Méthodologie de développement d'un outil d'évaluation de la sûreté du logiciel en langage OCaml

P. Ayrault[1] & T. Hardin[2] & M. Guesdon[1]

1: SURLOG S.A.
VELIZY PLUS - Bat E
1 bis, rue du petit-clamart
78140 Velizy-Villacoublay
surlog@club-internet.fr

2: Laboratoire Informatique Paris 6
LIP6, Thème SPI
8, rue du capitaine Scott
75015 PARIS
Therese.Hardin@lip6.fr

Résumé

AGFLⓡ est un outil d'analyse statique de logiciels sûrs de fonctionnement. Nous présentons la méthodologie de développement de cet outil en vue de sa qualification. Cette méthode s'appuie très classiquement sur un référentiel normatif industriel. Notre particularité est d'avoir choisi des langages applicatifs, OCaml pour le développement et Coq pour la preuve de correction. Nous montrons comment, à chaque étape du cycle de développement, nous avons bénéficié des particularités de ces langages (sémantique définie formellement, typage fort, langage de types très expressif, inférence de type, ...) pour garantir un niveau de sûreté élevé pour le produit logiciel obtenu. Le développement de l'outil AGFLⓡ prouve que l'utilisation de ces langages applicatifs pour le développement d'outils dans le milieu industriel est tout à fait pertinente. Le choix de ces langages participe au respect des exigences qualité nécessaires pour la qualification de l'outil. De plus, la réalisation d'AGFLⓡ nous a permis de mettre en place un atelier complet de développement pour le langage OCaml, que nous réutilisons pour d'autres développements.

1. Introduction

La Fiabilité, la Maintenabilité, la Disponibilité et la Sûreté (FMDS) des systèmes critiques automatisés (métros, trains, centrales nucléaires, avions, ...) dépendent de manière cruciale de la sûreté de fonctionnement de leurs composants logiciels. La qualité et la sûreté de fonctionnement de tels logiciels sont établies par des critères normatifs dont il faut vérifier l'application.

La société SURLOG S.A. propose un ensemble de compétences, de méthodes et d'outils pour l'analyse, l'évaluation et le contrôle de la sûreté de fonctionnement de tels logiciels. Nos clients comptent parmi les organismes en charge de certifier les équipements, les donneurs d'ordres, les exploitants de systèmes et les constructeurs.

Dans le but d'optimiser ses missions d'évaluation, la société SURLOG S.A. a développé l'outil

AGFLⓡ [1] avec l'aide de l'ANVAR [2] et en collaboration avec l'équipe SPI [3] du LIP6. Cet outil effectue des analyses statiques de composants logiciels. Celles-ci nécessitent la manipulation et la transformation de structures de données complexes. Pour le développement d'AGFLⓡ, nous avions l'obligation, et de maîtriser nos coûts de développement et de validation, et d'obtenir un logiciel ayant un bon niveau de qualité. Ces contraintes nous ont conduit à nous éloigner des choix classiques du milieu industriel (C++, Java, ...) et à nous tourner vers le langage OCaml développé à l'INRIA [4].

Le choix du langage de développement étant réalisé, il restait à choisir la méthode de vérification des principes. La spécification d'AGFLⓡ est décrite par de (nombreuses) règles d'inférence. Nous devions d'abord nous assurer de la bonne adéquation de la spécification de l'outil (respect de la sémantique des programmes manipulés). Nous devions, dans un deuxième temps, assurer la correction de son implémentation. Pour cela, il nous fallait choisir un langage pour prouver formellement les propriétés de la spécification. Nous avons choisi l'assistant d'aide à la preuve Coq développé également à l'INRIA.

Cet article commence par décrire succinctement la méthode outillée AGFLⓡ (section 2), puis nous présentons les critères ayant motivé le choix du langage OCaml pour le développement de cet outil (section 3). La section 4 présente la méthodologie mise en place ainsi que la description de chacune des étapes. Ensuite, nous présentons, en section 5, les outils OCaml utilisés ou développés dans le cadre de ce projet. Nous terminons sur les perspectives qui se dégagent au terme du développement et de la qualification de l'outil AGFLⓡ.

2. Présentation de la méthode outillée AGFLⓡ

La confiance en la sûreté de fonctionnement d'un logiciel s'établit en réalisant des analyses de sûreté (AMDE [5] du logiciel, revues formelles, tests de robustesse, ...) tout au long de son cycle de développement. Pour réaliser ces logiciels critiques, les constructeurs ont chacun leurs propres méthodes de développement dans le respect des référentiels normatifs applicables. Notre démarche d'évaluateurs indépendants est que, pour garantir la validité de nos analyses de sûreté, nous devons étudier le logiciel d'un point de vue différent de celui du constructeur. Les méthodes d'analyse employées doivent permettre, entre autres, de mettre à jour les possibles incomplétudes des méthodes de développement des constructeurs. Par conséquent, nous avons décidé, en 1994, de développer nos propres méthodes outillées, et en particulier la méthode AGFLⓡ que nous présentons ici.

