Couverture de code Caml pour la réalisation d'outils de développement certifiés

Bruno PAGANO[1], Benjamin CANOU[2], Emmanuel CHAILLOUX[3], Jean-Louis COLAÇO[4] et Philippe WANG[5]

1: Esterel Technologies,
8 rue Blaise Pascal, 78890 Elancourt.
`Bruno.Pagano@esterel-technologies.com`

2: ENS Cachan, antenne de Bretagne
Campus Ker Lann, F-35170 Bruz
`Benjamin.Canou@eleves.bretagne.ens-cachan.fr`

3 : Laboratoire d'informatique de Paris 6 (LIP6),
CNRS UMR 7606, Université Pierre et Marie Curie, Paris 6
`Emmanuel.Chailloux@lip6.fr`

4: Esterel Technologies,
9 rue Michel Labrousse, Park Avenue, 31100 Toulouse.
`Jean-Louis.Colaco@esterel-technologies.com`

5: Université Pierre et Marie Curie, Paris 6
`Philippe.Wang@etu.upmc.fr`

Résumé

Cet article est un retour d'expérience d'une étude menée sur l'utilisation du langage Objective Caml pour la réalisation d'outils de développement de logiciel critique. Dans le cas d'espèce, il s'agit d'un générateur de code embarqué pour le langage Scade^{TM}. Même si les contraintes pour la réalisation d'outils sont moins fortes que celles qui pèsent sur le code embarqué, elles demeurent néanmoins assez lourdes et liées à la nature des langages impératifs ordinairement utilisés pour ce type de développement. L'usage d'Objective Caml sort du cadre ordinaire autant par ses traits de haut niveau (langage fonctionnel d'ordre supérieur, polymorphisme paramétrique, filtrage par motif) que par les mécanismes de bas niveau mis en œuvre par la bibliothèque d'exécution (GC, exceptions). Dès lors, il est nécessaire de réinterpréter pour ce langage les normes de développement que la certification pour le logiciel critique exige, de développer les outils capables de mesurer le respect à ces normes et d'adapter OCaml pour que lui même les satisfasse. Nous proposons une limitation du langage et la simplification de sa bibliothèque d'exécution qui permettent de définir et de mesurer la couverture d'un programme écrit en OCaml selon les critères MC/DC. Cela ouvre des perspectives d'une diffusion plus large de ce type de langage en milieu industriel en élevant le niveau d'abstraction dans la conception des outils développés pour la production de programmes certifiés.

1. Introduction

La notion de logiciel critique (*safety-critical*) est traditionnellement associée aux systèmes de contrôle embarqués mais on la retrouve maintenant dans d'autres domaines où la sûreté du logiciel est une dimension prépondérante. Des niveaux de criticité ont été définis et standardisés selon le risque qu'un incident logiciel fait courir à ses utilisateurs. Parmi les applications les plus classiques, on peut citer les commandes de vol d'un avion, le système de signalisation d'un réseau ferré, le contrôle d'une centrale nucléaire, mais aussi les systèmes informatiques médicaux ou l'ABS[1] d'un frein de voiture. Leur point commun est que leur dysfonctionnement peut provoquer une « catastrophe » dont les conséquences peuvent être léthales pour des individus en relation avec ce système.

Le domaine de l'avionique civile s'est doté depuis une vingtaine d'années d'une procédure de certification pour le code embarqué dans les avions. La norme DO-178B [20][2] définit toutes les contraintes présidant à l'écriture d'un logiciel destiné à un aéronef. Cette procédure est incluse dans un processus global de certification d'un avion. Cette norme se décline maintenant pour d'autres secteurs industriels concernés par le logiciel critique (FDA Class III pour l'industrie médicale, IEC 61508 pour l'industrie automobile, etc.)

La certification DO-178B impose des procédures très méticuleuses dont l'activité prépondérante est la vérification indépendante de chaque phase du développement. Nous ne nous intéresserons dans cet article qu'au développement du logiciel et aux procédures de traque des erreurs qu'il pourrait éventuellement contenir ; mais la portée de la norme DO-178B déborde largement de cet aspect. Le développement du code tel qu'il est reconnu par les organismes de certification suit le traditionnel cycle de développement en V cher aux tenants du génie logiciel ; les contraintes en sont renforcées mais le principe reste le même : les spécifications du produit sont écrites par raffinements successifs en commençant par l'expression des besoins, puis les spécifications externes, la conception globale, la conception détaillée, jusqu'à enfin obtenir le code. À chacune des étapes est associée une phase de validation indépendante reliant un pour un les spécifications et les tests.

Le processus suivi pour réaliser du code embarqué satisfaisant une telle certification impose la définition d'un « standard de codage » que les développements respecteront. Ce standard doit préciser un ensemble de règles strictes pour la définition des spécifications, pour leur implantation et pour la traçabilité entre spécifications et réalisations. En particulier, le standard de codage doit faire apparaître l'obligation de couvrir le code entièrement. La certification DO-178B impose que cette couverture soit effectuée selon les critères de mesure MC/DC [12] (*Modified Condition/Decision Coverage*).

La norme DO-178B s'applique aux outils de développement de code embarqué de la même manière qu'à ce code lui même. C'est-à-dire que le développement de l'outil devra suivre son propre standard de codage. Celui-ci pourra être moins exigeant que pour du code embarqué ; l'allocation dynamique ou l'utilisation de fonctions récursives sont admissibles dans un outil et refusés quand le code est embarqué.

Dans ce contexte, le développement d'outils dans un langage particulier doit lui aussi suivre les contraintes posées par la norme DO 178B ; c'est-à-dire avoir une couverture MC/DC du source du programme. De la même manière, la bibliothèque d'exécution (*runtime*), s'il en existe une, doit être certifiée. Pour un développement en C, cela se traduira par une maîtrise des appels à la `libc` dans le code et par une vérification des mécanismes mis en œuvre par le compilateur. Pour des langages plus modernes comme Java, cela nécessiterait la certification de l'ensemble de sa (très grande) bibliothèque.

Le langage Objective Caml (OCaml dans la suite) est particulièrement bien adapté pour la construction de compilateurs et d'outils formels d'analyse de programme ; outre lui-même[16], il est utilisé dans les compilateurs de langages synchrones à la Lustre comme Lucid Synchrone [18] ou dans le système d'aide à la preuve Coq[19]. Il y a dix ans, l'utilisation du langage Caml dans l'ingénierie logicielle pour le développement de programmes temps réels sûrs intéressait déjà des acteurs industriels de l'avionique (Dassault [7]). L'expérience de Surlog avec AGFL[2] montre qu'OCaml peut s'intégrer dans un cycle de développement de logiciels critiques et qu'il y apporte son modèle bien fondé. Avec Astrée [10], OCaml fait la preuve de son adéquation à la réalisation d'outils pour le logiciel critique.

La société Esterel-Technologies commercialise SCADE[3, 5] qui est un environnement de développement de logiciel embarqué. Cet atelier intègre un générateur de code transformant des programmes Lustre en programmes C qui a été développé en respectant tous les standards de la DO 178B. Ce compilateur a été réalisé en C, mais depuis 2001, Esterel-Technologies prototype son prochain générateur de code en OCaml. Ce choix a permis de tester de nouvelles techniques de compilation[9] et des extensions de langages[8]. Il est

[1] Anti-lock Braking System

[2] norme américaine, son équivalent européen est l'ED-12B

apparu qu'OCaml permettait de réduire la distance entre les spécifications de l'outil et son implantation ce qui améliore la traçabilité entre spécification et code exigée par les standards de développement DO 178B. L'objectif de l'étude menée pour Esterel-Technologies a été d'explorer les moyens et les outils pour certifier du code écrit en OCaml.

