés sont les arcs transitifs qu'on va éliminer ; les nœuds sont dassés en couches par leur hauteur : la hauteur de chaque couche est présentée à gauche de la figure. Le

Tableau 17 présente le plan du test d'intégration de notre outil TestPlan quand on a deux testeurs. Dans le Tableau 17, les cellules hachurées présentent les retests.

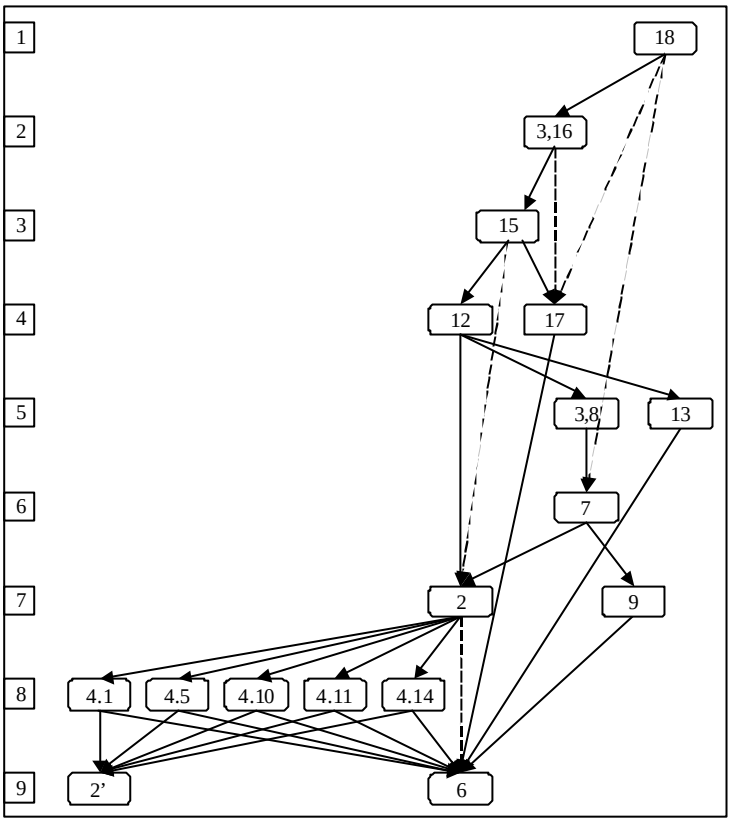

Figure 78. Parallélisation : le GDT acyclique de l'outil TestPlan.

Tableau 17. Parallélisation – l'outil TestPlan : Plan de test d'intégration avec trois testeurs.

Etape	1er Testeur		2nd Testeur	
	Classe	Nœud	Classe	Nœud
1	Bouchon de Graph	2'	Node	6
2	Iterator – AllNodes	4.1	Iterator – Leafs	4.5
	Iterator – Predecessors	4.10	Iterator – Roots	4.11
3	Iterator – Successors	4.14	ParallelisationNode	9
4	Graph	2	SCCNode	13
5	Iterator – AllNodes	4.1	Iterator – Leafs	4.5
	Iterator – Predecessors	4.10	Iterator – Roots	4.11
6	Iterator – Successors	4.14	ParallelisationGraph	7
7	GraphBuilder – ParallelisationGraphBuilder	3.8	TarjanGraphNode	17
8	SCC	12		
9	TarjanGraph	15		
10	GraphBuilder – TarjanGraphBuilder	3.16		
11	TestPlan	18		

5. Résumé

Dans ce chapitre, nous avons présenté notre travail sur la parallélisation du test d'intégration. On a utilisé le chemin critique comme critère principal de la parallélisation. La parallélisation s'effectue à chaque étape en allouant les testeurs aux feuilles des chemins les plus longs. On est ainsi certain de réduire strictement la durée maximale d'intégration (sauf dans le cas où le nombre de testeurs est inférieur strictement aux nombre de chemins critiques). Les résultats expérimentaux sont présentés dans le chapitre suivant : ils révèlent que l'heuristique est particulièrement efficace pour ce problème

.

Chapitre VI.

Expérimentations et étude comparative des différentes stratégies

Dans ce chapitre, nos travaux d'expérimentation sont présentés : ils ont été effectués dans le projet Triskell de l'IRISA en collaboration étroite avec la société Softeam à Rennes.

Dans le premier paragraphe, nous présentons les outils que nous avons utilisés pour faire ces expérimentations. Nous présentons brièvement Objecteering, un outil industriel de Softeam (www.softeam.fr) qui comporte un module pour le test d'intégration implantant notre approche.

Dans le second paragraphe, nous présentons des résultats de décomposition de CFCs, obtenu par notre outil TestPlan (voir la conception et l'utilisation de cet outil dans l'annexe), sur six études de cas. Les résultats obtenus par les autres stratégies, présentées dans le Chapitre II, « État de l'art » sont aussi détaillés afin de permettre la comparaison (sauf celle de Labiche car elle ne traite pas le problème des interdépendances). Nous synthétisons l'efficacité relative de chaque stratégie par rapport à la meilleure sur chaque étude, ce qui permet de tirer des résultats généraux de l'ensemble des études.

Dans le paragraphe 3 du Chapitre IV, nous avons proposé de ne pas décomposer les CFCs « petites » et de fixer un seuil pour la taille maîtrisable d'une CFC (voir le Hypothèse 8 dans le paragraphe 3 du Chapitre IV). Les résultats de cette stratégie intermédiaire entre intégration progressive et Big-Bang sont donnés également, en faisant varier la valeur du seuil d'intégration de 1 à 10 (le seuil représentant la taille maximale d'un interdépendance qu'on estime pouvoir intégrer en une étape).

Concernant la parallélisation des étapes de test, nous donnons les résultats en se comparant au minorant introduit avec la propriété Propriété 9 du Chapitre V, « Parallélisation du test d'intégration ». Enfin, dans le dernier paragraphe, nous donnons nos commentaires sur les stratégies et résultats obtenus.

1. L'outil et la valorisation industrielle

Pour faire les expérimentations, nous avons créé l'outil TestPlan qui implémente notre stratégie de test d'intégration. La structure de cet outil est présentée dans l'Annexe V , « Mode d'emploi de l'outil TestPlan ».

Cet outil est dédié à l'expérimentation et n'a pas d'interface graphique. On le déclenche par une ligne de commande. Il prend en entrée trois données : le fichier qui décrit la structure de dépendances du système sous test ; le seuil de complexité maîtrisable d'une CFC (le nombre de nœud dans chaque CFC simple) et le nombre de testeurs qui partagent le test. Cet outil renvoie l'identification des nœuds qu'on doit simuler et un plan détaillé de la tâche de test. Le mode d'emploi détaillé de cet outil est présenté dans l'Annexe V, « Mode d'emploi de l'outil TestPlan ».

En extension de TestPlan, nous avons aussi implémenté trois autres stratégies, celle de Kung [Kung 1995 #1, Kung 1995 #2 et Kung 1996], de Tai-Daniels [Tai-Daniels 1999] et de Briand [Briand 2001]. Le mécanisme et l'utilisation de ces variantes de TestPlan sont identiques. On utilise la même ligne de commande pour saisir le nom de fichier de la structure de dépendances. Puisque ces stratégies ne font pas la parallélisation, on ne fournit pas le nombre de testeurs. Ces stratégies ne font que la décomposition totale de CFCs, on n'a donc pas à fournir le seuil de complexité maîtrisable d'une CFC – la taille d'une CFC simple.

Dans le domaine industriel, la société Softeam a implémenté notre idée par le module TestStrategy dans Objecteering, un produit commercialisé. Cette implémentation a été réalisée par Clémentine Nébut. On peut voir une présentation plus détaillée de ce module TestStrategy dans l'Annexe VI, « Module TestStrategy d'Objecteering ».

2. Les études de cas

Les six études de cas sont les suivantes (triées par l'ordre alphabétique) :

1. La bibliothèque InterView : Nous reprenons l'étude de cas présentée par Kung dans [Kung 1995 #2]. Il s'agit d'une librairie graphique de grande taille puisque comportant 146 classes et 419 dépendances.
2. Une partie du compilateur orienté objets Java : J2SE v1.31 (http://java.sun.com/j2se/1.3/docs/api/index.html) ; elle comporte de 588 classes et 1935 dépendances.

La bibliothèque Pylon : il s'agit d'une librairie Eiffel (http://www.eiffel-forum.org/archive/arnaud/pylon.htm) qui offre diverses structures de données et tous les services classiques de ces structures de données : elle peut être intégrée à d'autres librairies plus spécialisées. Elle comporte 50 classes et 133 dépendances.

3. Une partie de compilateur Orienté objet : le compilateur GNU Eiffel, lequel est un gratuiciel dont le code source est libre et qui est distribué sous licence GNU (GNU General Public License selon les termes de la Free Software Foundation). Il est adapté à une grande variété de plate-formes. Il est actuellement distribué par http://SmallEiffel.loria.fr et inclue les compilateurs Eiffel vers C et Eiffel en Java bytecode ainsi qu'un outil de

documentation, un « pretty printer » et différents autres utilitaires avec leur code source. Le logiciel comporte plus de 300 classes mais nous n'avons traité qu'une sous-partie de 104 classes et de 140 dépendances de ce compilateur. Son diagramme de classes UML est accessible aux formats MDL, PDF ou Postcript à www.irisa.fr/pampa/UMLAUT/smalleiffel.[mdl|pdf|ps]).

4. Commutateur Télécom SMDS (Switched Multimegabits Data Service) : il comporte 37 classes et 72 dépendances.
5. La bibliothèque Swing de J2EE v1.31 ; elle comporte de 694 classes et 3819 dépendances.

Pour comparer la performance entre des stratégies existantes dans la phase de décomposition des CFCs, il faut fixer des structures de dépendances identiques : nous n'ajoutons donc pas les arcs dus au polymorphisme et nous ne fusionnons pas les nœuds à cause de l'abstraction.

3. Les résultats

Les résultats sont présentés dans deux catégories extrêmes, le nombre de bouchons réalistes si l'on suppose que l'on ne produit que des bouchons réalistes et le nombre de bouchons spécifiques dans l'autre cas.

Les deux premiers tableaux de chaque catégorie (le Tableau 18 et le Tableau 25) présentent le nombre de bouchons que toutes les stratégies proposent pour décomposer totalement les CFCs.

Pour mieux voir la comparaison d'efficacité entre les solutions, nous utilisons un « *indice d'efficacité* ». Plus cet indice est proche de 100, plus la stratégie est efficace. Etant donné une étude de cas ***c***, l'indice d'efficacité $\boldsymbol{I_{s.c}}$ d'une stratégie ***s*** est calculé par l'équation :

$$I(s,c)=\frac{\min_{s\in S}\langle n(s,c)\rangle}{n(s,c)}\times 100\%$$

où

- S : l'ensemble des stratégies

  $S = \{Kung, Tai\text{-}Daniels, Briand, Triskell/Nœud, Ttriskell/Arc\}$

- n(s, c) : la fonction qui renvoie le nombre de bouchons obtenu sur l'étude de cas ***c*** avec la stratégie ***s***.

Intuitivement, une valeur $\boldsymbol{I_{s.c}}$ de 50 signifie qu'on a produit 2 fois plus de bouchons que nécessaire par rapport à la meilleure stratégie sur l'étude de cas considérée. Une valeur de 100 signifie qu'on a exactement 100% d'efficacité relative, ce qui signifie que le nombre de bouchons obtenus est égal au meilleur score. Une valeur de 25 signifie qu'on a 4 fois plus de bouchons que nécessaire.

Etant donné qu'il s'agit de valeurs relatives, on peut faire la moyenne de ces indices d'efficacité pour chaque stratégie et les comparer indépendamment des études de cas particulière. Soit ***E*** l'ensemble des étude de cas *c*, on appelle efficacité relative moyenne ***EffMoy(s)*** d'une stratégie ***s***, la moyenne des ***I(s.c)*** sur l'ensemble des études de cas ***c*** de ***E***.

$$EffMoy(s)=\frac{1}{|E|}\sum_{c\in E} I(s,c)$$

On peut grâce à cet estimateur (qui n'est pas une «*mesure* » au sens strict du terme met un indicateur), avoir un moyen de comparer les efficacités moyennes des stratégies.

L'indice d'efficacité pour minimiser les bouchons réalistes (présenté dans le Tableau 19) est calculé avec les données du Tableau 18. Pour la minimisation du nombre de bouchons spécifiques, les données présentées plus loin au Tableau 25 permettent d'obtenir les indices présentés dans le Tableau 26.

Les tableaux suivants (du Tableau 20 au Tableau 24 et du Tableau 27 au Tableau 32) présentent le nombre de bouchons que notre stratégie demande en utilisant les différents seuils de complexité. Nous présentons les résultats obtenus avec 10 valeurs différentes de ce seuil, de 1 à 10. A chaque tableau est associé un graphique qui permet de visualiser les tendances.

3.a. Le nombre de bouchons réalistes

Le Tableau 18 présente le nombre de bouchons réalistes pour toutes les études de cas en décomposant totalement les CFCs, jusqu'à obtenir seulement des CFCs triviales. Dans ce tableau, la 1ère colonne liste les études de cas, les autres colonnes donnent le nombre de bouchons pour chaque étude de cas et chaque stratégie : la 2nde colonne est le résultat de Kung ; la 3e est celui de Tai-Daniels ; la 4e est celui de Briand ; la 5e colonne présente le nombre de bouchons réalistes, obtenus par notre stratégie, appelée « Triskell » en utilisant les critères qui tentent à minimiser le nombre de bouchons réalistes (la Critère 11 dans le paragraphe 3 du Chapitre IV, « Décomposition des composantes fortement connexes ») – on appelle « Nœud » cette version de notre stratégie ; la dernière colonne de ce tableau présente les résultats, obtenus en utilisant les critères pour minimiser le nombre les bouchons spécifiques (le Critère 10 dans le paragraphe 3 du Chapitre IV, «Décomposition des composantes fortement connexes ») – on appelle « Arc » cette version de la stratégie Triskell.

Pour nos deux options, les résultats présentés dans ce tableau sont obtenus en fixant le seuil de complexité maîtrisable d'une CFC à 1, autrement dit, les CFCs maîtrisable sont seulement les CFCs triviales.

Tableau 18. Etudes de cas : le nombre de bouchons réalistes – Décomposition totale.

Etudes de cas	Kung	Tai-Daniels	Briand	Triskell/Nœud	Triskell/Arc
InterViews	13	22	10	7	8
Java	9	55	10	9	10
Pylon	6	9	6	4	4
SmallEiffel	4	10	5	1	1
SMDS	11	14	10	9	11
Swing	16	61	16	16	16

La Figure 79 présente les données du Tableau 18 sous forme graphique.

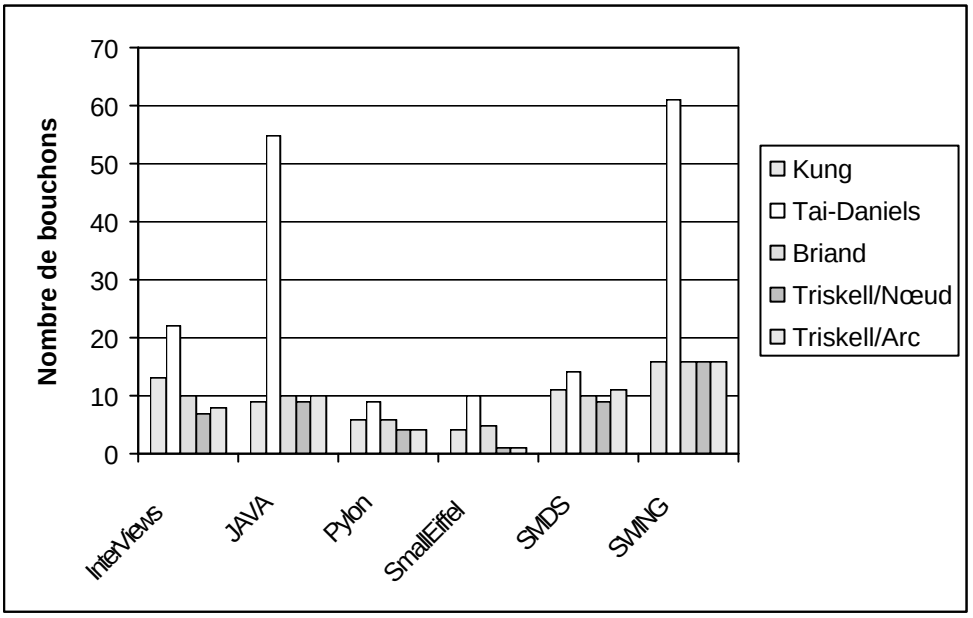

Figure 79. Etudes de cas : le nombre de bouchons réalistes – Décomposition totale.

