Test de conformité : une approche algébrique

Agnès Arnould, Pascale Le Gall

Test de conformité : une approche algébrique

Agnès Arnould* — **Pascale Le Gall****

* *IRCOM-SIC, UMR 6615, Université de Poitiers*
SP2MI
F-86962 Futuroscope Cedex
arnould@sic.sp2mi.univ-poitiers.fr

** *L.a.M.I., Université d'Évry*
253, place des Terrasses
F-91000 Evry Cedex
legall@lami.univ-evry.fr

RÉSUMÉ. *Nous proposons dans cet article une formalisation du test de conformité contre des spécifications algébriques. Nous interprétons par des modèles algébriques les comportements fonctionnels des programmes et concrétisons les tests abstraits issus des spécifications en tests concrets dans les langages de programmation cibles. Cette double passerelle nous permet de lier la correction sémantique d'un programme avec la soumission en succès de son jeu de tests de référence. Notre formalisation du test va ainsi de la définition des tests jusqu'à leurs soumissions. Enfin nous montrons que notre approche est en prise directe avec les pratiques industrielles et les outils de génération automatique de tests de conformité existants.*

ABSTRACT. *In this article, we propose a formalization of conformance testing with respect to algebraic specifications. We interpret functional behavior of programs with the help of algebraic models. We materialize abstract tests extracted from the specification with concrete tests written in a given programming language. This double bridge allows us to link the semantic correctness of a program with the successfull submission of its reference tests set. Thus our testing formalization includes both tests definition and tests submission. Finally we show that our approach mirrors industrial practices and the existing tools of automatic conformance tests generation.*

MOTS-CLÉS : *Test de conformité, Spécifications algébriques, Jeu complet de tests, Concrétisation des tests.*

KEYWORDS: *Conformance testing, Algebraic specifications, Complete tests set, Tests materialization.*

RSTI - TSI – 21/2002. Test de logiciel, pages 1219 à 1242

1. Introduction

Les spécifications constituent des références de correction. Elles sont donc à l'origine de certaines vérifications notamment au travers des tests de conformité. Lorsque les spécifications sont formelles, il est possible de définir des jeux de tests de référence. Ces jeux de tests sont exhaustifs, c'est-à-dire que leur exécution avec succès est synonyme de correction. Bien sûr, la taille importante, voire infinie de ces jeux de tests rend impraticable leur exécution complète. Cependant leur existence assure une qualité asymptotique réelle à l'activité de test.

En outre, les spécifications formelles alliées à des techniques de résolution de contraintes permettent de construire des jeux de tests selon divers critères de sélection. Différents outils mettant en œuvre ces méthodes ont montré leur efficacité dans un cadre de recherches académiques. Une telle automatisation de l'activité de test permet d'allier une hausse de la qualité des tests de conformité à une baisse importante du coût et des délais. Les spécifications formelles ont donc un avenir économique certain dans le domaine de la vérification.

Cependant, la nature profondément différente des spécifications et des programmes rend leur comparaison difficile. Ainsi, si les raffinements de spécification et les vérifications associées font l'objet de recherches et de publications nombreuses [ORE 93, BID 95], la vérification des programmes constitue une activité beaucoup plus modeste. En effet, la comparaison de deux objets (spécification et programme) exprimés dans deux langages différents ayant deux sémantiques différentes n'est pas possible *a priori*. Cette comparaison nécessite d'établir une passerelle entre deux mondes, ce qui n'est possible que sous certaines hypothèses. La vérification entre un programme et une spécification s'effectue donc toujours sous la réserve de certaines hypothèses (explicites ou implicites), quelle que soit la méthode de vérification : par la preuve, par génération automatique de code [HAL 91], ou par le test [BER 91, LEG 96, MAC 00]. Ces hypothèses marquent la limite de chaque méthode. La différence des hypothèses atteste de la complémentarité des méthodes. Ainsi, si les méthodes formelles accroissent le rapport qualité/prix de la vérification, elles préservent la complémentarité des approches.

Dans cet article, nous établissons une passerelle entre le monde des spécifications et celui des programmes. Cette passerelle nous permet à la fois de formaliser la correction d'un programme par rapport à sa spécification et de concrétiser les tests issus de la spécification en vue de leur soumission au programme. Cette formalisation de l'ensemble des étapes du test nous permet de construire un jeu de tests de référence et de caractériser de manière précise ses qualités en fonction des propriétés du programme et de sa spécification. A des fins pédagogiques, nous illustrerons ces concepts en utilisant le langage de spécification CASL et le langage de programmation ADA. CASL est un nouveau langage algébrique [CoF 01] qui regroupe les différents principes des langages antérieurs (opérations partielles, sous-sortes, structuration ...).

En section 2, nous proposons une passerelle sémantique et une notion de correction associée. En section 3, nous établissons la passerelle syntaxique et le jeu de tests de

référence. Enfin dans les sections 4 et 5, nous montrons que notre formalisation fonde l'ensemble des pratiques courantes du test de conformité et en particulier les outils existants de génération automatique à partir des spécifications formelles algébriques. Les présentations des méthodes et outils de sélection faites dans cet article restent cependant sommaires. Les lecteurs intéressés par plus de détails pourront consulter [MAR 91, BER 91, ARN 99].

2. Correction d'un programme par rapport à une spécification

L'approche que nous proposons est indépendante du formalisme de spécifications considéré. Nous utilisons donc une notion de formalisme générique classique [WIR 90, MES 89]. Un formalisme $\mathcal{L}$ est caractérisé par une classe de signatures $Sig_{\mathcal{L}}$, et pour chaque signature Σ de $Sig_{\mathcal{L}}$ par la classe de ses modèles $Mod_{\mathcal{L}}(\Sigma)$, l'ensemble de ses formules $Sen_{\mathcal{L}}(\Sigma)$, la relation de satisfaction des formules par les modèles $\models_{\mathcal{L},\Sigma}$ et la relation de déduction des formules à partir des ensembles de formules $\vdash_{\mathcal{L},\Sigma}$. Cette dernière n'est pas utile pour définir les concepts introduits en sections 2 et 3, mais elle est indispensable en pratique pour construire les tests. Ce dernier point sera abordé en sections 4 et 5. Il est à noter que les notations précédentes sont généralement définies à l'aide de la théorie des catégories [GOG 92]. Cependant, par souci de simplicité dans cet article, aucune référence aux aspects catégoriques ne sera proposée.

Considérons un formalisme arbitraire $Spec$. Classiquement, une spécification SP pour un tel formalisme est définie comme la donnée d'une signature Σ de Sig_{Spec} et d'un ensemble Ax de formules de $Sen_{Spec}(\Sigma)$ appelées axiomes. La classe des modèles $Mod_{Spec}(SP)$ de la spécification est alors souvent simplement définie comme la classe des Σ-modèles validant les axiomes de SP. Cependant, pour de nombreux formalismes, les définitions relatives aux spécifications peuvent être plus subtiles. Il est ainsi courant d'ajouter aux spécifications des contraintes supplémentaires en vue de sélectionner plus finement les modèles. Ces contraintes peuvent être corrélées à la présence de primitives de structuration ou bien à celle de constructeurs parmi les opérations de la signature. Dans un souci de généralité et de simplicité, nous caractérisons donc une spécification directement par l'ensemble de ses modèles sans attacher d'importance à la manière dont ces derniers sont définis. Nous verrons plus tard (sections 4 et 5) que la définition concrète des spécifications reprend toute son importance lors du calcul effectif des tests.

Définition 1
Une **spécification** SP *d'un formalisme* $Spec$ *est caractérisée par sa signature* $\Sigma \in Sig_{Spec}$ *et la sous-classe de ses modèles* $Mod_{Spec}(SP) \subset Mod_{Spec}(\Sigma)$.

Traditionnellement, un programme est présenté comme la dernière étape de raffinement d'un développement formel et est abstrait par un modèle particulier. Si cette vision pédagogique donne une bonne idée du but d'un développement formel, cette vue est trop simpliste pour être utile en pratique. Notamment, elle ne donne pas suffi-

samment d'éléments pour aborder précisément le thème de la correction (ou de la non correction) d'un programme par rapport à une spécification.

