

Rapport Technique - Phase 2
Étude et amélioration de la démarche de test
liée à POK

Telecom ParisTech

22 octobre 2010

Introduction

Le travail réalisé en phase 2 est constitué de deux tâches. La fonction de ce rapport est de faire le point sur les enjeux réels de l'utilisation de l'outil `xcov` dans une démarche de test de «systèmes enfouis», ou dont le code est automatiquement généré. Dans une seconde partie, nous présentons un outil automatisant une partie de l'interprétation des rapports fournis par `xcov`. Ce travail a été réalisé dans le cadre du projet COUVERTURE. Cette interprétation a été mise en oeuvre pour du code C mais pourrait être réalisée pour n'importe quel langage de programmation source dont les structures de contrôle entraînant des «choix» sont identifiables dans la grammaire du langage.

Tout d'abord, nous réalisons un bilan sur la démarche de test entreprise en phase 1 sur le code du support d'exécution POK. Ensuite, nous proposons un outil permettant d'accélérer la lecture des informations produites par `xcov` pour une application codée en C. Ce rapport est organisé en deux chapitres structurés de la manière suivante :

- État des lieux sur la démarche de test de POK utilisant `xcov`
 - Les démarches de test
 - L'environnement de test de POK
 - Analyse d'une cas pathologique.
- Conception et réalisation de l'analyseur de rapport `xcov`
 - Analyse des entrées sorties possibles de l'analyseur.
 - Présentation du prototype, de sa logique de fonctionnement, de ses limitations
 - Présentation d'une mise en page lisible à partir du fichier XML.

Chapitre 1

État des lieux sur la démarche de test de POK utilisant *xcov*

1.1 Présentation de la démarche de test lié à POK

L'évaluation de la couverture de code d'un système testé est requise du point de vue du processus DO-178C pour qualifier le logiciel embarqué. Lors de la première phase du projet, nous avons établis un ensemble de cas de test servant au test des fonctionnalités d'un support d'exécution partitionné, POK [POK]. Avant d'aller plus loin, nous tenons à faire un bref rappel du contexte du test, en général, puis des contraintes spécifiques introduites par le processus DO-178C. Par la suite nous utiliserons le terme "noyau" pour désigner la fraction logiciel du support d'exécution (ici POK par exemple).

1.1.1 Les démarches de test, et POK

Il est possible de distinguer trois types de démarches de test liées à l'évaluation de la fiabilité du logiciel :

- le test unitaire (cible de l'approche MC/DC) permettant d'éliminer les fautes de développement au sein du code métier (essentiellement) [DL00],
- le test d'intégration (test visant à vérifier les propriétés de l'architecture complète du système en fonctionnement «normal») permettant de s'assurer de la bonne coopération des différents modules du système,
- le test de robustesse (visant à vérifier le comportement du système dans un environnement adverse) permettant d'évaluer certains attributs de la sûreté de fonctionnement du système par exemple [FSMA99].

Malgré les récents progrès dans la conception des campagnes de tests du logiciel, le test d'un noyau reste une opération relativement difficile. Le test unitaire n'apporte pas nécessairement un degré de précision suffisant pour que ces tests seuls servent d'arguments pour justifier le bon fonctionnement d'un noyau ou d'un système d'exploitation.

En effet, un noyau possède un état partagé à moitié un nombre important de variables logique et des registres dans matériel physique qu'il contrôle. Dans tous les cas, la plupart des appels systèmes ont un effet de bord sur cet état. Ainsi, les appels systèmes ne sont pas des «fonctions» puisqu'elle cet effet de bord est habituellement l'objectif principal de l'appel système.

Ainsi, le test d'un système d'exploitation se réalise à une granularité assez grossière, en termes d'unités de test. Il est pratiquement impossible de tester les appels systèmes sans déployer une fraction très importante du système d'exploitation. Pour tester POK, nous avons choisis de déployer le système d'exploitation dans son intégralité pour mener les tests. Chaque test cible une partie des fonctionnalités offertes par le noyau et génère les rapport de tests correspondant.

