LIF

Laboratoire d'Informatique Fondamentale
de Marseille

Unité Mixte de Recherche 6166
CNRS – Université de Provence – Université de la Méditerranée

Notes sur la conception d'une interface pour les Critères Communs de Sécurité

Line Jakubiec-Jamet

Rapport/Report 30-2006

Mars 2006

Notes sur la conception d'une interface pour les Critères Communs de Sécurité

Line Jakubiec-Jamet

LIF – Laboratoire d'Informatique Fondamentale de Marseille
UMR 6166
CNRS – Université de Provence – Université de la Méditerranée

Parc Scientifique et Technologique de Luminy
Case 901
163, Avenue de Luminy
13288 Marseille Cédex 9, France

Line.Jakubiec@lif.univ-mrs.fr

Abstract/Résumé

In this note, we propose a description of a framework dedicated to the validation of programs. This framework is inspired of the Common Criteria security certification model. We focus on the methods for testing programs which rely on that used for the natural language processing as for example, syntactic analysis, construction of BNF grammars, visualisation and covering of trees or graphs.
Keywords: Common Criteria security certification, methods for testing programs, syntactic analysis, BNF grammars.

Dans cette note, nous proposons une description d'un outil de validation de programmes inspiré du modèle des Critères Communs de Sécurité. Nous développons plus particulièrement les méthodes de test de programmes ayant un lien avec celles utilisées pour le traitement des langues naturelles, comme par exemple, l'analyse syntaxique, la construction de grammaires, la visualisation et le parcours d'arbres ou de graphes.
Mots-clés : Critères Communs de Sécurité, méthodes de test de programmes, analyse syntaxique, grammaires BNF.

Relecteurs/Reviewers: Robert Pasero and Jean-Luc Massat.

Notes: Ces notes font suite à la proposition d'un projet en Master Professionnel 2ème année (option Intégration des Systèmes Logiciels). Ce projet s'intitule *Une interface Web pour les Critères Communs de Sécurité*.

1 Introduction et rappels du projet

Les *Critères Communs de Sécurité* ont pour objectif d'évaluer la sécurité des systèmes informatiques [7] [6]. Ils mettent en place un standard de certification internationale (norme reconnue par plusieurs pays européens, le Canada et les Etats-Unis) et offrent un ensemble de recommandations précises pour analyser, développer, valider, maintenir et utiliser des systèmes informatiques. En ce sens, ils proposent une méthode de support pour les développeurs, les évaluateurs et les utilisateurs finaux. Leur cadre d'application est large puisqu'ils peuvent servir de référence pour tous les systèmes informatiques : systèmes d'exploitation, réseaux et leurs équipements (firewall, système d'accès, ...), applications. Ils traduisent en particulier des exigences de sécurité et permettent de poser des labels de qualité garantissant la fiabilité, l'intégrité et la robustesse des systèmes considérés.

Dans le cadre de la fiabilité, un des concepts clés du modèle des *Critères Communs de Sécurité* repose sur différents niveaux d'analyse des systèmes (ces niveaux correspondent à différentes couches de certification). Ils servent de base pour évaluer et déterminer le "degré de confiance" que l'on peut attribuer à un système en général et, dans notre cas d'étude, à une application [1]. En particulier, les *Critères Communs de Sécurité* décrivent 7 niveaux d'évaluation et d'analyse de programmes (*Evaluation and Analysis Level* notés EAL). Ces niveaux sont numérotés de 1 à 7 et spécifient les méthodes de vérification à appliquer selon le degré de sécurité (et de fiabilité) que l'on souhaite obtenir pour l'application à certifier. Plus l'application est dite critique, plus le niveau doit être élevé. Globalement, du niveau EAL1 au niveau EAL4, les applications sont dites de bonne qualité ; elles ont été testées intensivement, avec des méthodes de tests appropriées. Du niveau EAL5 au niveau EAL6, les applications sont dites de très bonne qualité avec une analyse et une vérification liée à la sécurité particulièrement poussée ; la notion de modélisation formelle apparaît à ces niveaux, ce qui a pour effet d'acccroître le degré de confiance qui va être attribuée à l'application. Le niveau EAL7 certifie les applications d'excellente qualité et pour lesquelles la stratégie de sécurité est fondamentale : l'ensemble de l'application (ou ses parties très sensibles) sont vérifiées formellement avec l'aide d'un outil formel.

Le tableau ci-dessous énumère les différents niveaux de certification et propose une méthode d'analyse et de vérification pour la réalisation de chaque EAL que nous allons étudier :

[1] Dans le cadre du projet, les applications à évaluer auxquelles nous nous sommes intéressées sont des programmes relativement simples. Nous les décrivons en section 9.

Numéro du niveau	Définition du niveau	Méthode proposée
EAL1	Test fonctionnel	Test des extrêmes et des anomalies
EAL2	Test structurel	Test d'intégration avec regroupement de modules ou fonctions
EAL3	Test et vérification méthodiques	Test par rapport à une *checklist* ou à une matrice de validation
EAL4	Conception, test et vérification méthodiques	Test par construction d'une base de chemins
EAL5	Conception semi-formelle et test	Test basé sur la syntaxe des entrées
EAL6	Conception formelle, vérification semi-formelle et test	Branchement sur un outil de preuve
EAL7	Conception vérifiée formellement et test	Branchement sur un outil de preuve

En ce qui concerne le projet proposé aux étudiants du Master, nous nous sommes intéressés à la construction d'un environnement inspiré de ce modèle. Le but est de développer une application destinée à tester et à vérifier le comportement des programmes au travers d'une interface conviviale. L'application aura une structure basée sur ces 7 niveaux. L'utilisateur final souhaitant vérifier un programme, se connectera au serveur sur lequel l'application sera installée. Puis, il choisira l'un de ces 7 niveaux (reliés alors à une méthode de vérification mise en oeuvre dans l'application) et il soumettra son programme à l'application. Notons que les étapes du projet sont les suivantes:

1. Elaboration du cahier des charges avec étude détaillée de la méthode à implémenter pour chaque niveau
2. Analyse de l'application avec Merise (création d'une base de donnée qui permettra la gestion des programmes, des jeux de tests, des niveaux et des méthodes employées)
3. Analyse fonctionnelle de l'application avec des diagrammes UML
4. Programmation de l'application et de son interface graphique (avec le serveur et les postes clients)
5. Production des jeux de tests (programmes ou échantillons de programmes utiles pour tester l'application à développer)
6. Tests de l'application avec maintien d'un rapport d'incident de tests
7. Rédaction d'une documentation d'exploitation et d'utilisation.