Plaçons-nous dans le cas d'une observation comportementale du programme, c'est-à-dire selon une approche de type "boîte noire". Même en imaginant observer l'intégralité du comportement du programme (par exemple par une infinité de tests), il est souvent impossible de déterminer **le** modèle du programme. En effet, le plus souvent, seul un ensemble de modèles observationnellement équivalents peut être caractérisé. À titre d'exemple, il existe de nombreuses implantations et donc au moins autant de modèles différents, du type abstrait des ensembles, que ce soit les listes triées, les listes avec éléments redondants ... La multiplicité des modèles provient ici essentiellement de l'hypothèse qu'il n'y a pas de comparaison directe possible entre deux représentations des ensembles, mais seulement à travers les accès à ces ensembles, comme l'appartenance d'un élément à un ensemble ou bien le cardinal de l'ensemble. Comme l'approche "boîte noire" induit ainsi un accès partiel aux seules observations licites pour le programme, cet accès limité ne permet pas de distinguer le modèle du programme de tous les modèles autorisant exactement les mêmes observations, soit les modèles observationnelement équivalents.

Changeons maintenant d'approche et "ouvrons la boîte" du programme. Le comportement interne du programme est désormais précis, mais malheureusement il n'a souvent plus rien de fonctionnel. En effet, de nombreux programmes contiennent des effets de bord, des mécanismes de levée et rattrapage d'exceptions, d'autres ne sont tout simplement pas déterministes. Par exemple, le résultat d'une heuristique par méthode stochastique peut souvent être vu comme fonctionnel en observant le résultat à ε près, mais son comportement interne est totalement non déterministe. Il est donc impossible d'associer systématiquement un modèle déterministe au comportement interne des programmes, sauf si l'on restreint la classe des programmes considérés aux seuls programmes fonctionnels et déterministes.

D'autre part, les langages de programmation peuvent être très éloignés des formalismes de spécification. Ainsi, il peut être difficile de faire un lien entre l'interface d'un programme et une signature du formalisme de spécification.

Nous proposons donc de considérer un programme dans son propre formalisme, *a priori* différent de celui de sa spécification. Notons que les programmes considérés pour le test ne sont pas des applications complètes, mais plutôt des modules (ou unités de programmes) avec des interfaces suffisamment larges pour permettre une bonne observation de leurs comportements.

Définition 2
Soit $Prog$ un formalisme. Un **programme** *P dans $Prog$ est caractérisé par son interface I de Sig_{Prog} et le prédicat d'observation de son comportement : $(P \rightsquigarrow) \subset Sen_{Prog}(I)$.*

La classe $Mod_{Prog}(P)$ des **modèles** *de P est définie par :*

$$\{M \in Mod_{Prog}(I) \mid \forall \tau \in Sen_{Prog}(I), M \models_{Prog,I} \tau \ \textit{ssi} \ P \rightsquigarrow \tau\}$$

Le choix du formalisme $Prog$ dépend du langage de programmation dans lequel est développé le programme à tester P et éventuellement de l'instrumentation de test disponible. Intuitivement I est l'interface du programme. Cela comprend non seulement les types, fonctions et procédures définis par le programme proprement dit, mais aussi ceux issus des bibliothèques et ceux directement fournis par le langage de programmation lui-même. $Sen_{Prog}(I)$ est l'ensemble des observations ou des tests que l'on peut effectuer sur le programme enrichi par le langage de programmation et le cas échéant par son instrumentation de test. Nous considérons ainsi une notion riche du test, qui comprend non seulement les entrées de test et les fonctions et procédures à tester, mais aussi l'instrumentation de test et son **oracle**. Ainsi un test est une expérience complète qui s'exécute directement en vrai ou faux. Par exemple, une fonction `somme` du langage Caml, qui réalise la somme des éléments d'une liste d'entiers, peut être testée par l'expression `somme [1;2;3] = 6`, ou toute autre expression booléenne construite par application de la fonction `somme`. On peut donc choisir de considérer comme tests Caml, l'ensemble des expressions booléennes que l'interpréteur peut évaluer après chargement du programme. De même, les tests ADA peuvent être l'ensemble des fonctions booléennes sans argument que l'on peut compiler après importation du programme. Certaines approches [MAC 00] formalisent les tests par des termes qui représentent des exécutions du programme. Mais ce choix n'inclut pas la formalisation de l'oracle (quelle est la valeur attendue du terme ? est-elle définie ? ...). C'est pour cela que nous reprenons le choix de [BER 91] et préférons formaliser les tests par des formules.

Il n'est pas possible de comparer le comportement d'un programme à celui décrit par sa spécification directement à travers les formules. En effet, les formules de la spécification et celles du programme (les tests) sont de nature très différente. En particulier la majorité des formules de la spécification ne sont pas traduisibles en tests. Il est donc impossible, par exemple, de soumettre via une simple traduction syntaxique les axiomes de la spécification. En effet, ces formules sont souvent beaucoup trop compliquées. Elles peuvent contenir des variables, qui nécessitent leur remplacement par des valeurs. Elles peuvent aussi utiliser des prédicats ou des constructions prédéfinies du langage de spécification qui ne sont pas implantées dans le programme ou son instrumentation. Par exemple, beaucoup de données complexes ne supportent pas le test d'égalité et ne peuvent être comparées que partiellement. En conclusion, ces formules qui ne peuvent être traduites en tests, posent le problème dit de l'oracle.

Nous proposons donc de comparer un programme à sa spécification au travers des modèles. Nous choisissons simplement d'interpréter le comportement d'un programme sur la même classe de modèle que sa spécification.

Définition 3
Deux formalismes $Spec$ et $Prog$ sont **compatibles** *pour les signatures Σ de Sig_{Spec} et I de Sig_{Prog} si et seulement si*

$$Mod_{Spec}(\Sigma) = Mod_{Prog}(I)$$

Bien entendu, cette interprétation des programmes dans le "monde" des spécifications peut apparaître artificielle. Elle ne correspond pas à une réalité, par exemple liée à la sémantique du langage de programmation. Cependant elle est très utile, car l'hypothèse de modélisation sous-jacente permet de mesurer la portée des vérifications possibles par le test (cf section 4).

Notons, de plus, que ce choix d'interprétation peut mener à un ensemble vide de modèles pour certains programmes ($Mod_{Prog}(P) = \emptyset$). C'est le cas de tous les programmes qui ne peuvent être décrits selon le formalisme de spécifications choisi. Par exemple, certains programmes impératifs comportent des effets de bord qui rendent leur comportement dépendant de leur état. Ils ne peuvent donc pas être décrits par des modèles fonctionnels. Le test de programmes avec effets de bord contre une spécification fonctionnelle n'a donc pas de sens. En effet, un même test soumis plusieurs fois peut être en succès et en échec selon l'état courant du programme. Nous dirons qu'un tel programme ne vérifie pas les **hypothèses de test** ou n'est pas **testable**. Dans la suite de cet article, nous supposerons que le programme considéré est testable au sens où il admet au moins un modèle ($Mod_{Prog}(P) \neq \emptyset$).

La vérification des hypothèses de test constitue un prérequis au test formel présenté dans cet article. Nous ne proposons pas ici de cadre ou de méthodes pratiques visant à leur vérification. Par exemple, pour vérifier qu'un programme impératif a un comportement fonctionnel, on peut s'assurer du passage systématique des variables en paramètres ou tester préalablement la convergence d'un algorithme stochastique vers une unique valeur. La vérification formelle des hypothèses de test pourrait passer par l'interprétation du comportement des programmes dans leur propre sémantique et l'établissement d'un pont entres ces deux classes de modèles. Cependant une telle formalisation sort du cadre de notre étude et n'a jamais été explorée à notre connaissance.

Comme nous l'avons déjà écrit, l'encapsulation dans les programmes empêche d'associer un unique modèle à un programme. En effet, l'encapsulation rend équivalent, d'un point de vue comportemental, différents modèles. Cela permet d'implanter de manière multiple un même type abstrait. Cette équivalence comportementale (ou observationnelle) des modèles algébriques a déjà été largement étudiée dans le cadre du raffinement de spécifications [ORE 93] car elle permet de modéliser l'intuition selon laquelle il existe plusieurs implantations possibles pour une même description abstraite des attentes utilisateur.

Définition 4
Soit SP une spécification dans le formalisme $Spec$ et P un programme dans le formalisme $Prog$, tels que $Spec$ et $Prog$ soient compatibles pour les signatures Σ de SP et I de P. Soit $Obs_{\Sigma,I} \subset Sen_{Spec}(\Sigma)$ un ensemble de formules dites observables.