On peut d'ores et déjà constater que la stratégie Triskell/Nœud se comporte le mieux, alors que celle de Tai-Daniels donne des résultats décevants. On commentera plus loin l'ensemble des résultats comparatifs. Notons que Kung [Kung 1995 #2] affirme qu'il faut 8 bouchons afin de décomposer totalement toutes les CFCs de la bibliothèque InterViews mais notre implémentation de l'algorithme de Kung demande 13 bouchons. Cette différence est liée à l'incertitude de la stratégie de Kung. Elle ne précise pas le critère pour choisir des bouchons spécifiques. De fait, avec notre solution, on obtient seulement 7 bouchons pour décomposer totalement toutes les CFCs de cette bibliothèque.

Dans le Tableau 19, nous présentons l'indice d'efficacité pour minimiser les bouchons réalistes, calculé avec les données du Tableau 18.

Tableau 19. Indice d'efficacité des stratégies pour minimiser le nombre de bouchons réalistes.

Etude de cas	Kung	Tai-Daniels	Briand	Triskell/Nœud	Triskell/Arc
InterViews	53,85	31,82	70,00	100,00	87,50
Java	100,00	16,36	90,00	100,00	90,00
Pylon	66,67	44,44	66,67	100,00	100,00
SmallEiffel	25,00	10,00	20,00	100,00	100,00
SMDS	81,82	64,29	90,00	100,00	81,82
Swing	100,00	26,23	100,00	100,00	100,00
EffMoy	71,22	32,19	72,78	100,00	93,22

Numériquement, les efficacités relatives moyennes révèlent que la stratégie de Tai-Daniels est 3 fois moins efficace que la meilleure, alors que celles de Kung et de Briand sont très proches (et réclament moitié plus de bouchons que Triskell/Nœud, la stratégie qui minimise le mieux le nombre de bouchons).

Les tableaux, du Tableau 20 au Tableau 24, présentent les nombres de bouchons réalistes pour chaque étude de cas. Ces résultats sont obtenus par notre stratégie en variant le seuil de complexité de 1 à 10.

Dans ces tableaux, la première ligne présente la taille qu'on a fixée, la 2nd présente le nombre de bouchons réalistes, la 2nd ligne présente le nombre de bouchons réalistes, obtenus avec la version « Nœud » ; la dernière ligne présente le nombre de bouchons réalistes, obtenus avec la version « Arc ».

On ne présente pas le résultat pour la partie du compilateur SmallEiffel car nous trouvons une seule CFC de taille 16 dans cette partie, il n'y a donc pas de gain quand la taille d'une CFC simple est inférieur à 17.

Tableau 20. Etudes de cas : le nombre de bouchons réalistes – Décomposition partielle pour InterViews.

Taille de CFC	1	2	3	4	5	6	7	8	9	10
Nœud	7	3	3	3	3	2	2	2	2	2
Arc	8	5	5	4	4	4	3	3	3	3

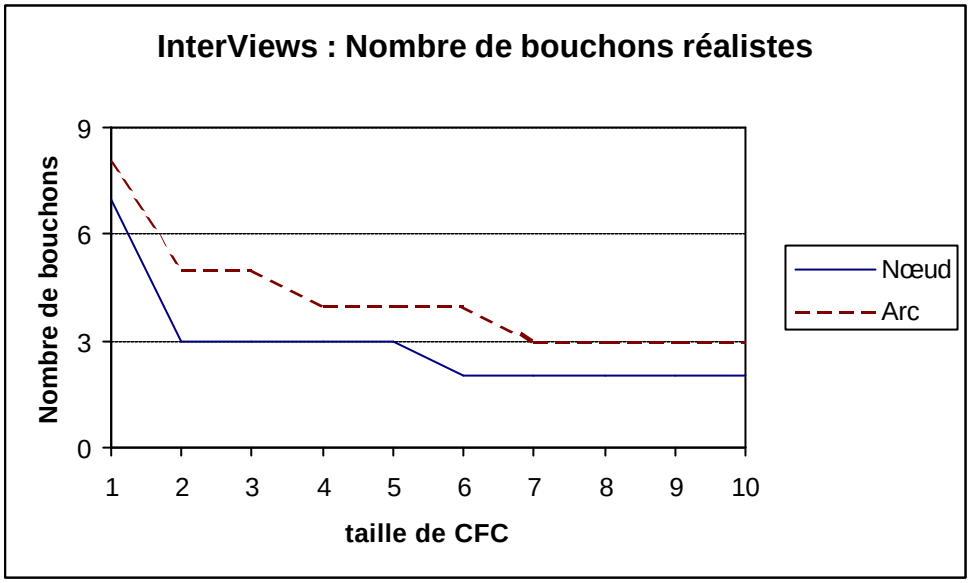

Figure 80. Etudes de cas : le nombre de bouchons réalistes – Décomposition partielle pour InterViews.

On peut déjà noter que la stratégie intermédiaire permet de réduire considérablement le nombre de bouchons nécessaires. Ainsi, si on intègre les classes directement interdépendantes en une seule étape, on réduit le nombre de bouchons de 7 à 3 (pour la version Noeud). Cette tendance va se confirmer sur les exemples suivants. Ces courbes ont un autre intérêt : elles révèlent la répartition des interdépendances dans le système. On voit apparaître la cartographie du système du point de vue de sa connectivité, si les interdépendances sont plutôt grandes (SmallEiffel) ou petites (InterViews) ou encore réparties régulièrement (Java).

Tableau 21. Etudes de cas : le nombre de bouchons réalistes – Décomposition partielle pour Java.

Taille de CFC	1	2	3	4	5	6	7	8	9	10
Nœud	9	8	7	6	5	4	3	2	1	0
Arc Taille de CFC	10	10	9	8	8	8	6	2	1	0

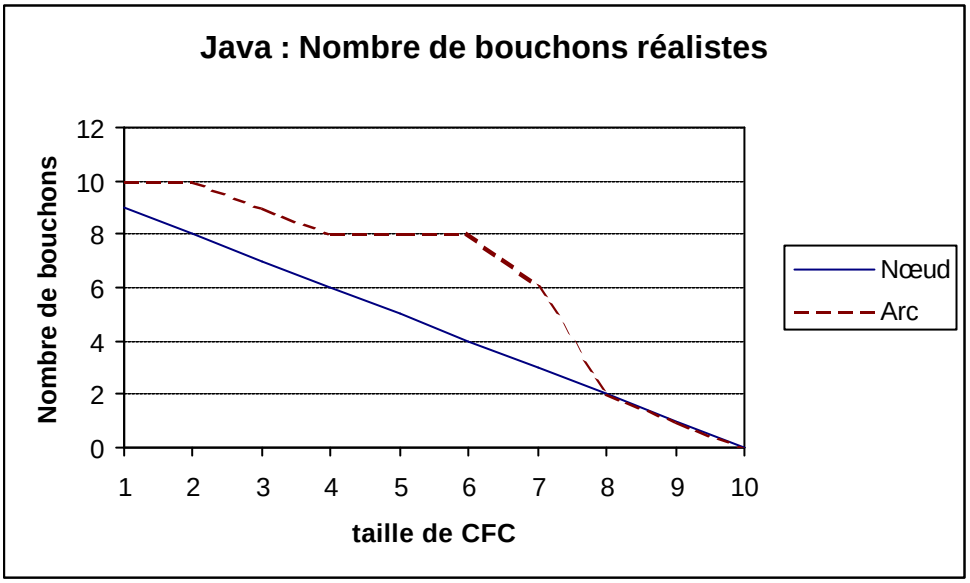

Figure 81. Etudes de cas : le nombre de bouchons réalistes – Décomposition partielle pour Java.

Tableau 22. Etudes de cas : le nombre de bouchons réalistes – Décomposition partielle pour Pylon.

Taille de CFC	1	2	3	4	5	6	7	8	9	10
Nœud	4	2	1	1	1	0	0	0	0	0
Arc	4	2	2	2	1	0	0	0	0	0

Dans l'exemple Pylon, on constate que la CFC de plus grande complexité comporte seulement 6 classes. Cette étude est proche de Swing, qui comporte malgré tout une composante très complexe. La détection de composantes complexes est d'un grand intérêt pour le test d'intégration, car elles sont difficiles à déceler sans l'aide d'un moyen automatique : elles peuvent par exemple traverser des packages de classes, et il serait illusoire d'espérer les déterminer sans l'aide d'un outil.

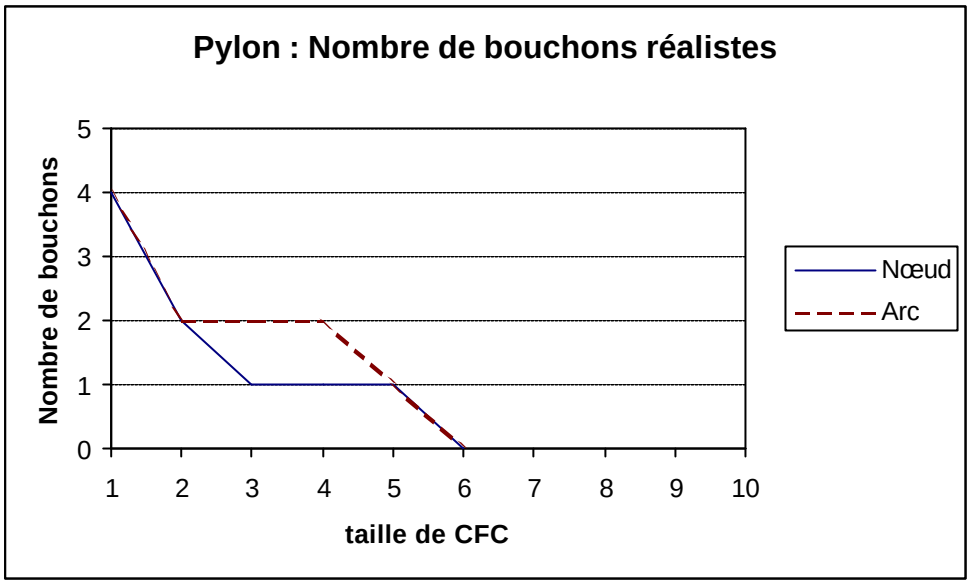

Figure 82. Etudes de cas : le nombre de bouchons réalistes – Décomposition partielle pour Pylon.

Tableau 23. Etudes de cas : le nombre de bouchons réalistes – Décomposition partielle pour SMDS.

Taille de CFC	1	2	3	4	5	6	7	8	9	10
Nœud	9	8	8	7	7	6	6	6	5	5
Arc	11	11	10	10	10	9	9	8	8	8

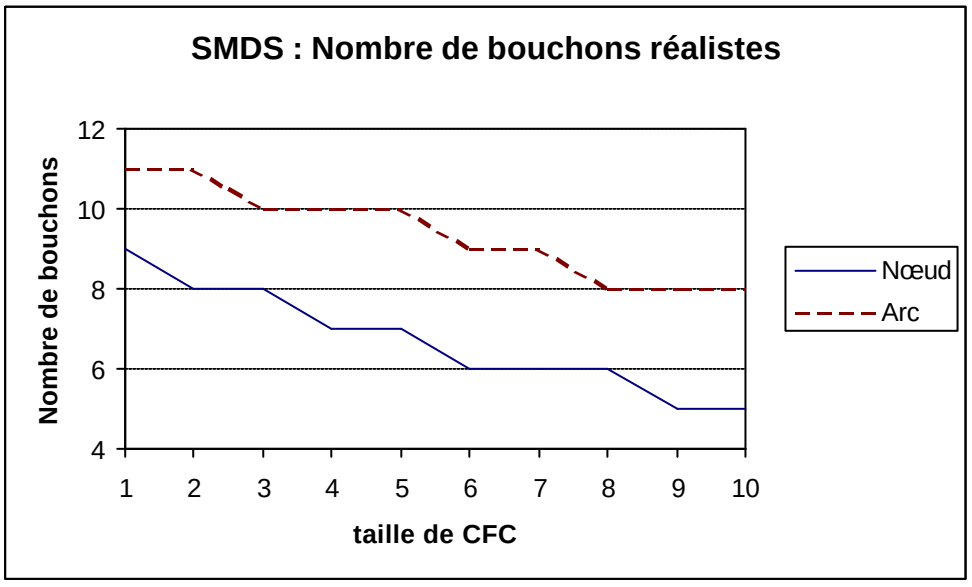

Figure 83. Etudes de cas : le nombre de bouchons réalistes – Décomposition partielle pour SMDS.

Tableau 24. Etudes de cas : le nombre de bouchons réalistes – Décomposition partielle pour Swing.

Taille de CFC	1	2	3	4	5	6	7	8	9	10
Nœud	16	3	3	2	1	1	1	1	1	1
Arc	16	4	3	2	2	1	1	1	1	1

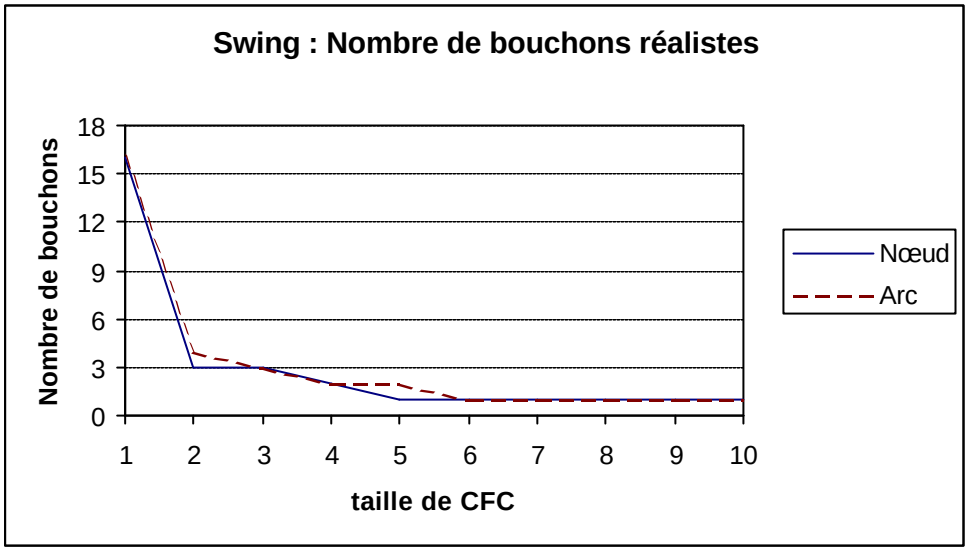

Figure 84. Etudes de cas : le nombre de bouchons réalistes – Décomposition partielle pour Swing.

3.b. Le nombre de bouchons spécifiques

On reprend la même étude avec un décompte des bouchons spécifiques. Les mêmes tendances se retrouvent, c'est pourquoi nous ne les commenterons pas. Le Tableau 25 présente le nombre de bouchons spécifiques pour toutes les études de cas en décomposant totalement les CFCs, c'est-à-dire, après avoir décomposé, il n'existe plus de CFC. La structure de ce tableau est identique à celle du Tableau 18.

Les tableaux, du Tableau 27 au Tableau 32, présentent les nombres de bouchons réalistes pour chaque étude de cas. La construction et la structure de ces tableaux sont identiques à celles des tableaux, du Tableau 20 au Tableau 24.

Les figures, de la Figure 85 à la Figure 91 ont la même fonction que celles des figures, de la Figure 79 à la Figure 84.

Tableau 25. Etudes de cas : le nombre de bouchons spécifiques – Décomposition totale.

Etude de cas	Kung	Tai-Daniels	Briand	Triskell/Nœud	Triskell/Arc
InterViews	44	46	12	13	11
Java	152	199	27	26	26
Pylon	17	23	6	7	6
SmallEiffel	29	28	6	6	6
SMDS	17	17	21	20	20
Swing	162	171	20	20	18

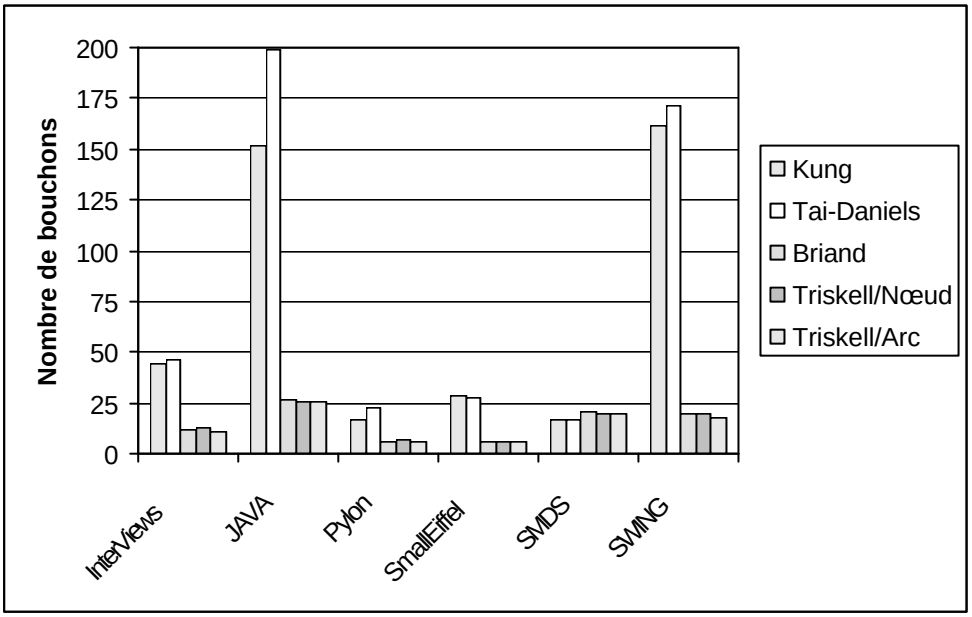

Figure 85. Etudes de cas : le nombre de bouchons spécifiques – Décomposition totale.