Dans un milieu industriel plus classique produisant du logiciel critique, ne connaissant traditionnellement que C ou Ada et dont le processus de développement passe par un usage intensif des tests, l'intrusion d'OCaml change la formulation des problématiques de qualification. Plusieurs traits du langage surprennent l'automaticien ou le programmeur impératif, en premier lieu le fait qu'OCaml soit un langage d'expressions, mais aussi les traits de plus haut niveau comme le polymorphisme paramétrique, le filtrage par motif, la gestion des exceptions et le mécanisme de récupération automatique de mémoire[14] (*Garbage Collector* ou GC).

À l'inverse, les notions de couverture de code et des conditions/décisions sont définies pour des langages impératifs d'instructions comme le langage C. Il faut donc adapter cette notion aux expressions booléennes Caml. Par contre pour ce qui concerne le polymorphisme et le filtrage par motif, la notion de couverture MC/DC doit être précisée pour traiter ces deux traits les plus marquants du langage Caml. À cela il est aussi nécessaire de mettre en conformité la bibliothèque d'exécution avec le standard de codage. Il convient d'apporter une traçabilité entre les spécifications et le code de la bibliothèque d'exécution tant pour le contrôle (mécanisme général d'application, exceptions) que pour la gestion mémoire automatique. Cela rend difficile l'utilisation du *runtime* d'origine de la distribution Inria du langage et milite pour la construction d'un *runtime* alternatif compatible.

La section 2 développe la problématique du test en milieu industriel, tout particulièrement la mesure MC/DC de couverture de code. La section 3 définit le sous-ensemble du langage OCaml étudié pour ensuite préciser la notion de mesure de couverture que l'on veut associer au code Caml. La section 4 répond aux besoins de certification de la bibliothèque d'exécution pour compléter les propriétés de couverture à l'ensemble de l'exécutable. Il est alors temps de conclure par une discussion sur les différentes chaînes de production de code Caml pour une telle certification.

2. Validation dans un contexte certifié

L'aviation civile américaine (FAA) exige pour donner l'autorisation de voler à un avion que tout programme informatique qu'il embarque respecte les normes de développement DO 178B. Selon la criticité du logiciel, c'est-à-dire selon qu'il s'agisse des commandes de vol ou des commandes du lecteur de DVD de chaque fauteuil, cette norme se décline sur plusieurs niveaux (A, B, C, etc.). La DO 178B est très rigoureuse sur les procédures à suivre mais ne donne aucune contrainte sur le langage de programmation à utiliser ni même sur les paradigmes que celui-ci doit suivre. Pour autant, il existe des règles qui viennent préciser cette norme et qui restreignent drastiquement les types de programmes acceptés. Au niveau A qui est le plus haut niveau de criticité, un programme embarqué ne peut faire d'allocation dynamique de mémoire, ni employer de récursion, ni faire quoi que ce soit qui ne soit borné statiquement en temps et en espace avec de surcroît une borne connue. Pour ce genre de développement, l'utilisation de Caml ou de tout autre langage généraliste d'ailleurs, est hors de propos. L'usage est d'utiliser un langage d'assemblage ou des sous-ensembles limités de C ou de ADA.

Cependant, il est permis et de plus en plus indispensable, que ces logiciels soient conçus avec des générateurs de code ou vérifiés avec des outils automatisés ; mais à la condition que ces outils soient eux-mêmes certifiables DO 178B au même niveau de criticité. À titre d'exemple, la mesure de couverture dont nous parlons dans la suite peut être faite manuellement par une relecture indépendante du code ou par un outil si celui-ci se soumet aux exigences de la DO 178B.

Dans le cadre de la réalisation de tels outils, il est admis de « relâcher » les règles les plus contraignantes mais les exigences restent fondamentalement les mêmes. Si par exemple on autorise la récursion ou l'allocation dynamique, il est tout de même requis de circonscrire l'usage de l'outil au domaine où le tas et la pile vont être suffisants pour une utilisation nominale. Même si à la différence d'un programme embarqué, un outil peut échouer dans son exécution, il est indispensable de montrer qu'il ne produira jamais un résultat faux. C'est à ce titre que la phase de validation prend une place prépondérante dans le cycle développement.

Tests : critères de mesure de couverture Dans un processus industriel, la phase de validation prend place après le développement et vise à établir la correspondance entre le produit et sa spécification. On appelle *mesure de couverture*, la quantification objective de la relation entre une activité de validation et les objectifs qu'elle s'est fixée ; elle est habituellement exprimée comme le pourcentage de l'activité effectué. La finalité de la mesure de couverture est de permettre aux auditeurs de s'assurer que la validation a été menée jusqu'à son terme.

La norme DO 178B définit plusieurs activités de validation et parmi celles-ci plusieurs activités de tests. Une base de tests doit être constituée pour couvrir l'ensemble des spécifications du logiciel qu'on souhaite valider. La *mesure de couverture structurelle* qui nous intéresse en premier lieu, détermine comment le code est exécuté par cette base de tests et établit la *traçabilité* entre la structure du code et les cas de tests. Il est demandé par la norme que toute partie du code ait été utilisée dans l'exécution d'un test et que son utilisation ait été conforme aux exigences de la spécification.

Les critères de couverture structurelle se divisent en deux ensembles : ceux sur le flot de données et ceux sur le flot de contrôle. L'analyse du flot de données consiste à caractériser par une métrique particulière le lien entre l'assignation d'une variable et son utilisation. La norme DO 178B n'exige pas de critères particuliers pour la mesure de couverture structurelle du flot de données ; nous allons dans la suite nous focaliser sur celle du flot de contrôle.

Le flot de contrôle est mesuré sur les instructions exécutées, les expressions booléennes évaluées et les branches des instructions de contrôle empruntées. Dans ce qui suit, nous citons les principales mesures.

Couverture des lignes de code (*Statement Coverage*) C'est sans doute le critère le plus simple à comprendre et à mettre en œuvre, il s'agit de vérifier que pour chaque instruction du programme il existe un test pour lequel cette instruction est exécutée. C'est un critère considéré comme trop faible comme le montre le prochain exemple.

Couverture des décisions (*Decision Coverage*) On appelle *décision*, l'expression évaluée dans une instruction de branchement pour décider quelle branche sera exécutée. Classiquement, il s'agit d'une expression booléenne dont il faut s'assurer qu'il existe un test pour lequel sa valeur calculée est `true` et un autre test où la valeur est `false`.

L'exemple C suivant illustre la différence entre les deux critères de couverture :

```
int abs (int x) { if (x<0) x = -x;  return x; }
```

Un seul test avec une valeur négative suffit pour couvrir toutes les instructions par contre la couverture des décisions nécessite un second test avec une valeur positive pour que la condition `(x<0)` prenne aussi la valeur `false`.

Couverture des conditions (*Condition Coverage*) On appelle *condition* les sous-expressions atomiques d'une décision. Par exemple, la décision `x && (y<0) || f(z)` est constituée de trois conditions `x`, `y<0` et `f(z)`. Une condition est couverte s'il existe des tests où cette condition prend les valeurs `true` et `false`.