P est **correct** *par rapport à SP à $Obs_{\Sigma,I}$ près si et seulement si pour tout modèle $M \in Mod_{Prog}(P)$ du programme il existe un modèle $N \in Mod_{Spec}(SP)$ tel que $M \equiv_{Obs_{\Sigma,I}} N$.*

Rappelons que deux modèles M et N sont dits équivalents à observation près sur $O \subset Sen_{Spec}(\Sigma)$, ce qui se note $M \equiv_O N$, si et seulement si pour toute formule $\varphi \in O$, $M \models_{Spec,\Sigma} \varphi$ si et seulement si $N \models_{Spec,\Sigma} \varphi$.

Notons que l'observation $Obs_{\Sigma,I}$ dépend bien sûr du formalisme de spécification et de la signature de la spécification mais aussi du langage de programmation et de l'interface du programme. En effet, le fait que l'on puisse tester l'égalité entre deux données, par exemple, ne dépend pas de la spécification mais bien du programme. Est-elle prédéfinie dans le langage ? A-t-elle été implantée dans le programme ? A-t-elle été ajoutée dans l'instrumentation de test ? etc.

En clarifiant d'emblée la notion de correction appropriée pour la vérification par test, notre cadre est similaire aux cadres proposés dans le domaine de la vérification par test des protocoles de télécommunications ([DSS 87, BRI 88, PHA 94, GRO 96]). En effet, ces derniers explicitent en tout premier lieu la relation d'implantation d'un programme (par hypothèse modélisé par un automate) par rapport à une spécification (elle aussi modélisée par un automate). La relation d'implantation se définit alors via une équivalence de traces jugée appropriée. Comme, par hypothèse, les deux objets, programme et spécification s'expriment dans le même cadre, la notion de correction est de fait plus simple car elle ne fait pas appel à un filtre d'observations qui rend compte de la différence de nature des spécifications et des programmes[1].

Il est remarquable de noter que la correction d'un programme est définie selon un cadre formel qui symétrise les rôles de la spécification et du programme. Chacun de ces deux objets est décrit selon son propre formalisme et selon deux points de vue opposés. Tandis que la spécification est définie par un ensemble de modèles à partir duquel on peut déduire un ensemble de formules, le programme est défini par un ensemble de formules duquel est construit un ensemble de modèles. Notre cadre introduit donc une dualité entre le monde des spécifications et celui des programmes.

Il nous faut désormais transcrire la notion de correction au niveau des tests et des jeux de tests. Dans une pure approche "boîte noire", cette transcription induit une traduction des tests abstraits en des tests concrets susceptibles d'être soumis au programme sous test. Cette transcription va ainsi rompre le principe de dualité adopté jusqu'ici. C'est précisément l'objet de la section suivante.

3. Des tests abstraits aux tests concrets

Nous souhaitons construire des tests de conformité à la spécification dans le formalisme de spécification, puis évaluer ces tests sur le programme dans le langage de programmation. Il est donc nécessaire de préciser comment transposer les tests issus de la spécification en des exécutions du programme qui permettent d'affirmer

1. Notons que même si la notion de correction est plus simple pour le domaine du test de protocoles, le problème de l'oracle subsiste néanmoins car les automates considérés, étant souvent non-déterministes, peuvent conduire à des verdicts de test inconclusifs.

que le test est en succès ou en échec. Autrement dit, nous devons disposer d'une traduction des tests abstraits issus de la spécification, vers les tests concrets soumis au programme.

Définition 5
Soit deux formalismes $Spec$ et $Prog$ compatibles pour les signatures $\Sigma \in Sig_{Spec}$ et $I \in Sig_{Prog}$. $Obs_{\Sigma,I} \subset Sen_{Spec}(\Sigma)$ désigne l'ensemble des formules observables sur Σ. Une **concrétisation** *est une application $\gamma_{\Sigma,I} : Obs_{\Sigma,I} \rightarrow Sen_{Prog}(I)$.*

La concrétisation $\gamma_{\Sigma,I}$ est **correcte**, *si pour toute formule observable $\varphi \in Obs_{\Sigma,I}$ et pour tout modèle $M \in Mod_{Spec}(\Sigma)$, $M \models_{Spec,\Sigma} \varphi$ si et seulement si $M \models_{Prog,I} \gamma_{\Sigma,I}(\varphi)$. La concrétisation $\gamma_{\Sigma,I}$ est* **complète**, *si elle est surjective.*

Les formules observables sont exactement les tests abstraits potentiels. La concrétisation permet donc de traduire les tests abstraits en tests concrets. Notons qu'en pratique, il n'est pas nécessaire de définir une concrétisation pour chaque programme. Une concrétisation générique pour un formalisme de spécification et un langage de programmation donné peut en général tout à fait être défini. Pour chaque nouveau programme, il restera alors à paramètrer cette concrétisation par la traduction des symboles de la spécification et de son formalisme vers ceux du programme et de son langage. Par exemple, on peut définir une concrétisation générique du langage de spécification CASL[2] [CoF 01] vers le langage de programmation ADA [BAR 88]. Cette traduction générique doit permettre, par exemple, de concrétiser certaines fonctions en procédures. Ainsi dans l'exemple classique des piles (figures 1 et 2), le dépilement d'un élément est spécifié[3] par 2 fonctions (top qui retourne l'élément de tête et pop qui modifie la pile), mais il est implanté par une seule procédure (`POPTOP` qui initialise son paramètre de retour avec l'élément de tête et modifie la pile passée en paramètre). Comme nous le verrons plus tard, cette traduction doit aussi permettre de relier une constante de la spécification (par exemple $empty$) à la valeur d'initialisation de données du programme (par exemple la valeur d'initialisation du type `STACK`).

from Basic/Numbers **version** 0.4 **get** NAT

spec PILES =
  NAT
then
  free type $Stack ::= empty \mid push(top :?Nat; pop :?Stack)$;
end

Figure 1. *Spécification CASL des piles*

2. "The Common Algebraic Specification Language"
3. Remarquons au passage que le langage CASL permet de spécifier de façon extrêmement concise les piles : ici top et pop sont simplement désignés ici comme les accesseurs du constructeur $push$.

La concrétisation générique de CASL vers ADA peut ensuite être spécialisée pour chaque programme par la donnée de la traduction des symboles. On peut par exemple traduire certaines bibliothèques de base de CASL [ROG 00] vers les types de données prédéfinis en ADA et leurs opérations de manipulation. Par exemple, la sorte Nat des entiers naturels décrite par la spécification NAT de la bibliothèque des types de base peut être concrétisée en ADA par le type `NATURAL` et les opérations arithmétiques associées.

Détaillons maintenant la concrétisation de la spécification CASL des piles (figure 1) vers le programme ADA des piles (figure 2). Cette spécification des piles décrit le support de la sorte $Stack$ comme l'ensemble des valeurs librement construites par $empty$ et $push$ à partir de la sorte Nat prédéfinie par la spécification NAT de la bibliothèque Basic/Numbers. Les accesseurs top et pop sont spécifiés comme les deux inverses du constructeur $push$ et sont indéfinis sur le constructeur $empty$ (le symbole " ?" indique une opération partielle). La figure 2 propose un module ADA d'implantation classique des piles. Seule son interface est précisée, puisque en suivant une approche fonctionnelle du test (de type "boite noire"), la connaissance du corps du programme devient de fait sans intérêt.

```
package PILES is
       type STACK is limited private ;
       procedure PUSH (ELEM : in NATURAL; PILE : in out STACK) ;
       procedure POPTOP (PILE : in out STACK; ELEM : out NATURAL) ;
private
       ...
end
```

Figure 2. *Package ADA des piles*

La sorte $Stack$ sera traduite dans le type limité privé `STACK`. L'égalité prédéfinie sur la sorte $Stack$ ne sera donc pas concrétisée puisqu'il n'existe pas de prédicat d'égalité sur le type `STACK` dans le programme ADA. Cette dernière sera alors exclue des formules observables.

Le constructeur constant $empty : Stack$ sera traduit par l'initialisation par défaut des variables de type `STACK`.

Le constructeur $push : Nat \times Stack \rightarrow Stack$ sera traduit par la procédure `PUSH`, où le premier paramètre du constructeur $push$ est traduit par le premier paramètre d'entrée `ELEM` de la procédure `PUSH`, le deuxième paramètre de la fonction $push$ est traduit par le deuxième paramètre d'entrée `PILE` de la procédure `PUSH` et enfin le résultat de sortie de la fonction $push$ est traduit par l'unique paramètre de sortie `PILE` de la procédure `PUSH`.