Dans ce rapport, nous nous intéressons aux 5 premiers niveaux, de EAL1 à EAL5, que nous mettons en pratique sur des méthodes de tests connues [11] [10]. Pour les niveaux EAL6 et EAL7, nous envisageons seulement de lancer l'assistant de preuve Coq [9] qui permettra alors à l'utilisateur de mener ses propres développements.

Cette note est organisée comme suit : les 7 sections suivantes (section 2 à section 8) correspondent aux 7 niveaux de certification des *Critères Communs de Sécurité*. Chacune de ces sections expose différents points de vue qui permettront de mettre en place une méthode de test de programmes. La section 9 donne des échantillons de programmes que l'application devrait pouvoir tester. Dans la dernière section, nous concluons sur ce projet.

2 EAL1 : Test fonctionnel

Le niveau EAL1 assure que le bon fonctionnement du programme à valider est évalué. Ce niveau est applicable lorsque la sécurité n'est pas considérée comme un élément central du programme. En général, le programme est testé par une approche en boîte noire et il est possible que les tests soient réalisés par le développeur lui-même.

Pour mettre en oeuvre ce niveau de certification dans notre application, nous proposons d'implémenter une méthode de test basée sur la recherche d'extrêmes et d'anomalies possibles d'un programme donné. Pour pouvoir réaliser ce test, l'utilisateur de l'application doit avoir une bonne connaissance du comportement du programme qu'il veut tester. Cependant, il n'a pas besoin d'en connaître les détails de l'implémentation (approche en boîte noire).

Le test d'extrêmes et d'anomalies est un des moyens les plus productif pour trouver des erreurs. En effet, il est intéressant de chercher des valeurs limites d'une exécution de programme pour connaître précisemment son comportement lorsqu'il est soumis à des cas extrêmes et anormaux (par exemple si on manipule des listes d'éléments, il est assez naturel de se poser les questions : Que se passe-t-il si la liste est vide? Que se passe-t-il si on a un très grand nombre d'éléments? Que se passe-t-il si on manipule le premier élément? Que se passe-t-il si la liste est de longueur l et qu'on accède au $(l+1)^{ieme}$ élément? ...).

D'une part, l'interface que nous nous proposons de développer devra offrir à l'utilisateur un moyen de saisir des cas limites, de les stocker dans un fichier et de les appliquer de façon automatique au programme. D'autre part, elle devra permettre de saisir les résultats attendus sur les cas limites à soumettre au programme. L'application mettra alors en évidence le fait qu'un résultat obtenu soit différent d'un résultat attendu. De plus, elle devra produire un tableau récapitulatif des tests réalisés dans une version imprimable.

3 EAL2 : Test structurel

Le niveau EAL2 s'applique aux programmes qui nécessitent plus d'efforts de développement et de mise au point. En particulier, dès que le programme est divisé en plusieurs unités (modules, appels de fonctions, bibliothèques, ...), il peut s'avérer utile de choisir ce niveau. Dans ce cas, le travail de test est réalisé en boîte noire par un évaluateur extérieur au programme (chef de projet, groupe de test, société d'audit par exemple) mais il nécessite la collaboration du développeur (pour élaborer des jeux de tests et fournir d'éventuelles informations complémentaires).

Le test structurel, appelé aussi test d'intégration avec regroupement de modules, permet à l'utilisateur de s'assurer que l'ensemble de son application fonctionne correctement et que l'interaction des unités n'engendre pas d'erreurs supplémentaires. Ainsi, ce niveau de test englobe deux aspects différents :

1. le test de fonctionnement qui consiste à assembler les programmes individuels et à vérifier que l'ensemble se comporte bien. Nous pouvons envisager ici de faire des *tests de charge et de performance* afin de fournir une prédiction du comportement de l'application à tester dans des cas réels d'utilisation (on va soumettre une charge maximale de travail à l'application en donnant un grand nombre de données d'entrée générées de façon aléatoire). Nous pouvons également prévoir de vérifier comment l'application est capable de récupérer les erreurs (dans ce cas précis on parlera alors de *test de stress*).
2. le test d'intégration qui consiste à tester plus précisemment la communication entre les programmes individuels. Nous nous intéressons ici aux programmes faisant appel les uns aux autres et il s'agit de mettre l'accent sur la réutilisation des résultats obtenus sur un programme p par un programme p' (utilisant donc les résultats de p). Ce type de test peut faire appel à des notions de plannification c'est-à-dire que l'on va s'intéresser aussi à l'ordre dans lequel les fonctions vont être assemblées et testées. Des scripts de tests pourront être développés à cet effet.

Dans le cadre du projet, il s'agit de mettre au point une méthode de test basée sur la structure des programmes à vérifier. L'interface à développer devra présenter à l'utilisateur un graphe faisant état de l'appel des fonctions les unes par rapport aux autres. Elle devra ensuite suggérer un ordre de test des fonctions pour lequel il faudra établir des jeux de tests adaptés (éventuellement basés sur la partie de la section 2). Puis, afin de pouvoir tester un programme dans son ensemble, il faudra s'assurer que, de façon globale, les résultats obtenus correspondent bien aux résultats attendus et que les erreurs sont répertoriées et gérées de façon satisfaisante.

