Méthodes formelles pour le développement et la vérification de logiciels

Didier Buchs

Ecole Polytechnique Fédérale, Lausanne
Laboratoire de génie logiciel
1015 Lausanne Ecublens
Suisse

email: Didier.Buchs@ di.epfl.ch

http://lglwww.epfl.ch/~buchs

téléphone: +41 (21) 693.27.37 - fax: +41 (21) 693.50.79

RÉSUMÉ Cet article présente les différents composants d'une méthode de développement de logiciel intégrée. Nous soulignerons la nécessité de fournir un cadre cohérent à l'intégration des activités de spécification, d'implémentation et de vérification. L'usage de méthodes formelles sera étudié en montrant, l'importance de la sémantique dans la formalisation des techniques de développement ainsi que la nécessité à supporter un développement incrémental. L'incrémentalité est située au niveau de l'élaboration de la spécification et au niveau de la construction des programmes. Finalement nous présenterons une réalisation concrète de système de développement basé sur le formalisme CO-OPN que nous avons menée à l'EPFL, dont une partie est supportée par des outils intégrés dans l'environnement SANDS.

1 Introduction

L'existence de méthodes de développement de logiciel, ne permet pas d'éviter de nombreuses catastrophes économiques et humaines. Ceci nous mène à douter de la pertinence des méthodes actuellement appliquées. Les projets à risques dans lesquels les logiciel est l'élément central de contrôle s'accroissent en nombre. En conséquence, il est nécessaire que le taux de panne soit extrêmement faible. Un exemple particulièrement médiatique est la récente catastrophe du lanceur Ariane 501 le 4 juin 1996. Le rapport des experts [10] montre clairement que la catastrophe résulte d'au moins quatre fautes dans le développement du système de contrôle de trajectoire. Ces fautes liées à une mauvaise réutilisation de composants informatiques se sont produites lors de la spécification (spécification incorrecte et incomplète), de la conception (modèle inadapté des fautes), lors de l'implémentation (code insuffisamment robuste) et du test (test d'intégration incomplet).

La réalisation de systèmes fiables est un domaine multi-disciplinaire ou la notion de faute, c'est à dire l'origine d'un comportement déviant par rapport aux attentes (la défaillance résultante de l'erreur), est l'élément central de réflexion. De manière résumée la communauté de la sûreté de fonctionnement, a pour objectifs de spécifier, concevoir, réaliser et exploiter des systèmes où la faute est naturelle, prévue et tolérable [9]. L'obtention d'un système sûr de fonctionnement passe par l'utilisation combinée dans un processus de développement d'un ensemble de méthodes qui peuvent être classées en: prévention, tolérance, élimination et prévision des fautes.

22 Octobre 1996

- La prévention étudie les moyens d'empêcher l'occurrence ou l'introduction de fautes. Les méthodes structurées et formelles permettent généralement d'améliorer la capture du cahier des charges et sa présentation non-ambiguë dans un cadre formel.
- La tolérance permet de fournir un service à même de remplir la fonction du système en dépit de fautes. De nombreux principes permettent dans le cadre général de la duplication de composants d'accepter la défaillance de certains d'entre eux.
- L'élimination sert à réduire la présence des fautes en, par exemple, confrontant la réalisation à une description du système attendu. Les techniques de test statique ou dynamique, sont des méthodes utilisables dans ce but.
- La prévision essaie d'estimer la présence, la création et les conséquences des fautes.

Comme indiqué précédemment, l'obtention d'un système sûr de fonctionnement passe par l'utilisation intégrée de méthodes spécifiques. Nous allons dans le reste de cet article montrer les possibilités d'intégration de différentes méthodes permettant de prendre en compte la prévention, la tolérance et l'élimination des fautes. L'intégration nécessite la définition de liens très clairs et bien formalisés entre les méthodes, ellesmêmes rigoureuses. Les méthodes formelles, à cause du fait de leurs caractéristiques non-ambiguës, peuvent elles seules garantir la qualité de l'intégration de méthode et la production d'outils automatisant le processus de développement.

