Génération automatique de tests à partir de modèles SysML pour la validation fonctionnelle de systèmes embarqués

Jonathan Lasalle

Université de Franche-Comté **UFR Sciences et Techniques**

École doctorale SPIM

Génération automatique de tests à partir de modèles SysML pour la validation fonctionnelle de systèmes embarqués

THÈSE

présentée et soutenue publiquement le 29 juin 2012

pour l'obtention du grade de

Docteur de l'Université de Franche-Comté

(Spécialité Informatique)

par

Jonathan Lasalle

Composition du jury

Président :	Mr Jacques Julliand, Professeur à l'Université de Franche-Comté
Directeurs :	Mr Fabrice Bouquet, Professeur à l'Université de Franche-Comté Mr Fabien Peureux, Maître de Conférences à l'Université de Franche-Comté
Rapporteurs :	Mr Jean-Michel Bruel, Professeur à l'Université de Toulouse Mr Pierre-Alain Müller, Professeur à l'Université de Haute-Alsace
Examinateur :	Mr Bernard Botella, Ingénieur au CEA/LIST

Département Informatique des Systèmes Complexes - FEMTO-ST - UMR CNRS 6174

Remerciements

En premier lieu, je souhaite remercier Fabrice Bouquet et Fabien Peureux, pour m'avoir fait l'honneur d'être mes encadrants durant cette thèse, et pour m'avoir soutenu et fait confiance tout au long de cette aventure. Je remercie vivement Pierre-Alain Müller et Jean-Michel Bruel d'avoir accepté d'être rapporteurs de mes travaux et de m'avoir transmis une partie de leurs connaissances à travers leurs remarques. Merci également à Bernard Botella pour son expertise en qualité d'examinateur. Enfin, un grand merci à Jacques Julliand pour avoir accepté de présider le jury. Merci à vous tous pour l'évaluation faite de mes travaux.

Je tiens également à remercier Emilie Oudot, Brice Wittmann ainsi que Frédéric Fondement pour leur collaboration, contribuant ainsi à l'obtention des résultats présentés dans ce mémoire. Merci à mes parents et à Isabelle Jacques pour m'avoir aidé à chasser les fautes d'orthographe de ce document. J'ai ici également une pensée pour Christelle Bloch, Pascal Chatonnay et Fabrice Bouquet qui m'ont offert l'opportunité d'effectuer un stage de M2 recherche au sein du DISC, m'ouvrant ainsi la porte du monde de la recherche.

Je remercie également tous les membres du département DISC pour leur accueil et pour leur sympathie à mon égard. Merci à Jean-Christophe Lapayre et Olga Kouchnarenko pour m'avoir successivement accueilli au sein du LIFC puis du DISC. Merci à Frédéric Dadeau de m'avoir intégré au comité d'organisation de l'école d'été TAROT 2012, me permettant ainsi de contribuer à sa réussite. Enfin, merci à Fabien Peureux, encadrant, collègue et désormais ami, pour ses leçons phylosophico-culturelles de toute beauté.

Je profite également de cette occasion pour remercier tous ceux qui m'ont fait confiance en m'offrant la possibilité d'effectuer des enseignements. Merci à Jean-Michel Hufflen pour sa confiance en m'ayant laissé seul maître à bord des TD de Scheme durant 5 années. Merci également à Françoise Greffier, Nicolas Janet, Christophe Lang, François Piat et aux 3 Fab' du bureau 425C.

Je voudrais remercier mes frères et soeurs ainsi que mes parents qui, de par leurs encouragements inconditionnels, m'ont permis d'atteindre ce but. Maman, papa, votre soutien sans faille durant ces vingt-trois années de scolarité se voit ici récompensé. Je ne vous en serai jamais assez reconnaissant.

A l'issu de toutes ces années, un bilan peut être dressé. Plus aucun doute n'est permis. Tu as raison Pierre-Christophe, je suis définitivement bavard. Je te remercie de m'avoir supporté et écouté (enfin... plutôt "entendu") durant toutes ces années d'étude. Merci à Stéphane, pour son soutien et pour son amitié inconditionnelle. Merci à Elizabeta, pour nos improbables discussions aux accents de l'Est. Merci également à Kalou, son petit frère et leur geekitude commune.

Merci à tous, compagnons de galère, compagnons de croissants, compagnons d'happy : Adrien, Ben, Elena, Jérôme, Julien, Laloi, Oscar, Rami, Régis, Sarga, Sébastien, Thomas, Yvan et tous ceux que j'oublie, pour leurs conseils (plus ou moins) avisés, pour cet environnement de travail convivial, pour cette ambiance détendue, éléments nécessaires à la réussite d'une telle épreuve. En espérant vivre avec vous d'autres aventures.

Enfin, un grand merci à tous les étudiants que j'ai croisés, durant toutes ces années, qui m'ont apporté tant de joie et de satisfaction... et tout particulièrement une étudiante, la plus ravissante de toutes, qui me montre jour après jour la voie, et qui est dorénavant mon épouse : Solaine, je t'aime.

A ma chère et tendre que j'aime infiniment . . .

Table des matières

Table des figures

Liste des tableaux

Première partie

Contexte, problématiques et état de l'art

Chapitre 1

Introduction

Sommaire

Ce mémoire de thèse présente une contribution à la génération de tests à partir de modèles pour la validation fonctionnelle des systèmes embarqués mis en œuvre dans le contexte automobile. L'objectif de ces travaux est la mise en place d'une chaîne outillée allant de la modélisation du système sous test à l'exécution des tests générés sur un banc de test. Dans ce contexte, la contribution des travaux présentés dans ce document se situe plus particulièrement au niveau de la définition d'un langage de modélisation adapté aux systèmes embarqués et à la définition de stratégies de génération de tests. L'implémentation des résultats théoriques obtenus a permis d'évaluer leur pertinence dans un contexte opérationnel, offrant l'assurance d'une qualité optimale de ce type de systèmes.

Ce chapitre présente le contexte et les objectifs de ces travaux ainsi que les éléments permettant de répondre aux problématiques soulevées. La section 1.1 présente le contexte général de ces travaux. La section 1.2 présente le projet VETESS, cadre dans lequel nos travaux se sont déroulés. La section 1.3 introduit les problématiques inhérentes à la mise en place d'un tel processus et la section 1.4 présente le plan de ce mémoire. Enfin, la section 1.5 décrit l'exemple fil rouge utilisé tout au long de ce document afin d'illustrer nos travaux.

1.1 Contexte

Un système embarqué est un système électronique, piloté par un logiciel, qui est complètement intégré au système qu'il contrôle. Lorsque ce système comporte également une partie mécanique, on parle de système mécatronique [GB03]. Un des premiers systèmes modernes embarqués reconnaissables a été le système de guidage des missions lunaires du programme Apollo (AGC pour *Apollo Guidance Computer*) [Tom88]. De nos jours, les systèmes embarqués sont présents dans des domaines de plus en plus nombreux et variés tels que l'automobile, l'aéronautique, les télécommunications, le médical... En 2008, une étude du groupe VDC research [1] [VDC08] a évalué la taille du marché de l'embarqué à 46,8 M$ avec une croissance de 20% pour la période 2007-2010, ce qui permettait d'estimer ce marché début 2012 à 100 M$, en supposant que ce taux de croissance se soit maintenu en 2011. La complexité grandissante des systèmes embarqués, combinée aux contraintes constantes de qualité et aux délais de commercialisation de plus en plus courts, nécessite la mise en place de techniques toujours plus efficaces permettant de garantir la qualité du produit.

De nos jours, les véhicules contiennent un nombre élevé de systèmes embarqués. Les véhicules les plus complets peuvent contenir jusqu'à 70 ECU (unité électronique de contrôle). Chaque ECU est composée d'une couche physique et de plusieurs couches de logiciels embarqués. Les systèmes embarqués étant de plus en plus présents, de nouvelles normes ont été mises en place. La norme 26262 [26211], adaptée de la norme CEI 61508 [61510] sur la sécurité fonctionnelle, s'intéresse par exemple à la sécurité fonctionnelle des véhicules routiers. Cette norme décrit, entre autres, les étapes nécessaires au test pour garantir la conformité de ce type de systèmes tels que : inspection du code source, inspection du modèle, couverture de code, test basé sur les exigences...

Afin de pouvoir garantir la qualité d'un système, il faut connaître les exigences auxquelles le système doit répondre. Elles sont regroupées en deux grandes catégories. La première regroupe les exigences fonctionnelles. Elles décrivent les fonctionnalités du système et leurs caractéristiques : en d'autres mots, ce que le système doit être capable de faire. La seconde regroupe les exigences non fonctionnelles. Il s'agit des propriétés définissant la qualité des services proposés par le système. On peut citer par exemple la sécurité, la fiabilité, la performance, la maintenabilité, la portabilité... Ces exigences sont plus difficilement validables car, au contraire des exigences fonctionnelles, elles ne correspondent pas toujours à un cas d'utilisation précis du système. Pour ce genre d'exigences, le processus permettant de garantir qu'elles sont respectées par le système est plus complexe

1. `http://www.vdcresearch.com/`

et nécessite généralement une phase d'analyse afin d'identifier le sens concret de la propriété vis-à-vis du système réel [ATF09, HTES09]. Dans la suite de ce chapitre, nous nous intéresserons uniquement aux exigences fonctionnelles.

Dans le domaine du génie logiciel, les processus permettant d'évaluer la conformité du produit vis-à-vis des exigences peut se découper en deux grandes catégories : la validation et la vérification. La norme ISO 9000 [90005], relative à la gestion de la qualité, fournit les définitions suivantes à l'égard de ces termes :

- Validation : confirmation par l'examen et la fourniture de preuves objectives que les exigences, pour un usage ou une application voulue, ont été remplies.
- Vérification : confirmation par l'examen et la fourniture de preuves objectives que des exigences spécifiées ont été remplies.

La différence se situe donc au niveau des cas dans lesquels les exigences doivent être respectées. Dans le cas de la vérification, il faut montrer que les exigences sont respectées quel que soit le cas d'utilisation. Pour ce faire, il existe deux approches principales : le *Model-Checking* et la preuve. Le *Model-Checking* [Alu11, HLC+09] est une technique consistant à explorer de manière exhaustive tous les états atteignables ou toutes les exécutions d'un modèle et de vérifier, pour chacun d'entre eux, la validité d'un ensemble de propriétés. La preuve [Lan98] consiste à prouver avec des axiomes et des règles d'inférences que des formules sont valides pour une représentation donnée du système.

Dans le cadre de nos travaux, nous nous sommes intéressés à l'activité de validation, c'est-à-dire à montrer que les exigences sont remplies dans certains cas d'utilisation spécifiques du produit fini. Il existe plusieurs techniques de validation, formelles ou semi-formelles, qui sont plus ou moins aisées à mettre en place rigoureusement dans un contexte industriel. Le *prototypage* [AHD11] consiste à réaliser une version incomplète du système sous test afin de contrôler par exemple que les entrées/sorties sont compatibles. Cette technique est généralement utilisée pour effectuer la validation d'un système en cours de développement ou expérimenter au plus tôt la viabilité du développement. Cela permet d'obtenir rapidement une maquette permettant de valider certaines exigences afin de confirmer au plus tôt les choix de conception et d'anticiper les mauvais choix d'implémentation. La *simulation* [WZEK10] est une activité consistant à exécuter un programme informatique reproduisant les fonctionnalités et les réactions du système réel et à contrôler que le système réel réagit de la même manière. C'est une technique très répandue dans le domaine automobile du fait de la complexité des systèmes. En effet, cette technique permet l'exécution et la validation d'un grand nombre de séquences d'exécution en un temps restreint. Le développement de la version simulée peut cependant prendre beaucoup de temps. Cette technique peut être couplée à du Model-Checking [KEP05] afin de vérifier

que la version simulée respecte bien certaines propriétés.

La technique de validation la plus répandue dans le domaine industriel est le test [KT09]. Un test consiste à exécuter un système ou une certaine partie de ce système en lui soumettant un ensemble de stimuli choisis, et d'observer son comportement pour déterminer si celui-ci est conforme aux spécifications et aux exigences initialement définies dans son cahier des charges. Toutefois, l'inconvénient du test, comme l'a souligné Dijkstra, est qu'il peut être utilisé pour détecter les bugs, mais jamais pour montrer qu'il n'y en a pas [Dij70]. En effet, l'explosion combinatoire couplée à l'infinité de scénarios potentiels empêche généralement de tester un logiciel de manière exhaustive. Le test participe donc à augmenter le niveau de confiance dans le logiciel lorsqu'aucun défaut ne se manifeste, tout en rendant possible la maîtrise des coûts de validation : la phase de test peut être interrompue à tout instant sans remettre en cause les résultats déjà obtenus. Le test est une activité primordiale pour l'obtention de logiciels fiables et c'est actuellement la principale technique utilisée dans le monde industriel pour certifier de la qualité d'un logiciel.

Le test permet aux entreprises de réaliser des économies substantielles s'il est correctement mis en œuvre. En effet, en 2002, une étude de l'Institut National (américain) des Normes et des Technologies (le NIST, *National Institute of Standards and Technology*) a montré que chaque année, 59.5 milliard de dollars américains sont perdus aux Etats-Unis à cause de problèmes de qualité au sein des applications logicielles et des systèmes embarqués. Ces problèmes tels que des produits qui ne correspondent pas aux besoins fonctionnels ou commerciaux, des erreurs, des interruptions de services, engendrent parfois de lourdes pertes financières [Tas02]. Par exemple, le 4 juin 1996, la fusée Ariane 5 a été détruite en plein vol, du fait d'un problème logiciel, engendrant une perte financière de plus d'un milliard de francs [Dow97, Nus97]. Cet événement a mis en lumière le manque de validation des systèmes embarqués. Cet incident n'a heureusement pas provoqué de perte humaine mais ce n'est pas toujours le cas. Citons l'exemple des défibrillateurs en libre service affichant un message d'erreur et empêchant leur fonctionnement au moment crucial [Har12]. Les systèmes embarqués sont des systèmes particulièrement complexes du fait de la combinaison de parties matérielles et logicielles. Les bugs sont parfois d'origines multiples et peuvent avoir des conséquences complexes comme dans l'exemple du vol 72 de la compagnie aérienne australienne Qantas qui a effectué de violents piqués à cause d'un problème d'origine combiné mécanique et logiciel [Qan08].

Ces différents incidents créent encore aujourd'hui un fort besoin pour le développement de nouvelles techniques de génération de tests. Cependant, dans le contexte du développement logiciel, le test est souvent mal perçu pour les raisons suivantes :

- C'est une variable d'ajustement : dans la dynamique d'un projet où les exigences au niveau des délais de livraison sont strictes, les phases de test sont souvent repoussées en fin de processus, lorsqu'il n'est plus possible d'attendre.
- C'est une punition : les équipes responsables de la phase de test sont souvent nommées par défaut et manquent souvent de connaissances et de compétences dédiées.
- Cela perturbe le développement : lorsque des dysfonctionnements sont signalés à l'équipe de développement, elle dépense un pourcentage croissant de son temps à corriger ces anomalies.
- Les délais sont variables : on ne peut pas prédire la fin et le résultat d'une phase de test. Est-ce que toutes les parties de l'application vont-être couvertes ? Est-ce suffisant ?

Ainsi, afin que cette tâche ait meilleure réputation et soit la plus efficace possible, il est nécessaire de définir des stratégies et des processus, adaptés à chaque situation, afin que chaque application/système puisse être testée avec un investissement contrôlable, une performance optimale, et des délais tenus. Des techniques de gestion de projet dites *agiles* [BBvB+01] (SCRUM, XP...) existent également afin de faciliter l'intégration des phases de test au sein du développement.

De façon générale, les techniques de test peuvent être répertoriées en plusieurs catégories suivant différents critères. Tout d'abord, en terme de visibilité du système, on discerne deux catégories. On parle de test boîte blanche lorsque que l'on accède et que l'on prend en compte le fonctionnement interne du système pour déterminer la suite de stimuli à exercer. C'est le cas lorsque l'on souhaite valider directement le code même du logiciel. Le test boîte blanche est généralement réalisé au cours du développement. En revanche, si l'on considère le système dans sa globalité, on parlera de test boîte noire. Dans ce cas, les seuls éléments pris en compte pour générer les tests sont les accesseurs disponibles à un niveau utilisateur du système. On parle parfois de test boîte grise lorsque l'on se positionne à un niveau global en ayant tout de même accès à certains éléments de fonctionnement interne. Idéalement, le test boîte blanche et le test boîte noire coexistent au sein d'un même projet [MSBT04]. Dans notre cas, nous nous intéressons au test boîte noire car notre but est de tester le produit fini, dans sa globalité.

On distingue également une classification des tests en fonction des quatre niveaux suivants. Le test de *composant* ou *unitaire* s'intéresse à la validation d'une portion de programme. Il s'agit d'une phase de test généralement réalisée au cours du développement afin de vérifier que chaque partie est correctement construite indépendamment des autres. Les tests *d'intégration* permettent de valider la mise en commun des différentes parties développées. Les tests *système* permettent de valider la globalité du système vis-à-vis des

exigences. Enfin, le test de *recette* correspond à l'étape durant laquelle le client valide le produit livré. Nos travaux se positionnent au niveau du test *système* permettant de valider le système dans sa globalité.

Les techniques de génération de tests sont nombreuses. Celles qui nous intéressent consistent à générer des tests à partir d'un modèle abstrait représentant le système sous test. Les réactions du système et du modèle sont alors comparées lors de l'exécution des tests. Mais la génération de tests peut se baser sur d'autres procédés comme l'utilisation de scénarios [AMS06] ou sur des techniques de vérification comme le Model-Checking [HLM+08, CSE96, FMS09a]. Les travaux présentés dans ce mémoire ont été réalisés dans le cadre du projet VETESS présenté dans la prochaine section.

1.2 Le projet VETESS

VETESS (*vérification de systèmes embarqués VEhicules par génération automatique de TESts à partir des Spécifications*) est un projet labellisé par le pôle de compétitivité *Véhicule du Futur*, financé par le FUI (*Fonds Unique Interministériel*), qui s'est déroulé du 1er septembre 2008 au 31 août 2010 [VET10a]. L'objectif stratégique du projet VETESS a été de produire des outils conceptuels, méthodologiques et techniques pour la vérification de systèmes mécatroniques embarqués dans les véhicules par génération automatique de tests à partir des spécifications.

Le projet VETESS s'inscrit dans le contexte de la montée en puissance de l'ingénierie dirigée par les modèles pour la conception de systèmes embarqués dans les véhicules. Il s'agit là d'un enjeu stratégique pour l'industrie automobile afin de maîtriser la fiabilité de systèmes embarqués de plus en plus complexes, mais aussi pour raccourcir le délai de la mise sur le marché des nouveaux modèles et de maîtriser les coûts de conception. L'ingénierie dirigée par les modèles constitue la colonne vertébrale de cette démarche continue et la génération de tests à partir des modèles un moyen pour garantir cette continuité. La chaîne outillée part de modèles représentant le fonctionnement attendu du système sous test et produit des tests exécutables sur les bancs de test dédiés à cette fonction. Les problématiques globales résolues par ce projet concernent l'optimisation des coûts et l'attestation de la fiabilité de systèmes mécatroniques complexes, en s'appuyant sur une continuité, basée sur les modèles, entre les phases de conception et les phases de qualification des équipements et des organes.

Pour remplir ces objectifs, le projet VETESS s'appuie sur un partenariat fortement complémentaire avec une expertise importante dans l'ingénierie dirigée par les modèles et la génération automatique de tests. Ce projet a regroupé les acteurs suivants :

- Smartesting, leader du projet, est une société basée à Besançon (Franche-Comté) qui développe et commercialise un logiciel de génération de tests à partir de modèles (Test Designer^{TM}) reconnu entre autres dans le domaine des technologies de l'information.
- Clemessy, société basée à Mulhouse (Alsace), spécialisée dans le domaine de l'embarqué automobile, propose notamment la mise en place de bancs de test pour la validation de systèmes. La solution proposée par cette société contient une plateforme d'exécution de tests baptisée *Test In View* (TIV).
- Le laboratoire MIPS (*Modélisation, Intelligence, Pocessus et Systèmes*) de l'Université de Haute-Alsace est reconnu pour son expertise dans l'ingénierie dirigée par les modèles.
- Le laboratoire DISC (*Département Informatique des Systèmes Complexes*) de l'Université de Franche-Comté est reconnu pour son expertise dans la génération de tests à partir de modèles.
- PSA Peugeot-Citroën, constructeur automobile, a guidé nos travaux en fonction de ses besoins et a proposé les cas d'étude du projet.

Les travaux présentés dans ce mémoire de thèse correspondent majoritairement aux éléments de début du processus mis en place au cours du projet VETESS à savoir la modélisation du système sous test et la génération de tests qui y est appliquée. Les autres éléments de la chaîne (concrétisation et exécution des tests) sont brièvement introduits en fin de document. Du fait de la collaboration avec la société Smartesting, l'objectif a consisté à développer une chaîne outillée basée sur l'outil de génération de tests Test Designer^{TM} développé par cette société.

Smartesting Test Designer^{TM}

La solution Smartesting propose un processus continu de génération de tests allant de la modélisation du système à la génération automatique de tests couvrant les comportements attendus du système sous test [BBC+06] en passant par la représentation des exigences fonctionnelles.

Cet objectif pose la problématique de la représentation des fonctionnalités d'une manière explicite et détaillée. Test Designer^{TM}, support logiciel de la solution Smartesting, permet de générer des cas de test à partir d'un modèle, spécifiant le système sous test, écrit avec un sous-ensemble de la notation UML baptisé UML4MBT [Sma09] (*UML for Model-Based Testing*). Le modèle de test doit être suffisamment complet et détaillé afin de permettre la génération de scénarios de test. L'outil Test Designer^{TM} propose de générer des tests à partir d'un tel modèle en assurant une couverture structurelle de celui-ci.

UML et UML4MBT

UML (*Unified Modeling Language*) est un langage graphique de modélisation des données et des traitements. Il résulte de la fusion de plusieurs méthodes d'analyse telles que OMT [RBP+90], OOSE [Jac92] et la méthode Booch [Boo93]. Au fil des années, UML a réussi à s'imposer et devenir le langage de modélisation objet le plus utilisé dans le monde. Il est standardisé par l'*Object Management Group* (OMG [OMG10]) depuis 1997 ; la version 2 d'UML a été adoptée par ce consortium en 2005 [RJB05]. UML est donc aujourd'hui le standard incontournable dans le domaine de la modélisation de logiciels. La notation UML 2 regroupe 13 types de diagrammes qui permettent de modéliser tout un système sous différentes vues (cas d'utilisation, logique, implémentation, processus et déploiement) [MG03]. Toutefois, chaque secteur d'activité possède ses propres règles et caractéristiques métier. Par rapport au formalisme UML standard, la notation UML laisse alors à chacun la possibilité de personnaliser les éléments de modélisation UML standards pour tenir compte de la spécificité de son domaine d'application au moyen de profils tels que SysML (*System Modeling Language*) [FMS09b], MARTE (*Modeling and Analysis of Real-Time and Embedded Systems*) [Gro07]...

Le logiciel Test Designer^{TM} utilise un sous-ensemble d'UML offrant des caractéristiques permettant la définition d'éléments de modélisation précis, nécessaires et suffisants à la représentation des comportements des systèmes d'information. Concrètement, un modèle UML4MBT utilise les 3 types de diagrammes suivants :

- Le diagramme de classes permettant de modéliser la structure statique du système sous test en décrivant une vue abstraite des entités du système (classes, attributs) et leurs dépendances (associations) ainsi que les actions proposées par le système (opérations).
- Le diagramme d'objets modélisant les objets concrets manipulés par les cas de test et l'état de ces entités dans l'état initial du système. Il s'agit d'une instanciation particulière du diagramme de classes associé. Les données de test représentées dans ce diagramme sont utilisées par l'outil Test Designer^{TM} pour définir les valeurs d'entrée des paramètres ainsi que pour calculer les valeurs attendues. La création (resp. suppression) dynamique d'entités au sein du système est simulée par la création (resp. suppression) de liens entre les objets du modèle.
- Le diagramme d'états-transitions permettant de représenter une vue dynamique du système. Il spécifie l'effet des actions réalisées par l'utilisateur et par les composants extérieurs au système en terme de modifications des valeurs d'attributs des entités. Le modèle de test représente de ce fait les comportements attendus des différentes actions qui impactent le système sous test.

La représentation dynamique est complétée par des contraintes OCL (*Object Constraint Language*) afin de formaliser les comportements attendus permettant une modélisation non ambigüe [BBH02, SIAB11].

OCL et OCL4MBT

OCL est un langage de modélisation orienté objet défini comme un standard ajouté à la notation UML et géré par l'OMG (*Object Management Group*). OCL sert généralement à préciser de manière formelle les comportements du système. Il permet d'exprimer des aspects essentiels qui ne peuvent pas être exprimés uniquement à l'aide de diagrammes. C'est pour cela que la combinaison d'OCL et d'UML est connue comme une solution appropriée à la modélisation des fonctionnalités d'un système.

Dans le cadre de l'outil Test Designer^{TM}, un sous-ensemble d'OCL (OCL4MBT pour *OCL for Model-Based Testing*) est utilisé. Cet ensemble apparaît comme suffisant pour la modélisation de comportements fonctionnels du système sous test. Les expressions OCL sont utilisées au sein du modèle UML4MBT pour décrire des gardes et des effets de transition. Une garde de transition est une contrainte permettant de spécifier les conditions devant être remplies pour que cette transition soit franchissable. Il s'agit d'une expression booléenne qui doit être évaluée à vrai pour que la transition puisse être franchie. Les effets de transition décrivent les modifications des états du système à travers la mise à jour des variables du système. OCL4MBT offre la syntaxe permettant l'expression de ce type d'action. En effet, afin de pouvoir exécuter les expressions OCL4MBT dans le cadre de la génération de tests, une interprétation opérationnelle spécifique d'OCL est utilisée. L'expression OCL `self.attribute=true` peut être interprétée de deux manières différentes : passive ou active. Le contexte passif permet d'exprimer une contrainte sur le système sous test alors que le contexte actif permet la représentation d'un changement d'état. Dans le cadre booléen (pour une garde de transition), elle sera interprétée de manière passive (comme nativement en OCL), c'est-à-dire que `self.attribute` sera comparé à `true` et une valeur booléenne sera retournée. Cependant, dans le cadre d'un effet de transition, cette expression aura pour action l'affectation de `true` à la variable `self.attribute`.

Génération de tests

Le modèle UML4MBT est ainsi suffisamment précis et l'absence d'ambiguïté permet la compréhension et la manipulation automatique par le générateur de tests des comportements modélisés. Cela rend possible la simulation de l'exécution du modèle et son utilisation tel un oracle permettant la prédiction des réactions du système sous test.

Les expressions OCL associées aux diagrammes d'états-transitions ajoutent un niveau de formalisation pour la génération de tests à partir de modèles. Le modèle réalisé, définissant tous les comportements à tester du système sous test, est exporté à partir du modeleur vers le générateur de tests. Le générateur de tests propose une couverture automatique de tous les comportements du modèle sous test. Chaque test correspond à une séquence d'actions à exécuter sur le système. Un test est composé de trois parties :

1. mise en contexte : séquence d'activations préliminaires permettant la mise en contexte du système afin d'accéder au comportement à tester,
2. invocation du comportement à tester, incluant la comparaison du résultat obtenu avec l'oracle,
3. retour à l'état initial afin qu'un autre test puisse être joué automatiquement.

Finalement, il est possible d'exporter les tests générés vers un dépôt de tests ou vers une plateforme d'exécution des tests.

1.3 Problématiques

Nos travaux sont motivés par la réalisation d'une chaîne outillée originale, de par la mise en collaboration d'outils performants utilisés généralement indépendamment et de par sa facilité d'utilisation au regard de la complexité des systèmes qu'elle permet de valider. En effet, elle doit permettre l'exécution des tests générés, aussi bien sur un banc de test physique que sur un simulateur et ce, quel que soit le type de système embarqué considéré (discret ou continu).

La définition d'une chaîne outillée, de génération de tests à partir de modèle, dédiée au domaine de l'embarqué pose plusieurs problématiques originales. Tout d'abord, il est nécessaire d'utiliser un langage de modélisation adapté à de tels systèmes. En effet, les systèmes embarqués ont des caractéristiques spécifiques telles que la présence de parallélisme au niveau de l'évolution des différents composants du système ou des connections avec des composants extérieurs. La première étape consiste donc en la définition d'un langage de modélisation adapté aux systèmes que l'on veut tester.

Ensuite, il faut définir une stratégie de génération de tests adaptée aux modèles ainsi réalisés. Le but est d'obtenir, par l'application de cette stratégie, une couverture adéquate et relevante du modèle réalisé apportant l'assurance d'une qualité optimale au regard des exigences du système testé.

Finalement, un élément stratégique dans le domaine du test industriel consiste en l'automatisation du procédé afin de faire face à la taille parfois conséquente des systèmes sous tests. La stratégie mise en place se doit donc d'être automatisable en impliquant un minimum d'intervention humaine. De plus, elle doit être adaptable aux différents systèmes afin de proposer un retour sur investissement significatif.

Afin de pouvoir expérimenter et utiliser les éléments permettant de répondre à ces problématiques, ils doivent être implémentés au sein d'une chaîne outillée. Dans le cadre du projet VETESS, l'outil de génération de tests choisi est Test Designer^{TM}. Il se pose alors le problème de l'adaptation de notre processus à cet outil prenant en entrée uniquement des modèles UML4MBT et proposant une couverture spécifique de ces modèles. Nos travaux se sont placés dans une optique de preuve de concept, nécessitant ainsi la mise en place de connecteurs pour une meilleure adaptation aux outils existants.

Finalement, l'efficacité du processus mis en place est alors évaluée par l'expérimentation de cette nouvelle chaîne outillée sur plusieurs cas d'étude. Le domaine de l'embarqué automobile étant vaste, nous avons sélectionné un panel de cas d'étude représentatifs.

La prochaine section présente le plan de ce mémoire en associant chaque partie aux problématiques auxquelles elle répond.

1.4 Structure du document

Tout d'abord, le chapitre 2 présente les travaux de la littérature en rapport avec les principaux éléments présentés dans ce mémoire. Après un aperçu des différentes techniques de validation des systèmes embarqués, ce chapitre fait un focus sur les techniques de génération de tests à partir de modèles appliquées au domaine de l'embarqué ainsi que sur les différentes techniques de modélisation.

Le chapitre 3 présente la solution que nous proposons à la problématique de la modélisation des systèmes embarqués. En l'occurrence, ce chapitre présente le langage SysML et l'utilisation qui en est faite.

Le chapitre 4 propose une stratégie de génération de tests adaptée aux modèles réalisés permettant une couverture efficace des comportements modélisés.

Le chapitre 5 présente une solution pour l'implémentation des éléments présentés dans les sections précédentes dans une optique de preuve de concept. Cette section décrit les algorithmes et les connecteurs mis en place pour la définition d'une chaîne outillée automatisée.

Enfin, les résultats de l'application de cette chaîne outillée sur trois cas d'étude sont présentés en chapitre 6. Ce chapitre permet ainsi de montrer les points forts et les limites de notre approche.

La dernière partie de ce document conclut ces travaux et présente d'éventuelles perspectives permettant d'améliorer le processus mis en place.

L'exemple fil rouge présenté dans la section qui suit permet d'illustrer les différentes parties de ce mémoire.

1.5 Exemple fil rouge : système d'éclairage avant de véhicule

Afin d'illustrer l'ensemble des éléments de ce document, nous définissons dans cette partie, la spécification d'un cas d'étude fil rouge. Cet exemple est réutilisé tout au long de ce mémoire afin d'illustrer les différents points de ce travail.

Ce cas d'étude concerne le système d'éclairage avant d'un véhicule. Dans le cas d'une voiture par exemple, la majorité des modèles possède un commodo, proche du volant (généralement sur la gauche), permettant d'allumer et d'éteindre les feux et les phares. Il est également possible, à l'aide de ce commodo, de réaliser des appels de phares. Basé sur ce cas *classique*, un cas particulier où le commodo serait remplacé par un écran tactile placé sur les bords du volant est considéré. La spécification détaillée est la suivante.

Le véhicule est équipé d'un panneau tactile également appelé panneau de contrôle. Ce panneau est composé d'un écran dynamique (l'affichage peut donc varier) et d'une surface tactile. La communication entre les différents composants se fait par la transmission de signaux électriques. Ce panneau, ainsi que les lumières (feux de croisement et phares), sont désactivés lorsque le véhicule est éteint. Lorsque le véhicule est en marche, initialement, aucune lumière n'est allumée et le panneau tactile est affiché en conséquence. Deux fonctions sont alors disponibles : allumer les feux de croisement ou faire un appel de phares. Ces fonctions sont accessibles à l'aide de deux boutons distincts affichés sur l'écran du panneau tactile. Si on allume les feux de croisement, les fonctions évoluent. Il est alors possible : d'allumer les phares, d'éteindre les feux de croisement ou de faire un appel de phares.

Si, à cette étape, l'utilisateur allume les phares, les feux de croisement sont alors éteints en échange de l'allumage des phares. Une fois les phares allumés, il est toujours possible de faire un appel de phare. L'autre possibilité consiste à éteindre les phares (ce qui aura pour effet de rallumer les feux de croisement). Quel que soit l'état du système, si la fonction est accessible, l'appel de phare consiste à allumer les feux de croisement ainsi que les phares tout au long de l'appui de la zone du panneau de contrôle correspondante. Dès que la zone est relâchée, le système retourne dans l'état dans lequel il était précédemment. Si l'on coupe le contact du véhicule, le panneau de contrôle est désactivé, tout comme les feux de croisement et les phares.

Il est à noter que d'autres fonctions telles que le détecteur de luminosité pourraient être spécifiées au sein de ce cas d'étude. Elles sont cependant écartées dans le cadre de ce mémoire afin de simplifier les manipulations de modèles et rendre plus claires les illustrations proposées.

Chapitre 2

Etat de l'art

Sommaire

Ce chapitre présente l'état actuel des travaux scientifiques en relation avec la génération de tests à partir de modèles pour la validation de systèmes embarqués. Après une introduction au test à partir de modèles et aux grands domaines d'étude associés en section 2.1, la section 2.2 aborde les stratégies de génération de tests utilisées dans ce domaine. Ensuite, la section 2.3 dresse le paysage des différents langages, outils et techniques permettant la modélisation des systèmes embarqués. Finalement, la synthèse permet de nous positionner parmi ces travaux existants.

2.1 Model-Based Testing

Depuis un dizaine d'années, les méthodes d'ingénierie des systèmes à partir de modèles (MBSE : *Model-Based System Engineering*) ont émergé [Est07] et ont mis en avant l'utilisation des modèles aux différentes étapes de spécification du système. Dans cette dynamique, le code n'est plus la seule donnée à prendre en compte afin de concevoir les tests, la spécification sous forme de modèle entre en jeu [UL06].

Le Model-Based Testing (MBT) ou génération de tests à partir de modèle correspond à une approche où le système est représenté sous la forme d'un modèle abstrait qui représente les comportements du système (tous ou en partie). Cette approche consiste à raisonner sur ce modèle de spécification pour calculer automatiquement des séquences abstraites (car de même niveau de détail que le modèle) de stimuli qui constituent les tests. Ces séquences de stimuli sont ensuite concrétisées puis exécutées sur le système (réel) sous test. Le verdict est donné en comparant les résultats obtenus sur le système sous test avec les résultats produits par le modèle. Le modèle constitue ainsi l'oracle de test car il permet de prédire les résultats attendus sur le système cible [DNT10, UPL11]. Il représente une vue dynamique et précise l'état initial du système en formalisant le comportement du système et en capturant les points de contrôle et d'observation du système sous test. Le modèle se doit d'être suffisamment précis et formel afin de permettre une interprétation non ambigüe pour disposer d'une génération automatique reproductible [ZB09].

Les techniques de génération de tests à partir de modèles ont connu depuis plusieurs années un intérêt et un essor considérables. Cette attractivité, de la part des milieux scientifique et industriel, s'explique notamment par la complexité toujours croissante des systèmes informatisés et le besoin récurrent de garantir une qualité de service et de fiabilité optimum. En effet, générer des tests à partir de modèles de spécification permet d'assurer un niveau de confiance élevé vis-à-vis du respect des exigences énoncées dans le cahier des charges de l'application à tester grâce à l'utilisation de critères de couverture. Ce type de technique commence à trouver sa place du fait de son automatisation partielle ou complète rendue possible par la capacité à raisonner sur les modèles de spécification formelle. Enfin, l'utilisation du modèle peut être initiée très tôt durant le cycle de développement, permettant un suivi et une validation du système étape par étape ce qui est un point fort dans un domaine où le développement d'un système peut prendre plusieurs années.

En dépit des nombreux résultats, le domaine est toujours très actif pour répondre notamment aux nouvelles exigences du marché : normalisation de plus en plus stricte, augmentation de la qualité de service attendue, volonté de maîtriser les coûts...et évi-

demment, l'arrivée de systèmes de plus en plus complexes tels que les systèmes embarqués modernes. C'est pourquoi, garantir la qualité d'un système représente encore aujourd'hui un secteur dynamique pour la recherche et l'innovation comme le prouve le nombre important de conférences et de journaux spécialisés dans cette thématique.

Outre la problématique de la capture des comportements du système à travers un modèle, le MBT présente plusieurs problématiques au niveau de la démarche de génération de tests à proprement parlé. En effet, la génération de tests doit être pertinente, c'est-à-dire permettre l'obtention de tests assurant un certain niveau de qualité, tout en étant maîtrisée afin d'éviter l'explosion combinatoire. La philosophie de ces travaux s'inscrit directement dans ces problématiques puisqu'il s'agit d'optimiser la pertinence des tests tout en garantissant une maîtrise de la complexité algorithmique et du nombre de cas de test générés. La génération de tests à partir de modèles présente donc trois grands domaines d'étude :

1. la modélisation des systèmes sous test,
2. la définition de stratégies de génération adaptées aux besoins et permettant d'assurer une qualité optimale,
3. et l'implémentation de ces éléments au sein d'une chaîne outillée pour assurer un processus reproductible et si possible automatique.

Dans la suite, nous vous présentons l'étude de l'existant pour répondre à notre problématique vis-à-vis de ces trois domaines.

2.2 Stratégies de génération de tests

Cette section présente les travaux concernant les stratégies de génération de tests à partir de modèles en introduisant les techniques fondamentales les plus utilisées [UL06].

2.2.1 Test aléatoire

Le critère de sélection aléatoire est certainement le plus naïf de tous puisqu'il consiste à choisir aléatoirement les séquences d'exécution à utiliser comme tests. La génération de ce type de test peut être interrompue à tout instant sans remettre en question le critère de sélection. Concrètement, il s'agit de parcourir les spécifications aléatoirement (en choisissant de façon aléatoire les stimuli à enchaîner). Cette stratégie de génération est ainsi très facile à mettre en œuvre mais ne permet pas de garantir une couverture structurelle de la spécification : certains stimuli peuvent ne jamais être invoqués du fait du caractère aléatoire du tirage.

Bien que la pertinence des tests générés ne soit pas maîtrisable par ce critère et qu'aucune garantie de couverture du modèle ne soit assurée, les études tendent à montrer que ce type de sélection peut être efficace lorsqu'il s'agit de générer un nombre restreint de tests (en nombre et/ou en taille) dans un temps imparti court [DN84].

Certains travaux ont proposé des améliorations à ce critère car il est particulièrement adapté aux modèles représentant beaucoup de comportements (plus de 1000 états et plus de 4000 transitions par exemple). En effet, pour ce type de modèles, l'application de critère de couverture, même stricte (critère générant a priori peu de tests), peut engendrer la génération d'un nombre de tests trop important pour être manipulé. L'approche adoptée afin de rendre le test aléatoire plus relevant tout en conservant son adaptabilité aux modèles de grande taille consiste à appliquer des probabilités sur la sélection des données en entrée. Ces probabilités sont calculées à l'aide de critères spécifiques et permettent donc d'obtenir une génération aléatoire guidée [DGG+12]. Une autre manière de profiter de la diversité apportée par la génération aléatoire tout en permettant une génération pertinente peut consister à utiliser les principes tirés de l'algorithmique génétique et évolutionnaire [IAB11]. En revanche, plus l'effort porté sur les tests est important (en temps ou en quantité), moins ces techniques s'avèrent efficaces comparées aux techniques présentées dans les sections suivantes [DN84].

2.2.2 Test exhaustif

Le test exhaustif ne présente en fait aucune sélection particulière. Cette stratégie consiste effectivement à produire un ensemble de tests qui permet de couvrir l'intégralité de tous les comportements et les successions de comportements possibles du modèle (qui représente le système sous test). Il est rare qu'une telle approche soit praticable dans la réalité : en effet, le nombre de comportements et de permutations de comportements d'un système est bien souvent infini, ce qui rend caduque toute idée de test exhaustif. Du moment où le modèle (et donc le système) contient au moins une boucle, c'est-à-dire qu'il est possible de quitter un état et d'y revenir après une ou plusieurs actions, le nombre de tests est infini [Wei10]. Même lorsque le modèle ne contient aucune boucle et qu'ainsi le nombre théorique de tests est fini, la combinatoire des permutations de comportements rend irréaliste cette approche.

Une alternative consiste a considérer l'exhaustivité au niveau des données d'entrée du système. Ce critère assure la couverture de chaque valeur pouvant être prise par chacune des données d'entrée du système. Bien que le nombre de cibles soit plus facilement maîtrisable, cette stratégie produit généralement un nombre de tests trop élevé.

Pour pallier cette perspective idéaliste, les approches classiques s'appuient sur des critères de sélection qui ont pour objectif de définir (intentionnellement) un sous-ensemble de tests jugés suffisamment représentatifs et pertinents pour atteindre un niveau de confiance vis à vis du système sous test et ce, en toutes circonstances, nous parlons alors de couverture.

2.2.3 Couverture structurelle

Un moyen de contingenter le nombre de tests tout en assurant un niveau de qualité précis consiste à définir au préalable des critères de couverture du modèle. Les premiers travaux pour définir des critères de couverture sont issus des techniques de test structurel (procédé de type boîte blanche) qui consistent à définir les entités du code source que l'on veut couvrir avec les tests. Dans le contexte de test boîte noire, il ne s'agit plus d'appliquer ces critères sur le code lui-même, mais de les transposer sur le modèle (de test). L'application des critères permet d'extraire du modèle des cibles (ou objectifs) de tests qui correspondent aux éléments à couvrir. La génération de tests consiste ensuite à couvrir ces cibles par la génération de séquences de stimuli. Dans certains cas, un test peut couvrir plusieurs cibles. On distingue trois familles de critères structurels :

- Critères basés sur les transitions : ils sont naturellement uniquement destinés aux langages de modélisation basés sur des transitions (UML, IOLTS...). Ils se basent sur la couverture des éléments graphiques de modélisation, par exemple, le critère *Tous les états* assure la couverture de chaque état modélisé [OXL99].
- Critères orientés flot de contrôle : ils concernent la couverture des différentes décisions, boucles ou chemins modélisés sous forme d'expressions conditionnelles dans le modèle de test [VB01]. Par exemple, le critère *Decision Coverage (DC)* consiste à couvrir chaque décision modélisée, c'est-à-dire que pour chaque expression booléenne modélisée, (au moins) un test doit l'évaluer à vrai et (au moins) un test doit l'évaluer à faux.
- Critères orientés flot de données : ces critères se basent quant à eux sur l'utilisation des variables manipulées (mise à jour, lecture) au sein du modèle [FW88, RW85]. Par exemple, le critère *Toutes les définitions* assure la génération de tests qui permettent de couvrir au moins une fois chaque traitement d'accès en écriture des variables du modèle de test.

Tous ces types de critères n'empêchent pas l'accroissement du nombre de cibles générées mais permettent en tout cas d'en maîtriser l'explosion. Si le nombre de cibles se trouve être néanmoins trop important, il est possible de partitionner et de regrouper les points d'entrée du modèle tout en conservant les mêmes points d'observation [OB88, DF93].

Les critères de couverture structurelle sont utilisés au sein d'outils de génération automatique de tests, dans lesquels le modèle se suffit à lui-même pour produire des cas de test. En effet, le modèle spécifie le système sous test, et une transformation lui est appliquée pour dériver automatiquement les objectifs de test, base de la génération de tests elle-même.

Par exemple, l'outil *Test Designer*^{TM} de la société *Smartesting* [Sma09] est issu des travaux de recherche autour de l'environnement *BZTT* [ABC+02] (utilisant initialement une modélisation à base des langages de modélisation B [Abr96] et Z [Spi92]). Cet outil se base sur un critère orienté flot de contrôle pour calculer les objectifs de test, et les cas de test correspondant, à partir de modèles de spécification UML/OCL [BGL+07, BGLP08].

L'outil *conformance kit* [MRS+97] propose la génération de tests à partir d'un modèle réalisé en VHDL, langage destiné à décrire des systèmes matériels. Cet outil propose l'application de divers critères structurels pour la sélection des tests tels que le critère de couverture de toutes les transitions ou le critère visant à sélectionner un ensemble de tests minimaux (en terme de nombre de pas) visant à couvrir l'ensemble des transitions du système.

D'autres stratégies basées sur la limitation du nombre de pas de test ou sur la limitation du nombre de parcours des cycles sont également utilisées par l'outil *TorX* [TB03] prenant en entrée des spécifications exprimées par des systèmes de transition.

Quelque soit les critères utilisés, l'objectif final est toujours d'assurer la couverture des exigences exprimées dans le cahier des charges.

2.2.4 Couverture des exigences

Les phases de validation intégrées au sein des processus de gestion de projet visent à valider le système vis-à-vis des exigences. Ainsi, il est nécessaire de lier les tests générés avec les exigences. L'ALM (*Application Lifecycle Management*) [Cha08] est une technique pour la gestion du cycle de vie des applications. Il permet d'exprimer et d'identifier les exigences dans l'ensemble du processus. Ainsi, il devient possible d'assurer une traçabilité de leur évolution.

Le mise en œuvre consiste à relier les exigences avec les éléments de modèle qui permettent de les satisfaire. La génération de tests consiste alors à couvrir toutes les exigences à travers la couverture des éléments de modèles auxquels elles sont associées. Il existe aujourd'hui deux approches.

La première consiste à exprimer cette notion au niveau des langages de modélisation. Par exemple, dans le paradigme UML, il existe le diagramme d'exigences SysML [FMS09b]

qui permet d'exprimer cette liaison. Ainsi, la couverture de ces entités par la génération de tests permettra d'avoir l'assurance de la couverture des exigences.

La seconde consiste à lier le modèle et les exigences au sein des outils de génération de tests. Par exemple, l'outil Test Designer^{TM} présenté précédemment intègre également une solution en proposant des annotations OCL dédiées à l'expression des exigences. Il est alors possible d'assurer une couverture spécifique d'expressions OCL satisfaisant des exigences spécifiques.

2.2.5 Scénarios

Une technique de validation répandue est l'utilisation de scénarios [GRSC08]. La description de scénarios d'utilisation permet l'exécution du système dans des cas définis. Les scénarios sont généralement décrits à l'aide de langage facile d'accès, éventuellement graphique, comme les diagramme de séquence UML par exemple [MBKL10].

Le principe général de cette technique consiste en la création de scénarios de test par l'ingénieur validation qui détermine ainsi un "patron" permettant de filtrer les enchaînements de stimuli qu'il veut voir tester. Les techniques utilisées pour rédiger les scénarios peuvent reposer uniquement sur le savoir-faire de l'ingénieur qui se laisse guider par son expertise pour établir des patrons pertinents vis-à-vis d'une exigence fonctionnelle particulière à tester par exemple. D'autres approches emploient des modèles environnementaux qui caractérisent des cas d'utilisation typiques (nominaux) de l'application.

Ce type d'approche est utilisé par bon nombre d'outils. Par exemple, AGATHA [FGG07, BFG+03, LBV+04] est un outil pour la génération symbolique de cas de test qui prend en entrée des spécifications écrites en UML [RJB05], SDL [EHS97], STATEMATE [Har87], ou ESTELLE [90790]. Quel que soit le langage de spécification, le modèle est traduit en IOSTS (Input Output Symbolic Transition System). Des propositions d'objectifs de test sont alors générées à partir des sous-arbres (parties de l'arbre général d'exécution du système), calculés automatiquement puis sélectionnés par l'utilisateur. L'outil TGV [JJ05], quant à lui, calcule le produit synchronisé d'un objectif de test (définit manuellement par l'utilisateur à l'aide d'un IOLTS) avec le modèle de test. Il en résulte un graphe qui contient tous les cas de test couvrant l'objectif initial. L'outil *SpecExplorer* [VCG+08] propose une génération de tests à partir de modèles écrits en Spec# [BRLS04]. La stratégie utilisée par cet outil nécessite la définition d'états d'acceptation représentant les objectifs de test. [Tis09] propose une approche semblable où les objectifs de test sont définis à l'aide de schémas qui, synchronisés avec le modèle réalisé en langage B, permettent de restreindre les exécutions possibles.

L'inconvénient de ce type d'approche concerne la couverture du modèle qui dépend entièrement du savoir-faire du concepteur des scénarios de test. De plus, la définition manuelle des scénarios à couvrir peut être contraignant pour les systèmes complexes et empêche une automatisation complète du processus. Cette automatisation est très importante car les interventions manuelles sont connues pour être coûteuses et sources d'erreurs. Il a été également montré que l'automatisation permet un bon retour sur investissement au niveau des équipes validation qui le mettent en œuvre [DGG09].

2.2.6 Bilan

Dans le contexte de la génération de tests à partir de modèles, la spécification du système sous test constitue le référentiel d'entrée d'une chaîne automatisée ou semi-automatisée qui permet de calculer des séquences de stimuli pour lesquelles le modèle joue le rôle d'oracle en calculant les résultats attendus. La génération de tests consiste donc à couvrir de manière spécifique le modèle par la sélection de séquences de stimuli permettant une animation du système.

La sélection des tests s'effectue par application d'une stratégie spécifique. Un premier ensemble de stratégies consiste à définir manuellement les objectifs de test en définissant des scénarios ou des états particuliers à atteindre par exemple. Il est ainsi possible de guider la génération de tests et d'adapter les objectifs en fonction du système sous test en privilégiant certaines parties par exemple. Cependant, la définition des objectifs devant être réalisée manuellement, la qualité dépendra des choix faits par l'ingénieur validation.

Une alternative consiste à utiliser des critères de couverture structurelle assurant la couverture du modèle d'une manière spécifique. Cette technique a l'avantage de permettre la mise en place de processus totalement automatiques en assurant une qualité constante et précise, indépendante des ingénieurs de l'équipe en charge de la validation. Il est cependant nécessaire d'utiliser des critères de couverture adaptés aux systèmes modélisés et permettant l'assurance d'une qualité suffisante tout en évitant les problèmes d'explosion combinatoire.

Parmi ces travaux, nous nous intéressons plus particulièrement aux critères de couverture structurelle car ils permettent de définir un processus reproductible et assurent une qualité fixe du système sous test (indépendante des utilisateurs de la chaîne outillée).

Quelle que soit la technique utilisée, il ne faut cependant pas oublier que ces stratégies sont dépendantes de la modélisation et afin d'obtenir un processus performant, il est nécessaire de réaliser un modèle représentant correctement et suffisamment précisément le système sous test. Le choix du langage de modélisation et la façon de réaliser le modèle sont donc des éléments déterminants dans ce contexte.

2.3 Modélisation des systèmes embarqués

Cette section présente un aperçu des langages de modélisation permettant de spécifier les systèmes embarqués, et ainsi servir de point d'entrée aux différentes techniques de test à partir de modèles. Pour ce faire, nous avons décomposé cette section en trois parties qui correspondent à trois paradigmes de modélisation utilisés dans les systèmes embarqués. Quand elles existent, nous présenterons en même temps les techniques de test associées. Tout d'abord, le principe de modélisation environnementale nécessaire dans le domaine des systèmes embarqués est présenté. Ensuite, différents langages de modélisation adaptés à ce type de système sont détaillés. Enfin, avant de dresser le bilan, une partie présente brièvement les problématiques particulières de modélisation du temps.

2.3.1 Modélisation environnementale

Les systèmes mécatroniques sont des systèmes qui combinent des composants logiciels, électroniques et mécaniques permettant la réalisation d'une fonction particulière. Dans ce contexte, les logiciels embarqués correspondent uniquement à une partie d'un système plus grand [GB03]. De ce fait, un système embarqué n'existe que parce qu'un système plus conséquent a besoin de lui. Il apparaît donc que le test d'un système embarqué ne peut certainement pas être efficace sans tenir compte du système qui le contient. Afin de pouvoir analyser et valider de manière pertinente ce type de système, il faut donc considérer l'ensemble des composants susceptibles de communiquer avec lui.

Depuis 1990, le programme de simulation PELOPS[2] a été développé selon l'idée que pour analyser le comportement d'un système embarqué dans un véhicule que l'on spécifie, il est nécessaire de représenter le véhicule, le conducteur ainsi que l'environnement [EH00]. Ce triplet est représenté sur la figure 2.1 selon [EH00]. Ce type de modélisation peut être validé par simulation : les résultats théoriques issus de ces modélisations sont comparés avec les résultats concrets donnés par le système physique correspondant [GMSM02].