Couverture C/DC (*Condition/Decision Coverage*) La couverture C/DC cumule les deux précédents critères.

Couverture MC/DC (*Modified Condition/Decision Coverage*) La couverture MC/DC étend le critère C/DC en introduisant l'exigence que chaque condition doit *indépendamment affecter* la valeur de la décision. Plus prosaïquement, chaque condition doit recevoir deux tests qui donnent les mêmes valeurs aux autres conditions et qui couvrent la décision par les deux valeurs possibles.

Couverture des conditions multiples (*Multiple Condition Coverage*) Enfin, il s'agit dans ce critère de fournir des tests pour toutes les combinaisons possibles de valeurs des conditions.

Nous allons illustrer ces définitions par un exemple et par les tests que ces critères de mesure requièrent. Nous examinons une instruction de test figurant quelque part dans un programme : `if ((a || b ) && c) { ... }`. Il existe huit tests différents pour stimuler cette instruction qui sont les huit valuations des trois variables booléennes a, b et c. Nous qualifierons ces valuations et le résultat de l'expression de vecteurs de test et nous les noterons `[TTT T]`, `[FTT T]`, `[TFT T]`, `[FFT F]`, `[TTF F]`, `[FTT F]`, `[TFT F]` et `[FFF F]`. Ils correspondent à la table de vérité de l'expression.

Selon le type de couverture mesurée, les tests exigés ne sont pas les mêmes :
- couverture des instructions : un seul test prenant la valeur T est exigé.
- couverture de la décision : deux tests, l'un donnant T et l'autre F, suffisent : par exemple `[TTT T]` et `[FTT F]`.
- couverture des conditions : chaque condition doit prendre les deux valeurs, donc deux tests qui diffèrent sur toutes les conditions suffisent : `[TTF F]` et `[FFT F]` (on note que cet ensemble ne couvre pas la décision).
- couverture C/DC : toujours deux tests qui diffèrent sur toutes les conditions mais qui diffèrent en plus sur la valeur globale : `[TTT T]` et `[FFF F]`.
- couverture MC/DC : pour chaque condition, il est nécessaire d'exhiber deux tests qui ne diffèrent que pour cette condition et qui font prendre à la décision les deux valeurs de vérité. Par exemple, le couple de vecteurs `[TFT T]` et `[FFT F]` montre l'indépendance de la condition a. Les vecteurs peuvent servir pour l'indépendance de plusieurs variables. En général, pour N conditions, il faut N+1 vecteurs de test. Dans cet exemple, quatre vecteurs sont suffisants : `[TFT T]`, `[FTT T]`, `[TFF F]` et `[FFT F]`.
- couverture des conditions multiples : les huit vecteurs sont nécessaires.

La certification DO 178B au niveau A requiert que l'intégralité du code ait une couverture MC/DC de 100%. Le critère MC/DC est apparu comme un compromis raisonnable entre une exigence trop faible de deux tests et une exigence inatteignable de 2^n tests.

La pertinence de la couverture MC/DC est un sujet abondamment débattu[11, 15]. Notre propos est de montrer quel sens cette mesure prend en OCaml sachant que les organismes de certification de l'aviation civile l'exigent. Il faut comprendre que l'analyse MC/DC du code est un des maillons d'un processus qui s'emploie à valider chacune des étapes du développement. Même s'il est théoriquement possible de contourner les analyses de couverture par des « astuces » de codage, en pratique ces « astuces » seront rejetées soit par ceux en charge de la relecture du code, soit par ceux qui valident la mesure MC/DC.

3. Couverture de code Caml

Comme le disent Chilenski et al.[12], la couverture de code n'est pas un test mais une mesure. Dans cet esprit, nous définissons dans cette section la mesure de couverture pour un programme Caml. Nous poserons les principes de l'analyse des critères MC/DC pour les expressions du langage Caml, en particulier pour la structure de contrôle originale que constitue le filtrage par motif.

Nous avons choisi dans le cadre de cette étude de nous restreindre au noyau fonctionnel et impératif d'OCaml. Nous avons choisi de ne pas traiter les traits les plus modernes du langage mais nous discuterons de leur pertinence dans la conclusion.

3.1. Couverture des expressions

Caml est un langage d'expressions : la notion de couverture doit être adaptée en conséquence. En lieu et place de la couverture des instructions nous nous intéresserons à la couverture du calcul des expressions.

Dans un langage impératif, la couverture doit prendre en compte les instructions de contôle pour vérifier indépendamment que chaque branche est exécutée. Avec Caml, la problématique est très similaire avec les constructions du langage où certaines sous-expressions (l'expression conditionnelle par exemple) peuvent ne pas être évaluées.

```
if (x<y) {
  min = f(x) ;
} else {
  min = f(y) ;
}
```

```
let min =
  if x<y
    then f x
    else f y
```

Au même titre que la couverture du programme C mesure quelles sont les branches du `if` qui ont été exécutées, la couverture du programme Caml doit examiner quelles sont les sous-expressions de l'opérateur `if` qui ont été évaluées.

Une instrumentation du code est nécessaire pour tracer cette information ; elle est légèrement différente de celle réalisée par un outil de profilage (comme `ocamlcp`) puisqu'ici on ne cherche pas à savoir si un calcul a été atteint mais à vérifier qu'il s'est terminé. Le principe général est de remplacer toute expression `expr` que l'on souhaite tracer par l'expression `(let aux = expr in mark() ; aux)` où `aux` est une variable qui n'est pas libre dans `expr` et `mark` une fonction capable par effet de bord de conserver l'information que ce point du programme a été atteint. Sous la réserve que la fonction `mark` soit indépendante du programme initial et qu'elle termine, l'expression instrumentée et l'expression originale ont le même type et la même sémantique.

Nous formalisons cette transformation en lui adjoignant deux cas particuliers qui sont : si l'expression est atomique (un littéral ou une variable) son évaluation ne peut pas échouer et si l'expression est de type `unit`, il n'est pas nécessaire de sauvegarder sa valeur. Ces cas particuliers ne sont pas fondamentaux, mais ils permettent d'alléger la transformation.

a	$\longrightarrow$	`mark() ; a`	où a est un atome
e	$\longrightarrow$	`e ; mark()`	où `e` est de type `unit`
e	$\longrightarrow$	`let aux = e in mark() ; aux`	

Le cas de l'atome peut être généralisé à toute expression pour laquelle on sait statiquement que son évaluation n'échouera jamais ; c'est-à-dire que son évaluation termine et qu'aucune exception ne peut être levée pendant ce calcul. Sachant que la bibliothèque d'exécution de Caml est susceptible de lever des exceptions, notamment si l'allocation dans la pile ou dans le tas échoue, les expressions satisfaisant cette propriété se limitent principalement au cas des atomes.

À l'exemple de l'opérateur `if`, un certain nombre de constructions primitives du langage Caml introduisent plusieurs branches de calcul. L'analyse de la couverture des expressions nécessite de tracer l'évaluation de chacune des branches indépendamment des autres. Pour éviter le « sur-marquage », on décompose cette instrumentation par deux transformations qui explorent l'arbre de syntaxe d'une expression. La transformation $\mathscr{G}$ parcourt l'arbre et appelle $\mathscr{F}$ sur les sous-expressions que l'on souhaite marquer. À l'inverse, la transformation $\mathscr{F}$ parcourt l'arbre et appelle $\mathscr{G}$ sur les sous-expressions que l'on ne désire pas marquer mais dont on souhaite marquer les éventuelles branches.