Cet article reflète de manière très résumée les résultats obtenus dans le contexte de différents projets sur le développement du formalisme CO-OPN (Concurrent Object Oriented Petri Nets), dont le financement est actuellement partiellement assuré par le projet Esprit LTR 20072 "Design for Validation" (DeVa) avec le support financier de l'Office Fédéral pour l'Education et de la Science et par le projet FNRS 2000-40583.94 "Formal Methods for Concurrency".

2 Processus de développement évolutif

Actuellement, un modèle naturel de processus de développement doit permettre l'évolution du développement. La spécification, l'analyse de risque, l'implémentation et la vérification doivent être effectivement intégrées. Le modèle en spirale de Boehm illustre bien cette approche combinée des différentes techniques proposées. Néanmoins la formalisation de chaque étape est un moyen supplémentaire pouvant assurer la cohérence, ou en d'autres mots l'intégration sémantique, de ces différentes activités. Cette cohérence est un prérequis à la conception de techniques et d'outils appropriés au développement de systèmes complexes, outils permettant d'établir clairement et précisément les propriétés du système en développement. Les méthodes formelles ont pour objectif de fournir un tel cadre cohérent, néanmoins il faut préciser que de nombreuses méthodes, qualifiées à tort de formelles, n'ont pas réellement cette capacité (par exemple, ce n'est que dernièrement qu'une sémantique formelle à été fournie pour Z).

2.1 Intégration sémantique des formalismes

Sans entrer dans tous les détails, l'intégration sémantique nécessite la définition mathématique de différents domaines représentant d'une part la forme du discours et d'autre part les modèles associés représentant la sémantique. Le raisonnement automatisé ne pouvant se faire que sur l'univers symbolique du discours. La frontière du monde formel avec le monde informel (figure 1) est le lieu d'interprétation humaine que ce soit par l'évaluation des fonctionnalités d'un prototype lors de la validation ou, l'élicitation des propriétés exprimées dans un cahier des charges.

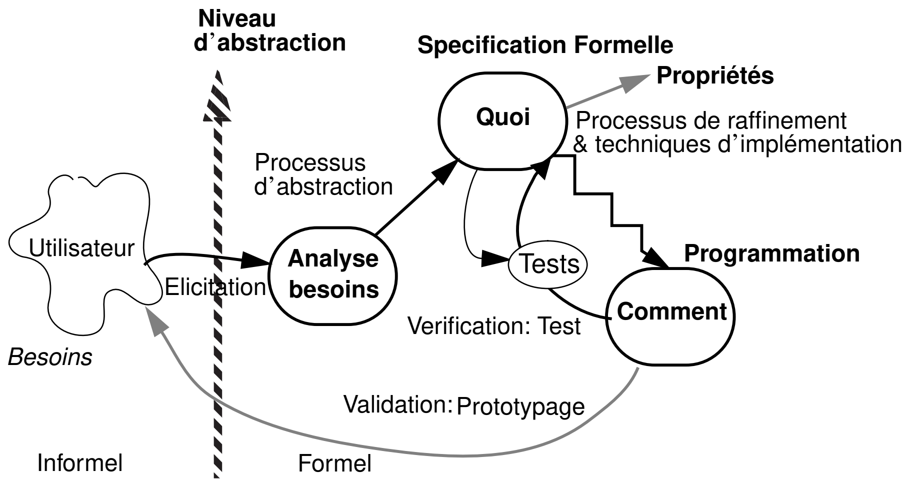

Fig. 1. Relations entre les activités du processus de développement

2.2 Aspects fonctionnels et non-fonctionnels

Les informations véhiculées dans l'analyse, la spécification et le codage sont de deux natures: les aspects fonctionnels et non-fonctionnels. Bien que cette distinction soit plutôt floue, nous pouvons principalement attribuer les aspects fonctionnels aux propriétés garanties par la spécification, tandis que les aspects non-fonctionnels seront des informations guidant l'obtention de l'implémentation. Il est clair que selon la nature du système cible les aspects non-fonctionnels seront différemment répartis, par exemple les contraintes temporelles dans un système temps réel sont des contraintes fonctionnelles alors qu'elles sont non-fonctionnelles dans un système traditionnel. Dans ce qui suit nous considérerons que la validité des programmes obtenus à partir de la spécification concerne les aspects fonctionnels.