Un composant logiciel peut être modélisé par les classes d'équivalence de ses chemins d'exécution. Chaque classe d'équivalence est appelée *chemin fonctionnel du composant.* A partir d'une spécification ou d'un code source, l'outil AGFLⓡ fournit les chemins fonctionnels ainsi que les jeux de tests permettant de valider ce composant.

Une des méthodes d'analyse de sûreté fréquemment employée est *l'AMDE du logiciel.* Cette analyse permet de vérifier la robustesse d'un logiciel vis-à-vis des défaillances internes ou externes du système et de déterminer les composants d'importance majeure pour la sûreté. Elle nécessite la modélisation fonctionnelle du logiciel pour la propagation des modes de défaillance. SURLOG S.A. a pris le parti de modèler le logiciel sous la forme de modèles des flux entre composants et des chemins fonctionnels de chaque composant.

Dans ce cadre, la méthode AGFLⓡ permet d'assister l'analyste devant réaliser :

- une analyse de sûreté de type AMDE du logiciel, en effectuant la première étape de cette analyse, à savoir *le modèle fonctionnel*

1. Analyse par Graphe Fonctionnel du Logiciel
2. Agence Nationale de Valorisation et d'Aide pour la Recherche
3. Sémantique, Preuves et Implémentation
4. Institut National en Recherche Informatique et Automatique ; `http://pauillac.inria.fr`
5. Analyse des Modes de Défaillance et de leurs Effets

- la Lecture Critique de Code, en vérifiant certains critères du sûreté du logiciel et en déterminant les chemins fonctionnels du logiciel
- les tests d'un logiciel, en déterminant l'ensemble des jeux de tests unitaires et d'intégration pour une couverture de tests du logiciel par classes d'équivalence.

Aujourd'hui, la méthode AGFL© est utilisée par SURLOG S.A. et ses partenaires industriels. De plus, elle est reconnue par plusieurs autorités de sûreté (IPSN, INRETS, ...).

3. Critères de choix des langages OCaml et Coq

Les critères de choix d'un langage de spécification ou de programmation portent sur trois aspects :

1. l'adéquation entre la spécification du logiciel à développer et le pouvoir d'expression du langage retenu
2. l'adéquation entre les temps de développement et d'exécution requis et la performance du langage (temps de compilation, d'exécution et de mise au point)
3. l'adéquation entre les contraintes du développement industriel (maintenance, réutilisation, ...) et la pérennité du langage.

Comme nous l'avons déjà présenté dans l'introduction, AGFL© est un outil de manipulation symbolique de programmes (écrits en C, Ada, ...). Il manipule différentes syntaxes concrètes, différentes syntaxes abstraites, à l'aide d'un ensemble de règles d'inférence écrites en sémantique naturelle. Il réalise aussi de nombreuses simplifications sur les termes représentant les programmes analysés. Notre choix de langages pour la réalisation d'AGFL© a donc été étayé par les critères suivants.

Pour la partie programmation, nous avons retenu le langage OCaml [5] car :

- il permet de coder directement une spécification exprimée sous la forme de règles d'inférence. En effet, il suffit d'implémenter chaque règle d'inférence par un filtrage sur la structure du terme à analyser. Cela nous a permis de réaliser très rapidement un prototype grandeur réelle d'AGFL© . Nous avons pu, en même temps, utiliser AGFL© pour différentes analyses, tout en faisant évoluer graduellement vers des versions de plus en plus optimisées. Cette conjonction utilisation/évolution, associée à des tests de non-régression, est importante d'un point de vue économique.
- il permet de manipuler des structures dynamiques et récursives sans utilisation de pointeurs : pas d'allocation et de libération de mémoire explicite grâce au *garbage collector*. Ceci nous a permis de nous concentrer sur les aspects importants des algorithmes sans avoir à nous préoccuper des aspects bas niveau de la programmation.
- il permet de détecter les "anomalies" de codage au plus tôt, dès la compilation grâce à son puissant mécanisme d'inférence de types. Cette garantie de cohérence interne du programme, fournie par le typeur, nous donne aussi une certaine assurance sur la qualité du produit logiciel.
- les performances du code généré aussi bien en *bytecode* qu'en natif se révèlent très satisfaisantes.

