Code-coverage sur les systèmes embarqués

Couverture de test sur petites cibles

Même sur les systèmes embarqués, les logiciels doivent être testés suffisamment avant que le produit puisse être lancé sur le marché. Cependant, une telle surveillance dans les progrès des tests, exigée par de nombreuses normes de sécurité, n'est pas tout à fait anodin à ce stade. Souvent, le volume des ressources des cibles embarqués est limité, la mémoire et le processeur peuvent facilement atteindre leurs limites. Mais avec un peu de créativité, il est cependant possible de mesurer et prouver la couverture de test dans le domaine des systèmes embarqués.

Par Klaus Lambertz, gérant de Verifysoft Technologies GmbH

Les exigences pour les systèmes embarqués augmentent de plus en plus. Surtout, en ce qui concerne l´ IoT (l'Internet des objets connectés), les systèmes embarqués y prennent une place importante, voire critique. Selon une étude de juin 2018 de la société de conseil IDC, le marché des technologies de l'IoT va croître au rythme de 13 % par an pour atteindre un chiffre d'affaire global de 1.2 milliards de dollars d'ici 2020. Les systèmes embarqués devenant une part de plus en plus importante de l'activité de beaucoup d'entreprises, la sécurité de ces systèmes deviendra-elle aussi une priorité, comme elle est déjà une composante essentielle du développement de ces systèmes dans d'autres domaines à l'heure actuelle.

Dans le domaine de l'équipement médical, l'aéronautique, l'automobile ou l'industrie ferroviaire, des normes strictes régissent le test et la certification des systèmes embarqués pour que ces derniers soient mis sur le marché. En dehors de ces branches traditionnellement très axées sur la sécurité, il existe manifestement un énorme retard à combler. Selon un sondage de Gartner réalisé au printemps 2018, près de 20 % des entreprises sondées ont été victimes d'une attaque informatique via l'IoT (l'Internet des objets connectés). L'une des raisons avancées par Gartner est la lente évolution des normes de sécurité appliquées à l'IoT ; ils manquent encore de Security by Design.

Le Code-coverage est souvent obligatoire

Afin de développer des systèmes embarqués sûrs, les tests sont un levier fondamental. Ce n'est pas sans raisons que les normes de développement de logiciels critiques pour la sécurité des biens et des personnes, incluent des exigences très précises en termes de méthodes et de couverture des tests.

En règle générale, les normes, comme DO-178C en aéronautique ou ISO 26262 dans l'industrie automobile, requièrent la couverture des instructions et des chemins (*Statement Coverage* et *Path coverage*). Le *Modified Condition/Decision Coverage* (MC/DC) est également souvent requis. Fondamentalement, plus les exigences de sécurité d'un logiciel sont élevées, plus la couverture de test requise doit être élevée.
Tous les niveaux de couverture de code n'ont pas de sens dans chaque scénario, et peuvent fournir aussi des informations différentes :

- La couverture des fonctions (Function Coverage) ignore complètement le fonctionnement interne du logiciel, son utilité est donc relativement faible
- La couverture des instructions (Statement Coverage) recherche quelles instructions sont exécutées lors des tests. Elle permet donc d'identifier le code mort et les instructions pour lesquelles il n'y a pas encore de test.

