Équivalence entre Propriétes Simulink et Critères de Couverture de Test

Manel Tekaya
Université de Carthage
TELNET Innovation Labs, Tunisie
manel.tekaya@telnet-consulting.com

Mohamed Taha Bennani
Université de Tunis ElManar
Taha.Bennani@enit.rnu.tn

Samir Ben Ahmed
Université de Tunis ElManar
Samir.Benahmed@fst.rnu.tn

Anis Youssef
TELNET Innovation Labs, Tunisie
anis.Youssef@groupe-telnet.net

Résumé

Les normes de sécurité, telles que la norme ISO 26262, mettent l'accent sur le "Model Based Development at software level" afin d'argumenter la conformité entre un modèle et son implémentation. Dans ce cadre, des critères de couverture structurelle et comportementale sont définis afin de générer un ensemble d' entrées de test. Le modèle ainsi que son implémentation génèrent leurs résultats respectifs aprés l'exécution des entrées de test. Si ces résultats sont différents, un problème de conformité sera levé. TELNET Innovation Labs modélise les systèmes de contrôle automobile et aéronautique en utilisant Simulink/Stateflow (SL/SF). Le test unitaire de ces solutions est effectué en utilisant un processus semi-automatique. Dans l'objectif d'automatiser cette phase, nous définissons dans cet article notre processus qui permet la génération de l'ensemble des entrées de test. Ce processus exploite le potentiel du model checker de générer des contre-exemples relatifs à la violation des propriétés. Nous montrons dans cet article comment nous définissons un ensemble de propriétés qui permettent de générer des contre-exemples équivalents à des entrées de test assurant des critères de couverture standards.

mots clés

Systèmes Embarqués, Test Unitaire, ISO 26262-6, Model checking, Entrées de test

1. INTRODUCTION

La norme ISO 26262 (Véhicules routiers - Sécurité fonctionnelle) est une norme émergente pour les systèmes de sécurité dans les véhicules routiers à moteur. Selon cette norme, chaque risque sera classé selon un niveau d'exigence en terme de sécurité de A à D (Automative Safety Integrety Level - ASIL). L'ISO 26262 :2011 se compose de neuf parties normatives et un dixième chapitre qui constitue le guide d'utilisation de la norme. Nous allons nous intéresser à la partie 26262-6 qui spécifie les exigences pour le développement des produits au niveau logiciel. L'ISO 26262-6 définit sept exigences (5 à 11) parmi lesquelles la *clause* de test unitaire (Exigence N°9). Le test unitaire a pour objectif de tester séparément le bon fonctionnement de chaque module logiciel de bas niveau. Il repose par ailleurs sur la définition de cinq besoins parmi lesquels les critères de couverture (Exigence 9.4.5).

Le tableau 1 présente la classification de ces critères de couverture. Chaque niveau exige l'application d'un ou d'une combinaison de critères de couverture.

Table 1: Classification des critères de couverture structurelle selon l'ISO 26262-6 (Exigence 9.4.5)

Niveau de couverture de test \ ASIL	A	B	C	D
Couverture des instructions	++	++	+	+
Couverture des branches	+	++	++	++
Couverture de condition/décision modifiée MC/DC	+	+	+	++

++ "fortement recommandé", + "recommandé"

Les travaux présentés dans ce article sont réalisés dans le cadre d'une thèse MOBIDOC (Ref. 58/7019) supportée par le projet PASRI de l'union européenne et Telnet-Innovation Labs.

Ces critères de couverture struturelle sont définis comme suit :

- Couverture des instructions : mesure le pourcentage des instructions qui ont été exécutées au moins une fois au cours du test. La couverture du code au niveau “Conditions” est requise.
- Couverture des branches : mesure le pourcentage des flux de contrôle des branches qui ont été pris en compte au cours du test. La couverture du code au niveau “Conditions/Décisions” est requise.
- MC/DC(Modified Condition/ Decision Coverage) : chaque point d’entrée et de sortie dans le programme a été invoquée au moins une fois, toutes les conditions d’une décision dans le programme prend tous les résultats possibles au moins une fois, et chaque décision prend tous les résultats possibles au moin une fois, et chaque condition dans une décision influence indépendement le résultat de la décision. Cette influence est montrée en faisant varier qu’une seule conditon tout en fixant toutes les autres.

Les quatre niveaux ASIL sont : A,B,C et D. Tous les niveaux présentent une combinaison de trois critères de couverture cités ci-dessus. En revanche, pour le niveau A le critère de couverture des instructions est fortement recommandé. Pour le niveau B les critères de couverture des instructions et des branches sont fortement recommandés. Pour le niveau C, le critère de couverture des branches est fortement recommandé. Concernant le niveau D, qui représente le niveau de risque le plus élevé, les critères de couverture des branches et MC/DC sont fortement recommandés.

Le processus de test unitaire de Telnet Innovation Labs est composé de trois étapes. Premièrement, le testeur génère les entrées de test manuellement à partir du modèle sous test. Deuxièmement, il implémente le modèle en utilisant le langage de programmation C selon la norme MISRA [3]. Troisièmement, le testeur fournit à IBM Rational Test Real Time platform (RT-RT)[25]le code C et les entrées de test générés manuellement. Par la suite, il génère ces propres entrées de test en analysant le code C et les compare aux entrées de test qui sont fourni en entrée. Ce processus présente deux problèmes majeurs : le taux de couverture du modèle par les entrées de test générées manuellement et l’évaluation de la conformité entre le modèle et l’implémentation. En effet, ces entrées de test ne peuvent pas garantir une couverture exhaustive du modèle.

Notre objectif vise à alléger l’effort humain, réduire le “*Time to market*” et améliorer l’efficacité de test unitaire. Par conséquent, nous définissons un nouveau processus de test unitaire qui permet de générer automatiquement des entrées de test sous l’environnement Matlab/Simulink utilisé par Telnet Innovation Labs. Ce processus se base sur la technique du model checking [10]. Cette dernière, consiste à prendre en entrée un ensemble de propriétés et permet de générer un ensemble des contre-exemples, si ces propriétés ne sont pas valides. Nous montrons dans cet article que ces contre-exemples correspondent à des entrées de test pouvant assurer un critère de couverture décrit dans le tableau 1, afin d’être conforme à la partie 9.4.2 de l’ISO 26262-6. Nous avons adapté ces critère de couverture sur les modèles Matlab/Simulink logiques, qui peuvent être exprimés sous forme d’expression booléenne.

L’article est organisé comme suit : La section 2 présente les travaux connexes. La section 3 définit les différentes étapes de notre processus de génération des entrées de test. Dans la section 4, nous montrons qu’un ensemble de propriétés permet de satisfaire un critère de couverture définit dans la partie 9.4.5 de l’ISO 26262-6.

2. TRAVAUX CONNEXES

Il existe, dans la littérature plusieurs techniques de test basé sur les modèles (Model Based Testing MBT) [17],[32],[31].

TELNET Innovation Labs uitlise les modèles sous l’environnement Matlab/Simulink. Pour cela, nous limitons notre étude sur les outils qui permettent la génération des entrées de test à partir des modèles matlab/simulink.

Les outils commerciaux qui permettent la génération automatique des entrées de test à partir des modèles SL/SF sont : Reactis de “Reactive Systems Inc.“[24], “Embedded Tester from BTC” [9], REDIRECT [26] et Simulink Design Verifier “SLDV” [20].