2.3 Notre approche: SANDS/CO-OPN

Notre approche [4,5,6] propose une méthode de développement formel incrémental couvrant (figure 2):

- La modélisation au moyen d'un langage de spécification orienté objet supportant la concurrence, le raffinement, l'analyse et la simulation.

- La génération automatique de code au moyen d'un outil de prototypage, ainsi que l'ajout incrémental de portion de code à ce prototype afin d'obtenir une implémentation proche du produit final.
- La génération de tests à partir des spécifications en utilisant une technique de réduction de tests basée sur l'introduction d'hypothèses sur les programmes.

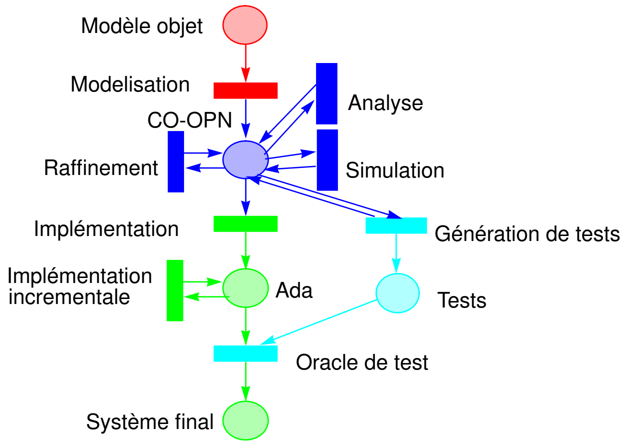

Fig. 2. Les activités et formalismes de la méthodologie SANDS/CO-OPN

3 Composants: spécifications, tests et programmes.

Dans la figure 1 les formalismes que nous étudions sont ceux de la spécification formelle, du test et de la programmation. Chacun de ces formalismes est interprété dans un univers sémantique qui décrit les modèles potentiels. Partant d'une adéquation des tests pour les spécifications, nous pouvons définir la validité des programmes en terme de satisfaction des tests.

3.1 Domaine du discours: signature et formules

De manière classique chaque domaine syntaxique comporte sa signature, les signatures étant compatibles entre elles. Nous avons donc la signature de la spécification (axiomes $SPEC_{\Sigma_{SP}}$), la signature des tests (tests $TEST_{\Sigma_T}$) et la signature des programmes (programmes $PROG_{\Sigma_P}$). A partir de ces signatures l'ensemble des formules valides dans les différents domaines syntaxiques est introduit.

Exemples de modèles CO-OPN

Comme illustration de nos travaux, nous introduisons un exemple simple mais significatif de spécification CO-OPN: l'automate de téléphone à carte. Cet exemple modélise une partie du comportement d'un automate téléphonique qui peut être utilisé avec une carte à prépaiement. Chaque carte inclut un code d'identification (le ADT Pin), qui est connu de l'utilisateur et demandé par l'automate quand l'utilisateur tente d'effectuer un appel téléphonique en payant avec cette carte. L'argent encore utilisable étant inscrit sur la carte. Nous modélisons ce système au moyen de différents ADT (Pin, Money, Bool et Nat) ainsi que de deux classes (PhoneCard et Telephone).