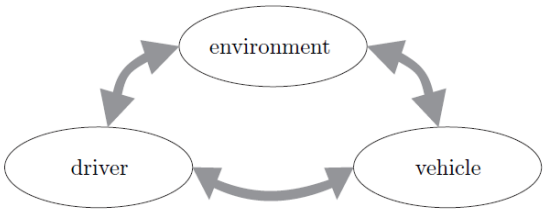


FIGURE 2.1 – Triplet de modélisation des systèmes embarqués véhicule

2. cf. `http://www.pelops.de/UK/index.html`

Le triplet ainsi défini doit contenir tous les éléments qui influencent le système embarqué. Il est utilisé dans beaucoup de travaux et est également utilisé comme base des travaux présentés dans ce mémoire. Cependant, la prise en compte de chacun des éléments de l'environnement introduit rapidement des problèmes d'explosion combinatoire, particulièrement quand chaque partie du système est testée indépendamment [vRT04]. Qui plus est, pour détecter les comportements indésirables, il est nécessaire d'étudier les interactions entre les différents composants [PEML10].

Plusieurs approches pour le test de systèmes embarqués sont basées sur un tel modèle environnemental [HLM+08, TMJL09]. Typiquement, comme proposé dans [IAB11], un cas de test est défini comme une séquence de stimuli qui sont envoyés depuis l'environnement au système embarqué. Afin de générer des cas de test, les approches de test aléatoire et de test basé sur de la recherche à la volée (SBT - Search-Based Testing) sont utilisées. Ces techniques permettent de réduire l'explosion combinatoire durant la phase de calcul mais retournent peu d'informations au niveau de la couverture du modèle et rend difficile l'assurance de la qualité. [AMS06] propose une approche où la modélisation de l'environnement permet de définir des scénarios spécifiques permettant de valider le système sous test. Des probabilités sont associées aux événements représentés dans le modèle de l'environnement permettant de faire varier son comportement dans le but d'obtenir une génération pseudo-aléatoire des tests permettant de valider les objectifs.

Cette approche peut être difficile à mettre en place du fait de l'hétérogénéité des différents composants. Il est ainsi nécessaire de réaliser une représentation des composants à un niveau d'abstraction adéquat et uniforme pour toutes les parties. [BF04] montre par exemple que l'utilisation d'un modèle basé sur UML/SysML est efficace pour l'automatisation et pour capturer la complexité des systèmes embarqués. De plus, pour éviter l'explosion combinatoire lors de l'application de stratégies de génération de tests, il s'avère nécessaire de représenter uniquement les informations nécessaires à la simulation du comportement du système par rapport à l'évolution de son environnement.

2.3.2 Langage de modélisation

Concernant le choix du langage de modélisation, on peut dénoter deux grandes catégories. Comme présenté dans [TMJL09], la première approche propose l'utilisation d'un langage formel comme les réseaux de Petri ou VHDL-AMS. Ce type de langage est adapté à la représentation des expressions mathématiques et physiques mais ne sont pas faciles à appréhender et ne correspondent pas toujours aux connaissances initiales des ingénieurs validation issus du domaine informatique.

Afin d'obtenir un processus plus facilement utilisable par la communauté visée, la seconde approche propose d'utiliser un langage moins formel et éventuellement graphique. De plus, commencer la description d'un système en utilisant un langage trop formel n'est pas commode. Il est généralement nécessaire de commencer la représentation par un niveau de notation plus abstrait afin de maîtriser la complexité du système. Par exemple, dans [IAB10], les auteurs proposent le développement d'un modèle à partir d'un diagramme de classes pour représenter la structure générale de l'environnement (relation, propriétés et contraintes). Ensuite, des diagrammes d'états-transitions sont utilisés pour la représentation des parties dynamiques du système. D'autres approches [MBKL10] utilisent des diagrammes de séquence pour modéliser les comportements ou pour représenter les arbres de classification de test. Dans [EPML07], l'auteur propose la réalisation d'un modèle SysML pour spécifier initialement le système. Ce langage permet de capturer les différents aspects des composants du système mécatronique sous test et permet ainsi une interaction aisée entre les différentes équipes d'ingénieurs issus de domaines différents. D'une manière générale, les langages de modélisation peuvent être répertoriés en plusieurs familles. Nous présentons ici uniquement les catégories adaptées à la modélisation des systèmes embarqués.

Système de transitions étiquetées

Les systèmes de transitions étiquetées (ou LTS - Labelled Transition Systems) sont souvent liés aux algèbres de processus car ils sont employés pour décrire la sémantique de l'algèbre de processus. Un système de transition est généralement défini par un tuple $\{S, s_0, \Sigma, \delta\}$ où S dénote un ensemble d'états (potentiellement infini), s_0 est l'état initial, Σ est l'alphabet des étiquettes, et $\delta : S \times \Sigma \rightarrow S$ définit la relation de transition entre les états. Des exemples de langage de modélisation s'appuyant sur une telle sémantique sont le Π-calcul [Mil99], LOTOS [88089], SDL [EHS97] ou Promela [Hol93, Hol97].

Ce type de langage peut être étendu avec, par exemple, la définition d'entrées/sorties : on parle alors de système de transitions étiquetées entrée/sortie (ou IOLTS - Input/Output Labelled Transition Systems). Inspiré de cette dernière sémantique, un langage de spécification davantage abstrait a été inventé : il s'agit des systèmes de transitions symboliques entrée/sortie (ou IOSTS - Input/Output Symbolic Transition Systems). Ces deux types de langage se prêtent idéalement à la génération de tests et ont été la base de nombreux outils dont les plus célèbres sont certainement TGV [JJ05] et STG [CJRZ02]. Dans ce contexte, de nombreux travaux ont été menés pour caractériser les relations de conformité d'une exécution du système sous test vis-à-vis du modèle de spécifications : on peut citer par exemple la relation de conformité IOCO [Tre08].

Structure de Kripke et logique temporelle

La structure de Kripke définit un automate à états finis. Un tel modèle, représentant le comportement du système, a été initialement utilisé pour effectuer de la vérification de propriétés, également spécifiées en logique temporelle, par des techniques de model-checking [CE81, QS82]. Une fonction d'étiquetage fait correspondre à chaque état un ensemble de propositions logiques vraies dans cet état. La logique temporelle est traditionnellement utilisée dans ce type de modèle pour spécifier ces propriétés. Lorsqu'une propriété donnée est violée par l'automate, il est possible de récupérer la séquence d'états, et donc de stimuli, qui la met en défaut : on parle alors de contre-exemple.

Ce type de modèle généralement utilisé dans les processus de vérification a été utilisé plus récemment pour générer des tests. La technique consiste à prendre en entrée un automate décrivant un système et une propriété en logique temporelle pour laquelle on veut générer des tests (i.e. trouver une séquence de stimuli qui permet d'exercer cette propriété). On soumet alors la négation de la propriété logique afin de calculer un contre-exemple qui constitue le cas de test désiré [JM99].

Machine à états finis

Une machine à états finis (ou diagramme à états finis) est définie par un automate constitué d'états et de transitions qui relient ces états. L'automate est dit *fini* car il possède un nombre fini d'états distincts, ce qui le différencie des systèmes de transition évoqués plus tôt. Le comportement du diagramme est dirigé par les stimuli reçus : l'automate passe ainsi d'état en état en fonction des stimuli reçus qui conditionnent l'activation des transitions. On peut voir un tel automate comme un graphe fini orienté et étiqueté, dont les états sont les sommets et les transitions sont les arêtes étiquetées par trois éléments : un nom d'action, une garde et des actions de modification des variables d'état.

Les notations basées sur ces structures sont très populaires, dans le secteur de la recherche comme celui du monde industriel, car leur expressivité limitée les rend particulièrement faciles à traiter automatiquement et, en même temps, elles sont suffisamment expressives pour spécifier tout système discret réactif.

Parmi les langages qui exploitent ce paradigme, on peut notamment citer les machines d'état finis (FSM, EFSM) [CK93], les diagrammes d'états (Statechart) [Har98], les diagrammes d'états/transitions UML [RJB05] et SysML [FMS09b]. Les diagrammes d'états-transitions permettent une représentation complète de la vue dynamique du système et ont l'avantage d'être couplés à d'autres modèles permettant de compléter cette vue. Parmi ces langages, SysML étant dédié à la modélisation des systèmes, il est de plus en plus populaire dans le domaine de l'embarqué. SysML est alors parfois couplé à UML,

SysML servant à la modélisation du système et UML à la modélisation de la partie logiciel [BBHP08]. [HSRH10] montre que l'utilisation de modèles SysML est un atout pour la validation des systèmes embarqués du fait de son expressivité permettant une validation plus complète grâce à la prise en compte des aspects matériels en plus des aspects logiciels nativement adressés par UML. Certains travaux s'intéressent également à la vérification d'exigences de sécurité à l'aide de modèles SysML [PEML10] montrant par ce fait que SysML est un langage pertinent pour la représentation des systèmes.

Modélisation par contrat

Il existe une vaste gamme de langages s'appuyant sur des techniques de modélisation par contrat. Un tel modèle explicite de manière formelle les caractéristiques à satisfaire pour exécuter chaque opération du système (pré-condition) et ce que l'opération s'engage alors à offrir comme service (post-condition). De ce fait, ce type de langage donne rarement lieu à des modèles graphiques (automates) mais permet de produire des spécifications textuelles s'appuyant sur une notation formelle. Parmi les langages les plus connus, on peut citer Eiffel [Mey97], VDM [Jon90], Z [Spi92], B [Abr96], JML [LBR06] et OCL [WK96] .

Du fait de leur nature formelle, ces langages sont utilisés aussi bien dans des travaux de vérification que de validation. Concernant cette deuxième partie, citons par exemple l'outil LTG-B [BLPT05] qui propose, à partir d'une spécification B, la génération automatique de tests basée sur des critères de couverture des décisions et des données. Le langage JML, très présent dans le domaine de la vérification de programmes JAVA (outils JACK [BRlLV03], KRAKATOA [MPmU04]) est également usité dans le domaine de la validation avec par exemple la validation d'applications JAVA/JML par la génération de tests [Dad06]. Les expressions JML sont extraites afin de construire un modèle pouvant être animé symboliquement afin d'extraire les séquences de test. Au sein de ce même paradigme, OCL est un langage souvent associé aux modèles UML afin de préciser les comportements de manière plus formelle en spécifiant des contraintes telles que des pré- et post-conditions.

Les outils tels que Test Designer^{TM} [BGLP08] ou UML-CASTING [vAJ03] utilisent des modèles UML/OCL comme données d'entrée à la génération de tests. Dans UML-CASTING, l'utilisateur sélectionne les décompositions à appliquer au niveau des expressions OCL définissant les objectifs de couverture. Concernant Test Designer^{TM}, OCL permet de préciser les comportements associés aux opérations et aux transitions. La stratégie appliquée lors de la génération de tests assure la couverture de toutes les transitions représentées par les diagrammes d'états-transitions, ainsi que toutes les décisions et les conditions modélisées à l'aide d'OCL.

Langages de description de matériels

Plusieurs langages permettent la description et la programmation des systèmes embarqués réactifs. Les modèles de flots de données (ou Dataflow models) décrivent le système à travers les vecteurs de données qui lui sont soumis et qu'il émet en réaction. Les systèmes modélisés sont considérés comme des boîtes noires avec des entrées et des sorties régies par une exécution synchrone. Les langages de flots de données sont bien adaptés pour les systèmes réactifs synchrones et sont mis en œuvre, le plus souvent, pour modéliser des systèmes critiques de sûreté. Les langages les plus connus sont ESTEREL [BG92] et LUSTRE [CPHP87]. D'autres langages semblables tels que VHDL [Ash01], Verilog [Pal03] ou SystemC [MSGL02] permettent la description des systèmes matériels. Ces langages ont l'avantage de permettre la simulation des systèmes électroniques grâce à une description suffisamment complète des réactions du système par rapport à des événements précis (prise en compte de la concurrence...).

Plusieurs plateformes utilisent ce type de langage. L'environnement de développement SCADE [3] est basé sur le langage LUSTRE et est spécialisé dans la conception de logiciels critiques pour l'aéronautique, le ferroviaire et les centrales nucléaires. Cet outil est comparable au logiciel Simulink intégré à la plateforme Matlab [4]. SCADE et Simulink permettent la modélisation de systèmes physiques complexes et leur simulation. Les modèles réalisés à l'aide de Simulink peuvent ainsi servir à la validation de systèmes embarqués [BK08]. L'outil de génération de tests *GATeL* [MA00] se base sur des spécifications LUSTRE. La stratégie utilisée consiste à décrire précisément des états du système à atteindre (représentés à l'aide de contraintes exprimées également en LUSTRE). Le langage ESTEREL est utilisé afin d'appuyer la validation de système électronique en permettant de préciser des comportements grâce à son expressivité [ABB+99]. L'outil de génération de tests *conformance kit* [MRS+97] présenté précédemment utilise le langage VHDL pour la modélisation du système.

3. http ://www.esterel-technologies.com/products/scade-suite/
4. http ://www.mathworks.fr/

Bilan

Les structures de Kripke et les LTS conviennent parfaitement aux processus de vérification car elles proposent une notation simple et formelle. Ce formalisme permet l'expression de propriétés strictes mais souffre d'une diversité limitée au niveau des entités manipulées. Les LTS ont l'avantage et l'inconvénient d'être des automates potentiellement non-finis, c'est-à-dire de pouvoir contenir un nombre d'états potentiellement infini.

Les machines à états finis tels que les diagrammes d'états-transitions permettent de représenter des automates finis de manière plus détaillée que les structures de Kripke. La diversité des entités manipulables et la présence de structures complexes de haut niveau (états composites...) permettent d'attribuer des sémantiques spécifiques aux éléments modélisés. Les machines à états finis sont généralement représentées sous forme graphique, ce qui peut être un atout dans la mise en place d'un processus industriel.

L'inconvénient de ce type de langage est le manque de sémantique qui peut toutefois être fixé par l'utilisation d'un langage par contrat tel qu'OCL. Ces langages permettent la représentation de pré- et post-conditions pour définir précisement les caractéristiques des comportements modélisés. Du fait de leur formalisme et de leur expressivité suffisament complets, ces langages sont utilisés pour mettre en place des processus rigoureux et automatiques. Les langages de modélisation de matériels tels que LUSTRE et VHDL ont pour objectif la description de matériel et répondent donc à des problématiques différentes des langages présentés précédemment. En effet, il s'agit de langages de bas niveau permettant de représenter une vue concrète du système. Ce type de langage est couramment utilisé dans le domaine de la simulation car ils servent à l'implémentation des systèmes. Dans le cadre des travaux présentés dans ce mémoire, le langage SysML a été choisi car il permet une représentation générale du système sous test et de son environnement. Afin de préciser certains comportements, l'utilisation d'OCL s'est imposée de part sa compatibilité avec les modèles SysML et de part son expressivité permettant de compléter l'expressivité du langage SysML et de donner une sémantique précise. L'inconvénient de SysML est qu'il ne propose pas de structure permettant une représentation spécifique du temps qui est un élément redondant dans le domaine des systèmes embarqués.

2.3.3 Représentation du temps

Les systèmes embarqués se définissent par un composant en interaction continue avec son environnement. Le temps fait donc partie intégrante du fonctionnement d'un système embarqué. Certains langages de modélisation présentés précédemment sont adaptés à la modélisation de telles contraintes.

Tout d'abord, la logique temporelle associée aux structures de Kripke permet la représentation de contraintes de successions d'opérations. La représentation est cependant limitée et ne permet pas de quantifier le temps entre deux actions par exemple.

Les langages LUSTRE et ESTEREL sont des langages synchrones et réactifs. Ils permettent donc une synchronisation des événements et la représentation de contraintes temporelles complexes. Il en est de même pour VHDL qui propose une prise en compte totale du temps par la représentation de contraintes.

Concernant le langage SysML dédié à la modélisation des systèmes, les éléments permettant de modéliser le plus précisément les effets de l'écoulement du temps sont les diagrammes paramétriques (utilisation de descriptions mathématiques avancées) et les diagrammes de séquence (description graphique du séquencement des exécutions dans le temps). En complément de ces structures natives à SysML, il existe des langages ou annotations dédiés à l'expression de contraintes de type temps réel comme les profils UML TURTLE [AdSSL+01] ou MARTE [Gro07].

TURTLE (*Timed UML and RT-LOTOS Environment*) ajoute à UML une sémantique formelle dédiée à l'expression du temps réel. Ce profil étend ainsi UML de plusieurs manières. Tout d'abord, les classes UML se voient dotées de portes de communication par rendez-vous. Ce format permet de représenter des informations de parallélisme, de séquences et de synchronisation entre différentes tâches. Les diagrammes d'activité, quant à eux, se voient dotés d'opérateurs temporels permettant de décrire plusieurs types de délais. Enfin, ce profil introduit l'utilisation de RT-LOTOS qui permet l'expression de contraintes temporelles. TURTLE trouve son champ d'application principalement dans le domaine de la validation d'architectures de communication et de systèmes embarqués.

MARTE (*Modeling and Analysis of Real Time and Embedded systems*) est un profil normalisé par l'OMG. Ce profil est dédié à la représentation du temps réel au sein des systèmes embarqués. Le profil MARTE étend UML en terme de stéréotypes permettant d'inclure, par exemple, des horloges (stéréotype `Clock`) et des événements liés aux horloges (stéréotype `TimedEvent`), chaque stéréotype fournissant des propriétés propres au temps (répétition, délais...). Les profils SysML et MARTE sont compatibles et tout à fait complémentaires pour la modélisation de systèmes embarqués à différents niveaux d'abstraction [ECSG09].

2.4 Synthèse

La validation des systèmes embarqués est un problème complexe toujours d'actualité. Parmi les techniques de validation présentées dans ce mémoire, nous avons choisi de travailler sur la génération de tests à partir de modèles afin de proposer une chaîne outillée simple à manipuler et permettant d'assurer une qualité mesurable.

Le choix de cette technique de validation implique donc la nécessité de représenter le système sous test à l'aide d'un modèle. Tous les types de langages présentés dans cette section permettent la représentation des comportements d'un système embarqué. Certains sont plus adaptés que d'autres compte tenu de la chaîne outillée que l'on souhaite établir, le modèle initial devant être exprimé dans un langage suffisamment complet pour exprimer correctement et de manière complète les comportements du système tout en ayant une syntaxe accessible aux ingénieurs validation pour en faire un outil utilisable en contexte industriel. Les machines à états finis, en l'occurrence les diagrammes d'états-transitions, constituent une solution adaptée. Cependant, l'expressivité limitée de ce genre de langage risque de nuire au niveau de la qualité de système que l'on pourra garantir. Il s'est avéré plus pertinent d'exprimer certains comportements à l'aide d'un langage plus précis. OCL a été choisi du fait de sa parfaite compatibilité et de sa courante utilisation au sein des diagrammes d'états-transitions UML, permettant ainsi d'améliorer l'expressivité d'un langage facilement manipulable. Le langage SysML, variante du langage UML, a été sélectionné car il propose les entités sémantiques nécessaires à une représentation complète de la structure et des comportements d'un système embarqué. La représentation des contraintes de temps étant limitée au niveau de SysML, une perspective de prise en compte du profil MARTE pour cette partie est envisagée mais n'a pas été définie pour l'instant.

Finalement, concernant la stratégie de génération de tests, nous proposons d'utiliser les techniques basées sur les critères de couverture structurelle. En effet, ces techniques permettent d'assurer, pour une représentation du système donnée, une qualité précise et reproductible quel que soit le système et l'ingénieur en charge de la validation au contraire des critères basés sur la couverture des exigences par exemple. Dans ce mémoire, nous présentons l'élaboration d'une nouvelle stratégie basée sur de nouveaux critères de type flot de données couplés a un critère de type flot de contrôle (D/CC).

Deuxième partie

Contributions

Chapitre 3

Modélisation en SysML pour le test

Sommaire

Comme nous venons de le voir, nous avons décidé d'utiliser le langage SysML. Ce langage vient de l'INCOSE (*organisation internationale de l'ingénierie système*) qui, pour répondre plus spécifiquement aux caractéristiques métier des applications de l'ingénierie système, initie au début des années 2000 des travaux afin de définir un langage, basé sur UML, adapté et totalement compatible avec ce domaine. En 2006, le *SYStems Modeling Language* (ou SysML) est officialisé du fait de sa normalisation par l'OMG.

Le but de notre approche est la génération de tests à partir de modèles SysML. La première étape consiste à étudier le langage SysML. Ensuite, nous présentons l'ensemble des éléments de SysML pris en compte dans notre démarche. Par analogie avec UML4MBT présenté en partie 1.2, cet ensemble est appelé SysML4MBT. Après une comparaison entre SysML4MBT et UML4MBT, nous donnons un cadre formel pour SysML4MBT puis, finalement, nous présentons la modélisation de l'exemple fil rouge (cf. partie 1.5), à l'aide de ces éléments.

3.1 Présentation du langage SysML

SysML est un langage de modélisation basé sur UML et normalisé par l'OMG en juillet 2006. La version 1.0 a vu le jour en septembre 2007 et nous en sommes actuellement à la version 1.2 (depuis juin 2010). SysML prend la forme d'un profil UML qui redéfinit la sémantique et la forme de certains éléments UML tout en définissant de nouveaux concepts et de nouveaux éléments (cf. figure 3.1).

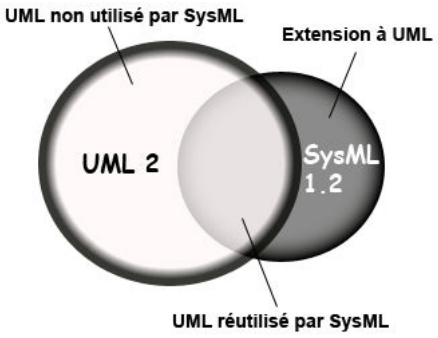


FIGURE 3.1 – Recouvrement des langages SysML et UML2

Tout comme le langage UML, SysML permet la spécification, l'analyse, la conception, la vérification et la validation de nombreux systèmes. Cependant, SysML est spécifiquement dédié aux processus et caractéristiques métier mis en œuvre pour modéliser les systèmes comme ceux issus des domaines de l'ingénierie automobile, de l'ingénierie acoustique et de l'aéronautique. Concrètement, ce standard reprend 7 des 13 diagrammes d'UML 2.0 (parmi ceux-ci, trois sont modifiés) et comporte deux nouveaux types de diagrammes. Ces diagrammes peuvent être regroupés dans trois catégories. Tout d'abord, le diagramme d'exigences, seul dans sa catégorie, permet de représenter les exigences du système. Ensuite, on trouve quatre types de diagrammes dits *structurels*. Il permettent de donner une vue statique du système. Et enfin, quatre diagrammes comportementaux qui permettent de représenter la partie dynamique du système. La figure 3.2 montre le positionnement de chacun des diagrammes SysML par rapport à UML2. Certains diagrammes UML sont repris à l'identique en SysML. Ils sont représenté par les cases blanches à encadrement continu. Ensuite, certains types de diagrammes existent dans les deux langages mais ont été modifiés pour SysML. Il s'agit des carrés en pointillés. Et enfin, les diagrammes inédits dans SysML sont représentés par les carrés foncés.

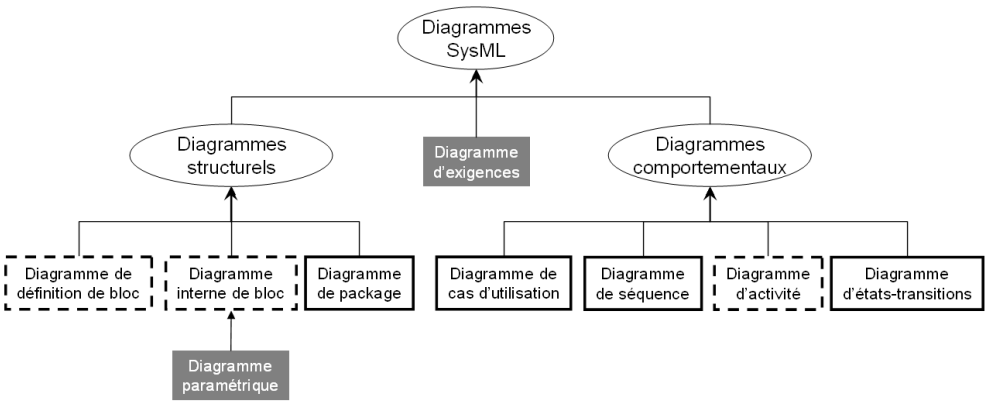


FIGURE 3.2 – Diagrammes SysML et liens avec UML

Nous allons maintenant présenter un aperçu de chacun des diagrammes SysML. Tout d'abord, la partie 3.1.1 présente les diagrammes structurels. Ensuite, la partie 3.1.2 introduit les diagrammes comportementaux. Enfin, un aperçu du diagramme d'exigences est donné dans la partie 3.1.3.

3.1.1 Les diagrammes structurels

Les diagrammes structurels permettent la représentation d'une vue statique du système. SysML en propose quatre : le diagramme de définition de bloc, le diagramme interne de bloc, le diagramme paramétrique ainsi que le diagramme de package. Alors que les deux

premiers sont basés sur des diagrammes UML existants, le diagramme paramétrique est inédit. Le diagramme de package est quant à lui identique en UML2 et en SysML. Cette section les présente succinctement dans cet ordre.

Diagramme de définition de bloc (Block Definition Diagram)

Le diagramme de bloc est dérivé du diagramme de classes UML. A son instar, il permet la représentation de la hiérarchie des blocs du système et une vue boîte noire de chacun d'entre eux. Un bloc représente une partie du système, logicielle ou matérielle. On retrouve dans ce type de diagramme les mêmes concepts que ceux utilisés dans le diagramme de classes UML tels que les associations, les compositions, les attributs (appelés propriétés en SysML), les ports ou encore les opérations. On note l'introduction en SysML d'un nouveau type de ports : le flow port (port de flux). Il permet de représenter les échanges de toutes sortes (signaux électriques, matière...) entre les blocs en complément des ports UML permettant la représentation des services requis ou offerts par les composants.

Le concept UML de classes est enrichi en SysML par le concept de blocs afin d'adapter le pouvoir d'expression de SysML au domaine de l'Ingénierie Système. Ainsi, là où une classe UML représentait une partie logicielle, un bloc SysML englobe les concepts de logiciel, de matériel, de données et également de processus.

Diagramme interne de bloc (Internal Block Diagram)

Le diagramme interne de bloc est dérivé du diagramme de structure composite UML. Il permet de représenter les liens de communications entre les différentes parties du système. Il permet aussi bien de représenter des échanges d'informations logiciels (structures de données) que des échanges matériels (matière ou énergie par exemple). Un diagramme interne est attaché à un bloc, les parties qui composent ce bloc sont instanciées et assemblées par des connecteurs, au travers des ports. Comparé à UML2, ce type de diagramme ajoute entre autres, la prise en compte des ports de flux.

Diagramme paramétrique (Parametric Diagram)

Inédit en SysML, ce diagramme sert à modéliser les contraintes physiques et quantitatives du système. Il représente les contraintes qui s'appliquent sur les paramètres physiques du système. On utilise pour cela des blocs de contraintes qui représentent des expressions mathématiques dont chaque paramètre peut faire référence à un élément du modèle (propriété de bloc par exemple).

Diagramme de package (Package Diagram)

Déjà présent de manière identique en UML, le diagramme de package montre l'organisation générale du modèle. Il permet une représentation très simple des packages utilisés lors de la modélisation et des différents liens qui existent entre eux.

3.1.2 Les diagrammes comportementaux

Les diagrammes comportementaux permettent la représentation de la vue dynamique du système. En SysML, il existe quatre types de diagrammes comportementaux : le diagramme de cas d'utilisation, le diagramme de séquence, le diagramme d'activité ainsi que le diagramme d'états-transitions. Excepté le diagramme d'activité qui a été étendu par rapport à la version UML, ces diagrammes sont directement issus de la notation UML. Ils sont brièvement présentés dans cet ordre dans cette partie.

Diagramme de cas d'utilisation (Use Case diagram)

Les diagrammes de cas d'utilisation permettent de représenter les services que le système doit fournir aux utilisateurs (humains ou machines). Le diagramme de cas d'utilisation permet de représenter, à l'aide d'un langage simple, les fonctionnalités et le fonctionnement global d'un système.

Diagramme de séquence (Sequence diagram)

On utilise le diagramme de séquence pour représenter l'ordre d'exécution des actions représentées dans le diagramme de cas d'utilisation. Le diagramme de séquence modélise la chronologie des interactions entre les éléments du système ou entre le système et son environnement. Afin de modéliser un scénario (une séquence), chacun des acteurs est représenté à l'aide de lignes de vie. Chacune de ces lignes contient des actions qui sont reliées entre elles afin de décrire le déroulement du scénario.

Diagramme d'activité (Activity diagram)

Le diagramme d'activité modélise, sous forme d'organigrammes, les flux d'informations et les flux d'activités du système. Il permet de décrire graphiquement un processus (ou le déroulement d'un cas d'utilisation) en termes de suites d'activités. Il permet ainsi de spécifier, construire ou documenter le comportement de n'importe quel élément de modélisation. Les nœuds d'activités représentent les étapes des processus et ils sont reliés entre eux à l'aide de transitions et de nœuds spécifiques (nœud de bifurcation...).

Le diagramme d'activité a été étendu en SysML afin de pouvoir supporter les systèmes continus en spécifiant la nature du débit sur les flots (continu ou discret). Les diagrammes d'activité SysML proposent également les notions de taux et de probabilité sur les flux de contrôle ou d'objets.

Diagramme d'états-transitions (State machine diagram)

Le diagramme d'états-transitions permet de représenter le comportement qui doit être respecté par le bloc qui le contient. Il modélise l'ensemble des états possibles de la composante concernée. La représentation se fait suivant des états et décrit les changements entre ces états à l'aide d'un automate d'états finis.

3.1.3 Le diagramme d'exigences

Le diagramme d'exigences est un type de diagramme inexistant en UML. Il permet de collecter et d'organiser toutes les exigences du système. Il peut être utilisé pour modéliser les exigences techniques, légales, physiques, commerciales, normatives ou autres d'un projet. Les exigences ont pour but d'assurer l'adéquation de la solution (le système réalisé) avec les besoins. Ce diagramme permet d'organiser de manière structurée et hiérarchisée ces exigences. Chaque exigence est représentée sous forme textuelle. Ce diagramme a pour particularité d'être transversal au modèle. En effet, il est possible de relier une exigence à n'importe quel élément de modélisation. Le lien `satisfy` permet de représenter le fait qu'un élément de modèle permet de satisfaire une exigence. Le lien `verify`, quant à lui, représente le fait qu'une exigence est vérifiée par un élément de modèle. On peut mettre en place une hiérarchie entre les exigences à l'aide des liens `containment` (composition) ou `derive` (une exigence dérive d'une autre). Enfin, le lien `refine` permet de préciser une exigence (à l'aide d'un cas d'utilisation par exemple).

3.2 SysML pour le test (SysML4MBT)

Comme expliqué en section 1.2, nous nous basons sur des travaux proposant la génération de tests à partir d'un sous-ensemble d'UML, appelé UML4MBT, qui définit les éléments nécessaires à la génération de tests. A l'instar de ces travaux, nous avons défini un sous-ensemble de SysML, appelé SysML4MBT, permettant d'adresser la problématique de génération de tests pour les systèmes embarqués. Ainsi, SysML4MBT définit un sous-ensemble de SysML qui permet de représenter un modèle suffisamment complet,

précis et interprétable afin de spécifier les comportements de systèmes embarqués en vue de générer des tests. Les diagrammes contenus dans SysML4MBT sont :

- le diagramme de bloc qui permet de représenter une vue statique du système et de son environnement,
- le diagramme interne de bloc afin de préciser les interconnexions entre les différentes parties modélisées dans le diagramme de bloc,
- le diagramme d'états-transitions pour la représentation de la vue dynamique du système,
- et enfin le diagramme d'exigences afin d'exprimer les exigences fonctionnelles et de les relier avec les éléments de modèles qui permettent de les satisfaire.

Certains éléments de ces diagrammes pourront contenir des annotations OCL afin de préciser formellement les comportements. Chacun de ces éléments est présenté brièvement dans cette section. La dernière sous-section détaille la sémantique opérationnelle et les contraintes posées sur les modèles SysML4MBT.

3.2.1 Diagramme de bloc SysML4MBT

Le diagramme de bloc permet de représenter une vue statique du système. La majorité des éléments du diagramme de bloc SysML sont autorisés en SysML4MBT. Chacune des sous-parties suivantes expose le fonctionnement des éléments pris en compte et leur sémantique.

Les blocs

Un système se décompose en plusieurs parties ou composantes. Chacune d'entre elles est ainsi modélisée par un bloc (cf. figure 3.3).

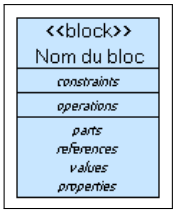


FIGURE 3.3 – Représentation graphique d'un bloc SysML4MBT

Les propriétés

Chaque bloc peut contenir des propriétés : il s'agit de ses variables d'état. Les propriétés sont représentées par un nom, un type et une valeur par défaut. Cette valeur par défaut représente la valeur à l'initialisation du système. Les propriétés représentent, au travers des différents blocs auxquels elles sont attachées, les variables d'état du système et les informations stockées dans le système. Comme pour les attributs de classes UML4MBT, les seuls types autorisés pour les propriétés de blocs SysML4MBT sont les entiers, les booléens et les énumérations.

Les opérations

On peut également attacher à un bloc une ou plusieurs opérations. Une opération représente une action que peut exécuter ce bloc. Elles sont équivalentes aux opérations UML4MBT.

Les associations

Afin de relier les blocs entre eux, on utilise des associations. La composition (non présente en UML4MBT) est également prise en compte dans SysML4MBT. L'association standard représente un lien quelconque, un simple rapport entre des éléments. La composition, quant à elle, représente une imbrication, le fait qu'un élément se découpe en plusieurs composants. La figure 3.4 montre la représentation graphique d'une association et d'une composition. Sur cette figure, le bloc A est composé de 4 ou 5 blocs B.

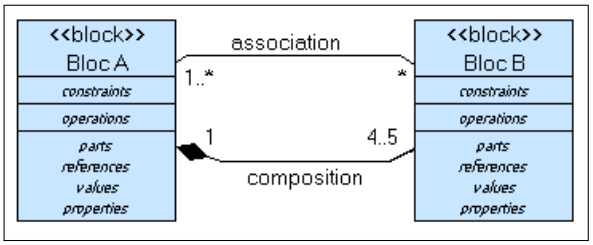


FIGURE 3.4 – Représentation graphique des associations SysML4MBT

Les signaux

Le diagramme de bloc peut également contenir des signaux. Un signal représente un élément pouvant être transmis d'un bloc à l'autre. En comparaison avec les opérations, les signaux représentent un stimulus alors que les opérations représentent des services proposés par les blocs. Un signal se représente dans un diagramme de bloc comme des classes UML stéréotypées (cf. figure 3.5).

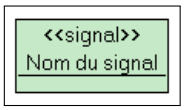


FIGURE 3.5 – Représentation graphique d'un signal SysML4MBT

Les spécifications de flux

Les spécifications de flux sont des interfaces (stéréotypes des interfaces UML). Elles permettent de préciser l'ensemble des éléments qui peuvent être reçus/envoyés par un port. Une spécification de flux regroupe plusieurs propriétés, chacune représentant un élément. Un sens (in, out ou inout) est attribué à chaque propriété. Ainsi, par exemple, un port typé par la spécification de flux `Specification1` représentée sur la figure 3.6 peut recevoir les signaux `Signal1` et `Signal2` et seul le signal `Signal1` peut être envoyé à travers ce port.

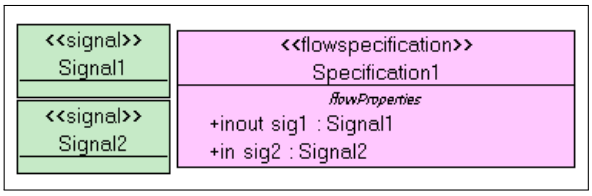


FIGURE 3.6 – Représentation graphique d'une spécification de flux SysML4MBT

Les ports

Les ports précisent les points de communication d'un bloc. On les définit au niveau du diagramme de bloc sans les représenter graphiquement. Un port est défini par un nom et un type. Il existe deux types de ports en SysML : le port standard et le port de flux. Cependant, en SysML4MBT, nous avons décidé d'exploiter uniquement le port de flux. Un port de flux permet de représenter un échange de matière, d'énergie ou d'information. Il est possible de préciser le sens de l'échange (in, out ou in/out). En SysML4MBT, un port peut être typé par un booléen, un entier, une énumération, un signal ou une spécification de flux.

3.2.2 Diagramme interne de bloc SysML4MBT

Afin de préciser la représentation statique du système, les diagrammes internes de bloc sont utilisés. Ce type de diagramme permet de représenter l'intérieur du système du point de vue des communications et des envois de signaux.

Les propriétés

Les principaux éléments d'un diagramme interne de bloc sont les propriétés (figure 3.7). Une propriété dans le diagramme interne d'un bloc donné peut représenter :

- une propriété du bloc en question,
- un autre bloc qui compose le bloc en question (dans ce cas, par convention, on lui donnera comme nom son nom de rôle dans la composition).

Dans ce deuxième cas, on peut représenter des propriétés à l'intérieur de cette propriété pour représenter un niveau supplémentaire. Chaque propriété a également un type qui correspond à l'élément qu'elle représente.

FIGURE 3.7 – Représentation graphique d'une propriété du diagramme interne de bloc SysML4MBT

Les ports

Les ports définis dans le diagramme de bloc sont représentés dans le diagramme interne de bloc. Les ports du bloc dont on est en train de représenter la structure sont représentés sur les parois du diagramme interne. En revanche, les ports des blocs représentés sous forme de propriétés sont placés sur les parois des propriétés. La représentation graphique des ports est visible sur la figure 3.8. Tous les ports typés par une spécification de flux ont la même représentation. La représentation des autres ports (typés par un booléen, un entier, une énumération ou un signal) dépend de leurs sens.

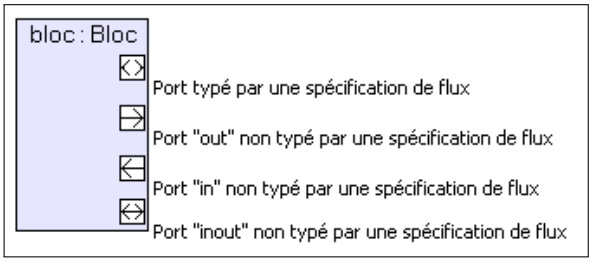


FIGURE 3.8 – Représentation graphique des ports du diagramme interne SysML4MBT

Les connecteurs

Les connecteurs représentent les chemins de communication entre les éléments. Le connecteur représenté sur la figure 3.9 représente un chemin de communication du port `P1` du bloc `Property1` vers le port `P2` du bloc `Property2`. Seul le signal `Signal1` peut transiter sur ce connecteur.

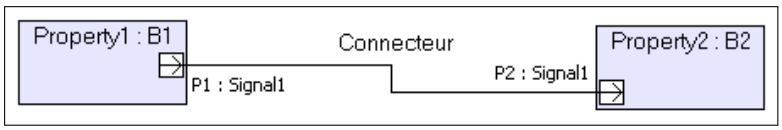


FIGURE 3.9 – Représentation graphique d'un connecteur SysML4MBT

3.2.3 Diagramme d'états-transitions

Le diagramme d'états-transitions représente la vue dynamique du système. Cette section présente l'ensemble des éléments pris en compte en SysML4MBT au niveau de ce type de diagramme. En SysML4MBT, outre les transitions, les structures autorisées sont l'état simple, l'état final, l'état composite, l'état parallèle (tous représentés sur la figure 3.10) ainsi que les pseudos-états tels l'état initial, le point de choix, la barre de fraction et la barre de jonction, l'état historique et l'état historique profond (figure 3.11). Les pseudos-états sont des états temporaires c'est-à-dire dans lesquels le système ne sera pas amené à stationner.

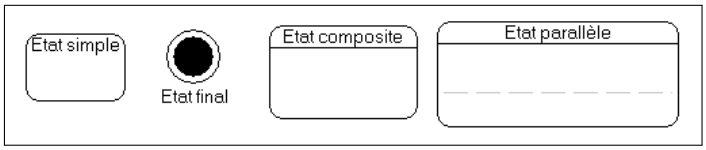


FIGURE 3.10 – Représentation graphique des états SysML4MBT

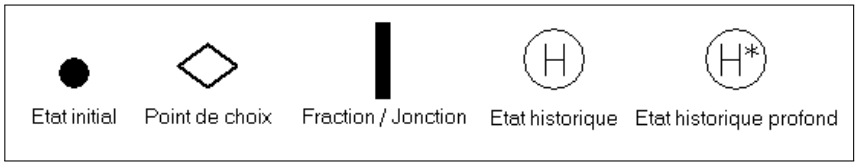


FIGURE 3.11 – Représentation graphique des pseudo-états SysML4MBT

L'état initial

L'état initial désigne l'état actif à l'initialisation du modèle (ou de l'élément qui le contient).

L'état simple

Un état simple représente un état du système, une configuration particulière des variables du système ou encore le statut du système. Il est possible de préciser à l'aide des paramètres onEntry et onExit les effets qui se produisent à l'entrée et à la sortie de l'état. Ces paramètres sont également disponibles pour les états composites et pour les états parallèles.

L'état composite

L'état composite (ou super-état) est un état qui peut en contenir d'autres, il s'agit d'un regroupement d'états. Il permet de hiérarchiser les états du système.

L'état parallèle

Les états parallèles permettent de représenter des exécutions en parallèle. Un état parallèle contient des régions qui peuvent contenir des états et des transitions.

L'état final

L'état final est un point de sortie du système. Il désigne l'état de fin d'exécution du système.

Les transitions

Les transitions relient les états entre eux. La représentation graphique est une flèche simple. Une transition est définie par trois paramètres optionnels :

- Le déclencheur précise un évènement qui, lorsqu'il se produit, permet de franchir la transition. Le déclencheur peut être la réception d'appel d'une opération ou la réception d'un signal. Dans ce dernier cas, le port de réception est précisé.
- La garde est une expression booléenne qui, si elle n'est pas respectée, bien que le déclencheur ait été activé, empêchera l'exécution de la transition.
- L'effet d'une transition est une expression qui sera exécutée si le déclencheur est activé et que la garde est respectée, c'est-à-dire lorsque la transition est franchie.

Le point de décision

Le point de décision est la modélisation d'une alternative au franchissement d'une transition. Une transition arrive sur un point de décision (ou point de choix) et plusieurs transitions alternatives en repartent.

La barre de fraction et la barre de jonction

Les barres de fraction et de jonction permettent de mettre en place une parallélisation de transitions. La sémantique est semblable à celle de l'état parallèle. Les chemins représentés entre une barre de fraction et une barre de jonction sont concurrents.

L'état historique et l'état historique profond

Les états historiques sont des pseudos-états qui représentent des états mémoires. Un état historique mémorise le dernier état parcouru de l'état composite qui le contient. L'état historique profond aura la particularité de prendre en compte tous les états de l'état composite, quel que soit leur niveau d'imbrication, alors que l'état historique simple ne considère que les états directements contenus dans l'état composite.

3.2.4 OCL

A l'instar d'UML4MBT, certains éléments peuvent être annotés par des expressions OCL. Celles-ci permettent de préciser le comportement des éléments concernés. Au niveau du diagramme de bloc, OCL permet de préciser les pré et post-conditions des opérations modélisées au sein des blocs. Dans le cadre des diagrammes d'états-transitions, les expressions OCL4MBT peuvent être utilisées au niveau :

- des éléments onEntry et onExit attachés aux états,
- de la garde des transitions,
- de l'effet des transitions.

Dans nos travaux, nous utilisons un sous-ensemble d'OCL, OCL4MBT, également utilisé au sein d'UML4MBT (cf. section 1.2). Dans le cadre de nos travaux sur SysML, nous ajoutons simplement à cet ensemble l'opérateur noté ^ qui permet de représenter un envoi de signal. Son fonctionnement est le suivant :

```
Bloc.Port^SignalEnvoyé()
```

où `Bloc` représente le bloc qui contient `Port` identifiant le port vers lequel on envoie le signal, et `SignalEnvoyé` identifiant le nom du signal (défini dans le diagramme de bloc) envoyé.

3.2.5 Représentation des exigences

Les exigences présentes dans le cahier des charges se modélisent à l'aide d'un diagramme d'exigences.

Exigence

Chaque exigence du diagramme d'exigences est définie par un descriptif et un identifiant. Elle est représentée graphiquement comme sur la figure 3.12.

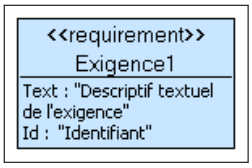


FIGURE 3.12 – Représentation graphique d'une exigence SysML4MBT

Hiérarchie

Il est possible de mettre en place une hiérarchie entre les exigences. Pour modéliser le fait qu'une exigence peut se découper en plusieurs sous-exigences, on utilise des liens intitulés *derive requirement* représentés graphiquement par un trait en pointillé intitulé `deriveReqt`. La figure 3.13 représente ce lien. La signification est, pour cet exemple, que l'exigence 1 est en fait l'union de l'exigence 2 et de l'exigence 3.

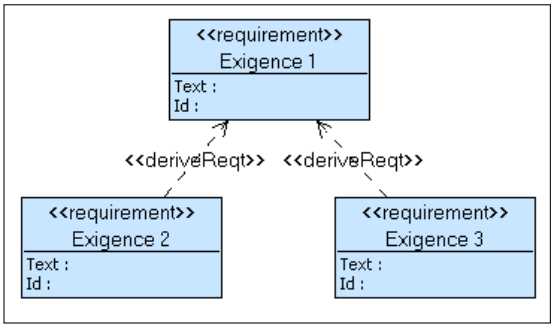


FIGURE 3.13 – Représentation graphique du lien `deriveReqt` SysML4MBT

Satisfaction

Il est possible de relier les éléments de modèles avec les exigences qu'ils doivent satisfaire. Pour ce faire, le lien `satisfy` représenté sur la figure 3.14 est utilisé. Par exemple, cette figure représente le fait que l'état `State 1` satisfait l'exigence `REQ`.

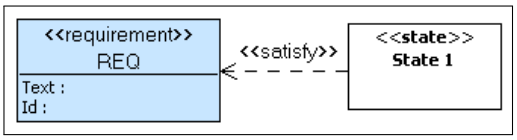


FIGURE 3.14 – Représentation graphique du lien `satisfy` SysML4MBT

On peut ainsi relier les exigences avec les éléments de modèles suivants :
- Diagramme de bloc : bloc, propriété, opération, signal, port, énumération, littéral d'énumération, spécification de flux et propriété de spécification de flux.
- Diagramme interne de bloc : propriétés et connecteurs.
- Diagramme d'états-transitions : machine complète, état, pseudo-état, transitions, déclencheur d'une transition, garde d'une transition et effet d'une transition.

3.2.6 Sémantique opérationnelle et contraintes

SysML4MBT étant basé sur SysML qui est un profil UML, SysML4MBT respecte les contraintes de modélisation fixées par les normes OMG d'UML et de SysML. SysML4MBT peut ainsi être considéré tel un profil d'UML4MBT. Ainsi, les contraintes de modélisation qui s'appliquent sur UML4MBT sont transposées aux modèles SysML4MBT. Nous définissons également certaines contraintes de modélisation propres à SysML4MBT dans l'optique de définir un langage de modélisation adapté à la modélisation des systèmes embarqués dans le cadre du MBT.

Cette partie regroupe toutes les contraintes de modélisation, celles héritées des normes OCL, celles héritées d'UML4MBT et celles mises en place au cours de ces travaux. On précise également dans cette partie la sémantique opérationnelle des éléments pris en compte.

Diagrammes

Le seul diagramme obligatoire pour la réalisation d'un modèle SysML4MBT est le diagramme de bloc. Il est nécessaire d'avoir au moins la représentation de la structure du système. Le diagramme interne permet d'avoir une vue des connections entre les parties du système modélisé. Il est possible d'en réaliser plusieurs afin de pouvoir détailler toutes les interconnexions. Les diagrammes d'états-transitions sont également facultatifs. En effet, dans certains cas, il est possible de modéliser les comportements du système uniquement à l'aide des opérations. Au contraire d'UML4MBT, il est possible d'avoir plus d'un diagramme d'états-transitions. Enfin, plusieurs diagrammes d'exigences sont autorisés pour la représentation des exigences.

Diagramme d'états-transitions

Chaque diagramme d'états-transitions n'autorise qu'un seul état initial (contrainte UML) et une seule transition doit en partir (contrainte UML4MBT) afin d'éviter tout indéterminisme. Lorsque le système se trouve dans l'état initial, la transition qui en part doit pouvoir être franchie directement (absence de garde) étant donné que l'état initial est un pseudo-état (le système ne peut donc pas stationner sur l'état initial). Au niveau des états finaux, il peut y en avoir plusieurs mais aucune transition ne peut en partir (contrainte de modélisation UML).

Transition

En SysML4MBT, une transition peut :

- relier simplement deux états,
- être réflexive (l'état de départ est le même que celui d'arrivée),
- être interne (on ne réalise pas de sortie d'état),
- pointer sur un super-état (dans ce cas, lors du franchissement de la transition, on se positionne dans l'état initial du super-état),
- partir d'un super-état (dans ce cas, la transition est franchissable depuis n'importe quel état du super-état).

La restriction selon laquelle une transition ne peut pas relier deux états qui ne sont pas dans le même super-état (transition trans-hiérarchique) est reprise d'UML4MBT afin de simplifier l'interprétation des modèles. Graphiquement, une transition ne peut donc pas traverser la bordure d'un super état. Une transition représente l'intégralité du changement d'état. Au niveau opérationnel, il existe deux types de transitions :

- Les transitions non-automatiques : elles sont déclenchées par un appel d'opération. Une opération représente une action effectuée par l'utilisateur ou à partir de l'environnement du système modélisé. Il faut donc que l'opération soit appelée et que la garde soit vraie pour que la transition soit franchie.
- Les transitions automatiques : il s'agit des transitions qui sont automatiquement franchies du fait de l'absence de déclencheur ou déclenchée par la réception d'un signal. En effet, ce deuxième type de transition est également considéré comme automatique car elles sont franchies automatiquement du moment que le signal correspondant est en attente (si tant est que la garde soit vraie).

Les transitions automatiques sont prioritaires sur les transitions non-automatiques. Lorsqu'un signal est émis par le franchissement d'une transition, il est en attente de réception tant qu'aucune transition non-automatique n'est franchie ou tant qu'il n'est pas consommé

par le franchissement d'une transition déclenchée par la réception du signal en question. La garde et l'effet d'une transition sont spécifiés en utilisant OCL4MBT (cf. partie 1.2).

Etat composite

Etant donné qu'une transition ne peut pas être trans-hiérarchique, un état composite contient obligatoirement un état initial. Ainsi, lorsque l'on arrive sur la paroi d'un état composite, on arrive dans son état initial. Si l'on franchit une transition qui quitte l'état composite, on quitte celui-ci quelle que soit sa situation. Les transitions contenues dans l'état composite sont prioritaires par rapport à celles qui partent de la bordure de l'état composite.

Etat parallèle

Comme pour l'état composite, chaque région d'un état parallèle doit contenir un état initial. Lorsque l'on franchit une transition qui pointe sur un état parallèle, toutes les régions sont activées en même temps (positionnement dans l'état initial de chaque région). Ensuite, les régions évoluent indépendamment les unes des autres. Un état parallèle possède également la même propriété qu'un état composite c'est-à-dire que si une transition qui part de l'état parallèle est déclenchée, on quitte celui-ci quelle que soit la situation. L'ordre de priorité des transitions est identique à celui appliqué pour les états composites. La figure 3.15 représente un exemple d'utilisation d'un état parallèle.

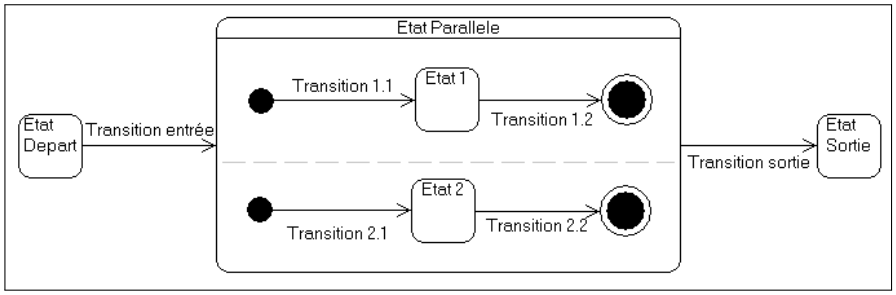


FIGURE 3.15 – Représentation graphique d'un état parallèle SysML4MBT

Dans cet exemple, lorsque la transition `Transition entrée` est exécutée on arrive sur la bordure de l'état parallèle donc directement dans les deux états initiaux. Les transitions `1.1` et `2.1` sont alors exécutées automatiquement et en parallèle. A partir de ce moment là, les deux régions évoluent en parallèle. Les transitions `1.2` et `2.2` peuvent donc être activées indépendamment. La transition `Transition sortie` est activable à partir de n'importe quel état de l'état parallèle.