On retrouve bien les mêmes résultats principaux qu'avec un décompte des bouchons réalistes, avec un nombre total de bouchons plus importants (ce qui est prévisible puisque on simule un comportement spécifique vis à vis d'un client donné). Cependant, dans ce tableau, on voit que la stratégie Triskell/Arc est maintenant plus efficace que Triskell/Nœud. On remarque aussi que pour l'étude SMDS, les stratégies de Kung et Tai-Daniels sont plus efficaces.

Dans le Tableau 26, nous présentons l'indice d'efficacité pour minimiser les bouchons spécifiques, calculé avec les données du Tableau 25.

Tableau 26. Indice d'efficacité des stratégies pour minimiser le nombre de bouchons réalistes.

Etude de cas	Kung	Tai-Daniels	Briand	Triskell/Nœud	Triskell/Arc
InterViews	25,00	23,91	91,67	84,62	100,00
Java	17,11	13,07	96,30	100,00	100,00
Pylon	35,29	26,09	100,00	85,71	100,00
SmallEiffel	20,69	21,43	100,00	100,00	100,00
SMDS	100,00	100,00	80,95	85,00	85,00
Swing	11,11	10,53	90,00	90,00	100,00
EffMoy	34,87	32,50	93,15	90,89	97,50

On remarque que pour la minimisation des bouchons spécifiques, aucune stratégie n'est à coup sur la meilleure. En terme de tendances, les stratégies Triskell/Arcs et Briand sont les meilleures (scores supérieurs à 90 %), alors que celle de Kung et de Tai-Daniels sont les moins efficaces. On remarque ici qu'une stratégie peut être efficace pour la minimisation des bouchons réalistes et inefficace pour celle des bouchons spécifiques : c'est le cas de la stratégie de Kung. Ceci illustre le fait que les deux problèmes – minimisation des bouchons spécifiques ou réalistes – sont orthogonaux.

Tableau 27. Etudes de cas : le nombre de bouchons spécifiques – Décomposition partielle pour InterViews.

Taille de CFC	1	2	3	4	5	6	7	8	9	10
Nœud	13	9	9	9	9	7	7	7	7	7
Arc	11	8	8	7	6	6	5	4	4	4

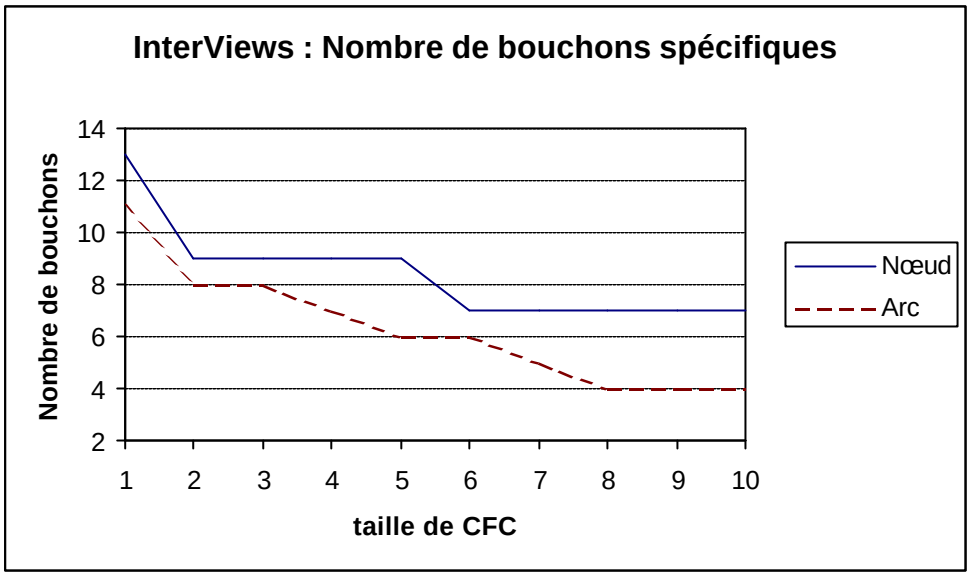

Figure 86. Etudes de cas : le nombre de bouchons spécifiques – Décomposition partielle pour InterViews.

Tableau 28. Etudes de cas : le nombre de bouchons spécifiques – Décomposition partielle pour Java.

Taille de CFC	1	2	3	4	5	6	7	8	9	10
Nœud	26	25	23	22	20	18	16	13	7	0
Arc	26	25	23	22	21	19	8	2	1	0

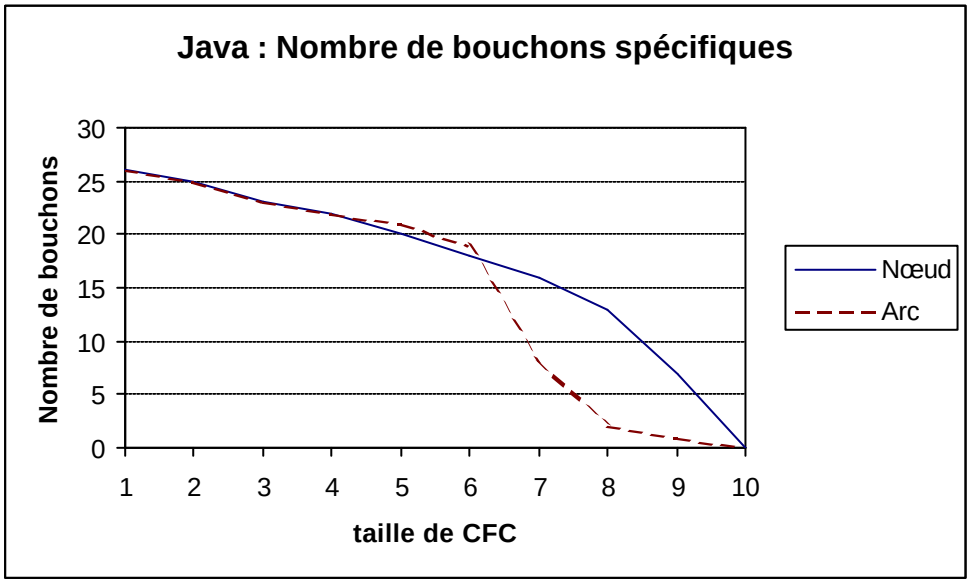

Figure 87. Etudes de cas : le nombre de bouchons spécifiques – Décomposition partielle pour Java.

Tableau 29. Etudes de cas : le nombre de bouchons spécifiques – Décomposition partielle pour Pylon.

Taille de CFC	1	2	3	4	5	6	7	8	9	10
Nœud	7	5	4	4	4	0	0	0	0	0
Arc	6	4	4	3	1	0	0	0	0	0

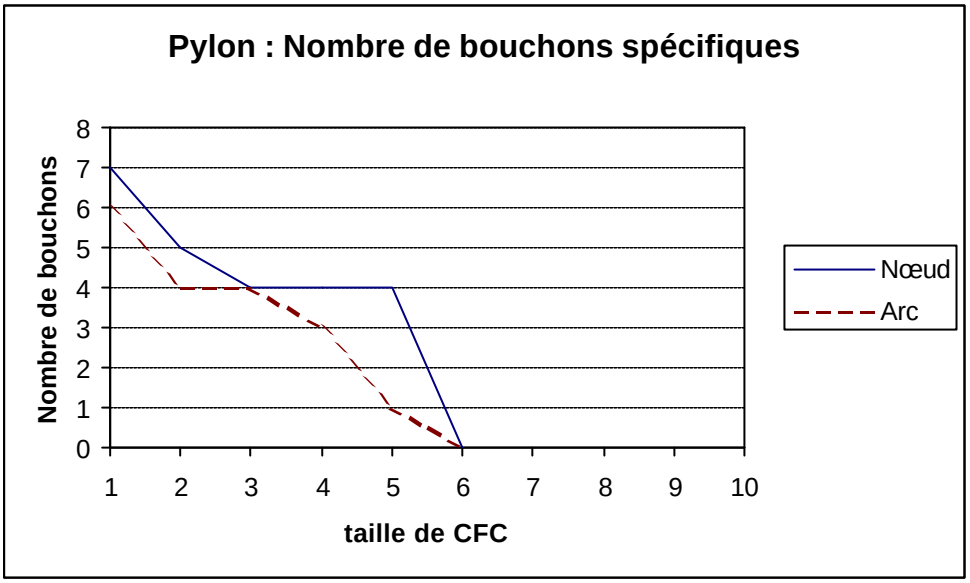

Figure 88. Etudes de cas : le nombre de bouchons spécifiques – Décomposition partielle pour Pylon.

Tableau 30. Etudes de cas : le nombre de bouchons spécifiques – Décomposition partielle pour SmallEiffel.

Taille de CFC	1	2	3	4	5	6	7	8	9	10
Nœud	6	6	6	6	6	6	6	6	6	6
Arc	6	5	4	4	4	3	3	3	3	2

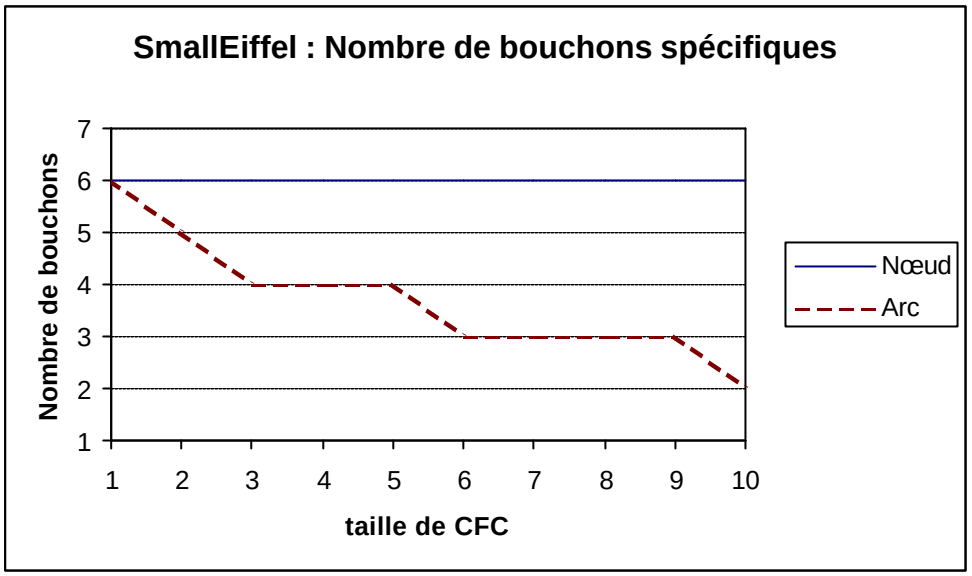

Figure 89. Etudes de cas : le nombre de bouchons spécifiques – Décomposition partielle pour SmallEiffel.

Tableau 31. Etudes de cas : le nombre de bouchons spécifiques – Décomposition partielle pour SMDS.

Taille de CFC	1	2	3	4	5	6	7	8	9	10
Nœud	20	19	19	18	18	17	17	17	15	15
Arc	20	20	19	19	18	17	17	16	16	15

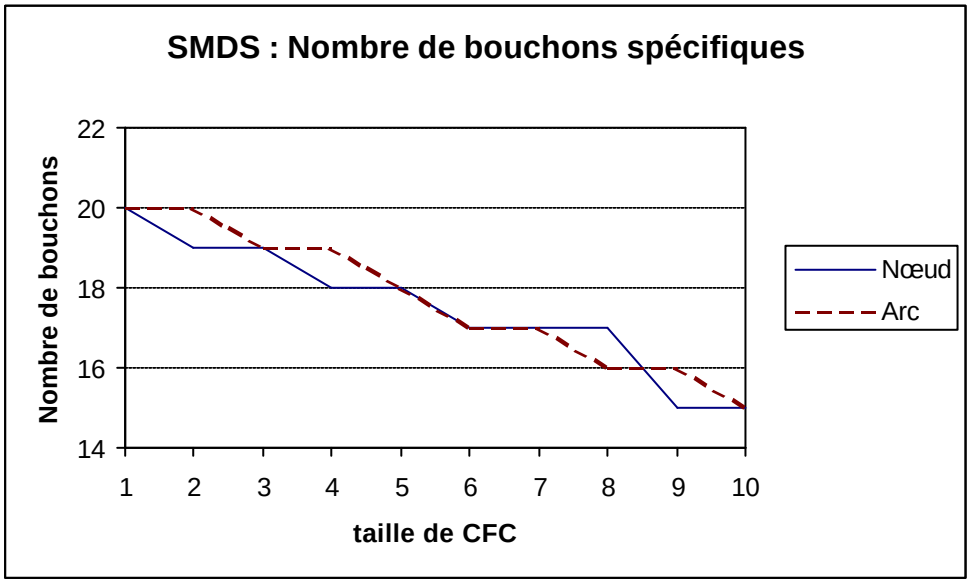

Figure 90. Etudes de cas : le nombre de bouchons spécifiques – Décomposition partielle pour SMDS.

Tableau 32. Etudes de cas : le nombre de bouchons spécifiques – Décomposition partielle pour InterViews.

Taille de CFC	1	2	3	4	5	6	7	8	9	10
Nœud	20	7	7	5	4	4	4	4	4	4
Arc	18	5	4	2	2	1	1	1	1	1

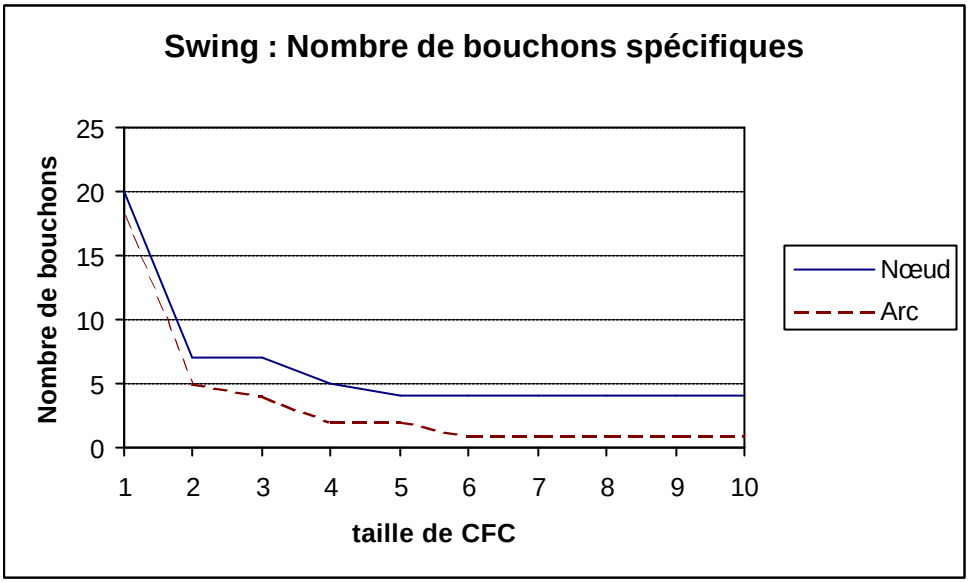

Figure 91. Etudes de cas : le nombre de bouchons spécifiques – Décomposition partielle pour Swing.

4. La durée de parallélisation

Dans cette partie, nous présentons la comparaison entre la durée d'intégration que nous obtenons avec notre stratégie et le seuil théorique (minorant) défini au chapitre 5. Nous constatons que notre heuristique est toujours proche de ce minorant, proche de l'optimal en fait. Sans pouvoir le démontrer, nous suspectons les architectures objets de produire des structures de graphes simples, telle que le problème n'est pas aussi complexe qu'il a été posé avec un graphe acyclique quelconque. L'heuristique proposée fournit en tout cas des résultats proches de l'idéal.