Pour la partie spécification, nous avons retenu l'assistant d'aide à la preuve Coq [8] car :

- il permet de décrire des structures récursives très compliquées, ce que n'autorisent pas les ateliers commercialisés d'aide à la preuve. En effet, la richesse du langage de l'outil Coq permet de spécifier directement la grammaire et la sémantique d'un programme et de vérifier des propriétés sur ces programmes.
- il permet de spécifier des règles d'inférence "à la Prolog", ce qui correspondait à une culture maîtrisée au sein de l'entreprise. Cela a facilité sa prise en main.
- il permet de refaire automatiquement la preuves des propriétés lors de l'évolution du modèle formel, car la preuve elle-même est un objet de "première classe" en Coq. Ainsi, après avoir

étendu les structures, on peut relancer la preuve et ne reprendre à la main que les preuves qui échouent à cause des modifications.
- il permet l'extraction d'un programme OCaml pour la construction d'un objet vérifiant certaines propriétés à partir de la preuve de son existence. Cependant, lors de cette extraction, nous nous sommes heurtés aux limitations de la construction `Fixpoint` pour modéliser des structures récursives (nécessité d'exhiber une relation d'ordre sur nos structures afin de démontrer implicitement la terminaison).

Les critères que nous avons retenus pour couvrir l'aspect industriel sont :

- la portabilité directe du *bytecode* sur plusieurs plates-formes (SUN, Linux, ...)
- la possibilité d'accès à un support technique *via* une liste de diffusion de messages électroniques [6] pour la résolution des différents problèmes sur le langage ou les outils associés
- la possibilité de recruter des ingénieurs formés à la fois à la syntaxe et à la sémantique du langage OCaml
- l'accès aux sources des logiciels OCaml et Coq nous garantissant la pérennité des outils
- la gratuité du langage et des outils associés ainsi que l'apport permanent de nouveaux utilitaires OCaml et Coq par la communauté les utilisant.

Ces différents critères nous ont donc conduits, en 1994, à choisir les langages Caml et Coq. Nous avons eu à accepter les côtés difficiles de ce choix, à savoir :

- la faible diffusion de ces langages dans le milieu industriel
- le manque de méthodologie de développement industriel
- le peu d'outils facilitant la tâche du développeur (générateur de documentation, règles de codage)
- le peu d'outils facilitant la tâche du testeur (outils de tests).

Durant ces années, nous avons suivi (et bénéficié de) les évolutions du langage Caml-light vers le langage OCaml et les différentes versions de l'atelier Coq. Il est à noter que le passage d'AGFLⓡ de Caml 0.73 à Caml Special Light 1.07, puis à OCaml 2.01, nous a amené à reprendre la totalité des sources de l'outil. Cela s'est toujours fait avec une grande facilité et en peu de temps.

4. Description de la méthodologie de développement

4.1. Présentation du cycle de développement

La figure 1 montre le cycle de développement utilisé pour le développement de l'outil AGFLⓡ. Il est fondé sur le cycle de vie classique [1] complété de phases de vérification formelle et de prototypage qui permettent de valider les principes décrits dans la spécification.

Les sections suivantes décrivent les activités effectuées dans chacune des phases du cycle de développement.

4.2. Phase de spécification

L'outil AGFLⓡ est composé de trois grands sous-systèmes :

1. le noyau, effectuant les manipulations de structures de données (programmes, variables, expressions, ...),
2. la gestion de projet et de configuration, implémentant les fonctionnalités de traçabilité et de cohérence de l'outil,
3. l'interface homme-machine (IHM), avec la définition des fenêtres.