4 EAL3 : Test et vérification méthodiques

Le niveau EAL3 permet d'obtenir une analyse plus poussée du programme. Le travail de test est réalisé en boîte grise, c'est-à-dire qu'il n'est pas imposé à l'évaluateur extérieur, de procéder à un contrôle de toutes les intructions du programme, mais ceci peut s'avérer nécessaire dans certains cas (en particulier pour renseigner un rapport d'incident de tests dont un exemple est donné en annexe A). En outre, l'application de ce niveau exige que le degré de couverture des tests par rapport au cahier des charges soit évalué (par exemple, 90% des spécifications relatives aux droits d'accès d'une application ont été testées avec succès). Il est à remarquer que ce niveau recommande la production d'une documentation conséquente et de qualité.

Pour ce niveau, nous nous proposons de mettre en place une méthode permettant de tester la complétude de jeux de tests (pré-établis, réalisés, exécutés avec succès) par rapport aux spécifications énoncées dans le cahier des charges (on vérifie que toutes les fonctionnalités prévues dans l'application soient implémentées correctement). En général, ce type de test est réalisé par rapport à une *checklist* ou à une matrice de validation suivant les étapes suivantes :

1. Dresser la liste de toutes les fonctionnalités énoncées dans le cahier des charges (en langage naturel)

2. Traduire chaque fonctionnalité dans un formalisme *ad hoc* (graphique mettant en évidence les entrées et les sorties par exemple)
3. Rechercher la fonctionnalité dans l'application à tester
4. Produire les jeux de test et les résultats attendus pour chaque fonctionnalité (ces données pourront être stockées dans des fichiers et des scripts de chargement de ces derniers pourront être développés)
5. Tester automatiquement chaque fonctionnalité et analyser le résultat obtenu

Il est quelquefois utile de reformuler les énoncés pour pouvoir passer la certification de ce niveau ; ceci permet souvent de mieux les comprendre et par là-même de mieux déceler les erreurs ou incomplétudes.
Pour le projet, nous envisageons de construire une interface qui facilitera la recherche dans le cahier des charges des différentes fonctionnalités de l'application à tester (recherche par mots-clés sur un document, mise en ligne de pages, techniques facilitant la lecture, etc...). Pour produire la documentation, nous souhaiterions programmer ou intégrer un outil convivial d'aide à la création et à la rédaction de documents écrits (avec correcteur orthographique et surlignage de mots clés par exemple). En ce qui concerne les questions de lisibilité et d'analyse des résultats des tests, nous pourrions également joindre dans les documents décrivant ces derniers (journal et rapport d'incident) des graphiques d'évaluation de l'état des tests. Ces graphiques pourraient être présentés sous la forme de tableaux, de courbes ou d'histogrammes par exemple.

5 EAL4 : Conception, test et vérification méthodiques

Le niveau EAL4 exige l'application de méthodes de tests rigoureuses et permet d'obtenir une assurance élevée de bon fonctionnement du programme. Un évaluateur extérieur va s'attacher à valider la conception détaillée (par exemple, il validera les choix établis, au cours de l'analyse, des différents scénari possibles) et à analyser le code source (les tests doivent être réalisés en boîte transparente). Le fait de disposer du code source écrit dans les règles de la programmation (variables nommées de façon cohérente et explicite, décomposition modulaire en différentes fonctions, programmes indentés, etc...) permettra à l'évaluateur de procéder à une analyse précise dans des délais convenables. Nous nous proposons ici de mettre en oeuvre la méthode de test par construction d'une base de chemins, car elle permet une analyse fine du programme à tester et offre à l'évaluateur un cadre précis pour développer son contrôle. Elle s'applique aux langages de programmation procéduraux ou à certaines parties des langages objets. C'est une technique systématique partant de la représentation du corps du programme sous la forme d'un graphe orienté, et permettant la construction d'un jeu minimal de tests. Remarquons que pour obtenir la représentation du corps du programme, il est nécessaire de procéder à une analyse syntaxique du code source. Il est à noter que les programmes récursifs nécessitent une transformation préalable avant de pouvoir être testés à l'aide de cette méthode (des techniques existent pour remplacer un programme récursif par sa version itérative, tout programme récursif possèdant une version équivalente non récursive [1]). Les parties suivantes décrivent les différentes étapes nécessaires à la mise en oeuvre de cette méthode.

5.1 Première étape : construction d'un graphe par analyse syntaxique du programme

Pour construire le graphe qui représentera le programme à tester et qui permettra de générer des jeux de tests, il est utile de faire une analyse syntaxique du programme en remplaçant chaque instruction de contrôle par le sous-graphe correspondant. L'analyse syntaxique pourra se faire avec une recherche de mots clés introduisant les instructions de contrôle. La figure suivante définit la correspondance entre les instructions de contrôle et les sous-graphes :

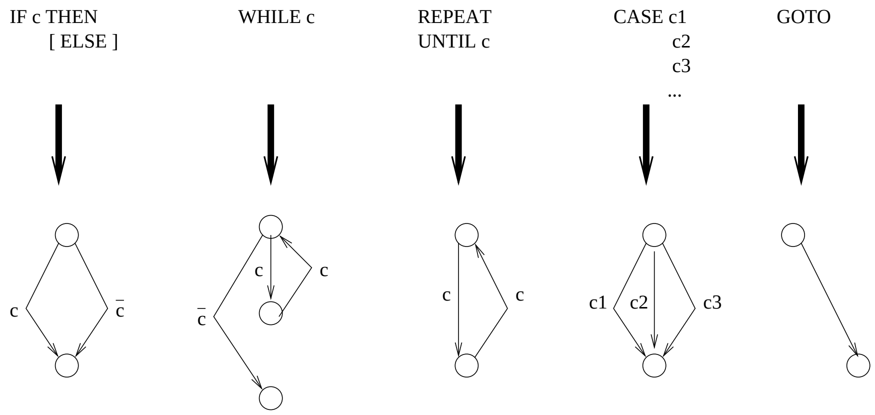

Figure 1: Représentation des instructions de contrôle

Une instruction simple (affectation, entrée-sortie, appel de fonctions, etc...) sera représentée par un noeud dans le graphe. Par suite, un chemin est défini par une suite d'instructions qui représentera une exécution possible du programme, la première instruction correspondant au premier noeud du graphe et la dernière au dernier noeud. Le graphe ainsi créé donne une représentation de l'ensemble des instructions du programme à tester. Par exemple, si on considère le programme *Read_and_deroute* suivant (avec f_1 et f_2 deux fonctions quelconques auxquelles le programme fait appel) :