1.1.2 Test de systèmes configurables

Le support d'exécution POK est hautement configurable. Cette fonctionnalité est un avantage pour tout ce qui touche à l'optimisation des performances et la réduction de l'empreinte mémoire. Cependant, cela représente un réel défi pour toute la démarche de certification du système. En effet, en théorie chaque configuration devrait être testée pour s'assurer du bon fonctionnement du système. Heureusement, il existe un nombre très réduits d'implémentations différentes d'une même fonction. La configuration se borne dans la plupart des cas à désactiver un service.

Usuellement, un système d'exploitation est testé grâce à une charge : une application artificielle qui réalise une séquence d'appels systèmes correspondant au test. Nous avons réalisé en phase 1 plus d'une trentaine de configurations et charges pour couvrir les services d'ordonnancement, les fonctions de communications intra-partitions (entre tâches d'une même partition), inter-partitions et réseau, ainsi que les fonctionnalité de gestion des erreurs et chiffrement des données. Ce travail a principalement été réalisé durant la phase 1 du projet COUVERTURE.

1.2 L'environnement de test intégrant *xcov*

Les tests conçus pour POK sont exécutables sous deux cadres d'exécution destinés à évaluer les tests : *xcov-QEMU* et *SPOQ*. Ces deux outils ont des objectifs différents. *SPOQ* concentre une chaîne d'outils permettant de déterminer si le test s'est correctement exécuté (notamment en utilisant les traces d'«événements système» générés par le support d'exécution *QEMU* modifié pour POK.

L'outil *xcov* permet de déterminer, pour un test ou une campagne de tests, le taux de couverture de code obtenu lors de l'exécution du test. POK est compilé grâce à un compilateur C standard. Il était donc impossible d'utiliser les obligations de couverture de code sur POK, seule l'analyse de taux de couverture de code objet fournie par *xcov* était applicable. Cette analyse établit le lien entre les adresses mémoires parcourues lors de l'exécution du test et le code source assembleur utilisé pour générer le binaire testé. Cette information est directement visible si l'on utilise le format de sortie "asm".

xcov supporte un second niveau d'interprétation permettant de lier, grâce aux informations de débogage, ce code assembleur à au code utiliser pour programmer le système (source). La qualité de cette association dépend de celle des informations de débogage permettant d'assurer la traçabilité entre le code source et le code objet. Cette association suppose qu'une seule instance de la fonction réside en mémoire, ce qui est tout à fait naturel dans le cadre du test unitaire pour une fonction en C ou Ada. Cette hypothèse n'est plus vérifiée dans le cas d'un système partitionné. Adacore a adapté l'outil *xcov* de façon à supporter le suivi de la couverture pour chaque instance d'une fonction dans le cas où son code varie (et apparaît donc plusieurs fois dans le binaire).

Des scripts intégrés à POK permettent d'éditer les rapports de couverture pour chaque instance d'une fonction et donc de s'assurer que les rapports de couverture correspondent bien à l'implémentation étudiée.

Les étapes du processus de test décrit dans la figure 1.1 sont les suivantes :

- La préparation de la cible 1) repose sur des modèles AADL permettant de modéliser l'architecture de la cible sous forme de composants aux interfaces explicitement définies.
- L'assemblage de la cible en prévision de son exécution par *xcov* est assurée par un ensemble de scripts et la boîte à outils Ocarina. Ocarina regroupe des fonctions de vérification des modèles AADL, le processus de configuration de POK, et celui de génération du code glu assurant l'interfaçage entre les différents composants métiers de l'application et le support d'exécution. Le code source supporté est C et Ada pour la partie applicative et C uniquement pour l'intergiciel.