6. `caml-list@inria.fr`

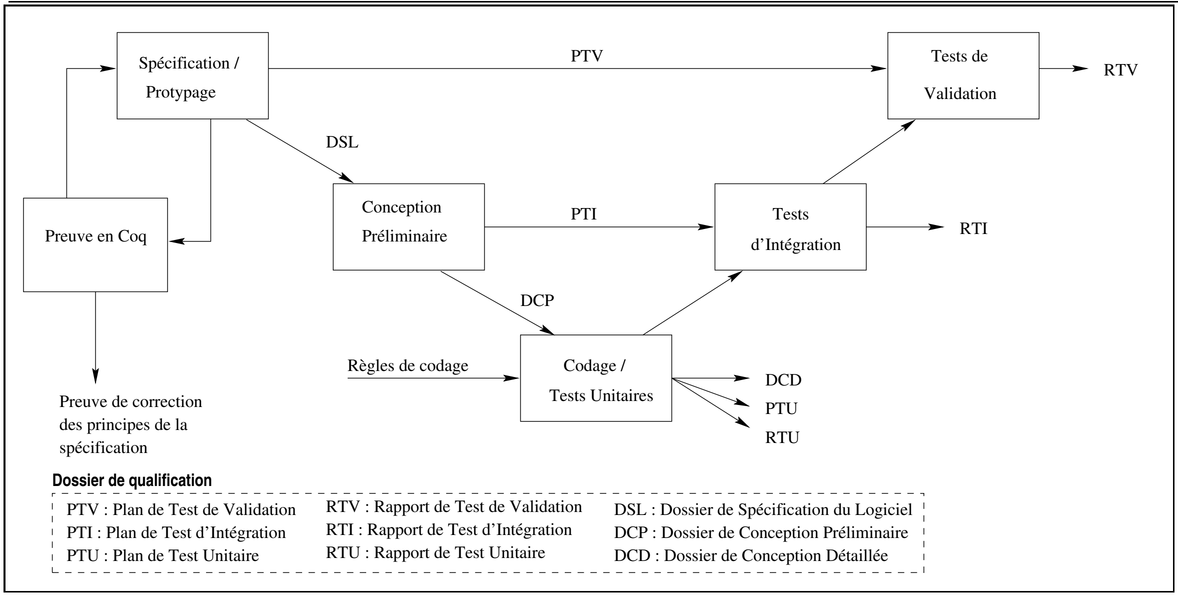

FIG. 1 – *Cycle de développement de l'outil AGFL*Ⓡ

Ces sous-systèmes étant d'importance différente pour la sûreté de l'outil AGFLⓇ, nous avons employé trois méthodes différentes pour leur spécification. Les paragraphes suivants présentent ces méthodes.

4.2.1. Le noyau

Pour le noyau, les transformations des données et autres manipulations ont été définies formellement par des règles d'inférence écrites en sémantique naturelle.

La définition des fonctions du noyau de manière formelle a permis d'obtenir une spécification claire et non ambiguë, ce qui est primordial pour le reste du cycle de développement. Un prototype du noyau a tout d'abord été développé afin d'effectuer une étude de faisabilité de l'outil et de détecter les incomplétudes de la spécification. Dans ce prototype, les règles de transformation des programmes étaient implémentées directement, c'est-à-dire en utilisant un filtre par règle. Le prototype a pu ensuite évoluer vers le développement réel, avec une représentation des données et des algorithmes optimisés.

Les règles ainsi définies ont été soumises à l'assistant d'aide à la preuve Coq [3], afin de s'assurer que les transformations appliquées aux programmes analysés sont déterministes, i.e. :

$$\forall p, p_1, p_2 : programmes, \left(p \xrightarrow{transf} p_1\right) \bigwedge \left(p \xrightarrow{transf} p_2\right) \Rightarrow (p_1 = p_2)$$

et correctes, i.e. :

$$\forall p, q : programmes, \forall e1, e2, e3 : etats \quad ([e1]\, p\, [e2]) \bigwedge \left(p \xrightarrow{transf} q\right) \bigwedge ([e1]\, q\, [e3]) \Rightarrow ([e2] \simeq [e3])$$

L'étape de preuve des principes du noyau est importante car elle permet de nous assurer que le développement du logiciel s'effectue avec une base correcte. Elle permet aussi de détecter, en amont du cycle de développement, les inconsistances de la spécification. En effet, plus on est avancé dans le cycle de développement, plus la modification de la spécification coûte cher (en coût et en délai). Enfin, la preuve des principes d'un logiciel participe à l'obtention d'un logiciel ayant un haut niveau de sûreté. Ce modèle formel doit être périodiquement mis à jour afin de garantir que l'ajout de nouvelles fonctionnalités (donc de nouvelles règles d'inférence) ne perturbe pas le système existant.

4.2.2. La gestion de projet et de configuration

Les fonctionnalités classiques de gestion de projet et de gestion de configuration ont été définies en vue de leur intégration dans AGFLⓡ. Leurs principes s'appuyent sur l'outil SCCS [7], avec lequel une interface OCaml a été définie. De plus, des fonctions de gestion de projet et de cohérence entre éléments gérés par l'outil ont été implémentées.

4.2.3. L'interface homme-machine

L'interface Homme-Machine a été réalisée à l'aide de la bibliothèque OCaml-Tk. Ceci nous a permis de réaliser la spécification de cette interface par une méthode de type "*try and test*". En effet, nous avons avec l'interface OCaml-Tk développé rapidement plusieurs prototypes d'IHM (sans aucun traitement applicatif) et choisi ensuite, avec les utilisateurs, celui qui leur convenait le mieux. Le dossier de spécification de l'interface a été réalisé à partir des captures d'écran du prototype d'IHM.