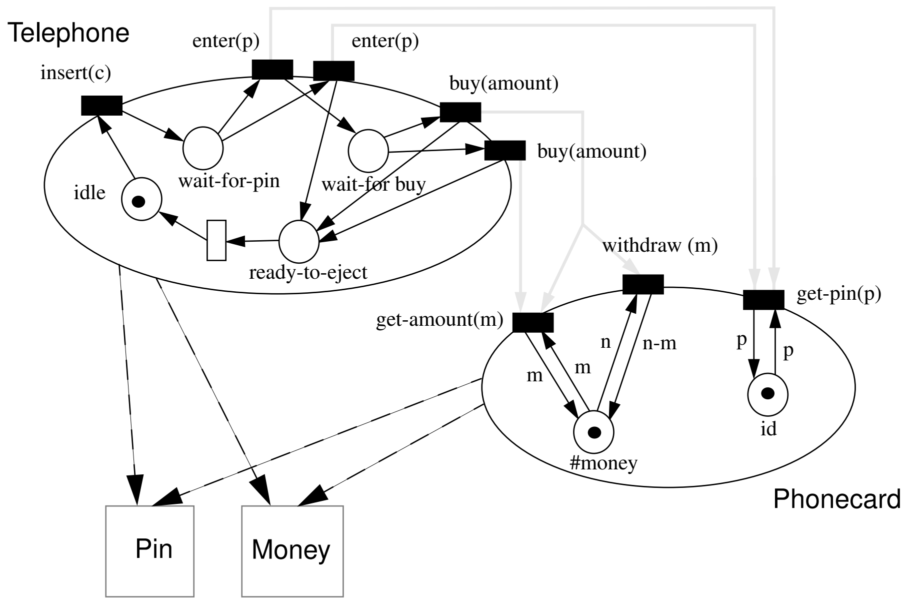

Fig. 3. Les classes, leurs descriptions internes et leurs synchronisations (vue partielle)

Dans la figure 3, nous donnons une représentation graphique partielle des classes, de leurs synchronisations et de leurs relations d'utilisation des types abstraits algébriques. Dans chaque ovale, représentant une classe encapsulée, un réseau de Petri représente le comportement d'une classe. Ce réseau de Petri représente l'état des objets au moyen des places (cercles) et les méthodes de l'objet au moyen des transitions (rectangles noirs). Deux types de transitions sont disponibles en CO-OPN:

- les transitions internes (les transitions usuelles des réseaux de Petri), qui décrivent les réactions spontanées et apparaissent sur le graphe comme des rectangles blanc,
- les transitions externes, ou méthodes, qui sont des événements visibles depuis l'extérieur d'une classe et apparaissent comme des rectangles noirs sur la bordure des classes. Ces transitions peuvent être paramétrées de façon à faire transiter des informations depuis l'objet appelant vers la méthode appelée.

Les synchronisations entre transitions sont dessinées au moyen de flèches grises.

3.2 Sémantiques associées aux spécifications, tests et programmes

Chaque composant syntaxique possède une sémantique: $\mathrm{Sem_{SP}}$ est la sémantique de la spécification, $\mathrm{Sem_T}$ la sémantique des tests et $\mathrm{Sem_P}$ la sémantique des programmes. Il est bien clair que ces sémantiques répondent à des contraintes liées à la nature des objets mathématiques choisis comme modèles. Dans le contexte algébrique le fait que chaque modèle soit une Σ-algèbre est indispensable à la théorie employée, de la même manière les programmes doivent se comporter comme une Σ-algèbre, ce qui signifie entre autre l'absence d'effet de bord. Il en résulte que l'adéquation entre ces formalismes est conditionnées par des hypothèses parfois fortes que nous appelons hypothèses de complétude des tests H_O. L'ensemble des tests complet est noté T_O.

Naturellement des relations entre formalismes doivent être définies de manière à ce que la relation de satisfaction $\models$ relie spécification et programme et la relation de satisfaction $\models_O$ relie test et programme. Le lien entre ces deux relations devant être, sous réserve des hypothèses initiales, la relation d'adéquation: (P satisfait H_O) $\Rightarrow$ ($P \models SP$ $\Leftrightarrow$ $P \models_O T_O$). Les principes énoncés précédemment sont à la base de la méthode de test que nous avons développée [2] à partir de [3].

Structuration des composants, incrémentalité

La complexité habituelle d'un système demande de pouvoir raisonner en termes plus abstraits que l'unité de modélisation que sont les fonctions ou les événements. Dans ce but les approches modernes de programmation et plus récemment de spécification ont introduit l'approche objet (voir [8] pour une présentation des évolutions orientées objets de VDM et Z). L'approche objet permet naturellement d'encapsuler des comportements et de modéliser des abstractions. Néanmoins ces caractéristiques à elles seules ne sont pas suffisantes pour produire un modèle pertinent, la composition de ces systèmes est également une caractéristique fondamentale permettant l'incrémentalité des développements propre à un modèle en spirale.