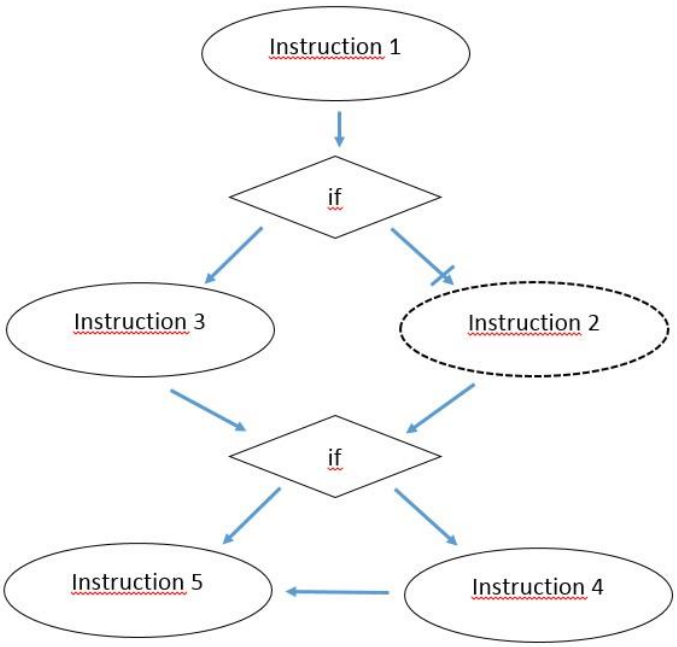
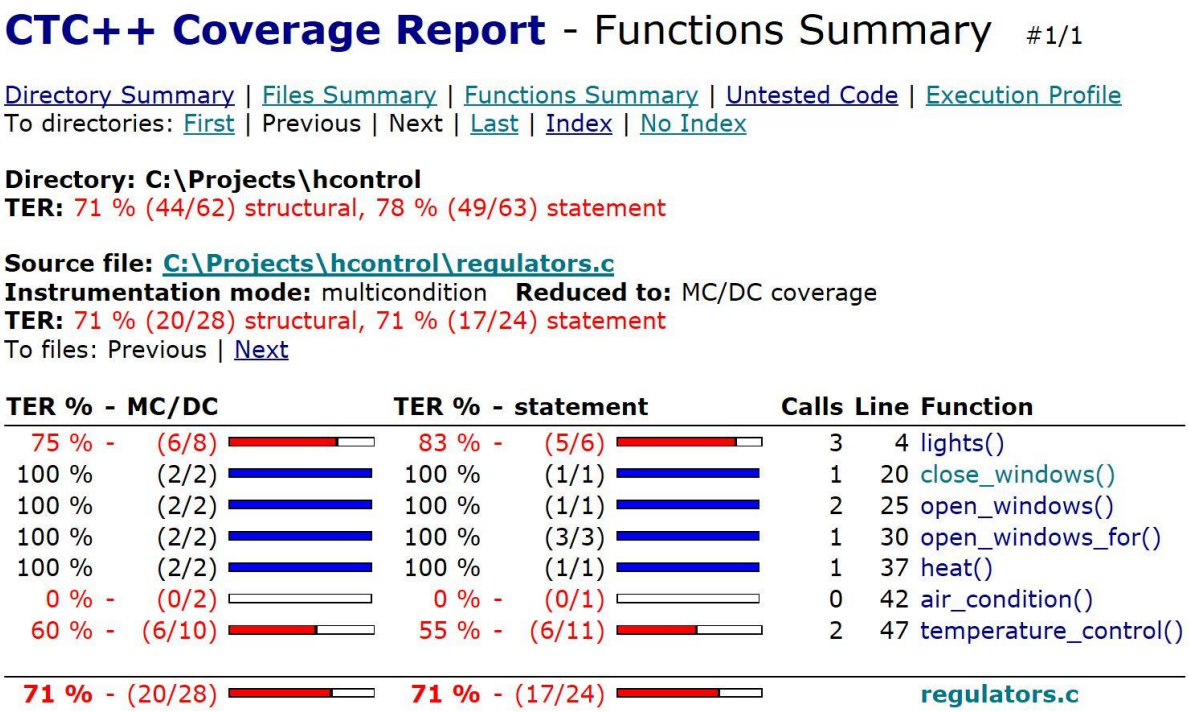


Figure 1: La couverture des instructions indique combien d'instructions ont été exécutées. Dans cet exemple, 6 des 7 instructions ont été exécutées : la couverture des instructions est de 85%

- La couverture des points de tests (Condition Coverage) vérifie si toutes les conditions ont été exécutées et vérifiées au moins une fois. Elle est donc plus complète que la couverture des instructions et est une couverture minimale atteignable exigeant un minimum de tests à un coût raisonnable.
- La couverture de condition/décision modifiée (Modified Condition/Decision Coverage (MC/DC)) est le niveau de couverture de test le plus élevé requis par les normes. L'évaluation de toutes les combinaisons possibles des conditions demandant un travail phénoménal, la couverture de type MC/DC cherche à minimiser l´effort de test. Cette méthode va observer chaque condition atomique d'une condition complexe. Pour chacune de ces conditions atomiques, MC/DC va tester une paire d'entrée qui fait changer le résultat final de la condition complexe sans que les autres conditions atomiques ne changent.