4.3. Phase de développement

Les phases de développement de l'outil AGFLⓡ ne sont pas différentes d'un projet classique. Elles comportent :

- une phase de conception préliminaire durant laquelle nous avons effectué la découpe en modules du logiciel. Elle aboutit à l'élaboration d'un Dossier de Conception Préliminaire contenant l'architecture du logiciel ainsi que la description des interfaces entre les différents modules.
- une phase de conception détaillée/codage durant laquelle nous avons défini et codé les algorithmes. Cette phase aboutit à l'élaboration d'un Dossier de Conception Détaillée. Ce document est au format HTML. Il est généré automatiquement à partir des commentaires du code grâce à l'outil **Gendal** (section 5.2).

Afin d'améliorer la lisibilité et la maintenabilité du code, nous avons défini des *règles de codage*. Elles définissent un style de programmation et sont applicables à toute personne réalisant des développements de logiciels en OCaml au sein de SURLOG S.A. Ces règles sont :

- les composants doivent être fonctionnels purs, c'est-à-dire qu'il est interdit d'utiliser la séquence, les références et l'affectation
- les paramètres des fonctions doivent être déclarés explicitement
- obligation de fournir une description fonctionnelle de tous les éléments définis dans le logiciel (fonction, variable globale, type, ...)
- l'obligation de préfixer chaque élément par le module dans lequel il est déclaré (nom complet).

4.4. Phase de tests

Les activités de tests consistent à vérifier, pour chaque étape de la phase de développement, la conformité du logiciel par rapport à sa documentation *au moyen de son exécution*. Les activités de tests comprennent :

1. l'élaboration du Plan de Test. Ce plan contient pour chaque niveau de tests (tests unitaires, tests d'intégration, tests de validation) :
   - l'organisation et les responsabilités de l'équipe de tests
   - les types de tests à réaliser sur le logiciel (tests nominaux, tests aux limites, ...)

  - la couverture de tests à atteindre pour chaque type de tests. Cette couverture définit les objectifs de tests pour chaque niveau de tests
  - la description de l'environnement de tests (outils de tests, banc de tests)
  - les fiches de tests
2. le développement des outils ou du banc de tests
3. l'exécution des tests
4. le dépouillement des résultats de tests et la rédaction du rapport de tests.

L'application de cette méthodologie à notre outil et ses résultats sont décrits en détail dans [3] et [4]. La définition de la couverture de tests structurels pour un langage impératif est clairement définie [2]. La difficulté a consisté à transcrire ces définitions de couverture pour un langage applicatif, en l'occurrence pour le langage OCaml, c'est à dire à répondre aux questions suivantes :

- quelles sont les vérifications effectuées par le compilateur OCaml permettant d'alléger la phase de tests ?
- comment tester une fonction partielle ?
- comment tester une fonction anonyme ?
- comment tester une fonction générique ?

4.4.1. Tests Unitaires

L'objectif des tests unitaires d'un composant est de vérifier la structure d'un composant et la conformité du composant par rapport à sa spécification. La couverture choisie est la couverture des chemins fonctionnels des composants. Nous avons réalisé ces tests à l'aide des outils **Ospat** et **Gentest** (section 5.3).

Types de tests Pour les tests unitaires, nous avons classiquement retenu comme type de tests :

- les tests nominaux (TN)
- les tests aux limites (TL) des domaines fonctionnels des données
- les tests hors limites (THL) des domaines fonctionnels des données.

Tests nominaux (TN) Les tests nominaux consistent à déterminer au moins un jeu de tests avec des données dans leur domaine fonctionnel pour chaque comportement du composant testé (chemin d'exécution). Lorsque le composant testé effectue des appels de fonctions, il est nécessaire de définir le comportement des fonctions appelées à l'aide de *bouchons*. Ces bouchons positionnent les sorties attendues des composants appelés, de façon à pouvoir accéder à chaque chemin fonctionnel du composant testé.

Tests aux limites (TL) Les tests aux limites des domaines fonctionnels des données portent surtout sur les types de base (entier, chaîne, ...) pour les instructions conditionnelles. Des tests aux limites ont aussi été effectués sur les structures complexes lorsqu'un ordre était défini sur ces structures dans la spécification de AGFLⓡ.

Tests hors limites (THL) Les tests hors limites des domaines fonctionnels des données ne sont pas applicables pour l'outil AGFLⓡ. En effet, il ne manipule que des structures et n'effectue pas de calcul. Il n'y a donc pas de risque de débordement. De plus, le langage OCaml interdisant les variables non initialisées, il est impossible de trouver des valeurs hors limites pour nos structures de données. Enfin, aucune contrainte n'est imposée sur les domaines des types de base (entier, chaîne, ...) dans la spécification de l'outil.