Barre de fraction et barre de jonction

Les barres de fraction et de jonction permettent de mettre en place une parallélisation de transitions au même titre qu'un état parallèle. Le fonctionnement est le suivant : une transition arrive sur une barre de fraction et plusieurs transitions en partent. Les transitions qui partent de cette barre sont des chemins d'exécution qui se réalisent en parallèle. Chacun de ces chemins évolue à sa manière. Afin de fusionner ces chemins (de manière à interrompre le parallélisme), on utilise une barre de jonction. Les chemins d'exécution pointent sur cette barre, chacun à l'aide d'une transition. Une seule transition repart de cette barre. Pour que cette transition puisse s'exécuter, il faut que toutes les transitions qui arrivent sur la barre aient été franchies. Même si un des chemins arrive sur un état final, les autres continuent leur exécution. La figure 3.16 représente un exemple d'utilisation de ces barres. Cet exemple représente les mêmes chemins en parallèle que la figure 3.15. Cependant, sur la figure 3.15, il est possible de traverser la transition `Transition sortie` quelle que soit la situation au sein de l'état parallèle alors que dans le cas des barres de fraction et de jonction, il faut que les transitions `1.2` et `2.2` aient été franchies pour emprunter la transition `Transition sortie`.

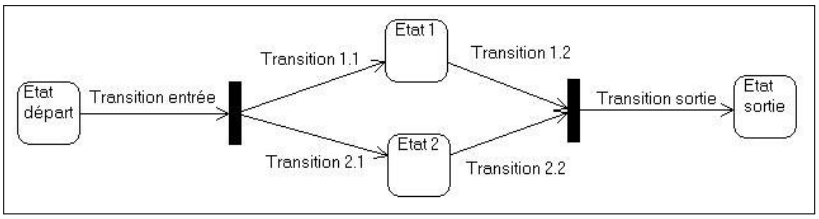


FIGURE 3.16 – Représentation graphique de barres de fraction et de jonction SysML4MBT

Le point de décision (ou point de choix)

La transition qui arrive sur un point de décision est une transition quelconque (automatique ou non-automatique). En revanche, les transitions qui en repartent ne doivent pas porter de déclencheur mais doivent obligatoirement porter une garde. Il est également possible de définir une garde `else` qui sera utilisée si aucune des autres transitions ne peut être activée. Une fois positionné sur le point de décision, plusieurs cas sont possibles :

- Une seule garde est respectée : la transition correspondante est exécutée.
- Plusieurs gardes sont respectées : une des transitions activables est sélectionnée arbitrairement puis exécutée.

- Aucune garde n'est respectée :
  - Une transition avec la garde `else` est présente : cette transition est exécutée.
  - Il n'y a pas de transition avec la garde `else` : le modèle contient une incohérence.

La figure 3.17 représente un exemple d'utilisation du point de choix. Sur cet exemple, à partir de l'état `1`, si l'événement `Evt()` est appelé, la transition aboutissant au point de choix est franchie et il y a alors deux cas :

- Si l'expression `guard` est respectée, l'expression `act` est exécutée et le système passe dans l'état `2`.
- Sinon, l'expression `actSinon` est exécutée et le système passe dans l'état `3`.

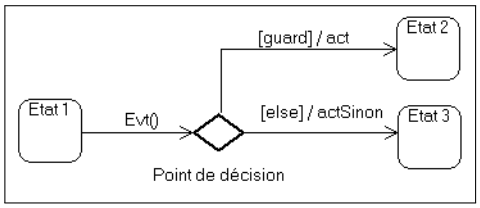


FIGURE 3.17 – Représentation graphique d'un point de décision SysML4MBT

État historique et état historique profond

L'état historique est un pseudo-état qui se place dans un état composite. C'est un état qui mémorise le dernier état actif de l'état composite. Lorsqu'une transition qui pointe vers un état historique est franchie, le système se repositionne directement dans l'état mémorisé.

Par exemple, sur la figure 3.18, que l'on soit dans l'état `State 1` ou `State 2` on peut activer la transition `Tr2`. Lorsque l'on emprunte cette transition, on se repositionne directement dans le dernier état actif de l'état composite.

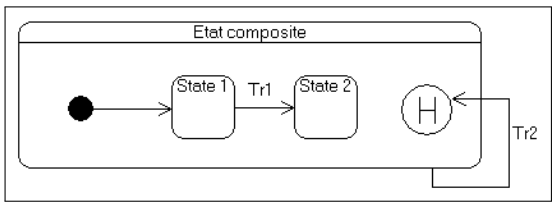


FIGURE 3.18 – Représentation graphique d'un état historique SysML4MBT

L'état historique permet de retourner dans le dernier état visité du niveau courant (niveau d'imbrication auquel est positionné l'état historique). Dans le cas d'états composites imbriqués, seul l'état historique profond permet de retourner dans le dernier état visité tous niveaux confondus.

OCL4MBT

Comme précisé dans la partie 3.2.4, le langage OCL4MBT peut être utilisé à plusieurs endroits au sein d'un modèle SysML4MBT. Une expression OCL qui représente la pré-condition d'une opération doit être vérifiée afin que l'appel de l'opération puisse avoir lieu. Si c'est le cas, la post-condition est exécutée. Dans le cadre des diagrammes d'états-transitions, en ce qui concerne les différentes possibilités d'utilisation, leur fonctionnement est le suivant :

- Les actions onEntry et onExit attachées aux états sont réalisées respectivement à l'entrée et à la sortie d'un état.
- En ce qui concerne la garde d'une transition, il s'agit d'une expression booléenne qui doit être vérifiée pour autoriser le franchissement de la transition.
- Enfin, l'effet d'une transition est exécuté lorsque la transition est franchie.

Ainsi, dans l'ordre, les étapes de franchissement d'une transition non-automatique sont les suivantes : la pré-condition de l'opération et la garde sont vérifiées. Si elles sont vraies, la post-condition de l'opération est exécutée, suivie de l'action onExit de l'état que l'on quitte, de l'effet de la transition et enfin, de l'action onEntry de l'état auquel on accède.

3.3 Comparatif entre SysML4MBT et UML4MBT

Cette partie dresse la liste exhaustive des éléments pris en charge dans SysML4MBT. Chaque tableau est composé de trois colonnes :

1. la colonne *Elément* donne les noms des éléments pris en compte dans SysML4MBT,
2. la colonne *Description* rappelle brièvement à quoi servent ces éléments,
3. la colonne *UML4MBT* indique si les éléments sont présents dans UML4MBT. Dans le cas ou l'élément n'est pas présent dans UML4MBT, il est inscrit *Non couvert*. Dans le cas contraire, il est inscrit *Couvert* et le nom en UML de cet élément est précisé entre parenthèses si ce n'est pas le même qu'en SysML.

3.3.1 Diagramme de bloc

Le tableau 3.1 récapitule les éléments du diagramme de bloc pris en compte dans SysML4MBT.

Elément	Description	UML4MBT
Bloc	Représente une partie du SUT	Couvert (Classe)
Association standard	Relie les blocs entre eux	Couvert
Composition	Relie les blocs entre eux avec une signification de composition	Couvert
Opération	Représente une action qui peut être appelée de l'extérieur au niveau du bloc qui la contient	Couvert
Propriété	Variable qui caractérise l'instance du bloc	Couvert (attribut)
Port de flux	Point de communication	Non couvert
Signal	Stimulus transmis à travers les ports	Non couvert
Spécification de flux	Ensemble de signaux	Non couvert
Booléen	Vrai ou faux	Couvert
Entier	Nombre entier	Couvert
Enumération	Liste de choix	Couvert

TABLE 3.1 – SysML4MBT - Diagramme de bloc

3.3.2 Diagramme interne de bloc

Le tableau 3.2 résume les éléments pris en compte dans SysML4MBT au niveau du diagramme interne de bloc. Aucun de ceux-ci n'est pris en compte dans UML4MBT étant donné qu'il n'y a pas dans UML d'équivalent au diagramme interne de bloc.

Elément	Description	UML4MBT
Propriété (propriété)	Elément qui pointe sur la propriété du bloc représenté	Non couvert
Propriété (bloc)	Elément qui pointe sur un bloc qui compose le bloc représenté	Non couvert
Port standard	Service proposé par un bloc	Non couvert
Port de flux	Point d'envoi/réception de données	Non couvert
Connecteur	Liaison entre les ports	Non couvert

TABLE 3.2 – SysML4MBT - Diagramme interne

3.3.3 Diagramme d'états-transitions

Au niveau de la représentation dynamique, le type de diagramme utilisé est le même en SysML4MBT qu'en UML4MBT : il s'agit du diagramme d'états-transitions. Cependant, il existe des éléments présents dans SysML4MBT qui ne sont pas supportés par UML4MBT (cf. tableau 3.3).

Elément	Description	UML4MBT
Etat initial	Point d'entrée dans le modèle	Couvert
Etat final	Point de sortie du modèle	Couvert
Etat simple	Représente un état du système	Couvert
Etat composite	Etat qui peut contenir d'autres états	Couvert
Etat parallèle	Permet la représentation de plusieurs exécutions en parallèle	Non couvert
Point de décision	Pseudo-état représentant un choix	Couvert
Barre de fraction/jonction	Permet la représentation de plusieurs exécutions en parallèle	Non couvert
Etat historique	Permet de retourner au dernier état actif de l'état composite qui le contient	Non couvert
Etat historique profond	Permet de retourner au dernier état actif (quel que soit son niveau d'imbrication) de l'état composite qui le contient	Non couvert
Transition	Lien entre deux états	Couvert
Déclencheur	Evénement qui rend possible le franchissement de la transition	Couvert
Port événement	Précision du port par lequel arrive le signal quand le déclencheur est une réception de signal	Non couvert
Garde	Expression booléenne qui valide le franchissement de la transition	Couvert
Effet	Action réalisée par la transition	Couvert
Action onEntry	Action s'exécutant à l'entrée d'un état	Couvert
Action onExit	Action s'exécutant à la sortie d'un état	Couvert
Envoi de signal	Action OCL d'envoyer un signal	Non couvert

TABLE 3.3 – SysML4MBT - Diagramme d'états-transitions

3.3.4 Diagramme d'exigences

Le diagramme d'exigences, tout comme le diagramme interne est un diagramme qui n'existe pas en UML. Ainsi tous les éléments du diagramme d'exigences pris en compte en SysML4MBT ne le sont donc pas en UML4MBT excepté la représentation des exigences qui est définie en UML4MBT par l'utilisation d'annotations OCL dédiées (tableau 3.4).

Elément	Description	UML4MBT
Exigence	Exigence du cahier des charges	Couvert (OCL @REQ)
Hiérarchie (DeriveRqt)	Hiérarchie entre les exigences	Non couvert
Elément de modèle	Lien sur un élément de modèle	Non couvert
Satisfy	Lien entre les exigences et les éléments de modèles qui les satisfont	Non couvert

TABLE 3.4 – SysML4MBT - Diagramme d'exigences

3.4 Cadre formel pour SysML4MBT

Cette partie présente une formalisation des modèles SysML4MBT. Cette étape permet de définir formellement les critères de couverture présentés dans la suite de ce mémoire. Cette partie contient une première section qui présente les structures que nous allons utiliser au sein de la formalisation, puis plusieurs sections contenant la formalisation en elle-même, qui est au coeur des contributions de ces travaux de thèse. Seuls les éléments qui sont nécessaires aux définitions exposées plus tard dans ce document (chapitre 4) sont formalisés.

3.4.1 Structures nécessaires

Cette section présente les structures utilisées dans la définition du cadre formel pour SysML4MBT. Différentes structures tels que les n-uplets, les ensembles ou les séquences sont utilisées pour formaliser les différents éléments de modélisation.

Les n-uplets

Le n-uplet est une structure ordonnée généralement de taille fixe, qui peut contenir plusieurs éléments. Chaque élément est accessible à l'aide d'un nom. Les différents éléments ne sont pas forcément de même type. On définit le type à la construction.

Par exemple, un objet `O` représenté sous la forme d'un 2-uplet s'écrira : `O=⟨P1,P2⟩`. Les différents noms des paramètres sont très importants car ils permettent d'y accéder. Par exemple, pour connaître la valeur du deuxième paramètre (`P2`), on utilisera la formule `O.P2`. On précise ensuite la signification et le type de chaque paramètre. Par exemple : `P1` est un entier et `P2` est une chaîne de caractères.

Enfin, on peut instancier chacune de ces structures. On pourra par exemple définir `O1` une instance de `O` telle que : `O1=⟨5,“test”⟩`. On a alors : `O1.P1=5` et `O1.P2=“test”`.

Dans le cas où une des valeurs du n-uplet n'est pas précisée, le caractère “_” est utilisé. Le paramètre représente alors n'importe quelle valeur de son ensemble de définition.

Les ensembles

Un ensemble désigne une collection d'objets de taille variable. Les opérateurs ensemblistes traditionnels tels que l'union ($\cup$), l'intersection ($\cap$), le symbole appartient ($\in$)...peuvent être utilisés. Un ensemble est non ordonné et ne peut pas contenir de doublon.

Pour définir un ensemble, la syntaxe `{a,b,c...}` est utilisée. Par exemple, la notation `{9,5,7}` définit un ensemble de trois entiers. L'ensemble vide se représente par `{}` ou $\emptyset$.

Les séquences

Une séquence désigne une collection d'objets de même type où l'ordre des éléments est pris en compte (à la différence des ensembles).

La représentation se fait à l'aide de la suite des éléments séparés de points virgules, le tout entre crochets. Par exemple, la notation `[5;7;9]` définit une séquence de trois entiers. Une séquence vide se représente : `[]`

On définit les opérateurs booléens $<_{\texttt{seq}}$ et $>_{\texttt{seq}}$. L'opérateur $<_{\texttt{seq}}$ (resp. $>_{\texttt{seq}}$) permet de savoir si un élément précède (resp. succède à) un autre au sein de la séquence `seq`. Exemple : pour `seq=[EltA;EltB;EltC]`,

- `EltA` $<_{\texttt{seq}}$ `EltC` retourne `vrai`.
- `EltB` $>_{\texttt{seq}}$ `EltC` retourne `faux`.

3.4.2 Modèle SysML4MBT

Comme introduit en section 3.2.6, un modèle SysML4MBT contient un diagramme de bloc, éventuellement plusieurs diagrammes internes de bloc, diagrammes d'états-transitions et diagrammes d'exigences. Cependant, les diagrammes d'exigences et les diagrammes internes de bloc ne sont pas nécessaires à la formalisation des processus de génération de tests décrits dans ce document. Ainsi, seuls le diagramme de bloc et les diagrammes d'états-transitions sont formalisés.

Définition 1 (Modèle) *Un modèle SysML4MBT est un 2-uplet* $\langle BDD, SMS \rangle$ *où* BDD *représente le diagramme de bloc et* SMS *l'ensemble des diagrammes d'états-transitions (ensemble de* SM*).*

3.4.3 Diagramme de définition de bloc

Un diagramme de bloc peut contenir des blocs, des associations et des compositions, des signaux, des spécification de flux. Seuls les signaux et les blocs sont formalisés car les autres éléments ne sont pas nécessaires pour la suite de ce document.

Définition 2 (BDD) *Un diagramme de bloc est un 2-uplet* $\langle SIGS, BLOCKS \rangle$ *où* $SIGS$ *est l'ensemble des signaux et* $BLOCKS$ *est l'ensemble des blocs (*$BLOCK$*)*

Un bloc

Un bloc peut contenir des opérations, des propriétés et des ports. Il n'est pas nécessaire de formaliser la structure de ces éléments représentés par des ensembles.

Définition 3 (BLOCK) *Un bloc est représenté par un triplet :*
$BLOCK=\langle OPS, PROPS, PORTS\rangle$ *où* OPS *est l'ensemble des opérations,* $PROPS$ *est l'ensemble des propriétés et* $PORTS$ *est l'ensemble des ports.*

Accesseurs

Pour plus de lisibilité dans la suite de ce mémoire, trois accesseurs sont définis :
- `allOps` représente l'ensemble des opérations d'un modèle. Ainsi, pour un modèle `M` : `M.allOps={op|∃ b.(b ∈ M.BDD.BLOCKS ∧ op ∈ b.OPS)}`
- `allProps` représente l'ensemble des propriétés d'un modèle. Ainsi, pour un modèle `M` : `M.allProps={prop|∃ b.(b ∈ M.BDD.BLOCKS ∧ prop ∈ b.PROPS)}`
- `allPorts` représente l'ensemble des ports d'un modèle. Ainsi, pour un modèle `M` : `M.allPorts={port|∃ b.(b ∈ M.BDD.BLOCKS ∧ port ∈ b.PORTS)}`

3.4.4 Diagramme d'états-transitions

Un diagramme d'états-transitions est composé d'états et de transitions :

Définition 4 (SM) *Un diagramme d'états-transitions est défini par un 2-uplet* $\langle STATES, TRANS\rangle$ *où* $STATES$ *est l'ensemble des états et* $TRANS$ *est l'ensemble des transitions du diagramme.*

Les états (STATES)

Il existe plusieurs types d'états. Tous les états d'une même catégorie sont regroupés au sein du même sous-ensemble.

Définition 5 (STATES) *L'ensemble* $STATES$ *représente l'ensemble des états.*
$STATES$ *se décompose en 9 sous-ensembles :*
- ST_{init} *pour les états initiaux,*
- ST_{final} *pour les états finaux,*
- ST_{stand} *pour les états standards,*
- ST_{comp} *pour les états composites,*
- ST_{par} *pour les états parallèles,*
- ST_{choice} *pour les états de choix,*
- ST_{fork} *pour les barres de fraction,*
- ST_{join} *pour les barres de jonction,*
- ST_{hist} *pour les états historiques.*

Afin de rendre plus claires les démonstrations contenues dans ce document, nous définissons l'ensemble des pseudos-états.

Définition 6 (Pseudos-etats) *Tous les pseudos-etats sont regroupés dans l'ensemble* ST_{pseudo}. *On a ainsi* $ST_{pseudo}=ST_{init} \cup ST_{choice} \cup ST_{fork} \cup ST_{join} \cup ST_{hist}$

Pour la majorité des états, il est suffisant de les définir par leur nom et par l'état qui les contient. C'est le cas pour tous les états sauf l'état historique et l'état parallèle. Dans le cas de l'état parallèle, il est utile de dresser la liste des régions qu'il contient.

Définition 7 (ST_{par}) *Un état parallèle est défini par un 3-uplet* $\langle STname, STmaster, STregions \rangle$ *où :*

- $STname$ *est le nom de l'état.*
- $STmaster \in (ST_{comp} \cup \{reg / \exists\ par.(par \in ST_{par} \wedge reg \in par.STregions)\})$. *Il représente l'état composite ou la région d'un état parallèle qui le contient.* $STmaster$ *est facultatif. Dans le cas où on ne l'utilise pas, on lui attribue la valeur* $NULL$.
- $STregions$ *est l'ensemble des régions de l'état parallèle.*

Dans le cas d'un état historique, il est nécessaire de connaître son type (à savoir s'il est profond ou non) ainsi que le dernier état actif de l'état composite qui le contient.

Définition 8 (ST_{hist}) *Soit* ST_{hist} *l'ensemble des états historiques du système. Un état historique* $h \in ST_{hist}$ *a pour structure un 4-uplet* $\langle STname, STmaster, STdeep, STtarget \rangle$ *où :*

- $STname$ *est le nom de l'état.*
- $STmaster \in ST_{comp}$. *Il représente l'état composite qui le contient.*
- $STdeep$ *est une valeur booléenne permettant de savoir s'il s'agit d'un état historique profond (*$STdeep=true$*) ou non (*$STdeep=false$*).*
- $STtarget \in (STATES\text{-}STATES_{pseudo})$ *permet de stocker le dernier état visité de l'état composite.*

Dans tous les autres cas, un état est formalisé de la manière suivante :

Définition 9 (STATES - ($ST_{par} \cup ST_{hist}$)) *Chaque élément de* $STATES$ - $(ST_{par} \cup ST_{hist})$ *est un 2-uplet* $\langle STname, STmaster \rangle$ *où :*

- $STname$ *est le nom de l'état.*
- $STmaster \in (ST_{comp} \cup \{reg / \exists\ par.(par \in ST_{par} \wedge reg \in par.STregions)\})$. *Il représente l'état composite ou la région d'un état parallèle qui le contient.* $STmaster$ *est facultatif. Dans le cas où on ne l'utilise pas, on lui attribue la valeur* $NULL$.

Les transitions (TRANS)

Les états sont reliés entre eux par des transitions. Une transition est donc définie par un état de départ et un état d'arrivée. Une transition est déclenchée par un événement (trigger) et est protégée par une garde (expression booléenne). Si cet événement se produit et que la garde est vraie, alors un des comportements associés à la transition se réalise. Ainsi, une transition est définie par un état de départ, un état d'arrivée, un déclencheur, une garde et un ensemble de comportements.

Définition 10 (Les transitions (TRANS)) *TRANS est l'ensemble des transitions du diagramme d'états-transitions. Pour un modèle M, si* $t \in TRANS$ *alors t est un 5-uplet* $\langle TRstart, TRend, TRtrig, TRguard, TRbhvs \rangle$ *où :*

- $TRstart$ *est l'état de départ de la transition.*
  $TRstart \in (STATES - (ST_{final} \cup ST_{hist}))$
- $TRend$ *est l'état d'arrivée de la transition.* $TRend \in (STATES - ST_{init})$.
- $TRtrig$ *correspond au déclencheur de la transition.*
  $TRtrig \in ((M.BDD.SIGS \times M.allPorts) \cup M.allOps)$
- $TRguard$ *est la garde. Il s'agit d'une expression booléenne.*
- $TRbhvs$ *est l'ensemble des comportements de la transition.*

Les comportements (BHVS)

Chaque comportement correspond a un effet de transition. Il se réalise au franchissement de la dite transition si certaines conditions sont réunies. En effet, chaque comportement est associé à une condition (expression booléenne) qui doit être vraie pour que le comportement soit exercé. Théoriquement, un comportement peut être associé à plusieurs transitions. Dans le cadre de cette formalisation, pour simplifier les manipulations futures, un comportement est associé à une seule transition. Ainsi, pour représenter un comportement associé à plusieurs transitions, le comportement en question est dupliqué.

Définition 11 (Comportements) $TRbhvs$ *est l'ensemble des comportements d'une transition. Pour un modèle M, pour une transition* t *de ce modèle, si* $b \in t.TRbhvs$ *alors* b *est un 2-uplet* $\langle BHVdec, BHVaction \rangle$ *où :*

- $BHVdec$ *est la décision. Il s'agit d'une expression booléenne définissant la condition qui doit être respectée pour que le comportement soit exercé.*
- $BHVaction$ *est l'ensemble des actions du comportement. Une action est de la forme :*
  $$(M.BDD.SIGS \times M.allPorts) \cup (M.allProps \times newValues)$$
  ($newValues$ *représentant une nouvelle valeur à appliquer à la propriété).*

On définit également un accesseur permettant, à partir d'un comportement, de connaître la transition qui le contient.

Définition 12 (`tr`) *Pour un comportement* `b`*,* `b.tr` *représente la transition auquel le comportement appartient. Ainsi :*

$$\forall\ bhv.(\exists\ t.(t \in \{trans \mid \exists\ sm.(sm \in M.SMS \wedge trans \in sm.TRANS)\} \wedge bhv \in t.TRbhvs \Rightarrow bhv.tr{=}t))$$

3.5 Exemple fil rouge

SysML4MBT a été défini afin de permettre la modélisation de systèmes embarqués. Il est ainsi possible de modéliser l'exemple fil rouge.

3.5.1 Représentation des exigences

La première étape consiste généralement à représenter le diagramme d'exigences. Il permet de se rendre compte de l'intégralité des fonctionnalités à représenter. Dans notre cas, l'exigence globale est d'avoir un système de gestion de feux/phares de véhicule. Cette exigence peut être découpée en plusieurs sous-exigences telles que :

- Le panneau de contrôle doit être inactif lorsque le véhicule est éteint.
- Il doit être possible d'allumer les feux de croisement.
- Il doit être possible d'allumer les phares.
- Il doit être possible de faire des appels de phares.

Dans le cadre de cet exemple, il est possible de définir encore d'autres exigences et de re-découper celles-ci. Cependant ces quatre exigences suffisent à donner un aperçu de ce type de diagramme. La représentation graphique de ces exigences est donnée par la figure 3.19.

3.5.2 Représentation de la structure

En premier lieu, on dresse la liste des composants qui impactent le fonctionnement du système afin de pouvoir réaliser le diagramme de bloc. Le système est composé de l'écran tactile qui communique avec les feux de croisement (*dipped headlights*) et les phares (*full headlights*). Les feux et les phares sont considérés comme un unique composant. Le démarreur du véhicule est également pris en compte car il permet l'activation et l'extinction

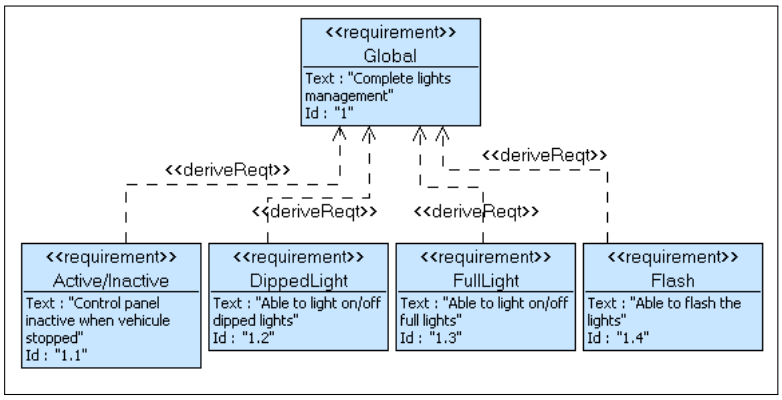


FIGURE 3.19 – Diagramme d'exigences SysML4MBT de l'exemple fil rouge

du panneau. De ce fait, le système (`System`) est composé de trois éléments : le panneau de contrôle (`ControlPanel`), les feux et phares, (`Lights`) et le démarreur (`Ignition`).

Ensuite, on définit l'ensemble des opérations, c'est-à-dire, l'ensemble des actions qui peuvent être appelées depuis l'environnement afin d'actionner le système. Au niveau du démarreur, l'utilisateur peut l'actionner (allumage du véhicule) et le couper (extinction du véhicule). Il y a donc deux opérations à ajouter au niveau du bloc `Ignition` :

- `start` qui représente l'allumage du véhicule,
- `stop` qui modélise l'extinction du véhicule.

Ces opérations ne nécessitent pas de paramètre. Au niveau du panneau de contrôle, en totalisant toutes les situations accessibles, six actions sont possibles :

- Allumer les feux de croisement (`swOnDippedLights` pour *switch on dipped headlights*).
- Eteindre les feux de croisement (`swOffDippedLights` pour *switch off dipped headlights*).
- Allumer les phares (`swOnFullLights` pour *switch on full headlights*).
- Eteindre les phares (`swOffFullLights` pour *switch off full headlights*).
- Enclencher le bouton d'appel de phares (`flashLights`).
- Relâcher le bouton d'appel de phares (`stopFlashLights`).

Ce sont les seules actions permettant d'utiliser directement le panneau tactile. Ces opérations n'ont pas besoin de paramètre. Elles sont ajoutées au bloc `ControlPanel`. On obtient donc le diagramme représenté sur la figure 3.20.

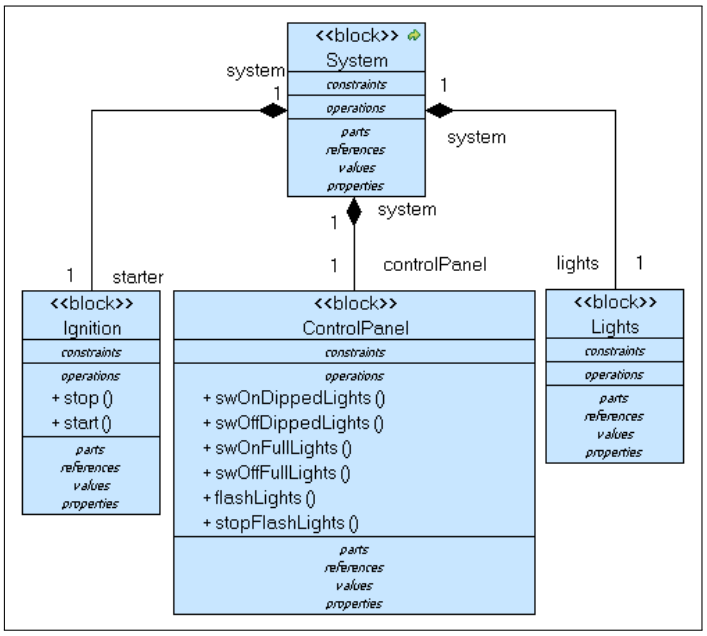


FIGURE 3.20 – Blocs et opérations du modèle SysML4MBT de l'exemple fil rouge

3.5.3 Représentation des communications

Une fois la structure mise en place, il est possible de représenter les communications entre les blocs en représentant les ports (définis mais non représentés dans le diagramme de bloc), les connecteurs qui relient les ports (diagramme interne de bloc) et les signaux qui vont y transiter (définis dans le diagramme de bloc).

Lorsque l'on actionne le démarreur (appel de l'opération `start`), l'écran tactile s'allume et il est alors possible de l'utiliser. A l'inverse, lorsque l'on coupe le moteur (appel de l'opération `stop`), le panneau tactile se désactive. Il en est de même avec les feux/phares. Ainsi, il est nécessaire de représenter un connecteur entre le bloc `Ignition` et le bloc `ControlPanel` qui permet l'allumage/l'extinction du panneau et un connecteur entre le bloc `Ignition` et le bloc `Lights` qui permet l'activation/désactivation des feux/phares. Ces connecteurs sont unidirectionnels (de `Ignition` vers les autres blocs). Pour ce faire, un port *out* est mis en place au niveau du bloc `Ignition`. Il servira à la communication avec le panneau et les feux/phares. Ensuite, un port *in* au niveau du bloc `ControlPanel` et un port *in* au niveau du bloc `Lights` sont mis en place. Deux signaux peuvent transiter sur ces ports : le signal envoyé à l'allumage (`startSignal`) du véhicule et celui envoyé à l'extinction (`stopSignal`). Ces signaux sont regroupés au sein d'une spécification de flux afin de pouvoir typer les ports.

Les autres chemins de communication se trouvent entre le panneau de contrôle et les lumières. En effet, lorsque l'utilisateur actionne l'allumage des feux de croisement par exemple, un signal est envoyé au composant des feux/phares afin de réaliser l'allumage des lumières concernées. Afin de représenter ces communications, on met en place un port *out* au niveau du panneau de contrôle qui permettra d'envoyer l'information d'allumage/d'extinction aux feux/phares. Au niveau du bloc `Lights`, deux ports *in* sont définis : un pour la réception des signaux concernant les feux de croisement, et un pour la réception des phares. Deux signaux suffisent :

- `swOnSignal` pour l'allumage,
- `swOffSignal` pour l'extinction.

Le port sur lequel le signal est transmis définit si ce sont les feux de croisement ou les phares qui sont concernés. Les signaux et les spécifications de flux définies sont représentés sur la figure 3.21.

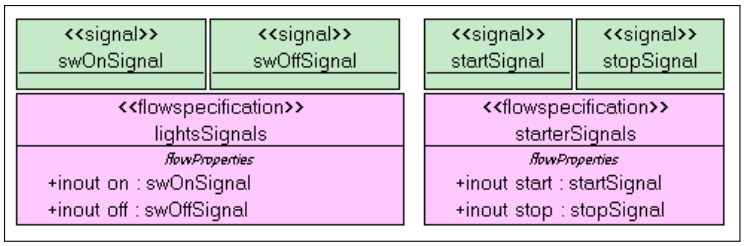


FIGURE 3.21 – Signaux du modèle SysML4MBT de l'exemple fil rouge

On peut ainsi représenter le diagramme interne de bloc de cet exemple. Du fait de la simplicité de cet exemple, une seule vue est nécessaire pour représenter les communications. Le diagramme interne est représenté sur la figure 3.22.

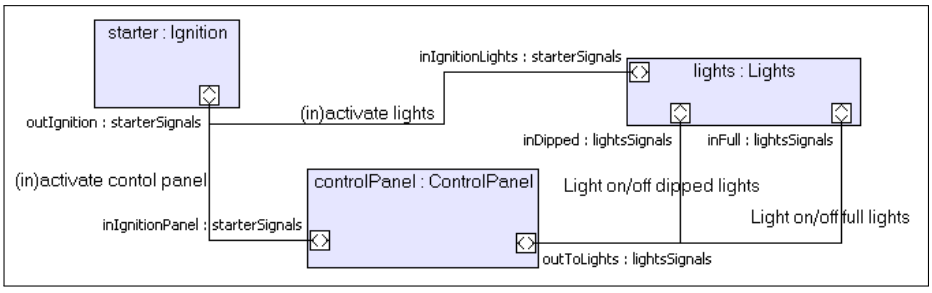


FIGURE 3.22 – Diagramme interne de bloc du modèle SysML4MBT de l'exemple fil rouge

3.5.4 Représentation dynamique

L'étape suivante consiste à la réalisation des diagrammes d'états-transitions de chaque bloc qui compose le système. Il faut donc en réaliser trois. Tout d'abord, le diagramme d'états-transitions du bloc `Ignition`.

Ignition

Le démarreur se définit par seulement deux états : soit allumé, soit éteint. Initialement, le véhicule est éteint (état `ignitionOff`). Lorsque l'opération `start()` est appelée (ce qui correspond à l'action de démarrer le véhicule), un signal est envoyé au panneau de contrôle et au composant feux/phares pour les activer (signal `startSignal`). On arrive alors dans l'état `ignitionOn`. Dans cet état, il est possible de couper le moteur en appelant l'opération `stop()`. Un signal d'extinction `stopSignal` est alors transmis au panneau et aux lumières pour les désactiver. On obtient le diagramme d'états-transitions de la figure 3.23.

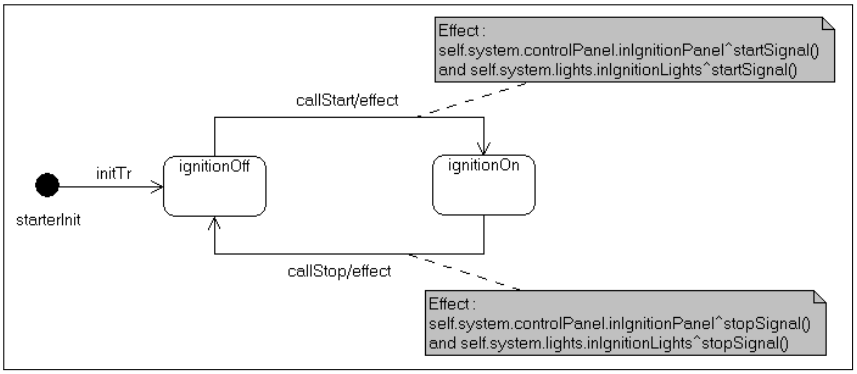


FIGURE 3.23 – Diagramme d'états-transitions du bloc `Ignition` du modèle SysML4MBT de l'exemple fil rouge

ControlPanel

Le panneau de contrôle est initialement éteint (état `panelOff`). Lorsqu'il reçoit le signal `startSignal` envoyé par le composant `Ignition`, le panneau de contrôle démarre. Il est donc dans l'état `panelOn`. A l'inverse, une fois dans cet état, la réception du signal `stopSignal` éteindra le panneau de contrôle (état `panelOff`).

Lorsqu'il est allumé, le panneau peut prendre plusieurs états différents. De ce fait, `panelOn` est un état composite. L'action d'appel de phares étant activable depuis plusieurs

états, nous laissons dans un premier temps cette fonctionnalité en attente. Lorsque le panneau est allumé, il est initialement dans l'état où toutes les lumières (feux et phares) sont éteintes (état `allOff`). Il est possible d'allumer les feux de croisement par l'appel de l'opération `swOnDippedLights`. Cette action aura pour effet d'envoyer le signal `swOnSignal` sur le port `inDipped` du bloc `Lights`. Le panneau changera d'état pour arriver dans l'état `onlyDipped`.

Une fois dans cet état (`onlyDipped`), il est possible d'éteindre les feux de croisement ou d'allumer les phares. La première possibilité se traduit par l'appel de l'opération `swOffDipped` qui entraine l'envoi du signal `swOffSignal` sur le port `inDipped` du bloc `Lights`. Cette action remet le panneau de contrôle dans l'état `allOff`.

Depuis l'état `onlyDipped`, il est possible d'allumer les phares par l'appel de l'opération `swOnFull`. Cette action a pour incidence d'éteindre les feux de croisement (`Dipped`). L'appel de cette opération engendre donc l'envoi du signal d'allumage (`swOnSignal`) aux phares (port `inFull`) et l'envoi du signal d'extinction (`swOffSignal`) aux feux de croisement (port `inDipped`). Le système se trouve alors dans l'état `onlyFull`. La transition inverse (celle qui va de l'état `onlyFull` à l'état `onlyDipped`) est déclenchée par l'appel de l'opération contraire (c'est-à-dire `swOffFull`) et engendre l'envoi des signaux opposés.

On obtient ainsi le diagramme d'états-transitions représenté sur la figure 3.24.

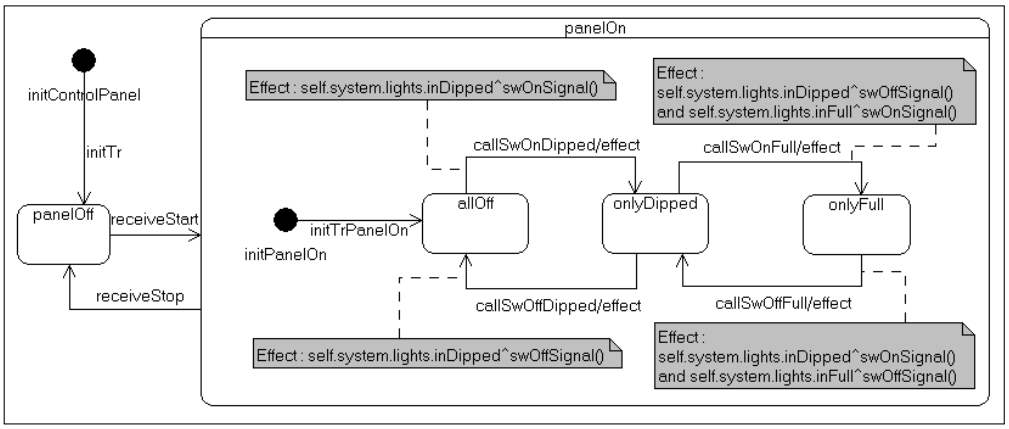


FIGURE 3.24 – Diagramme d'états-transitions du bloc `ControlPanel` du modèle SysML4MBT de l'exemple fil rouge

En ce qui concerne la fonction appel de phares, elle est théoriquement activable à partir de n'importe quel état contenu dans l'état composite `PanelOn`. Pour représenter cette fonction, une transition qui part de la bordure de l'état composite et qui arrive dans un nouvel état (`flash`) est modélisée. Cette transition est déclenchée par l'appel de l'opération `flashLights` et a pour effet d'envoyer un signal d'allumage aux feux de

croisement et aux phares. Afin de quitter le mode appel de phares, l'utilisateur doit relâcher le bouton. Cette action se traduit par la réception de l'appel de l'opération `stopFlashLights`. A la suite de l'appel de phares, le système doit retourner dans le dernier état de l'état composite `panelOn` parcouru. Ainsi, cette transition pointe vers un état historique contenu dans l'état composite. Cette transition permettra donc au système de se repositionner dans le dernier état actif du panneau de contrôle. Cependant, il faut également rétablir la situation des feux de croisement et des phares. Il est donc nécessaire d'ajouter deux propriétés au bloc `ControlPanel` qui permettront de se souvenir de l'état des feux de croisement et des phares avant l'appel de phares afin de pouvoir rétablir la situation en transmettant les bons signaux.

On insère donc, dans le diagramme de bloc, au sein du bloc `ControlPanel`, deux propriétés de type booléen :

- `dippedOn` représentant l'état des feux de croisement (vrai pour allumés et faux pour éteints),
- `fullOn` représentant l'état des phares (vrai pour allumés et faux pour éteints).

Ces deux propriétés prennent la valeur `faux` par défaut. La nouvelle version du bloc `ControlPanel` au niveau du diagramme de bloc est représentée figure 3.25).

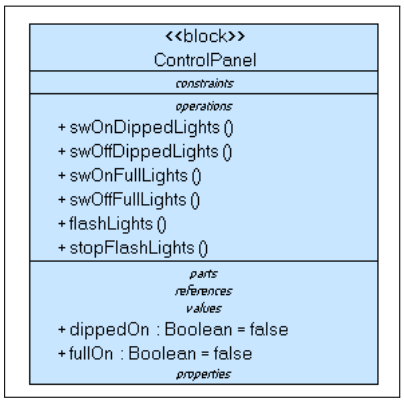


FIGURE 3.25 – Bloc `ControlPanel` du modèle SysML4MBT de l'exemple fil rouge

Ces propriétés sont mises à jour en utilisant les actions onEntry de certains états :

- A l'entrée dans l'état `allOff`, elle sont toutes deux mises à `faux`.
- A l'entrée dans l'état `onlyDipped`, `dippedOn` est mise à `vrai` et `fullOn` est mise à faux.
- A l'entrée dans l'état `onlyFull`, `dippedOn` est mise à `faux` et `fullOn` est mise à vrai.

De cette manière, il est possible d'envoyer les signaux permettant de rétablir la situation à la fin d'un appel de phares.

L'effet de la transition qui quitte l'état `flash` contient deux `if...then...else` signifiant :

- Si `dippedOn` est vrai, alors on ne fait rien, sinon le signal d'extinction des feux de croisement est envoyé.
- Si `fullOn` est vrai, alors on ne fait rien, sinon le signal d'extinction des phares est envoyé.

On obtient finalement le diagramme d'états-transitions représenté sur la figure 3.26.

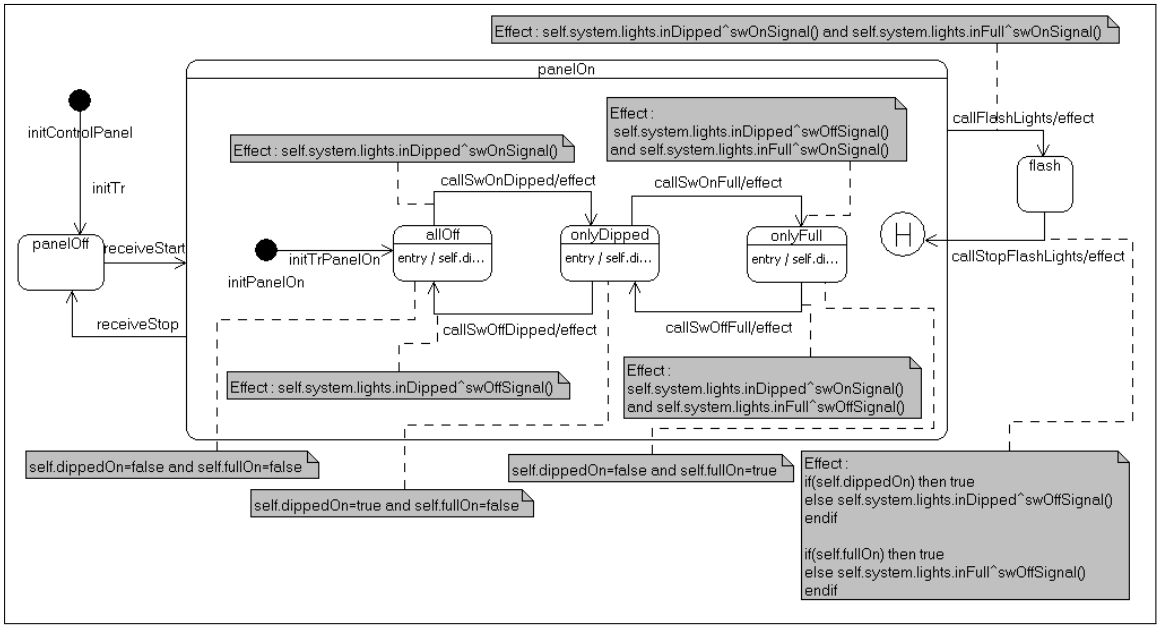


FIGURE 3.26 – Diagramme d'états-transitions du bloc `ControlPanel` du modèle SysML4MBT de l'exemple fil rouge

Lights

Comme pour le panneau de contrôle, le premier état des lumières (appelé `inactive`) est en attente du signal de démarrage. Lorsque ce signal est reçu, la composante passe dans l'état actif dans lequel plusieurs situations vont pouvoir se présenter : cet état est donc un état composite (`active`). Depuis cet état, il est alors possible de recevoir le signal `stop` indiquant que le moteur a été coupé. Dans ce cas, cette composante retourne dans l'état inactive.

Au sein de l'état `active`, la composante est initialement dans l'état où toutes les lumières sont éteintes (`allOff`). Depuis cet état, il est possible de recevoir un signal d'allumage des feux de croisement (signal `swOnSignal` sur le port `inDipped`) qui effectue un changement d'état vers l'état `onlyDipped` (seuls les feux de croisement sont allumés).

De même, la réception du signal `swOffSignal` sur le port `inDipped` depuis ce nouvel état ramène à l'état précédent. Depuis l'état `onlyDipped`, il est également possible de recevoir le signal `swOnSignal` sur le port `inFull`. On arrive alors dans l'état `both` qui désigne le fait que toutes les lumières sont allumées. Depuis cet état, deux possibilités : soit un signal d'extinction des phares est reçu, dans ce cas le système se retrouve dans l'état `onlyDipped`, soit un signal d'extinction des feux de croisement est reçu et dans ce cas, le système se retrouve dans l'état `onlyFull`. Depuis l'état `onlyFull`, la seule réception définie est celle du signal `swOnSignal` sur le port `inDipped`.

On obtient le diagramme d'états-transitions représenté sur la figure 3.27.

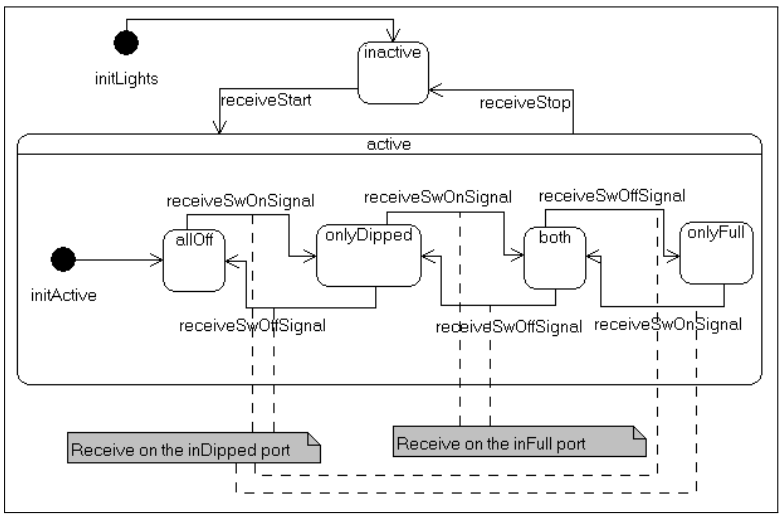


FIGURE 3.27 – Diagramme d'états-transitions du bloc `Lights` du modèle SysML4MBT de l'exemple fil rouge

3.5.5 Complément au diagramme d'exigences

Maintenant que tous les diagrammes sont réalisés, il est possible de relier les exigences aux éléments de modèle qui permettent de les satisfaire. Les exigences sont satisfaites de cette manière :

- L'exigence selon laquelle le panneau de contrôle doit être inactif lorsque le véhicule est éteint est satisfaite par l'état panelOff du diagramme d'états-transitions du bloc `ControlPanel`.
- En ce qui concerne l'exigence qui indique qu'il est possible d'allumer les feux de croisement, elle est satisfaite par le couple de la transition qui envoie le signal d'allumage des feux et de la transition de réception de ce signal.
- Il en est de même pour le cas des phares.

– Et enfin, la transition `flashLights` du diagramme d'états-transitions du panneau de contrôle satisfait l'exigence qui concerne les appels de phares.

On obtient le diagramme représenté sur la figure 3.28.

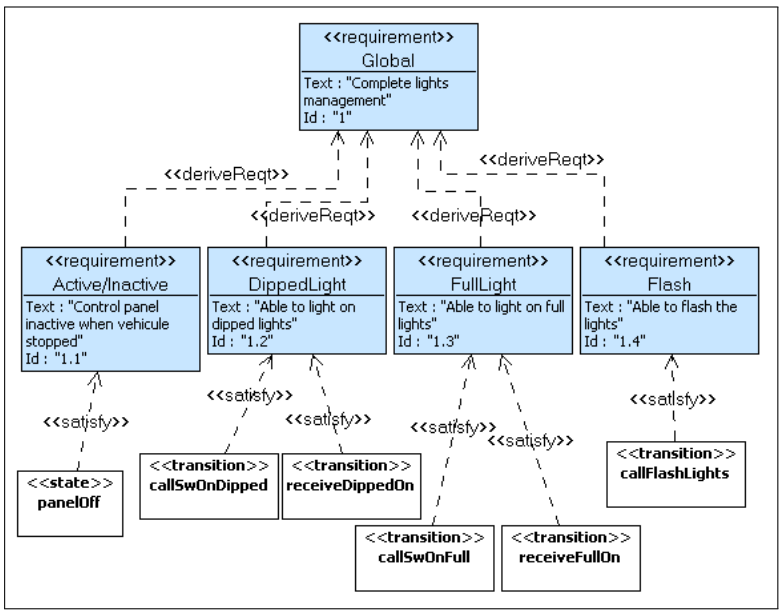


FIGURE 3.28 – Diagramme d'exigences du modèle SysML4MBT de l'exemple fil rouge

3.6 Synthèse

Cette partie a détaillé l'ensemble des éléments de modélisation SysML qui sont pris en compte dans le cadre de nos travaux. Chaque élément de cet ensemble baptisé SysML4MBT a été décrit et sa sémantique a été donnée. La partie 3.3 a permis de visualiser tous les éléments pris en compte dans SysML4MBT ainsi que les différences avec UML4MBT sur lequel nous nous sommes basés. Cette formalisation permet de décrire formellement les stratégies de génération de tests expliquées dans la suite de ce mémoire. La modélisation de l'exemple fil rouge permet d'illustrer le fait que SysML4MBT convient à la représentation de systèmes hybrides (combinants logiciel et matériel) et permet la capture de tous les comportements de chacun des composants. La modélisation des communications se fait de manière précise et intuitive. Enfin, le diagramme d'exigences permet une représentation claire des exigences du système et de leur couverture par les éléments de modèle.

Chapitre 4

Stratégies de couverture des modèles SysML

Sommaire

Comme expliqué au début de ce mémoire, nous proposons d'effectuer de la génération de tests à partir de modèles en utilisant des critères de couverture pour sélectionner les tests. Dans la partie précédente sont présentés les éléments de modélisation SysML pris en compte et la façon dont ils sont utilisés. Dans cette partie, nous introduisons les critères de couverture dédiés aux modèles SysML4MBT. Les définitions des critères sont exprimées sur la base de la formalisation présentée en section 3.4.

Comme vu dans l'état de l'art, il existe plusieurs familles de critères de couverture. La famille de critères à laquelle nous nous intéressons plus particulièrement concerne les critères de couverture orientés flot de données. Toutefois, nous positionnons également nos travaux vis-à-vis des critères basés sur les transitions. Nous utilisons, comme base de travail, la hiérarchie des critères de couverture définie dans [FW88, RW85] et illustrée par la figure 4.1, dans laquelle les critères encadrés représentent les critères basés sur les transitions tandis que les autres sont des critères de type flot de données. La flèche indique que la satisfaction du critère de départ implique la satisfaction du critère de couverture d'arrivée.

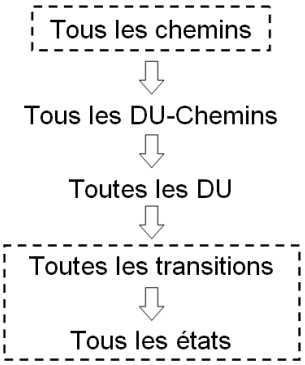


FIGURE 4.1 – Hiérarchie des critères de couverture

La première section définit, en accord avec la formalisation présentée précédemment, la formalisation des tests puis nous présentons en détail en section 4.3 les critères de référence présentés sur la figure 4.1. La section 4.4 détaille l'utilisation qui est faite de ces critères dans le cadre de la génération de tests à partir de modèles UML4MBT puis analyse la pertinence de cette stratégie vis-à-vis des modèles SysML4MBT. Ensuite, les nouveaux critères dédiés aux modèles SysML4MBT sont présentés en section 4.5. Afin d'illustrer ces différents critères, un exemple jouet (présenté en section 4.2) est utilisé. Pour terminer ce chapitre, les différentes stratégies sont appliquées sur l'exemple fil rouge.

4.1 Formalisation des tests

Cette partie définit la formalisation d'une séquence d'animation (également appelée trace) puis d'un test.