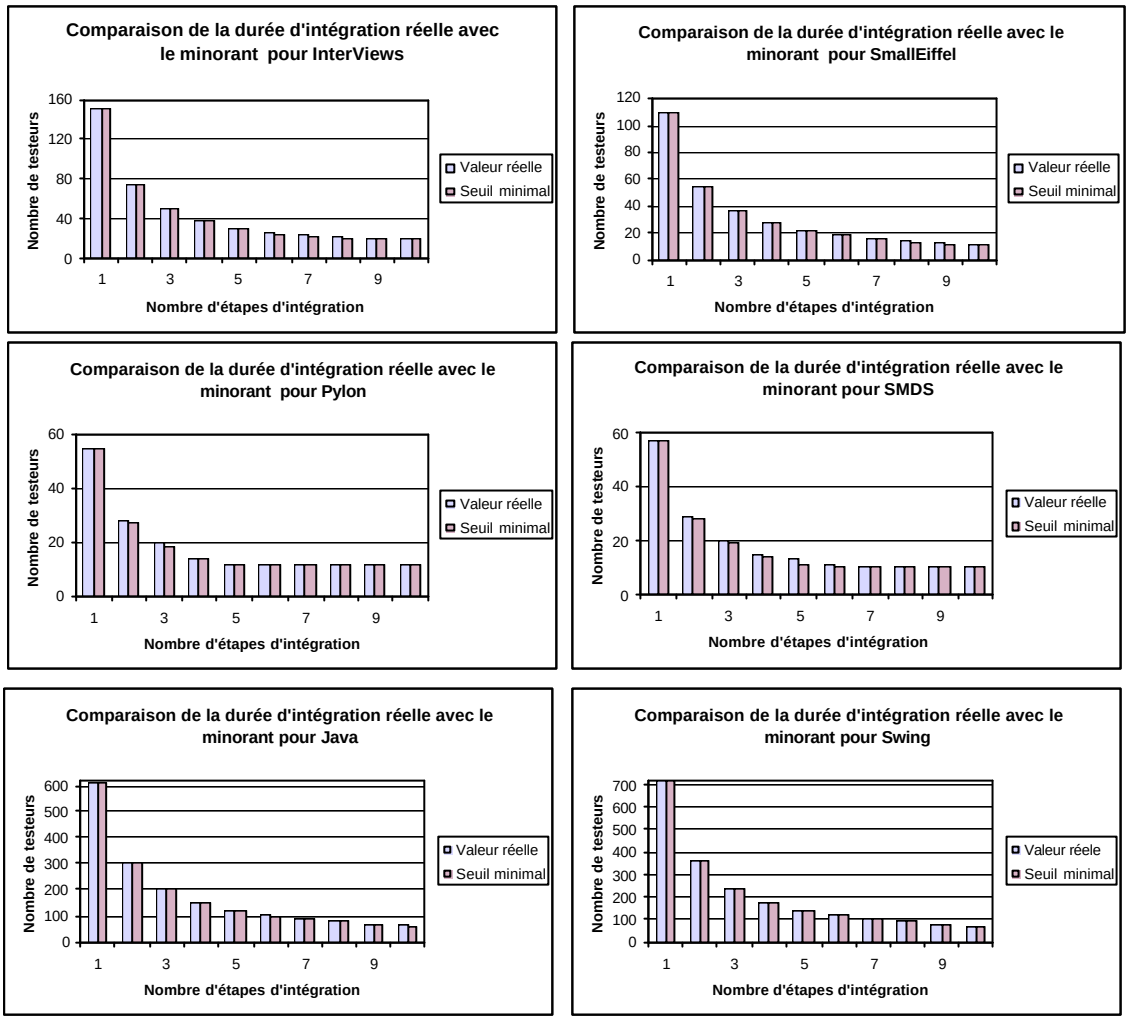

Figure 92. Comparaison de l'efficacité de la parallélisation par rapport au minimum théorique

5. Notre commentaire

Ces résultats nous apprennent que les stratégies que nous proposons sont sensiblement plus efficaces que celles d'autres groupes de recherches, ainsi que le révèlent les indices d'efficacité moyennes (voir les tableaux 18 et 25). Quand nous utilisons la décomposition partielle, le gain est plus net. Plus la taille maîtrisable augmente, plus le gain est grand (voir les tableaux, du Tableau 20 au Tableau 24 et du Tableau 27 au Tableau 32). Le gain de notre solution se porte sur le fait qu'on utilise le nombre de cycles comme critère principal pour choisir des bouchons. On constate en effet que les cycles sont les éléments de base, les « briques » qui déterminent les interdépendances.

Le résultat de Tai-Daniels est nettement plus élevé que les autres. La cause en est la création des bouchons « inter-couches » qu'on peut souvent éviter (voir notre commentaire sur la solution de Tai-Daniels dans le paragraphe 3.f du Chapitre II).

Dans le Tableau 25, pour la bibliothèque Pylon, notre solution Nœud donne un résultat qui n'est pas aussi bon que celui de Briand. La solution de Briand se focalise sur les bouchons spécifiques mais notre solution Nœud se concentre sur les bouchons

réalistes. Notre solution Arc qui cherche aussi à minimiser le nombre de bouchons spécifiques comme celle de Briand, donne des résultats qui sont égaux (dans le cas de la bibliothèque Pylon – Tableau 25) ou meilleurs (dans les autres tableaux) que ceux de Briand. L'utilisation de la solution Nœud est strictement réservée à la minimisation du nombre de bouchons réalistes et la solution Arc – du nombre de bouchons spécifiques.

L'assertion, généralement admise, qu'il y a peu d'interdépendances (CFCs) dans les systèmes à objets est contredite par notre expérience. La plupart des systèmes à objets ont une forte connectivité, même ceux qui sont bien conçus comme Java où l'on a trouvé plus de 80 cycles élémentaires.

Chapitre VII.

Conclusion et perspectives

Nous avons présenté dans ce mémoire nos travaux sur le test d'intégration d'un système à objets.

Dans le premier chapitre, nous avons présenté une vue générale sur le test et particulièrement sur le test d'intégration d'un système à objets. Le second chapitre a détaillé comment travaux précédents ont traité ce problème. Les travaux présentés sont ceux de David C. Kung [Kung 1995 #1, Kung 1995 #2 et Kung 1996], de Kuo Chung Tai et Fonda J. Daniels [Tai-Daniels 1999], de Lionel Briand [Briand 2001] et de Yvan Labiche [Labiche 2000 #1 et Labiche 2000 #2].

En se basant sur ces travaux, et dans le prolongement des travaux engagés auparavant dans le projet Triskell (Jéron 1999 et LeTraon 2000), nous avons proposé notre solution. Elle se compose de trois grandes étapes : modélisation de la structure de dépendances, décomposition des interdépendances et planification du test d'intégration. Les deux premières étapes sont communes aux autres équipes, et nous avons complétée la dernière en introduisant la parallélisation, qui nous semble importante dans le cadre d'un gros projet comportant une équipe de testeurs. Concernant la modélisation, les règles de production d'un graphe depuis un modèle UML sont originales, les travaux existants se basant sur le code.

La modélisation utilise un diagramme de classes UML comme donnée d'entrée. La sortie est une structure de dépendances qu'on appelle un graphe de dépendances de test (GDT). En plus de la classe, notre modélisation accepte aussi la méthode comme une unité d'intégration. La modélisation des méthodes permet de préciser l'ordre d'intégration et de diminuer le nombre de bouchons, éventuellement introduits pour décomposer des CFCs par rapport à la modélisation au niveau de la classe. L'ensemble des types de dépendances introduit par David C. Kung est élargi pour adopter une nouvelle catégorie, la dépendance impliquant une méthode.

Concernant le traitement du polymorphisme et de l'abstraction, notre solution est proche des travaux de Labiche. Cependant, dans notre solution, ces traitements ne s'effectuent pas après la phase de décomposition des CFCs comme dans la solution de Labiche. Nous en tenons compte dès la modélisation. Ce changement permet d'éviter les nouvelles CFCs qui apparaissent à cause du polymorphisme.

Dans la phase de décomposition des CFCs, nous avons proposé, dans un premier temps, une heuristique basée sur plusieurs critères afin de minimiser le nombre des

bouchons et le nombre de retests. Pour compter les bouchons spécifiques, on élimine successivement les arcs qui participent au plus grand nombre de cycles élémentaires. Les unités successeurs des arcs éliminés sont simulés et deviennent les bouchons spécifiques. Quant aux bouchons réalistes, on simule successivement les nœuds qui participent au plus grand nombre de cycles élémentaires. Ces critères permettent de minimiser le coût de création des bouchons. Enfin, en cas d'égalité, le bouchon sera le noeud qui a le plus petit nombre de clients. Ce choix permet de réduire les risques d'utilisation de bouchons et le nombre de retests, introduits à cause de l'usage de bouchons.

Dans un second temps, afin d'éviter de briser des interdépendances correspondants à des « sous-systèmes » fonctionnellement cohérents et de complexité maîtrisable, nous avons proposé une stratégie intermédiaire entre la stratégie Big-Bang (sur un sous-système) - où le sous-système est testé comme une seule unité - et l'approche incrémentale stricte - où les CFCs sont complètement décomposées. Ceci correspond mieux à une approche par objets pour laquelle il est souvent artificiel de créer des bouchons de test au sein d'une petite série de classes coopérant fortement (comme de nombreux design patterns).

Cette stratégie intermédiaire a permis d'améliorer la minimisation du coût de création des bouchons. Pour utiliser cette stratégie, on doit fixer un seuil de complexité. Les CFCs dont le nombre de nœuds est inférieur à ce seuil sont testées comme un seul nœud. Les CFCs plus complexes sont décomposées en plusieurs CFCs simples. Pour décomposer les CFCs œmplexes, parmi les nœuds équivalents selon les critères, on va simuler le nœud participant au cycle de taille maximale. Ce processus est répété jusqu'au moment où les CFCs complexes n'existent plus.

Nous avons finalement proposé une planification qui permet non seulement d'ordonner le test d'intégration mais aussi de le paralléliser pour un groupe de testeurs. Cette parallélisation se base sur la sélection des chemins critiques : l'efficacité de notre solution a été illustrée au dernier chapitre.

En résumé, parmi les trois grandes étapes d'une stratégie de test d'intégration, nous avons proposé un ensemble de règles pour construire la structures de dépendances, une décomposition efficace des CFCs et un ordonnancement simple.

Globalement, notre solution n'est pas optimale. De nombreux problèmes restent à étudier. En particulier, nous n'avons pas trouvé d'argument qui puisse permettre de mesurer, ou même de simplement classer, la complexité des unités d'intégration. En effet, les unités d'intégration ne sont pas toujours équivalentes et la complexité du code ne révèle pas nécessairement la difficulté de l'intégration. Par défaut, on considère que le concepteur doit informer le modèle si il sait que telle ou telle unité est particulièrement critique et lui affecter un poids, correspondant à la durée estimée d'intégration relativement aux autres. Pour affiner un plan d'intégration, il faudrait déterminer les propriétés du logiciel qui ont un impact sur l'intégration. Plus que la complexité du code, des notions telles que la fréquence d'utilisation de l'unité, la position « clé » de l'unité dans l'architecture pour distribuer les traitements, ont un impact très important. Il est très difficile d'appréhender de telles propriétés uniquement avec les vues statiques d'UML : un travail ultérieur devrait se concentrer sur les vues dynamiques d'UML : diagrammes de séquences, d'activités et d'états. A terme, il sera possible de fournir au concepteur un outil permettant d'estimer, avec une bonne précision, le coût et la durée de l'intégration et au testeur une planification détaillée des étapes d'intégration.

Annexe I.

Prise en compte de l'abstraction dans l'approche de Yvan Labiche

Dans cette annexe, nous présentons l'application des règles pour tenir compte l'abstraction de Yvan Labiche, sur le graphe de la Figure 32. Le résultat est le graphe de la Figure 33.

i. La ligne 1 : (C#) est supprimée car C est abstraite.

ii. La ligne 3 : (B, C, E#, F_G, G) est supprimée car E est abstraite.

iii. La ligne 6 :
(A, B, C, D, D_A, E+, F_G, G, H#)
devient
(A, B, C, D, D_A, (E#+), F_G, G, H#)
pour regrouper l'intégration de la classe abstraite E et sa version réalisée H.

iv. La ligne 5 : (A, B, C, D#, D_A, E, F_G, G) est supprimée et assemblée avec la ligne 6 parce que cette étape demande l'intégration de la classe E qui sera testé à la ligne 6.

v. La ligne 6 : le résultat de l'étape (iii) devient
(A, B, C, D#, D_A, (E#+), F_G, G, H#)
pour intégrer la classe D.

vi. La ligne 4 : (A#, B, C, D_A, E, F_G, G) et (B, C+, E, F#, F_G, G) sont traitées comme dans la ligne 5.

vii. La ligne 6 : le résultat de l'étape (v) devient
(A#, B, C, D#, D_A, (E#+), F#, F_G, G, H#)
pour intégrer les classes A et F.

viii. La ligne 6 : le résultat de l'étape (vii) devient
(A#, B, (C#+), D#, D_A, (E#+), F#, F_G, G, H#)
pour regrouper l'intégration de la classe abstraite C et sa version réalisée F.

ix. La ligne 3 : (D_A#) est gardée.

x. La ligne 2 : (B#, C) et (C, F_G, G#) sont supprimés et assemblés avec la ligne 6 parce que cette étape demande l'intégration de la classe C qui sera testée à la ligne 6.

xi. La ligne 6 : le résultat de l'étape (viii) devient
(A#, B#, (C#+), D#, D_A, (E#+), F#, F_G, G#, H#)
pour intégrer les classes B et G.

xii. La ligne 1 : (F_G#) est gardée.

xiii. La ligne 7 et la ligne 8 : les parenthèses sont ajoutées autour de C et E pour préciser le rôle parental de C et E.

Annexe II.

Illustration de l'algorithme de Tarjan

Pour mieux comprendre l'algorithme de Tarjan, nous l'appliquons sur quatre exemples de quatre GDTs, présentés dans quatre figures :Figure 93, Figure 94, Figure 95, et Figure 96. Le premier GDT est le plus simple. Il ne contient qu'une seule CFC et la CFC ne contient qu'un seul cycle. Les GDTs suivants ajoutent des branches au premier GDT afin d'illustrer les comportements différents de l'algorithme de Tarjan face aux CFCs.

Cette illustration suppose que les nœuds du GDT soient numérotés et que l'opérateur « *Elément* » (voir le Tableau 12) rende un nœud par l'ordre numérique. Dans les exemples, les types de dépendances sont ignorés car l'algorithme de Tarjan ne traite pas tes types de dépendances. Notons que s'il y pas de référence précise, toutes les références sur les lignes sont appliquées à la Figure 62, c'est-à-dire, si l'on écrit tout court «*ligne.16* », on veut dire «*ligne 16 de la Figure 62* » ; si on veut dire « *ligne N°4 du Tableau 33* », on doit l'écrire complètement «*ligne N°4 du Tableau 33* ».

1. Une CFC simple

Voyons l'exemple du GDT de la Figure 93, il y a une CFC {3, 4, 5} qui contient un seul cycle (3 4 5). Le Tableau 33 présente la trace d'exécution de l'algorithme de Tarjan, appliqué à ce GDT. Cette trace est révélée après chaque exécution de la ligne 21 que l'on va appeler une « *itération* ».

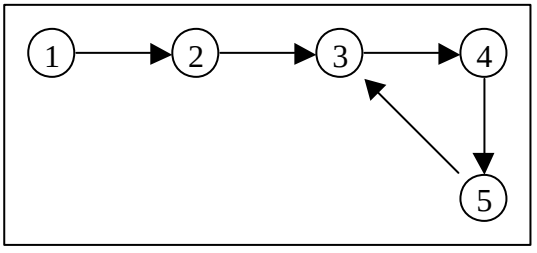

Figure 93. Décomposition des CFCs : Une CFC simple.

Dans le Tableau 33, la colonne « Itération » indique le nombre d'exécution de la ligne 21 dans le texte de l'algorithme de Tarjan (Figure 62); la colonne «ligne » note une ligne dans la Figure 62, de laquelle la trace est révélée ; la colonne «*p.Id* » note l'identification (ou bien le nom) du prédécesseur « *p* » dans l'algorithme de Tarjan.

Tableau 33. Décomposition des CFCs : Algorithme de Tarjan applique au GDT de la Figure 93.

N°	**Itération**	**Ligne**	**TP**	**p**			**s**	**Note**
				Id	**IdCFC**	**Position**		
0	0	13	ϕ					Début
1	1	22	1	1	1	1	2	Descendre
2	2	22	1, 2	2	2	2	3	
3	3	22	1, 2, 3	3	3	3	4	
4	4	22	1, 2, 3, 4	4	4	4	5	
5	5	22	1, 2, 3, 4, 5	5	5	5	3	Trouver cycle (3 4 5)
6	5	25	1, 2, 3, 4, 5	5	3	5	ϕ	Commencer à reculer dans le cycle (3 4 5)
7	6	45	1, 2, 3, 4, 5	4	3	4	ϕ	Reculer dans le cycle (3 4 5)
8	8	33	1, 2, 3, 4, 5	3	3	3	ϕ	Arrêter le recul, sortir la CFC {3, 4, 5}
9	8	50	1, 2	2	2	2	ϕ	Reculer sans cycle
10	9	50	1	1	1	1	ϕ	Dernier dépilage
11	10	54	ϕ					Fin

Après avoir empilé le nœud 5 (ligne N°5 du Tableau 33), on voit que son successeur, le nœud 3, existe dans la pile. La condition de la ligne 23 est satisfaite, on a trouvé un cycle (3 4 5) dont le nœud 3 est la racine. On commence à reculer (ligne N°6 du Tableau 33). L'IdCFC des nœuds 5 et 4 prennent la valeur de l'IdCFC du nœud 3 – la racine du cycle (lignes N°6 et N°7 du Tableau 33).