L'accesseur $pop : Stack \rightarrow ?Stack$ est traduit par la procédure `POPTOP`. Son paramètre d'entrée est traduit par l'unique paramètre d'entrée `PILE` de la procédure et son paramètre de sortie est traduit par le paramètre de sortie `PILE` de la procédure

POPTOP. Comme la procédure POPTOP modifie son entrée, les sous-termes seront recalculés dans la traduction ADA autant de fois qu'ils apparaissent dans la formule CASL. Ainsi, lors de la concrétisation du test :

$$top(push(3, empty)) = top(pop(push(1, push(3, empty))))$$

les deux occurrences du sous-terme $push(3, empty)$ seront calculés deux fois dans deux variables de type STACK différentes.

L'accesseur $top : Stack \rightarrow ?Nat$ est traduit par la même procédure POPTOP avec le même paramètre d'entrée. Par contre, le paramètre de sortie de l'accesseur top est le deuxième paramètre de sortie ELEM de la procédure POPTOP.

Enfin, le prédicat de définition prédéfini par CASL est traduit par le test de non levée d'exception lors du calcul du terme concrétisé. Notons qu'en CASL, il n'existe pas de moyen de spécifier le nom des exceptions, seule la levée (ou la non levée) d'une erreur quelconque pourra donc être testée contre une spécification CASL. Ainsi le prédicat de définition CASL sera toujours traduit en ADA par le filtrage indéfini others.

```
function test1 return boolean is
          pile : Stack ;
          elem : Natural ;
begin
          poptop(pile, elem) ;
          return false ;
exception
          when others =>
                return true ;
end test1 ;
```

Figure 3. *Concrétisation de* $\neg Def\ pop(empty)$

Ainsi le test abstrait $\neg Def\ pop(empty)$ sera traduit par le test concret de la figure 3, et le test abstrait $top(push(2, push(0, empty))) = 2$ sera traduit par le test concret de la figure 4. Notons que dans les deux cas la variable pile est initialisée à la pile vide lors de sa déclaration.

Une concrétisation permet de soumettre au programme les formules observables issues du formalisme de spécification, mais bien sûr toutes ne sont pas des tests pertinents.

Définition 6
Soit $\gamma_{\Sigma,I} : Obs_{\Sigma,I} \rightarrow Sen_{Prog}(I)$ *une concrétisation correcte.*

Un test abstrait $\varphi \in Obs_{\Sigma,I}$*, ou sa concrétisation* $\gamma_{\Sigma,I}(\varphi)$*, est* **non biaisé** *si et seulement si il est vérifié par tout programme P correct à* $Obs_{\Sigma,I}$ *près par rapport à une spécification donnée (autrement dit,* $P \rightsquigarrow \gamma_{\Sigma,I}(\varphi)$*).*

```
function test2 return boolean is
          pile : Stack ;
          elem : Natural ;
begin
          push(0, pile) ;
          push(2, pile) ;
          poptop(pile, elem) ;
          return elem = 2 ;
exception
          when others =>
                return false ;
end test2 ;
```

Figure 4. *Concrétisation de* $top(push(2, push(0, empty))) = 2$

Les tests non biaisés sont les seuls pertinents. Ils sont directement liés au comportement attendu du programme. Il est donc indispensable de pouvoir les déduire de la spécification.

Théorème 1
Soit SP *une spécification de signature* Σ *dans le formalisme* $Spec$ *et* $\gamma_{\Sigma,I} : Obs_{\Sigma,I} \rightarrow Sen_{Prog}(I)$ *une concrétisation correcte.*

L'ensemble des conséquences observables de la spécification $Obs_{\Sigma,I} \cap Th_{Spec}(SP)$ *est un ensemble de tests non biaisés.*

On note $Th_{Spec}(SP)$ *l'ensemble des conséquences sémantiques de* SP *défini comme* $\{\phi \in Sen_{Spec}(\Sigma) \mid \forall M \in Mod_{Spec}(SP), M \models_{Spec,\Sigma} \phi\}$.

Preuve :
Soit $\varphi \in Obs_{\Sigma,I} \cap Th_{Spec}(SP)$ une conséquence observable quelconque. Par définition de $Th_{Spec}(SP)$, pour tout modèle de la spécification $N \in Mod_{Spec}(SP)$, $N \models_{Spec,\Sigma} \varphi$. Donc par correction de P par rapport à SP à $Obs_{\Sigma,I}$ près, pour tout modèle du programme $M \in Mod_{Prog}(P)$, $M \models_{Spec,\Sigma} \varphi$, car $\varphi \in Obs_{\Sigma,I}$. Puis par correction de $\gamma_{\Sigma,I}$, pour tout modèle du programme $M \in Mod_{Prog}(P)$, $M \models_{Prog,I} \gamma_{\Sigma,I}(\varphi)$. Finalement par définition de $Mod_{Prog}(P)$, $P \rightsquigarrow \gamma_{\Sigma,I}(\varphi)$. □

En conclusion, si l'on dispose d'une concrétisation correcte, alors $Obs_{\Sigma,I} \cap Th_{spec}(SP)$ est un jeu de tests abstraits non biaisés. Si de plus on dispose d'un calcul dans le formalisme de spécification, on pourra construire ces tests.

La correction de la concrétisation est donc nécessaire pour permettre le test. Elle fait partie des hypothèses de test. Là encore, le plus souvent aucune vérification formelle n'est faite sur ce point. Cependant, les vérifications traditionnelles sont généralement suffisantes. En effet une erreur dans la traduction a infiniment plus de chance

de révéler une erreur imaginaire que de masquer une erreur réelle. La correction de la concrétisation et donc du processus de soumission des tests est donc un problème d'ingénierie classique, mais n'est pas un maillon faible de la validation formelle comme cela peut l'être dans le cadre de la génération de code.

Définition 7
Un ensemble de tests concrets $\Gamma \subset Sen_{Prog}(I)$ *est* **complet** *si et seulement si il est non biaisé*[4]*, et si pour tout test non biaisé* $\tau \in Sen_{Prog}(I)$ *et pour tout programme* P *d'interface* I, $P \not\rightsquigarrow \tau$ *implique*[5] $P \not\rightsquigarrow \Gamma$.

Autrement dit un jeu de test est complet s'il est non biaisé et maximal. Le caractère maximal est à prendre au sens où tout test supplémentaire est inutile, car il ne peut détecter de nouvelles erreurs dans le programme.

Théorème 2
Soient $\gamma_{\Sigma,I} : Obs_{\Sigma,I} \rightarrow Sen_{Prog}(I)$ *une concrétisation correcte et complète.*

$\gamma_{\Sigma,I}(Obs_{\Sigma,I} \cap Th_{Spec}(SP))$ *est un jeu de tests complet.*

Preuve :
Soit $\tau \in Sen_{Prog}(I)$ un test concret. Par complétude de $\gamma_{\Sigma,I}$, il existe une formule observable $\varphi \in Obs_{\Sigma,I}$ telle que $\gamma_{\Sigma,I}(\varphi) = \tau$.

Supposons que $\varphi \in Th_{Spec}(SP)$ soit une conséquence de SP. Par définition, $\tau \in \gamma_{\Sigma,I}(Obs_{\Sigma,I} \cap Th_{Spec}(SP))$.

Supposons que $\varphi \notin Th_{Spec}(SP)$. Par définition de $Th_{Spec}(SP)$, il existe un modèle $M \in Mod_{Spec}(SP)$ tel que $M \not\models_{Spec,\Sigma} \varphi$. Donc tout programme P d'interface I tel que $M \in Mod_{Prog}(P)$ vérifie $P \not\rightsquigarrow \tau$. Or par définition de $Mod_{Prog}(P)$, pour tout modèle $N \in Mod_{Prog}(P)$, on a $N \equiv_{Sen_{prog}(I)} M$. Donc par complétude de $\gamma_{\Sigma,I}$, $N \equiv_{Obs_{\Sigma,I}} M$. Donc par définition, P est correct par rapport à SP à $Obs_{\Sigma,I}$ près. □

La correction et la complétude de la concrétisation entraîne donc automatiquement l'existence d'un jeu complet de tests. En l'occurrence, l'ensemble de tous les tests non biaisés.

Définition 8
Un ensemble de tests concrets $\Gamma \subset Sen_{Prog}(I)$ *est* **exhaustif** *si et seulement si son exécution en succès par un programme* $P \rightsquigarrow \Gamma$ *est synonyme de correction*[6] *de* P.