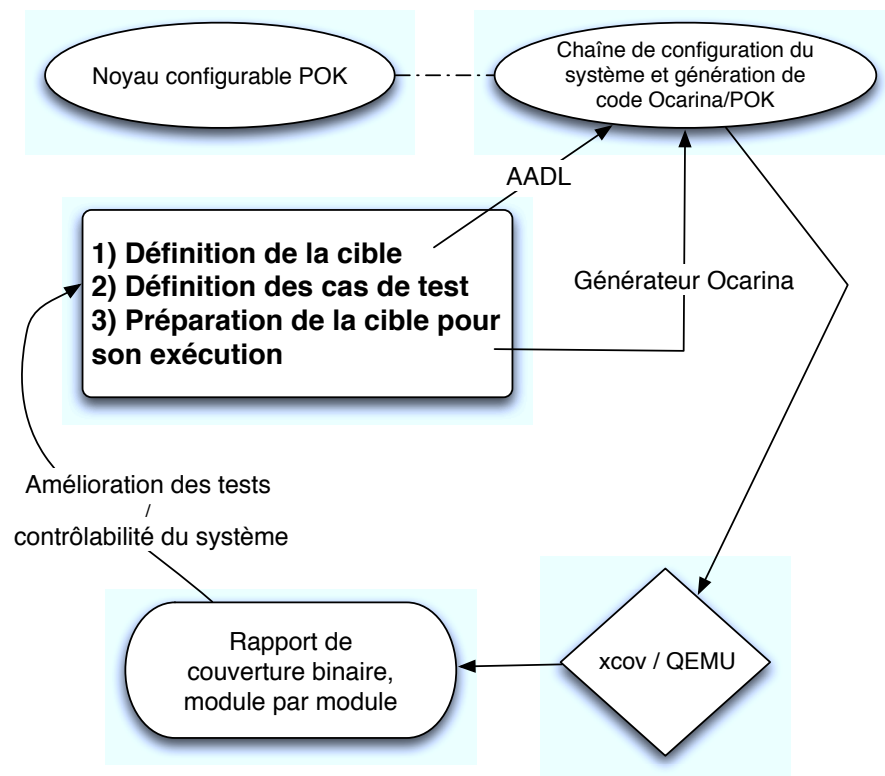


FIGURE 1.1 – Description de haut niveau du processus de test

- L'exécution du test, la collecte de la trace d'exécution et la production du rapport de couverture sont assurées par l'outil *xcov*. L'enchaînement de ces étapes est défini dans un script inclus dans la chaîne de compilation de POK.
- L'interprétation du rapport consiste à déterminer les causes des défauts de couverture de code. Cette interprétation nécessite de connaître la sémantique du code source cible. Cette interprétation doit permettre d'identifier les défauts de couverture faciles à éliminer, par exemple à travers l'altération des données d'entrée.
- Une fois que l'interprétation est faite, les cas de test doivent pouvoir être corrigés de façon à augmenter la couverture des cas de test. Dans le cas des systèmes enfouis, il se peut qu'il ne soit pas possible de sensibiliser toutes les branches du codes surtout si ces branches correspondent à du recouvrement d'erreur dans les appels systèmes.

1.3 Présentation d'un cas pratique

Les fonctions de synchronisation au sein d'une partition ne sont pas directement utilisables mais sont utilisées par la bibliothèque *libpok* qui fournit l'interface de programmation compatible ARINC-653. Deux approches sont possibles pour tester ces fonctions (dont l'implémentation est concentrée dans `lockobj.c`) :

- soit nous évaluons leur impact sur l'état du noyau de manière ponctuelle en isolation de l'exécution de l'ordonnanceur.
- soit nous tentons d'établir des scénarios de déploiement complets permettant de sensibiliser ces services pour la campagne de test en parallèle de la fonction d'ordonnancement.

La seconde solution peut sembler séduisante mais pose le problème suivant dans le cas des primitives de synchronisation : un appel bloquant maintiendra le thread de l'application de test soit définitivement dans l'état bloqué, soit ne suspendra jamais son activité (en supposant que le thread de test est le seul à accéder au verrou). Le test unitaire fournit la notion de contexte pour pallier ce problème. Cependant, ce contexte est censé rester figé à partir du début de l'exécution du système. Le danger réside dans cette notion de contexte dynamique réside dans la complexité de sa mise en oeuvre. Cela devient presque aussi complexe que de déployer le noyau.

Ce constat nous a amené à choisir la seconde approche. Dans la seconde approche, l'objectif est de créer des scénarios d'interactions suffisamment riches entre l'application (la charge de test) et le support d'exécution (la cible).