Le recul s'arrête au nœud 3 (ligne N°8 du Tableau 33) où la condition de la ligne 33 est satisfaite. On va sortir la CFC {3 4 5} (lignes 35.. 42), les nœuds de cette CFC sont dépilés.

Comme les nœuds 2 et 1 n'ont plus de successeur, on ne descend plus. Puisque les attributs IdCFC et Position de ces deux nœuds gardent les même valeurs

(respectivement de chaque nœud), ces deux nœuds sont dépilés comme les CFCs triviales (lignes N°9 et N°10 du Tableau 33).

2. Une CFC de deux cycles – 2nde racine moins profonde

Le GDT dans la Figure 94 est le GDT de la Figure 93, ajouté un cycle (4 5 6). Le Tableau 34 présente une partie de la trace d'exécution de l'algorithme de Tarjan, appliqué à ce GDT. La structure et la construction de ce tableau sont identiques de celles de Tableau 33.

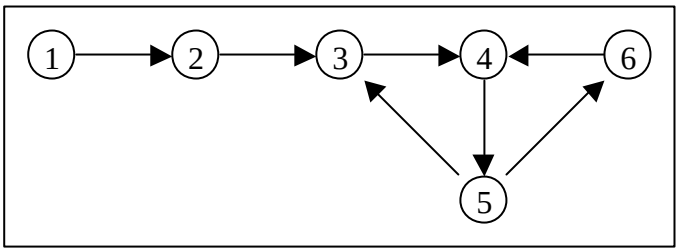

Figure 94. Décomposition des CFCs : Une CFC de deux cycles – la 2nde racine est moins profonde.

Puisque le comportement de l'algorithme de Tarjan pour ce GDT dans cinq première itérations est identique avec celui pour le GDT de la Figure 93, le Tableau 34 ne présente la trace d'exécution qu'à partir de la 5e exécution (ligne N°5 du Tableau 34).

Notons que le nœud 5 a deux successeurs et on a supposé que l'opérateur « *Elément* » (voir le Tableau 12) rend un nœud par l'ordre numérique, le nœud empilé suivant est donc le nœud 3 mais pas le nœud 6.

Après le premier recul au nœud 5 (ligne N°6 du Tableau 34), le recul dans le cycle (3 4 5 3) est interrompu. Puisque le nœud 5 a encore un successeur, le nœud 6, après la condition de la ligne 21, l'algorithme de Tarjan redescend (ligne N°7 du Tableau 34).

Le successeur 4 du nœud 6 existe dans la pile, on a détecté un autre cycle (4 5 6) dont la racine est le nœud 4 (ligne N°7 du Tableau 34). On recommence à reculer avec l'IdCFC du nœud 4 (ligne N°8 du Tableau 34). L'IdCFC du nœud ne change pas car il garde l'IdCFC de la racine la plus profonde (le nœud 3) parmi des racines des cycles (3 4 5) et (4 5 6) (les nœuds 3 et 4). Le recul continue dans le cycle (3 4 5) comme dans le Tableau 33 (lignes N°10 et N°11 du Tableau 34). Le dépilage donne la CFC {3 4 5 6}.

Tableau 34. Décomposition des CFCs : Algorithme de Tarjan applique au GDT de la Figure 94.

N°	Itération	Ligne	TP	p			s	Note
				Id	IdCFC	Position		
5	5	22	1, 2, 3, 4, 5	5	5	5	3	Trouver le cycle (3 4 5)
6	5	25	1, 2, 3, 4, 5	5	3	5	6	Commencer à reculer dans le cycle (3 4 5)
7	6	22	1, 2, 3, 4, 5, 6	6	6	6	4	Redescendre et trouver le cycle (4 5 6)
8	6	25	1, 2, 3, 4, 5, 6	6	4	6	ϕ	Commencer à reculer dans le cycle (4 5 6)
9	7	47	1, 2, 3, 4, 5, 6	5	3	5	ϕ	Reculer mais garder la racine la plus profonde
10	8	45	1, 2, 3, 4, 5, 6	4	3	4	ϕ	Reculer dans le cycle (3 4 5)
11	9	33	1, 2, 3, 4, 5, 6	3	3	3	ϕ	Arrêter le recul, sortir la CFC {3, 4, 5, 6}

Puisque le nœud 2 n'a plus de successeur, le recul continue comme les lignes de N°9 à N°11 du Tableau 33.

3. Une CFC de deux cycles – 2nde racine plus profonde

Dans le GDT de la Figure 95, le successeur du nœud 6 n'est plus le nœud 4 (comme dans le GDT de la Figure 94) mais le nœud 2. Le 2nd cycle trouvé est donc (2 3 4 5 6).

Le Tableau 35 présentent la trace de l'algorithme de Tarjan, appliqué au GDT de la Figure 95. La construction et la structure de ce tableau sont comme ceux du Tableau 34. La trace est aussi révélée à partir de la 5e itération.

Après avoir détecté le cycle (2 3 4 5 6), l'IdCFC du nœud 6 est mise à 2 – l'IdCFC de la racine du cycle (ligne N°8 du Tableau 35). L'IdCFC des nœuds 5, 4 et 3 est aussi mise à 2 (lignes N°9, N°10 et N°11 du Tableau 35) car le nœud 2 est la racine la plus profonde des cycles auxquels ces trois nœuds appartiennent.

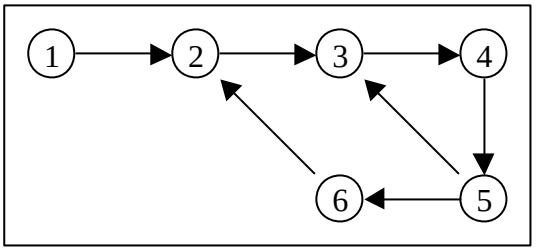

Figure 95. Décomposition des CFCs : Une CFC de deux cycles – la 2nde racine est plus profonde.

Tableau 35. Décomposition des CFCs : Algorithme de Tarjan applique au GDT de la Figure 95.

N°	Itération	Ligne	TP	p			s	Note
				Id	IdCFC	Position		
5	5	22	1, 2, 3, 4, 5	5	5	5	3	Trouver le cycle (3 4 5)
6	5	25	1, 2, 3, 4, 5	5	3	5	6	Commencer à reculer dans le cycle (3 4 5)
7	6	22	1, 2, 3, 4, 5, 6	6	6	6	2	Redescendre et trouver le cycle (2 3 4 5 6)
8	6	25	1, 2, 3, 4, 5, 6	6	2	6	ϕ	Commencer à reculer dans le cycle (2 3 4 5 6)
9	7	45	1, 2, 3, 4, 5, 6	5	2	5	ϕ	Reculer et prendre une nouvelle racine
10	8	45	1, 2, 3, 4, 5, 6	4	2	4	ϕ	Reculer dans le cycle (2 3 4 5 6)
11	9	45	1, 2, 3, 4, 5, 6	3	2	3	ϕ	
12	10	33	1, 2, 3, 4, 5, 6	2	2	2	ϕ	Arrêter le recul, sortir la CFC {3, 4, 5, 6}

4. Une CFC avec un cycle inclus dans l'autre

Dans le GDT de la Figure 96, le CFC {2 3 4 5} a deux cycles : (2 3 4 5) et (3 4 5) dont le 2nd cycle est inclus par le 1er cycle.

Le Tableau 36 présentent la trace de l'algorithme de Tarjan, appliqué au GDT de la Figure 96. La construction et la structure de ce tableau sont comme ceux du Tableau 34. La trace est aussi révélée à partir de la 5e itération.

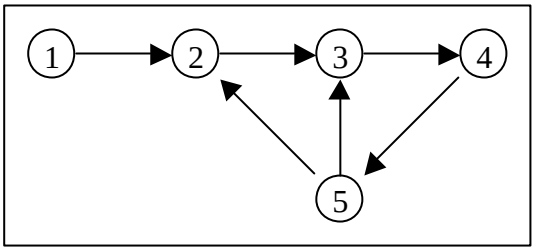

Figure 96. Décomposition des CFCs : Une CFC avec un cycle inclus dans l'autre.

La détection du cycle (3 4 5) ne change pas l'IdCFC des nœuds de la CFC (lignes N°7, N°6 et N°8 du Tableau 36) car la racine du 2nd cycle (le nœud 3) est moins profonde que celle du premier (le nœud 2).

Tableau 36. Décomposition des CFCs : Algorithme de Tarjan appliqué au GDT de la Figure 96.

N°	Itération	Ligne	TP	p			s	Note
				Id	IdCFC	Position		
5	5	22	1, 2, 3, 4, 5	5	5	5	2	Trouver le cycle (2 3 4 5)
6	5	25	1, 2, 3, 4, 5	5	2	5	3	Commencer à reculer dans le cycle (2 3 4 5) Trouver le cycle (3 4 5)
7	6	45	1, 2, 3, 4, 5, 6	4	2	4	ϕ	Recul dans le cycle (2 3 4 5 6)
8	7	45	1, 2, 3, 4, 5, 6	3	2	3	ϕ	
9	8	33	1, 2, 3, 4, 5, 6	2	2	2	ϕ	Arrêt de recul, sortir la CFC {3, 4, 5, 6}

Annexe III.

Illustration et amélioration de l'algorithme pour compter le nombre de cycles

Dans le paragraphe 1 de cette annexe, nous présentons la trace d'exécution de notre algorithme pour compter le nombre de cycles, appliqué à l'exemple de la CFC dans la Figure 65.

Dans le paragraphe 2, nous présentons les améliorations sur notre algorithme pour compter le nombre de cycles afin de diminuer le temps d'exécution de cet algorithme.

1. Illustration de algorithme pour compter le nombre de cycles

Le Tableau 37 présente la trace d'exécution de notre algorithme pour compter le nombre de cycles sur la CFC de la Figure 65. Dans ce tableau, la colonne « Itération » indique le nombre d'exécutions de la ligne 16 dans le texte de notre algorithme (Figure 64); la colonne « ligne » note une ligne dans la Figure 64, de laquelle la trace est révélée ; la colonne « p » note l'identification (ou bien le nom) du prédécesseur **p** dans notre algorithme et la colonne « s » – du successeur.

Tableau 37. Décomposition des CFCs : l'Illustration de l'algorithme de recherche des cycles d'une CFC.

N°	Itération	Ligne	Pile	p	s	s.EstExploité	Noté
1	0	10	ϕ				Début
2	1	17	1	1	2	F	
3	2	17	1, 2	2	3	F	
4	3	21	1, 2, 3	3	1	F	Trouver le cycle (1 2 3)
5	4	16	1, 2, 3	3	ϕ		
6	5	17	1, 2	2	4	F	
7	6	17	1, 2, 4	4	3	V	
8	7	21	1, 2, 4, 3	3	1	F	Trouver le cycle (1 2 4

							3)
9	8	16	1, 2, 4, 3	3	ϕ		
10	9	17	1, 2, 4	4	5	F	
11	10	17	1, 2, 4, 5	5	3	V	
12	11	21	1, 2, 4, 5, 3	3	1	F	Trouver le cycle (1 2 4 5 3)
13	12	16	1, 2, 4, 5, 3	3	ϕ		
14	13	21	1, 2, 4, 5	5	4	F	Trouver le cycle (4 5)
15	14	17	1, 2, 4, 5	5	6	F	
16	15	21	1, 2, 4, 5, 6	6	4	F	Trouver le cycle (4 5 6)
17	16	17	1, 2, 4, 5, 6	6	7	F	
18	17	17	1, 2, 4, 5, 6, 7	7	8	F	
19	18	21	1, 2, 4, 5, 6, 7, 8	8	6	F	Trouver le cycle (6 7 8)
20	19	16	1, 2, 4, 5, 6, 7, 8	8	ϕ		
21	19	30	1, 2, 4, 5, 6, 7	7	ϕ		
22	20	30	1, 2, 4, 5, 6	6	ϕ		
23	21	30	1, 2, 4, 5	5	ϕ		
24	22	30	1, 2, 4	4	ϕ		
25	23	30	1, 2	2	ϕ		
26	24	17	1	1	9	F	
27	25	17	1, 9	9	4	V	
28	26	17	1, 9, 4	4	3	V	
29	27	21	1, 9, 4, 3	3	1	F	Trouver le cycle (1 9 4 3)
30	28	16	1, 9, 4, 3	3	ϕ		
31	29	17	1, 9, 4	4	5	V	
32	30	17	1, 9, 4, 5	5	3	V	
33	31	21	1, 9, 4, 5, 3	3	1	F	Trouver le cycle (1 9 4 5 3)
34	32	16	1, 9, 4, 5, 3	3	ϕ		
35	33	23	1, 9, 4, 5	5	4	V	
36	34	17	1, 9, 4, 5	5	6	V	
37	35	23	1, 9, 4, 5, 6	6	4	V	

38	36	17	1, 9, 4, 5, 6	6	7	V	
39	37	17	1, 9, 4, 5, 6, 7	7	8	V	
40	38	17	1, 9, 4, 5, 6, 7, 8	8	6	V	
41	39	16	1, 9, 4, 5, 6, 7, 8	8	ϕ		
42	39	30	1, 9, 4, 5, 6, 7	7	ϕ		
43	40	30	1, 9, 4, 5, 6	6	ϕ		
44	41	30	1, 9, 4, 5	5	ϕ		
45	42	30	1, 9, 4	4	ϕ		
46	43	30	1, 9	9	ϕ		
47	44	30	1	1	ϕ		
48	44	34	ϕ				Fin

2. Eliminer des arcs « *isolés* »

Puisque notre algorithme doit trouver tous les cycles d'une CFC, il n'élimine pas des nœuds en reculant comme un DFS classique. Par conséquent, il y a des arcs qui sont visités plusieurs fois mais ces visites ne révèlent aucun cycle nouveau. Pour éliminer les visites inutiles, il faut détecter et éliminer ces arcs.

Par exemple, dans la Figure 97, quand on descend du nœud 9 au nœud 4, les arcs (5, 4), (5, 6) et tous les arcs des trois nœuds 6, 7 et 8 sont revisités mais on ne trouve plus de cycles nouveaux (lignes N°35, N°38 ..N°42 du Tableau 37). Ces arcs appartiennent aux cycles (4 5), (4 5 6) et (6 7 8) qui sont trouvés auparavant (lignes N°14 ..N°19 du Tableau 37).

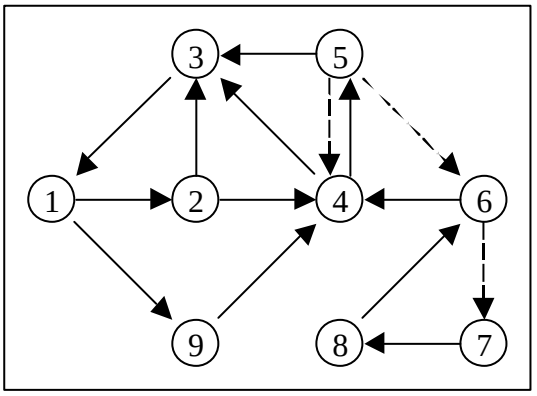

Figure 97. Décomposition des CFCs : l'élimination les arcs isolés.

Puisque tous les cycles auxquels ces arcs participent sont détectés dans la première visite de ces arcs et les visites ultérieures ne révèlent pas de cycles nouveaux, on trouve que ces arcs ont des attributs suivants :

i. Ils créent des suites d'arcs S_a :

$$S_a \leftarrow (a_1\ a_2 \ldots a_m) \mid \forall i = 1..m-1,\ a_i(p_i, s_i) \in A,\ (p_i, s_i) \in CFC^2,\ p_{i+1} \equiv s_i$$

Appelons N_{Sa} l'ensemble ordonné des extrémités de S_a, l'ordre des nœuds de N_{Sa} suit l'ordre des arcs correspondants dans S_a :

$$N_{Sa} \leftarrow \{n_1, n_2, \ldots, n_m\} \in CFC^m \mid \forall i = 1..m-1,\ \exists(n_i, n_{i+1}) \in S_a$$

ii. Il existe un parcours «*unique* » P_u $(p_1\ p_2 \ldots p_k)$, défini par Définition 26 dont le dernier nœud (le nœud p_k) est le premier nœud de N_{Sa} (le nœud n_1) et le premier nœud (le nœud p_1) est le dernier nœud de N_{Sa} (le nœud n_m) :

$$p_k \equiv n_1 \wedge p_1 \equiv n_m$$

Définition 26. *Un parcours «unique » P_u est une suite des nœuds $(p_1\ p_2 \ldots p_k)$ dont seulement le dernier nœud p_k qui a des arcs vers les nœuds qui ne sont pas de P_u :*

$$P_a \leftarrow (p_1\ p_2 \ldots p_k) \mid p_i \in CFC,\ \forall i = 1..k,$$

$$\forall j = 1..k-1,\ \exists(p_j, p_{j+1}) \in A,$$

$$\forall l = 1..k-1,\ \forall n \in CFC,\ n \notin P_u,\ ?\ (p_l, n) \in A$$

Appelons S_u la suite d'arcs qui relient les nœuds de P_u :

$$S_a \leftarrow (u_1\ u_2 \ldots u_m) \mid \forall i = 1..m-1,\ u_i(p_i, s_i) \in A,\ (p_i, s_i) \in P^2{}_u,\ p_{i+1} \equiv s_i$$

iii. Le successeur de tous les arcs qui sortent d'un nœud de N_{Sa} est de $(N_{Sa} \cup P_u)$:

$$\forall(n_i, n) \in A,\ n_i \in N_{Sa},\ i = 1..\ m,\ n \in (N_{Sa} \cup P_u)$$

Autrement dit, les arcs de S_a et de S_u créent un cycle (attributs i et ii). Les nœuds de N_{Sa} ne peuvent pas créer des cycles sans les arcs de S_u car tous leurs successeurs sont de $(N_{Sa} \cup P_u)$ – (attribut iii).