L’outil “Reactis tester” utilise une combinaison entre le test aléatoire et la simulation guidée. Concernant l’outil “Embedded Tester from BTC”, les modèles SL/SF en entrée doivent être transformé en code C en utilisant “TargetLink”, par la suite les entrées de test sont générés à travers l’analyse de ce code C. L’outil REDIRECT utilse une combinaison de (a) test aléatoire, (b) DART (Directed Automated Random testing) [15] et le test concolique hybride [19]. L’outil Simulink Design Verifier (SLDV) est un outil intégré dans Matlab/-Simulink fournissant un accés à trois fonctionnalités : La génération des entrées de test, la vérification des propriétés et la détection des erreurs de conception (Débordement d’entier, division par zéro). La fonctionnalité de génération des entrées de test est basée sur le test aléatoire [12].

Parmi les outils disponibles qui permettent la génération des entrées de test à partir des modèles SL/SF en utilisant le model checking sont : AutomotGen [14], la plateforme V & V Diversity[12], SmartTestGen [22] et SAL[11].

L’outil “SAL” est un environnement de spécification et d’analyse de systèmes concurrents spécifiés sous forme d’automates. Il propose une série d’outils qui permettent d’analyser les automates. Sal-atg “Sal- Automated Test Generator“[16] est l’un des outils présentés par l’environnement SAL. Il utilise le model checking symbolique SMC [21], le model checking borné BMC [7] et le infi-borné model checker infBMC qui utilise le solveur SMT Yices [13] afin de générer des entrées de test et prouver l’inaccessibilité ou l’accessibilité de certains objectifs de couverture “coverage goals”. Le modèle matlab Simulink/Stateflow est transformé en spécification SAL interprétable par le model checker. Le modèle SAL est instrumenté par les “trap variables” qui représentent les “coverage goals”. L’accessibilité de la trap variables implique l’accessibilité de l’élément du modèle. Ainsi, les traces accessibles deviennent des entrées de test. Ceci n’assure que la couverture des transitions et des états.

L’outil “AutoMOTGen” est un outil de génération automatique des entrées de test à partir des modèles matlab SL/SF. Il est appliqué dans le test des contrôleurs d’automobile.

L'approche de génération des entrées de test est basée sur le "model checking". L'implémentation actuelle du AutoMOT-Gen utilise la spécification SAL comme une représentation intermédiaire et utilise l'outil "Sal-atg" afin de générer les données de test et prouver l'inaccessibilité de certaines couvertures d'objectif. Les critères de couverture supportés par cet outil sont la couverture des états et la couverture des transitions. L'outil prend en entrée trois éléments : le modèle matlab SL/SF, les besoins de haut niveau et la spécification de test qui contient les objectifs de test. L'architecture de l'outil AutoMotGen est basée sur la transformation du modèle matlab SL/SF en spécification SAL interprétable par le "model checker" et un "générateur de propriétés" qui permet de générer des propriétés en logique temporelle linéaire LTL [23]. Par la suite la spécification SAL du modèle et la propriété LTL sont récupérées par le "model checker" afin de générer l'ensemble des entrées de test.

L'outil "SmartTestGen" est un outil de génération des entrées de test qui intègre plusieurs moteurs de génération tels que : le "model checking" en utilisant l'outil Sal-atg, le test aléatoire, le "solveur" de contrainte local et la couverture basée sur des heuristiques [22].

La palteforme "V & V Diversity" est composée de deux outils : Diversity-TG et Diversity-MC.

L'outil Diversity-TG permet de générer des entrées de test numériques récupérées à partir de l'exécution symbolique sur le modèle issu de la transformation du modèle initial matlab SL/SF en une représentation interne à Diversity. Cet outil utilise l'exécution symbolique basée sur le model checking afin de générer l'ensemble des tests.

Concernant l'outil "Diversity-MC", il permet de générer un arbre du modèle et de vérifier par la suite les propriétés introduites par l'utilisateur en fixant les entrées de test qui permettent l'exécution des chemins identifiés au niveau de l'arbre.

La plupart de ces outils ne sont pas commercialisables, sauf l'outil SAL-ATG qui permet de générer un ensemble des entrées de test à partir de la spécification SAL du modèle interprétable par cet outil. Par conséquent, l'utilisation de cet outil nous incite à transformer chaque modèle en spécification SAL. Le travail présenté dans [33] donne les limitations de la méthode suivie par Sal-ATG. Les outils cités au-dessus génèrent un ensemble des entrées de test selon leur propre modèle d'entrée, et n'acceptent pas d'autre langage de modélisation en entrée. Cependant, [27] montre que ce processus de transformation implique l'augmentation du coût du projet en raison de la nécessité de développer un outil qui transforme chaque langage de modélisation en un langage de spécification.

Ainsi, nous remarquons que l'ensemble des approches ci-dessus nécessitent une transformation du modèle SL/SF en un modèle équivalent. Notre approche "MB-ATG" (Model Based-Automatic Test Case Generation) a pour objectif de réaliser des traitements homogènes, c'est-à-dire l'ajout de propriétés sur le modèle lui-même sans aucune transformation. L'expression du modèle et des propriétés se fait dans un même langage de modélisation. Les entrées de test obtenus en sortie doivent satisfaire les critères de couverture définis dans le tableau 1.

3. DESCRIPTION DE L'APPROCHE MB-ATG

3.1 Exploitation et principe de "MB-ATG"

La figure 1 montre l'exploitation de "MB-ATG" dans le processus général de vérification du modèle par rapport à son implémentation.

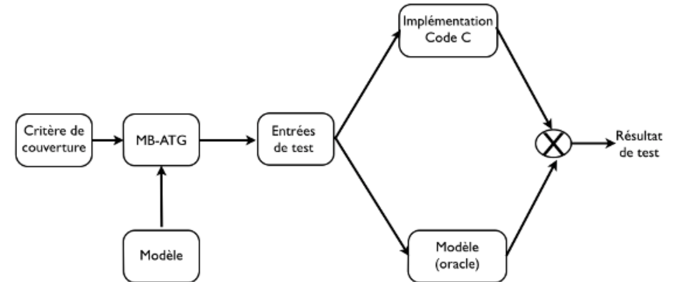

Figure 1: Exploitation de l'approche "MB-ATG"

Notre approche prend en entrée le modèle Matlab/Simulink et un critère de couverture, et génère en sortie, un ensemble des entées de test qui sera utilisé pour réaliser le test unitaire de l'implémentation. Dans le cadre des systèmes traités chez TELNET Innovation Labs, la construction se fait d'une manière incrémentale. Par conséquent, le test unitaire se fait d'une manière imbriquée. Cette gestion limite la taille des systèmes sous test. Le modèle joue le rôle d'un oracle. Les résultats fournis par ce dernier sont comparés à ceux générés par l'implémentation afin d'argumenter la conformité entre l'implémentation et la conception.

La figure 2 présente le principe de l'approche "MB-ATG".