Une séquence d'animation est une exécution particulière du système. Il s'agit d'une succession d'appels d'opérations faisant réagir le système afin de le mettre dans un état particulier. Au niveau du modèle, il s'agit d'un chemin au sein des diagrammes d'états-transitions, c'est-à-dire, une succession de franchissements de transitions. Chaque franchissement de transition correspond à l'activation d'un comportement (un parmi ceux associés à la transition). Le franchissement de cette transition peut éventuellement permettre de déclencher également plusieurs autres transitions automatiquement. Il faut donc formaliser la liste des opérations appelées ainsi que les transitions franchies automatiquement. Il est nécessaire de préciser le comportement qui est franchi au niveau de chacune des transitions. On dit qu'une trace *couvre* un comportement/une transition si la trace franchit le comportement/la transition lors de son exécution.

Définition 13 (Traces) *L'ensemble* $TRACES$ *représente toutes les traces possibles du système. Une trace* $tr \in TRACES$ *est une séquence de pas.*

Définition 14 (Pas) *Un pas est un triplet* $\langle StepOP, StepBhv, AllBhvs \rangle$ *où :*

- $StepOP$ *est l'opération responsable du déclenchement du pas.*
  $StepOP \in allOps$
- $StepBhv$ *est le comportement qui est franchi par la réception de l'appel d'opération débutant le pas.*
  $StepBhv \in \{bhv | \exists\ (sm,t).(sm \in SMS \land t \in sm.TRANS \land bhv \in t.TRbhvs)\}$
- $AllBhvs$ *est l'ensemble des comportements qui seront franchis par ce pas (y compris* $StepBhv$*).*
  $AllBhvs \subseteq \{bhv | \exists\ (sm,t).(sm \in SMS \land t \in sm.TRANS \land bhv \in t.TRbhvs)\}$
  Soit bhv *un comportement,* $AllBhvs$ *doit respecter la grammaire suivante :*
  - $AllBhvs = bhv \mid seq \mid par$
  - $seq = [\ (bhv \mid par)\ (;\ (bhv \mid par))^+]$
  - $par = \{\ (bhv \mid seq)\ (,\ (bhv \mid seq))^+\}$

  Une séquence de type seq *représente une succession ordonnée de comportements alors qu'une séquence de type* par *représente des exécutions de comportements en parallèle.*

Définition 15 (Tests) *L'ensemble* $TESTS$ *représente l'ensemble des tests générés lors d'une campagne de test. Il s'agit d'un sous-ensemble de* $TRACES$, $TESTS \subseteq TRACES$. *Un test est également appelé cas de test.*

4.2 Exemple d'illustration des critères de couverture

Afin d'illustrer les critères de couverture et les stratégies mis en place dans ce mémoire, nous définissons un exemple pédagogique. Il s'agit de la modélisation du système d'arrêt d'urgence d'un train. Le système est défini par les règles et les fonctionnalités suivantes :

- Le train peut être soit à l'arrêt, soit en mouvement.
- Il est possible de déclencher l'arrêt d'urgence soit en actionnant un bouton au niveau du conducteur, soit en actionnant un bouton au niveau des passagers.
- Quand un de ces deux boutons est actionné, un signal est envoyé au composant de gestion de l'arrêt d'urgence qui automatiquement :
  - arrête le train et déclenche l'alarme si le train est en mouvement,
  - prévient uniquement la régulation si le train est à l'arrêt.

Afin de simplifier la manipulation de cet exemple, nous décidons d'abstraire les actions du gestionnaire d'arrêt d'urgence. Dans le même ordre d'idée, nous ne représentons pas les diagrammes de bloc et interne de bloc qui ne sont pas nécessaires à l'illustration de cette section. Nous représentons donc simplement les trois diagrammes d'états-transitions, celui de l'état général du train (en mouvement ou à l'arrêt), celui des actionneurs (boutons conducteur et passagers) et celui du gestionnaire d'arrêt d'urgence. On numérote toutes les transitions pour faciliter la lisibilité.

Tout d'abord, le diagramme d'états-transitions représentant l'état général du train contient deux états : l'état `STOPPED` qui représente le train à l'arrêt et l'état `MOVING` qui représente le train en déplacement. L'expression `start()` (resp. `stop()`) représente l'appel de l'opération qui définit la mise en mouvement (resp. l'arrêt) du train. La figure 4.2 représente le diagramme d'états-transitions correspondant.

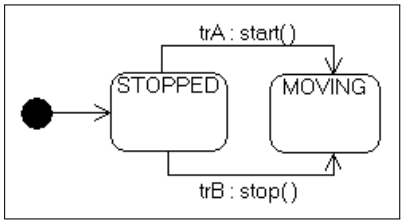


FIGURE 4.2 – Diagramme d'états-transitions de l'état général du train

Au niveau des boutons permettant d'activer l'arrêt d'urgence, l'état initial est l'attente de l'activation d'un de ces boutons (état `READY`). Quand l'un des deux est actionné (réception de l'appel de l'opération `driver()` quand le bouton au niveau du conducteur est actionné ou de l'opération `passengers()` quand le bouton au niveau des passagers est actionné), un signal (`stopSignal`) est envoyé au gestionnaire d'arrêt d'urgence. Cet

envoi est représenté graphiquement par l'expression `SendStop`. Ce diagramme d'états-transitions est représenté sur la figure 4.3

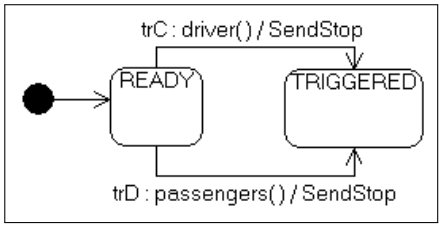


FIGURE 4.3 – Diagramme d'états-transitions des boutons d'arrêt d'urgence du train

Enfin, le gestionnaire d'arrêt d'urgence est en attente (état `WAIT`) de la réception d'un signal d'arrêt d'urgence (réception représentée par le déclencheur `ReceiveStop`). A la réception du signal, deux possibilités :

- Si le train est à l'arrêt (représenté par la garde `TRAIN IS STOPPED`), la régulation est prévenue (effet `Prevent the control center`).
- Si le train est en déplacement (représenté par la garde `TRAIN IS MOVING`), le train est arrêté et l'alarme est déclenchée (effet `Stop the train and set off the alarm`).

Ces comportements sont modélisés sur la figure 4.4.

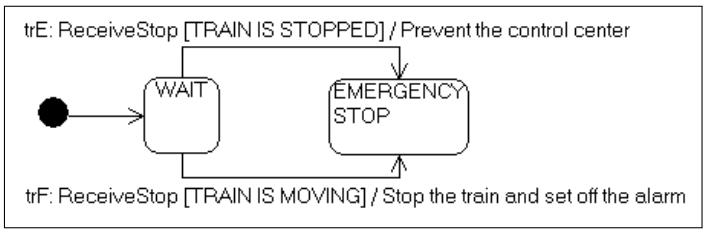


FIGURE 4.4 – Diagramme d'états-transitions du gestionnaire d'arrêt d'urgence du train

4.3 Critères classiques de sélection de tests

Dans cette section, les critères de la littérature de la figure 4.1 utilisés comme référence pour positionner et définir la pertinence de nos propres travaux sont définis. Cette présentation débute par le critère de couverture le plus faible (critère *Tous les états*) pour terminer par le plus fort (critère *Tous les chemins*).

Lors de la présentation des critères *Toutes les DU* et *Tous les DU-Chemins*, les définitions suivantes sont utilisées :

Définition 16 (Concept de définition de variable) *On appelle définition de variable, l'action d'affecter une valeur à une variable du système, c'est-à-dire d'accéder à cette variable en écriture. Ainsi, un comportement de définition de la variable VAR est un comportement qui affecte une valeur à la variable VAR.*

Définition 17 (Concept d'utilisation de variable) *On appelle utilisation de variable, l'action de lire la valeur de cette variable. Ainsi, un comportement d'utilisation de la variable VAR est un comportement qui contient un traitement qui lit la valeur de la variable VAR.*

4.3.1 Critère *Tous les états*

Le critère *Tous les états* assure la couverture de tous les états des diagrammes d'états-transitions, c'est-à-dire que pour chaque état du modèle, il existe au moins un cas de test qui couvre une des transitions qui atteint cet état. La figure 4.5 représente la formalisation de la couverture du critère *Tous les états* par l'ensemble de tests `TESTS` pour le modèle `M`.

```
∀ st.(st ∈ {state | ∃ sm.(sm ∈ M.SMS ∧ state ∈ sm.STATES)}
  ⇒ ∃ bhvTest.
        (bhvTest ∈ {b|∃ (t,pas).(t ∈ TESTS ∧ pas ∈ t ∧ b ∈ pas.AllBhvs)}
        ∧ bhvTest.tr.TRend = st))
```

FIGURE 4.5 – Formalisation du critère *Tous les états*

4.3.2 Critère *Toutes les transitions*

Le critère *Toutes les transitions* assure la couverture de toutes les transitions des diagrammes d'états-transitions, c'est-à-dire que pour chaque transition du modèle, il existe un cas de test qui couvre cette transition. La figure 4.6 représente la formalisation de la couverture du critère *Toutes les transitions* par l'ensemble de tests `TESTS` pour le modèle `M`.

```
∀ trans.(trans ∈ {t|∃ sm.(sm ∈ M.SMS ∧ t ∈ sm.TRANS)}
  ⇒ ∃ bhvTest.
        (bhvTest ∈ {b|∃ (t,pas).(t ∈ TESTS ∧ pas ∈ t ∧ b ∈ pas.AllBhvs)}
        ∧ bhvTest ∈ trans.TRbhvs))
```

FIGURE 4.6 – Formalisation du critère *Toutes les transitions*

4.3.3 Critère *Toutes les DU*

Le critère *Toutes les DU* (*Toutes les Définitions/Utilisations*), parfois appelé *Toutes les utilisations*, assure la couverture de toutes paires définition/utilisation des variables (cf. définitions 16 et 17), c'est-à-dire que pour tout comportement qui contient une définition de variable, pour chaque comportement qui contient une utilisation de cette même variable, il existe un cas de test qui couvre le comportement de définition puis le comportement d'utilisation sans redéfinir la variable entre temps. Ce critère est défini comme une surcharge du critère *Toutes les transitions*. De ce fait, l'application du critère *Toutes les DU* assure la couverture de toutes les transitions en plus de la couverture de toutes les paires définition/utilisation. L'ensemble de tests `TESTS` satisfait le critère *Toutes les DU* pour le modèle `M` si il satisfait les expressions exprimées sur les figures 4.6 et 4.7.

```
∀ (prop,bhvDef,bhvUse).
  ((prop ∈ M.allProps()
   ∧ bhvDef ∈ {b|∃ (sm,trans).
      (sm ∈ M.SMS ∧ trans ∈ sm.TRANS ∧ b ∈ trans.TRbhvs)}
   ∧ bhvUse ∈ {b|∃ (sm,trans).
      (sm ∈ M.SMS ∧ trans ∈ sm.TRANS ∧ b ∈ trans.TRbhvs)}
   ∧ ⟨prop,_⟩ ∈ bhvDef.BHVaction
   ∧ (⟨_,prop⟩ ∈ bhvUse.BHVaction ∨ prop ∈ bhvUse.tr.TRguard))
  ⇒
   (∃ pas.
      (pas ∈ {p|∃ t.(t ∈ TESTS ∧ p ∈ t)}
      ∧ bhvDef ∈ pas.AllBhvs ∧ bhvUse ∈ pas.AllBhvs
      ∧ bhvDef <pas.AllBhvs bhvUse
      ∧ ∄ redef.(redef ∈ pas.AllBhvs
         ∧ ⟨prop,_⟩ ∈ redef.BHVaction
         ∧ bhvDef <pas.AllBhvs redef <pas.AllBhvs bhvUse))
  ∨
    ∃ (test,pasDef,pasUse).
      (test ∈ TESTS ∧ pasDef ∈ test ∧ pasUse ∈ test
      ∧ bhvDef ∈ pasDef.AllBhvs ∧ bhvUse ∈ pasUse.AllBhvs
      ∧ pasDef <test pasUse
      ∧ ∄ redef.(redef ∈ pasDef.AllBhvs
         ∧ ⟨prop,_⟩ ∈ redef.BHVaction
         ∧ bhvDef <pasDef.AllBhvs redef)
      ∧ ∄ redef.(redef ∈ pasUse.AllBhvs
         ∧ ⟨prop,_⟩ ∈ redef.BHVaction
         ∧ redef <pasUse.AllBhvs bhvUse)
      ∧ ∄ (pasRedef,redef).(pasRedef ∈ test
         ∧ redef ∈ pasRedef ∧ ⟨prop,_⟩ ∈ redef.BHVaction
         ∧ pasDef <test pasRedef <test pasUse))))
```

Pour toutes les propriétés et les comportements de definition et d'utilisation associés.

Soit il existe un pas de test où la définition précède l'utilisation.

Soit il existe un test où le pas de définition précède le pas d'utilisation.

FIGURE 4.7 – Formalisation du critère *Toutes les DU*

4.3.4 Critère *Tous les DU-Chemins*

Le critère *Tous les DU-Chemins* assure la couverture de tous les chemins possibles pour chaque paire définition/utilisation de variable. L'ensemble de tests `TESTS` satisfait le critère *Tous les DU-Chemins* pour le modèle `M` si l'expression représentée figure 4.8 est respectée.

$$
\begin{array}{ll}
\forall\ (\texttt{prop},\texttt{bhvDef},\texttt{bhvUse},\texttt{trace}). & \\
\quad ((\texttt{prop} \in \texttt{M.allProps}() & \\
\quad\ \wedge\ \texttt{bhvDef} \in \{\texttt{b} \mid \exists\ (\texttt{sm},\texttt{trans}). & \\
\quad\quad (\texttt{sm} \in \texttt{M.SMS} \wedge \texttt{trans} \in \texttt{sm.TRANS} \wedge \texttt{b} \in \texttt{trans.TRbhvs})\} & \\
\quad\ \wedge\ \texttt{bhvUse} \in \{\texttt{b} \mid \exists\ (\texttt{sm},\texttt{trans}). & \\
\quad\quad (\texttt{sm} \in \texttt{M.SMS} \wedge \texttt{trans} \in \texttt{sm.TRANS} \wedge \texttt{b} \in \texttt{trans.TRbhvs})\} & \\
\quad\ \wedge\ \langle \texttt{prop},\_ \rangle \in \texttt{bhvDef.BHVaction} & \\
\quad\ \wedge\ (\langle \_,\texttt{prop} \rangle \in \texttt{bhvUse.BHVaction} \vee \texttt{prop} \in \texttt{bhvUse.tr.TRguard}) & \\
\quad\ \wedge\ \texttt{trace} \in \texttt{TRACES} & \\
\quad\ \wedge\ (\exists\ \texttt{pas}. & \\
\quad\quad\ (\texttt{pas} \in \texttt{trace} \wedge \texttt{bhvDef} \in \texttt{pas.AllBhvs} & \\
\quad\quad\ \wedge\ \texttt{bhvUse} \in \texttt{pas.AllBhvs} \wedge \texttt{bhvDef} <_{\texttt{pas.allBhvs}} \texttt{bhvUse} & \\
\quad\quad\ \wedge\ \nexists\ \texttt{redef}.(\texttt{redef} \in \texttt{pas.AllBhvs} & \\
\quad\quad\quad\ \wedge\ \langle \texttt{prop},\_ \rangle \in \texttt{redef.BHVaction} & \\
\quad\quad\quad\ \wedge\ \texttt{bhvDef} <_{\texttt{pas.AllBhvs}} \texttt{redef} <_{\texttt{pas.AllBhvs}} \texttt{bhvUse})) & \\
\quad\ \vee & \\
\quad\quad\ \exists\ (\texttt{pasDef},\texttt{pasUse}). & \\
\quad\quad\ (\texttt{pasDef} \in \texttt{trace} \wedge \texttt{pasUse} \in \texttt{trace} & \\
\quad\quad\ \wedge\ \texttt{bhvDef} \in \texttt{pasDef.AllBhvs} \wedge \texttt{bhvUse} \in \texttt{pasUse.AllBhvs} & \\
\quad\quad\ \wedge\ \texttt{pasDef} <_{\texttt{trace}} \texttt{pasUse} & \\
\quad\quad\ \wedge\ \nexists\ \texttt{redef}.(\texttt{redef} \in \texttt{pasDef.AllBhvs} & \\
\quad\quad\quad\ \wedge\ \langle \texttt{prop},\_ \rangle \in \texttt{redef.BHVaction} & \\
\quad\quad\quad\ \wedge\ \texttt{bhvDef} <_{\texttt{pasDef.AllBhvs}} \texttt{redef}) & \\
\quad\quad\ \wedge\ \nexists\ \texttt{redef}.(\texttt{redef} \in \texttt{pasUse.AllBhvs} & \\
\quad\quad\quad\ \wedge\ \langle \texttt{prop},\_ \rangle \in \texttt{redef.BHVaction} & \\
\quad\quad\quad\ \wedge\ \texttt{redef} <_{\texttt{pasUse.AllBhvs}} \texttt{bhvUse}) & \\
\quad\quad\ \wedge\ \nexists\ (\texttt{pasRedef},\texttt{redef}).(\texttt{pasRedef} \in \texttt{trace} & \\
\quad\quad\quad\ \wedge\ \texttt{redef} \in \texttt{pasRedef} & \\
\quad\quad\quad\ \wedge\ \langle \texttt{prop},\_ \rangle \in \texttt{redef.BHVaction} & \\
\quad\quad\quad\ \wedge\ \texttt{pasDef} <_{\texttt{trace}} \texttt{pasRedef} <_{\texttt{trace}} \texttt{pasUse})))) & \\
\Rightarrow\ \texttt{trace} \in \texttt{TESTS}) &
\end{array}
$$

Pour toutes les propriétés et les comportements de définition et d'utilisation associés et pour toutes les traces qui :

contiennent un pas incluant la succession d'une définition et d'une utilisation

ou qui contiennent la succession d'un pas de définition et d'un pas d'utilisation

la trace doit être testée.

FIGURE 4.8 – Formalisation du critère *Tous les DU-Chemins*

4.3.5 Critère *Tous les chemins*

Le critère *Tous les chemins* est le critère le plus fort. Il assure la couverture de tous les chemins possibles du modèle, c'est-à-dire que quel que soit le chemin d'exécution, il existe un cas de test qui le couvre. L'ensemble de tests `TESTS` satisfait le critère *Tous les chemins* pour le modèle `M` si :

$$\texttt{TESTS} = \texttt{TRACES}$$

4.4 Positionnement UML4MBT et analyse

Comme vue dans la section 1.2, la génération de tests à partir de modèle UML4MBT est basée sur le critère de couverture *Toutes les transitions* (également sur le critère *Tous les états* mais celui-ci apportant une couverture moindre, il est écarté de cette discussion). Cette stratégie est renforcée par l'utilisation du critère D/CC (*Decision/Condition Coverage*) afin de couvrir les divers comportements attachés aux transitions. En effet, ce critère assure la couverture de toutes les conditions et toutes les décisions attachées à chaque transition. Cela signifie que pour chaque effet de chaque transition :

- Chaque condition de chaque structure conditionnelle (`if...then...else...`) doit être évaluée par au moins un test à `vrai` et par au moins un test à `faux`.
- Chaque décision de chacune de ces conditions doit également être évaluée par au moins un test à `vrai` et par au moins un test à `faux`.

La combinaison de ces deux critères est destinée aux modèles UML4MBT et ne prend pas en compte les spécificités des modèles SysML4MBT tels que les envois/réceptions de signaux. Ces éléments ne sont donc pas spécifiquement couverts si l'on applique la même stratégie que celle utilisée pour les modèles UML4MBT. Prenons le cas de l'exemple jouet présenté au début de ce chapitre.

Dans le cadre de cet exemple, toutes les transitions sont atomiques, c'est-à-dire qu'aucune ne contient de structure conditionnelle. Ainsi, l'application de la stratégie dédiée aux modèles UML4MBT sur cet exemple correspond à la couverture du modèle par le critère *Toutes les transitions*. Les tests générés dans ce cas sont représentés dans le tableau 4.1.

Cibles	Tests
`trA` Démarrer le train.	`T1` `[⟨start,trA,trA⟩]`
`trB` Arrêter le train.	`T2` `[⟨start,trA,trA⟩;` `⟨stop,trB,trB⟩]`
`trC` Enclencher le bouton du conducteur.	`T3` `[⟨driver,trC,[trC;trE]⟩]`
`trD` Enclencher le bouton des passagers.	`T4` `[⟨passengers,trD,[trD;trE]⟩]`
`trE` Arrêt d'urgence du train (train déjà à l'arrêt).	Déjà couvert par `T3` et `T4`.
`trF` Arrêt d'urgence du train (train en mouvement).	`T5` `[⟨start,trA,{trA}⟩;` `⟨driver,trC,[trC;trF]⟩]`

TABLE 4.1 – Tests générés par la stratégie *Toutes les transitions*+D/CC sur l'exemple jouet

Etant donné que le test `T1` est inclus dans le test `T2`, `T1` n'est pas nécessaire. Pour satisfaire le critère *Toutes les transitions* sur cet exemple, il est donc nécessaire de générer les quatre tests `T2`, `T3`, `T4` et `T5`.

On peut remarquer que cette approche n'est pas adaptée car le scénario consistant à déclencher l'arrêt d'urgence du train en mouvement en actionnant le bouton des passagers n'est pas généré. Dans un cadre critique tel que celui-ci, il est nécessaire de tester ce cas.

Afin de combler cette lacune, de nouveaux critères de couverture dédiés aux modèles SysML4MBT et prenant en compte leur particularité vis-à-vis d'UML4MBT, à savoir les signaux, ont été définis.

4.5 Critères de sélection de tests pour SysML4MBT

Les critères de couverture de la littérature présentés dans la section précédente et la hiérarchie présentée dans l'introduction sont dédiés aux modèles de spécification qui sont composés exclusivement de transitions manipulant des variables. Or, en SysML4MBT, il est possible et courant de manipuler des signaux. La couverture des transmissions de ces signaux n'étant pas prévue par ce type de critère, nous avons défini de nouveaux critères, basés sur les mêmes principes que les critères DU (*Tous les DU-Chemins* et *Toutes les DU*), afin de considérer la couverture des envois/réceptions de signaux. La stratégie visant à appliquer ce type de critère est appelée ComCover (*COMmunication COVERage*). La hiérarchie revisitée de critères, adaptée aux modèles SysML4MBT, est représentée sur la figure 4.9.

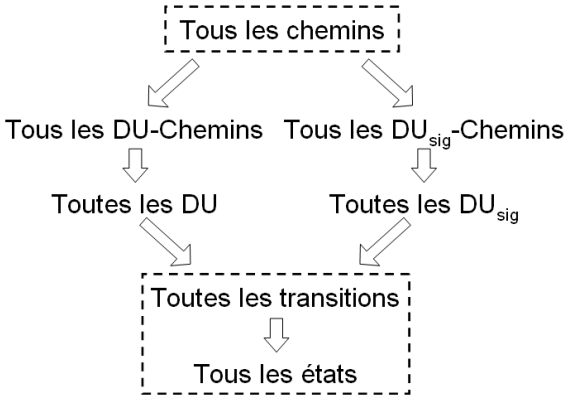


FIGURE 4.9 – Hiérarchie des critères de couverture des modèles SysML4MBT

Nous définissons à présent les critères de couverture ajoutés pour répondre à la problématique de couverture des envois/réceptions de signaux modélisés dans les modèles de spécification SysML4MBT ainsi que D/CC.

4.5.1 Critère D/CC

Le critère *Decision Condition Coverage* assure la couverture de toutes les décisions et de toutes les conditions des transitions des diagrammes d'états/transitions. Il s'agit d'un critère très utilisé dans le domaine de la génération de tests à partir de critère. Afin d'assurer la couverture de ce critère, il faut commencer par récupérer tous les comportements de toutes les transitions du modèle. Ensuite, pour chacun de ces comportements, il faut tout d'abord qu'un cas de test évalue la décision de ce comportement à `vrai`, et qu'un autre évalue la décision à `faux`. Ensuite, pour chacune de ces décisions, il faut également que chaque partie de la décision soit elle même évaluée au moins une fois à `vrai` et une fois à `faux` dans l'ensemble des tests.

Afin de formaliser ce critère, une réécriture permettant de détailler chaque comportement du système est effectuée. Pour ce faire, les réécritures définies dans le chapitre 6 de [Peu02] sont réutilisées. Ainsi, l'ensemble de tests `TESTS` satisfait le critère D/CC pour le modèle `M` s'il respecte l'expression représentée sur la figure 4.10.

```
∀ bhv.(bhv ∈ {b|∃ (sm,t).(sm ∈ M.SMS ∧ t ∈ sm.TRANS ∧ b ∈ t.TRbhvs)}
  ⇒ ∃ bhvTest.
        (bhvTest ∈ {b|∃ (t,pas).(t ∈ TESTS ∧ pas ∈ t ∧ b ∈ pas.AllBhvs)}
        ∧ bhvTest=bhv))
```

FIGURE 4.10 – Formalisation du critère D/CC

4.5.2 Critère *Toutes les DU_{sig}*

DU_{sig} signifie définition/utilisation des signaux. Le critère *Toutes les DU_{sig}* assure la couverture de toutes les paires envoi/réception de signaux, c'est-à-dire que pour tout comportement qui contient un envoi de signal sur un port, pour chacun des comportements qui réceptionne ce signal sur ce port, il existe un cas de test passant par le comportement d'envoi puis par le comportement de réception. A l'instar du critère *Toutes les DU*, on définit ce critère comme une surcharge du critère *Toutes les transitions*. De ce fait, l'ensemble de tests `TESTS` satisfait le critère *Toutes les DU_{sig}* pour le modèle `M` s'il satisfait les expressions énoncées figures 4.6 et 4.11.

```
∀ (sig,port,bhvSend,bhvRec).
   ((sig ∈ M.BDD.SIGS ∧ port ∈ M.allPorts()
   ∧ bhvSend ∈ {b|∃ (sm,t).(sm ∈ M.SMS ∧ t ∈ sm.TRANS
                ∧ b ∈ t.TRbhvs)}
   ∧ bhvRec ∈ {b|∃ (sm,t).(sm ∈ M.SMS ∧ t ∈ sm.TRANS
                ∧ b ∈ t.TRbhvs)}
   ∧ ⟨sig,port⟩ ∈ bhvSend.BHVaction ∧ bhvRec.tr.TRtrig=⟨sig,port⟩)

   ⇒ ∃ pas.
        (pas ∈ {p|∃ t.(t ∈ TESTS ∧ p ∈ t)}
        ∧ bhvSend ∈ pas.AllBhvs ∧ bhvRec ∈ pas.AllBhvs
        ∧ bhvSend <pas.AllBhvs bhvRec))
```

Pour tous les comportements d'envois et de réceptions associés :

Il existe un pas de test qui franchit l'envoi puis la réception.

FIGURE 4.11 – Formalisation du critère *Toutes les* DU_{sig}

4.5.3 Critère *Tous les* DU_{sig}*-Chemins*

Le critère *Tous les* DU_{sig}*-Chemins* assure la couverture de tous les chemins possibles pour chaque paire émission/réception de signal. L'ensemble de test `TESTS` satisfait le critère *Tous les* DU_{sig}*-Chemins* pour le modèle `M` s'il satisfait les expressions énoncées figures 4.6 et 4.12.

```
∀ (sig,port,bhvSend,bhvRec,trace).
   ((sig ∈ M.BDD.SIGS ∧ port ∈ M.allPorts()
   ∧ bhvSend ∈ {b|∃ (sm,t).(sm ∈ M.SMS ∧ t ∈ sm.TRANS
                ∧ b ∈ t.TRbhvs)}
   ∧ bhvRec ∈ {b|∃ (sm,t).(sm ∈ M.SMS ∧ t ∈ sm.TRANS
                ∧ b ∈ t.TRbhvs)}
   ∧ ⟨sig,port⟩ ∈ bhvSend.BHVaction ∧ bhvRec.tr.TRtrig=⟨sig,port⟩
   ∧ trace ∈ TRACES
   ∧ ∃ pas.(pas ∈ trace ∧ bhvSend ∈ pas.AllBhvs
           ∧ bhvRec ∈ pas.AllBhvs
           ∧ bhvSend <pas.allBhvs bhvRec))

   ⇒ trace ∈ TESTS)
```

Pour tous comportements d'envois et de réceptions associés et

pour toute trace contenant un pas qui franchit l'envoi puis la réception :

la trace doit être testée.

FIGURE 4.12 – Formalisation du critère *Toutes les* DU_{sig}

4.5.4 ComCover

Ces nouveaux critères (*Toutes les DU_{sig}* et *Tous les DU_{sig}-Chemins*) sont dédiés aux modèles SysML. Les critères de type *Tous chemins* (tels que *Tous les DU-Chemins* et *Tous les DU_{sig}-Chemins*) présentent des problèmes d'explosion combinatoire connus [Wei10]. En effet, l'exploitation de ce type de critère aboutit à la génération d'un nombre infini de cas de test si un diagramme d'états-transitions contient au moins une boucle (ou qu'une boucle peut se former en combinant les diagrammes d'états-transitions). Ainsi, seul le critère *Toutes les DU_{sig}* est exploité au sein de ces travaux. La stratégie de génération de tests visant à couvrir le modèle à l'aide de ce critère est appelée *ComCover*.

Il est possible de déterminer l'intervalle encadrant le nombre de cibles de tests générées par cette stratégie, une cible correspondant à un couple de transitions émission/réception. Toutes les cibles doivent être couvertes par des tests pour assurer la couverture du critère *Toutes les DU_{sig}*. Par défaut, il faut générer autant de tests que de cibles. Le nombre total de test générés peut cependant être inférieur au nombre de cibles, un test permettant parfois la couverture de plusieurs cibles.

Un modèle SysML4MBT quelconque peut tout à fait contenir des émissions de signaux sans réception correspondante (et vice-versa). Le nombre de cibles peut donc tout à fait être égal à zéro si aucune réception n'est associée aux émissions modélisées. Dans la suite de cette section sont uniquement considérés les modèles tels que pour toute émission de signal, il existe au moins une réception de ce signal concordante et vice-versa. Dans ce cadre, il est possible d'affirmer que pour un modèle qui contient un nombre e d'envois de signaux et un nombre r de réceptions de signaux, si *cibles* est le nombre de couples générés par la stratégie *ComCover*, on a :

$$\boxed{max(e, r) \leqslant cibles \leqslant e * r}$$

où $max(e, r)$ représente la plus grande valeur entre e et r. Si $e < r$ (resp. $e > r$), le nombre de cibles minimum correspond au cas où chaque réception (resp. émission) est associée à une unique émission (resp. réception). Dans ce cas, la combinatoire est minimum. A l'opposé, la valeur $e * r$ représente le cas où chaque émission concorde avec chaque réception modélisée. Ainsi, on obtient la combinatoire la plus élevée, c'est-à-dire la configuration générant le plus de combinaisons possibles.

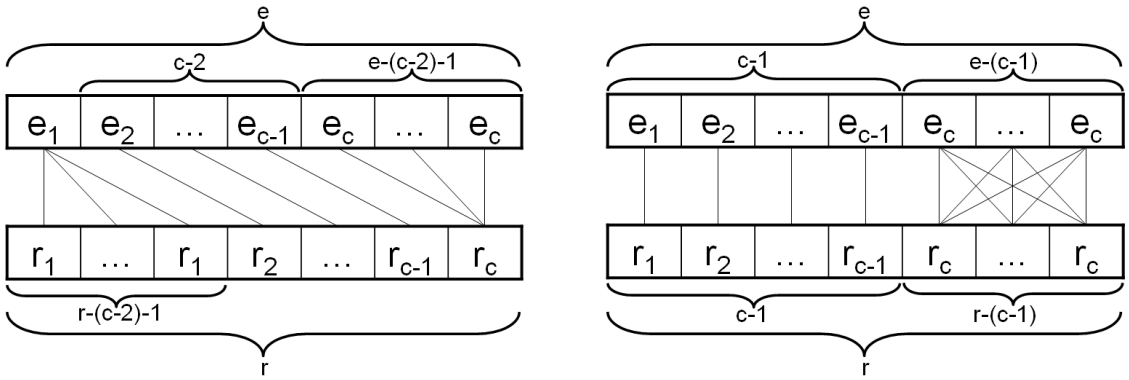
Il est cependant possible d'affiner cet intervalle en prenant en compte les connecteurs modélisés dans le diagramme interne de bloc. En effet, les connecteurs impliquent une distribution minimum des envois/réceptions de signaux. Il s'agit cependant de prendre uniquement en compte les connecteurs opérationnels, c'est-à-dire les connecteurs au travers desquels des signaux transitent effectivement d'après les diagrammes d'états-transitions.

De plus, il est nécessaire de détailler les connecteurs modélisés en connecteurs élémentaires, à savoir en connecteurs unidirectionnels qui ne peuvent envoyer qu'un seul signal. Finalement, deux connecteurs qui permettent d'envoyer le même signal sur le même port sont fusionnés car ils correspondent aux mêmes réceptions de signaux. Ainsi, une fois le décompte des connecteurs réalisé selon ces règles, il est possible d'établir que pour un modèle qui contient e émissions de signaux, r réception de signaux, et c connecteurs élémentaires opérationnels, le nombre de cibles générées par ComCover vérifie :

$$\boxed{r + e - c \leqslant cibles \leqslant (e - c + 1) * (r - c + 1) + (c - 1)}$$

Cet intervalle se justifie par la définition de la meilleure et de la pire (au sens du nombre de cibles générées) distribution des envois/réceptions au niveau des connecteurs élémentaires. En effet, la distribution de e envois et r réceptions sur c connecteurs, qui permet de générer un nombre minimum de couples, correspond à une distribution dans laquelle chaque envoi est associé à une seule réception ou à défaut, dans laquelle les réceptions qui sont associées au même envoi sont associées uniquement à celui-là.

Schématiquement, pour c connecteurs, il y a forcément c types d'envois et c types de réception. Il peut ensuite y avoir plusieurs occurrences pour chacun de ces types (au moins un). Soit par exemple e_i un envoi de signal sur le i^{eme} connecteur. Cet envoi est associé uniquement aux réceptions r_i. La distribution permettant l'obtention d'un nombre minimal de cibles peut se représenter comme sur la figure 4.13a. En ce qui concerne la borne maximale, elle correspond à la distribution représentée sur la figure 4.13b.



(a) Nombre de cibles minimum

(b) Nombre de cibles maximum

FIGURE 4.13 – Distribution des envois/réceptions de signaux par connecteur

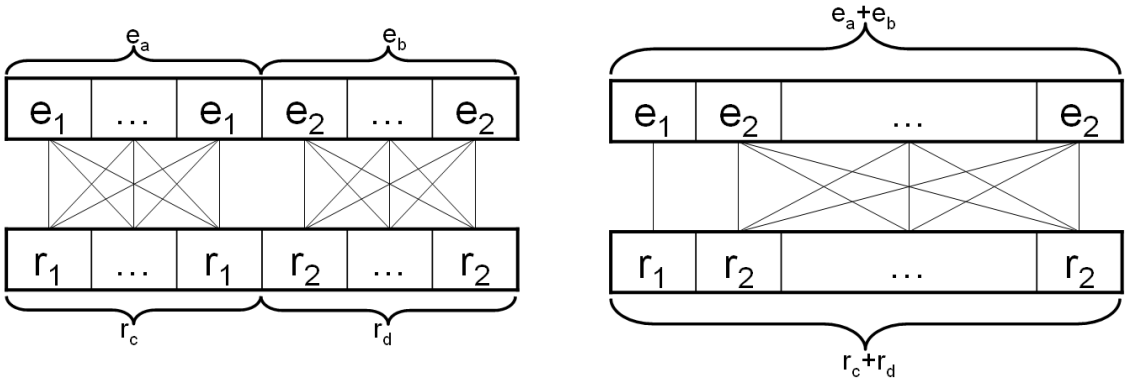
Il est impossible de générer moins de couples que dans la solution minimale car cela signifierait qu'un envoi n'est associé à aucune réception (ou l'inverse). On obtient donc la formule et la simplification suivantes :

$$\begin{aligned} cibles\ min &= r - (c-2) - 1 + c - 2 + e - (c-2) - 1 \\ &= r - c + 2 - 1 + c - 2 + e - c + 2 - 1 \\ &= r + e - c + c - c + 2 - 1 - 2 + 2 - 1 \\ &= r + e - c \end{aligned}$$

La borne maximale correspond à la situation dans laquelle une unique émission et une unique réception sont associées à chaque connecteur excepté pour un connecteur auquel sont associés tous les autres. On obtient ainsi :

$$cibles\ max = (e - c + 1) * (r - c + 1) + (c - 1)$$

Afin de démontrer que cette situation est bien celle qui permet la génération du plus grand nombre de cibles, il suffit de montrer qu'une distribution quelconque des émissions/réceptions est moins efficace (cf. figure 4.14a) que de privilégier au maximum un connecteur (cf. figure 4.14b).



(a) Distribution quelconque (b) Distribution en privilégiant un connecteur

FIGURE 4.14 – Distribution des envois/réceptions de signaux sur deux connecteurs pour la démonstration du pire cas

Dans le cadre du cas de la figure 4.14a, on fixe e_a (resp. r_c) le nombre d'envois (resp. de réceptions) associés au premier connecteur et e_b (resp. r_d), le nombre d'envois (resp. de réceptions) associés au deuxième connecteur. On obtient ainsi un nombre de couples égal à :

$$e_a * r_c + e_b * r_d$$

En conservant le même nombre d'envois et de réceptions, avec une distribution privilégiant au maximum un connecteur (cf figure 4.14b), on obtient le nombre de couples suivant :

$$1 + (e_a + e_b - 1) * (r_c + r_d - 1)$$

(e_a, e_b, r_c et r_d étant supérieurs ou égaux à 1)

Il s'agit donc de montrer que la deuxième solution donne au moins autant de couples que la première, c'est-à-dire que :

$$\begin{array}{l} e_a * r_c + e_b * r_d \leqslant 1 + (e_a + e_b - 1) * (r_c + r_d - 1) \\ \Leftrightarrow e_a * r_c + e_b * r_d \leqslant 1 + e_a * r_c + e_a * r_d - e_a + e_b * r_c + e_b * r_d - e_b - r_c - r_d + 1 \\ (e_a * r_c \text{ et } e_b * r_d \text{ présents des deux côtés}) \\ \Leftrightarrow 0 \leqslant 1 + e_a * r_d - e_a + e_b * r_c - e_b - r_c - r_d + 1 \\ \Leftrightarrow 0 \leqslant (e_a * r_d - e_a - r_d + 1) + (e_b * r_c - e_b - r_c + 1) \end{array}$$

En ce qui concerne cette dernière expression, chaque partie $(e_a * r_d - e_a - r_d + 1)$ et $(e_b * r_c - e_b - r_c + 1)$ est de la forme :

$$a * b - a - b + 1$$

(a et b étant des entiers strictement supérieurs à zéro). On démontre que cette expression est supérieure ou égale à zéro par récurrence. a et b étant des entiers strictement positifs, la valeur minimale pour chacune de ces variables est 1. Pour $a = 1$ et $b = 1$:

$$0 \leqslant 1 * 1 - 1 - 1 + 1 \Leftrightarrow 0 \leqslant 0$$

Ensuite, si on considère que $a * b - a - b + 1 \leqslant 0$ est vrai pour une valeur quelconque de a et de b, et a et b ayant le même poids dans l'expression, il suffit de vérifier que cette propriété est toujours vraie pour $a + 1$ par exemple soit :

$$\begin{array}{l} 0 \leqslant (a + 1) * b - (a + 1) - b + 1 \\ \Leftrightarrow 0 \leqslant ab + b - a - 1 - b + 1 \\ \Leftrightarrow 0 \leqslant ab - a \\ \Leftrightarrow a \leqslant ab \end{array}$$

Cette expression est vraie car b est un entier strictement positif. Il est donc possible d'affirmer que le pire cas énoncé précédemment est correct. De ce fait, quelle que soit la distribution des envois/réceptions de signaux par rapport aux connecteurs, le nombre de cibles générées par la stratégie ComCover est :

$$r + e - c \leqslant \mathit{cibles} \leqslant (e - c + 1) * (r - c + 1) + (c - 1)$$

Cet intervalle permet de se faire une idée du nombre de cibles générés par ComCover sur un modèle donné et ainsi pouvoir évaluer l'effort de validation nécessaire.

4.5.5 Illustration

Afin de démontrer l'intérêt de ce nouveau critère sur les modèles SysML4MBT, voici un ensemble de tests pour assurer la couverture du critère *Toutes les DU_{sig}* sur l'exemple jouet.

Ce modèle contient les envois de signaux : `SendStop` sur la transition `trC` et `SendStop` sur la transition `trD`. Le signal est le même dans les deux cas. La réception de ce signal permet d'activer deux transitions différentes : `trE` et `trF`. Il y a de ce fait quatre cibles de test :

- Le couple `C1` par le franchissement de `trC` suivi de `trE`.
- Le couple `C2` par le franchissement de `trC` suivi de `trF`.
- Le couple `C3` par le franchissement de `trD` suivi de `trE`.
- Le couple `C4` par le franchissement de `trD` suivi de `trF`.

Afin d'atteindre ces quatre cibles, il est nécessaire de générer les tests présents dans le tableau 4.2.

Cibles	Tests
`C1` Enclencher le bouton du conducteur lorsque le train est à l'arrêt.	`T6` `[⟨driver,trC,[trC;trE]⟩]`
`C2` Enclencher le bouton du conducteur lorsque le train est en mouvement.	`T7` `[⟨start,trA,trA⟩;` `⟨driver,trC,[trC;trF]⟩]`
`C3` Enclencher le bouton des passagers lorsque le train est à l'arrêt.	`T8` `[⟨passengers,trD,[trD;trE]⟩]`
`C4` Enclencher le bouton des passagers lorsque le train est en mouvement.	`T9` `[⟨start,trA,trA⟩;` `⟨passengers,trD,[trD;trF]⟩]`

TABLE 4.2 – Tests générés par ComCover sur l'exemple jouet

Etant donné que le critère *Toutes les DU_{sig}* est une extension au critère *Toutes les transitions*, ce dernier doit également être satisfait. Ainsi, le test `T10` est également généré afin de couvrir la transition `trB`.

```
T10 = [⟨start,trA,trA⟩;⟨stop,trB,trB⟩]
```

En comparaison des résultats obtenus avec la stratégie dédiée à UML4MBT sur ce même exemple, on constate que la séquence critique manquante dans le premier cas est bien représentée ici par la séquence `T9`. Les autres séquences sont présentes dans les

deux cas. Cette nouvelle stratégie nous permet donc d'assurer la couverture des envois et réceptions de signaux. De ce fait elle n'est pas adaptée aux systèmes peu communicants.

On peut vérifier les métriques présentées dans la section 4.5.4. Dans le cas de l'exemple jouet, un seul connecteur est instrumenté dans le modèle : le connecteur entre les boutons et le gestionnaire d'arrêt d'urgence. Il y a deux envois de signaux (un depuis la transition `trC` et un depuis la transitions `trD`). Il y a deux réceptions de signaux (un par la transition `trE` et un par la transition `trF`). En appliquant ces valeurs à la formule présentée précédemment :

$$
\begin{array}{|rl|}
\hline
 & r+e-c \leqslant \mathit{cibles} \leqslant (e-c+1)*(r-c+1)+(c-1) \\
\Leftrightarrow & 2+2-1 \leqslant \mathit{cibles} \leqslant (2-1+1)*(2-1+1)+(1-1) \\
\Leftrightarrow & 3 \leqslant \mathit{cibles} \leqslant 2*2+0 \\
\Leftrightarrow & 3 \leqslant \mathit{cibles} \leqslant 4 \\
\hline
\end{array}
$$

Dans ce cas, 4 cibles ont été générées. La formule est donc vérifiée dans ce cas. La section suivante présente les résultats de l'application de la stratégie ComCover sur l'exemple fil rouge.

4.6 Exemple fil rouge

Afin de se rendre compte de la pertinence de la stratégie ComCover, nous l'appliquons sur l'exemple fil rouge. Dans le cadre de ce mémoire, la première étape consiste à numéroter les transitions (et les comportements) afin de pouvoir exprimer clairement les tests générés.

4.6.1 Numérotation et rappels

Afin de définir les cibles de test engendrées par l'application de ComCover, il est bon de se remémorer le diagramme interne de bloc (Fig. 4.15) afin de visualiser les canaux de transmissions de signaux. Dans cet exemple, la partie `Ignition` peut transmettre des signaux aux lumières `Lights` ainsi qu'au panneau de contrôle `ControlPanel`. Le `ControlPanel` est ensuite capable de transmettre des signaux vers la partie `Lights`.

Les trois diagrammes d'états-transitions sont rappelés ci-dessous. Chaque transition et comportement a été numéroté afin de pouvoir exprimer clairement les tests générés : `TRa` et `TRb` pour le diagramme d'états-transitions de `Ignition` (Fig. 4.16), `TR1` à `TR8` (`TR7` contenant quatre comportements : de `TR7.1` à `TR7.4`) pour la partie représentant le panneau de contrôle (Fig. 4.17) et enfin, de `TRA` à `TRH` pour le diagramme d'états-transitions des lumières (Fig. 4.18).

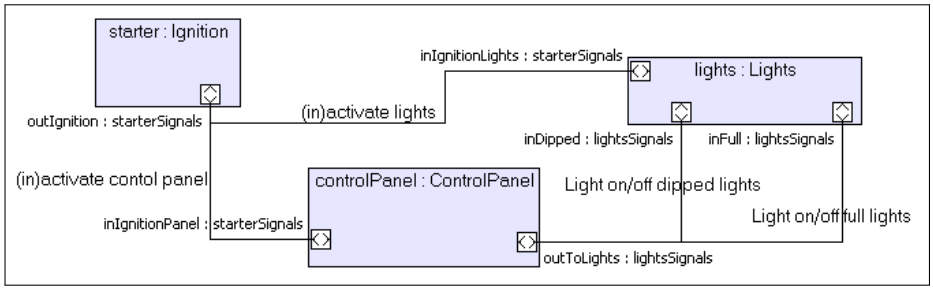


FIGURE 4.15 – Diagramme interne de bloc du modèle SysML4MBT de l'exemple fil rouge

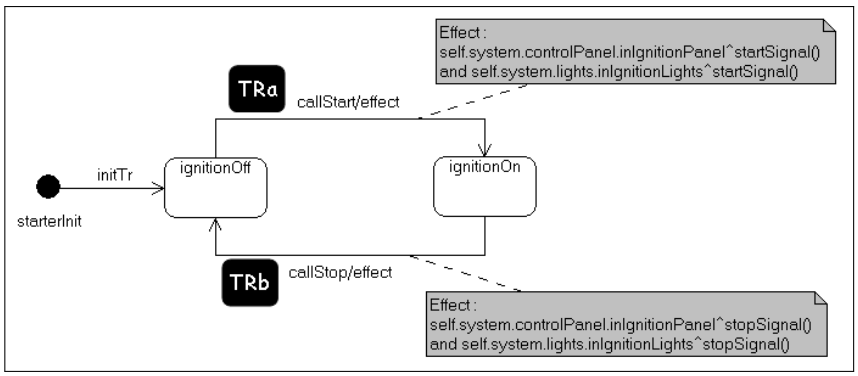


FIGURE 4.16 – Diagramme d'états-transitions du bloc `Ignition` du modèle SysML4MBT de l'exemple fil rouge

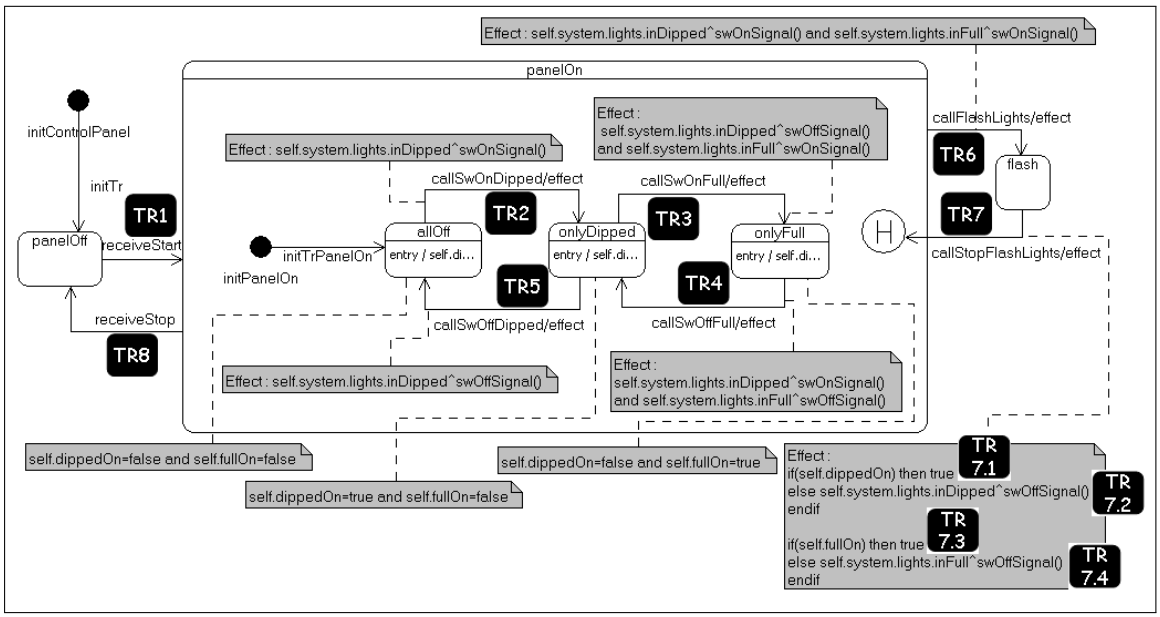


FIGURE 4.17 – Diagramme d'états-transitions du bloc `ControlPanel` du modèle SysML4MBT de l'exemple fil rouge

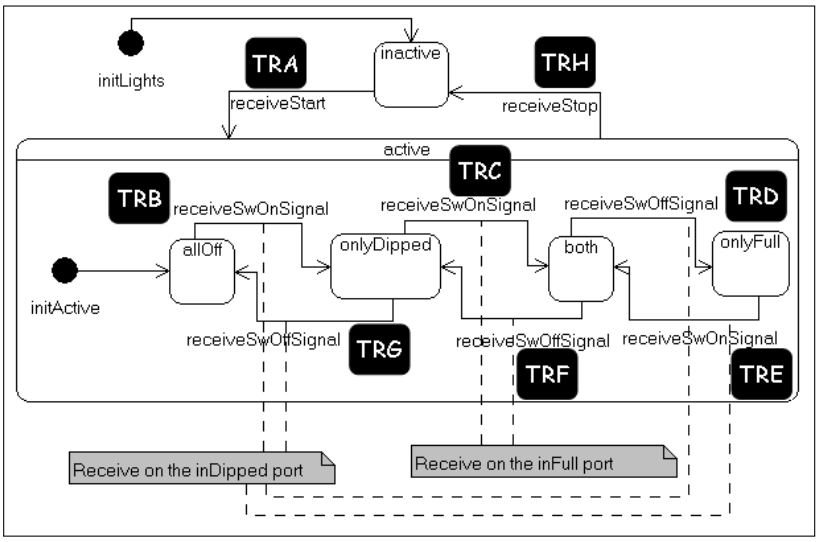


FIGURE 4.18 – Diagramme d'états-transitions du bloc `Lights` du modèle SysML4MBT de l'exemple fil rouge

4.6.2 Tests générés par la stratégie *Toutes les transitions*+D/CC

En ce qui concerne la stratégie appliquée sur UML4MBT, elle vise à couvrir l'ensemble des comportements du système. Parmi les tests générés, tous ceux qui sont contenus dans d'autres sont supprimés. Les tests représentés dans le tableau 4.3 sont ainsi obtenus.

4.6.3 Tests générés par ComCover

La stratégie ComCover consiste à couvrir toutes les communications modélisées. Pour ce faire, tous les envois de signaux sont extraits du modèle. Les transitions concernées sont `TRa`, `TRb`, `TR2`, `TR3`, `TR4`, `TR5`, `TR6`, `TR7.2` et `TR7.4`. Ensuite, pour chacun de ces comportements, on associe tous les comportements qui permettent la réception des signaux envoyés et le test correspondant est généré. Comme pour les tests générés dans le cadre de la stratégie mise en place pour les modèles UML4MBT, on représente ici (Fig. 4.4) uniquement les tests restants après suppression des tests qui sont inclus dans d'autres.