La difficulté réside donc dans la constitution de ces scénarios puisque l'application n'a pas directement accès à l'intégralité des fonctions à tester. Nous tombons alors dans des problèmes liés à la testabilité de systèmes enfouis. Le premier test mené consiste en un appel simple à la fonction envoyant une donnée sur un tampon partagé entre deux tâches de la partition (ce qui déclenche la prise et le relâchement de verrou sans suspension des tâches). Nous obtenons un taux de couverture inférieur à 50% à cause de l'état du noyau au moment de l'appel. Il est possible d'observer la cause de ce faible taux de couverture sur le rapport pour la fonction `pok_lockobj_lock ()` du module `lockobj.c`. La majorité du code non-couvert est concentré dans une branche non sensibilisé d'une structure conditionnelle "if... then... else".

```

287.pok_ret_t pok_lockobj_lock (pok_lockobj_t* obj, const pok_lockobj_lockattr_t* attr)
288.#{
289.    uint64_t timeout = 0;
290.
291.    if (obj->initialized == FALSE)
292.    {
293.        return POK_ERRNO_LOCKOBJ_NOTREADY;
294.    }
295.
296.    SPIN_LOCK (obj->spin);
297.
298.    if ( (obj->is_locked == FALSE ) && (obj->thread_state[POK_SCHED_CURRENT_THREAD] == LOCKOBJ_STATE_UNLOCK ))
299.    {
300.        obj->is_locked = TRUE;
301.        SPIN_UNLOCK (obj->spin);
302.    }
303.    else
304.    {
305.        /*
306.         * attr->time corresponds to the timeout for the waiting object
307.         */
308.        if ((attr != NULL) && (attr->time > 0))
309.        {
310.            timeout = attr->time;
311.        }
312.
313.        while ( (obj->is_locked == TRUE ) || (obj->thread_state[POK_SCHED_CURRENT_THREAD] == LOCKOBJ_STATE_LOCK) )
314.        {
315.            obj->thread_state[POK_SCHED_CURRENT_THREAD] = LOCKOBJ_STATE_LOCK;
316.
317.            if (timeout > 0)
318.            {
319.                pok_sched_lock_current_thread_timed (timeout);
320.                if (POK_GETTICK() >= timeout)
321.                {

```

FIGURE 1.2 – Couverture dépendant de l'état du noyau - extrait du module `lockobj.c`

Le problème de l'amélioration de ce cas de test repose sur la compréhension du parcours du graphe de contrôle de l'application. Cela suppose que l'on soit capable de comprendre la position des lignes non couvertes dans ce graphe et les conditions sur lesquelles il est nécessaire d'agir pour activer les branches du système dans un état réaliste du noyau.

Pour cela, nous proposons d'interpréter partiellement le code pour identifier sa structure intermédiaire entre la fonction et la ligne de code source.

Cela revient essentiellement à identifier les structures de contrôle du flot d'exécution propres au langage source utilisé.

Nous nous sommes focalisé sur le langage C mais la démarche doit pouvoir être adaptée à d'autres langages.

Chapitre 2

Prototype de présentation et parcours des rapports xcov

Le but de la démarche est de permettre une lecture rapide des rapports permettant de connaître pour un ensemble de ligne non couverte la position de ces lignes non couverte dans la structure interne du programme.

Cette connaissance est essentielle pour décider si le défaut de couverture pourra être corrigé facilement. De manière grossière cela permet de déterminer rapidement le nombre et la nature des conditions au niveau code source à contrôler pour atteindre les lignes non couvertes. Tout ce que nous dirons par la suite s'applique au cas de l'étude de la couverture de code objet.

2.1 Définition des entrées/sorties de l'analyseur

xcov fournit quatre formats de sorties pouvant servir d'entrée à notre outil. Ces formats attachent l'information de couverture à 5 niveaux d'abstraction possible d'un code objet : le programme complet, le module (le fichier lors de l'édition de lien), la fonction, la ligne de code source, et la ligne de code objet.