PROGRAM *Read_and_deroute*;
BEGIN
  read(a,b);
  if $a > 0$ **then**
    $f_1(a)$;
  if $b > 0$ **then**
    $f_2(b)$;
END.

on obtient le graphe correspondant qui est une représentation du programme qu'il faudra ensuite utiliser pour construire les jeux de test :

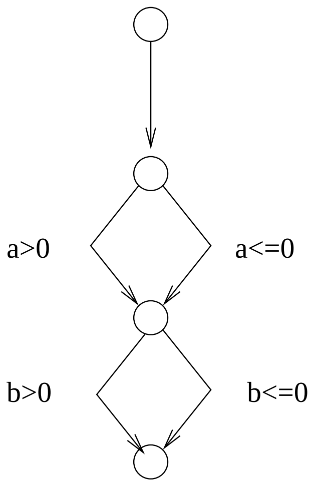

Figure 2: Graphe correspondant au programme *Read_and_deroute*

5.2 Deuxième étape : calcul de la complexité des tests

A partir du graphe obtenu, il faut calculer la complexité de chaque jeu de test à créer. En général, un jeu de test comporte un certain nombre de cas de tests qui permettront à l'utilisateur de vérifier son programme. Mais il est difficile de déterminer si le nombre de cas créé est suffisant. Cette méthode de test offre l'avantage de minimiser le nombre de cas tout en garantissant que toutes les instructions du programme pourront être exécutées au moins une fois sur chaque jeu de test créé. La complexité, notée C dans ce qui suit, représente le nombre de cas de tests à considérer pour chaque jeu de tests et elle peut se calculer des deux façons équivalentes suivantes :

$$C = nombre\ de\ régions + 1$$
$$C = (nombre\ d'arcs - nombre\ de\ noeuds) + 2$$

où une région est une surface close bordée par des chemins et un arc est une flèche reliant deux noeuds adjacents.

Sur l'exemple, on obtient $C = 2 + 1 = 3$ ou encore $C = (5 - 4) + 2 = 3$. Il y a donc 3 cas de tests à construire pour constituer un jeu de test dit *complet* et *couvrant*. Ainsi, l'exécution de ces 3 cas de tests sur le programme suffira pour obtenir une couverture complète du code du programme (c'est-à-dire que toutes les instructions du programme pourront être exécutées au moins une fois).
De façon générale, le calcul de la complexité C garantit que l'on peut obtenir une couverture complète et significative du code du programme en construisant au moins un jeu de tests avec C cas (le jeu de test à C cas est minimal).

5.3 Troisième étape : parcours du graphe et détermination des cas de tests

Le parcours du graphe et le calcul de la complexité C des tests va nous permettre de marquer C chemins dans le graphe. Ces chemins nous aideront à déterminer les C cas de tests pour chaque jeu de tests. La méthode générale pour construire C cas de tests est décrite par les étapes suivantes :

1. Choisir arbitrairement un cas définissant un chemin et marquer les flèches correspondantes.
2. Créer le cas suivant à partir du même chemin modifié de manière à parcourir au moins une flèche non marquée. Marquer celle-ci.
3. Continuer jusqu'à l'obtention de C chemins.
4. Etablir les valeurs à tester et définir les résultats attendus pour chaque cas.

Sur l'exemple traité, cela peut revenir à construire les 3 chemins de la figure 3 issus du graphe de la figure 2. Remarquons que dans ce cas simple, il suffirait de spécifier 2 chemins pour obtenir une couverture complète. Mais le jeu de test construit à partir de ces 2 cas ne permettrait pas d'obtenir une exécution suffisamment représentative du comportement du programme.

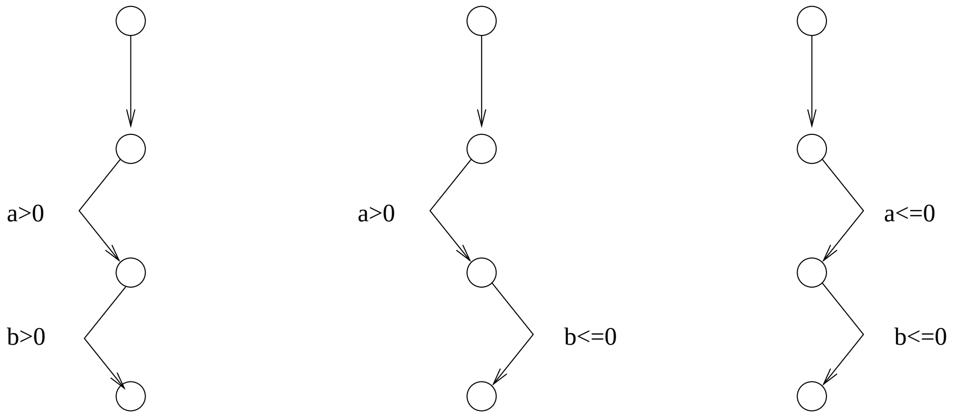

Figure 3: Construction des 3 chemins

D'après ce parcours du graphe de la figure 3, on peut par exemple, déterminer le jeu de tests suivant :

Tests à réaliser	Valeurs proposées	Résultats attendus
Deux nombres a et b positifs	$a = 3$ et $b = 4$	Appels de $f_1(3)$ et de $f_2(4)$
a positif et b négatif	$a = 3$ et $b = -245$	Appel de $f_1(3)$
Deux nombres a et b négatifs	$a = -20$ et $b = -20$	Rien

Il est alors possible de proposer d'autres jeux de tests (comportant chacun C cas de tests) en définissant soit d'autres valeurs sur le même parcours, soit d'autres valeurs sur des parcours différents.