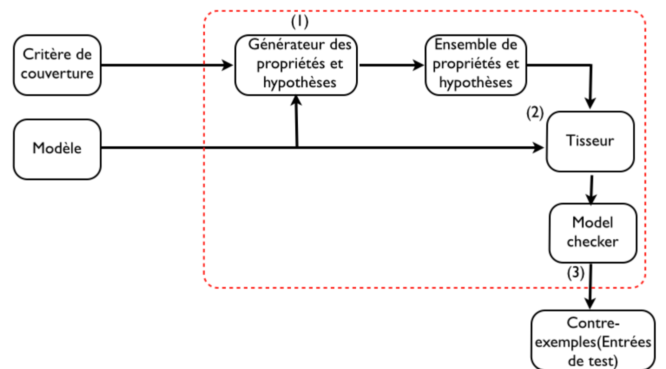

Figure 2: Principe de l'approche "MB-ATG"

Ce principe est composé de trois étapes : (1) Transformation des critères de couverture structurelle et comportementale en un ensemble de propriétés et hypothèses. (2) Tissage des propriétés et des hypothèses sur le modèle Simulink. (3) Vérification des propriétés par le *model checker* Prover plug-in. Le *model checker* vérifie les propriétés contraintes par les hypothèses et génère un contre-exemple (qui dénote la

violation de la propriété). Le contre-exemple est considéré comme une entrée de test.

3.2 Définition des propriétés et des hypothèses selon un critère de couverture

Cette phase présente l'étape(1) du principe de l'approche "MB-ATG". Elle prend en entrée un modèle et la définition d'un critère de couverture de test et génère un ensemble de propriétés. Chaque élément de cet ensemble correspond à une entrée de test. Nous démontrons dans cet article qu'un ensemble de propriétés peut générer un ensemble d'entrées de test assurant un critère de couverture. La mise en oeuvre de cette généraion ne fait pas l'objet de cet article. Les propriétés et les hypothèses sont exprimées en utilisant le même langage utilisé dans la conception du modèle. La conception de la propriété ψ (respectivement hypothèse H) est appelée observateur de propriété P (respectivement Hypothèse A). Ces observateurs, comme le modèle M sont mises en oeuvre avec les opérateurs Simulink. Ceux-ci sont appelés "Proof Objective" et "Assumption" , et ils sont accessibles par la bibliothèque SLDV. L'opérateur "Proof Objective" (respectivement Assumption) est identifiée par la lettre P (respectivement A).

3.3 Tissage des propriétés et des hypothèses sur le modèle Matlab/Simulink

La figure 3 décrit les étapes (2) nécessaires au tissage des propriétés et hypothèses sur le modèle.

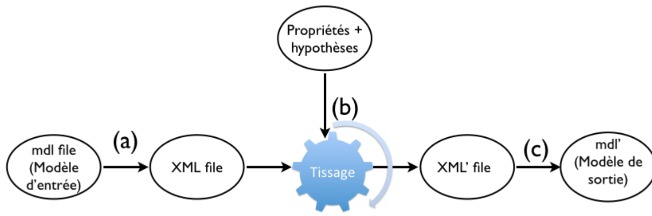

Figure 3: Implémentation du processus du tissage

Cette implémentation comporte trois parties principales : a) Transformation du fichier mdl en format XML b) Modification du fichier XML en tissant les propriétés et les hypothèses, et c) Transformation du nouveau fichier XML' en fichier mdl'. Les parties a) et c) sont assurées par l'outil $SimEx^{©}$ [1] qui permet la transformation réciproque mdl, xml. La partie b) est développée avec Jdom2/SAX API [2].

3.4 Le model checker Prover plug-in

Prover plug-in est commercialisé par Prover Technology [30]. Ce model checker est intégré à l'outil SLDV. La transformation des modèles Simulink vers le modèle d'entrée du model checker est automatique. Il présente le moteur de l'étape (3) de notre approche "MB-ATG". Il permet la génération des contre-exemples qui seront considérés comme un ensemble des entrées de test.

3.4.1 Principe de fonctionnement de prover plug-in

Le modèle Simulink est automatiquement transformé en un modèle TECLA interprétable par Prover Plug-In. Comme la plupart des model checkers, il peut renvoyer trois types de réponses. Si l'analyse formelle ne se termine pas, un message est renvoyé à l'utilisateur ("Manque de mémoire"). Si l'analyse se termine, alors

- si la propriété est valide, un message est renvoyé à l'utilisateur ("La propriété est valide"),
- si la propriété est violée, Prover Plug-In génère un contre-exemple.

Un contre-exemple est une exécution amenant le modèle dans un état violant la propriété P tout en respectant l'hypothèse H. C'est-à-dire, la sortie de l'observateur d'hypothèses est évaluée à vrai [8].

3.4.2 Vérification des propriétés

La technique du model checking permet de vérifier formellement des propriétés sur un système de transition. Selon [8], Prover Plug-in permet de vérifier des propriétés de sûreté sur des systèmes de transition [18]. Ces propriétés sont exprimées à l'aide des formules propositionnelle. La vérification des propriétés de sûreté consiste à vérifier que les propriétés sont des tautologies. Cette vérification consiste à déterminer si une formule propositionnelle donnée ψ composée d'un ensemble de variables p_i est vraie pour toutes les affectations des valeurs de p_i à vrai. D'après [28] le model checker Prover plug-in est basé sur la méthode de saturation Stålmarck [29] permettant de résoudre des problèmes de satisfiabilité.

Dans cet article, nous cherchons à montrer qu'un ensemble de propriétés fournit des contre-exemples permettant de satisfaire un critère de couverture. Afin de réaliser cette démonstration, nous avons besoin de présenter le principe de vérification des propriétés de sûreté par la méthode Stålmarck.

Cette méthode est basée sur un algorithme qui traite des formules exprimées à l'aide des cinq connecteurs $\wedge$, $\vee$, $\neg$, $\implies$ et $\iff$. Chaque formule est transformée en une formule α ne contenant que les connecteurs $\neg$ et $\implies$. La formule α peut être constituée de sous-formules. Chaque sous-formule P $\implies$ Q est remplacée par une variable v. Le triplet (v,p,q) est une représentation de la sous-formule de α où p et q sont des instances respectives des variables P et Q. Afin de transformer une formule avec les connecteurs $\neg$ et $\implies$ exclusivement, les transformations suivantes sont appliquées d'une façon répétitive :

$A \vee B$	$\neg A \implies B$
$A \wedge B$	$\neg(A \implies \neg B)$
$\neg\neg A$	A
$\neg A$	$A \implies$ Faux

Par conséquent, une formule en logique propositionnelle peut être exprimée sous la forme d'une conjonction de triplets. Par exemple, la formule $(A \wedge B) \implies C$ peut être tansformée en $\neg(A \implies \neg B) \implies C$. La représentation des triplets est la suivante :

(b',b,0)	$\neg B$
(d,a,b')	$A \implies \neg B$
(d',d,0)	$\neg(A \implies \neg B)$
(f,d',c)	$\neg(A \implies \neg B) \implies C$

Dans ce cas, a,b et c sont les représentations des variables A,B et C, tandis que, d,d',b' et f représente des sous-formules qui servent à lier les différents triplets. La variable f contient la totalité de la formule. Afin de prouver la validité de la formule, nous supposons que la formule est fausse et nous essayons de trouver une contradiction. Une contradiction est un triplet terminal (i.e un triplet est contradictoire en soi). Ci dessous les triplets terminaux possibles :

(0,y,1) (1,1,0) (0,0,x)

Afin de générer à partir d'un ensemble de triplets, un ensemble qui contient un triplet terminal, Stålmarck présente une application des règles simples (règles de propagation) ou une règle de branchement (*dilemma rule*).