Pour analyser la couverture du code, on a recours à des « Code Coverage Analyser », qui vont compléter le code, avec des compteurs en fonction des niveaux de test souhaités, avant de les transférer au compilateur. Ce processus s'appelle instrumenter le code

Il existe des outils d'une grande simplicité pour ce faire, comme par exemple, le testeur de couverture GNU gcov. En utilisant l'option gcc -ftest-coverage, le code peut être instrumenté pour déterminer combien de fois chaque ligne de code a été exécutée. L'option -fprofile-arcs orchestre les branches. Pour faire ses premières expériences dans le domaine de la couverture de code ou pour de petits projets, cet outil est parfaitement adapté. Outre le manque de niveaux de couverture de test avancés, nécessaires pour des besoins de test plus importants, son inconvénient réside dans le traitement et l'interprétation des informations obtenues. Ici, les outils commerciaux sont clairement supérieurs.



CTC++ Coverage Report - Functions Summary #1/1

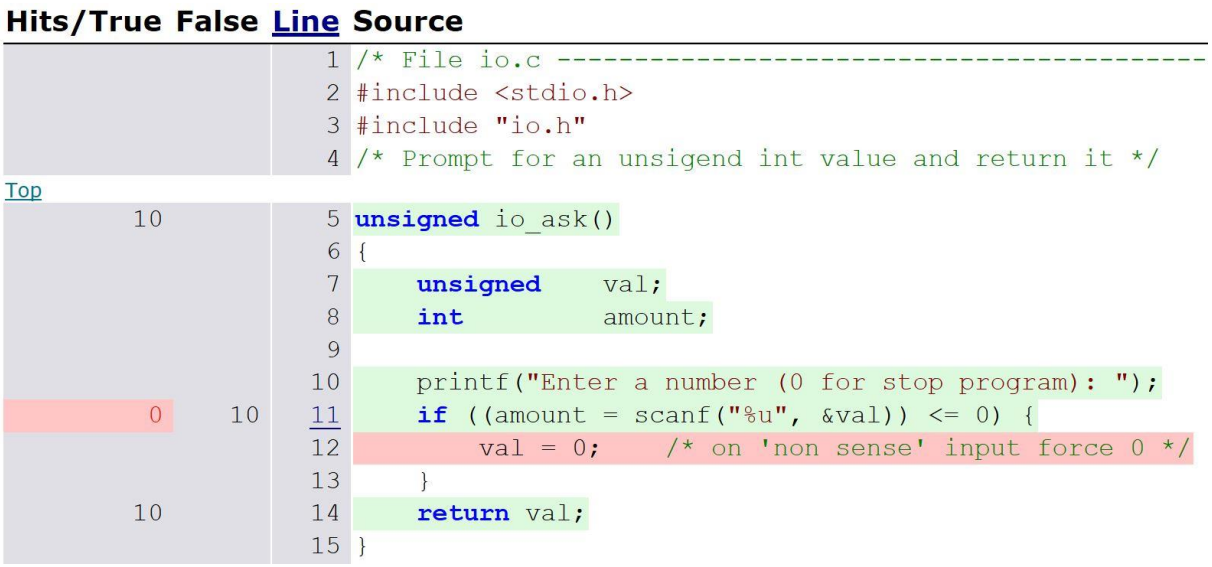
Directory Summary | Files Summary | Functions Summary | Untested Code | Execution Profile
To directories: First | Previous | Next | Last | Index | No Index

Directory: C:\Projects\hcontrol
TER: 71 % (44/62) structural, 78 % (49/63) statement

Source file: C:\Projects\hcontrol\regulators.c
Instrumentation mode: multicondition **Reduced to:** MC/DC coverage
TER: 71 % (20/28) structural, 71 % (17/24) statement
To files: Previous | Next

TER % - MC/DC		TER % - statement		Calls	Line	Function
75 % -	(6/8)	83 % -	(5/6)	3	4	lights()
100 %	(2/2)	100 %	(1/1)	1	20	close_windows()
100 %	(2/2)	100 %	(1/1)	2	25	open_windows()
100 %	(2/2)	100 %	(3/3)	1	30	open_windows_for()
100 %	(2/2)	100 %	(1/1)	1	37	heat()
0 % -	(0/2)	0 % -	(0/1)	0	42	air_condition()
60 % -	(6/10)	55 % -	(6/11)	2	47	temperature_control()
71 % -	(20/28)	**71 %** -	(17/24)			**regulators.c**

Figure 2: Résumé du rapport de couverture de code fournit par l'outil Testwell CTC++
Au cours de la fonction "lights", 5 des 6 instructions ont été exécutés. Ainsi, la couverture des instructions (TER, Test Effectiveness Ratio) est à 83 %. La couverture structurelle, içi MC/DC, est de 75 %



Hits/True	False	Line	Source
		1	/* File io.c ----------
		2	#include <stdio.h>
		3	#include "io.h"
		4	/* Prompt for an unsigend int value and return it */
Top			
10		5	unsigned io_ask()
		6	{
		7	unsigned val;
		8	int amount;
		9	
		10	printf("Enter a number (0 for stop program): ");
0	10	11	if ((amount = scanf("%u", &val)) <= 0) {
		12	val = 0; /* on 'non sense' input force 0 */
		13	}
10		14	return val;
		15	}

Figure 3: Rapport de Testwell CTC++ avec la couverture de code dans le code source. La condition à la ligne 11 a été exécutée 10 fois comme fausse et jamais comme vraie. Pour atteindre une couverture complète, il faudrait encore atteindre cette condition.