---

4. à observation près par rapport à une spécification donnée
5. Par extension nous noterons $(P \rightsquigarrow)$ le prédicat défini sur $\mathcal{P}(Sen_{Prog}(I))$ vérifié si et seulement si tous les tests de l'ensemble sont vérifiés par P.
6. à observation près par rapport à une spécification donnée

Notons qu'en toute généralité, il n'est pas possible d'affirmer que la concrétisation de $Obs_{\Sigma,I} \cap Th_{Spec}(SP)$ est exhaustive. En effet, le filtre de l'observation peut être déformant. C'est le cas notamment lorsque l'on ne peut conclure au succès d'une exécution isolée, mais seulement à un ensemble (éventuellement infini) d'exécution. Considérons l'exemple très simple d'une spécification qui comprend trois constantes a, b, c et un axiome $a = b \vee a = c$. Puis observons cette spécification par des équations. Le jeu de tests abstraits $Obs_{\Sigma,I} \cap Th_{Spec}(SP)$ est vide (aux tautologies près qui sont ici $a = a$, $b = b$ et $c = c$), puisque ni $a = b$, ni $a = c$ ne peuvent être individuellement requis pour la correction du programme. Un programme dont les trois valeurs a, b et c sont différentes passe donc le jeu complet de tests avec succès bien qu'il soit incorrect par rapport à sa spécification. L'exhaustivité des jeux complets de tests dépend donc de l'adéquation entre les observations possibles $Obs_{\Sigma,I}$ et le formalisme de spécification $Spec$. L'exhaustivité des jeux complets de tests doit donc être établie pour chaque formalisme de spécification et chaque observation. Nous en donnerons des exemples à la section suivante.

En pratique ce jeu complet de tests (exhaustif ou non) ne peut jamais être intégralement soumis. En effet, il est souvent infini et lorsqu'il est fini l'explosion combinatoire due au nombre et à la taille des données le rend impraticable. Cependant, la donnée d'un jeu exhaustif de tests est importante en pratique car elle fournit une procédure de semi-décidabilité de la correction d'un programme. En particulier, lorsque le formalisme de spécification possède un calcul correct et complet, l'ensemble des théorèmes sémantiques $Th_{Spec}(SP)$ est aussi l'ensemble des théorèmes par calcul. On possède donc une procédure semi-décidable de construction des conséquences de SP et donc des tests. De telles procédures ont déjà été implantées dans des générateurs automatiques de tests pour divers langages de spécification, notamment pour les spécifications algébriques conditionnelles positives [MAR 91], pour les spécifications orientées modèles [DIC 93, VAN 97b, LEG 01] et pour les spécifications à automates [PAR 96, MAR 00].

4. Exemples de jeux complets de tests

Nous venons de voir que $Obs_{\Sigma,I} \cap Th_{Spec}(SP)$ constitue un jeu complet de tests abstraits pour une concrétisation correcte et complète. Cependant ce jeu de tests est très vaste et plein de redondances. Pour rendre plus efficace le test et faciliter la phase de sélection (voir la section 5), il est souvent nécessaire de construire un jeu complet mais réduit de tests.

Considérons deux exemples différents de formalismes et d'observations.

4.1. *Fonctions partielles récursives minimalement définies*

Choisissons comme exemple l'un des sous-langages constructifs de CASL définis dans [MOS 98]. Ce langage contient des déclarations de sortes et de constructeurs

totaux libres (en fait des types libres), des déclarations d'opérations partielles et des axiomes de définition des opérations qui sont des équations fortes[7] de la forme :

$$f(u_1, ..., u_n) = t$$

et telles que $u_1, ..., u_n$ soient des termes avec variables construits sur les constructeurs, que les équations soient linéaires (c'est-à-dire ne contiennent pas deux occurrences d'une même variable) et deux à deux sans recouvrement (c'est à dire qu'aucun terme ne filtre deux membres gauches d'axiomes). De plus l'ensemble des opérations partielles (définitions et axiomes) sont placées dans des extensions libres. Ces spécifications ont donc un unique modèle : le modèle initial. La figure 5 présente un exemple de spécification des listes dans ce formalisme.

from Basic/Numbers **version** 0.4 **get** NAT

spec LISTES =
  NAT
then
  free type $List ::= [] \mid [\_ :: \_](Nat; List)$;
then free
{ op $insert : Nat \times Nat \times List \to ? List$;
  vars $n, m, i : Nat$;
  $L : List$
  • $insert(n, 0, L) = [n :: L]$
  • $insert(n, suc(i), [m :: L]) = [m :: insert(n, i, L)]$
}
end

Figure 5. *Spécification CASL des listes avec fonctions partielles minimalement définies*

Choisissons pour ensemble de formules observables $Obs_{\Sigma,I}$ l'ensemble des équations closes et des prédicats de définition (ou leur négation) sur les termes clos.

Théorème 3
Soit SP une spécification dans le formalisme ci-dessus, $Obs_{\Sigma,I}$ une observation et l'ensemble de formules suivantes :

$$\begin{aligned} Test = \{ \quad & f(u_1, ..., u_n) = v \quad / \quad u_1 ... u_n, v \text{ sont construits sur les constructeurs,} \\ & \qquad f \text{ est une opération partielle,} \\ & \qquad \text{et } f(u_1, ..., u_n) = v \in Th_{Spec}(SP) \quad \} \\ \cup \ \{ \quad & \neg Def\ f(u_1, ..., u_n) \quad / \quad \text{il n'existe pas de terme } v \\ & \qquad \text{construit sur les constructeurs tel que} \\ & \qquad f(u_1, ..., u_n) = v \in Th_{Spec}(SP) \quad \} \end{aligned}$$

7. Une équation forte $t = t'$ est validée par un modèle M, soit si t et t' sont tous les deux indéfinis dans M, soit s'ils sont tous les deux définis et interprétés par la même valeur de $|M|$.

$Test$ est un jeu de tests complet pour SP et $Obs_{\Sigma,I}$.

Preuve :
Montrons que $Test$ est non biaisé. D'après la forme de la spécification (équations fortes et extensions libres), un terme t est défini ($Def\ t \in Th_{Spec}(SP)$) si et seulement s'il existe un terme construit sur les constructeurs v tel que $t = v \in Th_{Spec}(SP)$. Donc $Test$ est bien un jeu de tests abstraits ($Test \subset (Obs_{\Sigma,I} \cap Th_{Spec}(SP))$) car toutes les formules de $Test$ sont des équations closes observables et sont des conséquences de la spécification.

Montrons que $Test$ est maximal, au sens où il révèle autant d'erreurs que $Obs_{\Sigma,I} \cap Th_{Spec}(SP)$. Montrons par récurrence sur la structure de t que si t est défini alors $t = v$ est une conséquence de $Test$ et sinon $\neg Def\ t$ est une conséquence de $Test$.

Cas de base : t est une constante ;

1) si t est un constructeur alors $t = t$ est de la bonne forme et est une tautologie, donc $t = t$ est une conséquence de $Test$;

2) si t n'est pas un constructeur : si t est défini alors il existe un terme construit sur les constructeurs v tel que $t = v \in Th_{Spec}(SP)$ et $t = v$ est une formule de $Test$, sinon il n'existe pas de terme construit sur les constructeurs v tel que $t = v \in Th_{Spec}(SP)$ et $\neg Def\ t$ est une formule de $Test$;

Pas de récurrence : t est de la forme $f(t_1, ..., t_n)$;

1) si pour $i \in [1, n]$, t_i est défini, alors par hypothèse de récurrence il existe un terme construit sur les constructeurs u_i tel que $t_i = u_i$ soit une conséquence de $Test$; donc par transitivité $t = f(u_1, ..., u_n)$ est une conséquence de $Test$;

a) si f est un constructeur, le résultat est immédiat ;

b) si f n'est pas un constructeur et t est défini, alors il existe un terme construit sur les constructeurs v tel que $t = v \in Th_{Spec}(SP)$; donc par transitivité $f(u_1, ..., u_n) = v \in Test$; donc $t = v$ est une conséquence de $Test$;

c) si f n'est pas un constructeur et t est indéfini, alors il n'existe pas de terme construit sur les constructeurs v tel que $t = v \in Th_{Spec}(SP)$; donc par transitivité $\neg Def\ f(u_1, ..., u_n) \in Test$; donc $\neg Def\ t$ est une conséquence de $Test$;

2) s'il existe $i \in [1, n]$ tel que t_i est indéfini, alors par hypothèse de récurrence $\neg Def\ t_i$ est une conséquence de $Test$; donc $\neg Def\ t$ est une conséquence de $Test$.