Les deux derniers disposent d'un état discret binaire ou ternaire. Les trois premiers se voient attacher un taux de couverture (précision de l'ordre du pourcentage). En général, il est possible d'attacher le nombre de lignes couverte et non couvertes dans le code assembleurs correspondant aux 4 premières représentations constituées, a priori, d'un ensemble de lignes de code objet.

Ce sont ces décomptes de lignes, couvertes ou non, que nous calculerons pour les structures intermédiaires entre la ligne de code source et la fonction.

Sortie du programme : À l'issue d'une exécution d'un test et de l'analyse du rapport, nous souhaitons pouvoir parcourir la structure d'une fonction depuis la fonction jusqu'au bloc «bloc» séquentiels élémentaire constituant son code. Cela signifie que la sortie de l'interpréteur selon ses structures de contrôle et avoir une synthèse des informations de couverture pour chaque structure de contrôle de l'exécution de l'application.

Entrée : Nous ne n'avons pas l'intention d'analyser le code assembleur. Ceci nous a poussé à considérer les sorties de rapport `xcov` masquant ce niveau de détail : les rapports bruts `xcov` et les rapports formatés `html`.

Le cas de la sortie `xml` est à part. Nous avons choisi de cibler les rapports bruts car ils étaient plus simples à analyser et beaucoup plus proches d'un fichier de code source. Ce choix est peut être limitant à long terme, mais cela nous a permis de mettre rapidement en oeuvre le prototype d'analyse à partir d'une grammaire existante pour analyser du code source C.

2.2 Principe de l'analyse du rapport brut «xcov»

Pour décrire l'analyse réalisée sur le rapport, nous devons en décrire la structure. Les rapports brut `xcov`, pour une application codée en C, sont constitués de lignes de code source C précédées d'un «prompt» déclarant les informations liées à la ligne qui suit.

Dans le cadre d'une analyse de couverture au niveau instruction, cet état peut avoir trois valeurs : «-» signifie non couvert, et «+» signifie couvert, et «.» signifie que la ligne de code source en question ne correspond à aucun code objet (soit c'est une déclaration, soit le processus d'optimisation a éliminé cette ligne).

Dans la table ci-dessous, nous avons reporté trois lignes différentes pour former un échantillon représentatif de ce que l'on trouve habituellement dans ce type de rapports.

```
10  . :  int i = 1 ;
21  + :  while (i<0) {
22  - :  i++ ;}
```

Nous utilisons une grammaire et un lexique de type `lex/yacc` pour analyser les rapports. Cette analyse se fait en deux étapes. Tout d'abord nous consommons l'information de couverture associée aux lignes lors de l'analyse lexicale, puis nous analysons la syntaxe du code C.

L'analyse syntaxique repose sur une grammaire constituée de règles de "production" de type BNF permettant d'identifier les structures de haut ni-

veau du code. Ainsi, il existe des règles permettant d'identifier une fonction, une boucle, une condition Le parcours de ces règles permet de générer une structure hiérarchique rendant compte de la structure du programme et la décorant par les informations de couverture. Cela nous permet concrètement de générer des balises XML et leur contenu pour représenter le rapport `xcov` sous une forme hiérarchique dépendant du code source.

Nous avons des balises spécifiques pour identifier :

- Les fonctions
- Les structures `if then \[else\]` et les blocs `{...}` contenu dans chaque branche.
- Les boucles

Le contenu associé à ces balises permet d'évaluer rapidement la position des lignes non couvertes dans la hiérarchie de structures de contrôles constituant le code de la fonction.

Nous attachons à chaque bloc "séquentiel" un résumé de couverture. Ces structures ont la propriété d'être associées à un ou plusieurs blocs séquentiels `{...}`. Un résumé de couverture pour un ensemble de lignes de code consiste en un triplé définissant le nombre de ligne couverte, non-couverte, plus celle qui ne sont pas incluses à la génération de code objet. Nous faisons remonter les résumés de couverture de structure en structure au moment de la réduction des règles de production de la grammaire.