Le mécanisme d'héritage permet d'inclure l'incrémentalité dans les spécifications, chaque niveau d'héritage devenant alors le résultat d'un incrément de développement. Ce mécanisme ajoute une dimension de spécialisation des modèles lors du développement d'une spécification. Nous avons introduits dans CO-OPN un mécanisme incrémental de spécialisation de classes au moyen d'une notion de sous-typage sémantique basé sur la subtitutabilité forte [5].

Sémantique de base des objets CO-OPN

La sémantique de base de CO-OPN est exprimée au moyen de système de transitions, dont les étiquettes sont les noms des méthodes. L'activité des objets correspond à un cycle de service des méthodes et de stabilisation des comportements internes décrits par les transitions internes. Ces principes font de CO-OPN un langage fortement synchrone. La sémantique de CO-OPN est centrée sur la vision purement observationelle des comportements des méthodes, les comportements internes étant abstraits. Un programme P satisfaisant une spécification CO-OPN SP sera un programme dont la sémantique se comportera de la même manière (par rapport à la relation de bisimulation [11]) que sa spécification. Cela signifie:

$$(P \models SP) \text{ iff } (Sem(P) \underline{\leftrightarrow} Sem(SP)).$$

Pour CO-OPN, le langage des tests est celui des formules de la logique HML de Hennessy-Milner (opérateurs: Next, Conjonction, Négation), le résultat d'adéquation de cette logique avec la bisimulation fournis un cadre cohérent aux développements de spécifications CO-OPN. Nous avons donc:

$$(P \models_O Test) \text{ iff } Sem(P) \models_{HML} Test$$

Ce qui implique la propriété de complétude des test évoquée précédemment. Nous verrons, lorsque nous aborderons les aspects de vérifications, que depuis l'ensemble de test complets nous sommes à même de réduire l'ensemble des tests complets (potentiellement infini) à un ensemble de tests pour lequel un oracle [12] pourra décider si le programme les valide.

3.3 Implémentation incrémentale d'une spécification CO-OPN

Sans entrer dans les détails, une implémentation CO-OPN est un ensemble de processus client-serveur coopérant sur un réseau [7]. Une tâche de contrôle est active par objet, celui-ci gérant de multiples requêtes correspondant aux tentatives de franchissement de méthodes. Le mécanisme d'échec et réexécution est similaire au backtracking distribué des langages prologs concurrents. Afin de protéger les objets, lors de l'évaluation, un protocole à deux phases (Two Phase-Locking (2PL) protocole) a été adopté. L'environnement d'implémentation est Ada95 et son annexe distribuée.

Les algorithmes ont été conçus pour être modulaires et réutilisables et sont décrits dans les classes symbolique (figure 4) alors que la classe abstraite correspond à la structure du prototype de la classe. L'implémentation incrémentale est rendue possible par l'utilisation de la programmation orientée objet qui permet un ajustement fin du code. Il est possible de changer les représentations internes, les algorithmes et d'ajouter des fonctionnalités de manière transparente dans les classes concrètes. Le prototype, ouvert et évolutif, utilise des principes proches des mécanismes orientés objets tels que : design patterns, prototype objects, dynamic binding.

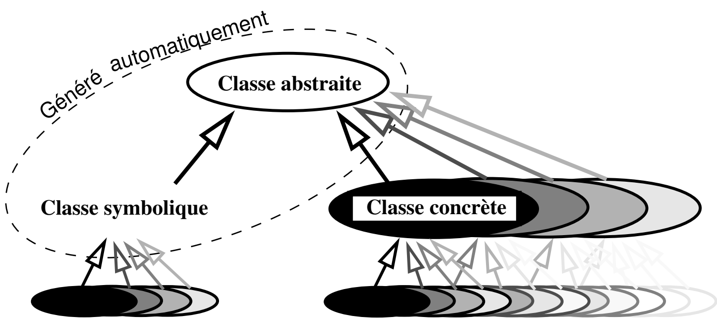

Fig. 4. Classes d'un prototype généré pour une classe de la spécification

3.4 Vérification d'un programme

La procédure de test illustrée dans la figure 5 comporte trois phases dont nous présentons la première, celle-ci consiste en la production de tests pertinents pour notre programme (voir [1,2] pour un présentation complète du test orienté objet).