$\mathscr{F}$(a)	=	`mark() ; a`	si a est un atome ou une variable
$\mathscr{F}$(e_1 e_2)	=	let aux = $\mathscr{G}$(e_1) $\mathscr{G}$(e_2) in `mark()` ; aux	
$\mathscr{F}$(fun x -> e)	=	`mark()` ; fun x -> $\mathscr{F}$(e)	
$\mathscr{F}$(e_1 ; e_2)	=	$\mathscr{G}$(e_1) ; $\mathscr{F}$(e_2)	
$\mathscr{F}$(if e_1 then e_2 else e_3)	=	if $\mathscr{G}$(e_1) then $\mathscr{F}$(e_2) else $\mathscr{F}$(e_3)	
$\mathscr{F}$(while e_1 do e_2 done)	=	while $\mathscr{G}$(e_1) do $\mathscr{F}$(e_2) done ; `mark()`	
$\mathscr{F}$(for x = e_1 to e_2 do e_3 done)	=	for x = $\mathscr{G}$(e_1) to $\mathscr{G}$(e_2) do $\mathscr{F}$(e_3) done ; `mark()`	
$\mathscr{F}$(match e with p_1 -> e_1 | ... | p_n -> e_n)	=	match $\mathscr{G}$(e) with p_1 -> $\mathscr{F}$(e_1) | ... | p_n -> $\mathscr{F}$(e_n)	
$\mathscr{F}$(let x = e_1 in e_2)	=	let x = $\mathscr{G}$(e_1) in $\mathscr{F}$(e_2)	
$\mathscr{F}$(try e with p_1 -> e_1 | ... | p_n -> e_n)	=	try $\mathscr{F}$(e) with p_1 -> $\mathscr{F}$(e_1) | ... | p_n -> $\mathscr{F}$(e_n)	

La principale différence entre les deux est que seule $\mathscr{F}$ marque la fin du calcul d'une expression ; les deux instrumentent toutes les branches contenues dans l'expression traduite.

$$\begin{array}{rcl}
\mathscr{G}(\text{a}) & = & \text{a} \\
\mathscr{G}(\text{e}_1\ \ \text{e}_2) & = & \mathscr{G}(\text{e}_1)\ \ \mathscr{G}(\text{e}_2) \\
\mathscr{G}(\text{fun x -> e}) & = & \text{fun x -> } \mathscr{G}(\text{e}) \\
\mathscr{G}(\text{e}_1\ ;\ \text{e}_2) & = & \mathscr{G}(\text{e}_1)\ ;\ \mathscr{G}(\text{e}_2) \\
\mathscr{G}(\text{if e}_1 \text{ then e}_2 \text{ else e}_3) & = & \text{if } \mathscr{G}(\text{e}_1) \text{ then } \mathscr{F}(\text{e}_2) \text{ else } \mathscr{F}(\text{e}_3) \\
\mathscr{G}(\text{while e}_1 \text{ do e}_2 \text{ done}) & = & \text{while } \mathscr{G}(\text{e}_1) \text{ do } \mathscr{F}(\text{e}_2) \text{ done} \\
\mathscr{G}(\text{for x = e}_1 \text{ to e}_2 \text{ do e}_3 \text{ done}) & = & \text{for x = } \mathscr{G}(\text{e}_1) \text{ to } \mathscr{G}(\text{e}_2) \text{ do } \mathscr{F}(\text{e}_3) \text{ done} \\
\mathscr{G}(\text{match e with p}_1 \text{ -> e}_1 \mid ... \mid \text{p}_n \text{ -> e}_n) & = & \text{match } \mathscr{G}(\text{e}) \text{ with p}_1 \text{ -> } \mathscr{F}(\text{e}_1) \mid ... \mid \text{p}_n \text{ -> } \mathscr{F}(\text{e}_n) \\
\mathscr{G}(\text{let x = e}_1 \text{ in e}_2) & = & \text{let x = } \mathscr{G}(\text{e}_1) \text{ in } \mathscr{G}(\text{e}_2) \\
\mathscr{G}(\text{try e with p}_1 \text{ -> e}_1 \mid ... \mid \text{p}_n \text{ -> e}_n) & = & \text{try } \mathscr{G}(\text{e}) \text{ with p}_1 \text{ -> } \mathscr{F}(\text{e}_1) \mid ... \mid \text{p}_n \text{ -> } \mathscr{F}(\text{e}_n)
\end{array}$$

Comme les autres opérateurs (accesseurs, affectation, etc..), les opérateurs booléens ne sont pas détaillés dans ces transformations mais traités comme des applications ce qui suffit à établir leur couverture simple. Nous verrons dans la section 3 le traitement particulier qui leur est réservé dans le cadre d'une analyse MC/DC de la couverture.

Récursion terminale La récursion terminale n'est pas une spécificité d'OCaml ou des langages fonctionnels. C'est une propriété des appels récursifs d'une fonction qui est exploitée par les compilateurs pour optimiser le code objet généré. Cette optimisation s'applique lorsque l'appel récursif d'une fonction est la dernière opération exécutée de cette fonction et consiste à remplacer le mécanisme d'appel classique par un simple branchement. Elle permet d'une part d'accélérer le code produit mais surtout d'économiser de l'espace dans la pile d'appel.

L'instrumentation que nous proposons, consistant à ajouter un effet de bord spécifique à la trace après chaque expression, brise systématiquement l'appel terminal et interdit l'optimisation. Ainsi, le programme instrumenté pourra nécessiter plus d'espace mémoire sur la pile d'exécution que le même programme non instrumenté.

En conséquence, les tests nécessaires à la couverture devront être choisis de sorte à ne pas faire exploser la pile. Quand ceci n'est pas possible, il devient nécessaire de dérécursiver le code.

Correction de l'instrumentation Sachant que `mark()` est de type `unit`, les transformations opérées par les fonctions $\mathscr{F}$ et $\mathscr{G}$ ne modifient pas le type de l'expression traduite.

Dans le même ordre d'idée, ces transformations ne font que rajouter `mark()` à certains endroits de l'expression. Il est facile de montrer que l'ajout de cet effet de bord ne modifie pas la valeur calculée par l'expression initiale.

La couverture d'une application est mesurée par l'examen des traces construites lors de l'exécution des fonctions `mark` introduites par la tranformation $\mathscr{F}$ appliquée à l'ensemble du code. Donc, un programme est couvert à 100%, si et seulement si, il existe des tests couvrant toutes les marques.

3.2. Couverture MC/DC

Il n'existe par réellement une définition précise de la couverture MC/DC ; cette notion est toujours plus ou moins dépendante du langage sur lequel elle s'opère et de l'outil que l'on utilise. En particulier, la notion de MC/DC est définie sur les expressions booléennes mais la couverture est effectuée sur les instructions de branchement.

Le point de vue adopté pour OCaml est différent. Nous ne nous intéressons pas à l'aspect syntaxique des expressions mais à leur type. Toute expression de type `bool` est analysée et instrumentée pour en mesurer sa couverture MC/DC et ce quelque soit le lieu où cette expression prend place. Une exception est faite pour les littéraux booléens `true` et `false`.