4.6.4 Vérification des métriques

Cette sous section présente la vérification de la formule présentée dans la partie 4.5.4 :

$$\boxed{r + e - c \leqslant \mathit{cibles} \leqslant (e - c + 1) * (r - c + 1) + (c - 1)}$$

Dans le cas de l'exemple fil rouge, il y a 4 connecteurs dans le diagramme d'états-transitions et ils sont tous utilisés dans les diagrammes d'états-transitions. Ils sont tous

Cibles	Tests
TRb Extinction du véhicule. (TRa, TR1, TRA, TR8, TRH)	T1 [⟨callStart,TRa,[TRa;{TR1,TRA}]⟩; ⟨callStop,TRb,[TRb;{TR8,TRH}]⟩]
TR7.1 Appel de phares en feux de croisement. (TR2, TRB, TR6, TRC, TR7.4, TRF)	T2 [⟨callStart,TRa,[TRa;{TR1,TRA}]⟩; ⟨callSwOnDipped,TR2,[TR2;TRB]⟩; ⟨callFlashLights,TR6,[TR6;TRC]⟩; ⟨callStopFlashLights,TR7,[TR7.1;TR7.4;TRF]⟩]
TR7.3 Appel de phares en phares. (TR3, TRD, TRE, TR7.2)	T3 [⟨callStart,TRa,[TRa;{TR1,TRA}]⟩; ⟨callSwOnDipped,TR2,[TR2;TRB]⟩; ⟨callSwOnFull,TR3,[TR3;TRC;TRD]⟩; ⟨callFlashLights,TR6,[TR6;TRE]⟩; ⟨callStopFlashLights,TR7,[TR7.2;TR7.3;TRD]⟩]
TR5 Extinction des feux de croisement. (TRG)	T4 [⟨callStart,TRa,[TRa;{TR1,TRA}]⟩; ⟨callSwOnDipped,TR2,[TR2;TRB]⟩; ⟨callSwOffDipped,TR5,[TR5;TRG]⟩]
TR4 Passage de phares à feux de croisement.	T5 [⟨callStart,TRa,[TRa;{TR1,TRA}]⟩; ⟨callSwOnDipped,TR2,[TR2;TRB]⟩; ⟨callSwOnFull,TR3,[TR3;TRC;TRD]⟩; ⟨callSwOffFull,TR4,[TR4;TRE;TRF]⟩]

TABLE 4.3 – Tests générés par la stratégie *Toutes les transitions*+D/CC sur l'exemple fil rouge

unidirectionnels. Chacun d'eux peut véhiculer deux signaux différents. Ainsi, le nombre de connecteurs élémentaires est $c = 8$.

Il s'agit maintenant de totaliser les envois et les réceptions. Dans le diagramme d'états-transitions du bloc `Ignition`, il y a 4 envois (2 sur `TRa` et 2 sur `TRb`). Dans le diagramme d'états-transitions du bloc `ControlPanel`, il y a 10 envois de signaux (`TR2`, `TR3`(*2), `TR4`(*2), `TR5`, `TR6` (*2) et `TR7` (*2)) et 2 réceptions (`TR1` et `TR8`). Enfin, au niveau du diagramme d'états-transitions des lumières, il y a 8 réceptions de signaux (1 sur chaque transition). Il y a donc au total 14 envois ($e = 14$) et 10 réceptions ($r = 10$).

$$
\begin{aligned}
& r + e - c \leqslant \mathit{cibles} \leqslant (e - c + 1) * (r - c + 1) + (c - 1) \\
\Leftrightarrow\ & 10 + 14 - 8 \leqslant \mathit{cibles} \leqslant (14 - 8 + 1) * (10 - 8 + 1) + (8 - 1) \\
\Leftrightarrow\ & 16 \leqslant \mathit{cibles} \leqslant 7 * 3 + 7 \\
\Leftrightarrow\ & 16 \leqslant \mathit{cibles} \leqslant 28
\end{aligned}
$$

Sur l'exemple fil rouge, 20 couples ont été générés. L'intervalle est donc respecté.

Cibles	Tests	
TRb/TR8 Extinction du véhicule. (TRa/TR1, TRa/TRA, TRb/TRH)	T6	[⟨callStart,TRa,[TRa;{TR1,TRA}]⟩; ⟨callStop,TRb,[TRb;{TR8,TRH}]⟩]
TR7.4/TRF Appel de phares en feux de croisement. (TR2/TRB, TR6/TRC)	T7	[⟨callStart,TRa,[TRa;{TR1,TRA}]⟩; ⟨callSwOnDipped,TR2,[TR2;TRB]⟩; ⟨callFlashLights,TR6,[TR6;TRC]⟩; ⟨callStopFlashLights,TR7,[TR7.1;TR7.4;TRF]⟩]
TR7.2/TRD Appel de phares depuis phares. (TR3/TRC, TR3/TRD, TR6/TRE)	T8	[⟨callStart,TRa,[TRa;{TR1,TRA}]⟩; ⟨callSwOnDipped,TR2,[TR2;TRB]⟩; ⟨callSwOnFull,TR3,[TR3;TRC;TRD]⟩; ⟨callFlashLights,TR6,[TR6;TRE]⟩; ⟨callStopFlashLights,TR7,[TR7.2;TR7.3;TRD]⟩]
TR7.2/TRG Appel de phares à partir de tout éteint. (TR6/TRB)	T9	[⟨callStart,TRa,[TRa;{TR1,TRA}]⟩; ⟨callFlashLights,TR6,[TR6;TRB;TRC]⟩; ⟨callStopFlashLights,TR7,[TR7.2;TR7.4;TRF;TRG]⟩]
TR5/TRG Extinction des feux de croisement.	T10	[⟨callStart,TRa,[TRa;{TR1,TRA}]⟩; ⟨callSwOnDipped,TR2,[TR2;TRB]⟩; ⟨callSwOffDipped,TR5,[TR5;TRG]⟩]
TR4/TRE Passage de phares à feux de croisement. (TR4/TRF)	T11	[⟨callStart,TRa,[TRa;{TR1,TRA}]⟩; ⟨callSwOnDipped,TR2,[TR2;TRB]⟩; ⟨callSwOnFull,TR3,[TR3;TRC;TRD]⟩; ⟨callSwOffFull,TR4,[TR4;TRE;TRF]⟩]
TR2/TRE - TR5/TRD TR4/TRB - TR3/TRG	Impossibles.	

TABLE 4.4 – Tests générés par la stratégie ComCover sur l'exemple fil rouge

4.7 Synthèse

Cette partie a présenté les critères de couverture de référence ainsi que de nouveaux critères dédiés aux modèles SysML4MBT. Ces nouveaux critères, basés sur les critères *Toutes les DU* et *Tous les DU-chemins* s'intéressent à la couverture des communications modélisées. Le critère *Toutes les* DU_{sig} assure ainsi la couverture de tous les couples d'envois/réceptions de signaux. Le critère *Tous les* DU_{sig}*-chemins*, quant à lui, propose la couverture de tous les chemins couvrant les couples d'envois/réceptions. Ce deuxième critère, posant des problèmes d'explosion combinatoire en cas de modèle contenant des boucles, a été écarté de nos expérimentations au profit du premier.

La stratégie ComCover mise en place correspond ainsi à l'application du critère *Toutes les* DU_{sig}. Elle permet de garantir une couverture complète des communications modélisées. Son application sur l'exemple fil rouge nous permet de constater son efficacité quant

à la production de tests inédits pertinents comparée aux critères de la littérature, tels ceux utilisés par l'outil Test Designer. Par exemple, le test consistant à faire un appel de phares tous feux éteints n'est pas généré dans le cas de l'application des critères de couverture *Toutes les transitions* appuyé par D/CC.

A l'instar du critère *Toutes les DU* duquel il est inspiré, le critère *Toutes les* DU_{sig} engendre dans certains cas la génération de cibles de test inaccessibles (trois cibles dans le cas de l'exemple fil rouge). C'est le cas lorsqu'un couple associe une émission à une réception modélisée uniquement pour être utilisée par d'autres émissions. Toutefois, ces configurations ne nuisent pas à la viabilité de la solution proposée du fait de leur détection aisée par le moteur de calcul de tests.

Troisième partie

Réalisations et expérimentations

Chapitre 5

Implémentation

Sommaire

Les chapitres précédents ont présenté l'ensemble des éléments théoriques constituant le processus mis en place durant nos travaux. Ce chapitre décrit l'implémentation de ces éléments, c'est-à-dire, la mise en place pratique des éléments définis précédemment. Les principales étapes du processus sont la modélisation du système sous test sous la forme d'un modèle SysML4MBT puis la génération de tests par application de la stratégie ComCover sur ce modèle.

Pour la phase de modélisation du système, le modeleur (basé sur Eclipse) Topcased [TOP10] a été choisi. Il s'agit du modeleur gratuit et open-source le plus complet à ce jour pour la réalisation de modèles SysML. Concernant la génération de tests, notre démarche s'est appuyée sur les travaux antérieurs, menés conjointement par le LIFC et la société Smartesting, à travers l'outil Test Designer^{TM}. En effet, ce logiciel, basé entre autres sur [Gra08], permet à partir d'un modèle UML4MBT de générer des tests assurant la couverture du modèle par les critères *Toutes transitions* et D/CC. Afin d'expérimenter, les éléments définis précédemment, notre approche a consisté à utiliser cet outil en l'état. De ce fait, des règles de réécriture permettant de transformer un modèle SysML4MBT en modèle UML4MBT ont été définies. La transformation de modèles est une pratique très répandue dans le domaine de l'ingénierie dirigée par les modèles. Par exemple, elle permet d'effectuer la vérification de modèle [ALL09] ou de la génération de code [Old04]. La transformation effectuée dans notre cas doit conserver les éléments qui sont nécessaires à l'application de la stratégie ComCover (transmissions de signaux). Ces éléments n'étant pas pris en compte par UML4MBT, des règles de transformation dédiées permettant de conserver l'information sous une autre forme ont été mises en place.

La transformation de modèle SysML4MBT vers UML4MBT ne suffit cependant pas à mettre en œuvre la stratégie ComCover. En effet, la stratégie implémentée au sein de Test Designer^{TM} vise à couvrir les critères *Toutes les transitions* et D/CC. Nous avons donc surchargé notre propre transformation de modèles afin de créer, au sein du modèle UML4MBT, des comportements spécifiques permettant de créer des cibles de test afin de conserver celles définies par la stratégie ComCover sur les modèles SysML4MBT [Wei10].

Ce chapitre présente tout d'abord les règles de réécriture permettant de transformer un modèle SysML4MBT en modèle UML4MBT (section 5.1) puis présente son application sur l'exemple fil rouge (section 5.2). La section 5.3 présente les règles supplémentaires permettant d'obtenir, en générant des tests avec l'outil TestDesigner^{TM} sur le modèle résultant, des tests équivalents (en terme de couverture de modèle) à ceux générés par la stratégie ComCover appliquée directement sur le modèle SysML4MBT. La dernière section dresse un bilan des notions vues dans ce chapitre.

5.1 Transformation de SysML4MBT vers UML4MBT

SysML4MBT et UML4MBT étant respectivement des sous-ensembles de SysML et UML, et SysML étant un profil d'UML, une manière classique de transformer un modèle SysML4MBT en un modèle UML4MBT consiste à supprimer le stéréotype SysML. Cependant, certains éléments sont autorisés en SysML4MBT mais n'ont pas de correspondance en UML4MBT (cf. section 3.3). En effet, SysML4MBT a été défini dans l'optique de proposer une adaptation d'UML4MBT pour la modélisation de systèmes embarqués. Ainsi, certains éléments propres à ce domaine non disponibles en UML4MBT peuvent être modélisés en SysML4MBT à l'aide d'entités spécifiques (signaux, ports...). La stratégie développée consiste à réécrire ces entités, à l'aide de règles dédiées, afin de conserver les informations nécessaires à l'application de la stratégie ComCover.

Concrètement, compte tenu des correspondances présentées section 3.3, la majorité des éléments du diagramme de bloc et les éléments séquentiels des diagrammes d'états-transitions peuvent être réécrits en supprimant le profil SysML. En ce qui concerne les autres entités, et particulièrement celles nécessaires au calcul des cibles de test pour l'application de la stratégie ComCover (cf. section 4.5), des règles de réécriture spécifiques sont nécessaires. Il s'agit en l'occurrence :

- du diagramme interne de bloc,
- des signaux du diagramme de bloc,
- des envois de signaux OCL (^) et des réceptions de signaux (transition déclenchée par une réception de signal) dans les diagrammes d'états-transitions,
- de toutes les entités modélisant du parallélisme autorisées dans les diagrammes d'états-transitions SysML4MBT (fork, join, états parallèles et diagrammes d'états-transitions multiples),
- des états historiques (diagramme d'états-transitions).

Concernant les diagrammes d'exigences (non présents en UML4MBT), bien qu'ils ne soient pas nécessaires à l'application de la stratégie ComCover, ils apportent néanmoins des informations relevantes dans une optique de validation de système en permettant notamment une traçabilité des exigences et l'évaluation de leur couverture. Des règles ont donc été définies pour permettre d'éventuels travaux autour de cette problématique.

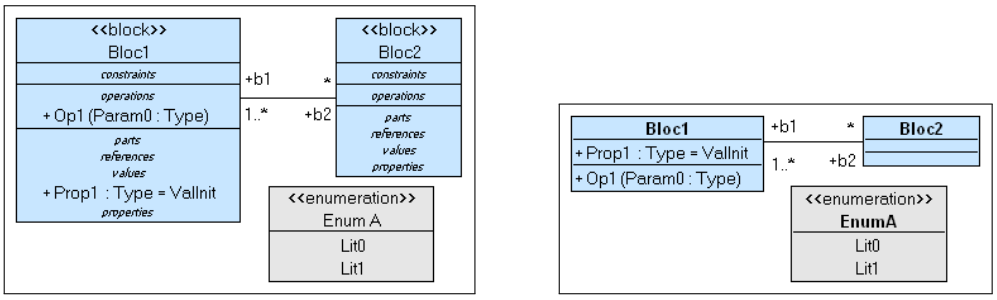
L'objectif étant d'obtenir un modèle UML4MBT complet, il faut également considérer le diagramme d'objets qui est nécessaire dans un modèle UML4MBT mais qui n'existe pas en SysML4MBT. Une étape de création de ce modèle a également été définie.

Cette section détaille les étapes de réécriture nécessaires à l'obtention d'un modèle UML4MBT à partir d'un modèle SysML4MBT. Une première sous-section présente l'en-

semble des éléments communs entre UML4MBT et SysML4MBT qui ne nécessitent pas de réécriture spécifique. La sous-section 5.1.2 explique les règles nécessaires à la réécriture des signaux et des ports. Ensuite, les règles de réécriture concernant les nœuds fork et join, les états historiques et parallèles ainsi que les diagrammes d'états-transitions multiples sont détaillées. Finalement, avant de conclure, les sous-sections 5.1.5 et 5.1.6 présentent respectivement la création du diagramme d'objets et la réécriture du diagramme d'exigences.

5.1.1 Sous-ensembles UML4MBT et SysML4MBT communs

Au niveau du diagramme de bloc SysML4MBT, les seuls éléments qui n'ont pas d'équivalents en UML4MBT sont les signaux, les ports et les spécifications de flux. Tous les autres éléments sont réécrits en supprimant le stéréotype SysML. De ce fait, un bloc devient une classe et les associations, opérations et attributs restent inchangés. Par exemple, le diagramme de bloc SysML4MBT de la figure 5.1a est retranscrit en diagramme de classes UML4MBT représenté sur la figure 5.1b.



(a) SysML4MBT (b) UML4MBT

FIGURE 5.1 – Transformation des blocs SysML4MBT en classes SysML4MBT

En ce qui concerne les diagrammes d'états-transitions, les éléments suivants sont autorisés aussi bien en SysML4MBT qu'en UML4MBT :

- les états initiaux,
- les états finaux,
- les états composites,
- toutes les transitions sans déclencheur ou déclenchées par la réception d'un appel d'opération,
- les expressions OCL qui ne contiennent pas l'accent circonflexe (^) représentant un envoi de signal. C'est le seul opérateur OCL qui n'est pas pris en compte dans UML4MBT par rapport à SysML4MBT.

Tous ces éléments peuvent donc être retranscrits à l'identique. Les autres éléments nécessitent, quant à eux, l'application de règles particulières énoncées dans la suite de ce chapitre.

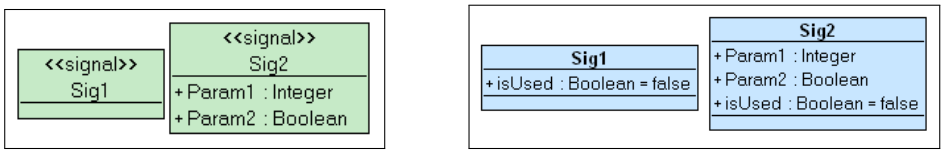
5.1.2 Signaux et ports

En SysML4MBT, les signaux sont représentés dans le diagramme de bloc, les ports sont modélisés (et reliés entre eux) dans le diagramme interne de bloc et tous deux sont mis en œuvre dans les diagrammes d'états-transitions. Les signaux et les ports, bien qu'autorisés en UML, ne sont pas pris en compte par UML4MBT. Des règles ont été définies afin de conserver les aspects sémantiques utiles à notre objectif (l'application de la stratégie ComCover) au niveau de ces entités.

Lors de l'animation du modèle pour l'application de la stratégie ComCover, il faut être en mesure de savoir quels signaux sont en attente sur quels ports à un moment donné. C'est dans cette optique que les réécritures suivantes sont effectuées.

Représentation statique d'un signal

Tout d'abord, chaque signal défini dans le diagramme de bloc est transformé en une classe du diagramme UML4MBT. Pour un signal donné, une classe portant le même nom est créée et les attributs du signal deviennent des attributs de classe. Pour la suite de la réécriture, il est également nécessaire d'ajouter, à chacune des classes créées dans ce cadre, un attribut de type booléen, nommé `isUsed`, et initialisé à `false`. Il permettra d'indiquer si le signal correspondant est en attente de réception ou non. La figure 5.2b représente le résultat de la transformation des signaux présentés par la figure 5.2a.



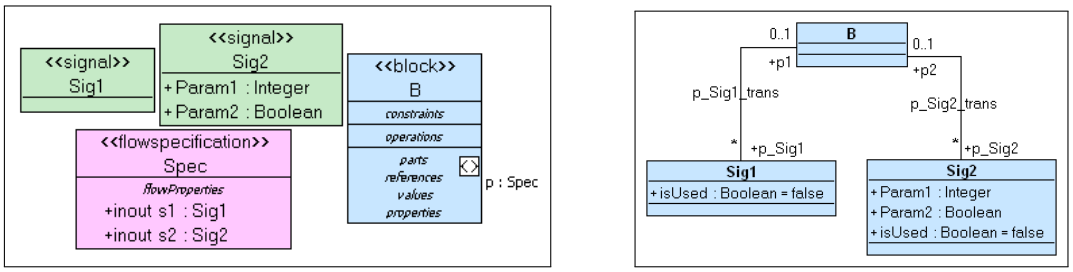
(a) SysML4MBT (b) UML4MBT

FIGURE 5.2 – Transformation des signaux SysML4MBT en classes UML4MBT

Représentation statique d'un port

La réécriture des ports se fait par la création d'associations. Ces associations symbolisent les signaux en attente sur les ports du bloc associé : la présence d'un lien entre un objet `port` et un objet `signal` caractérise que le signal donné est en attente sur le

dit port. Seuls les ports de type `IN` ou `IN/OUT` ont donc besoin d'être réécrits. Chaque couple signal/port (tel signal peut être reçu sur tel port) est représenté par une association reliant la classe qui représente le bloc qui héberge initialement le port et la classe qui modélise le signal concerné. Le rôle côté signal est nommé par la concaténation du nom du port et du nom du signal et porte la multiplicité `*`. L'autre extrémité porte `0..1` pour multiplicité et un nom quelconque. L'association porte également un nom quelconque. Les multiplicités sont déterminées ainsi car plusieurs instances différentes de signaux peuvent être en attente sur un même port à un moment donné (multiplicité `*`) mais une instance donnée d'un signal ne peut pas être en attente (donc reliée à un bloc) plusieurs fois en même temps. Par exemple, le port représenté sur la figure 5.3a est réécrit comme décrit sur la figure 5.3b.



(a) SysML4MBT (b) UML4MBT

FIGURE 5.3 – Transformation des ports SysML4MBT en associations UML4MBT

Emission de signal (Expression OCL ^)

Au sein des diagrammes d'états-transitions, les émissions de signaux sont représentées par l'opérateur OCL ^. Sa transformation se fait par l'instanciation de l'association définie dans la sous-section précédente ce qui signifie que le signal en question est envoyé sur ce port et est en attente de réception.

La figure 5.4a illustre l'expression OCL SysML4MBT représentant l'envoi du signal `Sig2` (avec les valeurs de paramètres `val1` et `val2`) sur le port `p` du bloc `B`. La réécriture de cette expression en UML4MBT se fait par l'instanciation de l'association qui relie la classe représentant le bloc (`B`) qui héberge le port destinataire (`p`) et la classe qui représente le signal transmis (`Sig2`), et qui a pour rôle *nom du port_nom du signal* (`p_Sig2`). Lors de la création du lien, l'objet représentant le signal sélectionné doit être inutilisé, c'est-à-dire que le paramètre `isUsed` doit être égal à `faux`. L'expression OCL d'envoi de signal présentée précédemment se réécrit donc comme sur la figure 5.4b.

```
B.p ^ Sig2(val1,val2)
```

(a) SysML4MBT

```
let s=Sig2.allInstances()
                → any(isUsed = false)
in s.Param1 = val1
    and s.Param2 = val2
    and s.isUsed = true
    and B.p_Sig2 → includes(s)
```

(b) UML4MBT

FIGURE 5.4 – Transformation d'un envoi de signal SysML4MBT en UML4MBT

Réception de signal

La réception de signal en SysML4MBT est représentée à l'aide d'un déclencheur de transition. Etant donné que l'envoi de signal est transcrit en UML4MBT par l'instanciation d'une association, la réception de signal est transcrite en garde de transition vérifiant que l'association correspondante a bien été instanciée en garde et en effet de transition afin de supprimer ce lien suite au franchissement de la transition.

Par exemple, un déclencheur de transition avec les caractéristiques représentées dans la figure 5.5a est réécrit en la garde représentée par la figure 5.5b et en l'effet représenté par la figure 5.5c.

```
Caractéristiques :
- port de réception : p
- bloc contenant le port : B
- signal recu : Sig2
```

(a) Déclencheur SysML4MBT

```
[B.p_Sig2→notEmpty()]
```

(b) Garde UML4MBT

```
let s =
   B.p_Sig2.allInstances()
                → any(true)
in s.isUsed=false
    and B.p_Sig2
                → excludes(s)
```

(c) Effet UML4MBT

FIGURE 5.5 – Transformation d'une réception de signal SysML4MBT en UML4MBT

5.1.3 Nœuds fork/join, états historiques et parallèles

Parmi les entités autorisées au sein des diagrammes d'états-transitions SysML4MBT, les suivantes ne sont pas autorisées dans les diagrammes d'états-transitions UML4MBT :

- les nœuds fork et join,
- les états parallèles,
- les états historiques (standard et profond).

Tout d'abord, afin de mutualiser les calculs et de limiter les étapes de transformation, et du fait de leur similarité, les nœuds fork et join sont transformés en états parallèles.

Ensuite, il est également utile de réécrire les états composites. En effet, pour supprimer les états parallèles sans perdre ni ajouter de comportements, nous réalisons un produit basé sur le produit cartésien entre les différentes régions. Afin de faciliter ce traitement, chacune des régions ne doit contenir aucune imbrication. De ce fait, les états composites sont également supprimés.

Les états composites, historiques et parallèles sont alors réécrits par ordre hiérarchique, c'est-à-dire que l'on réécrit une de ces entités uniquement s'il n'en contient pas d'autres d'un de ces types. Ainsi, pour supprimer un état composite (resp. parallèle), il ne doit pas contenir d'état parallèle (resp. composite). Si c'est le cas, les états *contenus* doivent être supprimés en premier lieu.

Nœuds fork et join

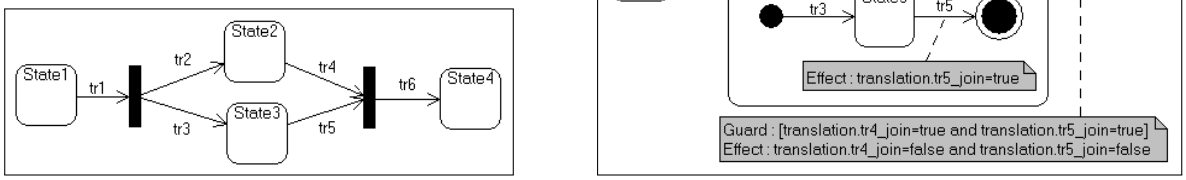
Les nœuds fork et join sont transformés en états parallèles afin de mutualiser la réécriture du parallélisme. La stratégie utilisée consiste à extraire chaque chemin d'exécution possible contenu entre le nœud fork et le nœud join et de le placer au sein d'une région d'un nouvel état parallèle. Ensuite, les transitions arrivant sur le nœud fork et partant du nœud join sont déplacées sur l'état parallèle. Enfin, il est nécessaire de reproduire la synchronisation en empêchant, à l'aide d'expressions OCL, de partir du nœud join avant que tous les chemins n'aient abouti.

Concrètement, avant la réécriture des nœuds fork/join, les éléments nécessaires à la synchronisation sont créés. Pour ce faire, de nouveaux attributs de classe permettant de contraindre les transitions quittant les nœuds join sont créés. Ces attributs sont ajoutés dans une nouvelle classe dédiée appelée `translation`. Pour chaque transition arrivant sur un nœud join, un nouvel attribut de type booléen, initialisé à `faux` est créé. On ajoute, dans l'effet de chacune de ces transitions, l'action OCL modifiant la valeur de l'attribut correspondant (passage à `vrai`). Ensuite, pour chaque transition qui part d'un nœud join, sont ajoutées :

- Au niveau de la garde : une expression OCL vérifiant que les attributs (précédemment créés) correspondant à chacune des transitions arrivant sur le nœud join ont pour valeur `vrai`.
- Au niveau de l'effet : une expression OCL modifiant la valeur de tous ces attributs (vérifiés dans la garde) à `faux`.

Une fois cette étape réalisée, la transformation des nœuds fork et join en état parallèle peut être effectuée. Pour chaque couple de nœuds fork/join, on crée un état parallèle. Pour chaque transition qui quitte le nœud fork, une nouvelle région est créée dans l'état parallèle, un état initial est ajouté, et la transition en question part dorénavant de ce

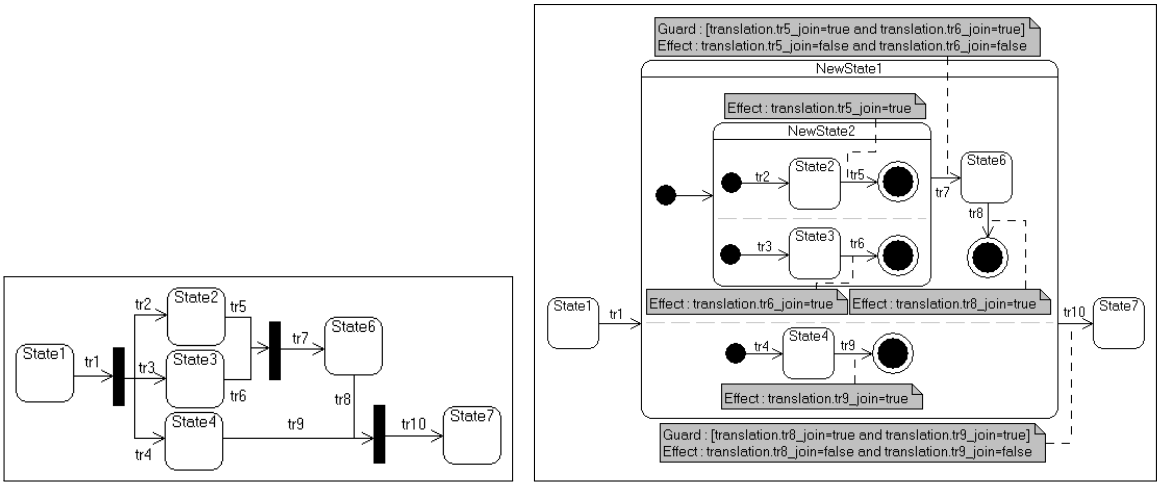
nouvel état. On extrait ensuite toutes les transitions et tous les états qui découlent de cette transition jusqu'à atteindre un nœud join que l'on remplace par un état final au sein de la région. Enfin, toutes les transitions qui arrivent sur le nœud fork ou qui partent du nœud join sont repositionnées sur le nouvel état parallèle. Par exemple, les nœuds fork et join de la figure 5.6a sont réécrits en état parallèle comme décrit sur la figure 5.6b.



(a) Nœuds fork/join (b) Etat parallèle

FIGURE 5.6 – Réécriture de nœuds fork/join en états parallèles

Dans les cas où plusieurs nœuds join sont associés à un même nœud fork, le même procédé est appliqué en utilisant des imbrications d'états parallèles afin de reproduire les comportements modélisés. Par exemple, les nœuds fork/join de la figure 5.7a sont réécrits en états parallèles comme décrits sur la figure 5.7b.



(a) Nœuds fork/join (b) Etats parallèles

FIGURE 5.7 – Réécriture de nœuds fork/join en états parallèles

Une fois la transformation des nœuds fork et join réalisée, il est possible de réécrire les états composites, historiques et parallèles par ordre hiérarchique.

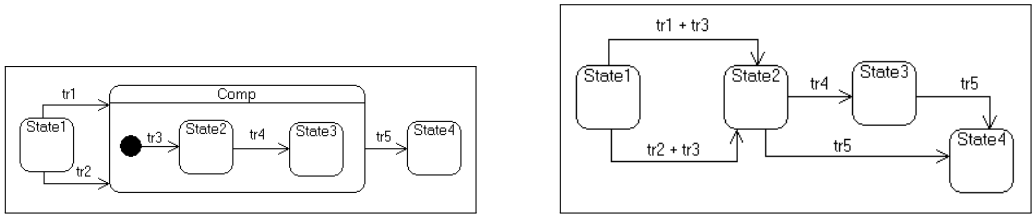
Etat composite

Le processus mis en œuvre pour la suppression d'un état composite consiste à rediriger et éventuellement dupliquer les transitions qui arrivent et qui partent de l'état composite en question. Il faut également considérer les actions onEntry et onExit et les transitions internes de l'état composite.

Afin de supprimer un état composite, les étapes suivantes sont appliquées :

1. Toutes les transitions qui aboutissent à l'état composite sont redirigées afin de pointer sur l'état ciblé par l'unique transition qui va de l'état initial à l'état composite.
2. L'action de la transition qui quitte l'état initial de l'état composite et l'action onEntry de l'état composite sont transposées dans l'effet de chacune des transitions précédemment déplacées.
3. L'état initial de l'état composite est ensuite supprimé.
4. Toutes les transitions qui partent de l'état composite sont dupliquées sur chaque état de l'état composite et l'action onExit de l'état composite est ajoutée dans l'effet de chacune de ces transitions.
5. Toutes les transitions internes de l'état composite sont dupliquées sur chaque état de celui-ci.
6. On sort les états de l'état composite et ce dernier est supprimé.

La figure 5.8b représente le diagramme correspondant à la réécriture de l'état composite représenté sur la figure 5.8a.



(a) Avec état composite (b) Sans état composite

FIGURE 5.8 – Réécriture d'un état composite

Etat historique

Un état historique se positionne au sein d'un état composite et représente le dernier état actif contenu dans l'état composite. Le processus mis en place pour la transformation de ce type d'état consiste à reproduire l'état mémoire à l'aide d'un attribut de classe afin de pouvoir se repositionner directement dans le dernier état actif lorsqu'une transition concernée est franchie.

Il y a deux types d'états historiques : standard et profond. Afin de conserver leur sémantique respective, pour un état historique profond contenu dans un état composite `C`, tous les sous-états composites et parallèles contenus dans `C` sont réécrits en premier. Dans le cas de la réécriture d'un état historique standard, c'est l'état historique qui est réécrit en premier lieu. Afin de réécrire un état historique, les étapes suivantes sont appliquées.

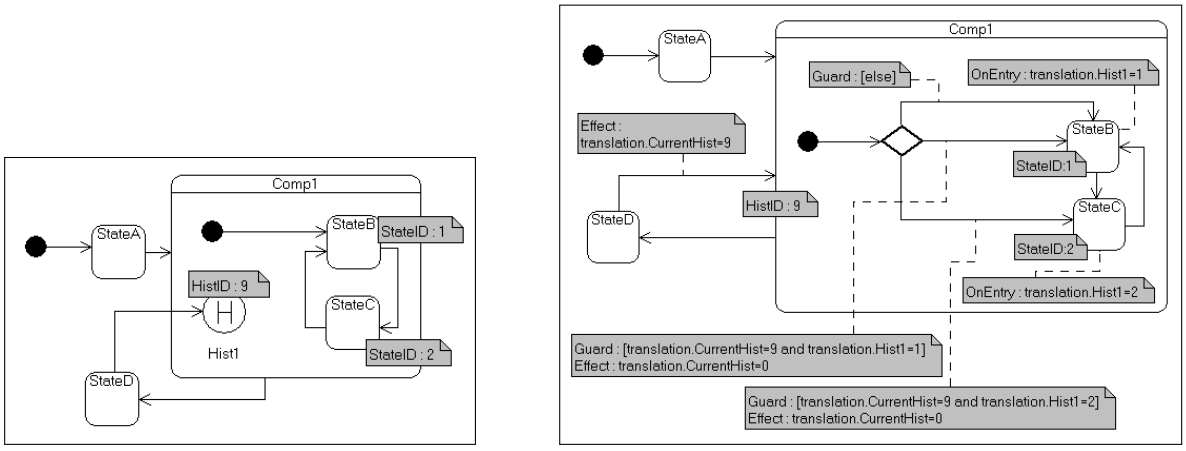
Tout d'abord, afin de représenter l'état mémoire, pour chaque état historique, un attribut est créé dans une classe dédiée nommée `translation`. Dans un souci de lisibilité du modèle obtenu, la classe utilisée est la même que celle créée pour la réécriture des nœuds fork/join. Cet attribut permet d'identifier le dernier état actif pour chaque état composite qui contient un état historique. Cet attribut est de type `entier` et a pour valeur initiale `0`. Un attribut nommé `CurrentHist` est également créé dans la classe `translation` permettant d'identifier quel état historique a été activé. Il est de type `entier` et initialisé à `0`. Afin de pouvoir utiliser les attributs précédemment créés, chaque état historique du modèle est identifié avec un entier positif unique (cet entier est appelé `HistID` dans la suite de cette section). De plus, chaque état contenu dans un état composite qui contient un état historique est numéroté (numérotation indépendante de la première). On appelle cet entier `StateID` dans la suite de ce document. Par exemple, la figure 5.9a représente une possibilité de numérotation d'un diagramme d'états-transitions contenant un état historique.

Afin de pouvoir stocker le dernier état actif, pour chaque état de l'état composite, l'action onEntry mettant à jour l'attribut associé à l'état historique en cours de traitement avec le `StateID` correspondant est ajoutée. Chaque transition qui pointe vers un état historique est alors modifiée afin de pointer sur l'état composite contenant l'état historique (permettant ainsi la suppression de celui-ci). On ajoute à l'effet de chacune de ces transitions l'expression OCL permettant de représenter l'activation de l'état historique, c'est-à-dire, la modification de la valeur de l'attribut `CurrentHist` avec l'`HistID` de l'état historique.

Finalement, à l'activation d'un état historique, pour réaliser le positionnement dans l'état mémorisé, un point de choix est mis en place à l'entrée de l'état composite (au niveau de la transition quittant l'état initial). Cette transition prend alors la valeur de transition par défaut (garde `else`). A partir de ce point de choix, des transitions sont créées afin de positionner le système directement dans chacun des états de l'état composite. Chacune de ces transitions :

- ne porte pas de déclencheur,
- a pour garde l'expression vérifiant que l'état historique a bien été activé (attribut `CurrentHist` associé au bon `HistId`) et que l'état mémoire stocké est bien celui pointé par la transition (vérification de la valeur de l'attribut représentant l'état historique),
- a pour effet : `CurrentHist=0` (désactivation de l'état historique).

Par exemple, dans le cas du diagramme représenté par la figure 5.9a, on obtient le diagramme représenté sur la figure 5.9b.



(a) Avec état historique (b) Sans état historique

FIGURE 5.9 – Réécriture d'un état historique

Etat parallèle

La stratégie utilisée pour réécrire un état parallèle est la même que pour fusionner les diagrammes d'états-transitions concurrents. Chaque région de l'état parallèle est considérée comme un diagramme d'états-transitions, la réécriture est effectuée et le résultat est inséré dans un état composite qui remplace l'état parallèle. La fusion des diagrammes d'états-transitions concurrents est expliquée dans la partie suivante.

5.1.4 Diagrammes d'états-transitions multiples

SysML4MBT autorise la spécification de plusieurs diagrammes d'états-transitions qui évoluent en concurrence. Ils sont fusionnés lors de la réécriture afin d'obtenir un unique diagramme d'états-transitions ne contenant aucun parallélisme dans le but d'obtenir un modèle cohérent vis-à-vis des règles de modélisation du langage UML4MBT. Les étapes nécessaires sont décrites en détail et l'algorithme est précisé à la fin de cette partie sous forme de pseudo-code.

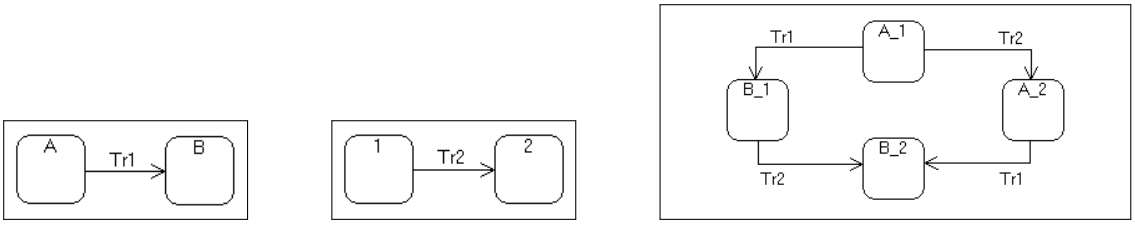
Il faut tout d'abord que tous les nœuds fork et join ainsi que les états composites, parallèles et historiques de tous les diagrammes d'états-transitions soient supprimés. La fusion est réalisée en une fois, c'est-à-dire que tous les diagrammes d'états-transitions sont fusionnés en même temps. Le processus utilisé est basé sur le produit cartésien. Cependant, afin de limiter le nombre de transitions infranchissables tracées dans le modèle résultat, des règles spécifiques à appliquer au cours du produit sont définies.

Les principales étapes sont les suivantes : les états initiaux de tous les diagrammes d'états-transitions sont fusionnés. Ensuite, le diagramme d'états-transitions résultat est construit transition par transition en parcourant les différents diagrammes. Des vérifications sont effectuées avant la création de chaque transition.

Construction générale

Le procédé mis en place consiste à fusionner les états à la façon d'un produit cartésien. Lors de la fusion de `n` diagrammes d'états-transitions, chaque état issu du produit est un n-uplet dont chaque composante correspond à un état d'un diagramme différent. Comme pour le produit cartésien, une transition relie deux états dont la seule différence se situe au niveau de la composante qui correspond au diagramme d'où est issue la transition. Par défaut n'importe quelle transition peut être tracée du moment que son état de départ est une composante de l'état issu du produit.

Par exemple, la figure 5.10c représente le résultat de la fusion des diagrammes représentés sur les figures 5.10a et 5.10b. L'état `A_1` correspond à la fusion de l'état `A` et de l'état `1`. La transition `Tr1` quittant cet état est reliée à l'état `B_1` car elle reliait initialement l'état `A` à l'état `B`.



(a) Diagramme 1 (b) Diagramme 2 (c) Fusion des Diagrammes 1 et 2

FIGURE 5.10 – Base de la fusion des diagrammes d'états-transitions parallèles

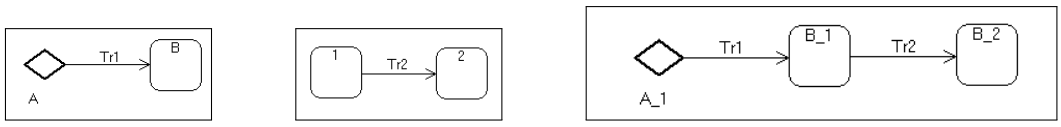
Les règles suivantes sont utilisées au niveau de la fusion des états :
- La fusion d'états initiaux donne un état initial.
- La fusion d'états finaux donne un état final.
- La fusion d'un point de choix avec d'autres états standards donne un point de choix.
- Toutes les autres combinaisons résultent en un état standard.

La construction se faisant pas à pas, un état ne peut pas être créé si aucune transition ne l'atteint. Afin de créer un diagramme d'états-transitions présentant les mêmes comportements que les diagrammes d'états-transitions concurrents et afin de limiter la création de transitions infranchissables, les deux règles suivantes sont appliquées.

Règle pour les points de choix

D'après les règles énoncées précédemment, le modèle résultat peut contenir des points de choix. Lorsque c'est le cas, une règle dédiée s'applique alors au niveau des transitions pouvant partir de cet état : seules les transitions qui partent du point de choix initial peuvent partir du point de choix obtenu à la suite de la fusion d'états.

Par exemple, concernant la fusion des diagrammes représentés sur les figure 5.11a et 5.11b, la fusion des états `A` et `1` permet l'obtention de l'état `A_1`. Ce nouvel état est un point de choix car l'état `A` est lui-même un point de choix. La règle énoncée dans cette section indique que seules les transitions qui partent de l'état `A` peuvent partir de l'état `A_1`. De ce fait, depuis l'état `A_1`, seule la transition `Tr1` peut être tracée emmenant le système dans l'état `B_1`. De ce nouvel état peut alors partir la transition `Tr2`. On obtient alors le diagramme de la figure 5.11c.



(a) Diagramme 1 (b) Diagramme 2 (c) Fusion des Diagrammes 1 et 2

FIGURE 5.11 – Fusion de diagrammes d'états-transitions : cas des points de choix

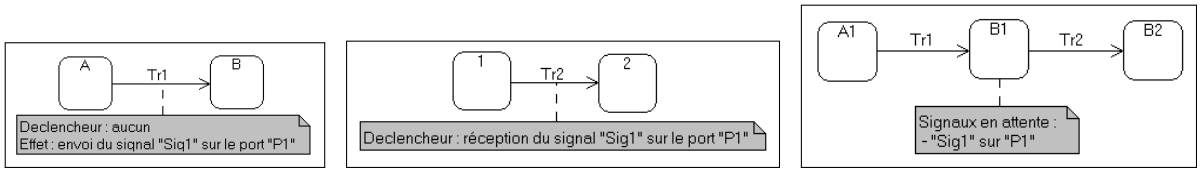
Règle sur les envois/réceptions de signaux

Les transitions déclenchées par une réception d'appel d'opération ou qui ne portent aucun déclencheur sont tracées sans autres restrictions que celles énoncées précédemment. Dans le cas des transitions déclenchées par une réception de signal, une règle spécifique est appliquée. En effet, afin de limiter la construction de transitions inaccessibles, une transition déclenchée par une réception de signal ne peut être tracée que si le signal a pu être émis précédemment. Le procédé appliqué afin de pouvoir respecter cette règle est le suivant : à chaque construction de transition, les signaux émis par les comportements attachés à celle-ci sont *stockés* au niveau de l'état à l'aide d'une structure dédiée. Une transition déclenchée par une réception de signal est alors tracée uniquement lorsque le signal en question est bien *stocké* au niveau de l'état de départ.

Concrètement, le stockage des signaux en attente est réalisé de la manière suivante à l'aide d'une structure intitulée *groupe*. Un groupe représente un ensemble de signaux en attente en même temps. Plusieurs groupes de signaux peuvent être stockés au niveau d'un état. En effet, si une transition contient deux comportements, chacun d'entre eux effectuant des envois de signaux, deux groupes sont alors créés au niveau de l'état, chacun contenant l'ensemble des signaux émis par un des comportements. Lorsqu'une transition déclenchée par une réception de signal est tracée, seuls les groupes contenant le signal en question sont transférés, amputés du dit signal, vers l'état cible. La construction d'une transition ne portant pas de déclencheur implique le transfert automatique de tous les groupes (éventuellement augmentés par les émissions possibles de la transition) de l'état source vers l'état cible. Ces groupes peuvent dans ce cas être modifiés si cette transition effectue un ou plusieurs envois de signaux.

Ce procédé ne permet pas de s'affranchir totalement de la construction de transitions infranchissables car les gardes OCL ne sont pas prises en compte dans ce processus. Il faudrait être en mesure d'animer le modèle au fil de la réécriture afin de s'assurer qu'il existera toujours au moins une possibilité d'animation permettant de franchir chacune des transitions tracées. Une telle solution engendre les problématiques d'animation du modèle et de parcours de tous les chemins, problématiques non abordées au cours de nos travaux.

Par exemple, dans le cas de la fusion des diagrammes représentés sur les figures 5.12a et 5.12b, à partir de l'état `A_1`, seule la transition `Tr1` peut être tracée car aucun signal n'a été émis au préalable. Cependant, à partir de l'état `B_1`, la transition `Tr2`, déclenchée par une réception de signal, peut être tracée car le signal en question a été émis par la transition `Tr1`. Au cours de la fusion, les signaux en attente sont *stockés* au niveau des états comme représentés figure 5.12c.



(a) Diagramme 1 (b) Diagramme 2 (c) Fusion des Diagrammes 1 et 2

FIGURE 5.12 – Fusion de diagrammes d'états-transitions : cas des transmissions de signaux

Algorithme

Un algorithme prenant en compte l'ensemble des règles énoncées a été défini. Afin de pouvoir formaliser cet algorithme, il est nécessaire de surcharger la formalisation des états

donnée section 3.4 de manière à prendre en compte les *groupes* présentés précédemment. Pour chaque état, le paramètre `STgr` est ajouté. Il s'agit d'un ensemble d'ensembles de couples port/signal. Chaque ensemble contenu dans `STgr` représente l'ensemble des signaux qui peuvent être attendus dans un cas d'animation. On définit n diagrammes $\texttt{SM}_1$, $\texttt{SM}_2$ ... $\texttt{SM}_\texttt{n}$ à fusionner. Le diagramme d'états-transitions résultat est représenté par $\texttt{SM}_{\texttt{res}}$. L'algorithme est représenté sur la figure 5.14 sous forme de pseudo-code et utilisant la formalisation donnée section 3.4.

La fonction `transfererGroupes` consiste à appliquer les transferts de groupes tels qu'ils sont énoncés dans les propriétés ci-dessus. L'algorithme correspondant à cette fonction est représenté sur la figure 5.13.

5.1.5 Création du diagramme d'objets

Un modèle UML4MBT valide doit contenir un diagramme d'objets définissant l'état initial du système. Etant donné qu'il n'existe pas de diagramme d'objets en SysML, il doit être construit de toutes pièces. Le processus est le suivant :

- Les classes représentant les signaux sont instanciées par autant d'instances que le plus grand nombre de fois où le signal est attendu au niveau d'un même état. Cette information est obtenue à partir des groupes utilisés dans la fusion des diagrammes d'états-transitions.
- Les autres classes du diagramme de classes UML4MBT obtenues sont instanciées une unique fois dans le diagramme d'objets.
- Les associations sont instanciées de manière à respecter les plus petites multiplicités définies dans le diagramme de classes.

5.1.6 Exigences

En SysML4MBT, le diagramme d'exigences permet l'association des exigences avec les éléments du modèle qui sont mis en jeu pour les satisfaire. En UML4MBT, ce type d'information est représenté à l'aide d'expressions OCL spécifiques. En effet, l'utilisation de commentaires OCL dédiés permet d'exprimer le fait qu'une partie spécifique du code OCL satisfait l'exigence exprimée dans le commentaire. La syntaxe est la suivante :

```
/**@REQ: "texte de l'exigence"*/
```

La syntaxe UML4MBT restreint donc la diversité des éléments auxquels peuvent être associées des exigences. En effet, seuls les éléments pouvant accueillir des expressions OCL sont concernés. Ainsi, seules les exigences comportementales, définies dans le diagramme

d'exigences SysML4MBT, satisfaites par un élément de modèle pouvant contenir du code OCL en UML4MBT sont transcrites. En l'occurrence, les exigences SysML4MBT associées (par un lien `satisfy`) à :

- une transition sont retranscrites en OCL dans l'effet de cette transition,
- une opération sont retranscrites en OCL dans la post-condition de l'opération,
- une action `onEntry` ou à un état du diagramme d'états-transitions sont retranscrites par des annotations OCL dans l'action `onEntry` de l'état,
- une action `onExit` sont retranscrites dans l'action `onExit` en annotations OCL.

On utilise le texte associé à l'exigence dans le modèle SysML4MBT comme texte de l'exigence UML4MBT. Les exigences associées à d'autres types d'éléments (bloc, association, connecteur...) ne peuvent donc pas être retranscrites en UML4MBT.

```
                    transfererGroupes(tr)
etatGroupes = ∅                                                          ⎫
SWITCH (tr.TRtrig)                                                       ⎪
   CASE (NULL):                                                          ⎪
      etatGroupes = tr.TRstart.STgr ∪ {{}}                               ⎪
   CASE (∈ allOps()):                                                    ⎪
      etatGroupes={{}}                                                   ⎪ Récupération des groupes
   DEFAULT:                                                              ⎪ de l'état de départ.
      ∀ gr ∈ tr.TRstart.STgr DO                                          ⎪
         IF (tr.TRtrig ∈ gr) THEN                                        ⎪
            etatGroupes = etatGroupes ∪ {gr - {tr.TRtrig}}               ⎪
         ENDIF                                                           ⎪
      DONE                                                               ⎪
ENDSWITCH                                                                ⎭

transGroupes = ∅                                                         ⎫
∀ bhv ∈ tr.TRbhvs                                                        ⎪
   ssGroupes = ∅                                                         ⎪
   ∀ elt ∈ bhv.BHVaction                                                 ⎪
      IF (NOT (elt ∈ allProps())) THEN                                   ⎪ Récupération des groupes
         ssGroupes = ssGroupes ∪ {elt}                                   ⎪ émis par la transition.
      ENDIF                                                              ⎪
   DONE                                                                  ⎪
   transGroupes = transGroupes ∪ {ssGroupes}                             ⎪
DONE                                                                     ⎭

∀ etatGr ∈ etatGroupes                                                   ⎫
   ∀ transGr ∈ transGroupes                                              ⎪
      tr.TRend.STgr = tr.TRend.STgr ∪ {etatGr ∪ transGr}                 ⎪ Fusion des groupes.
   DONE                                                                  ⎪
DONE                                                                     ⎭
```

FIGURE 5.13 – Algorithme de la fonction `transfererGroupes` de la transformation SysML4MBT vers UML4MBT

```
{ETAT} Todo=∅                } Ensemble d'états à parcourir
ETAT EnCours                 } Etat en train d'être parcouru.
TRANSITION TrEnCours         } Transition en cours de création.
ETAT EnCoursCible            } Etat cible de la transition en cours de création.
SM_res.STATES = ∅
SM_res.TRANS = ∅
init_1 ∈ SM_1.ST_init,...,init_n ∈ SM_n.ST_init
EnCours=⟨init_1_ ..._init_n,NULL,∅⟩
SM_res.ST_init = SM_res.ST_init ∪ {EnCours}
tr_1 = SM_1.TRANS.⟨init_1,bis_1,NULL,true,bhvs_1⟩
...
tr_n = SM_n.TRANS.⟨init_n,bis_n,NULL,true,bhvs_n⟩
EnCoursCible=⟨bis_1_..._bis_n,NULL,∅⟩
SM_res.ST_stand = SM_res.ST_stand ∪ {EnCoursCible}
TrEnCours = ⟨EnCours,EnCoursCible,NULL,true,bhvs_1 ∪ ...∪ bhvs_n⟩
SM_res.TRANS = SM_res.TRANS ∪ {TrEnCours}
transfererGroupes(TrEnCours)
Todo=Todo ∪ {EnCoursCible}

∀ EnCours=⟨E_1_..._E_n,NULL,gr⟩ ∈ Todo DO
   ∀ i FROM 1 TO n DO
      IF ((EnCours ∉ SM_res.ST_choice) or (E_i ∈ SM_i.ST_choice)) THEN
         ∀ TrEnCours ∈ SM_i.TRANS DO

            EnCoursCible = ⟨E_1_..._E_i-1_TrEnCours.TRend_E_n,_,_⟩
            IF (EnCoursCible ∉ SM_res.STATES) THEN
               SM_res.STATES = SM_res.STATE ∪ EnCoursCible
            ENDIF
            TRANSITION TrNouv = TrEnCours
            TrNouv.TRstart = = EnCours
            TrNouv.TRend = = EnCoursCible

            IF (TrNouv ∈ SM_res.TRANS) THEN
               transfererGroupes(TrNouv)

            ELSE
               IF (TrNouv.TrTrig ∈ allOPS
                   or TrNouv.TrTrig = NULL
                   or TrNouv.TrTrig ∈ EnCours.STgr) THEN
                  SM_res.TRANS = SM_res.TRANS ∪ {TrNouv}
                  transfererGroupes(TrNouv)
               ENDIF
            ENDIF
         DONE
      ENDIF
   DONE
DONE
```

Déclarations.

Création des deux premiers états et de la première transition.

Parcours de chaque transition partant de chaque partie de chaque état en attente.

Construction ou récupération de l'état cible.

Si la transition existe déjà, transfert uniquement,

sinon, vérification avant création.

FIGURE 5.14 – Algorithme principal de la transformation SysML4MBT vers UML4MBT

5.1.7 Synthèse

L'ensemble des règles de réécriture détaillées dans cette section permettent la transformation d'un modèle SysML4MBT en modèle UML4MBT sans perte de comportement. Un algorithme dédié a été défini afin que les différentes étapes de réécriture puissent se succéder sans effet de bord. Les règles décrites dans cette section doivent être exécutées dans l'ordre spécifique suivant :

1. Le diagramme d'exigences est réécrit.
2. Le diagramme de bloc et le diagramme interne de bloc sont transformés en diagramme de classes.
3. Les envois et les réceptions de signaux sont transformés.
4. Les nœuds fork et join et les états composites, historiques et parallèles sont réécrits.
5. Les diagrammes d'états-transitions sont fusionnés.
6. Le diagramme d'objets UML4MBT est construit.