Les règles simples sont dérivées à partir de la table de vérité de la relation d'implication (*implies*). Par exemple, si un ensemble de triplet contient le triplet (x,0,z), la variable x doit être toujours à "vrai". Par conséquent x peut être remplacée par 1. Ci-dessous la liste de toutes les règles simples :

$$Notation = \frac{triplet}{variable/substitution} \quad (1)$$

$$(R_1) = \frac{(0,y,z)}{y/1z/0} \quad (2) \qquad (R_2) = \frac{(x,y,1)}{x/1} \quad (3)$$

$$(R_3) = \frac{(x,0,z)}{x/1} \quad (4) \qquad (R_4) = \frac{(x,1,z)}{x/z} \quad (5)$$

$$(R_5) = \frac{(x,y,0)}{x/NOTy} \quad (6) \qquad (R_6) = \frac{(x,x,z)}{x/1} \quad (7)$$

$$(R_7) = \frac{(x,y,y)}{x/1} \quad (8)$$

Nous prenons comme exemple la formule $\alpha \iff$ p $\implies$ (q $\implies$ p) peut s'écrire sous forme de triplets (b1,q,p),(b2,p,b1) où b2 représente la formule α.

Stålmarck définit alors des règles de propagation sur ces triplets. Par exemple, p $\implies$ q est fausse si et seulement si p est vraie et q est fausse donne la règle

$$(R_1) = \frac{(0,p,q)}{p/1q/0}$$

: si l'on a le triplet (0,p,q) alors on peut remplacer p par 1 et q par 0 dans les autres triplets. Le but de ces règles est d'obenir un triplet terminal (contradictoire) : (1,1,0) est un triplet terminal par exemple (les autres triplets terminaux sont (0,x,1) et (0,0,x)).

Dans notre exemple, la seule règle R_1 suffit à prouver que α est valide (c'est-à-dire $\neg\alpha$ inconsistante). Posons b2 est fausse (ce qui correspond à $\neg\alpha$). Cela se traduit par le remplacement de b2 par 0 dans les triplets. On peut donc appliquer la règle R1 sur le triplet (0,p,b1), ce qui donne :

$$(b1,q,p)[p/1,b1/0] = (0,q,1)[p/1,b1/0]$$

Or (0,q,1) est terminal. La formule $\neg\alpha$ est donc inconsistante. α est donc valide.

Certaines de ces règles n'assignent pas de variables, mais expriment l'équivalence de deux littéraux (règles R_4 et R_5) L'application des règles simples ne garantit pas toujours de trouver un triplet terminal. C'est pourquoi une règle de branchement est introduite : Stålmarck l'appelle *dilemma rule*. Il s'agit de l'expansion de Shannon appliquée à la déduction.

4. MODÈLES, PROPRIÉTÉS ET HYPOTHÈSES

4.1 Modèles d'études

Dans cette section nous appliquons les différentes étapes de l'approche "MB-ATG" détaillées dans la section 3.1, sur des modèles Matlab/Simulink. Les figures 4 et 5 présentent respectivement une fonction logique *AND* et une composition séquentielle des deux opérateurs logiques *AND* et *OR*.

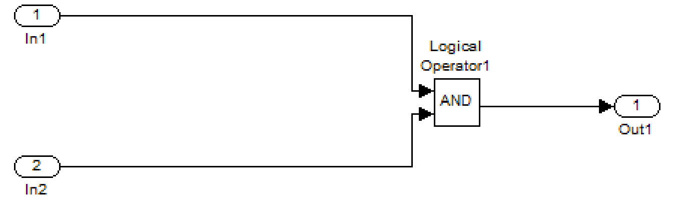

Figure 4: Fonction élémentaire AND

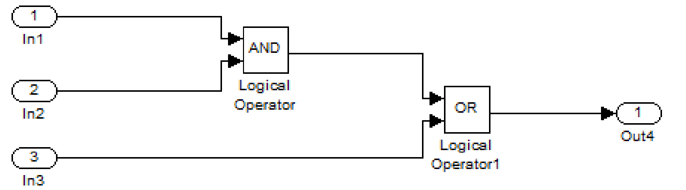

Figure 5: Composition séquentielle des deux fonctions AND et OR

Nous allons appliquer les critères de couverture définis au niveau ASIL D. Tels que, la Couverture des instructions, couverture des branches et le critère MC/DC. Nous allons nous focaliser sur le critère MC/DC, car il permet la couverture des conditions, décisions et des sous-décisions. Suite à l'analyse du fonctionnement du model checker, un seul contre-exemple est généré suite à la violation de chaque propriété. Par conséquent, le nombre de propriétés doit être égal au

nombre des entrées de test nécessaire à la couverture d'un critère de couverture donné. Dans la suite, "Vrai"(true) et "Faux"(False) sont remplacés respectivement par 1 et 0.

4.2 Propriétés et hypothèses

Le tableau 2 présente les propriétés et les hypothèses à tisser sur le premier modèle décrit dans la figure 4 afin d'assurer la couverture du critère MC/DC.

Table 2: Propriétés et hypothèses relatives au modèle AND

Propriétés et hypothèses
P1 : NOT(A ∧ B)
P2 : (A ∧ B)
P3 : ¬(A)
H : (A ∨ B)

Les propriétés P1 et P2 assure la couverture des Conditions/Décisions. Elles correspondent respectivement à la couverture de la décision 1 et 0. Ainsi, les propriétés P2 et P3 montrent le renforcement du critère de couverture MC/DC, relatif à la modification de condition qui affecte independamment la décision 0. L'hypothèse H permet de contraindre les propriétés et d'éliminer les entrées de test inutiles. Dans le cas de la fonction logique AND, l'entrée de test (0,0) ne doit pas être pris en compte. Le problème avec cette entrée de test, c'est qu'elle ne montre pas l'influence des conditions sur la décision finale : "0".

Le tableau 3 présente les propriétés et les hypothèses à tisser sur le deuxième modèle représenté dans la figure 5.

Table 3: Propriétés et hypothèses relatives au modèle à composition séquentielle AND et OR

Propriétés et hypothèses
P1 : NOT((A ∧ B) OR C)
P2 : (A ∧ B) OR C
P3 : NOT(A ∧ B)
P4 : (A ∧ B)
P5 : ¬(C)
H1 : (A ∨ B)
H2 : $(\overline{A \wedge B})$

Les propriétés P1 et P2 assure la couverture du critère Conditions/Décisions, qui correspondent respectivement à la décision 1 et 0. Les propriétés P3 et P4 assure la couverture de la sous-décision (A ∧ B) respectivement à 1 et 0. Ainsi, la propriété P5 est relative à la couverture de la condition "C". Concernant les hypothèses H1 et H2, elles permettent de contraindre ces hypothèses et d'éliminer les entrées de test inutile telles que (0,0,0) et (1,1,1).

4.3 Tissage des propriétés et des hypothèses

Les figures 6 et 7 décrivent respectivement le tissage des propriétés et des hypothèses sur les modèles présentés dans 4 et 5 en suivant le processus du tissage présenté dans la section 3.3.