6 EAL5 : Conception semi-formelle et test

La certification au niveau EAL5 s'appuie sur des tests très étendus doublés de représentations formelles de certaines parties du programme. Il requiert une étude approfondie du code source en mettant l'accent sur l'analyse de canaux cachés (par exemple recherche des effets de bord lors de l'éxécution d'un programme, rattrapage contrôlé des opérations anormales).

Pour ce niveau, nous souhaitons étudier la méthode de test de programmes basée sur la syntaxe des entrées. Largement utilisée dans le domaine de la théorie des langages et du traitement automatique des langues (pour les analyseurs, les compilateurs, les interpréteurs, les traducteurs...), cette méthode est intéressante car elle part d'une description formelle représentant les entrées possibles du programme. Cette méthode a également pour but de générer des jeux de tests significatifs qui serviront à s'assurer soit qu'un programme fonctionne convenablement sur des entrées correctes, soit qu'il génère des erreurs sur des entrées incorrectes. Pour mettre en oeuvre cette méthode, il faut d'abord établir une grammaire hors-contexte, formulée ici en BNF (Backus-Naur Form), qui décrira la syntaxe des entrées possibles du programme. Puis, il faut construire un graphe correspondant à cette grammaire. Le parcours de ce graphe permettra d'obtenir d'une part, un jeu de tests des cas valides (génération d'exemples) et, d'autre part, un jeu de tests des cas invalides (génération de contre-exemples). Par suite, il s'agit de vérifier que tous les cas valides sont acceptés par le programme et que tous les cas invalides sont rejetés avec, éventuellement, le message d'erreur adéquat. Plus précisemment, cette méthode de test suit les étapes suivantes :

1. Construction de la grammaire des commandes possibles pour le programme à tester (il faut en proposer une version formelle exprimée en BNF, à l'aide des symboles <>, ::= et |).
2. Définition du graphe correspondant à cette grammaire puis de la structure correspondante permettant de le représenter.
3. Spécification de cette grammaire à l'aide de cette structure.
4. Elaboration de l'algorithme qui décrit le parcours pour générer des cas valides.
5. Génération automatique d'un échantillon (qui sera suffisamment représentatif) des jeux de tests de cas valides à l'aide du parcours précédent.
6. Elaboration de l'algorithme qui décrit le parcours pour générer des cas invalides.
7. Génération automatique d'un échantillon (qui sera suffisamment représentatif) des jeux de tests de cas invalides à l'aide du parcours précédent.

6.1 Rappels sur les grammaires BNF

Le formalisme BNF (Backus Naur Form) a été introduit pour décrire la syntaxe des langages. Il offre une notation formelle composée des symboles suivants :

- ::= (qui signifie "se réécrit en"),
- | (qui signifie "ou"),
- <> (utilisé pour représenter les symboles non-terminaux).

Les règles d'une grammaire formulée en BNF sont des règles de réécriture.

Chaque règle représente la définition d'un non-terminal. Par exemple, un compilateur devra reconnaître des identificateurs. On considère la règle syntaxique suivante concernant des mini-identificateurs : un mini-identificateur commence toujours par une lettre qui peut être suivie par une autre lettre ou par un chiffre (des exemples de mini-identificateurs sont a, b, bd, $a1$, $z8$). Cette règle va pouvoir être spécifiée de la façon suivante dans une grammaire hors-contexte notée en BNF :

$< identificateur >::=< lettre > \mid < lettre >< lettre\_ou\_chiffre >$
$< lettre\_ou\_chiffre >::=< lettre > \mid < chiffre >$
$< lettre >::= a|b|c|...|y|z$
$< chiffre >::= 0|1|...|8|9$

Par suite, une expression sera correcte, si on peut la dériver, à partir du non-terminal $identificateur$, en appliquant les règles définies par la grammaire (on dit que $identificateur$ représente le symbole de départ de la grammaire). Elle sera incorrecte dans le cas contraire.

6.2 Représentation graphique des grammaires

A partir de la grammaire BNF établie, il faut définir un graphe orienté selon les règles suivantes :

1. Chaque sommet du graphe correspondra à un symbole non-terminal de la grammaire. Ce sommet est appelé *noeud non terminal*.
2. On appelle *noeud terminal* un sommet particulier du graphe qui ne pointe sur rien. Chaque noeud terminal correspondra à un symbole terminal de la grammaire (par abus de langage, un noeud terminal pourra être assimilé à une feuille).

Par exemple, pour les mini-identificateurs on obtiendra le graphe suivant :

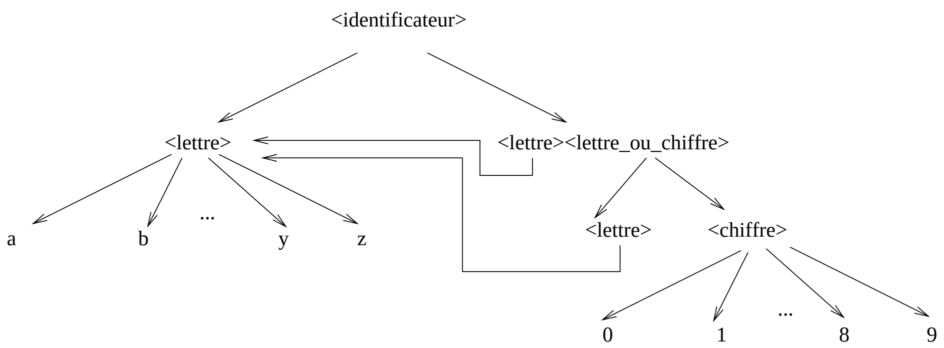

Figure 4: Graphe obtenu à partir de la grammaire des mini-identificateurs

Remarquons que, dans le graphe, un noeud non terminal qui pointe sur des noeuds terminaux pourra lui-même être pointé par d'autres noeuds portant le même nom (c'est le cas du noeud *lettre* qui est pointé par les autres noeuds *lettre* apparaissant dans la partie droite du graphe).