La section suivante présente l'application de cet algorithme sur l'exemple fil rouge.

5.2 Exemple fil rouge

Cette section illustre les règles de réécriture présentées dans la section précédente par leur application sur l'exemple fil rouge. Cette présentation se base sur la modélisation SysML4MBT de l'exemple fil rouge présentée en section 3.5.

5.2.1 Diagramme d'exigences

La réécriture du diagramme d'exigences se fait par l'insertion d'annotations OCL au sein des diagrammes d'états-transitions. Dans une optique de lisibilité, nous représentons dans cette partie uniquement les modifications concernant le diagramme d'états-transitions du panneau de contrôle. Compte tenu du diagramme d'exigences de l'exemple fil rouge présenté dans ce document (Fig. 3.28 page 73), le diagramme d'états-transitions initial du panneau de contrôle (Fig. 3.26 page 71) devient le diagramme d'états-transitions de le figure 5.15. Les modifications par rapport au diagramme d'états-transitions initial se situent au niveau des expressions OCL écrites en gras qui sont insérées dans l'effet des transitions qui sont initialement reliées à des exigences du diagramme d'exigences.

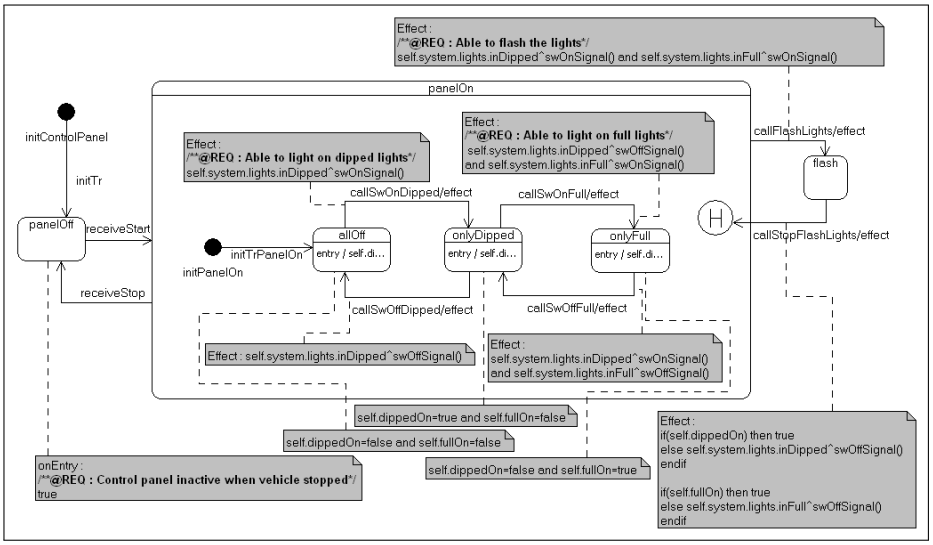


FIGURE 5.15 – Diagramme d'états-transitions du panneau de contrôle après réécriture du diagramme d'exigences

5.2.2 Partie statique

La partie suivante consiste en la réécriture du diagramme de bloc et du diagramme interne de bloc, c'est-à-dire, la transformation des blocs et des signaux en classes et des ports en associations. Ainsi, les diagramme de bloc SysML4MBT de l'exemple fil rouge (voir Fig. 3.25 page 70), y compris les signaux et les ports représentés dans le diagramme de bloc (cf. Fig. 3.21 page 67) et dans le diagramme interne de bloc (cf. Fig. 3.22 page 67) sont retranscrits afin d'obtenir le diagramme de classes représenté sur la figure 5.16.

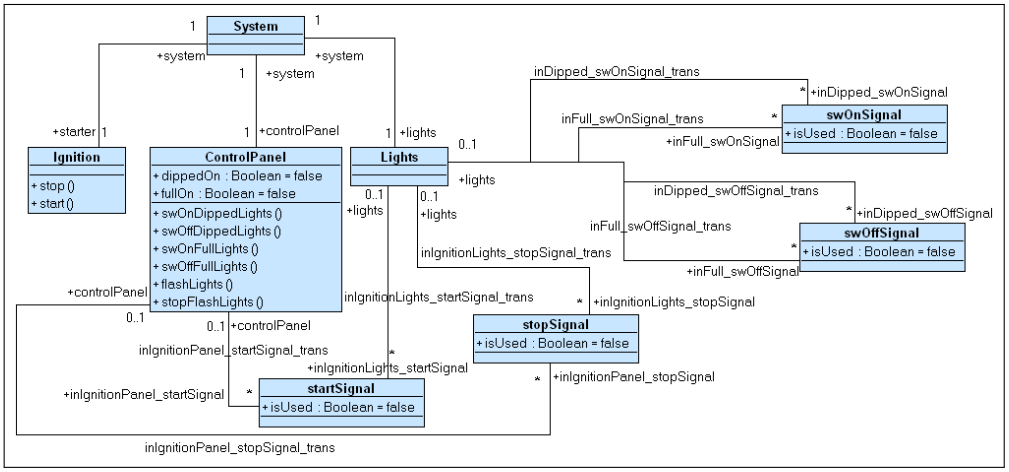


FIGURE 5.16 – Diagramme de classes UML4MBT résultant de la réécriture des signaux de l'exemple fil rouge

5.2.3 Partie dynamique

Au niveau de la représentation dynamique du système, la première étape consiste à retranscrire les envois et réceptions de signaux.

Envois et réceptions de signaux

Comme chacun des diagrammes d'états-transitions de l'exemple fil rouge effectue des envois et des réceptions de signaux, ils sont tous les trois modifiés au cours de cette étape. Afin de simplifier la présentation, seules les modifications du diagramme d'états-transitions du panneau de contrôle sont illustrées dans cette partie par la figure 5.17. Comme expliqué précédemment, les envois et les réceptions de signaux sont transformés en expressions OCL ayant pour actions respectives de créer et de vérifier puis de supprimer des instances d'associations qui représentent les signaux en attente. Les zones dont le texte est écrit en gras noir sur fond gris foncé correspondent aux zones où les envois de signaux ont été réécrits. Les zones dont le texte est en gras noir sur fond gris clair représentent les expressions OCL ajoutées par la phase de réécriture des réceptions de signaux.

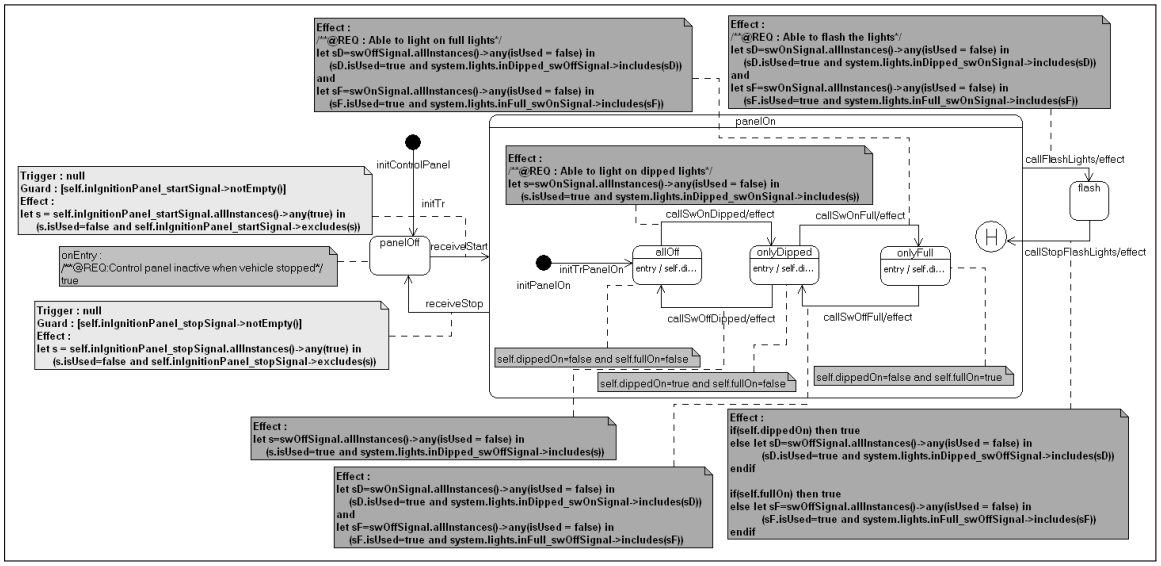


FIGURE 5.17 – Diagramme d'états-transitions du panneau de contrôle après réécriture des envois/réceptions de signaux

Nœuds fork/join, états historiques et parallèles

L'étape suivante consiste en la réécriture des nœuds fork et join et des états parallèles, composites et historiques. En l'occurrence, les diagrammes d'états-transitions du modèle (figures 3.23, 3.26 et 3.27) contiennent au total uniquement un état historique (diagramme d'états-transitions du panneau de contrôle) et deux états composites (diagrammes d'états-transitions du panneau de contrôle et des lumières). Dans cette partie, seule la réécriture du diagramme d'états-transitions du panneau de contrôle est détaillée, les étapes de transformation du diagramme d'états-transitions des lumières étant identiques. Comme expliqué précédemment, l'état historique doit être réécrit en premier lieu, afin de pouvoir réécrire l'état composite qui le contient. Par souci de lisibilité, les expressions OCL précisées sur les figures de cette section correspondent aux expressions OCL qui doivent être ajoutées à la suite de celles précisées dans les sections précédentes.

Pour la réécriture de l'état historique, la première étape consiste en la mise en place de la classe `translation` permettant la simulation de l'état mémoire. L'état historique doit être représenté par un attribut de cette classe. Cet attribut est appelé `HistPanel`. On obtient ainsi la classe `translation` représentée sur la figure 5.18.

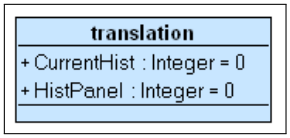


FIGURE 5.18 – Classe `translation` mise en place pour la réécriture de l'état historique

Ensuite, un numéro est attribué à chaque état concerné par l'état historique. On attribue également un numéro à l'état historique. On obtient le diagramme de la figure 5.19.

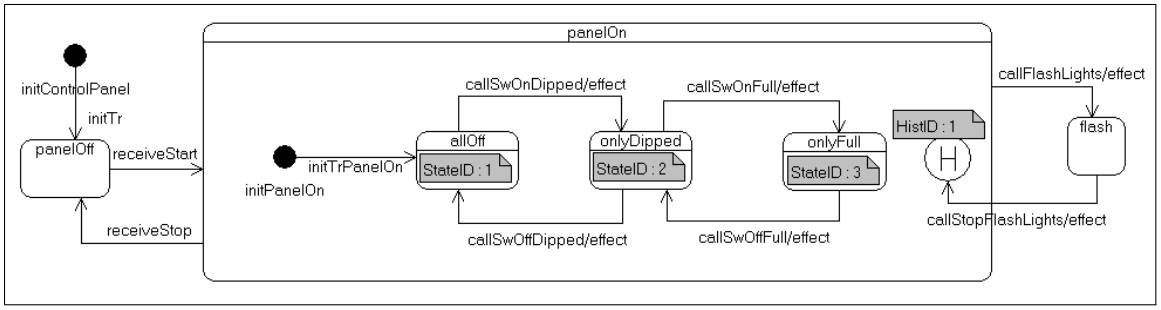


FIGURE 5.19 – Diagramme d'états-transitions du panneau de contrôle numéroté pour la réécriture de l'état historique

L'état historique est ensuite réécrit permettant l'obtention du diagramme d'états-transitions représenté sur la figure 5.20. L'état composite peut à son tour être supprimé comme illustré sur la figure 5.21

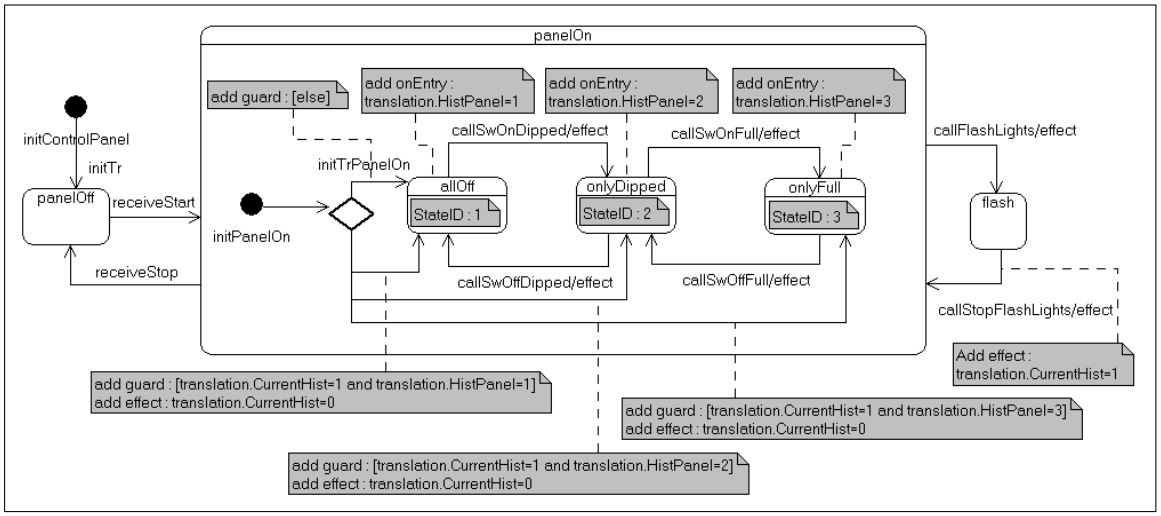


FIGURE 5.20 – Diagramme d'états-transitions du panneau de contrôle après réécriture de l'état historique

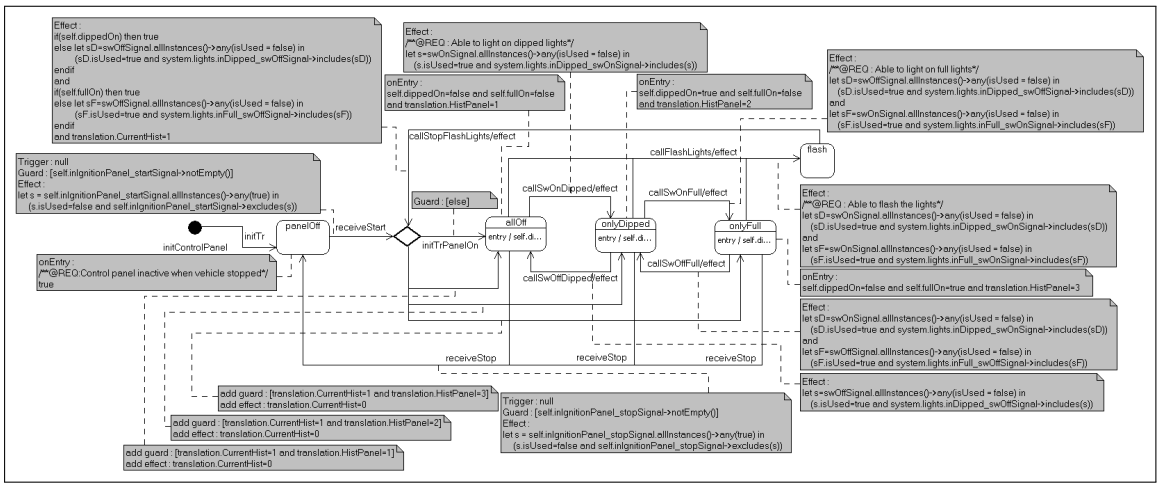


FIGURE 5.21 – Diagramme d'états-transitions du panneau de contrôle après réécriture de l'état composite

Fusion des diagrammes d'états-transitions

Au niveau de la représentation dynamique, la dernière étape consiste en la fusion des diagrammes d'états-transitions des figures 5.21 (issu de la réécriture du diagramme d'états-transitions du panneau de contrôle détaillé jusqu'ici), 5.22 (résultat de la réécriture du diagramme d'états-transitions du démarreur présenté figure 3.23 page 68) et 5.23 (résultat de la réécriture du diagramme d'états-transitions des lumières présenté figure 3.27 page 72).

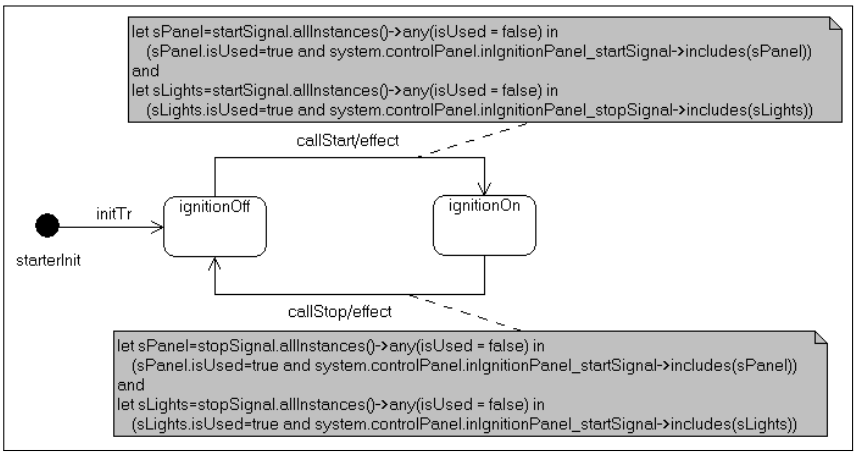


FIGURE 5.22 – Diagramme d'états-transitions du démarreur après réécriture

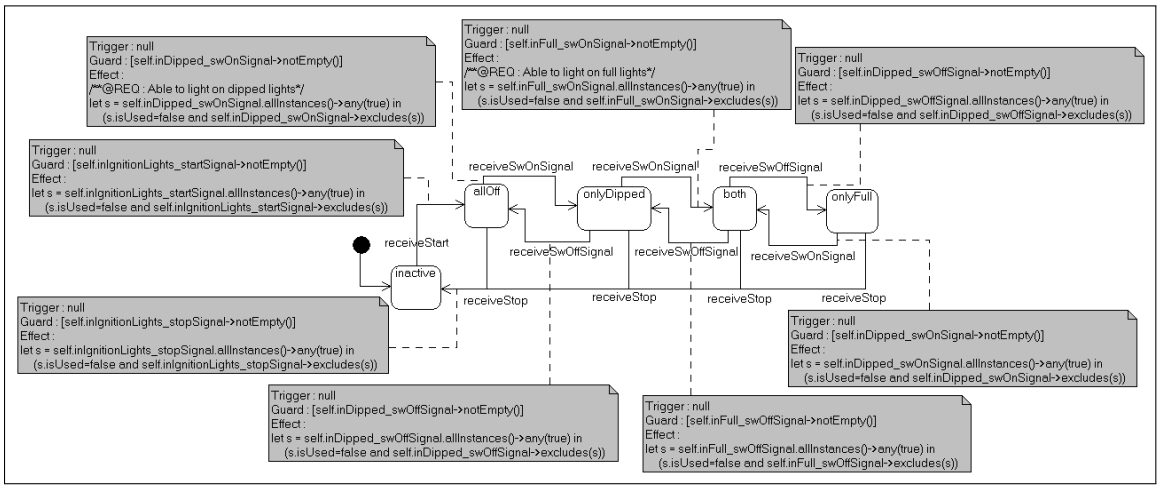


FIGURE 5.23 – Diagramme d'états-transitions des lumières après réécriture

Le résultat de cette fusion est présenté sur la figure 5.24. Bien que cette représentation graphique ne soit pas exploitable, elle permet de se rendre compte de la taille du diagramme d'états-transitions obtenu.

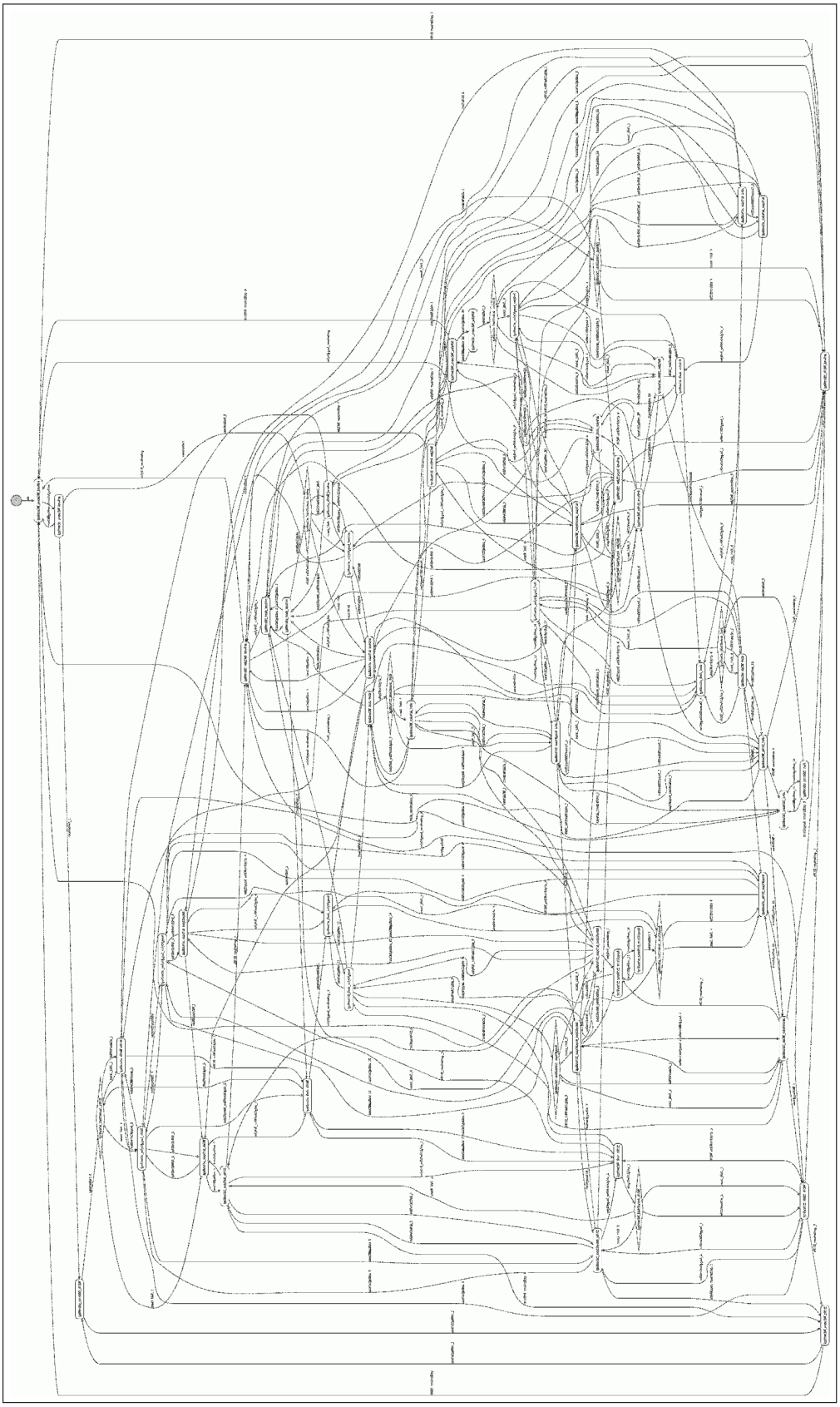


FIGURE 5.24 – Diagramme d'états-transitions résultant de la fusion des diagrammes des figures 5.21, 5.22 et 5.23

5.2.4 Diagramme d'objets

Le diagramme d'objets représenté en figure 5.25 est alors créé de toutes pièces afin d'obtenir un modèle UML4MBT valide. Les classes et les associations sont instanciées en respectant les règles présentées précédemment. D'après les informations recueillies durant la fusion des diagrammes d'états-transitions, chaque signal est attendu au plus deux fois en même temps. Ainsi, chaque classe représentant un signal est instanciée deux fois.

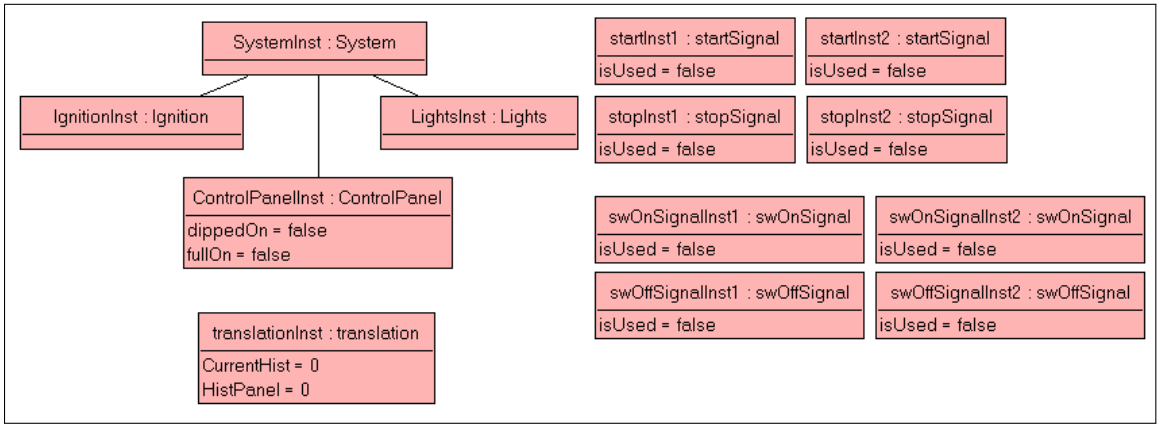


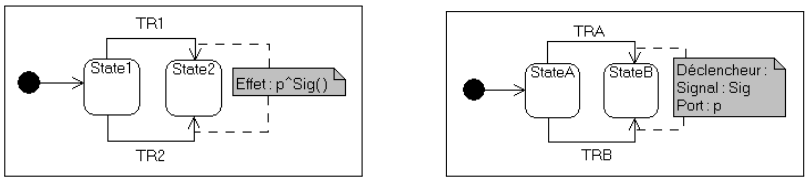
FIGURE 5.25 – Diagramme d'objets du modèle UML4MBT de l'exemple fil rouge

5.3 Application de ComCover

La phase de génération de tests s'appuie sur les algorithmes de base mis en œuvre dans Test Designer^{TM} qui assurent une couverture du critère *Toutes les transitions* et une couverture du critère D/CC. L'application de cette stratégie sur les modèles UML4MBT issus de la transformation ne permettent pas d'assurer la couverture du critère *Toutes les* DU_{sig} comme le montre cette section. Une surcharge des règles de réécriture a donc été définie afin d'assurer la couverture de ce critère à l'application de la stratégie mise en œuvre dans l'outil Test Designer^{TM}.

5.3.1 Justification de la surcharge

Pour permettre de mettre en œuvre la stratégie ComCover, nous devons apporter des modification à la traduction. Nous motivons et illustrons cette modification sur un exemple composé des deux diagrammes d'états-transitions SysML4MBT concurrents représentés sur les figures 5.26a et 5.26b.



(a) Diagramme 1 (b) Diagramme 2

FIGURE 5.26 – Exemple de diagrammes d'états-transitions SysML4MBT

La transformation de modèles SysML4MBT vers UML4MBT présentée dans ce mémoire nous permet d'obtenir le diagramme représenté sur la figure 5.27.

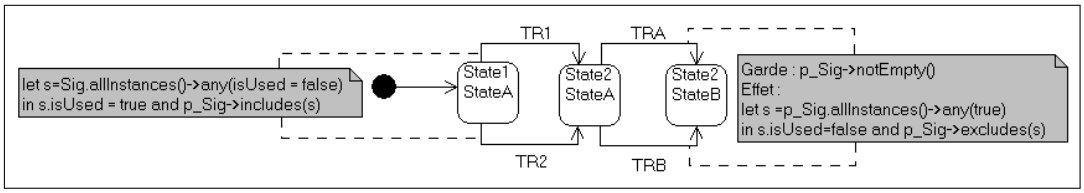


FIGURE 5.27 – Diagramme d'états-transitions UML4MBT résultant de la réécriture des diagrammes de la figure 5.26

La couverture de ce diagramme par les stratégies définies pour les modèles UML4MBT permet d'obtenir quatre cibles de test comme présentés dans le tableau 5.28a. En comparaison, l'application de la stratégie ComCover sur le modèle SysML initial permet d'obtenir les quatre cibles de test couvertes par les quatre tests représentés dans le tableau 5.28b.

Cibles	Tests
TR1	T1 [⟨TR1,TR1,[TR1;TRA]⟩]
TR2	T3 [⟨TR2,TR2,[TR2;TRA]⟩]
TRA	T1 [⟨TR1,TR1,[TR1;TRA]⟩]
TRB	T2 [⟨TR1,TR1,[TR1;TRB]⟩]

(a) Tests générés sur le modèle UML4MBT (Fig. 5.27)

Cibles	Tests
TR1/TRA	T1 [⟨TR1,TR1,[TR1;TRA]⟩]
TR1/TRB	T2 [⟨TR1,TR1,[TR1;TRB]⟩]
TR2/TRA	T3 [⟨TR2,TR2,[TR2;TRA]⟩]
TR2/TRB	T4 [⟨TR2,TR2,[TR2;TRB]⟩]

(b) Application de ComCover sur le modèle SysML4MBT (Fig. 5.26)

FIGURE 5.28 – Comparaison des tests générés avant et après transformation

On constate alors l'absence de la séquence TR2 puis TRB qui correspond à un test requis lors de l'application de ComCover sur le modèle SysML4MBT. Afin d'assurer une couverture équivalente à celle définie par la stratégie ComCover, la surcharge suivante de la transformation a été définie.

5.3.2 ComCover en appliquant la couverture D/CC

Le processus mis en place consiste à utiliser les propriétés du critère de couverture D/CC afin de mettre en œuvre la stratégie ComCover. De nouveaux comportements sont insérés au sein des transitions déclenchées par une réception de signal afin d'assurer la construction des cibles voulues.

Concrètement, pour chaque classe UML4MBT qui représente un signal du diagramme de bloc, un attribut intitulé `ComCover` de type entier est ajouté. A chaque expression OCL correspondant sémantiquement à un envoi de signal, un nombre unique intitulé `ident` est attribué et l'attribut `ComCover` associé au signal concerné est affecté à cette valeur. Enfin, pour chaque transition déclenchée par une réception de signal, pour chaque identifiant `ident` associé à une émission de signal correspondant à cette réception, un nouveau comportement vérifiant que la valeur de la variable `ComCover` est égale à `ident` est ajouté dans l'effet de la transition de réception.

La figure 5.29 représente le résultat de l'application de ce processus au niveau de l'exemple présenté précédemment. L'attribution des identifiants et l'affectation de la variable se font au niveau des transitions `TR1` et `TR2` et l'insertion des nouveaux comportements est réalisée au niveau des transitions `TRA` et `TRB`.

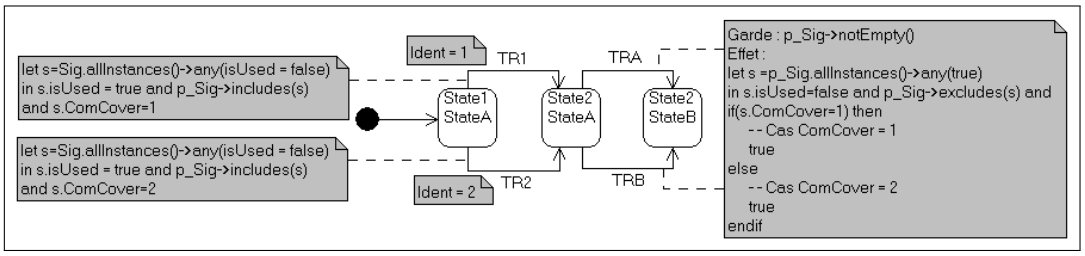


FIGURE 5.29 – Diagramme d'états-transitions UML4MBT résultant de la réécriture surchargée pour ComCover des diagrammes des figures 5.26a et 5.26b

L'application de la stratégie mise en place pour les modèles UML4MBT sur ce nouveau modèle, permet l'obtention des cibles et des tests représentés dans le tableau 5.1.

On remarque alors que tous les tests générés par la stratégie ComCover sur le modèle initial sont présents à l'application de la stratégie utilisée pour les modèles UML4MBT sur le modèle résultant de la transformation surchargée.

Cibles	Tests
TR1	T1 [⟨TR1,TR1,[TR1;TRA]⟩]
TR2	T3 [⟨TR2,TR2,[TR2;TRA]⟩]
TRA avec ComCover=1	T1 [⟨TR1,TR1,[TR1;TRA]⟩]
TRA avec ComCover=2	T3 [⟨TR2,TR2,[TR2;TRA]⟩]
TRB avec ComCover=1	T2 [⟨TR1,TR1,[TR1;TRB]⟩]
TRB avec ComCover=2	T4 [⟨TR2,TR2,[TR2;TRB]⟩]

TABLE 5.1 – Tests générés par la stratégie *Toutes les transitions*+D/CC sur l'exemple présenté sur la figure 5.29

5.4 Déploiement

Afin d'expérimenter les algorithmes mis en place durant nos travaux, un plugin, à été développé pour le modeleur basé sur Eclipse Topcased [TOP10]. Ce modeleur a été choisi car il s'agit du modeleur open-source permettant la représentation de modèles SysML le plus complet. L'interface de ce plugin est représentée sur la figure 5.30.

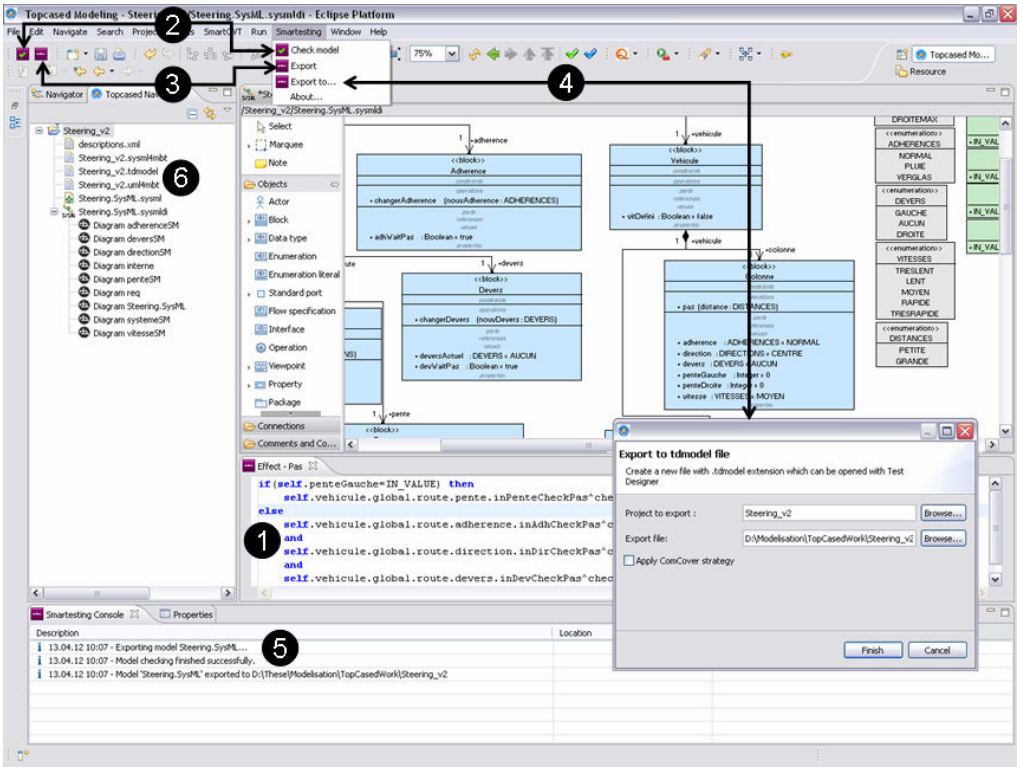


FIGURE 5.30 – Plugin Topcased

Ce plugin permet d'introduire un éditeur OCL proposant une coloration syntaxique et de l'auto-completion (item 1 de la figure 5.30). Il a été développé afin de faciliter la modélisation. Un vérificateur syntaxique peut être actionné (item 2 de la figure 5.30) afin de vérifier que les restrictions SysML4MBT sont bien respectées par le modèle courant et que ce modèle ne contient pas d'erreur de syntaxe. Le résultat s'affiche dans la console (item 5 de la figure 5.30). Si aucune erreur n'est détectée, il est alors possible d'exécuter l'export du modèle (item 3 de la figure 5.30). Cette action a pour effet :

1. D'exporter le modèle SysML4MBT sous la forme d'un fichier *.sysml4mbt*. Il s'agit d'un fichier XML contenant toutes les informations du modèle réalisé dont la structure est basée sur le métamodèle SysML4MBT.
2. D'appliquer les règles de transformation d'un modèle SysML4MBT vers un modèle UML4MBT présentées dans ce chapitre. Le modèle UML4MBT résultant est sauvegardé dans un fichier XML *.uml4mbt* dont la structure est basée sur le métamodèle UML4MBT.
3. De créer un fichier *.tdmodel* à partir du modèle UML4MBT. Il s'agit du fichier d'entrée Test Designer^{TM}, basé sur XML, dont la structure est propriétaire et spécifique à l'outil Test Designer^{TM}.

L'ensemble des fichiers générés apparaît dans l'arborescence (item 6 de la figure 5.30). Lors de l'appel de l'export, si le bouton *Export to...* est utilisé (item 4 de la figure 5.30), une nouvelle fenêtre s'ouvre permettant l'application (ou non) de la stratégie ComCover (détaillée section 5.3) en sélectionnant (ou non) la case *Apply CommCover strategy*.

L'ensemble du plugin (transformation comprise) a été développé en utilisant le langage JAVA afin d'obtenir une homogénéité du code (la transformation aurait pu être développée à l'aide d'autres langages dédiés tel qu'ATL [JK06]).

5.5 Synthèse

Ce chapitre décrit un algorithme de réécriture permettant de transformer automatiquement tout diagramme SysML4MBT en diagramme UML4MBT. Cette manipulation permet de rendre automatique la génération de tests à partir de modèles SysML4MBT tout en utilisant l'outil Test Designer^{TM} (qui prend en entrée un modèle UML4MBT). Une surcharge à cette transformation permet d'obtenir une couverture de test identique à celle obtenue par l'application de la stratégie ComCover sur le modèle SysML4MBT initial. L'implémentation de ce processus dans l'outil de modélisation Topcased a permis l'obtention d'un processus automatique, qui a ainsi pu être expérimenté sur plusieurs cas d'étude présentés dans le chapitre suivant.

Chapitre 6

Chaîne outillée et expérimentations

Sommaire

Nos travaux ont permis l'élaboration d'une chaîne outillée dans le cadre du projet VETESS. Cette chaîne outillée permet, à partir d'un modèle SysML, de générer et d'exécuter des cas de test dédiés aux systèmes embarqués. Les éléments SysML autorisés (regroupés dans SysML4MBT) dans cette chaîne permettent la réalisation d'un modèle comportemental suffisamment complet pour la génération de tests relevants.

Il est important durant la phase de modélisation de représenter le système et son environnement. En effet, l'environnement correspond généralement à la raison d'être du système embarqué et il constitue un des principaux actionneurs de celui-ci. La prise en compte de la dualité Système/Environnement est réalisée à travers la définition de plusieurs diagrammes d'états-transitions au sein du modèle. Un diagramme d'états-transitions permet de capturer le comportement du système sous test tandis que chacun des autres constituent une représentation de la dynamique d'une partie de l'environnement. Ces diagrammes d'états-transitions ont pour caractéritiques d'une part d'évoluer en parallèle (ce qui est assuré au niveau du modèle par la réalisation de diagrammes d'états-transitions propres à chaque composant), et d'autre part de se synchroniser par envoi d'informations (signaux). De cette manière, une modification de l'état de l'environnement peut impacter l'état du système sous test (et vice-versa) au moyen de ces canaux de communication et permettre ainsi une vue réaliste du système. Ces canaux de communication sont précisés au sein du diagramme interne de bloc et permettent de représenter aussi bien des échanges électriques que des actions mécaniques, des transferts de fluides... Une fois le modèle réalisé, nous utilisons le plugin Topcased développé durant le projet pour transformer automatiquement le modèle SysML4MBT en UML4MBT tel que présenté dans le chapitre 5. Le modèle est ensuite exporté afin de pouvoir être traité par le générateur de tests Test Designer^{TM} de la société Smartesting.

A partir du modèle exporté, l'outil de génération de tests est capable de générer les tests comme présenté dans ce mémoire. Une fois les tests générés, il est possible d'exporter les tests afin de les jouer sur une plateforme de simulation. Le modèle SysML4MBT représentant une vue abstraite du système, les tests générés ne sont pas exécutables en l'état et doivent être concrétisés afin de les repositionner au même niveau que le système sous test. Un test est une séquence d'appel d'opérations. On associe, à chaque appel d'opération abstraite, un appel ou une séquence d'appels concrets. On fait de même pour les données (attributs du modèle et paramètres des opérations). Par exemple, on associe un attribut de modèle à une propriété du système concret. Cette étape est réalisée durant l'exportation à l'aide d'une couche d'adaptation. Cette dernière permet de faire le lien entre les tests abstraits et la plateforme d'exécution. Dans le cadre du projet VETESS, la plateforme d'éxécution utilisée est Test In View (TIV) développée par la société Clemessy.

Nous avons ainsi obtenu la chaîne outillée présentée sur la figure 6.1. Sur cette figure, les flèches pleines représentent les éléments développés au cours de cette thèse et du projet VETESS.

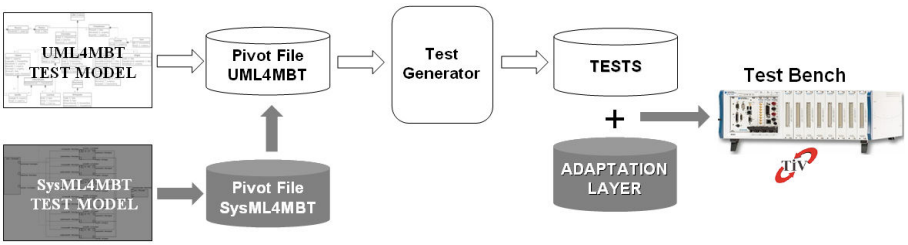


FIGURE 6.1 – Chaîne outillée VETESS

Il est à noter que SysML4MBT ne permet pas la représentation du temps. Or, le temps-réel est une notion très présente dans le domaine de l'embarqué (temps de réponse...). La première section de ce chapitre présente la manière d'appréhender cet aspect dans la chaîne outillée. Ensuite, afin de montrer la pertinence de nos travaux, nous allons présenter les résultats obtenus sur trois cas d'étude :

- La gestion du réglage électrique de la position d'un siège conducteur de véhicule présentant six caractéristiques différentes : position du siège, élévation de l'avant du siège, élévation de l'arrière du siège, profondeur de l'assise, inclinaison du dossier et hauteur de l'appui-tête.
- Le cas des essuie-glace avant d'un véhicule en considérant les fonctionnalités de balayage lent, rapide, intermittent ainsi que la fonction nettoyage.
- La colonne de direction d'un véhicule. Il s'agit en l'occurrence d'observer la réaction de la colonne de direction d'un véhicule en fonction de tracés de route.

Le premier cas d'étude a pour but de montrer la pertinence de l'approche SysML4MBT associée à ComCover alors que les deux autres permettent de montrer l'adaptabilité de la chaîne outillée aux systèmes discrets et continus.

6.1 Gestion des systèmes discrets et continus

Le temps-réel est une notion importante, en particulier dans le monde automobile. Par exemple, lorsque l'on appuie sur la pédale de frein et que les calculateurs d'ESP et d'ABS se mettent en marche, on souhaite que le véhicule freine effectivement et ce de façon rapide. Ensuite, au cours du temps, ils doivent prendre en compte les différentes conditions pour optimiser le freinage.

Dans ce contexte, la connaissance en continu de la situation exacte du système et le détail de son évolution s'avèrent être des éléments décisifs pour s'assurer de la fiabilité de tels systèmes. Les systèmes dont l'état n'est observé qu'à des moments précis sont dits *discrets*. Le détail de la variation du système entre deux états n'est pas prise en compte. A l'opposé, les systèmes dont l'état change en permanence au cours du temps et dont l'activité est fortement liée au temps qui s'écoule sont qualifiés de *continus*. Ces deux types de systèmes sont pris en compte dans la chaîne outillée bien que le modèle SysML4MBT ne permette qu'une représentation discrète.

Il existe des éléments du langage SysML qui permettent de modéliser les effets de l'écoulement du temps tels que les diagrammes paramétriques (utilisation de descriptions mathématiques avancées) et les diagrammes de séquence (description graphique du séquencement des exécutions dans le temps). Dans le cadre de ce travail, les diagrammes paramétriques et de séquence n'ont pas été retenus dans le langage SysML4MBT, et aucun profil dédié à l'expression de contraintes de type temps réel (comme MARTE) n'est supporté. Le langage SysML4MBT s'avère donc ne pas être assez complet pour permettre une gestion fine du temps-réel. Le choix de ne pas traiter ce type d'information, et les diagrammes SysML qui les caractérisent, est motivé par la nature du moteur de calcul mis en œuvre au sein de la chaîne outillée (technologie de l'outil Test Designer^{TM} basée sur des techniques de résolution de preuves). En effet, ce moteur de calcul n'est ni adapté à l'évaluation de formules mathématiques continues (qui requièrent la manipulation de flottants), ni en capacité de gérer des évaluations de type temps réel (réévaluation de l'état du système de manière continue par simulation d'horloge).

Pour adresser la problématique temps-réel, la démarche adoptée consiste à tirer parti de la dualité Système/Environnement du modèle SysML4MBT. Dans ce contexte, la démarche mise en œuvre pour la génération de tests à partir des deux familles de systèmes (continu et discret) va différer. Il est à noter qu'il est souvent possible de modéliser des systèmes continus en appliquant une abstraction plus ou moins forte pour les discrétiser.

6.1.1 Systèmes discrets

On parle de systèmes ou modèles *discrets* lorsque l'on s'intéresse à la situation du système à des moments précis (lorsque le système est dans un état stable) sans étudier les étapes de variation du système entre deux états stables. La réception d'un stimulus par un système peut ainsi provoquer un changement d'état de ce système : on s'intéresse alors à l'état stable résultant sans vouloir observer les étapes qui ont mené le système vers cet état stable ou le temps nécessaire à ce changement.

Ainsi, chaque stimulus implique des modifications connues sur les variables du système puisque les effets de chaque traitement ont été modélisés par une mise à jour des variables du système (par le biais des contraintes OCL dans le cas de la technologie développée dans le cadre du projet VETESS). Au terme de ces modifications, un état stable est atteint. Les valeurs assignées à chaque variable du modèle constituent alors les valeurs attendues sur le système réel après exécution de cette même succession d'envois de stimuli. Cette démarche permet ainsi d'utiliser le modèle SysML4MBT pour prédire les valeurs attendues des systèmes discrets et calculer le verdict de test. Le modèle sert aussi pour générer des cibles de test dont le calcul est directement basé sur les modifications induites par chaque stimulus.

La mise en œuvre de cette démarche est assurée par la chaîne outillée présentée sur la figure 6.1 (page 133). Dans le cas des systèmes discrets, la fin de la chaîne se précise comme représenté par la figure 6.2. L'étape de comparaison se fait automatiquement au sein de l'outil TestInView.

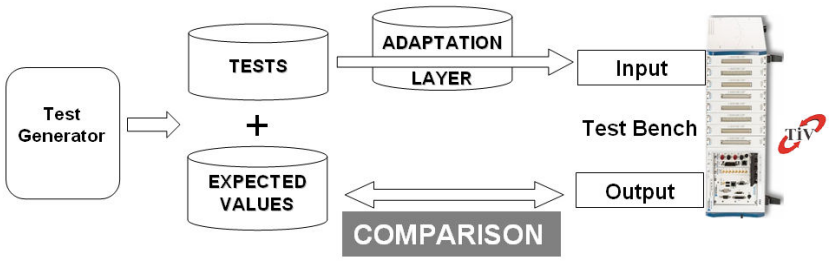


FIGURE 6.2 – Usage de la chaîne outillée VETESS pour les systèmes discrets

6.1.2 Systèmes continus

Dans le cadre de systèmes continus, la situation du système change (ou peut changer) au fil du temps sans qu'aucun stimulus ne soit explicitement invoqué (on parle d'effet d'auto-stabilisation ou d'auto-régulation du système). L'état du système est donc directement dépendant du temps qui s'écoule. Dans la démarche proposée dans ces travaux, le modèle défini avec le langage SysML4MBT ne permet pas de spécifier ce type de comportement. Par conséquent, ce modèle ne peut pas être utilisé seul pour prédire les valeurs attendues à observer sur le système réel. L'usage de la chaîne outillée a donc été adaptée pour gérer ce type de systèmes.

Le début de la chaîne outillée reste inchangé. Les différences se situent en aval de la génération de tests. Tout d'abord, au niveau de la couche d'adaptation, il est nécessaire de préciser le temps qui s'écoule entre deux appels d'opération. Ensuite, pour prédire les valeurs attendues, il est nécessaire d'utiliser un modèle complémentaire permettant de calculer l'évolution de certaines variables dépendantes du temps. Pour ce faire, un modèle Matlab/Simulink est utilisé. Le calcul des verdicts est réalisé automatiquement au sein de l'outil TIV à l'aide de ce modèle permettant ainsi de compléter les tests générés. La chaîne outillée mise en œuvre pour les systèmes continus est représentée sur la figure 6.3.

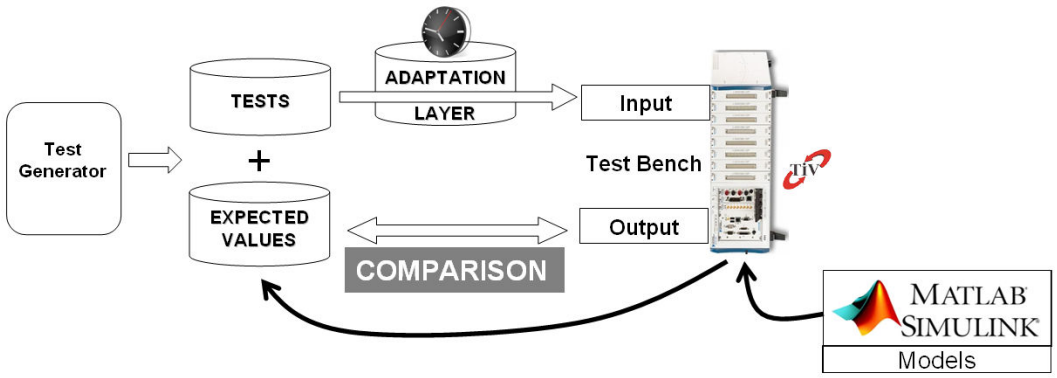


FIGURE 6.3 – Usage de la chaîne outillée VETESS pour les systèmes continus

Les parties suivantes présentent trois cas d'étude montrant la pertinence de ces chaînes outillées.

6.2 Réglage électronique d'un siège de véhicule

Cette première étude de cas correspond à la gestion du réglage électronique d'un siège de voiture. Il s'agit d'un système discret. N'ayant pas à notre disposition de banc de test ou de version simulée de ce système, ce cas d'étude a uniquement permis l'expérimentation de la première partie de la chaîne outillée (jusqu'à l'obtention du modèle UML4MBT). Cette spécification est tirée de [PFAR02].

Comme représenté sur le figure 6.4, l'utilisateur peut :

- avancer/reculer le siège entier (`LA`),
- élever/abaisser l'avant du siège (`FH`),
- élever/abaisser l'arrière du siège (`RH`),
- agrandir/rétrécir la profondeur de l'assise (`SD`),
- incliner le dossier (`B`),
- régler la hauteur de l'appui-tête (`HR`).

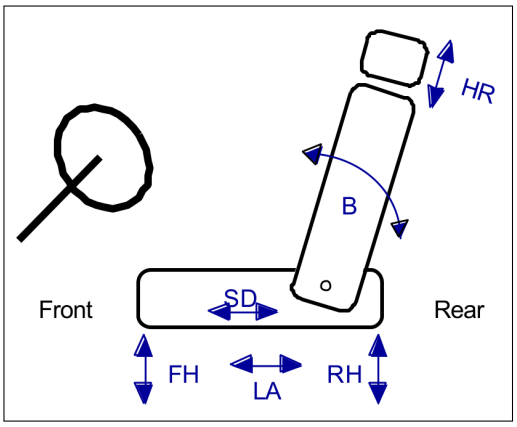


FIGURE 6.4 – Schéma représentant le contrôle de siège

Chacun de ces réglages est effectué à l'aide d'un moteur. Chaque moteur a une amplitude maximum. Un moteur peut être à l'arrêt (`ZERO`), tourner dans un sens (`PLUS`) ou dans l'autre (`MINUS`). Lorsqu'un moteur est en marche, des tops d'horloge émis par le système réalisent l'avancement effectif du moteur. L'amplitude est caractérisée par un nombre de tops donné. Chacun des moteurs a donc une position minimale et une position maximale.