Par le Propriété 2, après avoir dépilé le premier nœud de P_u, on peut éliminer les arcs de S_a. En pratique, l'élimination du premier arc de S_a suffira car cette élimination empêche tous les accès aux autres arcs de S_a.

Par exemple, pour le cycle (4 5) dans la Figure 97, N_{Sa} est {5, 4} ; P_u est {4, 5} ; l'arc éliminé sera l'arc (5, 4). Pour le cycle (6 7 8), N_{Sa} est {6, 7, 8} ; P_u est {6} ; l'arc éliminé sera l'arc (6, 7). Notons que l'élimination de l'arc (6, 7) entraîne l'élimination des arcs (7, 8) et (8, 6).

Notons aussi qu'après avoir éliminé des arcs, il y a des arcs qui s'affilient à l'ensemble d'arcs, causés des accès inutiles. Par exemple, avant d'éliminer les arcs (6, 7), (7, 8) et (8, 6), les arcs du cycle (5 6 4) ne satisfont pas les attributs i, ii et iii : le nœud 6 a un arc vers le nœud 7 qui n'est pas un nœud du cycle (5 6 4) ; quand les arcs (6, 7), (7, 8) et (8, 6) sont éliminés, les arcs du cycle (5 6 4) peuvent satisfaire ces attributs. Dans ce cas, $P_u = \{4, 5\}$, $N_{Sa} = \{5, 6, 7\}$; l'arc éliminé sera l'arc (5, 6).

Annexe IV.

La structure de l'outil TestPlan

Dans cette annexe, nous présentons les diagrammes de classes UML de notre outil TestPlan. Cet outil est construit pour implémenter les idées qu'on a présentées dans ce mémoire.

1. Les packages

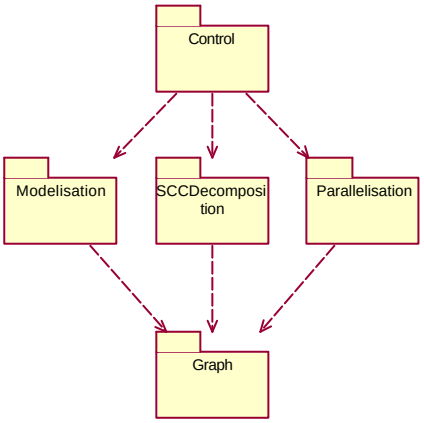

Figure 98. TestPlan : les packages.

2. La bibliothèque de graphe

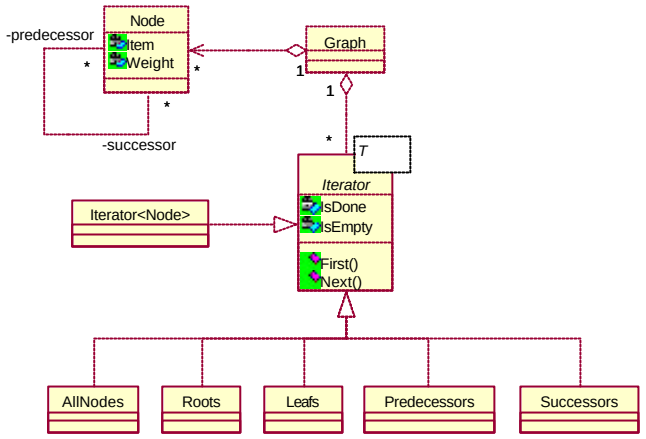

Figure 99. TestPlan : la bibliothèque de graphe.

3. La modélisation

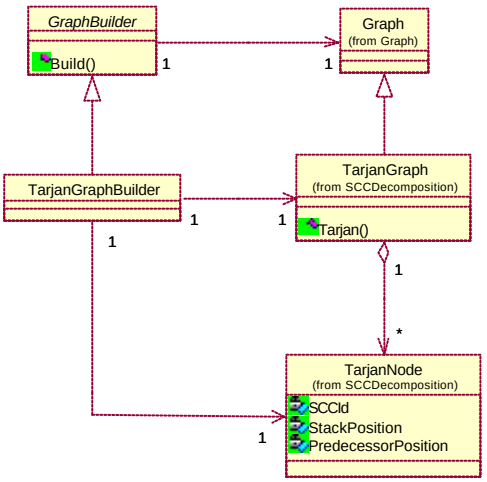

Figure 100. TestPlan : la modélisation.

4. La décomposition des CFCs

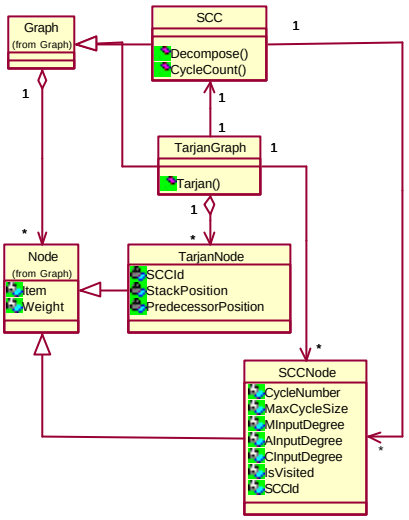

Figure 101. TestPlan : la décomposition de CFCs.

5. La parrallélisation

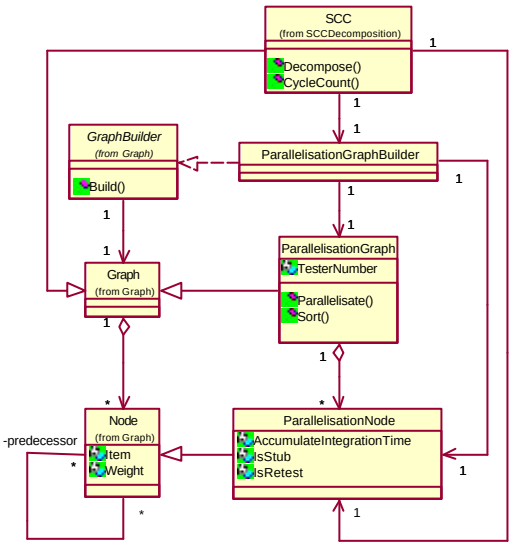

Figure 102. TestPlan : la parallélisation.

6. Le corps principal

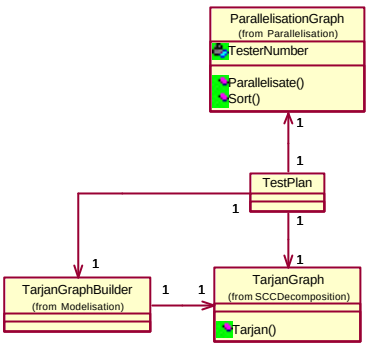

Figure 103. TestPlan : le corps principal.

Annexe V.

Mode d'emploi de l'outil TestPlan

Cet outil est développé pour expérimenter.

Syntaxe :

```
TestPlan   <fichier entré >
            [-G|-g][-S|-s][-N|-n]
            [-t nombre_de_testeurs].
            [-c seuil_de_complexité].
```

Fichier entré :

Le fichier entré est un fichier qui garde les données entrées pour cet outil. La structure de ce fichier est montrée dans la Figure 104.

```
[Names and Weights]
name_1 weight_1
…
name_n weight_n

[Edges]
Order_1 Order_2 Edge_type
…
Order_i Order_j Edge_type
```

Figure 104. La structure du fichier entré pour l'outil TestPlan.

Le fichier est commencé par une ligne de clé :

[Names and Weights]

Cette ligne annonce que les lignes suivantes sont les noms et les poids correspondants des unités du programme qu'on veut planifier. Toutes les lignes avant de cette ligne sont ignorées. Après cette ligne, une ligne suivante soit une ligne de

déclaration de nom et poids, soit une autre ligne clé qui fait arrêter la liste des noms et poids. L'outil va ignorer toutes les lignes qui ne conforment pas au format d'une de ces deux types de lignes. Une ligne de déclaration de nom et poids a un format :

`name weight`.

où :

- `name` : une chaîne (STRING), nommé une unité, suivi par une espace.
- `weight` : un nombre entier (INTEGER), annoncé le poids de l'unité.

La liste des noms et des poids des unités s'arrête à une autre ligne clé. C'est :

`[Edges]`.

Cette ligne annonce aussi que les lignes suivantes sont les dépendances d'intégration entre les unités du programme sous test. Après cette ligne, l'outil va ignorer aussi toutes les lignes qui ne conforment pas au format d'une ligne de dépendance. Le format d'une ligne de dépendance est :

`Order_1 Order_2 Edge_type`

où :

- `Order_1` : un nombre entier (INTEGER), suivi par une espace, correspond à l'ordre de prédécesseur dans la liste des noms et des poids.
- `Order_1` : un nombre entier (INTEGER), suivi par une espace, correspond à l'ordre de successeur dans la liste des noms et des poids.
- `Edge_type` : Un des quatre valeurs :
    1. Une dépendance de type « héritage ». Cette dépendance n'est jamais simulée pour créer un bouchon du test d'intégration.
    2. Une dépendance de type « composition ». Cette dépendance est le troisième choix pour créer un bouchon du test d'intégration.
    3. Une dépendance de type « association ». Cette dépendance est le second choix pour créer un bouchon du test d'intégration.
    4. Une dépendance de type «méthode ». Cette dépendance est le premier choix pour créer un bouchon du test d'intégration.

Si l'ordre d'un prédécesseur ou d'un successeur dépasse le nombre d'unités dans la liste des noms et des poids, cet ordre va être considéré comme un nom d'une nouvelle unité et le poids de cette unité est Zéro.

Options :

`-G|-g`	Afficher ou non le GDT.
	La valeur défaut est -g

-S|-s	Afficher ou non les CFCs. La valeur défaut est -c
-N|-n	Afficher ou non les paramètres de nœuds des CFCs. La valeur défaut est -n
-t nombre_de_testeurs :	Un nombre de testeurs est un nombre entier et positif. La valeur défaut est 1.
-c seuil_de_complexité :	Un seuil de complexité est un nombre entier et positif. La complexité totale d'un groupe des unités qui vont être intégré ensemble ne doit pas dépasser ce seuil. La valeur défaut est 0.

Exemples :

Planifier l'intégration d'un système dont la structure est stockée dans le fichier « toto » :

TestPlan toto.

Planifier l'intégration par deux testeurs d'un système dont la structure est stockée dans le fichier « titi ». Les testeurs peuvent gérer un groupe des unités dont la complexité ne dépasse pas 10 :

TestPlan titi –t 2 –c 10.

Annexe VI.

Module TestStrategy d'Objecteering

La société Softeam a intégré une partie de notre stratégie par le module TestStrategy dans son produit Objecteering, un outil de modélisation UML. Ce module est créé par Clémentine Nébut.

Dans le premier paragraphe de cette annexe, nous présentons brièvement l'outil Objecteering. Dans le deuxième paragraphe, nous allons montrer comment on peut obtenir un plan d'intégration à partir d'un diagramme de classes UML, grâce au module TestStrategy.

1. Objecteering

Objecteering est un outil de type «Atelier de Génie Logiciel» (AGL), c'est-à-dire, un outil pour faire la modélisation des logiciels. Plus qu'un simple outil de dessin, Objecteering assure une cohérence au sein d'un modèle établi par l'utilisateur. Il offre aussi la possibilité d'engendrer du code à partir du modèle dans un langage objet cible (C++, Java, VB…), ainsi que d'opérer des transformations automatiques de modèle.

L'idée principale d'un AGL comme Objecteering est de proposer au concepteur un moyen de décrire la solution à un problème de manière graphique, cohérente et formelle. Objecteering fournit tous les diagrammes qu'un modèle UML demande : le diagramme de classes, le diagramme de séquences, le diagramme d'états... Il contrôle au cours de la modélisation sa validité, grâce à plus de 200 contrôles en vol, et 80 métriques. Les contrôles portent par exemple sur l'absence de dépendances cycliques entre packages, l'absence de membre abstrait dans une classe non-abstraite ou l'unicité des attributs et des liens d'héritage d'une classe…

La fonction « UML Profile Builder » d'Objecteering permet de créer des profils, une manière de personnaliser et d'élargir Objecteering pour les cibles concrets. Par exemple, l'OMG standardise des profils UML pour le temps réel, pour les applications distribuées, pour les composants CORBA… Chaque profil est un module qui définit lui-même en UML par les méta-classes. C'est avec cette fonction que notre idée est implémentée dans Objecteering par le profil TestStrategy.

2. Module TestStrategy

Dans Objecteering, les profils sont organisés en hiérarchie comme dans la Figure 105. Les méta-classes du module TestStrategy sont aussi présentées.

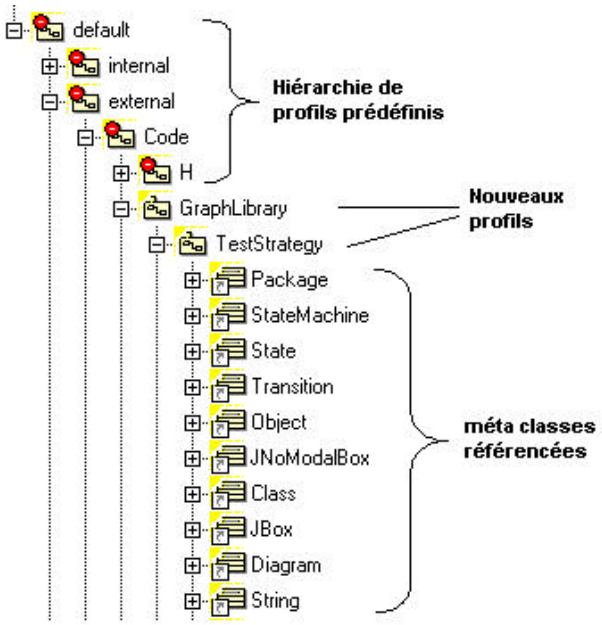

Figure 105. Objecteering : les profils UML.

UMLProfileBuilder d'Objecteering fournit un mécanisme pour personnaliser les réactions de profil. Dans la Figure 106, les paramètres sont présentés avec une clé à molette. Avec le module TestStrategy, on peut personnaliser plusieurs paramètres comme si on exploitait les paramètres de méthodes (use operation) l'élimination des arcs parallèles (arcs transitifs) après avoir décomposé des CFCs (transitive reduction on ordering diagram)…

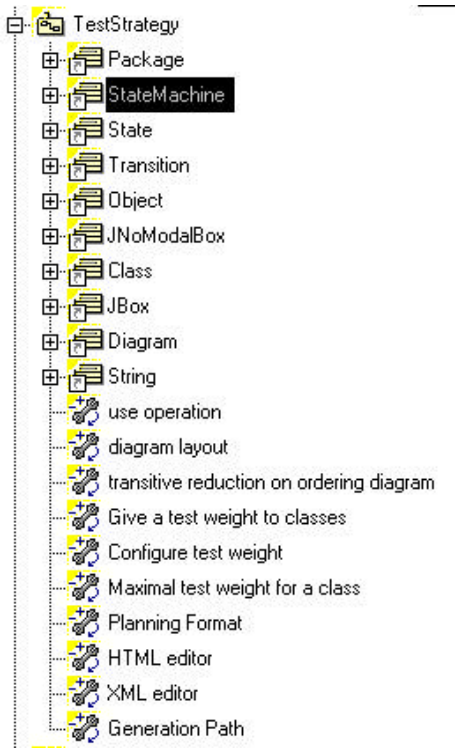

Figure 106. Objecteering – TestStrategy : les paramètres de module.

Ces paramètres sont changés par les fenêtres comme dans la Figure 107.