Prenons l'exemple du code suivant :

```

4  +: void f(int* x, y) {
5  +:     if ( (x*/y*)<5)
6  .:     {
7  -:         x*=x*+1;
8  -:         y*=y*/2+1;
9  .:     }
10 +: else y*=y*+1;
11 +: }
```

Dans ce contexte, l'information de couverture associée au bloc entre la ligne 6 et 9 est (0,2,2). Chaque bloc d'une structure dispose de son propre résumé. A plus haut niveau, tous les résumés d'une structure sont additionnés pour les fusionner en un unique résumé. En procédant de la sorte, nous sommes capables de concentrer les résumés pour améliorer la lisibilité des rapports lorsque plusieurs structures sont incluses les unes dans les autres.

Remarque : la détection des branches issues de structures `switch / case` n'a pu être assurée totalement à cause de la nature optionnelle de l'instruction

break avant un autre case. Nous nous sommes contentés de fournir un unique rapport pour l'ensemble de la structure.

2.3 Format du fichier XML

Voici la structure de nos fichiers XML :

```
report :: '<report>' target type coverage functionlist '</report>'
target :: '<target>' STRING '</target>'
type    :: '<type>' STRING '</type>'
coverage :: '<coverage>' INTEGER '</coverage>'
functionlist :: '<function_list>' funclist '</functionlist>'
funclist :: funclist '<function name =' STRING '>' block '</function>'
block    :: '<block nb_cov =' INT 'nb_uncov=' INT 'nb_undetermined=' INT '>'
           structlist '</block>'
structlist :: structlist struct;
|           ;

struct    :: '<iter_or_loop>' block '</iter_or_loop>';
|         '<if_then>' block '</if_then>';
|         '<if_then_else>' block block '</if_then_else>';
|         '<switch_node>' block '</switch_node>'
```

2.4 Exemple de mise en page

A partir du fichier XML, nous générons une mise en page "clickable" permettant de naviguer efficacement dans le rapport. Nous avons pour cela fourni une feuille de style adaptable en fonction des besoins de l'utilisateur. La mise en page résultante est illustrée sur la figure 2.1. Cette fiche peut aussi être réutilisée pour traiter une version étendue du fichier XML (par exemple une version où l'on aurait mis bout à bout les deux versions du fichier). Noté pour la fonction `pok_cons_write()` les informations de couvertures indiquent qu'il y a 9 lignes couvertes et 11 de non couvertes. Le rapport `xcov` est visible à droite et peut être consulté pour avoir le détail du code. A l'origine le détail des fonctions est masqué, l'utilisateur doit cliquer sur le nom de la fonction pour en faire apparaître le détail.