L'instrumentation étend le code

Les solutions professionnelles fonctionnent sur le même principe de base que gcov et instrumentent le code. Les compteurs sont de manière générale stockés sous forme de tableaux globaux. Quand et où ces compteurs seront modifiés dépendra du niveau de couverture configuré. L'exemple suivant d'une boucle while en C montre quelle est la conséquence de l'instrumentation :

```
while (! b == 0 )
{
   r = a % b;
   a = b;
   b = r;
}
result = a;
```

Suite à l'instrumentation, dans ce cas réalisée par l'outil de couverture de code Testwell CTC++, on obtient la structure suivante :

```
while ( (( ! b == 0 ) ? (ctc_t[23]++, 1) : (ctc_f[23]++, 0)) )
{
   r = a % b ;
   a = b ;
   b = r ;
}
result = a ;
```

Des ressources précieuses

L'instrumentation fait grossir le code. Les tableaux requis se trouvent dans la mémoire-données, il faut donc plus de mémoire dans la RAM, mais aussi dans la ROM. L'instrumentation va aussi influencer les temps d'exécution. Dans des applications sur serveur ou ordinateur de bureau, il est possible de négliger ces effets, mais dans les systèmes embarqués, il n'est souvent pas possible de le faire, les ressources étant calculées au plus juste pour des raisons de coût. Il convient de veiller à utiliser un analyseur de couverture de code avec un impact d'instrumentation faible, car sinon les compteurs dépasseraient rapidement les limites de la mémoire disponible. Cela est particulièrement vrai lorsque des niveaux de couverture de test très exigeants, tels que MC/DC, sont requis.