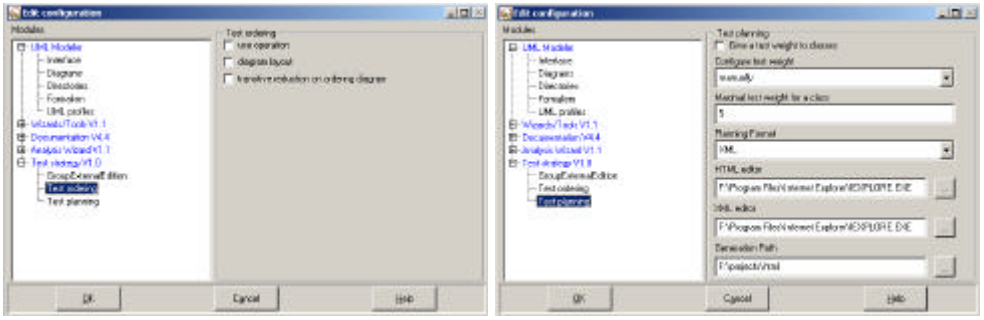

Figure 107. Objecteering – TestStrategy : Configuration du module de stratégie de test.

Depuis un diagramme de classes comme celui dans la Figure 108, La fonction « Get test ordering » (voir la Figure 109).

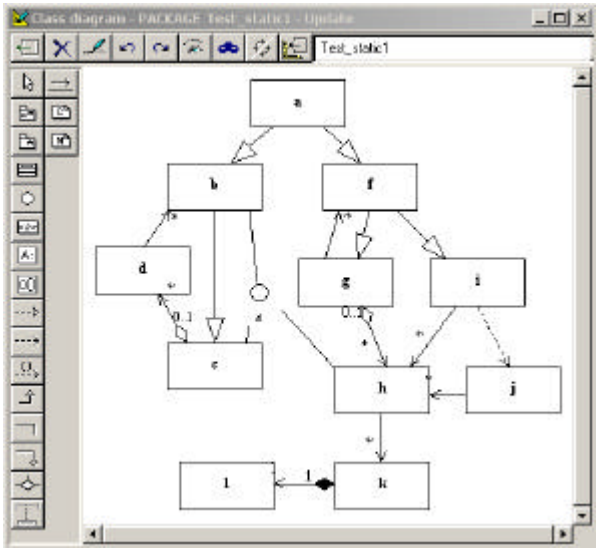

Figure 108. Objecteering – TestStrategy : la présentation de GDT.

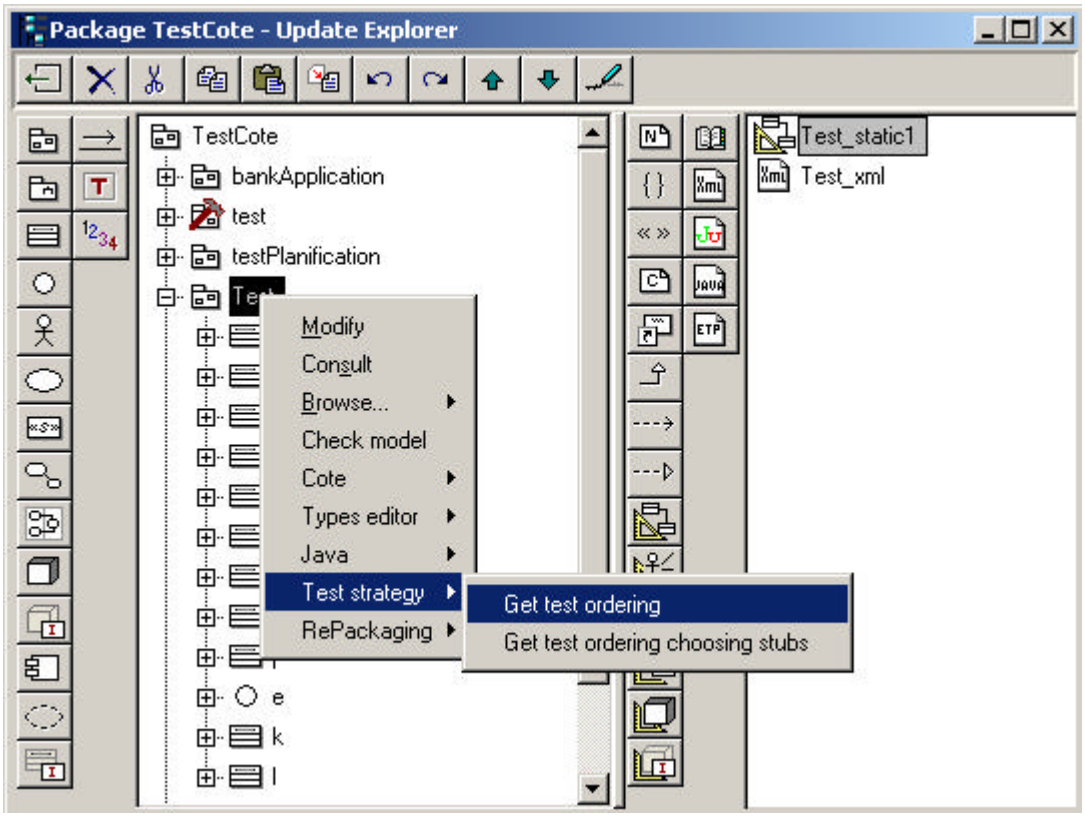

Figure 109. Objecteering – TestStrategy : Déclencher la décomposition.

Le module TestStrategy ne donne pas le GDT acyclique mais il donne directement un graphe d'ordonnancement qui présente la relation « avant – après » des nœuds (voir la Figure 110). Dans ce graphe, un arc orienté d'un nœud *a* à un nœud *b* montre que le nœud *a* va être testé avant le nœud *b*. Les nœuds qu'on doit simuler sont présentés en couleur rouge.

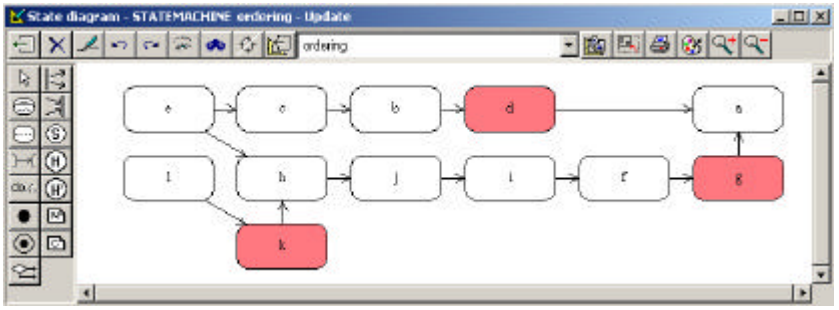

Figure 110. Objecteering – TestStrategy : le graphe d'ordonnancement.

On peut choisir les nœuds qu'on veut simuler par la fonction « Get test ordering choosing stubs » (voir la Figure 109). Quand on choisit cette fonction, la fenêtre pour choisir des bouchons s'ouvre (voir la Figure 111). Tous les arcs qui entrent dans ces nœuds sont supprimés avant la phase de décomposition des CFCs.

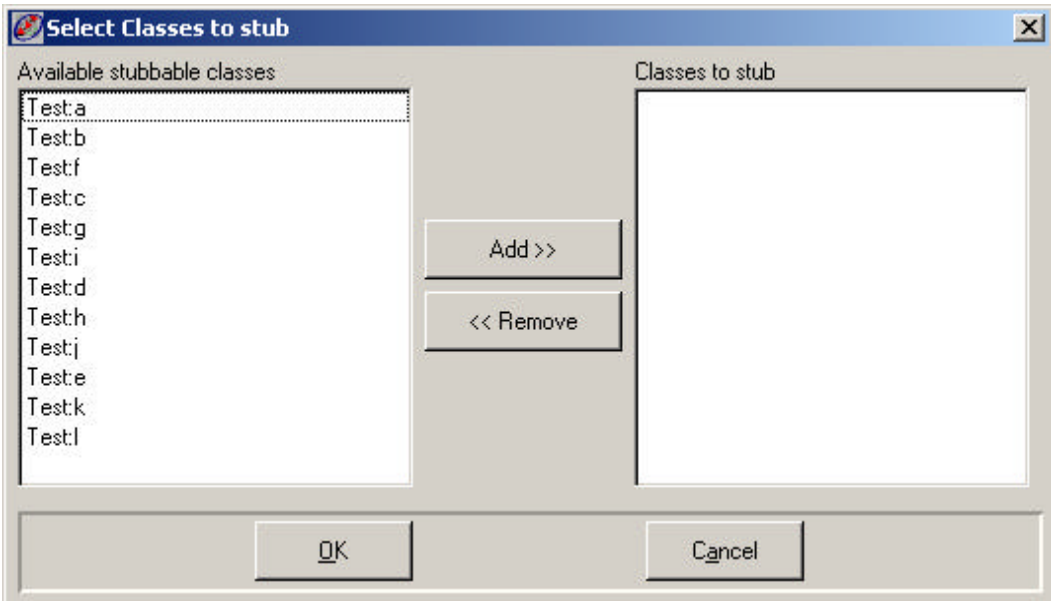

Figure 111. Objecteering – TestStrategy : Sélection des classes à simuler.

Après voir obtenu le graphe d'ordonnancement (N° 1 dans la Figure 112), le plan de test est généré par la fonction «Get test Planning » (N° 2 dans la Figure 112). Cette fonction va demander le nombre de testeurs (voir la Figure 113) Ce plan se change en fonction de nombre de testeurs. Par exemple, avec deux testeurs, le plan de test est comme dans la Figure 114.

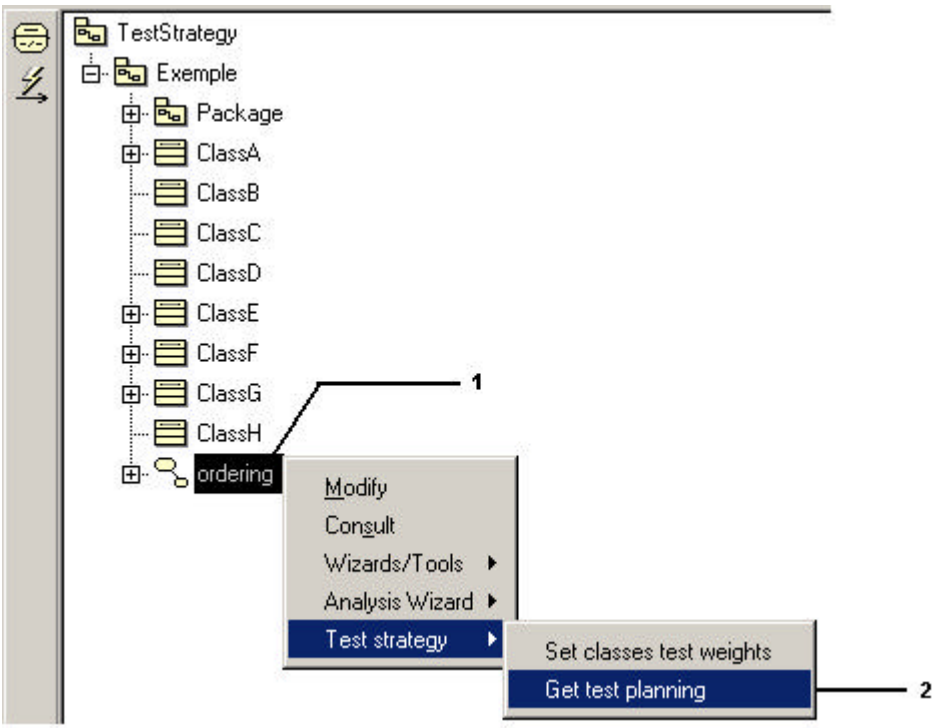

Figure 112. Objecteering – TestStrategy : Obtenir le plan de test d'intégration.

Figure 113. Objecteering – TestStrategy : Saisir le nombre de testeurs.

Step	Stubs	Tester 1	Tester 2	Retest
Step 1	k	l		
Step 2		k	e	l
Step 3	g	h		
Step 4		j		
Step 5	d	i	c	
Step 6		f	b	
Step 7		g	d	f h c
Step 8		a		

Figure 114. Objecteering – TestStrategy : Plan de test.

Bibliographie

[Arnold 1994] T.R. Arnold, W.A. Fuson
Testing in "A Perfect World"
pp 78-96, *Communications of the ACM*, vol. 37, no. 9, 09/1994.

[Barbey 1996] S. Barbey, D. Buchs, C. Péraire
A Theory of Specification-Based Testing for Object-Oriented Software
pp 303-320, Proc. of the 2nd European Dependable Computing Conference, Italia, LNCS, no. 1150, 10/1996

[Barbey 1997] S. Barbey
Test Selection for Specification-Based Testing of Object-Oriented Software Based on Formal Specification
Rapport de these, Ecole Polytechnique Fédéral de Lausanne, 1997.

[Barbey 1999] S. Barbey, D. Buchs, C. Péraire, A.Strohmeier
Incremental Test Selection for Specification Based Unit Testing of Object-Oriented Software Based on Formal Specification
Rapport EPFL-DI99/303, Ecole Polytechnique Fédéral de Lausanne, 1999.

[Bashir 1994] I. Bashir, A Goel
Testing C++ Classes
pp 43-48, Proc. of the International Conference on Software Testing, Reliability and Quality Assurance, 12/1994.

[Baudry 2000 #1] B. Baudry, H. Vu Le, J.M. Jézéquel, Y. Le Traon
Building Trust into OO Components using a Genetic Analogy
pp 4-14, Proc. of the 11th International Symposium on Software Reliability Engineering – ISSRE'2K, 10/2000.

[Baudry 2000 #2] B. Baudry, H. Vu Le, Y. Le Traon
Testing-for-Trust: the Genetic Selection Model Applied to Component Qualification
pp 108-119, Proc. of the Technology of Object-Oriented Languages and Systems Conference 2000 – TOOLS Europe' 2000, 06/2000.

[Beizer 1990] B. Beizer
Software testing techniques
Van Norstrand Reinhold, New York, 2nd Edition, ISBN 0-442-20672-0, 1990.

[Bernot 1991] G. Bernot, M.C. Gaudel, B. Marre
Software testing based on formal specification: a theory and a tool
pp 387-405, *Software Engineering Journal*, no. 6, 1991.

[Binder 1994] R.V. Binder
Design for Testability with Object-Oriented Systems
pp 87-101, *Communications of the ACM*, vol. 37, no. 9, 09/1994.

[Binder 1996] R.V. Binder
Testing Object-Oriented Software: a Survey

pp 125-252, *Journal of Software Testing, Verification and Reliability*, no. 6, 1996.

[Binder 1999] R.V. Binder
Testing Object-Oriented Systems Models, Patterns and Tools
Addison Wesley, ISBN 0-201-80938-9, 1999.

[Booch 1994] G. Booch
Analyse et conception orientées-objets
Addison-Wesley, ISBN 2-87908-609-X, 1994.

[Booch 1998] G. Booch, J. Rumbaugh, I. Jacobson
The Unified Modeling Language User Guide
Addison-Wesley, ISBN 0-201-57168-4, 1999.

[Bourdoncle 1993] F. Bourdoncle
Efficient Chaotic Iteration Strategies with Widenings,
pp128-141, Proc. of the International Conference on Formal Methods in Programming and their Applications, LNCS 735, Springer-Verlag, ISSN 0302- 9743, 1993

[Briand 2001] L.C. Briand, Y. Labiche, Y. Wang
Revisiting Strategies for Ordering Class Integration Testing in the Presence of Dependency Cycles
pp 287-296, Proc. of the 12th IEEE International Symposium on Software Reliability Engineering, Hong Kong, ISBN 0-7695-1306-9, 11/2001.

[Cavalli 1996] A.R. Cavalli, B.M. Chin, K.Chon
Testing Method for SDL Systems
pp 1669-1683, *Computer Network and ISDL Systems*, vol. 28, 1996.

[Chang 1998] K.H. Chang, S.S. Liao, S.B. Seidman, R. Chapman
Testing Object-Oriented Programs: From Formal Specification To Test Scenario Generation
pp 141-151, *Journal of System and Software*, no. 42, 1998.

[Chaudhuri 1992] P. Chaudhuri
Parallel algorithms design and analysis
Prentice-Hall, ISBN 0-13-351982-1, 1992

[Cheatham 1990] J. Cheatham, L.Mellinger
Testing Object-Oriented Software Systems
pp 161-165, Proc. of the 18th Annual Computer Science Conference, New York, USA, 02/1990.

[ChenP 1976] P.P.S. Chen
The Entity-Relationship Model – Toward a Unified View of Data
pp 9-36, *ACM Transactions on Database Systems*, vol. 1, no. 1, 1976.

[ChenW 1999] W.C. Chen, D.J. Cheng, C.G. Chung
An Expert-System Technique for Object-Oriented Program Testing
pp 25-35, 44, *Journal of Object Oriented Programming*, 02/1999.

[Chow 1978] T.S. Chow
Testing Software Desing Modeled by Finite-State Machines
pp 178-187, IEEE Transactions on Software Engineering, vol. SE-4, no. 3, 1978.

[Chung 1997] C. M. Chung, T. K. Shih, C. C. Wang, M. C. Lee
Integration Object-Oriented Software Testing and Metrics
pp 125-144, *International Journal of Software Engineering and Knowledge Engineering*, vol. 7, no. 1, 1997.