Décomposition d'une expression Le typage des expressions permet de caractériser les décisions ; une analyse de la structure de l'expression nous permet ensuite de préciser les conditions qui la composent. Nous nous contenterons d'isoler les arguments des opérateurs booléens `not`, `&&` et `||`. Par exemple :

`a && (f x) || not (b || c)` est composé de quatre décisions : `a`, `(f x)`, `b` et `c`.

L'instrumentation d'une décision doit conserver la sémantique de l'expression initiale. Les conditions n'étant pas obligatoirement des expressions fonctionnelles pures, l'instrumentation du code doit conserver l'ordre d'évaluation des conditions.

Nous proposons une transformation des expressions booléennes qui fait apparaître explicitement les différentes branches de calcul de cette expression et par la même, le vecteur correspondant à chaque branche.

$$\widehat{\text{expr}} = \mathscr{F}_b(\text{expr, true, false})$$

$$\begin{array}{rcl}
\mathscr{F}_b(\texttt{x},\ l,\ r) & = & \texttt{if x then } l \texttt{ else } r \\
\mathscr{F}_b(\texttt{not e},\ l,\ r) & = & \mathscr{F}_b(\texttt{e},\ r,\ l) \\
\mathscr{F}_b(\texttt{e}_1\ \texttt{\&\&}\ \texttt{e}_2,\ l,\ r) & = & \mathscr{F}_b(\texttt{e}_1,\ \mathscr{F}_b(\texttt{e}_2,\ l,\ r),\ r) \\
\mathscr{F}_b(\texttt{e}_1\ ||\ \texttt{e}_2,\ l,\ r) & = & \mathscr{F}_b(\texttt{e}_1,\ l,\ \mathscr{F}_b(\texttt{e}_2,\ l,\ r))
\end{array}$$

Cette traduction transforme une expression booléenne en une imbrication de `if` dont chaque test est une condition et dont les feuilles sont les constantes `true` ou `false`. Chaque feuille est un des vecteurs possibles (une valuation pour certaines conditions et la valeur de l'expression pour cette valuation). De la sorte, la couverture de chaque feuille nous indique quel vecteur a été couvert, ainsi, un outil analysant cette trace est à même de déterminer si l'indépendance de chaque condition a été atteinte.

Cette vision est plus restrictive que celle donnée par Chilenski[12] car elle autorise moins de vecteurs de test. En fait, elle ne distingue pas deux vecteurs qui ne diffèrent que sur des conditions qui ne sont pas atteintes par le flot de contrôle. Au final nous obtenons une mesure dont la couverture à 100% implique la couverture à 100% selon le critère MC/DC.

L'implantation naïve de la transformation proposée ci-dessus consistant à expliciter sous forme d'une imbrication de `if`, l'arbre du flot de contrôle d'une expression booléenne peut produire dans le pire cas, c'est-à-dire celui d'une alternance de conjonctions et de disjonctions, un arbre dont la taille est exponentielle par rapport à celle de l'expression. Une implantation moins naïve de cette transformation, décorant directement l'expression booléenne évitera cette explosion en conservant le partage des sous-expressions.

Remarquons que certaines conditions peuvent ne pas apparaître dans certaines branches, ce qui correspond à une condition qui n'est pas évaluée dans ce cas.

Un exemple booléen L'expression booléenne suivante est composée de quatre conditions indépendantes. Ce sont ici des variables mais elles peuvent être remplacées par des expressions plus complexes.

```
let pred a b c d = (a || b) && (c || d)
```

Il existe 2^4 tests possibles. Obtenir sa couverture simple en demande deux et sa couverture MC/DC en nécessite cinq. La transformation en `if` fait apparaître sept cas de tests.

```
let pred a b c d =                                      (*     a b c d  -> p *)
  if a then if c then true                              (* 1 : T _ T _  -> T *)
            else if d then true                         (* 2 : T _ F T  -> T *)
                      else false                        (* 3 : T _ F F  -> F *)
     else if b then if c then true                      (* 4 : F T T _  -> T *)
                      else if d then true               (* 5 : F T F T  -> T *)
                                else false              (* 6 : F T F F  -> F *)
            else false                                  (* 7 : F F _ _  -> F *)
```

Trouvons pour chaque condition, les paires de tests qui suffisent pour montrer l'indépendance de chaque variable, i.e. les paires qui ne diffèrent que par la valeur de la condition et le résultat global :

a : (1)+(7) b : (4)+(7) c : (1)+(3) ou (4)+(6) d : (2)+(3) ou (5)+(6)

Ce qui nous donne comme ensembles minimaux : {(1), (2), (3), (4), (7)} ou {(1), (4), (5), (6), (7)}

La couverture MC/DC complète peut-être établie avec cinq tests ce qui est conforme au résultat attendu. Nous notons qu'il n'est pas nécessaire de couvrir entièrement la forme exprimée par des conditionnelles pour obtenir le critère MC/DC.

3.3. Autres opérateurs dans une expression booléenne

Nous n'avons considéré pour l'instant que les décisions formées par la combinaison des opérateurs `&&`, `||` et `not`, les autres sous-expressions étant considérées comme des conditions. Dans cette section nous considérons les autres fonctions ou opérateurs prédéfinis dont le résultat est un booléen.

Prédicat Quand une telle fonction, dont le résultat est un booléen, intervient dans une décision, elle est considérée comme une seule condition :

```
let divide x y = (y mod x = 0) || (x mod y = 0)
let decision = cond1 || cond2 || (divide a b)
```

Dans cet exemple, quatre variables interviennent dans la décision mais nous ne définissons que trois conditions. Ce fait est connu et accepté dans la couverture MC/DC. Notons tout de même que bien que l'application de la fonction `divide` soit considérée comme une seule condition dans la décision, la fonction `divide` est elle-même instrumentée puisque son résultat est un booléen et la couverture MC/DC fera naître la nécessité de trois tests distincts pour la couvrir (x divise y, y divise x et ni l'un ni l'autre).

Dans le cas de l'utilisation d'un prédicat, la couverture de ce prédicat est traitée indépendamment de la couverture des décisions qui l'utilisent.

Fonction polymorphe Les fonctions polymorphes manipulent potentiellement des valeurs booléennes mais ne peuvent pas être instrumentées dans le cas général.

L'application à un booléen de la fonction `couple` qui suit, devrait être instrumentée conformément à ce qui précède afin de couvrir son usage pour les deux valeurs `true` et `false`.

```
let couple x = (x,x)          (* couple : 'a -> 'a * 'a *)
let cpl_true = couple true
```

Cependant, les fonctions polymorphes possèdent une propriété forte qui est celle de ne pas explorer la valeur de leurs arguments polymorphes. Dans l'exemple ci-dessus, la définition de `couple` n'exploite pas le fait qu'on lui passe des booléens et donc a fortiori la valeur de ces booléens. Les arguments d'une fonction polymorphe ne peuvent intervenir dans une expression booléenne et donc le corps de la fonction n'a aucune raison d'être instrumentée dans ce sens.