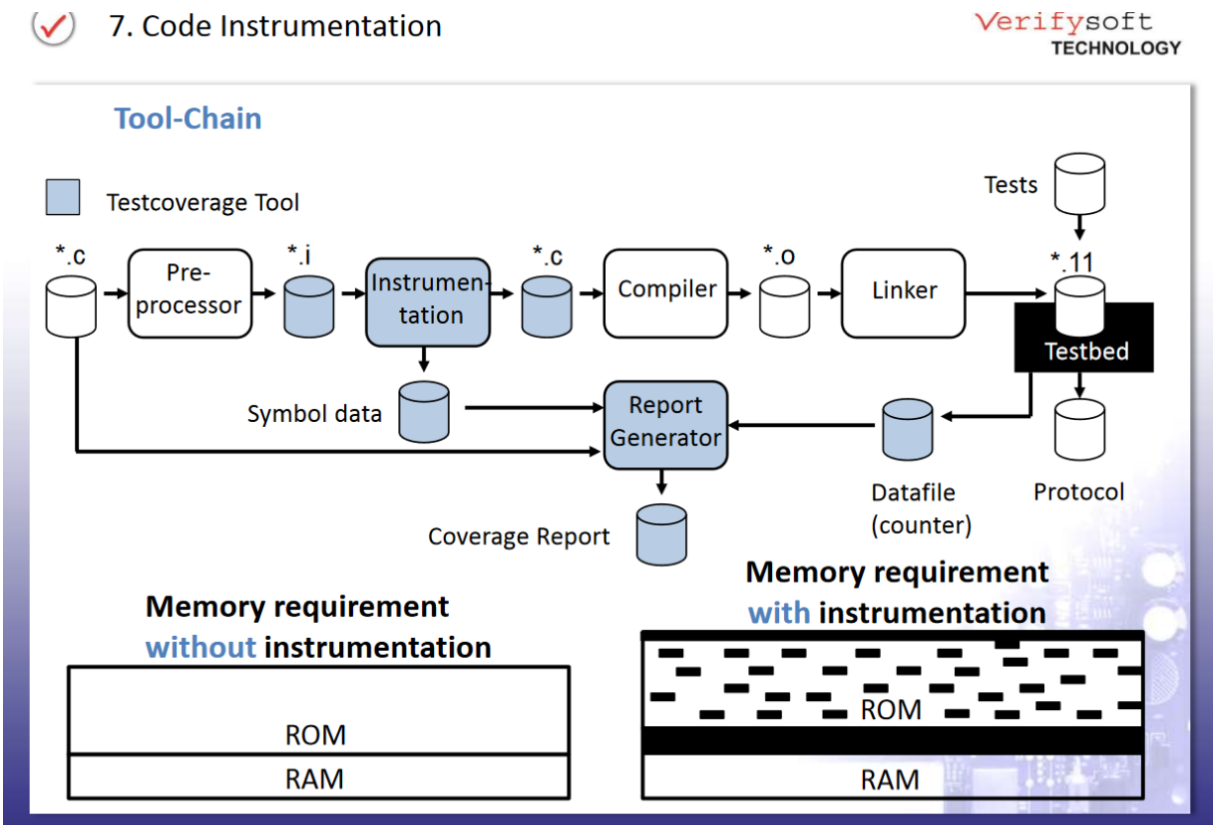


Figure 4 : En raison de l'instrumentation, les besoins en ressources augmentent drastiquement

Si l'outil de couverture de code a tout de même un impact trop élevé sur la RAM, il est possible de ne faire qu'une instrumentation partielle, au cours de laquelle de petits morceaux du programme sont testés les uns après les autres. Les tests sont alors exécutés successivement et les données récoltées sont fusionnées. Il est alors possible de vérifier la couverture du code pour l'ensemble du programme. Une autre approche sur les petites cibles consiste à limiter la taille des compteurs. En temps normal, les outils de couverture de code utilisent des compteurs de 32 bits mais il est théoriquement possible de réduire ceux-ci à 16 ou même 8 bits. Il faudra alors faire attention aux risques de dépassement de capacité. Dans le cas de compteurs réduits, il faut donc accorder une attention toute particulière à l'interprétation des données. Dans les cas les plus extrêmes, il est même

possible de réduire le compteur à un unique bit, quand il n'est pas intéressant de savoir combien de fois une portion du programme est traversée.

Besoins en ROM	
Sans instrumentation	60 Byte
Couverture des fonctions	67 Byte
Couverture branches	118 Byte
Couverture des points de tests	285 Byte
Besoins supplémentaires en RAM sans le Bit-Coverage (compteurs 32 bits)	
Couverture des fonctions	4 Byte
Couverture des branches	16 Byte
Couverture des points de tests	28 Byte
Besoins supplémentaires en RAM avec le Bit-Coverage	
Couverture des fonctions	1 Bit
Couverture des branches	4 Bit
Couverture des points de tests	7 Bit

Figure 5: Grâce au Bit-Coverage fourni par Testwell CTC++, les besoins en RAM se laissent considérablement réduire

Le niveau de couverture est influe également sur la quantité de RAM requise. La surcharge de la ROM n'est par contre presque pas influençable. L'acquisition des données obtenue par l'instrumentation, on a recours à une petite bibliothèque, notamment pour assurer la transmission des valeurs des compteurs à l'hôte. De plus, l'instrumentation ne charge pas uniquement la mémoire, mais aussi le processeur cible. Il est donc possible qu'un timing ne soit plus respecté, en particulier quand le processeur est déjà proche de 100 % de charge, il est possible que des erreurs aient lieu. La communication par les différents bus en est particulièrement affectée. C'est au testeur d'être particulièrement attentif à ce genre de comportements et de bien vérifier les résultats obtenus. Les outils de couverture de code professionnels sont néanmoins capables de ne pas influencer outre mesure les besoins supplémentaires en mémoire et puissance de calcul.

Conclusion

Les tests et la mesure de la couverture des tests deviennent des thèmes de plus en plus importants dans le développement pour les systèmes embarqués. Parce que, la qualité du logiciel devient de plus en plus importante et, même si tous les standards et normes n'exigeront pas 100 % de couverture MC/DC pour tous les types de logiciels, ce n'est qu'une question de temps avant que les commissions de standardisation et les associations d'industriels ne rehaussent les exigences pour les logiciels n'ayant pas d'exigences de sécurité. De meilleurs tests sont également dans l'intérêt du fabricant : en effet, des produits défectueux occasionnent d'importants coûts, sans compter les dommages causés à la réputation de la société. Le « Banana principle », celui du logiciel qui mûrit chez le client, bien connu dans les applications de bureau, ne sera certainement pas acceptable dans le développement pour plateformes embarquées.