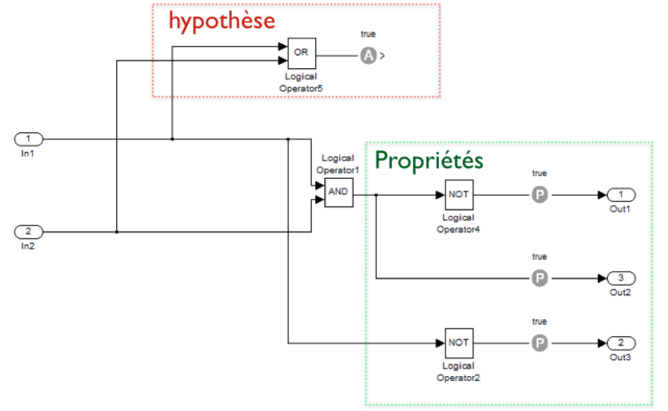

Figure 6: Transformation du modèle AND

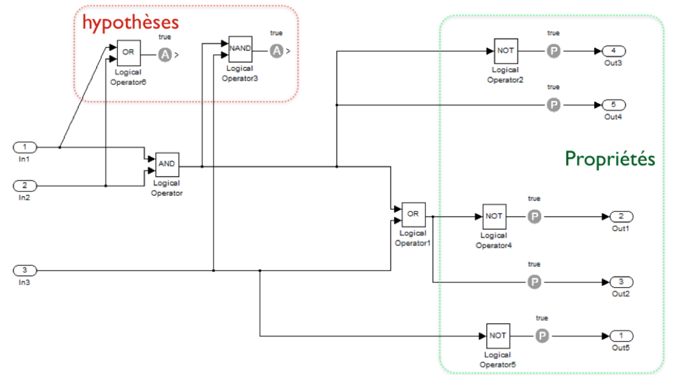

Figure 7: Transformation du modèle à composition AND et OR

Dans la section suivante, nous allons expliquer la phase de génération des contre-exemples relative aux deux modèles présentés ci-dessus. Nous allons montrer que les contre-exemples sont équivalents aux entrées de test en assurant le critère de couverture MC/DC.

5. PREUVE DE LA COUVERTURE

La phase de vérification est basée sur le model checker Prover plug-in. Au cours de cette phase, Prover plug-in génère un contre-exemple seulement si le résultat est insatisfiable. Si nous voulons garantir une équivalence entre les contre-exemples et les entrées de test, chaque entrée de test doit correspondre à la négation d'une propriété. Le modèle checker prend en entrée le système états/transitions représentant les différents états du modèle Matlab/Simulink initial et les propriétés à vérifier qui sont contraintes par les hypothèses. Afin de générer des contre-exemples, ce dernier utilise une extension de la méthode Stålmarck [5]. Une relation initiale d'équivalence est construite à partir des hypothèses, et les formules propositionnelles présentant les propriétés sont affectées à la classe d'équivalence "*Vraie*", ce qui donne $H_i \iff P_j$ avec H_i=1. Par la suite, l'algorithme de saturation, est appliqué à la relation d'équivalence, avec un degré de saturation donné, et le résultat est calculé de la manière

suivante : Si le résultat de saturation est contradictoire, un résultat (Unsat) est retourné et un contre-exemple est généré. Si la classe d'équivalence est non contradictoire, et la formule est en classe d'équivalence "*Vraie*" alors le résultat obtenu est **Sat**.

L'algorithme global de saturation fonctionne comme suit :

1. La négation de la formule est converti en triplets
2. L'algorithme 0-Saturation est appliqué. Si une contradition est obtenue, alors Fin.
3. Sinon, l'algorithme 1-Saturation est appliqué : pour chaque variable une *Dilemma rule* est utilisée, afin de déduire des nouvelles équivalences. Si une contradiction est obtenue, alors Fin.
4. Sinon, continuer à appliquer des niveaux supplémentaire de saturation jusqu'à obtenir une contradiction.
5. Sinon, retourner la valeur "Vrai" (SAT)

Á ce niveau, nous allons appliquer les deux premières étapes de l'algorithme global de saturation.

5.1 Cas d'un modèle élémentaire

La figure 8 représente le système états/transitions du modèle élémentaire de la figure 4. Cet automate présente tous les états possibles du système à tester. Ces derniers sont obtenus en variant une seule condition (a ou b) et en fixant l'autre.

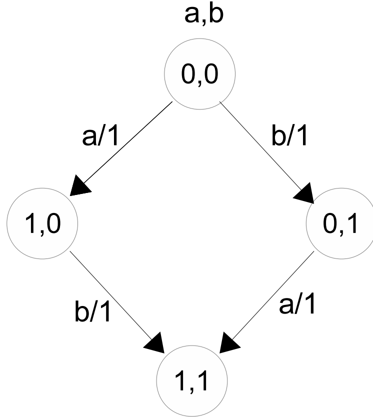

Figure 8: Système états/transitions de la fonction élémentaire AND

Le tableau 4 présente la première étape de l'algorithme global de saturation.

Table 4: Transformation des propriétés et hypothèses relatives au modèle AND

Propriétés	Transformation en négation
P1 : ¬(A ∧B)	A $\Longrightarrow$ ¬B
P2 : (A ∧ B)	(A $\Longrightarrow$ ¬B) $\Longrightarrow$ 0
P3 : ¬(A)	A $\Longrightarrow$ False

L'application de l'algorithme 0-saturation sur les propriétés décrites dans le tableau 4, consiste à appliquer les règles simples de propagation pour chaque relation d'équivalence $H_i \iff P_j$ avec H_i=1, afin de trouver le triplet terminal qui représente une contradiction. Si une contradiction est retrouvée, le processus de vérification s'arrête. Le triplet terminal représente le contre-exemple. Ce traitement se fait pour chaque état du système états/transitions représentant le modèle. Pour les états où H_i= 0, sont à éliminer.

Dans ce cas, nous avons trois propriétés à vérifier pour chacun des états représentés par le système états/transition décrit dans la figure 8.

Pour la première propriété P1, la formule propositionnelle équivalente est : H1 $\iff$ A $\Longrightarrow$ ¬B. Nous cherchons à trouver l'état qui représente une contradiction pour 1 $\iff$ A $\Longrightarrow$ ¬B. Les triplets qui représentent cette formule sont (h1,a,b') (b',b,0), où h1 représente la formule globale et a et b représentent respectivement des instanciations des variables A et B. La variable b' représente la sous-formule ¬B.

Table 5: Propagation des règles simples pour P1

État	Numéro de la règle	Propagation de la règle	Résultat
(0,0)	–	–	H_i= 0
(0,1)	R_3	1= 0 $\Longrightarrow$ ¬1 ; 1=0 $\Longrightarrow$ 0	Pas de contradiction
(1,0)	R_6	1= 1 $\Longrightarrow$ ¬0 ; 1=1 $\Longrightarrow$ 1	Pas de contradiction
(1,1)	R_6	1= 1 $\Longrightarrow$ ¬1 ; 1=1 $\Longrightarrow$ 0	Contradiction

D'aprés le tableau 5, l'état (1,1) entraîne une contradiction en appliquant la règle R_6 sur (1,1,b'), ce qui donne :

$$\begin{array}{l} h1 \iff f \Longrightarrow 0 \quad 1 \iff 1 \Longrightarrow 1 \\ b' \iff b \Longrightarrow 0 \quad \color{red}{1 \iff 1 \Longrightarrow 0} \end{array}$$

Par conséquent, le triplet terminal est : (h1,**a**,b')(b',**b**,0)-(1,**1**,1)(1,**1**,0) et le contre-exemple généré est : (a,b)(1,1).