En conclusion, pour toute équation close $t = t'$ de $Obs_{\Sigma,I} \cap Th_{Spec}(SP)$:

– soit t et t' sont définies et il existe un terme v construit sur les constructeurs tel que $t = v$ et $t' = v$ soient aussi des équations de $Obs_{\Sigma,I} \cap Th_{Spec}(SP)$ et donc des conséquences de $Test$; donc $t = t'$ est une conséquence de $Test$;

– soit t et t' sont indéfinies et $\neg Def\ t$ et $\neg Def\ t'$ sont des conséquences de $Test$; donc $t = t'$ est une conséquence de $Test$. □

Les axiomes de ce formalisme sont des égalités fortes qui permettent donc de définir à la fois les cas de définition et les valeurs des opérations partielles. Les cas de non définition sont implicites. Ils sont déduits des cas de définition par complément grâce à l'extension libre.

Reprenons l'exemple de la figure 5. Le jeu de tests défini par le Théorème 3 est le suivant :

$$
\begin{array}{ll}
Test = \{ & insert(n, 0, L) = [n :: L] \\
& insert(n, suc(0), [m :: L]) = [m :: [n :: L]] \\
& insert(n, suc(suc(0)), [m :: [p :: L]]) = [m :: [p :: [n :: L]]] \\
& ... \\
& \neg Def\ insert(n, suc(0), []) \\
& \neg Def\ insert(n, suc(suc(0)), []) \\
& \neg Def\ insert(n, suc(suc(0)), [m :: []]) \\
& ... \\
& / \ n, m, p \text{ soient construits sur les constructeurs de } Nat, \\
& \quad L \text{ soit construit sur les constructeurs de } List \qquad \}
\end{array}
$$

L'orientation de gauche à droite des axiomes de la spécification donne naturellement un algorithme de semi-construction de la première partie du jeu de tests proposé. En effet, la forme des membres gauches d'axiomes impose la confluence de la réécriture lorsque celle-ci termine. Mais l'absence de contrainte sur les membres droits ne permet pas d'assurer la terminaison. La deuxième partie de notre jeu de tests n'est donc constructible qu'en présence de contraintes supplémentaires comme la donnée d'un ordre bien fondé sur les termes qui permet d'assurer la terminaison du système.

4.2. *Observations bornées*

Considérons un formalisme algébrique classique avec opérations totales et constructeurs libres. Pour ces spécifications, les seuls théorèmes construits sur les constructeurs et les variables sont des tautologies (validées par tous les modèles de la signature). Ce type de contraintes peut être assuré syntaxiquement en restreignant les axiomes à des formules conditionnelles positives gracieuses comme dans l'exemple de la figure 6. Pour plus de facilité, cet exemple est présenté en utilisant la syntaxe CASL.

Soit $Bornes$ un ensemble de termes clos construits sur les constructeurs. Il est facile de définir un tel ensemble en bornant le nombre d'appels des différents constructeurs en fonction de la sorte des termes [ARN 96, ARN 97]. Un terme t est **sous les bornes** pour une spécification SP, si et seulement s'il existe un terme $v \in Bornes$ tel que $t = v \in Th_{Spec}(SP)$. Notons que comme la spécification a des constructeurs libres, si un terme clos construit sur les constructeurs est sous les bornes, c'est qu'il appartient à $Bornes$. Nous dirons qu'une formule close est sous les bornes si tous ses termes sont sous les bornes. En pratique, les formules sous les bornes représenteront les traitements normaux ne provoquant pas de débordement de données. Soit S_{Obs}

from Basic/Numbers **version** 0.4 **get** NAT

spec LISTES =
NAT
then
type $List ::= [] \mid [\_ :: \_](first : Nat; rest : List)$;
op $insert : Nat \times List \rightarrow List$;
vars $n, m, i : Nat$;
$L : List$
- $insert(n, []) = [n :: []]$
- $n < m \Rightarrow insert(n, [m :: L]) = [n :: [m :: L]]$
- $n \geq m \Rightarrow insert(n, [m :: L]) = [m :: insert(n, L)]$

end

Figure 6. *Spécification des listes avec constructeurs libres*

un ensemble de sortes observables. En pratique c'est l'ensemble des sortes pour lesquelles on dispose d'une concrétisation de l'égalité. Considérons comme ensemble de formules observables $Obs_{\Sigma,I}$, l'ensembles des formules sous les bornes dont toutes les équations sont observables (sur une sorte observable).

Reprenons l'exemple de la figure 6. Choisissons de borner les entiers naturels à $2^{64} - 1$, c'est à dire les termes construits sur 0 et suc avec au plus $2^{64} - 1$ applications de suc. Et bornons les listes à la longueur 1000, c'est-à-dire les termes construits sur [] et $[\_ :: \_]$ avec au plus 1000 applications de $[\_ :: \_]$ et des entiers naturels sous les bornes. De plus choisissons d'observer la seule sorte Nat.

Rappelons qu'un contexte est un terme à une seule occurrence de variable appelée place. Un contexte est dit observable s'il est de sorte observable et il est dit minimal si aucun de ses sous-contextes[8] n'est observable.

Théorème 4
Soit SP une spécification dans le formalisme ci-dessus et $Obs_{\Sigma,I}$ l'ensemble de formules observables.

$$Test = \{ \quad c[f(u_1, ..., u_n)] = c[v] \quad / \quad \begin{array}{l} u_1, ..., u_n, v \text{ sont des termes de } Bornes, \\ f \text{ est une opération non constructeur}, \\ c[\,] \text{ est un contexte observable minimal} \\ \text{et } f(u_1, ..., u_n) = v \in Th_{Spec}(SP) \quad \} \end{array}$$

$Test$ est un jeu de tests abstraits complet.

Preuve :
Le principe de la preuve est identique à celle du Théorème 3 et peut être trouvée en

8. Un sous-contexte est un sous-terme du contexte contenant la place.

intégralité dans [ARN 97]. □

Remarquons qu'outre les valeurs hors bornes, le jeu de tests ci-dessus ne contient pas nécessairement un test par entrée de fonction. En effet, si une fonction f est incomplètement spécifiée pour une entrée $u_1, ..., u_n$ (c'est-à-dire qu'il n'existe par de terme construit sur les constructeurs v tel que $t = v \in Th_{Spec}(SP)$), alors $Test$ ne contient aucun test de la forme $c[f(u_1, ..., u_n)] = c[v]$. Mais comme l'affirme le Théorème 4, ce n'est pas un oubli. En effet, si la spécification ne précise pas quels doivent être les comportements de certaines parties du programme, il n'y a pas lieu de vérifier ces comportements lors du test de conformité.

Dans le cas d'une spécification dont tous les axiomes sont conditionnels positifs et suivent une présentation gracieuse, le jeu de tests défini par le Théorème 4 est exhaustif [ARN 97]. Reprenons l'exemple de la figure 6 et l'observation proposée précédemment. Le jeu de tests ci-dessous est complet pour tout programme possédant une concrétisation correcte et complète.

$$
\begin{array}{ll}
Test = \{ & c[insert(n, [])] = c[[n :: []]] \\
& c[insert(n, [m :: L])] = c[[n :: [m :: L]]] \\
& c[insert(n, [p :: [m :: L]])] = c[[p :: [n :: [m :: L]]]] \\
& ... \\
& / \ n, m, p, [n :: [m :: L]], [p :: [n :: L]] \text{ soient sous les bornes ,} \\
& \quad c[\,] \text{ soient des contextes minimaux,} \\
& \quad n < m \in Th_{Spec}(Listes), \\
& \quad n \geq p \in Th_{Spec}(Listes) \qquad \qquad \}
\end{array}
$$

Nous venons de voir sur deux exemples qu'il est possible d'épurer le jeu de tests de départ en un jeu complet équivalent. Les deux jeux de tests proposés précédemment sont très proches de la pratique classique du test : on teste une fonction f sur différentes entrées $u_1, ..., u_n$ et on vérifie si son résultat est bien celui attendu (une donnée v ou une non définition). Cette vérification se fait soit directement à travers les prédicats $=$ et $\neg Def$ (ou leur concrétisation) ou soit à travers un oracle plus complexe comprenant des contextes d'observation par exemple.