6.3 Parcours à mettre en oeuvre pour construire des jeux de tests de cas valides

Dans la suite, il s'agit de parcourir le graphe obtenu (un parcours correspondra à un chemin créé). Les parcours que l'on définit permettront la construction d'un jeu de tests des cas valides. En d'autres termes, il y aura reconnaissance des entrées valides du programme à tester, par la grammaire. L'algorithme à mettre en place aura les caractéristiques suivantes :

1. Construction de l'ensemble des parcours du graphe : à partir du symbole de départ, tous les chemins conduisant à un noeud non-terminal sont exercés (on dit qu'on a une couverture complète du graphe sauf pour les noeuds terminaux).
2. Fin d'un parcours : au moins un noeud terminal doit être atteint pour chaque chemin créé.
3. Fin de l'algorithme : la couverture complète du graphe est réalisée (sauf pour les noeuds terminaux).
4. Construction du jeu de test : chaque cas du jeu de test correspondra à une valeur portée par le noeud terminal (cette valeur devra alors être mémorisée pour construire le jeu de test).

Ainsi, tout parcours du graphe obtenu par l'algorithme décrit dans cette section permet de définir des cas de tests reconnus par la grammaire.

6.4 Construction de jeux de tests de cas valides

Sur le graphe de la figure 4, on peut construire différents parcours permettant une couverture complète des noeuds non-terminaux. Par exemple, le jeu de test comprenant les 3 cas valides : a, $y3$, xy peut être obtenu en parcourant ce graphe.

6.5 Parcours à mettre en oeuvre pour construire des jeux de tests de cas invalides

Un autre parcours du graphe va permettre la construction d'un jeu de tests des cas invalides. Pour déterminer ces cas, il est possible de transformer le graphe en arbre, de telle façon que toutes les flèches introduisant une connexion entre les noeuds pointés et les autres noeuds portant le même nom (cas du noeud *lettre* dans la section 6.2) soient supprimées. Sur l'exemple des mini-identificateurs, on obtient l'arbre de la figure 5. L'algorithme à mettre en place se décomposera de la façon suivante :

1. Numéroter les différents niveaux de l'arbre (0 pour la racine portant le symbole de départ de la grammaire, 1 pour les fils de la racine...).
2. Commencer avec le niveau $n = 1$ (fils directs de la racine).
3. Créer des tests introduisant une erreur au niveau n.
4. Itérer sur chaque niveau $n + 1$ jusqu'à ce que tout l'arbre ait été traversé.

Ainsi, tout parcours de l'arbre obtenu par l'algorithme décrit dans cette section permet de définir des cas de tests rejetés par la grammaire.

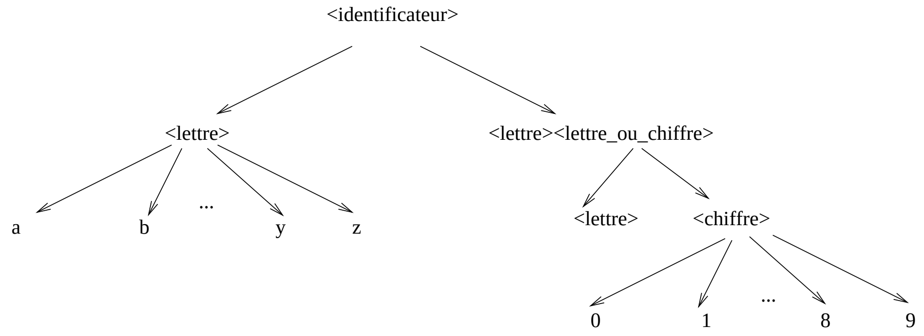

Figure 5: Arbre obtenu à partir de la grammaire des mini-identificateurs

6.6 Construction de jeux de tests de cas invalides

Sur l'arbre de la figure 5, il est possible de créer le jeu de test comprenant les 2 cas invalides suivants : 4 (pour le niveau 1), $e*$ (pour le niveau 2). Remarquons qu'en construisant un jeu de tests de cas invalides, on retrouve la démarche de la méthode de test qui consiste à rechercher des cas extrêmes et des anomalies d'un programme (voir section 2). Cette analogie existe dans le fait d'exhiber des cas significatifs d'erreurs. Cependant, dans la méthode de test basée sur la syntaxe des entrées, il y a une classification possible des extrêmes et anomalies, c'est-à-dire que l'on peut attribuer une classe d'erreur à chaque niveau de l'arbre sur lequel les erreurs sont introduites (sur l'exemple, 4 n'est pas un identificateur car la chaîne de caractères proposée commence par un chiffre et $e*$ contient un caractère non autorisé dans la grammaire). Sur des exemples plus conséquents, cette classification serait plus significative : erreurs typographiques, erreurs sur les arguments (en nombre, type ou nom), symbole non autorisé, etc...

7 EAL6 : Conception formelle, vérification semi-formelle et test

Le niveau EAL6 est essentiellement appliqué pour certifier les systèmes dédiés à la sécurité. En particulier, il exige des formalisations, dans un langage logique, de propriétés de programmes et de comportements ainsi que la vérification semi-formelle de ces propriétés [5] [2]. De plus, il impose une analyse rigoureuse des risques encourus. Pour notre projet, si le choix de l'utilisateur se tourne vers ce niveau, l'interface doit basculer sur l'assistant de preuves Coq [9].

8 EAL7 : Conception vérifiée formellement et test

A ce niveau d'assurance, la sécurité, la fiabilité et la robustesse des systèmes étant un enjeu central, toutes les fonctionnalités sont formalisées et prouvées correctes. De plus, la preuve de correction est donnée dans un outil formel [3] [8] [4]. Pour notre projet, si le choix de l'utilisateur se tourne vers ce niveau, l'interface doit basculer sur l'assistant de preuves Coq [9].