Pour la deuxième propriété P2, la formule propositionnelle équivalente est : H1 $\iff$ ¬(A $\Longrightarrow$ ¬B) ce qui nous donne H1 $\iff$ (A $\Longrightarrow$ ¬B) $\Longrightarrow$ false. Les triplets qui représentent cette formule sont (h1,f,0) (f,a,b') (b',b,0), où h1 représente la formule globale et a et b représentent respectivement des instanciations des variables A et B. Concernant f, elle représente la sous-formule (A $\Longrightarrow$ ¬B).

Table 6: Propagation des règles simples pour P2

État	Numéro de la règle	Propagation de la règle	Résultat
(0,0)	–	–	$H_i = 0$
(0,1)	R_5	$1 = (0 \implies \neg 1) \implies$ false ; $1 = (0 \implies 0) \implies$ false	Contradiction
(1,0)	–	–	–
(1,1)	–	–	–

Selon le tableau 6, l'état (0,1) entraîne une contradiction en appliquant la règle R_5 sur (1,f,0), ce qui donne :

$$\begin{array}{lllllllll} h1 & \iff & f & \implies & 0 & 1 & \iff & 0 & \implies 0 \\ f & \iff & a & \implies & b' & \color{red}{0} & \color{red}{\iff} & \color{red}{0} & \color{red}{\implies 0} \\ b' & \iff & b & \implies & 0 & 0 & \iff & 1 & \implies 0 \end{array}$$

Par conséquent, le triplet terminal est : (h1,f,b')(f,**a**,b')(b',**b**,0)-(1,0,0)(0,0,0)(0,**1**,0) et le contre-exemple généré est : (a,b)(0,1).

Pour la troisième propriété P3, la formule propositionnelle équivalente est : H1 $\iff$ (A $\implies$ false). Le triplet qui représente cette formule est (h1,a,0) où h1 représente la formule globale et a représente respectivement une instanciation de la variable A.

Table 7: Propagation des règles simples pour P3

État	Numéro de la règle	Propagation de la règle	Résultat
(0,0)	–	–	$H_i = 0$
(0,1)	R_3	$1 = (0 \implies$ false)	Pas de contradiction
(1,0)	R_6	$1 = (1 \implies$ false)	Contradiction
(1,1)	–	–	–

Selon le tableau 7, l'état (0,1) entraîne une contradiction en appliquant la règle R_5 sur (1,f,0), ce qui donne :

$$h1 \iff a \implies 0 \quad \color{red}{1 \iff 1 \implies 0}$$

Par conséquent, le triplet terminal est : (h1,**a**,0) (1,**1**,0) et le contre-exemple généré est : (a,b)(1,0).
Le tableau 8 présente la liste des contre-exemples générés pour ce modèle, suite à la vérification du model checker Prover plug-in des propriétés. Ainsi, nous avons vérifié que ces contre-exemples sont équivalents aux entrées de test assurant le critère de couverture MC/DC.

Table 8: Contre-exemples relatifs aux Propriétés et hypothèses (modèle AND)

Propriétés et hypothèses	Contre-exemples
P1 : $\neg(A \wedge B)$	(1,1)
P2 : $(A \wedge B)$	(0,1)
P3 : $\neg(A)$	(1,0)

5.2 Cas d'une composition séquentielle

La figure 9 représente le système états/transitions du modèle élémentaire de la figure 5. Ce système présente tous les états possibles du modèle à tester. Ces derniers sont obtenus en variant une seule condition (a ou b ou c) et en fixant les deux autres.

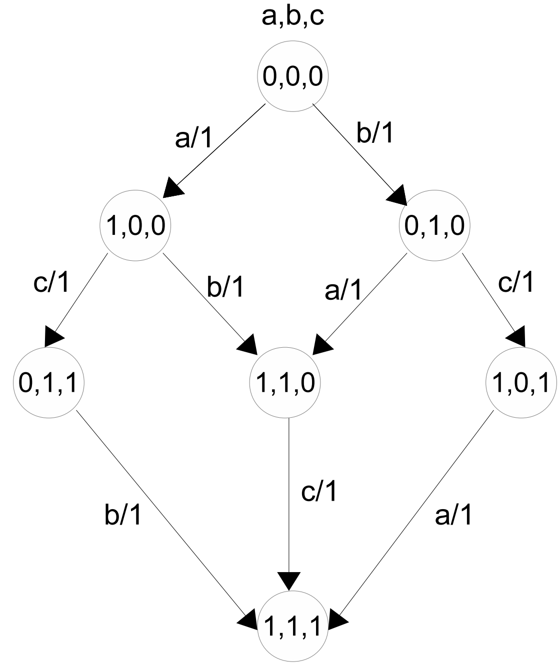

Figure 9: Système états/transitions de la composition séquentielle AND-OR

Le tableau 9 présente la première étape de l'algorithme 0-saturation.

Table 9: Propriétés et hypothèses relatives au modèle à composition AND et OR

Propriétés	Transformation en négation
P1' : $\neg((A \wedge B) \vee C)$	$(\neg (A \implies \neg B) \implies C) \implies 0$
P2' : $(A \wedge B) \vee C$	$(A \implies \neg B) \implies C$
P3' : $\neg(A \wedge B)$	$A \implies \neg B$
P4' : $(A \wedge B)$	$(A \implies \neg B) \implies 0$
P5' : $\neg(C)$	$C \implies$ False

Nous passons à la vérification des propriétés de la même manière que le modèle élémentaire.

Table 10: Propagation des règles simples pour P1'

État	Numéro de la règle	Propagation de la règle	Résultat
(0,0,0)	–	–	$H_i = 0$
(0,1,0)	R_5	$1 \wedge 1 \iff (\neg(0 \implies \neg 1) \implies 0) \implies 0$; $(\neg(0 \implies 0) \implies 0) \implies 0$	Pas de contradiction
(1,0,0)	R_5	$1 \wedge 1 \iff (\neg(1 \implies \neg 0) \implies 0) \implies 0$; $(\neg(1 \implies 0) \implies 0) \implies 0$	Pas de contradiction
(1,0,1)	R_5	$1 \wedge 1 \iff (\neg(1 \implies \neg 0) \implies 1) \implies 0$; $(\neg(1 \implies 1) \implies 1) \implies 0$	Pas de contradiction
(1,1,0)	R_5	$1 \wedge 1 \iff (\neg(1 \implies \neg 1) \implies 0) \implies 0$; $(\neg(1 \implies 0) \implies 0) \implies 0$	Pas de contradiction
(0,1,1)	R_5	$1 \wedge 1 \iff (\neg(0 \implies \neg 1) \implies 1) \implies 0$; $(\neg(0 \implies 0) \implies 1) \implies 0$	Contradiction
(1,1,1)	–	–	–

Pour la première propriété P1', la formule propositionnelle équivalente est : H1∧H2 $\iff$ ¬(¬(A $\implies$ ¬B) $\implies$ C), ce qui nous donne H1∧H2 $\iff$ (¬ (A $\implies$ ¬B) $\implies$ C) $\implies$ false.