[Coleman 1992] D. Coleman, F. Hayes, S. Bear
Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design
pp 9-18, *IEEE Transactions on Software Engineering*, vol. 18, 1992.

[Cormen 1994] T. Cormen, C.Leiserson, R. Rivest
Introduction à l'algorithmique
Dunod, ISBN 210-003128-7, 1994

[Daly 1996] J. Daly, A. Brooks, J. Miller, M. Roper, M. Wood
Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software
pp 109-132, *Journal of Empirical Software Engineering*, vol. 1, no. 2, 1996.

[Deo 1974] N. Deo
Graph Theory With Application to Engineering and Computer Science
Prentice Hall, ISBN 0-13-363473-6, 1974.

[Deveaux 2000] D. Deveaux, R. Fleurquin, P. Frison, J.-M. Jézéquel, and Y. Le Traon.
Composants objets fiables : une approche pragmatique
pp469—494, *L'objet*, No. 5 , Vol 3--4:, 04/2000

[Dick 1995] J. Dick, A. Faivre
Automating the Generation and Sequencing of Test Cases from Model-Based Specification
pp 268-284, Proc. of the Formal Method Europe in 1995 – FME'95, 1995.

[Diestel 1997] R. Diestel
Graph theory
Springer, ISBN : 0-387-98210-8, 1997

[Doong 1994] R. K. Doong, P. G. Frankl
The ASTOOT Approach to Testing Object-Oriented Programs
pp 101-130, *ACM Transactions on Software Engineering and Methodology*, vol. 3, no. 2, 1994.

[Duncan 1999] I. Duncan, D. Robson, M. Munro
Test-Case Development During Object-Oriented Lifecycle and Evolution
pp 36-40, 44, *Journal of Object Oriented Programming*, 02/1999.

[Fernandez 1999] J. L. Fernandez
Acceptance Testing of Object-Oriented Systems
pp 114-123, LNCS, no. 1622, 1999.

[Fiedler 1989] S.P. Fiedler
Object-Oriented Unit Testing
pp 69-74, *Hewlett-Packard Journal*, 1989.

[Fletcher 1996] R. Fletcher, A. S. M. Sajeev
A Framework for Testing Object-Oriented Software Using Format Specification
pp 159-170, LNCS, no. 1088, 1996.

[Gamma 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides
Design Pattern – Element of Reusable Object-Oriented Software
Addison-Wesley, ISBN 0-201-63361-2, 1995.

[Fowler 1999] M. Fowler, K. Scott
UML distilled, a Brief Guide to the Standard Object Modeling Language
Addison-Wesley, 2nd Edition, ISBN 0-201-65763-X, 1999.

[Gondran 1985] M. Gondran, M. Minoux
Graphes et Algorithmes
Eyrolles , ISSN 0399-4198, 1985.

[Harrold 1992] M. J. Harrold, J. D. McGregor,, K. J. Fitzpatrick
Incremental Testing of Object-Oriented Class Structures
pp 68-80, Proc. of 14th IEEE International Conference on Software Engineering – ICSE-14, Melbourne, Australia, 05/1992.

[Harrold 1994] M. J. Harrold, G. Rothermel
Performing Data Flow Testing on Classes
pp 154-163, Proc. of the 2nd ACM SIGSOFT Symposium on the Fondations of Software Engineering, 12/1994.

[Hayes 1994] J. H. Hayes
Testing of Object-Oriented Programming Systems (OOPS): A fault-Based Approach
pp 205-220, LNCS, no. 858, 1994.

[Hierons 1997] R.M. Hierons
Testing from Z Specification
pp 19-33, *Journal of Software Testing, Verification and Reliability*, vol. 7, 1997.

[Hill 1993] D.R.C. Hill
Analyse orientée objet et modélisation par simulation
Addison-Wesley, ISBN 2-87908-051-7, 1993.

[Hoffman 1997] D. Hoffman, P. Strooper
ClassBench: a Framework for Automated Class Testing
pp 573-597, *Software practice and Experience*, vol. 27, no. 5, 1997.

[Hong 1995] H.S. Hong, Y.R. Kwon, S.D. Cha
Testing of Object-Oriented Programs Based on Finite State Machines
pp 234-241, Proc. of the Asia-Pacific Software Engineering Conference in 1995 – APSEC'95, Brisbane, Australia, 12/1995.

[Hunt 1995] M. Hunt
Automatically tracking test case execution
pp 22-27, *Journal of Object Oriented Programming*, 11-12/1995.

[Ishida 1994] T. Ishida
Parallel, distributed and multiagent production systems
Springer, ISBN 3-540-58698-9, 1994

[Jéron 1999] T. Jéron, J.M. Jézéquel, Y. Le Traon,, P. Morel
Efficient Strategies for Integration and Non-regression Testing of OO Systems
pp 260-269, Proc. of the 10th International Symposium on Software Reliability Engineering – ISSRE'99, Boca Raton, Florida, 11/1999.

[Jézéquel 1996] J.M. Jézéquel
Object Oriented Software Engineering with Eiffel
Addison-Wesley, ISBN 1-201-63381-7, 03/1996.

[Jézéquel 1997] J.M. Jézéquel, B. Meyer
Design by Contract: The Lessons of Arian
pp 129-130, *IEEE Computer*, vol. 30, no. 2, 01/1997.

[Jorgensen 1994] P. C. Jorgensen, C. Erickson
Object-Oriented Integration Testing
pp 30-38, *Communications of the ACM*, vol. 37, no. 9, 09/1994.

[Kumar 1994] V. Kumar
Introduction to parallel computing design and analysis of algorithms
Benjamin Cummings, ISBN 0-8053-3170-0, 1994

[Kung 1993] D. C. Kung, N. Suchak, J. Gao, P. Hsia, Y. Toyoshima, C. Chen
On Object State Testing
pp 222-227, Proc. of the 18th Annual International Computer Software & Applications Conference, IEEE Computer Society Press, Los Alamitos, California, 11/1993.

[Kung 1995 #1] D. C. Kung, J. Gao, P. Hsia, J. Lin, Y. Toyoshima
Class Firewall, Test Order and Regression Testing of Object-Oriented Programs
pp 51-65, *Journal of Object Oriented Programming*, vol. 8, no. 2, 05/1995.

[Kung 1995 #2] D. C. Kung, J. Gao, P. Hsia
A Test Strategy for Object-Oriented Programs
pp 239-244, Proc. of the 19th Computer Software and Applications Conference, Dallas, Texas, 9-11/08/1995.

[Kung 1995 #3] D. C. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, Y. S. Kim, Y. S. Song
Developing an Object-Oriented Software Testing and Maintenance Environment
pp 75-87, *Communications of the ACM*, vol. 58, no. 10, 10/1995.

[Kung 1996] D. C. Kung, J. Gao, Jerry, Chen, Cris
On Regression Testing of Object-Oriented Programs
pp 31-40, The *Journal of Systems and Software*, vol. 32, no. 1, 01/1996.

[Labiche 2000 #1] Y. Labiche,
Construction au test des logiciels orientés-objet : Ordre de test, modèle et critères associés
Rapport de thèse no. 00440, LAAS-CNRS, 2000.

[Labiche 2000 #2] Y. Labiche, P. Thévenod-Fosse, H Waeselynck, M. H. Durand
Testing Level for Object-Oriented Software
pp 136-145, Proc. of 22nd International Conference on Software Engineering, Limerick, Ireland, 06/2000.

[Leighton 1992] F.T.Leighton
Introduction to parallel algorithms and architectures
Morgan Kaufmann, ISBN 1-558-60117-1, 1992

[LeTraon 1999] Y. Le Traon, D. Deveaux, J. M. Jézéquel
Self-testable components: from pragmatic tests to a design-for-testability methodology
pp 96-107, Proc. of the Technology of Object-Oriented Languages and Systems Conference in 1999 – TOOLS-Europe'99, 06/1999.

[LeTraon 2000] Y. Le Traon, T. Jéron, J. M. Jézéquel, P. Morel
Efficient OO Integration and Regression Testing
pp 12-25, *IEEE Transactions on Reliability*, 03/2000.

[McCabe 1994] T. J. McCabe, A. H. Watson
Combining Comprehension and Testing in Object-Oriented Development
pp 63-67, *Object Magazine*, 03-04/1994.

[McGregor 1993] J. D. McGregor, D.M. Dyer
A Note on Inheritance and State Machines
pp 61-69, *ACM SIGSOFT Software Engineering Notes*, vol. 18, no. 4, 1993.

[McGregor 1994 #1] J. D. McGregor
Constructing Functional Test Cases Using Incrementally Derived State Machines
Proc. of the 11th International Conference on Testing Computer Software, USPDI, Washington DC, 13-16/06/1994.

[McGregor 1994 #2] J. D. McGregor
Functional Testing Of Class
Proc. of the 7th International Software Quality Week, Software Research Institute, San Francisco, 051994.

[McGregor 1994 #3] J. D. McGregor, T. Korson
Integrating Object-Oriented Testing and Development Processes
pp 59-77, *Communications of the ACM*, vol. 37, no. 9, 09/1994.

[McGregor 1997 #1] J. D. McGregor
Component testing
pp 6-9, *Journal of Object-Oriented Programming*, vol.9, no.3, 1997.

[McGregor 1997 #2] J. D. McGregor
A component testing method
pp 5-9, *Journal of Object-Oriented Programming*, vol.9, no.5, 1997.

[McGregor 1997 #3] J. D. McGregor
Testing from Specification
pp 6-10, *Journal of Object-Oriented Programming*, vol.9, no.8, 1997.

[McGregor 2001] J. D. McGregor, D.A. Sykes,
A Practical Guide to Testing Object-Oriented Software
Addison-Wesley, ISBN 0-201-32564-0, 2001.

[Mercier 1998] F. Mercier, A. Bertolino, P. Le Gall, G. Bernot
A systematic approach for integration testing of complex systems
Proc. of the 11th International Conference on Software and System Engineering and their Applications in 1998 – ICSSEA98, Paris, France, 12/1998.

[Meyer 1991] B. Meyer
Conception et Programmation par Objets pour du logiciel de qualité
InterEdition, ISBN 2-7296-0272-0, 1990.

[Muller 1997] P. A. Muller
Modélisation objet avec UML
Eyrolles, 1997, ISBN 2-212-08966-X.

[Müllerburg 1990] M. Müllerburg
Le test du logiciel : un processus incrémental
pp 6-17, *Génie logiciel et Système Experts*, vol. 18, 1990.

[Myers 1979] G.J. Myers
The art of software testing
John Wiley and Sons, ISBN 0-471-04328-1, 1979.

[Murphy 1994] G. C. Murphy, P. Townsend, P. S. Wong
Experience with Cluster and Class
pp 39-47, *Communications of the ACM*, vol. 37, no. 9, 09/1994.

[Oleinick 1982] P.N. Oleinick
Parallel algorithms on a multiprocessor
UMI Research Press, ISBN 0-8357-1327-X, 1982

[Overbeck 1994] J. Overbeck
Testing Generic Classes
pp 41/1-41/11, Proc. of the European International Conference on Software Testing Analysus and Review in 1994 – EuroSTAR'94, 10/1994.

[Panas 1979] D.L. Panas
Designing software for ease extension and contraction
pp 128-138, *IEEE Transaction on Software Engineering*, vol. 5, no. 2, 1979.

[Perry 1990] D.E. Perry, G.E. Kaiser
Adequate testing and Object-Oriented Programming
pp 13-19, *Journal of Object-Oriented Programming*, vol. 2, no. 5, 1990.

[Poston 1994] R. M. Poston
Automated Testing from Object Model
pp 48-58, *Communication of the ACM*, vol. 37, no. 9, 09/1994.

[Ponder 1994] C. Ponder, B, Bush
Polymorphism Considered Harmful
pp 35-37, *Software Engineering Notes*, vol. 19, no. 2, 1994.

[Rapps 1985] S. Rapps, E. J. Weyuker
Selecting Software Test Data Using Data Flow Information
pp 367-375, *IEEE Transactions on Software Engineering*, vol. SE-11, 1985.

[Raynaud 1990] B. Raynaud, L. Saint-Gealine
Une approche formelle pour la validation du logiciel
pp 699-712, Proc. of 3e Journées Internationales : le génie logiciel et ses applications, Toulouse, France, 12/1990.

[Rumbaugh 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen
Object-Oriented Modeling and Design
Prentice-Hall, ISBN 0-13-630054-5, 1991.

[Rumbaugh 1998] J. Rumbaugh, I. Jacobson, G. Booch
The Unified Modeling Language Reference Guide
Addison-Wesley, ISBN 0-201-3099-8, 1998.

[Shütz 1993] W. Shütz
The testability of distributed real-time systems
Kluwer Academic Publishers, ISBN 0-7923-9386-4, 1993.

[Simon 1988] K. Simon
An improved algorithm for transitive closure on acyclic digraphs
pp 325-346, *Theoretical Computer Science*, vol. 58, 1988.

[Tai-Daniels 1999] K. C. Tai, F. J. Daniels
Interclass Test Order for Object-Oriented Software
pp 18-35, *Journal of Object Oriented Programming*, 07-08/1999.

[Tarjan 1972] R. E. Tarjan
Depth-first search and linear graph algorithms
pp 146-160, *SIAM Journal of Computer*., vol.1, no. 2, 06/1972.

[Thévenod-Fosse 1997] P. Thévenod-Fosse, H..Waeselynck
Towards a Statistical Approach to Testing Object-Oriented Programs
pp 99-108, Proc. of the 27th Annual International Symposium on Fault-Tolerant Computing – FTCS-27, Seattle, USA, 24-27/06/1997.

[Tse 1996] T.H. Tse, Z. Xu
Test Case Generation for Class Level Object-Oriented Testing
pp 4T4.0-4T4.12, Proc. of the 9th International Software Quality Week in 1996 –QW'96, 5/1996.

[Turner 1993] C.D. Turner, D.J. Robson
The State-Based Testing of Object-Oriented Programs
pp 302-310, Proc. of IEEE Conference on Software Maintenance, Los Almitos, USA 1993.

[UML 2001] OMG
UML specification
version 1.4, www.omg.org/cgi-bin/doc?formal/01-09-67

[Ural 1992] H. Ural
Formal method for test sequence generation
pp 311-325, *Computer Communications*, vol. 15, no. 5, 1992.

[Xanthakis 2000] S. Xanthakis, P. Régnier, C. Karapoulios
Le test des logiciels
Hermes, ISBN 2-7462-0083-X, 2000.

[Voas 1996] J.M. Voas
Object-Oriented Software Testability
pp 279-290, Proc. of International Conference on Achieving Quality in Software, 1996.

[VuLe 2001] H. Vu Le, K. Akif, Y. Le Traon, J. M. Jézéquel
Selecting an Efficient OO Integration Testing Strategy: An experimental Comparison of Actual Strategies
pp 381-401, Proc. of 15th European Conference on Object-Oriented Programming in Budapest, Hungary – ECOOP 2001, 06//2001
LNCS, no. 2072, 2001.

[Waeselynck 1999] H..Waeselynck, P. Thévenod-Fosse
A Case Study in Statistical Testing of Reusable Concurrent Objects
pp 401-418, Proc. of the 3rd European Dependable Computing Conference, 12/1999.

[Warmer 1998] J. Warmer, A. Kleppe
The Object Constraint Language: Precise Modeling with UML
Addison-Wesley, ISBN 0-201-37940-6, 1998.

[Wegner 1995] P. Wegner
Interactive Foundation of Object-Oriented Programming
pp 70-72, *IEEE Computer*, vol. 28, no. 10, 1995.

[Wegner 1997] P. Wegner
Why Interaction is more Powerful than Algorithms
pp 80-91, *Communication of the ACM*, vol. 40, no. 5, 1997.

[Weyuker 1986] E. J. Weyuker
Axiomatizing Software Test Data Adequacy
pp 1128-1138, *IEEE Transaction on Software Engineering*, vol. SE12, no. 12, 1986.

[Weyuker 1988] E. J. Weyuker
The evaluation of program based software test data adequacy criteria
pp 668-675, *Communication of the ACM*, vol. 31, no. 6, 1988.

[Wilde 1992] N. Wilde, R. Huitt
Maintenance Support for Object-Oriented Programs
pp 1038-1044, *IEEE Transaction on Software Engineering*, vol. 18, no. 12, 1992.

[Winter 1998] M. Winter
Managing Object-Oriented Integration and Regression Testing (without becoming drowned)
Proc. of the European International Conference on Software Testing Analysus and Review in 1999 – EuroSTAR'99, Munich, Germany, 11/1999.

[Zomaya 1996] A.Y.H. Zomaya
Parallel and distributed computing handbook
McGraw-Hill, ISBN 0-07-073020-2, 1996

[Zwenden 1992] S. H. Zwenden, W. D. Heym
Testing of Data Abstraction Based on Software Specification
pp 39-55, *Journal of Software Testing, Verification and Reliability*, vol. 1, no. 4, 1992.