9 Echantillon de programmes à tester

Afin de pouvoir tester notre application, nous nous proposons d'implémenter des programmes qui permettront d'en constituer un jeu de test. Nous pouvons envisager de rechercher un exemple qui couvrirait tous les niveaux (programme qui pourrait être testé de EAL1 à EAL7 et qui mettrait en évidence l'apport de chaque niveau de certification) mais, dans un premier temps, nous nous attacherons à développer un programme approprié pour chaque niveau afin de permettre une étude de cas plus empirique.

Pour construire cet échantillon de programme, il est intéressant de se placer sous un angle sécurité. En effet, chaque programme qui constitue l'échantillon, peut être vu comme une partie d'une application plus importante dans laquelle il est nécessaire de vérifier, à un certain niveau, certains aspects de la programmation.

9.1 Pour le niveau EAL1

Pour tester la partie de l'application qui correspond au niveau EAL1, nous nous intéressons à un programme d'analyse de triangles. Le but de ce programme est de trouver la nature d'un triangle représenté par la longueur de ses trois côtés. Les entrées du programme correspondent à trois nombres réels. Ces nombres sont séparés par des virgules ou des espaces. Le traitement consiste à :

- déterminer si les trois nombres forment un triangle valide ; si ce n'est pas le cas, le programme doit afficher *n'est pas un triangle* ;
- classer le triangle selon la longueur de ses côtés dans l'une des catégories *isocèle* (deux côtés égaux), *équilatéral* (trois côtés égaux), *scalène* (tous les côtés inégaux) ;
- classer le triangle selon l'angle le plus grand parmi *aigu*, *droit*, *obtus*.

La sortie du programme devra comprendre les trois nombres donnés en entrée et la classification du triangle (ou le message *n'est pas un triangle*). Par exemple :

- (3,4,5) est un triangle scalène avec angle droit,
- (6,1,6) est un triangle isocèle avec angle aigu,
- (5,1,2) n'est pas un triangle,
- (4,4,4) est un triangle équilatéral avec angle aigu.

Le test de ce programme par notre application doit être vu comme un test unitaire simplifié pour les besoins de l'étude. Sur cet exemple, il est en effet aisé de trouver des cas limites ou anormaux : cas limites pour classifier des côtés (par exemple, le test d'un triangle presque isocèle permettra d'effectuer un contrôle sur la précision du typage des variables), cas limites pour classifier des angles, cas limites pour légitimer des triangles, formats invalides, etc...

9.2 Pour le niveau EAL2

Pour réaliser un test du niveau EAL2, nous nous proposons d'implémenter un programme décrivant une fonction qui calcule la factorielle en un certain nombre d'itérations. Cette description permet d'étudier la communication et l'assemblage des différentes unités composant le programme. Celui-ci est décrit ci-dessous, à l'aide de plusieurs fonctions où l'entier naturel t représente le nombre d'itérations. Pour chaque fonction constituante, nous donnons sa signature puis sa définition :

- La fonction Mux : $boolean * N * N \rightarrow N$

$$Mux(b, n, m) = \begin{cases} n & \text{si } b \\ m & \text{sinon.} \end{cases}$$

- La fonction $Zerotest$: $N \rightarrow boolean$

$$Zerotest(n) = \begin{cases} true & \text{si n} = 0 \\ false & \text{sinon.} \end{cases}$$

- La fonction J : $N * N \rightarrow N$

$$J(i, t) = \begin{cases} 0 & \text{si } t = 0 \\ Mux(Zerotest(J(i, t-1)), i, J(i, t-1) - 1) & \text{sinon.} \end{cases}$$

- La fonction R : $N * N \rightarrow N$

$$R(i, t) = \begin{cases} DontCare & \text{si } t = 0 \\ Mux(Zerotest(J(i, t-1)), 1, R(i, t-1) * J(i, t-1)) & \text{sinon.} \end{cases}$$

Notons que:
N représente l'ensemble des entiers naturels,
boolean représente l'ensemble des booléens *ie* l'ensemble $\{true, false\}$,
DontCare représente une variable dont la valeur est indéfinie.

Il est intéressant de soumettre un programme basé sur cette description à notre application de façon à :

- mettre en évidence les valeurs d'entrées pour lesquelles le calcul de la factorielle est significatif (test de fonctionnement),
- étudier la communication entre les différentes fonctions (test d'intégration).

9.3 Pour le niveau EAL3

Pour illustrer la certification au niveau EAL3, nous allons étudier quelques fonctionnalités d'un système d'accès à un serveur. En particulier, nous nous intéressons à un système de gestion de la table de mots de passe permettant d'accéder à ce serveur. Cette table contient, pour chaque personne concernée, son mot de passe. Les opérations devant être effectuées sont les suivantes :

- ajout d'un nouveau mot de passe (éventuellement ajout d'un nom uniquement, sans mot de passe, si la personne a fait une demande d'accès mais qu'elle n'a pas encore reçu d'autorisation),

- suppression d'une personne (et de son mot de passe) à sa demande ou sur décision de l'administrateur,
- recherche du mot de passe d'une personne si celle-ci ne s'en souvient plus,
- recherche des personnes dont le mot de passe ne contient pas au moins deux caractères spéciaux (pour des raisons de sécurité).

De plus, afin de pouvoir rendre compte au mieux des outils de gestion documentaire développés pour ce niveau, le plus grand soin devra être porté à la documentation produite tant sur la partie concernant l'analyse que sur celle, plus technique, concernant la programmation et les tests.