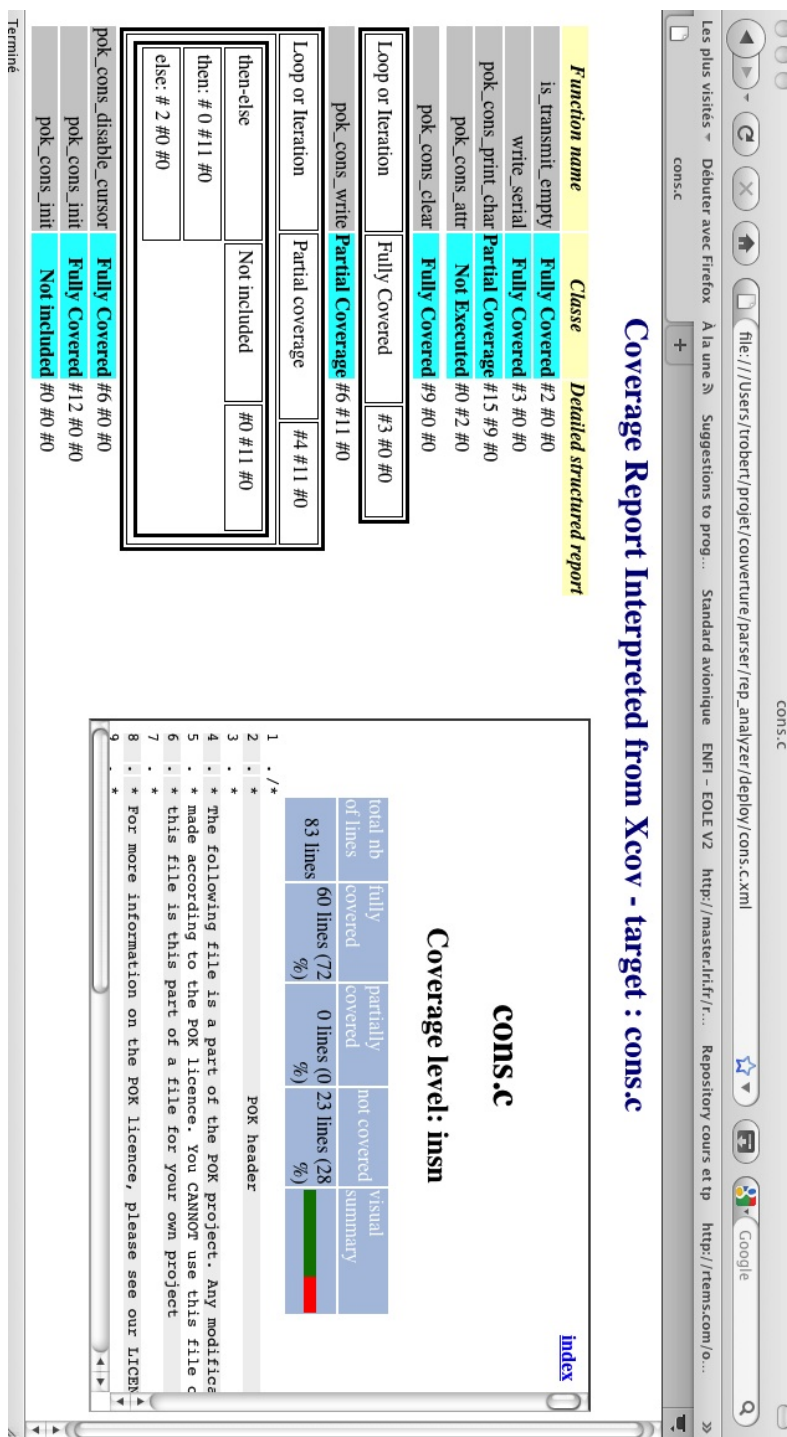


FIGURE 2.1 – Formatage du rapport interprété pour cons.c

Chapitre 3

Conclusion

Cet outil nous a permis de parcourir assez rapidement de nombreux rapports de couverture pour chaque déploiement. Le but final est de permettre de mettre à jour la charge applicative afin d'améliorer le taux de couverture du code du support d'exécution.

Les bénéfices sont doubles. L'intérêt de la démarche réside dans le fait qu'il n'est pas nécessaire d'avoir utiliser un compilateur spécifique pour générer le binaire du code testé. Le même travail peut être réalisé pour d'autres langages où de manière plus efficace en tentant par exemple de traiter le cas des "switch". Deuxièmement, nous montrons un exemple d'interprétation des rapports qui permet de guider la démarche d'amélioration des tests (par exemple, pour la fonction `pok_cons_write`, nous avons pu améliorer le taux de couverture en identifiant le fait que le défaut de couverture provenait d'une condition directement atteignable dans le corps de la fonction qui cependant n'était jamais satisfaite par la charge utilisée pour le test. La présentation offerte par notre outil permet de sélectionner les sections de codes qui seront les plus "faciles" à atteindre à priori.

Bibliographie

- [DL00] A. Dupuy and N. Leveson. *An empirical evaluation of the mc/dc coverage criterion on the hete-2 satellite software. volume 1, pages 1B6/1 –1B6/7 vol.1, 2000.*
- [FSMA99] J.-C. Fabre, F. Salles, M. Rodríguez Moreno, and J. Arlat. *Assessment of COTS microkernels by fault injection. In B. Weinstock, Charles and John Rushby, editors, Proceedings of the Conference on Dependable Computing for Critical Applications (DCCA-99), pages 25–46, Los Alamitos, CA, 1999. IEEE Computing Society.*
- [POK] *A partitionned operating kernel, [http ://pok.gunnm.org/](http://pok.gunnm.org/).*