Il existe malheureusement une exception qui est la fonction prédéfinie `compare` de type `'a -> 'a -> int`. Celle-ci, bien que polymorphe, explore les valeurs qui lui sont passées. Notons que cette fonction est impossible à définir directement en OCaml. Cette fonction est utilisée dans la définition des opérateurs de comparaison prédéfinis (=, <, etc.). Vu la nature fonctionnelle du langage, il peut-être difficile de vérifier qu'une de ces fonctions n'est pas utilisée dans un prédicat.

Il est toujours possible de monomorphiser le code incriminé, soit de façon manuelle soit de façon automatique. Dans le premier cas, il convient de refuser l'unification d'une variable de type d'un paramètre par un type contenant `bool`. L'inconvénient est que cette approximation rejette inutilement certains programmes.

Dans le second cas, l'outil d'instrumentation peut monomorphiser les fonctions posant problème. Mais alors l'instrumentation ne peut se faire qu'en connaissant l'ensemble de ces fonctions ; c'est-à-dire l'ensemble du programme. De plus, la monomorphisation duplique le code des fonctions polymorphes ce qui complexifie la notion de couverture : doit-on couvrir le code source ou chaque instance monomorphe du code ?

Afin de ne pas introduire de code, nous avons choisi la première solution consistant à interdire l'instanciation vers un type contenant `bool` d'une variable du schéma de type de la fonction. Cela signifie rejeter l'application de `couple` à un booléen. Si par contre, `couple` avait été défini comme suit, c'est-à-dire comme une fonction monomorphe, son code aurait été instrumenté et son application aurait été acceptée.

```
let couple (x : bool) = (x,x)     (* couple : 'bool -> 'bool * 'bool *)
let cpl_true = couple true
```

3.4. Mesure MC/DC pour Caml : filtrage par motif

Le filtrage par motif (*pattern matching*) est une particularité et une des richesses des langages de la famille ML. C'est une structure de contrôle en même temps qu'un moyen de déstructurer une valeur complexe. De plus, les analyses statiques [17], faites par le compilateur assurent d'un certain nombre de bonnes propriétés.

Définition des conditions dans un filtrage La difficulté est de faire correspondre conditions et décisions aux éléments syntaxiques des motifs d'un filtrage. Intuitivement, la présence d'un constructeur dans un filtre peut être associé à une condition sur la valeur filtrée. Pour autant, tous les constructeurs ne représentent pas une condition car certains sont redondants avec le fait que les filtres précédents ne sont pas satisfaits.

Dans la fonction suivante qui détermine si une liste est vide, selon l'écriture choisie nous pouvons avoir plus ou moins de constructeurs qui apparaissent dans les filtres :

```
let is_empty = function
  | [] -> true
  | _ -> false
```

```
let is_empty = function
  | [] -> true
  | _::_ -> false
```

La formulation de droite fait apparaître le constructeur `::` dans le filtre mais celui-ci ne constitue pas effectivement un test puisqu'il est redondant avec le premier filtre. En conséquence, leur indépendance est strictement impossible à observer et le critère MC/DC ne peut être atteint.

Une première approche serait de rejeter ce type de formulation et d'imposer au programmeur de n'écrire que des filtres qui ne se recoupent pas. Cette approche est à déconseiller car elle ne contribue pas à la clarté du code ; le *pattern matching* est précieux en ce qu'il permet d'exprimer une définition par cas. Rendre distincts les filtres conduit à introduire une partie de la compilation du *pattern matching* dans le programme source. D'autre part, le filtre ne sert pas seulement à introduire un test mais aussi à déstructurer la valeur filtrée. Dans un tel cas, ce test qui n'existe pas dans la compilation n'est même pas une intention du programme et en faire la couverture perd de son sens.

Nous proposons une seconde approche qui est de ne considérer comme une condition que les constructeurs qui apparaissent dans un filtre et pour lequel il y a un choix ; c'est-à-dire pour lequel le fait que les précédents filtres aient échoué ne force pas la présence de ce constructeur. Examinons l'exemple suivant :

```
let rec length_even = function
  | [] -> true                     (* 1 *)
  | _::[] -> false                 (* 2 *)
  | _::_::l -> (length_even l)     (* 3 *)
```

Le premier filtre contient un seul constructeur qui correspond à une condition. Le second filtre contient deux constructeurs (`::` et `[]`), mais le premier n'est pas une condition puisque si une liste n'est pas vide (ce qui a été établi par le premier filtre), elle a obligatoirement ce constructeur en tête. Le troisième filtre quant à lui contient deux constructeurs mais aucune condition.

En résumé, nous obtenons pour ce filtrage deux conditions qui peuvent s'exprimer ainsi :

1. la liste est vide
2. la liste n'est pas vide et sa queue est vide

L'indépendance de ces deux conditions se montre avec les trois tests que la formulation sous forme d'expressions booléennes nous donnait.

Définition des décisions dans un filtrage Reprenons maintenant la définition inversée de `length_even` et examinons les conditions qu'elle contient :

```
let rec length_even = function
  | _::_::l -> (length_even l)  (* 1 *)
  | _::[] -> false              (* 2 *)
  | [] -> true                  (* 3 *)
```

Nous obtenons cette fois trois conditions, les deux constructeurs du premier filtre et le constructeur `::` du second. Il y a de toute évidence une indépendance impossible à démontrer puisque deux des trois conditions sont identiques, à savoir « la liste n'est pas vide ».

La notion d'indépendance des conditions est trop forte pour le filtrage de motifs. En effet, répéter le même début du motif d'un filtre à l'autre est tout à fait classique et raisonnable dans un programme OCaml. Il n'y a aucune raison de l'interdire et cela conduirait à compliquer le code et à lui faire perdre sa lisibilité.

Nous proposons de ne pas considérer l'instruction de filtrage comme une seule décision mais comme un ensemble de décisions : la première décision correspond à la satisfaction du premier motif, la seconde décision correspond au second motif sachant que le premier n'est pas satisfait, etc. Il y a donc autant de décisions que de motifs.

Concrètement, cela vise à restreindre le critère de couverture à l'indépendance des conditions d'un même motif. On demande à une condition d'être indépendante des autres conditions du même filtre mais pas de la totalité.

```
let rec size_equal = function
  | [] , []          -> true
  | _::l1 , _::l2  -> size_equal (l1, l2)
  | _ , _            -> false
```

Le constructeur de couples n'est pas une condition du filtrage puisqu'il est unique. Donc nous avons deux conditions dans la premier filtre et deux autres dans le second.

On note `[]` la liste vide et `[1] [2]` des listes non vides quelconques.

- obligation de test pour le filtre `[] , []` :
  - pour la couverture simple : `[] []`
  - pour l'indépendance : `[] []` , `[] [1]` et `[1] []`
- obligation de test pour le filtre `_::l1 , _::l2` :
  - pour la couverture simple : `[1] [2]`
  - pour l'indépendance : `[1] [2]` , `[] [1]` et `[1] []`
- obligation de test pour le filtre `_ , _`
  - simple = MC/DC : `[] [1]` ou `[1] []`

En résumé : quatre tests sont nécessaires `[] []` , `[1] []` , `[] [1]` , `[1] [2]`.

Dans cet exemple, il existe une exigence MC/DC supplémentaire pour `(size_equal (l1, l2))` qui impose de façon indépendante de donner comme exemples deux listes non vides de même taille et deux listes non vides de tailles différentes. Du fait de l'appel récursif, ces deux exemples vont servir pour trois des quatre tests précédents. Au final, cet exemple nécessite trois tests distincts, la première liste est plus grande, plus petite ou de même taille que la seconde.