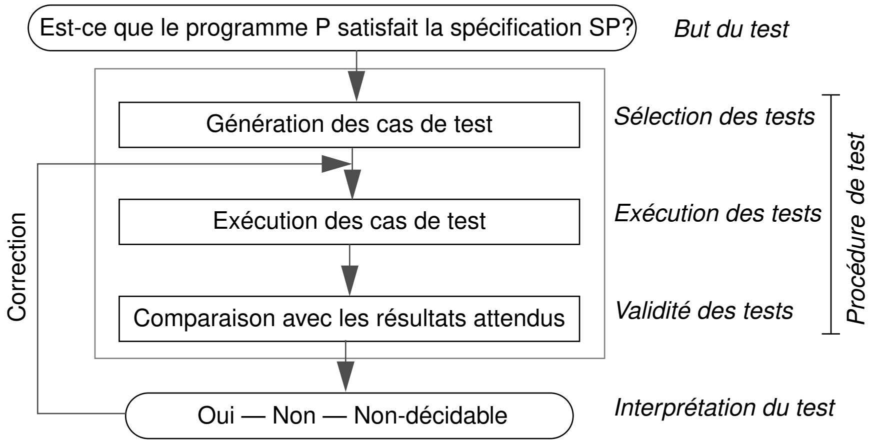

Fig. 5. La procédure pour le test dynamique

Cette phase consiste, pour une spécification donnée à progressivement réduire l'ensemble complet de test T_O en un ensemble de tests valides et non-biaisés (rejetant les programmes incorrects et n'acceptant que les programmes corrects sous réserve de la validité des hypothèses).

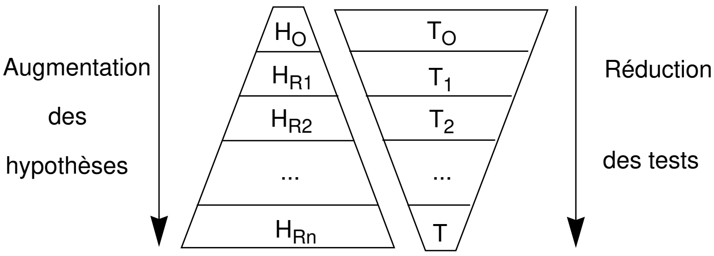

Fig. 6. La procédure de réduction des tests

Par exemple, supposons que nous avons créé une carte avec pin= 1234 alors des tests valides pour le téléphone sont les séquences HML:

<<cabin.insert card> <cabin.enter 1234> <cabin.buy 12> T, true>
<<cabin.insert card> <cabin.enter 4321> <cabin.buy 12> T, false>
<<cabin.insert card> <cabin.insert card> T, false>
<<cabin.insert card> <cabin.enter 1234> <cabin.buy 12> <cabin.insert card> <cabin.enter 1234> <cabin.buy 6> T, true>

L'obtention de ces tests résulte de l'application d'un certain nombre d'hypothèses qui correspond à l'intuition habituelle des testeurs, nous proposons principalement les hypothèses de régularité et d'uniformité comme mécanisme élémentaires de réduction des tests [2].

Incrémentalité dans le test

Le développement incrémental de logiciel doit également pris en compte au niveau des tests. D'une certaine manière l'usage des mécanismes d'héritage à sa contrepartie dans la procédure de test. Il est possible de considérer des tests de manière incrémentale en minimisant les test à réappliquer dans un sous-type.