Structure du langage OCAML	**Couverture de tests à réaliser**
`if C then I1 else I2`	1 test pour le cas C = true, 1 test pour le cas C = false, couverture de la condition (cf. figure 3)
`match d with F1 → I1 | F2 → I2 ... | Fn → In`	1 test par filtre
`try ... with exp1 → I1 | exp2 → I2 ... | expn → In`	1 test par exception du `try` (lorsque l'exception peut être levée)
`;`	Non utilisée dans la partie de l'outil testée
Appel d'un composant	Mise en place d'un bouchon simulant le composant appelé

FIG. 2 – *Tests par structure de contrôle*

Condition	**Couverture de tests à réaliser**
C1 ET C2 ET ... ET Cn	cas Vrai : tous les Cn à Vrai (1 test); cas Faux : faire circuler un Faux dans la condition (n tests); donc (n+1) tests au total
C1 OU C2 OU ... OU Cn	cas Vrai : faire circuler un Vrai dans la condition (n tests) et tous les Cn à VRAI (1 test); cas Faux : tous les Cn à Faux (1 test); donc (n+2) tests au total
NON C	cas Vrai : 1 test; cas Faux : 1 test
e1 oper_relationnel e2	cas Vrai : 1 test; cas Faux : 1 test

FIG. 3 – *Tests par type de condition*

Exigence de couverture Après la définition des types de tests retenus, il faut fournir des objectifs de couverture. Pour les tests nominaux, nous avons retenu de couvrir 100% des chemins fonctionnels du code.
Le code du noyau d'AGFL® est purement fonctionnel et ne comporte pas de séquence d'instructions, ce qui facilite la détermination de la couverture des chemins fonctionnels, en effet une branche d'une instruction conditionnelle ou un cas de filtrage est équivalent à un chemin fonctionnel.

La figure 2 montre la couverture pour chaque structure de contrôle du langage OCAML, tandis que la figure 3 montre les tests réalisés pour chaque type de condition.

Pour les tests aux limites, nous avons retenu une couverture de chaque borne du domaine fonctionnel (par exemple : min_int et max_int pour une donnée de type entier), ainsi qu'une couverture de chaque point remarquable du domaine fonctionnel (par exemple : si le domaine fonctionnel de la variable a est [0..100] et qu'il existe une condition $a > 5$, il faudra réaliser un test avec $a = 5$).

4.4.2. Tests d'Intégration

Les tests d'intégration doivent permettre de contrôler que :

- pour chaque sous-système, les exigences de sa spécification sont bien respectées lorsqu'on intégre ses composants,
- à chaque intégration de sous-sytèmes, les exigences des interfaces entre composants du système logiciel sont bien respectées.

Types de tests Comme pour les tests unitaires, nous avons réalisé des tests nominaux et des tests aux limites.

Exigence de couverture Pour la partie fonctionnelle des tests d'intégration, nous avons rejoué l'ensemble des jeux de tests unitaires en remplaçant les bouchons par les véritables composants appelés. Le langage OCaml vérifie lui-même certaines des interfaces entre composants à l'aide du typage fort. De plus, nous ne manipulons que des structures créées à partir de constructeurs. Ceci a donc permis d'alléger une bonne partie des tests d'interface entre composants. Il restait à vérifier certaines interfaces car le compilateur OCaml ne peut pas, par exemple, détecter l'inversion de deux paramètres de même type lors d'un appel de fonction.

4.4.3. Tests de Validation

Les tests de validation doivent permettre de vérifier que le logiciel est conforme aux exigences décrites dans la spécification.

Types de tests La spécification de l'outil AGFL® n'ayant que des exigences fonctionnelles, aucun test n'a donc porté sur les aspects performance, charge, ... de l'outil. Les tests de validation sont donc uniquement fonctionnels. Ils consistent à exécuter le logiciel dans des situations d'exploitation courantes. Les tests de robustesse de l'interface Homme-Machine et des interfaces externes n'ont pas encore été réalisés.

Exigence de couverture Les tests de validation effectués aujourd'hui concernent la partie *simplification des expressions* du noyau de l'outil AGFL® . La spécification de cette partie est donnée sous forme de règles d'inférence. Par exemple, une loi de De Morgan est décrite sous la forme :

$$\frac{\begin{array}{c}\neg e1 \xrightarrow{S} es1 \\ \neg e2 \xrightarrow{S} es2 \\ es1 \bigvee es2 \xrightarrow{S} es\end{array}}{\neg\,(e1 \bigwedge e2) \xrightarrow{S} es}$$

La question qui s'est posée était de savoir combien de tests sont nécessaires pour vérifier l'implémentation d'une règle d'inférence. Pour notre part, nous pensons qu'il faut réaliser 3 à 4 tests par règle :

1. le test où le jeu de test est directement le radical à tester : $\neg\,(e1 \bigwedge e2)$ (test à l'ocurrence racine du terme),
2. le test où le radical à tester est un sous-terme du jeu de test : $e1 \bigwedge \neg\,(e2 \bigwedge e3)$
3. les test où le radical à tester est créé au cours des réductions du jeu de test : $\neg\,(e1 \bigvee (e2 \bigwedge e3))$
4. le cas particulier du radical à tester qui se recrée lui-même : $\neg\,(e1 \bigwedge (e2 \bigwedge e3))$.