5. Sélection d'un jeu final de tests

La sélection est la phase de choix d'un nombre raisonnable de tests dans le jeu complet[9]. Cette phase est bien sûr indispensable car même après simplification (voir la section précédente) le jeu complet de tests reste impraticable, voire infini. Elle est de plus cruciale car elle détermine la qualité du jeu final de tests, c'est-à-dire sa capacité à révéler des erreurs dans le programme.

9. D'après le théorème 2 un jeu de tests abstraits ne peut être complet que s'il existe une concrétisation correcte et complète.

Le jeu final de tests n'est qu'une toute petite fraction du jeu complet de départ. Sa soumission avec succès n'assure donc en rien la correction du programme. Par contre l'échec de sa soumission atteste formellement d'une erreur. Le véritable rôle du test n'est donc pas de vérifier la correction d'un programme mais de trouver les erreurs, qui, de fait, existent toujours en pratique. La sélection est issue de l'expertise des ingénieurs dans chaque domaine d'application. Ces derniers fixent les **objectifs de test**, qui précisent les aspects du programme à vérifier. C'est-à-dire celles qui, conformément à leur expérience passée, ont le plus de chance de comporter des erreurs.

Ensuite, il est nécessaire de construire les cas de test qui permettront d'atteindre les objectifs de test. Cette construction est systématique, longue et fastidieuse à réaliser manuellement. Les méthodes formelles permettent d'automatiser cette phase de construction et ainsi de réduire de manière considérable le coût et les délais de la vérification des logiciels. En effet, si le choix des objectifs de test ne peut pas s'appuyer sur des critères formels, une fois ceux-ci posés, leur application et la construction des tests associés repose sur les méthodes formelles.

5.1. *Les critères de couverture*

Les critères de couverture du code source des programmes (ou plus exactement de son graphe de contrôle) sont largement répandus dans les pratiques industrielles. En quelques mots, il s'agit de sélectionner des tests qui permettent "d'activer" toutes les branches, ou tous les enchaînements du programme. De nombreux ateliers de génie logiciel intègrent des vérificateurs de ces critères de couvertures. Leur principe est simple, l'ingénieur sélectionne le critère choisi et saisit ses tests. Au fur et à mesure, l'outil lui indique les parties de code non encore couvertes. Le principal inconvénient de ces outils, outre la construction toujours manuelle des tests, est l'existence de parties du programme impossible à couvrir. En effet, pour des critères simples comme "toutes les branches", le nombre de branches impossibles à couvrir ou branches mortes, n'est guère important et est souvent signe d'erreur dans le programme. Mais pour des critères plus compliqués les parties impossibles à couvrir sont inévitables et en nombre beaucoup plus important. Les critères ne peuvent donc pas être couverts à 100% et cela pose inévitablement le problème de **l'arrêt des tests**. Quand faut-il arrêter de construire de nouveaux tests ? Lorsque l'ingénieur ne réussit plus à couvrir de nouvelles parties ? Lorsque il est convaincu que les parties restantes sont impossibles à couvrir ? Ou lorsqu'il l'a démontré ?

Des critères similaires de couverture des spécifications peuvent être utilisés pour sélectionner les tests, notamment pour les spécifications constructives. Il s'agit de partitionner le jeu de tests de départ en sous-jeux de tests correspondant à chaque partie de la spécification dont la couverture est souhaitée, puis de construire un ou plusieurs tests dans chaque partie du jeu de tests. Les spécifications constructives définissent généralement leurs fonctions par cas (un axiome par cas) et par récurrence. Il est donc possible, comme pour les langages de programmation, de définir des critères de couverture des spécifications fonctionnelles. Ces critères imposent la couverture

de chaque cas de définition des différentes fonctions et limitent le nombre d'appel récursif.

Reprenons l'exemple présenté figure 5 illustrant les fonctions partielles récursives minimalement définies développées dans la section 4.1. La fonction $insert$ d'insertion d'un entier dans une liste à une place donnée est définie par deux axiomes (l'insertion en tête de la liste et dans la suite de la liste) et le cas implicite de non définition de la fonction (l'insertion au delà de la taille de la liste). Le domaine de définition de la fonction $insert$ comprend donc trois sous-ensembles :

1) l'insertion d'un entier à la place 0 dans une liste quelconque,

2) l'insertion d'un entier à une place strictement positive dans une liste non vide,

3) l'insertion d'un entier à une place strictement positive dans une liste vide.

Cette partition du domaine de définition de l'insertion définit directement une partition du jeu complet de tests correspondant (proposé dans le théorème 3). Dans le premier cas, le résultat attendu de l'insertion est complètement défini par rapport aux constructeurs. Dans le troisième cas (implicite) la fonction est indéfinie, le résultat attendu est donc une levée d'exception. Au contraire des deux autres, dans le deuxième cas le résultat attendu de l'insertion est définie récursivement. Cette nouvelle occurrence de la fonction $insert$ définit une partition de son domaine de définition, qui définit une partition plus fine du jeu complet de tests :

1) l'insertion d'un entier à la place 0 dans une liste quelconque,

2.1) l'insertion d'un entier à la place 1 dans une liste à un élément,

2.2) l'insertion d'un entier à la place 2 ou plus dans une liste à au moins 2 éléments,

2.3) l'insertion d'un entier à la place 2 ou plus dans une liste à un seul élément,

3) l'insertion d'un entier à une place strictement positive dans une liste vide.

Dans le cas 2.2) l'insertion est à nouveau définie récursivement et peut donc donner lieu à un nouveau partitionnement.

De telles méthodes permettent aux ingénieurs de définir leur plan de test en définissant précisément le critère de sélection utilisé. Ensuite les méthodes formelles et les techniques de résolution de contraintes permettent de construire automatiquement un ou plusieurs tests par sous-ensemble de la partition exhibée et de détecter les ensembles vides. Ce type de sélection a été implanté par plusieurs logiciels [MAR 91, DIC 93, VAN 97a]. D'autres logiciels proposent des critères de nature différente, comme par exemple la sélection de tests au voisinage des bornes [ARN 99, LEG 01].

5.2. *Les objectifs de test*

Il est souvent intéressant d'exprimer les objectifs de test de manière plus libre. En particulier pour réduire le domaine de test au domaine d'utilisation du logiciel, ou

pour tester certaines configurations particulières. Par exemple, si la fonction $insert$ de la figure 6 est utilisée uniquement pour des listes triées, l'efficacité du test sera accrue si l'on restreint le domaine de test aux seules listes triées. En effet, comme on réduit *a priori* le jeu de tests aux seuls tests concernant des listes triées, l'effort de test sera concentré sur le domaine d'utilisation réel de la fonction. Si le programme manipule de très grandes listes implantées par des listes chaînées de tableau de 100 éléments, il peut être nécessaire de sélectionner spécifiquement des listes de 100, 200 ... éléments pour vérifier que l'insertion dans le tableau suivant s'effectue correctement. Une spécification fonctionnelle comme celle de la figure 6 ne permet pas de synthétiser automatiquement de tels objectifs. Mais elle peut permettre de construire les tests une fois l'objectif exprimé. En particulier en construisant l'oracle associé à chaque entrée de test.

D'autres objectifs peuvent permettre de tester plus spécifiquement certains comportements critiques du programme. Les objectifs de test permettent de caractériser un sous-ensemble du jeu de tests de départ. Ils peuvent être combinés aux découpages guidés par les critères de couverture. Il est bien sûr possible de construire des tests répondant à des objectifs tels que ceux ci-dessus avec [MAR 00] ou avec certains des outils précédemment cités.

6. Conclusion

Dans cet article nous avons établi une passerelle entre les mondes très différents des spécifications algébriques et des langages de programmation. Cette passerelle est basée sur une interprétation algébrique des comportements fonctionnels des programmes. Elle nous a permis tout d'abord de définir formellement la notion de correction (voir aussi [LEG 96]). Puis nous avons concrétisé les tests abstraits issus de la spécification en de réelles procédures de décision du succès ou de l'échec du test dans le langage de programmation cible. Enfin nous avons caractérisé un jeu de tests de référence et montré qu'il cristallise le maximum de garanties en terme de correction du programme par rapport à sa spécification.

Notons que notre approche est basée sur l'équivalence observationnelle introduite pour l'implantation abstraite [ORE 93]. Une autre approche développée dans [MAC 00] est basée sur l'égalité comportementale [BID 95]. [MAC 00] ne s'est guère intéressé ni à la concrétisation des tests, ni aux aspects de sélection de tests. Par contre, [MAC 00] a mené une étude approfondie sur la structuration dans les spécifications et son impact sur la soumission des tests. En particulier, elle définit l'ensemble des tests en accord avec la structure de la spécification, ce qui permet de corréler les tests avec les différents modules du programme.