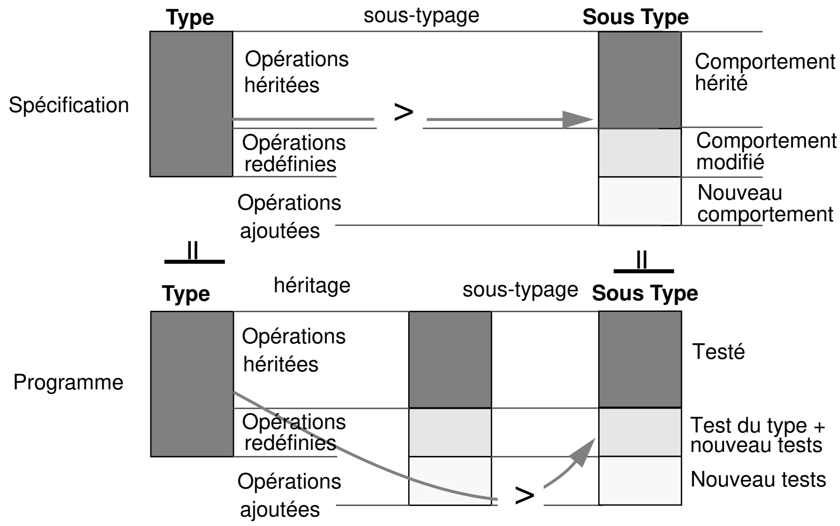

Fig. 7. Incrémentalité dans le test

La figure 7 illustre ce principe, pour lequel un sous-type dans une spécification (type conservant au niveau sémantique les comportements comparables de son super-type) peut produire des tests, limités aux nouvelles fonctionnalités du sous-type, pour un sous-type du programme à tester. Cette propriété n'est valide que si dans le programme le sous-type est un sous type sémantique.

4 Conclusion

Nous avons présenté succinctement les possibilités d'intégrer les méthodes formelles dans un cycle de développement supportant l'incrémentalité. Le processus de développement proposé comporte des phases de spécification, vérification et implémentation décrites au moyen des principes de l'orientation objet. La méthode proposée CO-OPN, bien que résultante de travaux de recherche fondamentaux, couvre en partie les besoins actuels en matière de développement de systèmes distribués critiques.

5 Remerciements

Je tiens à remercier toutes les personnes impliquées dans le projet ‹Conform› que ce soit dans le passé pour Jacques Flumet ou actuellement pour Stéphane Barbey, Olivier Biberstein, Christophe Buffard, Mathieu Buffo, Giovanna di Marzo, Jarle Hulaas, Nicolas Guelfi, Cécile Péraire et Pascal Racloz pour la qualité de leur travail dans ce projet.

6 Références

[1] Stéphane Barbey, Manuel Ammann, and Alfred Strohmeier. Open issues in testing object-oriented software. In Karol Frühauf, editor, *ECSQ '94 (European Conference on Software Quality)*, pages 257– 267, Basel, Switzerland, October 17-20 1994.

[2] Stéphane Barbey, Didier Buchs and Cécile Péraire, A Theory of Specification-Based Testing for Object-Oriented Software, EDCC2, Taormina, Italy, 1996, à paraitre dans LNCS, également Technical Report 96/163, EPFL-DI-LGL, January 1996.

[3] Gilles Bernot, Marie-Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *IEE Software Engineering Journal*, 6(6):387–405, November 1991.

[4] Olivier Biberstein and Didier Buchs. CO-OPN/2, an object-oriented specification language based on hierachical algebraic nets. In R. Wieringa and R. Fenstra, editors, *Proceedings of ISCORE '94*, pages 47–62, September 1994.

[5] Olivier Biberstein and Didier Buchs. Structured algebraic nets with object-orientation. In *Workshop on Object-Oriented Programming and Models of Concurrency '95*, Turin, June 1995.

[6] Didier Buchs and Nicolas Guelfi. A concurrent object-oriented Petri nets approach for system specification. In M. Silva, editor, *12th International Conference on Application and Theory of Petri Nets*, pages 432–454, Aahrus, Denmark, June 1991.

[7] Didier Buchs and Jarle Hulaas, "Evolutive Prototyping of Heterogeneous Distributed Systems Using Hierarchical Algebraic Petri Nets", IEEE International conference on Systems, Man and Cybernetics, Beijing, China, Oct. 14-17, 1996.

[8] Kevin Lano, Formal Object-Oriented Development, Springer Verlag, FACIT series, 1995.

[9] Jean-Claude Laprie (Ed.). Guide de la sûreté de fonctionnement, Cépaduès, 1995.

[10] J.L. Lyons, ARIANE 5, Flight 501 Failure Report by the Inquiry Board, http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html, Paris 19 juillet 1996.

[11] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[12] Elaine J. Weyuker. The oracle assumption of program testing. In *13th International Conference on System Sciences*, pages 44–49, Hawaii, USA, 1980.