Remarque : La stratégie de réduction n'étant pas connue (elle dépend de l'implémentation), on ne peut garantir que les jeux de tests 3 et 4 soient toujours pertinents.

5. Utilisation des outils OCaml

Pour le développement de l'outil AGFL® , nous avons utilisé une grande variété d'outils écrits en OCaml. Une partie de ces outils est issue de différentes contributions[7]. Nous en avons développé

7. `http://pauillac.inria.fr/caml/hump.html`

d'autres afin de couvrir nos besoins induits par les exigences qualité de nos méthodes de développement.

5.1. Outils de la contribution OCaml

Nous voudrions mentionner notre utilisation importante des outils durant le développement de l'outil AGFLⓡ :

OCamlLex / OCamlYacc L'outil AGFLⓡ analyse des programmes écrits dans différents langages (Ada, C, Modula-2) et plusieurs fichiers de paramétrage. Nous avons utilisé les outils **OCamlLex** et **OCamlYacc** pour générer les analyseurs lexicaux et syntaxiques.

OCamlTk La bibliothèque **OCamlTk** est une interface entre le langage OCaml et la bibliothèque graphique Tk [6]. L'interface Homme-Machine de l'outil AGFLⓡ a été réalisée avec cette bibliothèque.

OcamlProf L'outil **OCamlProf** permet d'obtenir un suivi d'exécution d'un programme OCaml, notamment de connaître le nombre de passages dans un filtre, dans une branche d'instruction conditionnelle, ... Cet outil a été utilisé lors de la phase de tests pour vérifier que les objectifs de couverture de tests définis avaient été atteints.

5.2. Outils de développement

OCaml est un langage récent et encore peu répandu en milieu industriel. Par conséquent, il existe peu d'outils de développement. Nous avons donc développé nos propres outils pour répondre à nos besoins :

Gestion de configuration : OCamlSCCS L'outil SCCS de SUN permet d'effectuer de la gestion de version sur des fichiers. L'outil AGFLⓡ réalise sa propre gestion de projet, de cohérence et de configuration dans un but de traçabilité des études réalisées. Nous avons choisi de nous appuyer sur l'outil SCCS pour réaliser les fonctionnalités de gestion de configuration de l'outil. Une interface **OCaml-SCCS** a donc été développée, qui permet d'exécuter directement toutes les commandes SCCS depuis un programme OCaml.

Génération de la documentation : Gendal Les règles de codage décrites à la section 4.3 ont permis la mise en place d'un générateur automatique de documentation à partir des commentaires du code. L'outil **Gendal** permet notamment de générer :

- la partie du Dossier de Conception Préliminaire (DCP) au format HTML, contenant la description de chaque module, chaque fonction, chaque variable globale, chaque type, ..., avec des liens dynamiques permettant de naviguer dans la documentation
- pour chaque fonction, une représentation graphique des différentes chaînes d'appels de fonctions durant son exécution (graphe d'appel)
- pour chaque fonction, un fichier de description utilisé lors de la phase de tests, spécifiant les paramètres de la fonction, son type, les variables globales utilisées, les fonctions appelées et les modules nécessaires à la compilation.

Gestion des modifications : Admin_fd Lorsqu'une anomalie est détectée ou qu'une amélioration doit être apportée à l'outil AGFL®, une fiche de demande est émise. Le développeur répond ensuite à cette demande, et sa réponse doit finalement être validée.
Pour faciliter ce processus de gestion des modifications et pour permettre une traçabilité aisée, l'outil **Admin_fd** gère les demandes de modifications de l'outil dans une base de données. Il permet ainsi de stocker, pour chaque demande, la version de l'outil concernée, la description de la demande, la priorité de la demande, ainsi que les versions des documents et modules qui seront modifiés.

L'outil **Admin_fd** utilise un prototype d'interface **OCaml-ODBC** [8] que nous développons, ainsi que la bibliothèque **OCamlGtk** [9], encore en développement, qui interface OCaml et la bibliothèque graphique Gtk [10].