Tous les moteurs ne peuvent pas fonctionner en même temps. Il y a deux groupes de moteurs : le groupe 1 qui contient les moteurs `LA`, `RH` et `SD` et le groupe 2 qui contient les moteurs `B`, `FH` et `HR`. Il ne peut y avoir simultanément et au maximum qu'un seul moteur en fonctionnement dans chaque groupe. En revanche, deux moteurs appartenant à des groupes distincts peuvent fonctionner au même moment.

Les moteurs ont un ordre de priorité. Si dans un groupe donné, on demande l'allumage d'un moteur alors qu'un moteur plus prioritaire est déjà en marche, la nouvelle commande est ignorée et annulée. De même un moteur doit être coupé si une commande d'allumage d'un moteur plus prioritaire du même groupe survient. L'ordre de priorité est le suivant (du plus prioritaire au moins prioritaire) :

– Groupe 1 : `LA - RH - SD`
– Groupe 2 : `B - FH - HR`

En plus des 6 moteurs décrits précédemment, le système contient également un panneau de contrôle duquel arrivent les commandes. Ce panneau est équipé de six boutons poussoirs doubles : un par moteur. Ces boutons sont des interrupteurs à trois positions revenant automatiquement en position centrale (position `STOP`) quand il sont relâchés comme ceux couramment utilisés pour le réglage des vitres électriques d'une voiture par exemple. Chaque bouton accepte trois positions : la position par défaut qui correspond à une absence d'activité, une position correspondant à l'actionnement du moteur dans

un sens et une position correspondant à l'actionnement du moteur dans l'autre sens. De par la structure même du bouton, si au cours d'un déplacement, on souhaite changer de sens, il faut nécessairement passer par la position `STOP`. Vis-à-vis des priorités, même si le moteur en action a atteint une limite (position minimale ou maximale), il est considéré comme utilisé tant que le bouton est pressé.

La gestion du contrôle de siège est un système continu (gestion d'une horloge qui commande le déplacement effectif du siège), il est cependant plutôt aisé de modéliser cette horloge à l'aide du modèle SysML4MBT car elle n'engendre pas de calculs complexes et il est facile de représenter une vue abstraite de cette horloge. La version *discrète* de la chaîne outillée est donc utilisée.

La section suivante décrit le modèle réalisé pour représenter ce cas d'étude. Ensuite, une section présente les tests générés sur ce cas d'étude par les stratégies présentes dans ce mémoire : celle utilisée par l'outil Test Designer^{TM} (couverture du critère *Toutes les transitions* appuyé par D/CC) et ComCover. Une synthèse dresse en fin de section un bilan de ce cas d'étude.

6.2.1 Modèle

Le modèle de ce cas d'étude est spécifié à l'aide des éléments SysML4MBT présentés dans la section 3.2. On réalise tout d'abord le diagramme d'exigences, on passe ensuite à la représentation statique du système (diagrammes de définition de bloc et diagramme interne de bloc) puis enfin, à sa représentation dynamique (diagramme d'états-transitions).

Diagramme d'exigences

D'après le descriptif énoncé précédemment, il est possible de définir plusieurs exigences :

- La priorité au niveau de l'utilisation des moteurs.
- La modification de la position du siège si rien ne l'empêche.
- Un moteur qui n'est pas concerné par une action ne doit pas s'actionner.
- La réalisation du mouvement de siège à chaque top d'horloge.

La figure 6.5 représente le diagramme d'exigences correspondant.

Chacune des exigences de fin (celles qui ne sont pas dérivées) est détaillée (au sein d'une autre vue non représentée ici dans un souci de lisibilité) afin de l'associer à chaque moteur pour pouvoir les faire correspondre aux éléments de modèles.

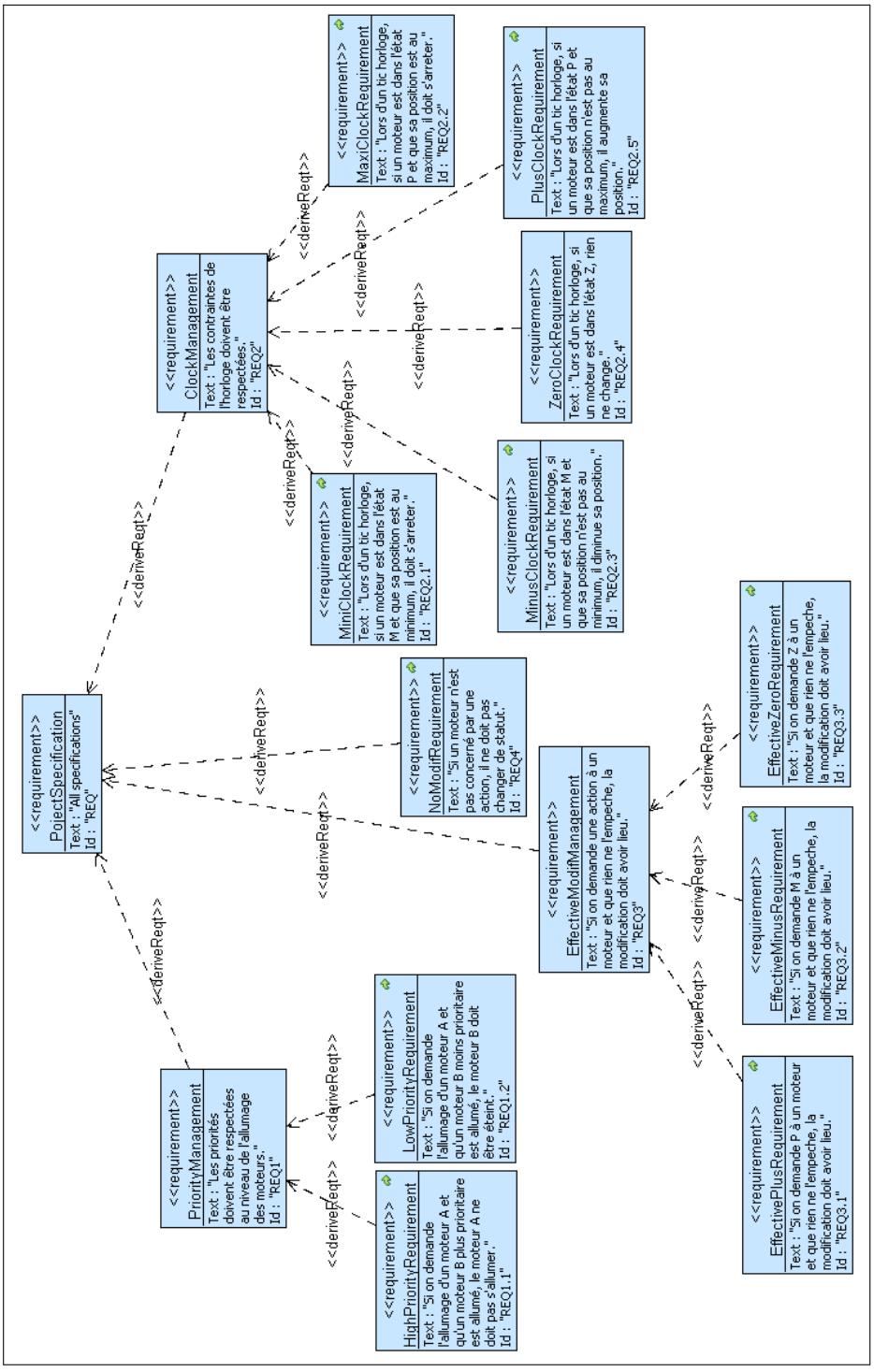


FIGURE 6.5 – Diagramme d'exigences de contrôle de siège

Représentation statique

Le système global est représenté dans le diagramme de bloc par le bloc `SitControl`. Ce bloc est composé de plusieurs blocs. Il y a :

- Le bloc `Command` qui représente le panneau de contrôles du système. C'est de là que l'utilisateur interagit avec le système.
- Le bloc `Clock` qui représente l'horloge du système. C'est la composante qui s'occupe de transmettre les tops d'horloge aux moteurs.
- Un bloc par moteur, c'est-à-dire six blocs : `LA`, `RH`, `SD`, `B`, `FH` et `HR`.

Le diagramme de bloc est représenté sur la figure 6.6.

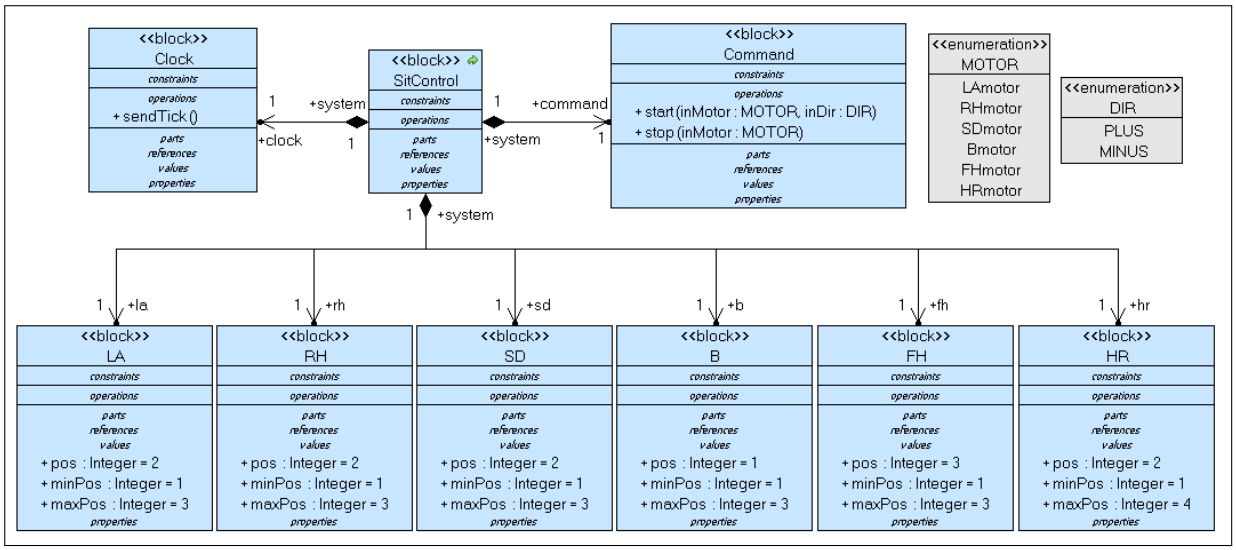


FIGURE 6.6 – Diagramme de bloc du contrôle de siège

Le bloc `Clock` contient une seule opération qui correspond au déclenchement d'un top d'horloge. L'opération `start` du bloc `Command` correspond à l'actionnement d'un des boutons du panneau de contrôle. Cette opération a deux paramètres d'entrée correspondant au moteur concerné et au sens dans lequel l'activation est demandée. On utilise deux énumérations : une pour représenter les six moteurs et une pour représenter les sens possibles. On considère le sens plus (`PLUS`) et le sens moins (`MINUS`). Ils correspondent aux deux directions que peuvent prendre chacune des composantes. L'action `stop` de ce même bloc représente le relâchement du bouton poussoir correspondant au moteur précisé en paramètre (ce qui correspond à l'arrêt du moteur).

Enfin, chacun des blocs représentant un moteur contient trois propriétés :

- une pour la position actuelle de la composante associée (`pos`),
- une pour sa position minimum (`minPos`),
- et une pour sa position maximum (`maxPos`).

On représente les positions à l'aide d'entiers. On considère trois positions : la position minimale (`1`), la position intermédiaire (`2`) et la position maximale (`3`). La position par défaut de chaque moteur est fixée à `2`.

Représentation des communications

Au niveau des communications à représenter, il y a tout d'abord la transmission des ordres du panneau de contrôle aux différents moteurs. En effet, lorsqu'un bouton est actionné (appel de l'opération `start`), un signal électrique est transmis au moteur correspondant pour ordonner son allumage. De même, lorsque le bouton est relâché (appel de l'opération `stop`), un signal ordonnant l'arrêt du moteur concerné doit être transmis à celui-ci. On met ainsi en place trois signaux : deux pour la transmission de l'ordre d'allumage d'un moteur (un pour le sens `PLUS` et un pour le sens `MOINS`) et un pour la transmission de l'ordre d'arrêt d'un moteur. Ces trois signaux sont regroupés au sein de la spécification de flux `MotorsSignals` afin de pouvoir spécifier les ports permettant de transporter ces signaux. On met également en place le signal `TICKsignal` qui correspond à la transmission des tops d'horloge aux moteurs. Ces éléments sont représentés sur la figure 6.7.

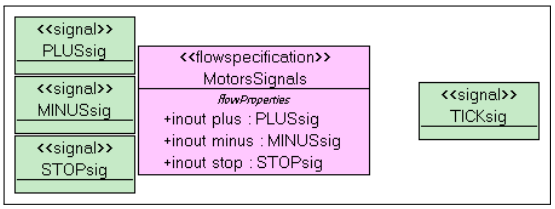


FIGURE 6.7 – Signaux et spécification de flux du diagramme de bloc du contrôle de siège

L'étape suivante consiste à définir les ports et les connecteurs permettant de faire transiter ces signaux. Chaque moteur comporte deux ports : un pour la réception des ordres de la part du panneau de contrôle et un pour la réception des tops d'horloge. Il est de ce fait également nécessaire de mettre en place les ports correspondant au niveau des blocs `Command` et `Clock`. On obtient donc le diagramme interne de bloc représenté sur la figure 6.8. On passe ensuite à la représentation dynamique du système.

Représentation dynamique

Tout d'abord, on représente le diagramme d'états-transitions du panneau de contrôle. Les priorités entre les moteurs sont gérées au niveau de ce composant. Ainsi, on utilise un état parallèle qui contient deux régions, une par groupe de moteurs. Initialement, chacune

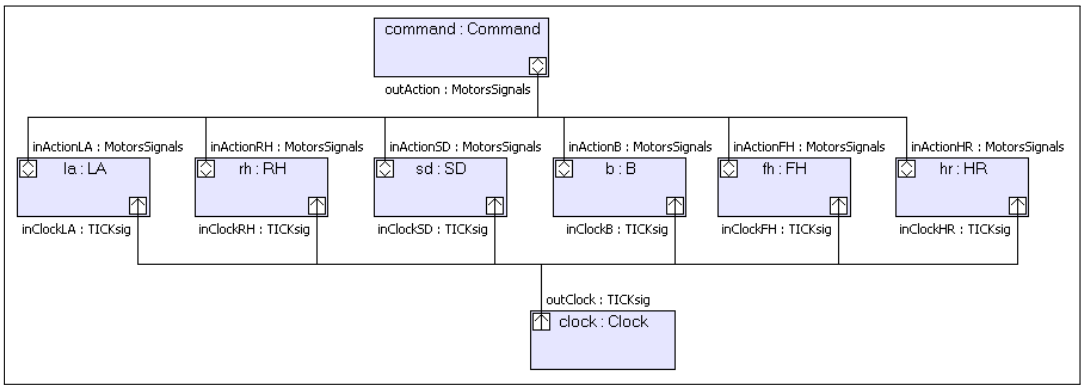


FIGURE 6.8 – Diagramme interne de bloc du contrôle de siège

de ces régions est dans l'état `OFF` qui correspond à l'état où chaque bouton est dans l'état par défaut (aucun bouton actionné). Ensuite, plusieurs transitions permettent de réaliser des changements d'états. Le diagramme d'états-transitions obtenu est représenté sur la figure 6.9. Les numérotations de transitions (`TR1`, `TR2`...) sont nécessaires pour les explications de la section suivante.

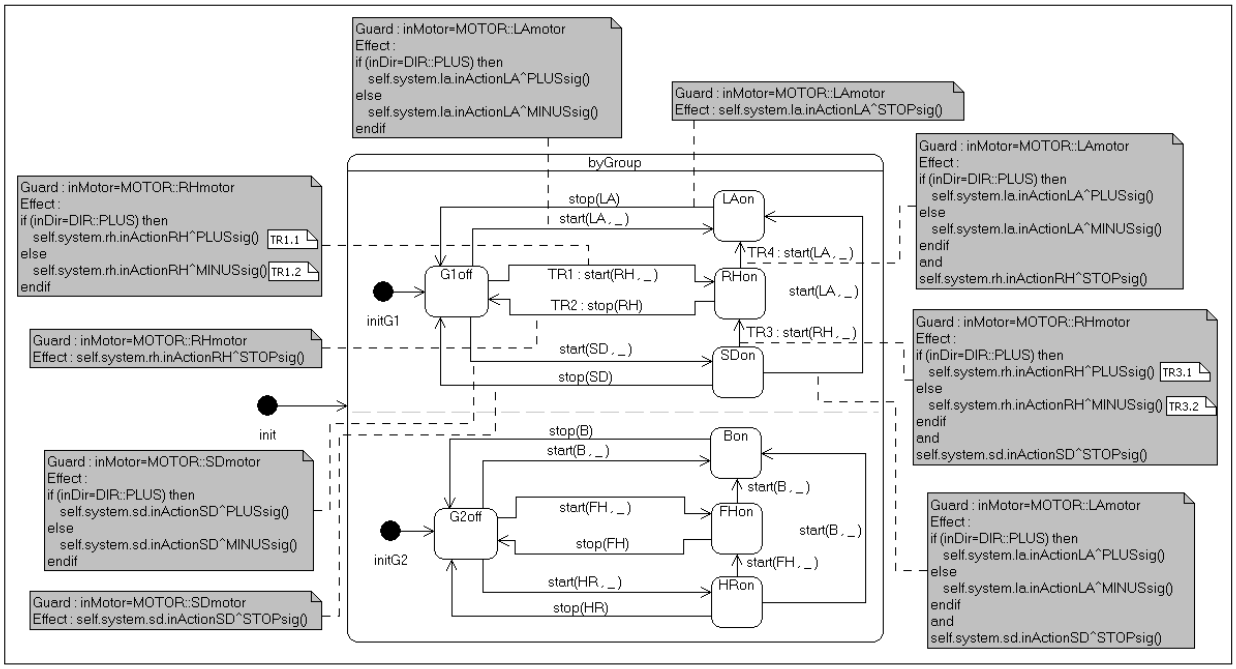


FIGURE 6.9 – Diagramme d'états-transitions du bloc `Command` du contrôle de siège

Seules les expressions OCL pour le groupe 1 (`LA`, `RH` et `SD`) sont précisées (région supérieure de l'état parallèle). Les mêmes expressions sont naturellement présentes au niveau du groupe 2 (région inférieure de l'état parallèle).

En ce qui concerne la représentation dynamique du bloc `Clock`, l'opération `sendTick` porte la post-condition suivante :

```
    self.system.la.inClockLA^TICKsig()
and self.system.rh.inClockRH^TICKsig()
and self.system.sd.inClockSD^TICKsig()
and self.system.b.inClockB^TICKsig()
and self.system.fh.inClockFH^TICKsig()
and self.system.hr.inClockHR^TICKsig()
```

Etant donné que la spécification de l'opération `sendTick` suffit à décrire le fonctionnement de l'horloge, le diagramme d'états-transitions est facultatif. Cependant, afin d'uniformiser le modèle, on définit le diagramme d'états-transitions représenté sur la figure 6.10.

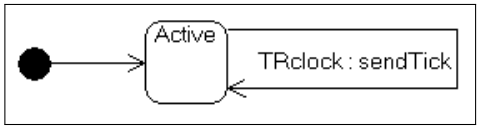


FIGURE 6.10 – Diagramme d'états-transitions du bloc `Clock` du contrôle de siège

Finalement, il est nécessaire de réaliser un diagramme d'états-transitions par moteur. Ces six diagrammes sont identiques. A titre d'exemple, le diagramme d'états-transitons du moteur `RH` est représenté sur la figure 6.11. Le diagramme d'états-transitions du moteur `LA` est identique à la différence près que les `RH` sont remplacés par `LA`.

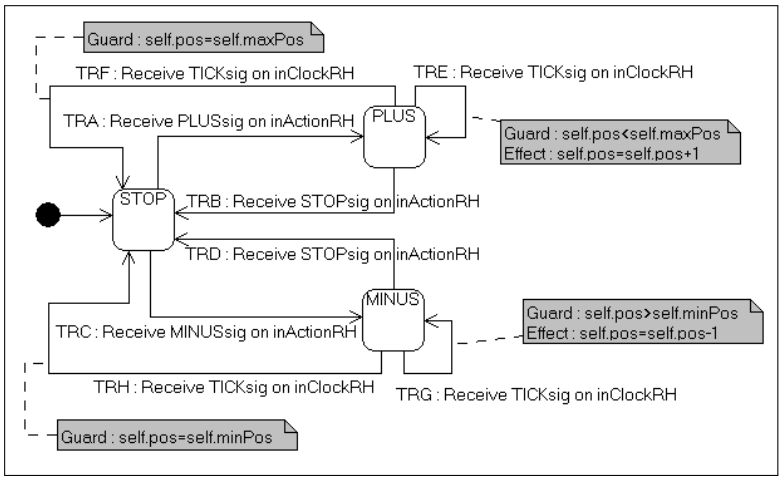


FIGURE 6.11 – Diagramme d'états-transitions du bloc `RH` du contrôle de siège

Toutes les transitions de ce diagramme correspondent à une réception de signal. Seules les réceptions de signal provenant de l'horloge engendrent un effet (action OCL) qui consiste à modifier la position du siège (propriété `pos` du bloc concerné).

Les autres transitions ont pour seul effet d'effectuer un changement d'état. Dans le cas de l'état `STOP`, aucune transition ne permet la réception du top d'horloge. En effet, si le moteur est à l'arrêt, le top d'horloge éventuellement transmis est ignoré. De plus, lorsqu'un top d'horloge est reçu mais que le moteur ne peut pas aller plus loin (limite minimale ou maximale atteinte), le moteur est arrêté même si le bouton n'est pas relâché. Le priorité est cependant conservée tant que le bouton est actionné.

Le modèle étant terminé, la stratégie de génération de tests mise en place durant ces travaux (ComCover) est appliquée.

6.2.2 Génération de tests

Dans un souci de lisibilité, seuls les tests qui concernent le moteur `RH` sont exprimés. L'approche s'applique cependant à l'identique à tous les autres moteurs.

Afin de pouvoir évaluer la pertinence de la stratégie ComCover sur ce modèle, nous effectuons une comparaison entre les tests générés avec cette stratégie et ceux générés avec les critères *Toutes les transitions* et D/CC.

Pour simplifier la présentation de la comparaison, nous parlerons uniquement des transitions numérotées dans les figures 6.9, 6.10 et 6.11. Ce sont celles qui concernent le moteur `RH`. Il y a treize transitions concernées (`TRclock`, de `TR1` à `TR4` et de `TRA` à `TRH`) dont deux transitions qui se décomposent en deux sous-cas (`TR1` et `TR3`). Ainsi, quinze cibles de test sont générées. Les tests permettant de couvrir ces cibles sont représentés dans le tableau 6.1. Comme dans la section 4.6, nous ne représentons ici que les tests qui ne sont pas inclus dans d'autres.

Afin de pouvoir simuler l'application de la stratégie ComCover, nous dressons la liste des cibles de test. Il est possible d'en prédire approximativement le nombre. En effet, d'après la formule présentée dans la partie 4.5.4, on a :

$$\boxed{r + e - c \leqslant \mathit{cibles} \leqslant (e - c + 1) * (r - c + 1) + (c - 1)}$$

Etant donné que deux connecteurs unidirectionnels sont liés au moteur RH, et que l'un deux permet la transmission de 3 signaux différents, on a $c = 4$. D'après les modèles réalisés, $e = 7$ (`TRclock`, `TR1`(*2), `TR2`, `TR3`(*2) et `TR4`) et $r = 8$ (de `TRA` à `TRH`). On obtient :

$$\boxed{(8 + 7 - 4 = 11) \leqslant \mathit{cibles} \leqslant ((7 - 4 + 1) * (8 - 4 + 1) + (4 - 1) = 23)}$$

Le nombre théorique de couples générés est donc compris entre 11 et 23. En l'occurrence, 12 couples sont générés. Ils sont représentés dans le tableau 6.2 et associés aux tests qui permettent de les couvrir.

Cibles	Tests
TRB Arrêt du moteur RH en position PLUS. (TR1.1, TR2, TRA)	T1 [⟨start(RH,PLUS),TR1.1,[TR1.1;TRA]⟩; ⟨stop(RH),TR2,[TR2;TRB]⟩]
TRD Arrêt du moteur RH en position MINUS. (TR1.2, TRC)	T2 [⟨start(RH,MINUS),TR1.2,[TR1.2;TRC]⟩; ⟨stop(RH),TR2,[TR2;TRD]⟩]
TRF Tic d'horloge avec RH en PLUS au maximum. (TRE,TRclock)	T3 [⟨start(RH,PLUS),TR1.1,[TR1.1;TRA]⟩; ⟨sendTick(),TRclock,[TRclock;TRE]⟩; ⟨sendTick(),TRclock,[TRclock;TRF]⟩]
TRH Tic d'horloge avec RH en MINUS au minimum. (TRG)	T4 [⟨start(RH,MINUS),TR1.2,[TR1.2;TRC]⟩; ⟨sendTick(),TRclock,[TRclock;TRG]⟩; ⟨sendTick(),TRclock,[TRclock;TRH]⟩]
TR3.1 Allumage de RH (PLUS) avec SD allumé.	T5 [⟨start(SD,_),_,_]⟩; ⟨start(RH,PLUS),TR3.1,[TR3.1;{TRA,_}]⟩]
TR3.2 Allumage de RH (MINUS) avec SD allumé.	T6 [⟨start(SD,_),_,_]⟩; ⟨start(RH,MINUS),TR3.2,[TR3.2;{TRC,_}]⟩]
TR4 Allumage de LA avec RH allumé.	T7 [⟨start(RH,PLUS),TR1.1,[TR1.1;TRA]⟩; ⟨start(LA,PLUS),TR4,[TR4;{TRB,_}]⟩]

TABLE 6.1 – Tests générés par la stratégie *Toutes les transitions*+D/CC sur le cas du contrôle de siège

Cibles	Tests
TR2/TRB (TR1.1/TRA)	T1
TR2/TRD (TR1.2/TRC)	T2
TRclock/TRF (TRclock/TRE)	T3
TRclock/TRH (TRclock/TRG)	T4
TR3.1/TRA	T5
TR3.2/TRC	T6
TR4/TRB Allumage de LA avec RH en PLUS.	T7 [⟨start(RH,PLUS),TR1.1,[TR1.1;TRA]⟩; ⟨start(LA,PLUS),TR4,[TR4;{TRB,_}]⟩]
TR4/TRD Allumage de LA avec RH en MINUS.	T8 [⟨start(RH,MINUS),TR1.2,[TR1.2;TRC]⟩; ⟨start(LA,PLUS),TR4,[TR4;{TRD,_}]⟩]

TABLE 6.2 – Tests générés par ComCover sur le cas du contrôle de siège

Les six premiers tests sont identiques à la couverture du modèle par le critère *Toutes les transitions*. Le septième test qui était précédemment suffisant pour couvrir la transition `TR4`, ne suffit pas à couvrir les deux couples associés à cette transition. Ainsi, on obtient les tests `T7` et `T8`. Un test de plus est donc généré.

On remarque que ComCover permet une plus grande couverture du modèle car cette stratégie engendre la génération de nouveaux tests par rapport à la stratégie initiale tout en conservant les tests générés par celle-ci. De plus, le nombre de cibles de test générées est moindre que pour la stratégie basée sur le critère *Toutes les transitions* (12 contre 15). La stratégie ComCover peut ainsi être considérée comme plus efficace sur cet exemple.

6.2.3 Bilan

Ce cas d'étude permet de montrer, comme avec l'exemple fil rouge, que SysML4MBT est adapté à la modélisation des systèmes embarqués. La stratégie ComCover a permis l'obtention de tests qui ne sont pas générés par la stratégie par défaut consistant à couvrir toutes les transitions du modèle (critère *Toutes les transitions*). Au total, sur l'ensemble du modèle, six tests générés par la stratégie ComCover ne le sont pas lors de l'application du critère *Toutes les transitions*. Afin de générer ces tests automatiquement, le modèle SysML4MBT est réécrit en modèle UML4MBT en utilisant les règles de réécriture dédiées présentées chapitre 5. Ces tests sont ensuite générés automatiquement par l'outil de génération de tests Test Designer^{TM}.

Au cours de ce mémoire, trois cas d'étude (Gestion de l'éclairage avant du véhicule (exemple fil rouge), arrêt d'urgence d'un train (chapitre 4) et réglage électrique du siège conducteur) ont montré que SysML4MBT est adapté à la modélisation des systèmes embarqués et que ComCover est relevant pour la couverture des communications. Les deux prochaines sections font un focus sur la pertinence et l'adaptabilité de la chaîne outillée complète vis-à-vis des systèmes discrets (à travers le cas d'étude des essuie-glace) et des systèmes continus (à travers le cas de la colonne de direction).

6.3 Essuie-glace avant d'un véhicule

Le cas d'étude FrontWiper vise à étudier un système discret qui concerne le fonctionnement de l'essuyage du pare-brise d'un véhicule. Ce système prend en compte les fonctions manuelles de balayage des essuie-glace en vitesse lente ou rapide. Une fonction intermittente dont la vitesse d'essuyage dépend de la vitesse du véhicule, une fonction de nettoyage et l'essuyage automatique avec capteur de pluie sont également considérés. Ces commandes sont activables et désactivables par l'utilisateur à l'aide d'un commodo.

6.3.1 Modèle

Nous commençons notre travail par la réalisation du modèle du système et de son environnement. La prise en compte de la dualité Système/Environnement, est réalisée à travers l'utilisation de blocs SysML4MBT. Ils représentent soit le système sous test (`FrontWiper_Subsystem`), soit son environnement (par exemple, le démarreur du véhicule est représenté par un bloc spécifique).

Le système (`FrontWiper_Subsystem`) est composé du module qui gère les priorités au niveau des demandes de mise en fonctionnement des essuie-glace (`Arbitration_module`), du module gérant le fonctionnement effectif des balais (`ManualWiperControlModule`) et du module dédié au nettoyage (`Washing_module`).

L'environnement est, quant à lui, composé de 7 parties. Chacune de ces parties communique avec le système (`FrontWiper_Subsystem`). Les modules sont les suivants :

- le système d'allumage qui permet de savoir si le contact est mis ou non : `WiredInput`,
- le module qui permet d'indiquer l'état du commodo manipulé par le conducteur : `CombinationSwitch`,
- le module `Serial_link` représentant la communication avec le capteur de pluie,
- le calculateur de mouvement du véhicule : `ComputingVehiculeMoving`,
- un ensemble de modules pour la gestion des temporisations,
- le bus CAN qui permet de transmettre les demandes d'essuyage,
- la mémoire EEPROM qui permet de stocker la configuration du véhicule (type du véhicule...).

Ces différents éléments communiquent avec le système à l'aide de la transmission de signaux. Tous les éléments sont illustrés par le diagramme de bloc de la figure 6.12.

Le diagramme de bloc contient 15 blocs. Il y a un total de 18 connecteurs et les 12 diagrammes d'états-transitions contiennent entre 2 et 53 transitions. Ces éléments de modélisation ne sont pas détaillés dans ce document. Pour plus de détails, vous pouvez consulter le livrable publique L4.1 [VET10b] du projet VETESS [VET10a].

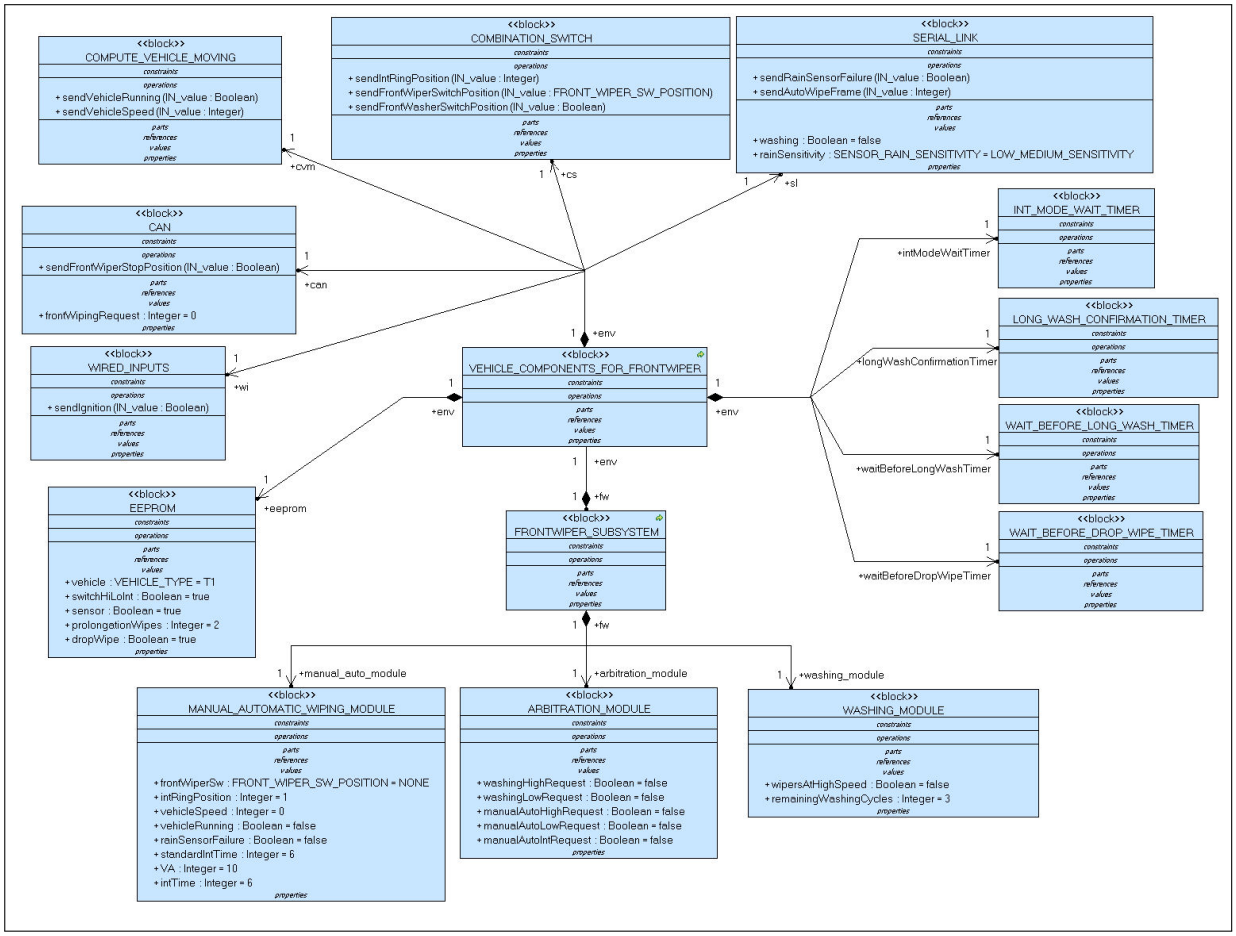


FIGURE 6.12 – Diagramme de bloc de FrontWiper

6.3.2 Génération de tests sur le modèle SysML

La première étape consiste à dérouler manuellement les stratégies de génération de tests énoncées dans ce mémoire. Sur le modèle SysML4MBT de ce cas d'étude, l'application du critère de couverture *Toutes les transitions* (appuyé par D/CC) engendre un total de 103 cibles de tests. 85 tests sont générés afin de couvrir ces cibles. Dans le cas de la stratégie ComCover, 130 couples d'envois/réceptions de signaux sont formés nécéssitant la génération de 104 tests (10 cibles inaccessibles) : 19 nouveaux tests sont donc générés par ComCover. Le livrable public L1.2 [VET09] du projet VETESS [VET10a] présente l'intégralité des tests générés par la stratégie ComCover sur le cas d'étude FrontWiper.

6.3.3 Génération de tests Test Designer^{TM}

Afin de pouvoir générer les tests automatiquement à l'aide de l'outil Test Designer^{TM}, il faut appliquer la transformation de modèle présentée au chapitre 5. Compte tenu de la taille du modèle SysML4MBT, la transformation du modèle SysML4MBT vers UML4MBT génère un modèle contenant 2526 états et 31873 transitions. Cette explosion combinatoire est due au nombre important de diagrammes d'états-transitions SysML4MBT en parallèles et à la stratégie utilisée pour les fusionner qui est basée sur un produit cartésien des diagrammes d'états-transitions. Les limitations de l'outil Test Designer^{TM} empêche malheureusement le chargement d'un tel modèle. Nous avons donc réalisé manuellement le modèle UML4MBT de ce système. Ce modèle est plus simple que celui issu de la transformation car l'utilisation de structures au sein des diagrammes d'états-transitions tels que les états composites ou les points de choix ont permis la fusion de certains comportements. Ensuite, les communications n'ont pas été représentées et les comportements associés ont été abstraits ce qui permet ainsi de diminuer le nombre de comportements modélisés.

Deux configurations de véhicule ont été considérées :

- Configuration 1 : les fonctionnalités considérées sont l'essuyage en vitesse lente, en vitesse rapide, en vitesse intermittente et le nettoyage. Dans cette configuration, on gère également l'essuyage automatique avec capteur de pluie et le ralentissement de la vitesse d'essuyage lorsque le véhicule est à l'arrêt. Dans ce cas, la vitesse d'essuyage du mode intermittent est directement proportionnelle à la vitesse du véhicule.
- Configuration 2 : le mode automatique avec capteur de pluie et le ralentissement de la vitesse à l'arrêt ne sont pas gérés. De plus, la vitesse d'essuyage du mode intermittent est calculée selon quatre fourchettes de vitesse de véhicule.

L'ensemble des tests pour ces deux configurations a été généré en 4 minutes 23 secondes. Les données de génération sont représentées dans le tableau 6.3. Bien que certains tests soient communs aux deux configurations, l'état initial étant différent, il ne sont pas considérés comme identiques. Ainsi, la dernière colonne correspond à la somme des deux précécentes.

	Configuration 1	Configuration 2	Config 1 et 2 confondues
Nbre total de comportements	74	61	135
Nbre de tests générés	59	54	113
Nbre de comportements couverts	73 (1 inatteignable)	60 (1 inatteignable)	133 (2 inatteignables)

TABLE 6.3 – Tests générés par la stratégie *Toutes les transitions*+D/CC sur le cas des essuie-glace

6.3.4 Publication et exécution des tests

Les tests générés sont ensuite exporté et concrétisés afin d'être exécutés. Dans le cadre de cet exemple, n'ayant pas accès à un système physique (banc de test), un modèle de simulation a été utilisé. Il s'agit d'une représentation virtuelle du système à tester. L'ensemble des tests générés par Test Designer^{TM} ont ainsi pu être exécutés sur le modèle de simulation au sein de l'outil TIV en 25 minutes et 26 secondes. L'interface de TestInView au cours de l'exécution d'un test est visible sur la figure 6.13.

6.3.5 Bilan

Ce cas d'étude a permis de montrer la pertinence de SysML4MBT pour la modélisation de systèmes embarqués. Il a cependant mis en lumière les limitations de la chaîne outillée dans le cas de modèles d'une certaine taille. Une perspective d'amélioration consiste donc à perfectionner le passage à l'échelle afin d'obtenir, à l'issu de la transformation SysML4MBT vers UML4MBT, des modèles contenant moins de comportements. Cette limitation a rendu irréalisable l'exploitation de la chaîne outillée d'un bout à l'autre sur ce cas d'étude. Néanmoins, afin de montrer la pertinence de la chaîne outillée vis-à-vis des systèmes discrets, un modèle UML4MBT, correspondant à une version simplifiée du modèle UML4MBT initialement obtenu, a été créé. Les tests générés à partir de ce modèle ont pu être exécutés sur une version simulée du système sous test.

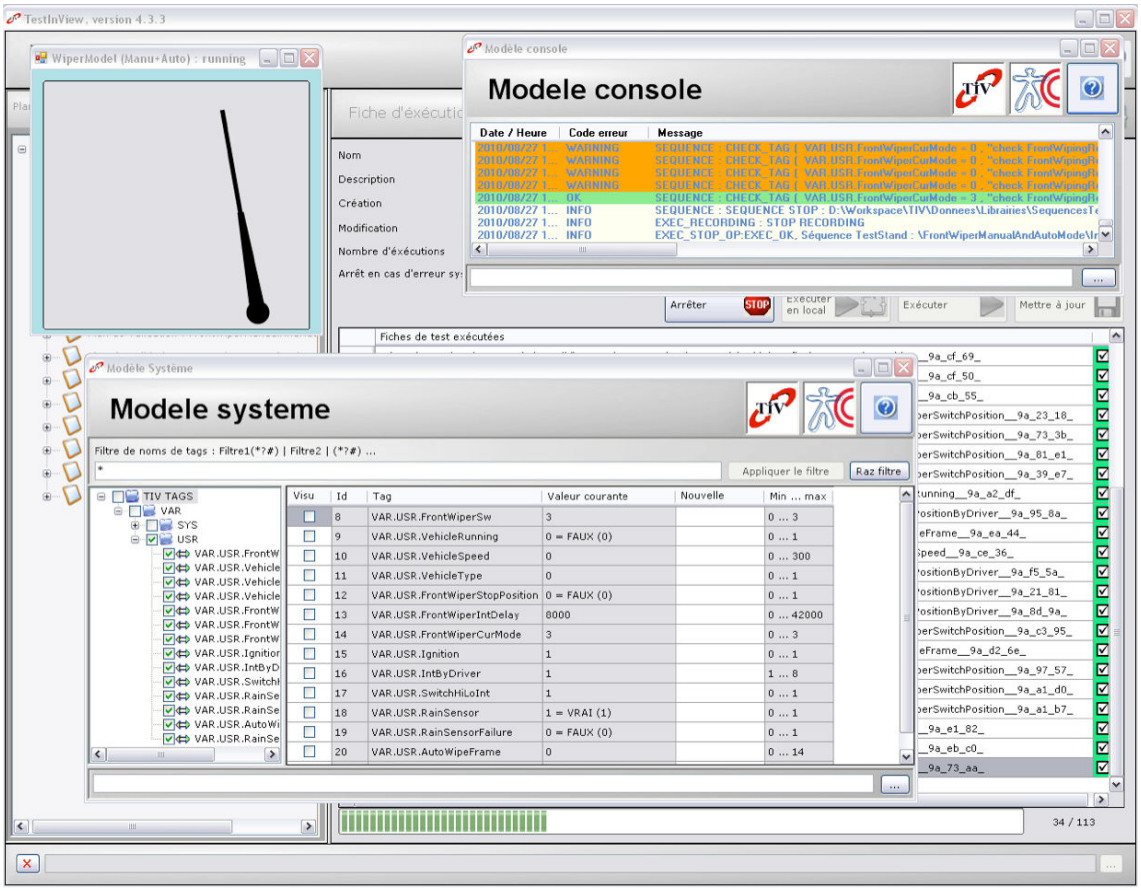


FIGURE 6.13 – Exécution des tests sur le modèle de simulation des essuie-glace

6.4 Colonne de direction d'un véhicule

Le cas d'étude de la colonne de direction vise à étudier le comportement du train avant d'une voiture. La figure 6.14 présente une vue schématique des éléments physiques constituant ce système.

Comme pour les autres cas d'étude, le système et son environnement sont modélisés. En l'occurrence, l'environnement de ce système correspond à la route et à ses caractéristiques. De ce fait, pour pouvoir valider ce système, il faut définir des tracés de route permettant la mise en situation de manière réaliste du train avant et de vérifier que sa réaction correspond bien à ce qui est attendu. Le modèle spécifie le profil de la route suivie par le véhicule (virages, pentes, dévers, trou, bosses...) ainsi que les caractéristiques dynamiques du véhicule (vitesse). On observe ensuite la réaction des suspensions (déformations), des roues (angles) et la position du volant. Il s'agit d'un système continu car la réaction du véhicule doit être observée et évaluée sans interruption.

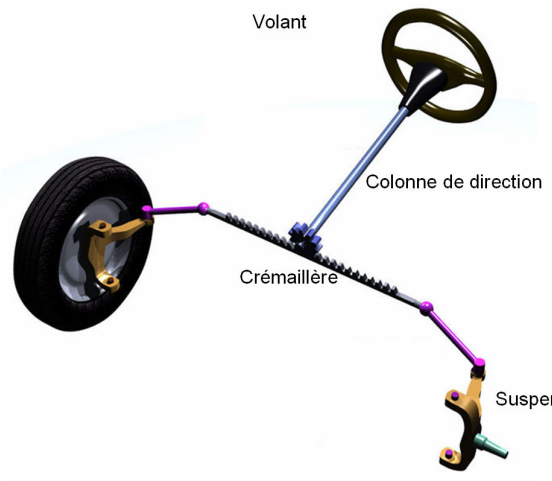


FIGURE 6.14 – Représentation schématique de la colonne de direction d'un véhicule

6.4.1 Modèle

L'activation d'une colonne de direction se fait par le déplacement du véhicule sur une route. De ce fait, les caractéristiques de la route et la vitesse du véhicule définissent les éléments qui permettent le changement d'état de cet organe mécatronique. Chaque caractéristique de la route (Adhérence, Direction, Dévers et Pente) et la vitesse du véhicule est représentée par un bloc. Le système est représenté par le bloc `Column`. Le diagramme de bloc SysML4MBT de ce système est illustré sur la figure 6.15.

L'approche de modélisation utilisée est la suivante : on utilise l'opération `step` pour représenter l'avancée effective du véhicule. Entre deux pas, il est possible de réaliser une seule modification pour chacune des quatre composantes de la route. les caractéristiques gérées sont la pente, le dévers, la direction et l'adhérence. Chacune de ces composantes est représentée par un bloc et un diagramme d'états-transitions. Lorsqu'une modification est faite pour l'une des composantes, un signal est transmis vers le bloc de la colonne (qui gère également les pas) afin que le pas suivant prenne en compte les modifications. On peut constater que les valeurs qui représentent les composantes de la route sont abstraites. En effet, en ce qui concerne l'adhérence par exemple, elle peut prendre trois valeurs : normal, pluie ou glace. Les calculs de la réaction du véhicule en fonction de l'adhérence sont trop complexes pour être spécifiés dans un modèle SysML4MBT. Ainsi, des valeurs abstraites sont spécifiées et la correspondance avec les données réelles sera définie au moment de la concrétisation des tests. Le cahier des charges de cet applicatif et sa modélisation complète en SysML4MBT sont décrits dans le livrable L4.1 du projet VETESS.

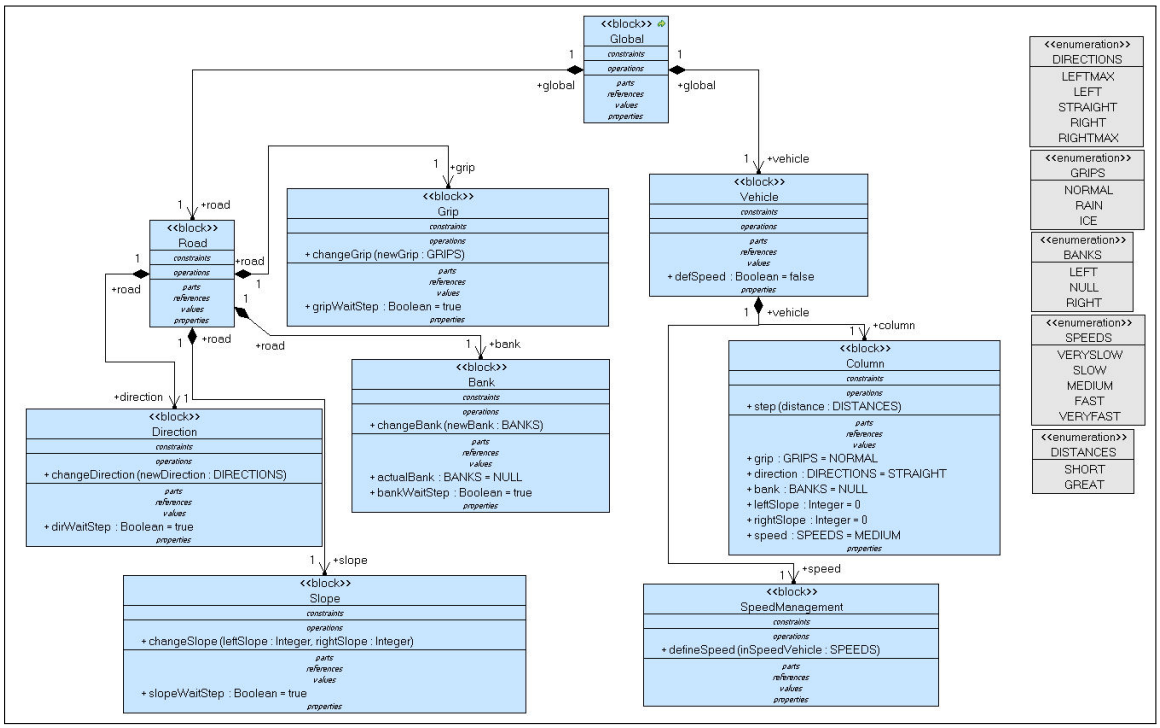


FIGURE 6.15 – Diagramme de bloc de la colonne de direction

6.4.2 Génération de tests sur le modèle SysML

Pour ce cas d'étude, 36 tests sont générés à l'aide de la stratégie ComCover contre 25 pour la stratégie de couverture de toutes les transitions (appuyé par D/CC). 11 nouveaux tests sont donc obtenus par l'application de la stratégie ComCover ce qui permet une meilleure couverture du système sous test. 7 (parmi les 45) couples d'envois/réceptions de signaux ne sont pas cohérents et ne peuvent ainsi pas être couverts. Le niveau d'abstraction du modèle influence directement le nombre de tests générés. En effet, plus le modèle est concret, plus il contient de données et de communications et de ce fait, plus le nombre de tests générés est potentiellement grand. Le niveau d'abstraction choisi a permis d'obtenir un ensemble de tests relevant car représentant des situations variées et réalistes.

6.4.3 Génération de tests Test Designer^{TM}

Contrairement au cas des essuie-glace, le modèle UML4MBT issu de la transformation du modèle SysML4MBT n'a pas posé de problème de limitation. Le modèle obtenu suite à cette transformation contient 315 transitions. L'outil Test Designer^{TM} a engendré la génération de 154 tests. A titre indicatif, la génération a nécessité 6 heures et 13 minutes de génération due à la complexité des cibles et à la longueur des tests.

6.4.4 Publication et exécution des tests

Deux publishers ont été développés pour ce cas. En plus du publisher permettant d'exporter les tests vers l'outil TestInView en vue de leur exécution, un publisher permettant de visualiser les tracés de route représentés par les tests générés par Test Designer^{TM} a été mis en place.

Publisher graphique

Ce publisher permet de donner une représentation graphique d'un test. Par exemple, la représentation graphique visible figure 6.16 correspond au test représenté dans le tableau 6.4.

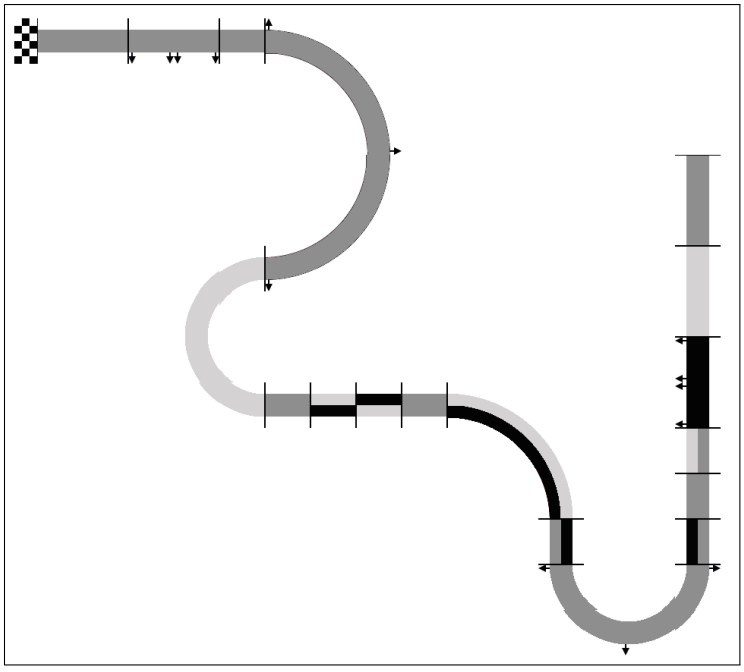


FIGURE 6.16 – Représentation graphique du test du tableau 6.4

Les traits noirs perpendiculaires séparent les différents pas. Les parties grises représentent une partie plate, les parties grises claires représentent les parties en descente et les parties noires, les sections en montée. Enfin, les flèches représentent le dévers. Cette représentation graphique est générée automatiquement par un publisher HTML à partir des tests produits par Test Designer^{TM}. Cela permet de représenter visuellement les tests générés afin de vérifier la cohérence des tests avant l'exécution.

L'autre publisher mis en place pour ce cas d'étude permet l'export des séquences de test dans le format d'entrée de la plateforme d'exécution TestInView.