Filtrage par motif des exceptions Le cas particulier des exceptions et de l'instruction `try with` se traite de la même manière que le cas général à la différence près que dans le cas de la récupération d'une exception on suppose par construction qu'il existe un dernier motif ramassant toutes les valeurs non filtrées par les précédents motifs et redéclenchant l'exception pour qu'elle soit traitée plus en amont du flot de contrôle.

3.5. Implantation d'un instrumenteur

Dans le cadre de cette étude, un outil a été prototypé pour réaliser l'instrumentation de programmes Caml et l'édition de rapports de couverture.

La couverture simple peut se faire de manière syntaxique à la manière de `ocamlcp` sous la forme d'un pré-processeur. Par contre, l'instrumentation pour MC/DC nécessite de disposer de l'information de typage d'une part pour détecter les expressions booléennes mais aussi pour détecter l'emploi de booléens comme arguments de fonctions polymorphes. Il s'est avéré plus simple de réécrire un typeur se limitant au sous-ensemble considéré de Caml que de reprendre le typeur de la distribution.

L'outil implante la couverture MC/DC des expressions booléennes mais pas encore celle du filtrage par motif. Dans [17], Maranget expose l'algorithme utilisé par le compilateur OCaml pour vérifier l'exhaustivité du filtrage et pour exhiber d'un cas non traité. Cet algorithme peut être repris pour construire l'instrumentation du filtrage proposé ci-dessus.

4. Certification de programmes Caml

Rappelons que la certification doit s'effectuer sur l'exécutable de l'application. D'une part cela correspond aux analyses faites sur le programme source et notamment la mesure de couverture MC/DC. D'autre part, il est nécessaire d'effectuer les mêmes analyses sur les bibliothèques utilisées dans l'application. En particulier, la bibliothèque d'exécution doit être entièrement spécifiée, respecter les standards de codage et être couverte par une analyse MC/DC.

Le *runtime* est l'ensemble des fonctions de base appelées par le code généré depuis Caml. Elles sont écrites dans un langage de bas niveau. Elles s'interfacent avec le système d'exploitation ou fournissent les primitives indispensables à l'exécution du code Caml compilé (gestion mémoire, appel de fonctions, gestion des exceptions, etc.).

La difficulté de spécifier et de vérifier un programme de bas niveau comme l'est un *runtime* nous conduit à ne pas conserver en l'état celui d'origine mais de le restreindre aux parties exploitées par l'application à certifier et de le simplifier.

4.1. Restriction du *runtime*

En premier lieu le *multi-threading* n'apporte pas de fonctionnalités nécessaires pour les outils de compilation ou d'analyse statique qui sont les objets visés par notre étude.

En second lieu, plusieurs modules de bas niveau manipulant de manière fine la représentation des données ne sont pas indispensables. Par exemple les opérations de sérialisation et désérialisation peuvent transgresser le système de types[13].

Ces allégements permettent aussi de simplifier les contraintes sur la récupération automatique de mémoire. Il n'est plus nécessaire de prendre l'ensemble des piles de chaque thread comme ensemble de racines. De même les pointeurs faibles[3] ne nous sont pas apparus comme indispensables alors qu'ils nécessitent un traitement particulier par le GC.

4.2. Certification d'un *runtime*

Le compilateur OCaml effectue de nombreuses hypothèses sur le *runtime* pour produire un code efficace, tant sur la représentation des données que sur les mécanismes de contrôle comme l'application ou la rupture de calcul (exceptions). Le *runtime* d'OCaml est écrit en C (avec quelques bribes d'assembleur), ceci permet d'y appliquer des outils de couverture MC/DC dédiés à ce langage. C'est une bibliothèque connue pour ses bonnes performances ; la contrepartie est qu'elle contient du code complexe à spécifier, principalement en ce qui concerne la gestion de la mémoire.

L'expérience qui a été menée a consisté au remplacement du GC d'origine[4] par un nouveau Stop&Copy[14] plus simple à écrire, plus simple à comprendre et donc plus simple à certifier.

Nous avons obtenu une couverture complète du code du GC en utilisant le programme KB (Knuth-Bendix) qui éprouve la partie fonctionnelle du langage et son mécanisme d'exceptions. Les cas d'erreurs (dépassement du tas) ont été obtenus séparément.

L'expérience nous a montré qu'il existait une imbrication forte du compilateur OCaml et de son *runtime* :

– Le GC actuel et le reste du *runtime* sont fortement imbriqués. En particulier, une quantité non négligeable de fonctions C modifie ou teste les drapeaux (*tags*) des blocs alloués dans le tas.
  Lors de la réécriture du GC, nous avons pu réécrire ces fonctions afin qu'elles ne fassent plus d'hypothèses sur les blocs de mémoire, et ainsi rendre ces fonctions indépendantes du GC.
– Nous avons dû garder certaines variables du GC et les comportements associés car le compilateur OCaml engendre du code modifiant directement ces variables. Cela implique que le *runtime* n'est pas suffisamment abstrait.
  Nous n'avons pas résolu ce problème car cela aurait nécessité une modification du compilateur, ce qui n'était pas désiré, mais il serait simple d'y pallier car le code engendré l'est simplement pour des raisons de performances : il s'agit d'expansion (*inlining*) d'expressions qui pourraient être remplacées par des appels de fonctions à ajouter au *runtime*, redonnant ainsi son étanchéité au *runtime*.
– En plus des primitives nécessaires au langage, une partie des fonctions de la bibliothèque standard est située dans le code du *runtime*. Leur séparation rendrait plus aisée la constitution de la base de tests pour couvrir les éléments de la bibliothèque de manière indépendante.

Le fait d'avoir débranché le GC d'origine d'OCaml pour le remplacer par une version algorithmiquement plus simple a permis de montrer que l'on pouvait s'abstraire de l'implantation du générateur de code natif pour ne s'intéresser qu'à la façon dont le code généré s'interface avec le *runtime*. Le nouveau *runtime* pourrait être encore simplifié si l'imbrication avec le compilateur n'était pas si forte.

Comme attendu, les performances d'un Stop&Copy, même très simple, restent bonnes tant que la partie conservée du tas n'atteint pas un pourcentage trop important.

5. Conclusion

Dans la communauté francophone des langages applicatifs, les arguments habituels de qualité, de sûreté et d'efficacité de code écrit en Caml sont depuis longtemps acceptés. Pour autant, convaincre les autorités de certification demande de se plier à leurs critères de mesure de la qualité. Cette expérience a montré que les

[3] On s'aperçoit alors que ces restrictions interdisent au compilateur de s'auto-amorcer (*bootstrap*) puisqu'il utilise ces particularités (comme les pointeurs faibles dans le typage).

[4] Gc incrémental à deux générations muni d'un Stop&Copy pour la jeune génération et d'un Mark&Sweep pour la génération ancienne pouvant être compactant en cas de fragmentation

notions de couverture MC/DC se déclinent pour le noyau fonctionnel/impératif du langage et pour un *runtime* simplifié. Bien que cela soit insuffisant pour imaginer d'embarquer du code écrit en Caml, cette satisfaction des critères de la DO 178B rend OCaml apte comme langage de spécification et d'implantation d'outils pour la conception de logiciels critiques.