9.4 Pour le niveau EAL4

Pour tester la partie de l'application qui correspond au niveau EAL4, on pourra reprendre l'algorithme de fusion de deux tableaux triés, donné ici sous forme d'une procédure en langage Pascal :

```
procedure FUSION (t1,t2:tableau; n1,n2:integer; VAR t:tableau;VAR n:integer);
VAR i,j,k,l:integer;
BEGIN
    i:=0;
    j:=1;
    k:=1;
    WHILE (j ≤ n1) AND (k ≤ n2) DO
        IF t1[j] < t2[k] THEN
            BEGIN
                i := i + 1;
                t[i] := t1[j];
                j := j + 1;
            END
        ELSE
            BEGIN
                i := i + 1;
                t[i] := t2[k];
                k := k + 1
            END
    FOR l := j + 1 TO n1 DO
        BEGIN
            i := i + 1;
            t[i] := t1[l];
        END
    FOR l := k + 1 TO n2 DO
        BEGIN
            i := i + 1;
            t[i] := t2[l];
        END
    n := i;
END;
```

Le choix de cet exemple est intéressant pour le projet car le programme possède des structures de contrôle imbriquées et plusieurs boucles (les instructions **FOR** sont à remplacer par des instructions **WHILE** qui leur seront équivalentes). De plus, les problèmes liés au débordement de tableaux peuvent être évités après une analyse poussée de ce style de programmes.

9.5 Pour le niveau EAL5

Pour tester la partie de l'application qui correspond au niveau EAL5, il serait intéressant d'implémenter un évaluateur d'expressions arithmétiques. La grammaire BNF des expressions arithmétiques pourra être déterminée aisément. Les expressions arithmétiques sont des nombres entiers ou des expressions bien formées composées des opérateurs binaires $+$, $-$, $*$ et $/$. Par exemple, $52 + 3$ et $45 + 123 * 5$ sont des expressions bien formées. Le programme devra retourner sur ces exemples les résultats 55 et 660 (les règles de priorité des opérateurs devant être respectées).

9.6 Pour les niveaux EAL6 et EAL7

Le test de la partie de l'application qui correspond aux niveaux EAL6 et EAL7 consistera seulement à s'assurer que le système Coq est lancé, et ce pour tout programme soumis à l'application.

10 Conclusion

Initialement prévu pour rendre le concept des *Critères Communs de Sécurité* plus accessible aux étudiants, le sujet proposé a demandé une étude détaillée des méthodes choisies pour illustrer l'application de chaque niveau d'analyse. C'est cet aspect qui a été présenté ici. Bien évidemment, cette note ne rend pas compte de l'ensemble des caractéristiques décrites dans les *Critères Communs de Sécurité*. Elle donne seulement une interprétation simplifiée du concept.

Ce qui nous paraît intéressant par la suite, c'est le développement ou l'intégration de méthodes et d'outils permettant une application plus systématique des *Critères Communs de Sécurité*. En particulier, nous avons vu que les méthodes de tests de programmes pouvaient être basées sur des approches utilisées dans le cadre du traitement automatique des langues naturelles. De plus, l'aspect formel de certaines de ces méthodes est particulièrement important pour mettre en oeuvre des techniques de vérification efficaces. Nous envisageons donc de poursuivre l'étude de telles approches pour étayer ce point de vue.

A Exemple de rapport d'incident de tests

Un rapport d'incident de tests est un document permettant de consigner et d'analyser tout événement demandant une correction ou un nouveau contrôle. Il comporte une phase de description des erreurs, une phase de correction et une phase d'analyse. Ce qui suit donne un exemple de rapport :

Rempli par :
Date :
Nom de l'application :
Version de l'application :
Nom du programme ou de la fonctionnalité testée :
Version du programme ou de la fonctionnalité testée :
Numéro de test :
Numéro de rapport :

A.1 Phase de détection

A.1.1 Mode de détection du problème

- Sortie erronée détectée par une procédure automatique de test
- Sortie erronée détectée par un utilisateur
- Test à la main
- ...

A.1.2 Effort passé à identifier l'erreur

- Temps CPU
- Nombre d'exécutions
- Temps humain
- ...

A.1.3 Description du problème en quelques phrases

A.1.4 Séquence provoquant l'erreur (si disponible)

A.2 Phase d'identification et de correction

A.2.1 Changement requis : identification de la cause

- Spécification incorrecte
- Spécification incomplète
- Spécification mal interprétée
- Erreur de typage avec perte de précision
- Erreur de logique
- Débordement ou violation de mémoire
- Boucle infinie
- ...

A.2.2 Pas de changement requis : cause

- Erreur dans le système d'exploitation
- Erreur non répétable
- Erreur rattrapée par le code du programme
- Surcharge du système
- Jeu de test erroné
- ...

A.3 Evaluation du coût

A.3.1 Taille et difficulté de la correction

- Temps CPU
- Nombre d'exécutions
- Temps humain
- Niveau des changements : analyse, code source, interface, ...
- Nature des changements : documentation, nombre de lignes de code source, nombre de modules,...
- ...

References

[1] D.W. Baron. *Techniques récursives en programmation.* Dunod, 1970.

[2] Thierry BRUGÈRE and Alain MOLLARD. *Mathématiques à l'usage des informaticiens.* Ellipses, 2003.

[3] Edmund M. CLARKE and Jeannette WING. Formal Methods : State of the Art and Future Directions. Technical Report CMU-CS-96-178, Carnegie Mellon University, August 1996.

[4] BOLIGNANO D., LE MÉTAYER D., and C. LOISEAUX. Approches Formelles pour l'Aide au Développement de Logiciels. *TSI, numéro spécial*, 20, October 2001.

[5] François MONIN. *Introduction aux méthodes formelles.* Collection Technique et Scientifique des Télécommunications. Hermès, 2000.

[6] Common Criteria Organisations. *CC-User Guide.* Official Documents, 1999.

[7] Common Criteria Organisations. *CC-An Introduction.* Official Documents, 2004.

[8] BEHM P., DESFORGES P., and J-M. MEYNADIER. Meteor : An Industrial Success in Formal Development. *LNCS*, 1393, October 1998.

[9] The Coq Development Team. LogiCal Project. *The Coq Proof Assistant. Reference Manual, v8.0.* INRIA, 2004-2006.

[10] John Watkins. *Test Logiciel en pratique.* Vuibert, 2004.

[11] Spyros Xanthakis, Pascal Régnier, and Constantin Karapoulios. *Le test des logiciels.* Hermès, 2000.