Les triplets qui représentent cette formule sont (b',b,0) (d,a,b') (d',d,0) (f,d',c) (h,f,0), où h représente la formule globale et a, b et c représentent respectivement les instanciations des variables A, B et C. Concernant d,d',f, b' représentent des sous-formules qui servent comme des liens entre les triplets.

Selon la vérification de la propriété P1' présentée dans le tableau 10, l'état (0,1,1) engendre une contradiction en appliquant la règle R_5 sur (1,f,0), ce qui donne :

$$\begin{array}{lllll} h \iff f \implies 0 & & 1 \iff 0 \implies 0 \\ f \iff d' \implies c & & 0 \iff 0 \implies 1 \\ d' \iff d \implies 0 & & 0 \iff 1 \implies 0 \\ d \iff a \implies b' & & 1 \iff 0 \implies 0 \\ b' \iff b \implies 0 & & 0 \iff 1 \implies 0 \end{array}$$

Par conséquent, le triplet terminal est : (h,f,b')(f,d',**c**)(d',d,0)-(d,**a**,b')(b',**b**,0) :(1,0,0)(0,0,**1**)(0,1,0)(1,**0**,0)(0,**1**,0) et le contre-exemple généré est : (a,b,c)(0,1,1).

Table 11: Propagation des règles simples pour P2'

État	Numéro de la règle	Propagation de la règle	Résultat
(0,0,0)	–	–	$H_i = 0$
(0,1,0)	R_5	$1 \wedge 1 \iff (0 \implies \neg 1) \implies 0$;$1 \iff (1 \implies 1) \implies 0$	Contradiction)
(1,0,0)	R_5	$1 \wedge 1 \iff (1 \implies \neg 0) \implies 0$;$1 \iff (1 \implies 1) \implies 0$	Contradiction)
(1,1,0)	–	–	–
(0,1,1)	–	–	–
(1,0,1)	–	–	–
(1,1,1)	–	–	–

Pour la deuxième propriété P2', la formule propositionnelle équivalente est : P2' = H1∧H2 $\iff$ (A $\implies$ ¬B) $\implies$ C. Les triplets qui représentent cette formule sont (b',b,0)(d,a,b') (h,d,c), où h représente la formule globale et a, b et c représentent respectivement les intantiations des variables A, B et C. Concernant d et b', elles représentent des sous-formules qui servent comme des connections entre les triplets.

Selon le tableau 11, l'état (1,0,0) engendre une contradiction en appliquant la règle R_5 sur (1,f,0), ce qui donne :

$$\begin{array}{lllll} h \iff f \implies 0 & & 1 \iff 0 \implies 0 \\ f \iff a \implies b' & & 0 \iff 1 \implies 0 \\ b' \iff b \implies 0 & & 0 \iff 0 \implies 0 \end{array}$$

Par conséquent, le triplet terminal est : (h,f,c)(f,**a**,b')(b',**b**,0) : (1,0,0)(0,**1**,0)(0,**0**,0) et le contre-exemple généré est : (a,b,c)(1,0,0).

Table 12: Propagation des règles simples pour P3'

État	Numéro de la règle	Propagation de la règle	Résultat
(0,0,0)	–	–	$H_i = 0$
(0,1,0)	R_5	$1 \iff (0 \implies \neg 1) \implies 0$; $1 \iff (0 \implies 0) \implies 0$	Contradiction
(1,0,0)	–	–	
(1,1,0)	–	–	–
(0,1,1)	–	–	–
(1,0,1)	–	–	–
(1,1,1)	–	–	–

Pour la troisième propriété P3', la formule propositionnelle équivalente est : P3' = H1 $\iff \neg(A \implies \neg B)$, ce qui nous donne : H1 $\iff (A \implies \neg B) \implies$ false.

Les triplets qui représentent cette formule sont (b',b,0)(f,a,b')(h,f,0), où h représente la formule globale et a et b représentent respectivement les intantiations des variables A et B. Concernant f et b', elles représentent respectivement les sous-formules : $(A \implies \neg B)$ et $\neg B$. La variable h contient la totalité de la formule.

Selon le tableau 12, l'état (0,1,0) entraîne une contradiction en appliquant la règle R_5 sur (1,f,0), ce qui donne :

$$\begin{array}{llll} h \iff f \implies 0 & & 1 \iff 0 \implies 0 \\ f \iff a \implies b' & & 0 \iff 0 \implies 0 \\ b' \iff b \implies 0 & & 0 \iff 1 \implies 0 \end{array}$$

Par conséquent, le triplet terminal est : (h,f,c)(f,**a**,b')(b',**b**,0) : (1,0,0)(0,**1**,0)(0,**0**,0) et le contre-exemple généré est : (a,b,c)(1,0,0).

Table 13: Propagation des règles simples pour P4'

État	Numéro de la règle	Propagation de la règle	Résultat
(0,0,0)	–	–	H_i= 0
(0,1,0)	R_6	$1 \iff (0 \implies \neg 1)$; $1 \iff (0 \implies 0)$	Pas de Contradiction
(1,0,0)	R_3	$1 \iff (1 \implies \neg 0)$; $1 \iff (1 \implies 1)$	Pas de contradiction
(1,1,0)	R_6	$1 \iff (1 \implies \neg 1)$; $1 \iff (1 \implies 0)$	Contradiction
(0,1,1)	–	–	–
(1,0,1)	–	–	–
(1,1,1)	–	–	–

Pour la quatrième propriété P4', la formule propositionnelle équivalente est : P4' = H1 $\iff \neg(A \implies \neg B)$, ce qui nous donne : H1 $\iff (A \implies \neg B) \implies$ false.

Les triplets qui représentent cette formule sont (b',b,0)(f,a,b')-(h,f,0), où h représente la formule globale et a et b représentent respectivement les instanciations des variables A et B. Concernant f et b', elles représentent respectivement les sous-formules : $(A \implies \neg B)$ et $\neg B$ et . La variable h contient la totalité de la formule.

Nous appliquons par la suite les règles simples de propagation sur l'ensemble des états du système états/transitions correspondant au deuxième modèle.

Selon le tableau 13, l'état (1,1,0) entraîne une contradiction en appliquant la règle R_6 sur (1,1,b'), ce qui donne :

$$\begin{array}{llll} h \iff a \implies b' & & 1 \iff 1 \implies 1 \\ b' \iff b \implies 0 & & 1 \iff 1 \implies 0 \end{array}$$

Par conséquent, le triplet terminal est : (h,**a**,b')(b',**b**,0) : (1,1,1)(1,**1**,0) et le contre-exemple généré est : (a,b,c)(1,1,0).

Table 14: Propagation des règles simples pour P5'

État	Numéro de la règle	Propagation de la règle	Résultat
(0,0,0)	–	–	H_i= 0
(0,1,0)	R_5	$1 \wedge 1 \iff (0 \implies 1)$	Pas de contradiction
(1,0,0)	R_5	$1 \wedge 1 \iff (0 \implies 0)$	Pas de contradiction
(1,1,0)	R_5	$1 \wedge 1 \iff (0 \implies 0)$	Pas de contradiction
(1,0,1)	R_6	$1 \wedge 1 \iff (1 \implies 0)$	Contradiction
(0,1,1)	–	–	–
(1,1,1)	–	–	–

Pour la cinquième propriété P5', la formule propositionnelle équivalente est : P5' = H1$\wedge$H2 $\iff$ C $\implies$ False

Les triplets qui représentent cette formule sont (h,c,0), où h représente la formule globale et c représente l'instanciation de la variable C.