5.3. Outils de tests

Afin d'automatiser les tâches fastidieuses du processus de tests (comme la saisie des jeux de tests, la création de bouchons, la création du moteur de tests, ...), nous avons développé des outils spécifiques :

Pour les Tests Unitaires et d'Intégration L'outil **Ospat** permet, à partir de la description d'une fonction comme celle fournie par notre outil **Gendal**, de saisir les jeux de tests unitaires et d'intégration, en fournissant au testeur les informations utiles aux tests.
Les jeux de tests saisis sont ensuite utilisés par l'outil **Gentest**, qui crée le moteur de tests, les bouchons, et les exécutables de tests. Après exécution des tests, l'outil **RapportTest** génère, à partir des résultats de tests, les rapports de tests au format HTML (d'autres formats sont prévus).

Pour les Tests de Validation L'outil **ValTest** permet d'effectuer les tests de validation et de constituer une base de tests, qui sert à effectuer des tests de non-régression à chaque nouvelle version de l'outil AGFL®.

Pour la gestion des tests Nous développons actuellement l'outil **Admin_test**, qui permettra de gérer en configuration l'ensemble des tests, en assurant la cohérence entre la version d'un test et la version du source associé. L'outil s'interfacera avec nos autres outils afin d'obtenir un véritable environnement de tests.

6. Conclusion

Au terme du développement de l'outil AGFL®, nous constatons que l'utilisation de langages applicatifs pour le développement d'outils dans le milieu industriel est viable et rentable. En effet, nous avons réalisé un outil ayant un haut niveau de qualité tout en respectant nos contraintes de coûts et de délais. Le tableau 4 présente des mesures sur le code développé.

Grâce au système d'aide à la preuve Coq, nous avons pu réaliser notamment une preuve de correction des principes de AGFL®, ce qui nous a permis d'affirmer la qualité et la sûreté de l'outil, point très important pour notre société.

Nous devons cependant mentionner que, malgré l'accroissement récent du nombre d'outils autour du langage OCaml, nous constatons encore un manque d'outils utilisables directement pour les développements industriels (accès aux bases de données, environnements de développement et de tests intégrés, ...).

8. Open DataBase Connectivity, interface standard d'accès aux bases de données relationnelles
9. http://www.cristal.inria.fr/~cuoq/mlgtk.html
10. http://www.gtk.org

Sous-système	Nombre de lignes de code	Nombre de fonctions
Noyau (analyseurs)	17000	150
Noyau (simplificateurs)	14000	240
Gestion de projet et de cohérence	3000	25
Interface Homme Machine	15000	300

FIG. 4 – *Taille des sous-systèmes de l'outil AGFL®*

7. Perspectives

La collaboration mise en place entre SURLOG S.A. et le thème SPI du LIP6 a principalement porté sur la correction des fondements théoriques de AGFL® et sur la spécification des outils d'analyse et de test. Elle a a mené en particulier à la définition d'un sujet de thèse, financé par une convention CIFRE. Il reste plusieurs questions ouvertes, notamment :

- la validité de la couverture de tests de validation pour un système de transformations et de simplifications
- l'extension des couvertures de tests unitaires et d'intégration à la totalité du langage OCaml.

SURLOG S.A. compte poursuivre ses développements internes d'outils d'analyse de sûreté du logiciel avec le langage OCaml. Un des projets est d'effectuer une refonte de notre méthode outillée RCSPL® [11] de réalisation automatisée d'AMDE et de combinaisons de pannes du logiciel.
Parallèlement, nous enrichissons nos propres outils de développement pour nos logiciels développés en OCaml. Enfin, nous réutiliserons l'outil Coq pour maintenir notre modèle formel, lors de l'extension des fonctionnalités de l'outil AGFL® .

Références

[1] GAMT17(V2), Méthodologie de développement des logiciels intégrés (dans les systèmes militaires), Ministère de la défense, 1989.

[2] M. MYERS The art of software testing, , 1974.

[3] Maxence GUESDON, Preuve et test d'un outil d'analyse statique de code en vue de sa certification, Rapport de DEA Sémantique Preuve et Langages, Septembre 1999.

[4] Othman HADDAD & Christophe LE THIERRY & Franck SADMI, Test du logiciel, Rapport de stage première année MST Informatique, Septembre 1999.

[5] Xavier LEROY, The Objective Caml system, INRIA, 1998.

[6] François PESSAUX & François ROUAIX, OcamlTk reference manual, INRIA, 1998.

[7] Don BOLINGER & Tan BRONSON, Applying RCS and SCCS, O'Reilly & Associates, 1995.

[8] PROJET COQ, The Coq Proof Assistant Reference Manual, 1999.

11. Recherche des Combinaisons Significatives des Pannes du Logiciel