Dans la seconde partie de cet article, nous avons montré que notre formalisation du test constitue une base solide pour la pratique. Notre approche permet de formaliser la pratique courante du test, à travers des tests dont la forme est familière et des critères de sélection classiques et variés. Des études de cas importantes [DAU 93] en

attestent. En raison de la jeunesse du langage de spécification CASL, il n'a pas encore été l'objet d'utilisation pour la génération de tests. Cependant, l'utilisation d'outils actuels, tel que LOFT [MAR 91], nécessite un simple traducteur syntaxique. De tels traducteurs ont déjà été développés pour d'autres outils (réécriture, preuve ...) et sont présentés sur les pages web du langage CASL, (voir par exemple à ce sujet le site http ://www.tzi.de/cofi/Tools/). Néanmoins, des recherches plus avancées restent à mener pour développer les méthodes et outils de sélections adaptées aux structures avancées du langage (quantificateurs, structuration ...).

Remerciements

Nous remercions les relecteurs pour leurs remarques constructives qui ont largement contribué à l'amélioration de l'article.

7. Bibliographie

[ARN 96] ARNOULD A., LE GALL P., MARRE B., « Dynamic testing from bounded data type specifications », *Dependable Computing - EDCC-2, Second European Dependable Computing Conference*, vol. 1150 de *LNCS*, Taormina, Italy, Octobre 1996, Springer Verlag.

[ARN 97] ARNOULD A., « Test à partir de spécifications de structures bornées : une théorie du test, une méthode de sélection, un outil d'assistance à la sélection », Thèse, LRI, Université de Paris-Sud, 91405 Orsay, Janvier 1997.

[ARN 99] ARNOULD A., MARRE B., LE GALL P., « Génération automatique de tests à partir de spécifications de structures de données bornées », *Technique et Science Informatiques*, vol. 18, n° 3, 1999, p. 297-322.

[BAR 88] BARNES J., *Programmer en ADA*, InterEditions, 1988.

[BER 91] BERNOT G., GAUDEL M., MARRE B., « Software testing based on formal specifications : a theory and a tool », *Software Engineering Journal*, vol. 6, n° 6, 1991.

[BID 95] BIDOIT M., HENNICKER R., WIRSING M., « Behavioural and abstractor specifications », *Science of Computer Programming*, vol. 25, n° 2-3, 1995, p. 149-186.

[BRI 88] BRINKSMA E., « A theory for the derivation of tests », *8th International Conference on Protocol Specification, Testing and Verification*, Atlantic City, North-Holland, 1988.

[CoF 01] COFI (COMMON FRAMEWORK INITIATIVE) TASK GROUP ON LANGUAGE DESIGN, « CASL The Common Algebraic Specification Language Summary, version 1.0.1 », rapport, march 2001, ftp ://ftp.brics.dk/Projects/CoFI.

[DAU 93] DAUCHY P., GAUDEL M.-C., MARRE B., « Using Algebraic Specifications in Software Testing : a case study on the software of an automatic subway », *Journal of Systems and Software*, vol. 21, n° 3, 1993, p. 229-244.

[DIC 93] DICK J., FAIVRE A., « Automating the generation and sequencing of test cases from model-based specifications », *FME'93 : Industial-Strenth Formal Methods, First International Symposium of Formal Methods Europe*, vol. 670 de *LNCS*, Odense, Denmark, April 1993, Springer Verlag, p. 268-284.

[DSS 87] DSSOULI R., BOCHMANN G., « Conformance testing with multiple observers », *Protocol Specification Testing and Verification, North-Holland*, 1987, p. 217-229.

[GOG 92] GOGUEN J., BURSTALL R., « Institutions : Abstract Model Theory for specification and Programming », *Journal of the Association for Computing Machinery*, vol. 39, n° 1, 1992, p. 95-146.

[GRO 96] GROZ R., RENÉVOT J., CHARLES O., SALVAIL P., « A tool for assessing the test coverage of protocol conformance test suites », *Conf. on Distributed Computing Systems*, Hong-Kong, may 1996.

[HAL 91] HALBWACHS N., RAYMOND P., RATEL C., « Generating Efficient Code From Data-Flow Programs », *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[LEG 96] LE GALL P., ARNOULD A., « Formal specification and test : correctness and oracle », *Recent Trends in Data Type Specification*, vol. 1130 de *LNCS*, Springer Verlag, 1996, p. 342-358, 11th Workshop on Specification of Abstract Data Types joint with the 9th general COMPASS workshop. Oslo, Norway, September 1995, Selected papers.

[LEG 01] LEGEARD B., PEUREUX F., « Génération de séqences de test à partir d'une spécification B en PLC ensembliste », *AFADL'2001*, 2001, p. 113-130.

[MAC 00] MACHADO P., « Testing from Structured Algebraic Specifications », *AMAST2000*, vol. 1816 de *LNCS*, 2000, p. 529-544.

[MAR 91] MARRE B., « Toward automatic test data set selection using Algebraic Specifications and Logic Programming », *ICLP'91, Eigth International Conference on Logic Programming*, Paris, France, 1991, MIT Press, p. 25-28.

[MAR 00] MARRE B., ARNOULD A., « Test Sequences Generation from LUSTRE Descriptions : GATEL », *Proceedings of ASE-00 : The 15th IEEE Conference on Automated Software Engineering*, Grenoble, Sep 2000, IEEE CS Press.

[MES 89] MESEGUER J., « General logics », *Proc. Logic. Colloqium '87*, Amsterdam, North-Holland, 1989.

[MOS 98] MOSSAKOWSKI T., « Two "Functional Programming" Sublanguages of CASL », CoFI Note n° L-9, 1998, http ://www.brics.dk/Projects/CoFI.

[ORE 93] OREJAS F., NAVARRO M., SANCHEZ A., « Implementation and Behavioural Equivalence : A Survey », *Recent Trends in Data Type Specification*, vol. 655 de *LNCS*, Springer Verlag, 1993, p. 144-163, 8th Workshop on Specification of Abstract Data Types joint with the 3rd COMPASS Workshop, Dourdan,

[PAR 96] PARISSIS I., OUABDESSELAM F., « Specification-based Testing of Synchronous Software », GARLAN D., Ed., *SIGSOFT'96 : Proceeding of the Fourth ACM SIGSOFT Symposium on the Foundatins of Software Engineering*, vol. 21 de *Software Engineering Notes*, San Francisco, California, USA, 1996, ACM.

[PHA 94] PHALIPPOU M., « Relations d'implantation et hypothèses de test sur des automates à entrées sorties », Thèse, Université de Bordeaux I, 1994.

[ROG 00] ROGGENBACH M., MOSSAKOWSKI T., SCHRODER L., « Basic Datatypes in CASL », CoFI Note n° L-12 - Version 0.4.1, May 2000, http ://www.brics.dk/Projects/CoFI/Notes/L-12/index.html.

[VAN 97a] VAN AERTRYCK L., BENVENISTE M., LE METAYER D., « Casting : a formally based software test generation method », *proc. IEEE Int. Conference on Formal Engineering Methods*, 1997, p. 101-111.

[VAN 97b] VAN AERTRYCK L., BENVENISTE M., LE METAYER D., « CASTING : une méthode formelle de génération automatique de cas de tests », *AFADL'97 : Approches Formelles dans l'Assistance au Développement de Logiciels.*, Toulouse, ONERA-CERT, 1997.

[WIR 90] WIRSING M., *Algebraic Specification*, vol. B de *Handbook of Theoretical Computer Science*, MIT Press, 1990.

Article reçu le 24 juillet 2001
Version révisée le 8 avril 2002
Rédacteur responsable : Farid Ouabdesselam

Agnès Arnould *est Maître de Conférences à l'université de Poitiers, dans l'équipe Signal, Image et Communications (IRCOM-SIC, UMR 6615 du CNRS). Elle effectue sa recherche sur les méthodes formelles et outils associés, notamment dans le domaine du test. Elle applique ses recherches au développement formel de modeleurs géométriques à base topologique.*

Pascale Le Gall *est professeur à l'université d'Evry. Ses centres d'intérêt concernent les applications des méthodes formelles pour le génie logiciel (test à partir de spécifications, spécifications hétérogènes). Les domaines d'application visés sont les services en télécommunication et la modélisation géométrique.*