Selon le tableau 14, l'état (1,0,1) entraîne une contradiction en appliquant la règle R_6 sur (1,c,0), ce qui donne :

$$h1 \iff c \implies 0 \quad 1 \iff 1 \implies 0$$

Par conséquent, le triplet terminal est : (h1,**c**,0) (1,**1**,0) et le contre-exemple généré est : (a,b,c)(1,0,1).

Le tableau 15 présente la liste des contre-exemples générés pour ce modèle, suite à la vérification du model checker Prover plug-in des propriétés. Ainsi, nous avons vérifié que ces contre-exemples sont équivalents aux entrées de test assurant le critère de couverture MC/DC.

Table 15: Contre-exemples relatifs aux Propriétés et hypothèses (composition AND et OR)

Propriétés et hypothèses	Contre-exemples
P1 : $\neg((A \wedge B) \vee C)$	(0,1,1)
P2 : $(A \wedge B) \vee C$	(1,0,0)
P3 : $\neg(A \wedge B)$	(1,1,0)
P4 : $(A \wedge B)$	(0,1,0)
P5 : $\neg(C)$	(1,0,1)

6. CONCLUSION ET TRAVAUX FUTURS

L'approche MB-ATG vise à automatiser le processus de test unitaire employé par Telnet Innovation Labs. Elle réalise la génération des entrées de test d'une manière imbriquée ce qui

permet la limitation de la taille du système à tester. Notre approche exploite le pouvoir des model checkers pour générer des contres exemples. Ces deniers sont provoqués de manière à correspondre à des entrées de test satisfaisant un critère de couverture structurel ou comportemental. Contrairement aux approches précédentes, notre proposition offre une solution homogène dans laquelle les propriétés et le modèle de l'application sont exprimés à l'aide du même langage, en l'occurrence Matlab/simulink. La démonstration qu'un ensemble de propriétés et d'hypothèses génère un ensemble de contre-exemples, ou plus particulièrement des entrées de test satisfaisant les différents critères de couvertures introduit par l'iso 26262-6 partie 9.4.5, repose sur la méthode stalmarck utilisée par le model checker SLDV. Nous visons, dans un futur proche, à générer automatiquement les propriétés et les hypothèses à partir d'un critère de couverture donné ; Et à étendre notre approche aux sous-modèles analogiques supportés par Matlab/simulink.

7. REFERENCES

[1] Itpower simex. available at : http ://www.itpower.de/.

[2] Jdom api documentation. available at : http ://www.jdom.org/downloads/docs.html, April 2013.

[3] Misra . available at : http ://www.misra.org.uk/, November 1994.

[4] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino *et al.* : An orchestrated survey on automated software test case generation. *Journal of Systems and Software*, 2013.

[5] G. Andersson, P. Bjesse, B. Cook et Z. Hanna : A proof engine approach to solving combinational design automation problems. *In Design Automation Conference, 2002. Proceedings. 39th*, p. 725–730. IEEE, 2002.

[6] M. Ben-Ari, A. Pnueli et Z. Manna : The temporal logic of branching time. *Acta informatica*, 20(3):207–226, 1983.

[7] A. Biere, A. Cimatti, E. Clarke et Y. Zhu : Symbolic model checking without bdds. *Tools and Algorithms for the Construction and Analysis of Systems*, p. 193–207, 1999.

[8] T. Bochot : *Vérification par model checking des commandes de vol : applicabilité industrielle et analyse de contre-exemples*. Thèse de doctorat, 2009.

[9] BTC-Embedded-Tester : http ://www.btc-es.de/.

[10] E. M. Clarke, O. Grumberg et D. A. Peled : *Model checking*. MIT press, 2000.

[11] L. De Moura, S. Owre, H. Ruess, J. Rushby, N. Shankar, M. Sorea et A. Tiwari : Sal 2. *In Computer Aided Verification*, p. 496–500. Springer, 2004.

[12] A. L. Diane Bahrami, Alain Faivre : Diversity-tg automatic test case generation from matlab/simulink models. *In Embedded real time software and systems, Toulouse, France*, 2012.

[13] B. Dutertre et L. De Moura : The yices smt solver. *Tool paper at http ://yices. csl. sri. com/tool-paper. pdf*, August 2006.

[14] A. Gadkari, A. Yeolekar, J. Suresh, S. Ramesh, S. Mohalik et K. Shashidhar : Automotgen : Automatic model oriented test generator for embedded control systems. *In Computer Aided Verification*, p. 204–208. Springer, 2008.

[15] P. Godefroid, N. Klarlund et K. Sen : Dart : directed automated random testing. *In ACM Sigplan Notices*, vol. 40, p. 213–223. ACM, 2005.

[16] G. Hamon, L. De Moura et J. Rushby : Automated test generation with sal. *CSL Technical Note*, 2005.

[17] A. Kull : *Model-Based Testing of Reactive Systems*. TUT Press, 2009.

[18] O. Kupferman et M. Y. Vardi : Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[19] R. Majumdar et K. Sen : Hybrid concolic testing. *In Software Engineering, 2007. ICSE 2007. 29th International Conference on*, p. 416–426. IEEE, 2007.

[20] I. Mathworks : Simullink design verifier 1 : User's guide. Rap. tech., 2012.

[21] K. L. McMillan : *Symbolic Model Checking*. Kluwer Academic Publishers Norwell, MA, USA, 1993.

[22] P. Peranandam, S. Raviram, M. Satpathy, A. Yeolekar, A. Gadkari et S. Ramesh : An integrated test generation tool for enhanced coverage of simulink/stateflow models. *In Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, p. 308–311. IEEE, 2012.

[23] A. Pnueli : The temporal logic of programs. *In Foundations of Computer Science*, p. 46–57. IEEE, 1977.

[24] Reactis : http ://www.reactive-systems.com/.

[25] N. Russ, G. Peter, R. Berlin et B. Ulmer : Lessons learned : on-board software test automation using ibm rational test realtime. *In Space Mission Challenges for Information Technology, 2006. SMC-IT 2006. Second IEEE International Conference on*, p. 1–pp. IEEE, 2006.

[26] M. Satpathy, A. Yeolekar et S. Ramesh : Randomized directed testing (redirect) for simulink/stateflow models. *In Proceedings of the 8th ACM international conference on Embedded software*, p. 217–226. ACM, 2008.

[27] B. Schlich et S. Kowalewski : Model checking c source code for embedded systems. *International journal on software tools for technology transfer*, 11(3):187–202, 2009.

[28] M. Sheeran : Prover plug-in documentation. 2000.

[29] M. Sheeran et G. Stålmarck : A tutorial on stålmarck's proof procedure for propositional logic. *In Formal Methods in Computer-Aided Design*, p. 82–99. Springer, 1998.

[30] P. Technology : http ://www.prover.com/.

[31] M. Utting et B. Legeard : *Practical model-based testing : a tools approach*. Morgan Kaufmann, 2007.

[32] M. Utting, A. Pretschner et B. Legeard : A taxonomy of model-based testing. 2006.

[33] R. Venkatesh, U. Shrotri, P. Darke et P. Bokil : Test generation for large automotive models. *In Industrial Technology (ICIT)*, p. 662–667. IEEE, 2012.