La société Esterel-Technologies commercialise un compilateur du langage Scade écrit en C qui est passé par la certification DO 178B. Ce générateur de code est utilisé par Airbus pour une bonne partie des logiciels critiques embarqués. Esterel-Technologies poursuit les travaux issus de cette étude dans le but de réaliser et de certifier un nouveau compilateur écrit en OCaml pour la prochaine version du langage Scade. Pour ce faire, un outil d'analyse de couverture de code Caml, issu du prototype de l'étude, est en cours de réalisation et sera certifié comme outil de vérification DO 178B niveau C. Il en est de même pour un *runtime* OCaml alternatif qui sera lui certifié DO 178B niveau A.

Les traits plus modernes du langage OCaml ne sont pas nécessairement souhaités par les organismes de certification pour la conception de logiciels critiques. Par exemple la programmation par objet, à la C++, n'est pas encore acceptée par la DO 178B, ou avec de grandes réserves. Il n'est pas certain que le polymorphisme de rangées de l'extension objet d'OCaml satisfasse tous leurs critères. Dans la même veine, les variants polymorphes apportent une notion d'extensibilité qui est peu compatible avec l'approche du logiciel critique qui doit spécifier exhaustivement toutes les structures de données manipulées. Par contre la généricité des modules paramétrés est intéressante pour ce type d'outils, mais il faut alors lui apporter les mêmes restrictions que celles du polymorphisme paramétrique en les répercutant au niveau de l'application de foncteur. Ces restrictions sont en cours d'analyse.

L'approche proposée dans cet article est de s'intéresser directement aux programmes OCaml et au compilateur de l'Inria. Il convient de limiter d'une part le langage principalement aux traits fonctionnels et impératifs munis d'un système de modules simples et d'autre part la bibliothèque d'exécution (GC, threads, pointeurs faibles, sérialisation, ...). Une difficulté restante est alors de pouvoir expliquer les schémas de compilation des constructions du langage et de leurs compositions.

Une autre approche de la certification de programmes OCaml serait d'utiliser un compilateur ML vers C [6, 1] puis de certifier le code produit en utilisant les outils prévus pour C. Là encore, la principale difficulté est de pouvoir vérifier le GC utilisé ; les GC à racines ambiguës utilisant la pile C [4] ou non [6] risquent de ne pas passer « statistiquement » la certification. Les GC simples à certifier comme un `Stop&Copy`[14] ne sont pas adaptés à C dans la mesure où ils déplacent les objets, en particulier ceux qui considèrent la pile C comme ensemble de racines.

Une troisième approche, qui peut s'inscrire dans cette démarche, est d'utiliser un interprète de code-octet (*byte-code*). Son avantage est de donner un meilleur contrôle tant de la gestion de la pile que celle des exceptions. De plus, l'interprète offre une possibilité d'analyser la couverture d'un programme à son exécution et non par son instrumentation. On peut agrémenter l'interprète d'un expanseur JIT (*Just in Time*) comme [21] pour en améliorer les performances. Là aussi un JIT est plus facile à détailler pour une phase de certification qu'un compilateur complet, principalement parce que ses optimisations sont moins complexes.

Bien que différentes ces trois approches permettent une introduction de langages de haut niveau pour la réalisation d'outils dans la conception de logiciels embarqués dans une optique de gain de temps de développement et de simplification de la procédure de certification.

Références

[1] J. Antonucci, B. Canou, A. Pierard et P. Wang. « Compilateur caml pédagogique », juin 2006. Rapport de projet de master STL, université Pierre et Marie Curie - Paris 6.

[2] P. Ayrault, T. Hardin et M. Guesdon. « Méthodologie de développement d'un outil d'évaluation de la sûreté du logiciel en langage OCaml ». *In Actes des Journées francophones des langages applicatifs*.

INRIA, janvier 2000.

[3] Gérard BERRY. « The Effectiveness of Synchronous Languages for the Development of Safety-Critical Systems ». Rapport Technique, Esterel-Technologies, 2003.

[4] HJ. BOEHM, M. WEISER et J. F. BARTLETT. « Garbage Collection in an Uncooperative Environment ». *Software - Practice and Experience*, septembre 1988.

[5] Jean-Louis CAMUS et Bernard DION. « Efficient Development of Airborne Software with SCADE Suite^{TM} ». Rapport Technique, Esterel-Technologies, 2003.

[6] Emmanuel CHAILLOUX. « An Efficient Way of Compiling ML to C ». *In Workshop on ML and its Applications*. ACM SIGPLAN, juin 1992.

[7] Emmanuel CHAILLOUX et Guy COUSINEAU. « Ingéniérie logicielle pour le développement de programmes temps-réel sûrs ». Rapport de contrat MESR 92S0766, 1994.

[8] Jean-Louis COLAÇO, Bruno PAGANO et Marc POUZET. « A Conservative Extension of Synchronous Data-flow with State Machines ». *In ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, sep 2005.

[9] Jean-Louis COLAÇO et Marc POUZET. « Clocks as First Class Abstract Types ». *In Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, oct 2003.

[10] P. COUSOT, R. COUSOT, J. FERET, L. MAUBORGNE, A. MINÉ, D. MONNIAUX et X. RIVAL. « The ASTRÉE Analyser ». *In European Symposium on Programming*. LNCS, avril 2005.

[11] Arnaud DUPUY et Nancy LEVESON. « An Empirical Evaluation of the MC/DC Coverage Criterion on the HETE-2 Satellite Software ». *In Digital Aviations Systems Conference (DASC)*, Philadelphia, Pennsylvania, USA, oct 2000.

[12] Kelly J. HAYHURST, Dan S. VEERHUSEN, John J. CHILENSKI et Leanna K. RIERSON. « A Practical Tutorial on Modified Condition/Decision Coverage ». Rapport Technique, NASA/TM-2001-210876, mai 2001.

[13] G. HENRY, M. MAUNY et E. CHAILLOUX. « Typer la désérialisation sans sérialiser les types ». *In Actes des Journées francophones des langages applicatifs*. INRIA, janvier 2006.

[14] Richard JONES et Rafael LINS. *Garbage Collection*. Wiley, 1996.

[15] Kalpesh KAPOOR et Jonathan P. BOWEN. « Experimental Evaluation of the Variation in Effectiveness for DC, FPC and MC/DC Test Criteria. ». *In ISESE*, pages 185–194. IEEE Computer Society, 2003.

[16] Xavier LEROY. « *The Objective Caml system release 3.09 : Documentation and user's manual* », 2005. (`caml.inria.fr`).

[17] Luc MARANGET. « Les avertissements du filtrage ». *In Journées Francophones des Langages Applicatifs*. INRIA, janvier 2003.

[18] Marc POUZET. « *Lucid Synchrone version 3.0 : Tutorial and Reference Manual* », 2006. (`www.lri.fr/~pouzet/lucid-synchrone`).

[19] The Coq Development Team LogiCal PROJECT. « *The Coq Proof Assistant Reference Manual* », 2006. (`coq.inria.fr/V8.1beta/refman`).

[20] RTCA/DO-178B. « Software Considerations in Airborne Systems and Equipment Certification », décembre 1992. Radio Technical Commission for Aeronautics RTCA.

[21] Basile STARYNKEVITCH. « OCamljit - a faster Just-In-TIme Ocaml implementation ». *In Workshop MetaOcaml*, juin 2004.