1	step(GREAT)	11	changeDirection(STRAIGHT) changeSlope(2,0) step(SHORT)
2	changeBank(RIGHT) step(GREAT)	12	changeDirection(LEFTMAX) changeBank(RIGHT) changeSlope(0,0) step(GREAT)
3	changeBank(NULL) step(SHORT)	13	changeDirection(STRAIGHT) changeBank(NULL) changeSlope(2,0) step(SHORT)
4	changeDirection(RIGHT) changeBank(LEFT) step(GREAT)	14	changeSlope(0,0) step(SHORT)
5	changeDirection(LEFTMAX) changeBank(NULL) changeSlope(-1,-1) step(GREAT)	15	changeSlope(-2,0) step(SHORT)
6	changeDirection(STRAIGHT) changeSlope(0,0) step(SHORT)	16	changeBank(LEFT) changeSlope(1,1) step(GREAT)
7	changeSlope(-2,2) step(SHORT)	17	changeBank(NULL) changeSlope(-1,-1) step(GREAT)
8	changeSlope(2,-2) step(SHORT)	18	changeSlope(0,0) step(GREAT)
9	changeSlope(0,0) step(SHORT)		
10	changeDirection(RIGHT) changeSlope(-1,1) step(SHORT)		

TABLE 6.4 – Exemple de test généré pour la colonne de direction

Publication TestInView

Comme pour le cas des essuie-glace, afin de pouvoir exécuter les tests sur le système réel, il faut donner la correspondance entre les données abstraites du modèle et les données concrètes du système. Les valeurs de concrétisation sont représentées dans le tableau 6.5.

Ces valeurs permettent de calculer le tracé concret de la route. Les calculs sont réalisés à l'aide d'un programme Matlab. Ce dernier permet de calculer le profil de la route et la position du véhicule (et ses caractéristiques) en fonction du temps. Par exemple, le code représenté figure 6.17 permet de calculer la pente et le dévers concrets de la route avec comme variable :

- `Xtraj` : la trajectoire du centre de gravité du véhicule suivant l'axe X,
- `Ytraj` : la trajectoire du centre de gravité du véhicule suivant l'axe Y,
- `ZtrajL` : la hauteur de la roue gauche,
- `ZtrajR` : la hauteur de la roue droite,
- `Course` : le cap de la route.

DISTANCE	GREAT	20 m
	SHORT	0.5 m
DIRECTION	MAXLEFT	0.06
	LEFT	0.05
	STRAIGHT	0
	RIGHT	-0.05
	MAXRIGHT	-0.06
BANK	NULL	0 rad
	LEFT	+0.08 rad
	RIGHT	-0.08 rad
SLOPE	2	0.35 rad
	1	0.1745 rad
	0	0 rad
	-1	-0.1745 rad
	-2	-0.35 rad

TABLE 6.5 – Valeurs de concrétisation pour la colonne de direction

```
%Slope calculation
slope = (leftSlope+rightSlope)/2;
Xtraj(indexPoint,1)=Xtraj(indexPoint-1,1)+r(indexPoint,1)*
                            sin(pi/2-slope)*cos(course(indexPoint,1));
Ytraj(indexPoint,1)=Ytraj(indexPoint-1,1)+r(indexPoint,1)*
                            sin(pi/2-slope)*sin(course(indexPoint,1));

%Bank calculation
Ztraj(indexPoint,1)=Ztraj(indexPoint-1,1)+r(indexPoint,1)*
                            cos(pi/2-slope);
ZtrajL(indexPoint,1)=Ztraj(indexPoint,1)+r(indexPoint,1)*
                            cos(pi/2-leftSlope)+b/2*sin(bank);
ZtrajR(indexPoint,1)=Ztraj(indexPoint,1)+r(indexPoint,1)*
                            cos(pi/2-rightSlope)-b/2*sin(bank);
```

FIGURE 6.17 – Calculs concrets de la pente et du dévers pour la colonne de direction

En utilisant ces calculs, on obtient le tracé concret de la route et le déplacement du véhicule à effectuer en fonction du temps. Cet élément est intitulé un vecteur (cf figure 6.18).

Calcul des valeurs attendues

Du fait de la complexité des calculs, bien que les données aient été concrétisées, les données attendues aux points d'observation ne sont pas connues et doivent être calculées. Un modèle Simulink est utilisé afin de calculer les réactions du véhicule en fonction du tracé. Ces valeurs théoriques sont ensuite comparées aux valeurs issues de l'exécution sur le banc de test.

Time	Xtraj	Ytraj	Course
0.000000	0.000000	0.000120	0.000120
0.010000	0.000094	0.000183	0.000183
0.020000	0.000282	0.000257	0.000257
0.030000	0.000564	0.000343	0.000343
0.040000	0.000939	0.000440	0.000440
0.050000	0.001409	0.000629	0.000629
0.060000	0.001973	0.000852	0.000852
0.070000	0.002630	0.001109	0.001109
0.080000	0.003382	0.001400	0.001400
0.090000	0.004227	0.001726	0.001726
0.100000	0.005167	0.002086	0.002086
0.110000	0.006200	0.002480	0.002480

FIGURE 6.18 – Exemple de vecteur pour le cas de la colonne de direction

Exécution des tests

Comme pour le cas d'étude concernant les essuie-glace, un modèle de simulation permet d'exécuter les tests sur une version simulée du système. La partie graphique a été réalisée en OpenGL et les calculs sont réalisés à l'aide d'un modèle Simulink. Le module de simulation est représenté sur la figure 6.19.

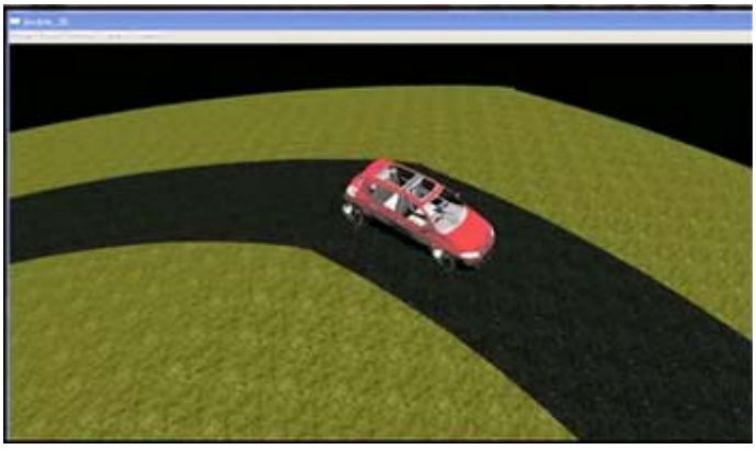


FIGURE 6.19 – Version simulée de la colonne de direction

Ce simulateur détermine les valeurs attendues mais permet également de valider le modèle Simulink utilisé. En effet, si l'exécution sur le simulateur engendre des erreurs, cela signifie soit que le test est faux (c'est-à-dire qu'il ne peut pas être joué) soit que le modèle ne correspond pas exactement au système réel car il n'arrive pas à exécuter une séquence pouvant être théoriquement exécutée sur le système réel.

Le banc de test représenté figure 6.20 intègre tous les actionneurs nécessaires à la simulation de la rotation de la colonne de direction ainsi qu'à la simulation des comportements verticaux des suspensions (pompage) en réaction au profil de la route et en réaction au mouvement de la caisse autour de l'axe longitudinal du véhicule (roulis). Des capteurs permettent de capturer la réaction du véhicule en temps-réel.

FIGURE 6.20 – Banc de test physique de la colonne de direction

Les tests sont exécutés en parallèle sur le version simulée et sur le banc de test réel. Les valeurs issues de l'exécution des tests sur le banc de tests sont comparées aux valeurs issues de la version simulée en temps réel. Une illustration de cette comparaison au niveau du roulis est visible sur la figure 6.21. Il s'agit des relevés de l'exécution du test présenté précédemment dans le tableau 6.4.

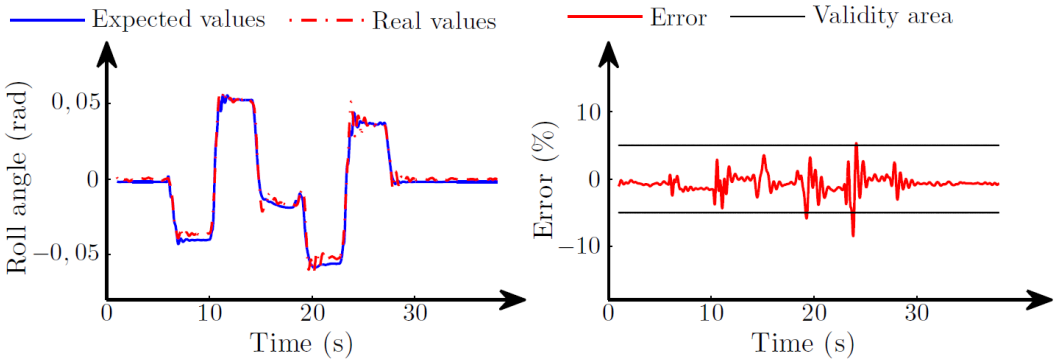


FIGURE 6.21 – Comparaison entre les valeurs attendues et les valeurs obtenues au niveau du roulis

La partie de gauche de la figure 6.21 montre les différences entre les valeurs attendues et les valeurs obtenues. La partie de droite montre le pourcentage d'erreurs au cours du temps. Les deux repères horizontaux représentent la marge de 5% d'erreur qui a été fixée.

On peut donc remarquer que pour ce test là, une différence dépassant la marge d'erreur a été détectée aux alentours de 24 secondes. Deux raisons peuvent être à l'origine de ce problème : soit le modèle Simulink comporte des erreurs, soit le système a un défaut. Dans ce second cas, le problème peut venir d'un problème de conception du système ou, comme c'est le cas en l'occurrence, d'un problème de réglage au niveau du banc de tests.

6.4.5 Bilan

Ce cas d'étude montre l'adaptabilité de la chaîne outillée aux systèmes continus. Les étapes allant de la modélisation à la génération de tests sont identiques quel que soit le type de système (discret ou continu). Une fois les tests générés, dans le cas d'un système continu, la chronologie détaillée des événements est définie. L'exécution des tests sur un banc de test réel permet la comparaison des réactions du système avec les réactions d'une version simulée de celui-ci afin de vérifier son bon fonctionnement. La version simulée est construite à l'aide de modèles Matlab/Simulink et fournit les valeurs attendues.

6.5 Synthèse

Cette partie a montré que la chaîne outillée VETESS s'adapte au test de systèmes embarqués de type discret et continu. Cette capacité s'appuie sur l'exploitation de la dualité Système/Environnement du système sous test. En effet, les tests sont générés sur la base d'un modèle SysML4MBT qui prend en compte à la fois les composantes du système sous test et celles de son environnement. Si le calcul des données de test s'appuie sur la même technologie pour les systèmes de nature discrète ou continue, l'établissement du verdict s'appuie, en revanche, sur une démarche dédiée selon cette nature.

S'il s'agit d'un système de type discret, les données de test et les valeurs attendues associées sont calculées par Test Designer^{TM} sur la base des comportements du système sous test et de son environnement décrits dans un modèle SysML4MBT. Dans le contexte particulier de systèmes de type continu, Test Designer^{TM} génère les cas de test qui prennent la forme de séquences d'envoi de stimuli de la part de l'environnement sans faire état du comportement attendu de la part du système sous test. Ces résultats attendus sont produits a posteriori au moyen d'un simulateur dédié (typiquement de type Matlab/Simulink) permettant une simulation mixte discrète/continue.

Le tableau 6.6 récapitule les données et les résultats de l'application de la chaîne outillée sur les différents cas d'étude présentés dans ce mémoire.

		Arrêt d'urgence d'un train	Fil rouge	Réglages de siège	Essuie-glace	Colonne de direction
SysMLAMBT	Blocs	4	4	9	15	9
	c / e / r	1/2/2	8/14/10	24/42/48 (4/7/8)	26/58/65	10/25/20
	SM	3	3	8	12	6
	Etats	[2,2,2]	[2,5,5]	[8,1,3,3, 3,3,3,3]	[1,1,1,1,1,2, 17,10,2,2,2,2]	[2,4,3,6,3,4]
	Transitions	[2,2,2]	[2,8,8]	[18,1,8,8, 8,8,8,8]	[2,3,2,4,1,3, 52,16,2,2,2,2]	[3,8,4,5,9,8]
Transitions + D/CC	Cibles	6	21	79 (15)	103	61
	Tests	4	5	36 (7)	85	25
ComCover	Cibles	4	20	72 (12)	130	45
	Min / Max	3/4	16/28	66/498 (11/23)	97/1345	35/185
	Tests	5	6	42 (8)	104	36
	Inaccessibles	0	4	0 (0)	10	7
	Bénéfice	1	1	6 (1)	19	11
UMLAMBT	Classes		9		29 (4)	16
	Objets		13		57 (4)	25
	Etats		50		2526 (11)	120
	Transitions		246		31873 (31)	315
	Tests générés				(113)	154
	Temps de génération				(4 min 23)	6 h 13

TABLE 6.6 – Résultats au niveau des différents cas d'étude

Les 5 premières lignes (notées *SysML4MBT*) représentent les quantités de chaque entité contenue dans les modèles SysML4MBT pour chacun des cas d'étude. La ligne intitulée *c/e/r* représente respectivement le nombre de connecteurs unitaires, d'envois et de réceptions modélisés (données nécessaire au calcul des métriques présentées en section 4.5.4). Les lignes *Etats* et *Transitions* détaillent pour chaque diagramme d'états-transitions du modèle, le nombre d'états et de transitions. Pour le cas des réglages de siège, les éléments entre parenthèses représentent les données en prennant en compte uniquement le moteur `RH` (données présentées section 6.2). Les modèles issus des cas d'étude sont de taille variée et permettent ainsi une certaine représentativité des systèmes embarqués.

Ensuite, les lignes *Transitions+D/CC* correspondent aux nombres de cibles et de tests générés pour assurer la couverture des critères *Toutes les transitions* et D/CC. Les lignes *ComCover* représentent les données résultantes de l'application du critère ComCover sur le modèle SysML. La ligne *Min/Max* représente les bornes encadrant le nombre de cibles générés. La ligne *Inaccessibles* représente quant à elle le nombre de cibles qui ne peuvent pas être couvertes. Finalement, la ligne *Bénéfice* représente le nombre de tests qui sont générés par ComCover et absents de la génération de tests initiale. Nous pouvons ainsi remarquer que le nombre de cibles générés par ComCover (ligne *ComCover : cibles*) respecte bien les métriques (ligne *Min/Max*) présentées section 4.5.4. De plus, cette partie du tableau permet de montrer que la stratégie ComCover apporte de nouveaux tests (ligne *Bénéfice*) et ainsi une meilleure couverture du système que quel que soit le cas d'étude. Toutefois, cette stratégie apporte également quelque cibles de tests inaccessibles (ligne *Inaccessibles*). Nous pouvons également observer une meilleure performance de la stratégie ComCover qui, par rapport à la stratégie initiale, permet la génération d'un nombre de tests supérieur à partir d'un nombre de cible inférieur.

La section UML4MBT du tableau représente les données du modèle issu de la transformation SysML4MBT vers UML4MBT. Seul le cas des essuie-glace a posé problème en dépassant la taille limite de modèles (en nombre de comportements représentés) acceptés par le générateur de tests Test Designer^{TM}. Les autres cas d'étude ne présentent pas ce défaut. Cela est dû à un nombre important de diagrammes d'états-transitions concurrents. L'amélioration de l'efficacité de la fusion des diagrammes d'états-transitions pourrait apporter une solution à ce problème. Pour ce cas d'étude, un modèle UML4MBT représentant une simplification du modèle UML4MBT généré a donc été construit manuellement afin de pouvoir expérimenter la suite de la chaîne outillée pour les systèmes discrets. Les données concernant ce modèle sont représentées entre parenthèses dans le tableau.

Les 2 dernières lignes représentent les données de génération de tests à l'aide de l'outil Test Designer^{TM}. On remarque alors un temps de génération nettement supérieur dans le cas de la colonne de direction, dû à la complexité et à la longueur des tests. Une solution pour diminuer ce temps peut consister à réaliser un modèle plus abstrait pour réduire le nombre de tests générés. Mais dans ce cas, la couverture du système sous test sera moindre.

Ce chapitre termine la présentation de nos contributions. Dans la partie suivante nous concluons notre document et proposons des perspectives à ce travail à plusieurs niveaux.

Conclusions et perspectives

Conclusion

Ce mémoire de thèse présente une approche originale de validation des systèmes embraqués dans les véhicules par la génération de tests à partir de modèles. La définition d'un langage de modélisation adapté à de tels systèmes et d'une stratégie de génération de tests adéquate sont les éléments clés de cette démarche. L'expérimentation de ce processus a été réalisée par la mise en place d'une chaîne outillée, automatique de bout en bout. Cette chaîne outillée originale permet une utilisation collaborative et automatique de plusieurs outils performants, couramment utilisés dans les domaines de la validation et des systèmes embarqués. Sa force se situe au niveau de sa compatibilité avec tous les types de systèmes embarqués, aussi bien discrets que continus. La possible connexion avec un simulateur ou avec un banc de test physique en fait un outil complet et opérationnel. La chaîne outillée obtenue est représentée sur la figure 6.22.

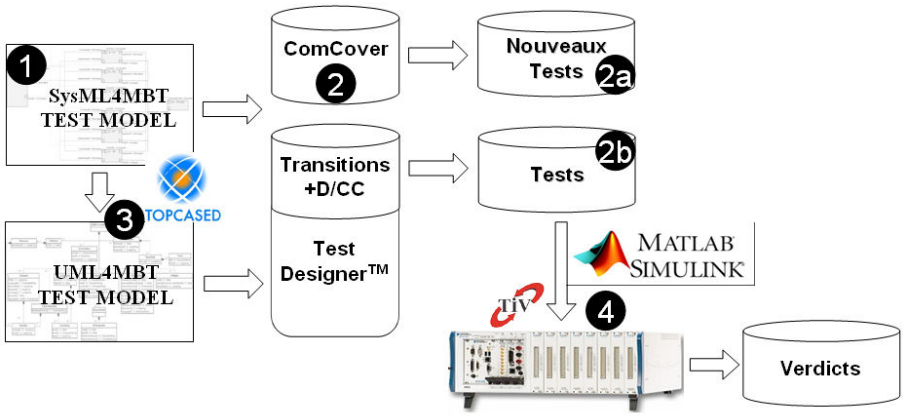


FIGURE 6.22 – Chaîne outillée issue de nos travaux

La première étape consiste à modéliser le système à l'aide du langage SysML permettant de représenter une vue globale du système sous test, de son environnement et de leurs comportements respectifs (item 1 de la figure 6.22). Les éléments de SysML pris en compte dans notre démarche sont rassemblés au sein de SysML4MBT (chapitre 3). Une sémantique opérationnelle et des règles précises ont été définies afin de guider la réalisation des modèles. Le modèle obtenu représente ainsi les stimuli et les comportements prévus du système sous test plutôt que leur gestion concrète exprimée dans le code source.

Un critère de couverture structurelle prenant en compte les spécificités des modèles SysML4MBT a été défini (chapitre 4). Ce critère est basé sur la couverture des communications. La stratégie de génération de tests, baptisée ComCover, est ensuite appliquée sur le modèle réalisé permettant de garantir ce critère (item 2 de la figure 6.22).

Ce processus a l'avantage de proposer une génération de tests reproductible, indépendante des caractéristiques et des contraintes d'implantation que le code source pourrait présenter. De plus, il n'est pas nécessaire d'avoir au préalable ni la connaissance technique d'un langage ou d'un environnement de programmation spécifique, ni du code source à tester. Cette démarche répond directement à la problématique de l'obtention d'un processus accessible aux ingénieurs validation. L'application de la stratégie ComCover sur le modèle SysML4MBT permet d'obtenir de nouveaux tests (item 2a) par rapport à l'application du critère de couverture de *Toutes les transitions* appuyé par D/CC (item 2b), issu des travaux sur lesquels nous nous sommes basés.

L'implémentation de cette démarche a permis son expérimentation. Les développements ont été réalisés dans une optique de preuve de concept, en intégrant et en complétant des outils existants (chapitre 5). L'articulation de ces outils au sein d'une chaîne outillée permettent d'automatiser un processus aujourd'hui traditionnellement discontinu. L'utilisation du modeleur open-source Topcased et de l'outil de génération de tests Test Designer^{TM} permettent de proposer une chaîne outillée aboutie et opérationnelle. Un plugin Topcased a été développé (item 3 de la figure 6.22) permettant la transformation des modèles SysML4MBT vers UML4MBT (langage d'entrée de Test Designer^{TM}). La surcharge de cette transformation permet d'assurer l'application de la couverture proposée par ComCover en réutilisant la stratégie de génération de tests définie dans l'outil Test Designer^{TM}. Les phases de concrétisation (propres à chaque cas d'étude) des tests est suivie par l'exécution des tests assurée par l'outil TestInView (item 4 de la figure 6.22).

Cette chaîne a été expérimentée sur les trois cas d'étude présentés dans ce mémoire (chapitre 6). Leur diversité permet de constater l'adaptabilité de notre approche à différents types de systèmes embarqués.

Ces travaux et les résultats concluants obtenus permettent d'ouvrir de nouveaux champs de recherche et d'application. Ces perspectives sont présentées dans la section suivante.

Perspectives

Les perspectives à ces travaux se déclinent selon trois directions : l'amélioration du passage à l'échelle, l'accroissement de l'ensemble d'éléments de modélisation pris en compte, et finalement, la définition de nouvelles stratégies de génération de tests permettant une meilleure couverture du système sous test. Les impacts de ces perspectives sur la chaîne outillée courante sont illustrés sur la figure 6.23.

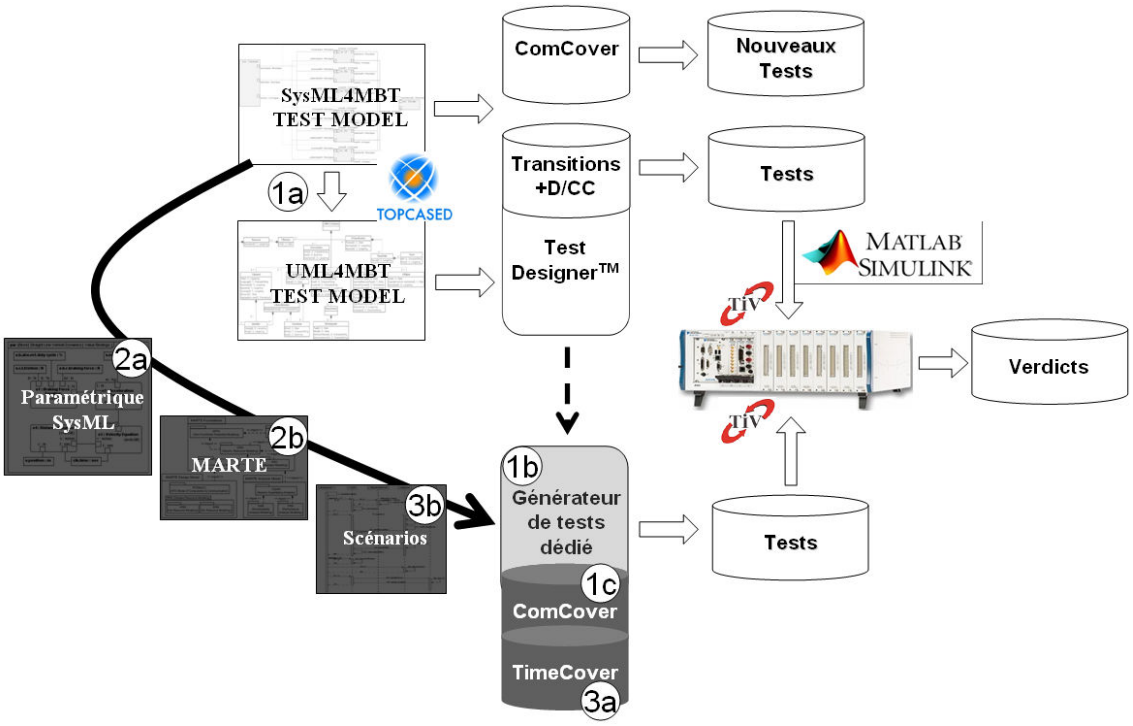


FIGURE 6.23 – Perspectives de nos travaux

En analysant les résultats présentés dans ce mémoire, une perspective consiste à améliorer le passage à l'échelle mis en lumière par le cas d'étude portant sur les essuie-glace (section 6.3). En effet, la mise en œuvre de notre chaîne outillée a été réalisée par l'utilisation de l'outil de génération de tests Test Designer^{TM}. De ce fait, une étape de transformation des modèles SysML4MBT en modèles UML4MBT a été définie. Dans certains cas, le modèle résultant est trop important en terme de nombre de comportements représentés pour être pris en compte par l'outil de génération de tests Test Designer^{TM}. Deux solutions sont envisagées afin de s'éloigner de ces limitations. La première consiste a diminuer la taille du modèle issu de la transformation. La principale cause de cette taille se situe au niveau de l'algorithme chargé de la fusion des diagrammes d'états-transitions concurrents. La première solution consiste à améliorer l'algorithme mis en œuvre, en in-

troduisant par exemple des états composites permettant de fusionner certaines transitions (item 1a de la figure 6.23). Cette solution n'est cependant qu'une solution d'appoint permettant de déplacer légèrement le problème de passage à l'échelle. Une seconde approche plus prometteuse pour résoudre ce problème concerne la prise en compte du parallélisme directement au sein du moteur de génération de tests (item 1b). Cela consisterait en l'implémentation, au sein de cet outil, des routines nécessaires à la manipulation des diagrammes SysML4MBT permettant ainsi l'application directe de la stratégie ComCover sur ces modèles (item 1c).

Nous avons vu que la notion de temps est importante pour les systèmes embarqués. Une autre amélioration potentielle s'intéresse donc au fait que SysML4MBT ne permette pas la représentation des équations qui régissent l'évolution des systèmes temps-réels. De même, aucune entité SysML4MBT dédiée ne permet de représenter les contraintes temporelles. Comme montré dans le cas d'étude du contrôle de siège (section 6.2), il est possible de discrétiser le temps qui s'écoule à l'aide d'opérations qui représentent des tops d'horloge par exemple. Cette représentation est cependant limitée aux contraintes temporelles simples. Cela empêche la prise en compte complète des systèmes continus. Comme illustré par le cas d'étude de la colonne de direction (section 6.4), la prise en compte du temps est réalisée en aval de la génération de tests lors de la phase de concrétisation des tests abstraits générés. La réalisation d'un modèle Matlab/Simulink permet le calcul des séquences temps réels équivalentes aux tests discrets générés initialement. Cette étape permet de calculer également les valeurs attendues en réalisant une animation du système continu.

Une possibilité pouvant permettre d'éviter la création du modèle Matlab/Simulink consiste à représenter les données contenues dans ce modèle directement au sein du modèle comportemental SysML4MBT. Cette solution permettrait de rendre le processus plus homogène, le modèle mathématique étant réalisé en même temps que le modèle de test et la phase de lien entre les deux modèles étant ainsi automatique. Au sein de SysML, le diagramme paramétrique peut être une solution à cette problématique (item 2a de la figure 6.23). Bien qu'il ne propose pas une expressivité aussi complète que Simulink, il pourrait être suffisant. Une autre alternative consiste à utiliser le profil UML MARTE dédié à la représentation des systèmes embarqués temps réels et qui propose toutes les entités nécessaires à la représentation des contraintes temporelles complexes (item 2b de la figure 6.23). SysML et MARTE sont réellement complémentaires et plusieurs stratégies peuvent être utilisées pour les combiner efficacement [ECSG09].

La mise en place de ces éléments de modélisation permettrait d'uniformiser les phases de modélisation et ainsi d'éviter la multiplication des plateformes utilisées. Une perspective à cette amélioration consisterait alors en la prise en compte de ces éléments au sein du générateur de tests. Dans le cas de modèles contenant suffisamment d'informations sur les contraintes temporelles du système, de nouvelles stratégies de génération de tests dédiées à ces éléments pourraient être mises en place (item 3a de la figure 6.23). La définition de scénarios, par l'utilisation des diagrammes de séquence par exemple, est également envisagée (item 3b). Cela permettrait d'exprimer des intentions de test engendrant ainsi un processus de génération de tests semi-automatique faisant appel aux compétences et à l'expertise des ingénieurs de validation.

Bibliographie

[26211] ISO 26262. Norme relative à la sécurité fonctionnelle des véhicules routiers, 2011.

[61510] CEI 61508. Norme relative à la sécurité fonctionnelle des systèmes électriques/électroniques/électroniques programmables, 2010.

[88089] ISO 8807, 1989. Information processing systems - open systems interconnection - LOTOS - a formal description technique based on the temporal ordering of observational behaviour.

[90005] ISO 9000. Norme relative aux systèmes de gestion de la qualité : principes essentiels et vocabulaire, 2005.

[90790] ISO 9074, 1990. Information processing systems - Open systems interconnection - Estelle - a formal description technique based on an extended state transition model.

[ABB+99] L. Arditi, A. Bouali, H. Boufaied, G. Clave, M. Hadj-Chaib, L. Leblanc, and R. de Simone. Using Esterel and formal methods to increase the confidence in the functional validation of a commercial DSP. In *Proc. of Floc'99, 4th ercim workshop of formal methods for industrial critical systems*, 1999.

[ABC+02] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. BZ-TT : A tool-set for test generation from Z and B using constraint logic programming. In *Proc. of the CONCUR'02 Workshop on Formal Approaches to Testing of Software (FATES'02)*, pages 105–120, Brno, Czech Republic, August 2002. INRIA Technical Report.

[Abr96] J-R. Abrial. *The B-BOOK : Assigning Programs to Meanings.* Cambridge University Press, 1996. ISBN 0-5214-9619-5.

[AdSSL+01] L. Apvrille, P. de Saqui-Sannes, C. Lohr, P. Sénac, and J. Courtiat. A new UML profile for real-time system formal design and validation. In M. Gogolla and C. Kobryn, editors, *UML 2001, The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in*

Computer Science, pages 287–301. Springer Berlin / Heidelberg, 2001. ISBN 978-3-5404-2667-7.

[AHD11] D. Aceituna and S.W. Lee H. Do. Interactive requirements validation for reactive systems through virtual requirements prototype. In *Proc. of the 1st Model-Driven Requirements Engineering Workshop (MoDRE)*, Trento, Italy, 2011. IEEE Computer Society Press.

[ALL09] M. Asztalos, L. Lengyel, and T. Levendovszky. A formalism for describing modeling transformations for verification. In *Proc. of the* 6^{th} *Int. Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa'09)*, pages 1–10, Denver, USA, October 2009. ACM. ISBN 978-1-6055-8876-6.

[Alu11] R. Alur. Formal verification of hybrid systems. In *Proc. of the* 11^{th} *Int. Conf. on Embedded Software (EMSOFT 2011)*, pages 273–278, Taipei, Taiwan, October 2011. ACM.

[AMS06] M. Auguston, J.B. Michael, and M.T. Shing. Environment behavior models for automation of testing and assessment of system safety. *Information and Software Technology*, 48(10) :971–980, 2006. ISSN 0950-5849.

[Ash01] P.J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, USA, 2^{nd} edition, 2001. ISBN 1-5586-0674-2.

[ATF09] W. Afzal, R. Torkar, and R. Feldt. A systematic review of search-based testing for non-functional system properties. *Information and Software Technologies*, 51(6) :957–976, June 2009. ISSN 0950-5849.

[BBC+06] E. Bernard, F. Bouquet, A. Charbonnier, B. Legeard, F. Peureux, M. Utting, and E. Torreborre. Model-based testing from UML models. In *Proc. of the Int. Workshop on Model-based Testing (MBT'2006)*, volume 94 of *LNCS*, pages 223–230, Dresden, Germany, October 2006. Springer Verlag.

[BBH02] M. Benattou, J.-M. Bruel, and N. Hameurlain. Generating test data from OCL specification. In *ECOOP'2002 Workshop on Integration and Transformation of UML Models (WITUML02)*, 2002.

[BBHP08] N. Belloir, J.M. Bruel, N. Hoang, and C. Pham. Utilisation de SysML pour la modélisation des réseaux de capteurs sans fil. In *Actes de la conférence Langages et Modèles à Objets (LMO'08)*, pages 2–7, Montréal, Canada, 2008.

[BBvB+01] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R.C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for agile software development, 2001.

[BF04] M. Bonfe and C. Fantuzzi. A practical approach to object-oriented modeling of logic control systems for industrial applications. In *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, volume 1, pages 980–985. IEEE Computer Society Press, 2004.

[BFG+03] C. Bigot, A. Faivre, J-P. Gallois, A. Lapitre, D. Lugato, J-Y. Pierron, and N. Rapin. Automatic Test Generation with AGATHA. In *Proc. of the Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 591–596, Warsaw, Poland, April 2003. Springer.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language : design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152, November 1992.

[BGL+07] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. A subset of precise UML for model-based testing. In *3rd int. Workshop on Advances in Model Based Testing (A-MOST'07)*, pages 95–104, London, United Kingdom, July 2007. ACM Press.

[BGLP08] F. Bouquet, C. Grandpierre, B. Legeard, and F. Peureux. A test generation solution to automate software testing. In *Proc. of the 3rd Int. Workshop on Automation of Software Test (AST'08)*, pages 45–48, Leipzig, Germany, May 2008. ACM Press.

[BK08] E. Bringmann and A. Kramer. Model-based testing of automotive systems. *Int. Conf. on Software Testing, Verification, and Validation*, pages 485–493, 2008.

[BLPT05] F. Bouquet, B. Legeard, F. Peureux, and E. Torreborre. Mastering test generation from smart card software formal models. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 70–85. Springer, 2005. ISBN 978-3-5402-4287-1.

[Boo93] G. Booch. *Object-Oriented Analysis and Design With Applications*. Benjamin/Cummings Publishing Company, 2nd edition, 1993. ISBN 0-8053-5340-2.

[BRlLV03] L. Burdy, A. Requet, J. l. Lanet, and La Vigie. Java applet correctness : a developer-oriented approach. In *In Proc. Formal Methods Europe*, pages 422–439. Springer, 2003.

[BRLS04] M. Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The spec# program-

ming system : An overview. volume 3362 of *LNCS*, pages 49–69. Springer, 2004.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proc. of Workshop on Logic of Programs*, volume 131 of *LNCS*, pages 52–71, Yorktown Heights, USA, May 1981. Springer Verlag.

[Cha08] D. Chappell. What is application lifecycle management ? *White Paper*, December 2008.

[CJRZ02] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG : a Symbolic Test Generation Tool. In *Proc. of the Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, Grenoble, France, April 2002. Springer Verlag.

[CK93] K.T. Cheng and A.S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proc. of the 30th Int. Design Automation Conf. (DAC'93)*, pages 86–91, Dallas, USA, June 1993. ACM Press.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J.A. Plaice. LUSTRE : a declarative language for real-time programming. In *Proc. of the 14th Symposium on Principles of programming Languages (POPL'87)*, pages 178–188, Munich, Germany, January 1987. ACM Press.

[CSE96] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using modelchecking. In *Proc. of the SPIN Int. Workshop*, Rutgers University, USA, August 1996.

[Dad06] F. Dadeau. *Évaluation symbolique à contraintes pour la validation - Application à Java/JML*. PhD thesis, LIFC, Université de Franche-Comté, July 2006.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proc. of the Int. Conf. on Formal Methods Europe (FME'93)*, volume 670 of *LNCS*, pages 268–284. Springer Verlag, April 1993.

[DGG09] E. Dustin, T. Garrett, and B. Gauf. *Implementing Automated Software Testing : How to Save Time and Lower Costs While Raising Quality*. Addison Wesley Professional, 2009. ISBN 0-3215-8051-6.

[DGG+12] A. Denise, M.C. Gaudel, S.D. Gouraud, R. Lassaigne, J. Oudinet, and S. Peyronnet. Coverage-biased random exploration of large models and application

to testing. *STTT, Int. Journal on Software Tools for Technology Transfer*, 14(1) :73–93, 2012.

[Dij70] E.W. Dijkstra. Notes on structured programming. Technical Report EWD249, Eindhoven University of Technology, 1970.

[DN84] J.W. Duran and S. Ntafos. An evaluation of random testing. *The journal IEEE Transactions on Software Engineering*, 10(4) :438–444, July 1984.

[DNT10] A.C. Dias-Neto and G.H. Travassos. A Picture from the Model-Based Testing area : concepts, techniques, and challenges. *Advances in Computers*, 80 :45–120, July 2010. ISSN 0065-2458.

[Dow97] M. Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2) :84, March 1997. ISSN 0163-5948.

[ECSG09] H. Espinoza, D. Cancila, B. Selic, and S. Gérard. Challenges in combining SysML and MARTE for model-based design of embedded systems. In *Proc. of the 5th European Conf. on Model Driven Architecture (ECMDA-FA'09)*, pages 98–113. Springer-Verlag, 2009. ISBN 978-3-6420-2673-7.

[EH00] D. Ehmanns and A. Hochstadter. Driver-model of lane change maneuvers. *7th World Congress on Intelligent Transportation Systems (ITS)*, November 2000.

[EHS97] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL : formal object-oriented language for communicating systems*. Prentice Hall, 1997.

[EPML07] D. Evrot, J-F. Pétin, G. Morel, and P. Lamy. Using SysML for identification and refinement of machinery safety properties. In IFAC, editor, *Proc. of IFAC Workshop on Dependable Control of Discretes Systems, DCDS'07*, Cachan, France, June 2007. Elsevier.

[Est07] J.A. Estefan. Survey of model-based systems engineering (mbse) methodologies. *Incose MBSE Focus Group*, 25, 2007.

[FGG07] A. Faivre, C. Gaston, and P. Le Gall. Symbolic model based testing for component oriented systems. In *Proc. of the 7th Int. Workshop on Formal Approaches to Testing of Software (FATES'07)*, volume 4581 of *LNCS*, pages 90–106, Tallinn, Estonia, June 2007. Springer.

[FMS09a] J.M. Faria, S. Mahomad, and N. Silva. Practical results from the application of model checking and test generation from UML/SysML model of on-board space applications. In *Proc. of the Int. Conf. on DAta Systems In Aerospace (DASIA'09)*, Istanbul, Turkey, May 2009. ESA Press.

[FMS09b] S. Friedenthal, A. Moore, and R. Steiner. *A Practical Guide to SysML : The Systems Modeling Language*. Morgan Kaufmann OMG Press, 2009. ISBN 978-0-1237-4379-4.

[FW88] P.G. Frankl and E.J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10) :1483–1498, 1988. ISSN 0098-5589.

[GB03] J. Ganssle and M. Barr. *Embedded Systems Dictionary*. Broché, 2003.

[GMSM02] S. Glaser, S. Mammar, and J. Sainte-Marie. Lateral driving assistance using embedded driver-vehicle-road model. In *Conf. on Engineering Systems Design and Analysis (ESDA)*, Istanbul, Turkey, July 2002.

[Gra08] C. Grandpierre. *Stratégies de génération automatique de tests à partir de modèles comportementaux UML/OCL*. PhD thesis, LIFC, Université de Franche-Comté, Besancon, 2008.

[Gro07] Object Management Group. UML profile for MARTE, draft revised submission. OMG document, OMG, April 2007.

[GRSC08] A. Gargantini, E. Riccobene, P. Scandurra, and A. Carioni. Scenario-based validation of embedded systems. In *Forum on specification and Design Languages (FDL '08)*, pages 191–196, Stuttgart, Germany, September 2008. IEEE Computer Society Press.

[Har87] D. Harel. Statecharts : a visual formalism for complex systems. *Science of Computer Programming*, 8(3) :231–274, 1987.

[Har98] D. Harel. *Modeling Reactive Systems With Statecharts*. Mac Graw Hill, 1998. ISBN 0-0702-6205-5.

[Har12] M. Harris. A shocking truth. *IEEE Spectrum*, 49(3) :30–58, 2012. ISSN 0018-9235.

[HLC+09] P.A Hsiung, S.W. Lin, Y.R. Chen, C.H. Huang, C. Shih, and W.C. Chu. Modeling and verification of real-time embedded systems with urgency. *Journal of Systems and Software*, 82(10) :1627–1641, 2009.

[HLM+08] A. Hessel, K.G. Larsen, M. Mikuèionis, B. Nielsen, P. Pettersson, and A. Skou. Testing real-time systems using UPPAAL. In *Formal Methods and Testing*, pages 77–117. Spring-Verlag, 2008. ISBN 3-5407-8916-2.

[Hol93] G. Holzmann. Design and validation of protocols. *Computer Network ISDN System*, 25(9) :981–1017, April 1993. ISSN 0169-7552.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5) :279–295, May 1997. ISSN 0098-5589.

[HSRH10] M. Hause, A. Stuart, D. Richards, and J. Holt. Testing safety critical systems with SysML/UML. In *Proc. of the 2010 15th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS '10)*, pages 325–330, Washington, USA, 2010. IEEE Computer Society Press. ISBN 978-0-7695-4015-3.

[HTES09] J.H. Hill, H.A. Turner, J.R. Edmondson, and D.C. Schmidt. Unit testing non-functional concerns of component-based distributed systems. *2009 Int. Conf. on Software Testing Verification and Validation (ICST)*, (1) :406–415, 2009.

[IAB10] M.Z. Iqbal, A. Arcuri, and L. Briand. Environment modeling with UML/MARTE to support black-box system testing for real-time embedded systems. In *Model Driven Engineering Languages and Systems*, volume 6394 of *LNCS*, pages 286–300. Springer, 2010. ISBN 978-3-6421-6144-5.

[IAB11] M.Z. Iqbal, A. Arcuri, and L. Briand. Automated system testing of real-time embedded systems based on environment models. Technical Report 2011-19, Simula Research Laboratory, 2011.

[Jac92] I. Jacobson. *Object Oriented Software Engineering : A Use Case Driven Approach*. Addison-Wesley, 1992. ISBN 0-2015-4435-0.

[JJ05] C. Jard and T. Jeron. TGV : theory, principles and algorithms : A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. Journal on Software Tools Technology Transfer (STTT)*, 7(4) :297–315, August 2005. ISSN 1433-2779.

[JK06] F. Jouault and I. Kurtev. *Transforming Models with ATL*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer Berlin / Heidelberg, 2006.

[JM99] T. Jéron and P. Morel. Test generation derived from model-checking. In *Proc. of the 11th Int. Conf. on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 108–121, Trento, Italy, July 1999. Springer Verlag.

[Jon90] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 2nd edition, 1990. ISBN 0-1388-0733-7.

[KEP05] D. Karlson, P. Eles, and Z. Peng. Validation of embedded systems using formal method aided simulation. In *Proc. of the 8th Euromicro Conf. on Digital System Design (DSD '05)*, pages 196–201, Washington, USA, 2005. IEEE Computer Society Press. ISBN 0-7695-2433-8.

[KT09] M. Krichen and S. Tripakis. Conformance testing for real-time systems. *Formal Methods System Design*, 34 :238–304, June 2009. ISSN 0925-9856.

[Lan98] G. Le Lann. Proof-based system engineering and embedded systems. In G. Rozenberg and F. Vaandrager, editors, *Lectures on Embedded Systems*, volume 1494 of *Lecture Notes in Computer Science*, pages 208–248. Springer Berlin / Heidelberg, 1998. ISBN 978-3-5406-5193-2.

[LBR06] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML : a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3) :1–38, May 2006. ISSN 0163-5948.

[LBV+04] D. Lugato, C. Bigot, Y. Valot, J-P. Gallois, S. Gérard, and F. Terrier. Validation and automatic test generation on UML models : the AGATHA approach. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 5 :124–139, March 2004. ISSN : 1433-2779.

[MA00] B. Marre and A. Arnould. Test sequences generation from LUSTRE descriptions : GATEL. In *Proc. of the 15th IEEE Int. Conf. on Automated software engineering (ASE '00)*, page 229, Washington, USA, 2000. IEEE Computer Society Press. ISBN 0-7695-0710-7.

[MBKL10] W. Muller, A. Bol, A. Krupp, and O. Lundkvist. Generation of executable testbenches from natural language requirement specifications for embedded real-time systems. In *Distributed, Parallel and Biologically Inspired Systems*, volume 329 of *IFIP*, pages 78–89, Brisbane, Australia, September 2010. Springer. ISBN 978-3-6421-5233-7.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall PTR, 2nd edition, 1997. ISBN 0-1362-9155-4.

[MG03] P-A. Muller and N. Gaertner. *Modélisation objet avec UML*, volume 514. Eyrolles, 2nd edition, 2003. ISBN 2-2121-1397-8.

[Mil99] R. Milner. *Communicating and Mobile Processes : the π-calculus*. Cambridge University Press, 1999. ISBN 0-5216-5869-1.

[MPmU04] C. Marche, C. Paulin-mohring, and X. Urbain. The Krakatoa tool for certification of java/javacard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1-2) :89–106, 2004.

[MRS+97] J.R. Moonen, J.M.T Romijn, O. Sies, J.G. Springintveld, L.G.M Feijs, and R.L.C. Koymans. A two-level approach to automated conformance testing of VHDL designs. Technical report, Amsterdam, The Netherlands, 1997.

[MSBT04] G.J. Myers, C. Sandler, T. Badgett, and T.M. Thomas. *The Art of Software Testing.* Wiley, 2^{nd} edition, 2004. ISBN 978-0-4714-6912-4.

[MSGL02] G. Martin, S. Swan, T. Grötker, and S. Liao. *System Design with SystemC.* Springer, 2002. ISBN 1-4020-7072-1.

[Nus97] B. Nuseibeh. Ariane 5 : Who dunnit ? *IEEE Software*, 14(3) :15–16, May 1997. ISSN 0740-7459.

[OB88] T.J. Ostrand and M.J. Balcer. The category-partition method for specifying and generation functional test. *Proc. of the ACM Conf.*, 31(6) :676–686, June 1988. ISSN 0001-0782.

[Old04] J. Oldevik. *UMT : UML Model Transformation Tool.* SINTEF Information and Communication Technology, Oslo, Norway, March 2004.

[OMG10] The OMG web site. `http://www.omg.org/`, 2010.

[OXL99] A.J. Offut, Y. Xiong, and S. Liu. Criteria for generating specification-based tests. In *Proc. of the 5^{th} Int. Conf. on Engineering of Complex Computer Systems (ICECCS'99)*, pages 119–131, Las-Vegas, USA, October 1999. IEEE Computer Society Press.

[Pal03] S. Palnitkar. *Verilog HDL : a guide to digital design and synthesis.* Prentice Hall Press, Upper Saddle River, USA, 2^{nd} edition, 2003. ISBN 0-1304-4911-3.

[PEML10] J.F. Pétin, D. Evrot, G. Morel, and P. Lamy. Combining SysML and formal methods for safety requirements verification. In *22^{nd} Int. Conf. on Software and Systems Engineering and their Applications*, Paris, France, 2010.

[Peu02] F. Peureux. *Génération de tests aux limites à partir de spécifications B en programmation logique avec contraintes ensemblistes.* PhD thesis, LIFC, Université de Franche-Comté, 2002.

[PFAR02] M-A. Peraldi-Frati, C. André, and J-P. Rigault. UML et le paradigme synchrone : Application à la conception de contrôleurs embarqués. pages 71–89, Paris, France, March 2002. Teknea.

[Qan08] Qantas airbus a330 accident media conference. October 14th, 2008. `http://www.atsb.gov.au/newsroom/2008/release/2008_43.aspx`.

[QS82] J-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of the 5^{th} Int. Symposium on Programming*, volume 137 of *LNCS*, pages 337–351, Torino, Italy, April 1982. Springer Verlag.

[RBP+90] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Modeling Technique.* Prentice-Hall, 1990. ISBN 0-1362-9841-9.

[RJB05] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 2^{nd} edition, 2005. ISBN 0-3212-4562-8.

[RW85] S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4) :367–375, 1985. ISSN 0098-5589.

[SIAB11] A. Shaukat, M. Zohaib Iqbal, A. Arcuri, and L. Briand. A search-based OCL constraint solver for model-based test data generation. In *Proceedings of the 11^{th} Int. Conf. on Quality Software (QSIC 2011)*, pages 41–50. IEEE Computer Society Press, 2011. ISBN 978-0-7695-4468-7.

[Sma09] The Smartesting web site. `http://www.smartesting.com`, 2009.

[Spi92] J.M. Spivey. *The Z notation : A Reference Manual.* Prentice-Hall, 2^{nd} edition, 1992. ISBN 0-1397-8529-9.

[Tas02] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical Report 7007.011, May 2002.

[TB03] G.J. Tretmans and H. Brinksma. Torx : Automated model-based testing. In *1^{st} European Conf. on Model-Driven Software Engineering*, pages 31–43, Nuremberg, Germany, December 2003.

[Tis09] R. Tissot. *Contribution à la génération automatique de tests à partir de modèles et de schémas de test comme critères de sélection dynamiques.* PhD thesis, LIFC, Université de Franche-Comté, Decembre 2009.

[TMJL09] R.A. Thacker, C.J. Myers, K. Jones, and S.C. Little. A new verification method for embedded systems. In *Proc. of the 2009 IEEE Int. Conf. on Computer design (ICCD'09)*, pages 193–200, Lake Tahoe, USA, 2009. IEEE Computer Society Press. ISBN 978-1-4244-5029-9.

[Tom88] J.E. Tomayko. Computers in spaceflight : The NASA experience. *Contractor*, 2, 1988.

[TOP10] The Topcased web site. `http://www.topcased.org/`, 2010.

[Tre08] J. Tretmans. Model-Based Testing with Labelled Transition Systems. In *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 1–38. Springer Verlag, 2008. ISBN 3-5407-8916-2.

[UL06] M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach.* Elsevier Science, 2006. ISBN 0-1237-2501-1.

[UPL11] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 2011.

[vAJ03] L. van Aertryck and T. Jensen. UML-CASTING : Test synthesis from UML models using constraint resolution. In J.M. Jézéquel, editor, *Proc. Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2003)*. INRIA, 2003.

[VB01] S.A. Vilkomir and J.P. Bowen. Formalization of software testing criteria using the Z notation. In *Proc. of the 25th Int. Conf. on Computer Software and Applications (COMPSAC'01)*, Chicago, USA, October 2001. IEEE Computer Society Press.

[VCG+08] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-Based Testing of object-oriented reactive systems with spec explorer. In *Formal Methods and Testing*, volume 4949, pages 39–76. Springer-Verlag, 2008. ISBN 3-540-78916-2.

[VDC08] The embedded software market intelligence program, 2008.

[VET09] Projet VETESS. Définition des objectifs de test et des principes de convertures. Livrable Projet L1.2, Projet Pôle de Compétitivité (2008-10), July 2009.

[VET10a] The VETESS web site. `http://lifc.univ-fcomte.fr/vetess/`, 2010.

[VET10b] Projet VETESS. Principes d'établissement des verdicts. Livrable Projet L1.4, Projet Pôle de Compétitivité (2008-10), February 2010.

[vRT04] H.M. van der Bijl, A. Rensink, and G.J. Tretmans. Compositional testing with ioco. In *Formal Approaches to Software Testing (FATES)*, volume 2931 of *LNCS*, pages 86–100, Berlin, 2004. Springer Verlag.

[Wei10] S. Weißleder. Simulated satisfaction of coverage criteria on UML state machines. In *Proc. of the 3rd Int. Conf. on Software Testing, Verification and Validation (ICST'10)*, pages 117–126, Paris, France, April 2010. IEEE Computer Society Press.

[WK96] J. Warmer and A. Kleppe. *The Object Constraint Language : Precise Modeling with UML*. Addison-Wesley, 1996. ISBN 0-2013-7940-6.

[WZEK10] G. Weiss, M. Zeller, D. Eilers, and R. Knorr. Approach for iterative validation of automotive embedded systems. In *Proc. of the 3rd Int. Workshop on Model Based Architecting and Construction of Embedded Systems*, pages 65–83, Aachen, Germany, 2010.

[ZB09] H. Zhu and F. Belli. Advancing test automation technology to meet the challenges of model-based software testing. *Information and Software Technology*, 51(11) :1485–1486, november 2009. ISSN 0950-5849.

Résumé

Les travaux présentés dans ce mémoire proposent une méthode originale de génération automatique de tests à partir de modèles SysML pour la validation de systèmes embarqués. Inspiré de travaux sur la génération de tests à partir de modèles UML4MBT (sous-ensemble d'UML), un sous-ensemble du langage SysML (appelé SysML4MBT) regroupant les éléments de modélisation pris en compte dans notre approche a été défini. Ce sous-ensemble permet une représentation suffisante des comportements du système et de son environnement pour permettre une génération de tests pertinente de manière automatique.

Une stratégie de génération de tests dédiée (intitulée ComCover) a été définie afin d'exploiter au maximum les comportements modélisés et ainsi compléter la stratégie implantée initialement au sein de l'outil de génération de tests à partir de modèle UML4MBT. Cette stratégie, basée sur les principes du critère de couverture de modèles bien connu Def-Use, s'intéresse à la couverture des communications (envois / réceptions) au sein du système et entre le système et son environnement. La mise en œuvre opérationnelle d'un prototype a nécessité la définition de règles de réécriture permettant la transformation du modèle SysML4MBT vers le format d'entrée natif du générateur de tests (UML4MBT) tout en conservant l'expressivité de SysML4MBT. Afin d'obtenir, à l'aide de l'outil de génération de tests, la couverture définie par la stratégie ComCover, la transformation de modèles à été surchargée permettant ainsi la génération automatique de tests dédiés embarqué.

Finalement, les étapes de concrétisation des tests en scripts exécutables et l'établissement automatique du verdict lors de l'exécution sur banc de test définis durant le projet VETESS permettent l'établissement d'une chaîne outillée opérationnelle de génération et d'exécution automatique de tests à partir de spécifications SysML. Cette chaîne outillée a été étrennée sur plusieurs cas d'étude automobile tels que l'éclairage avant, les essuie-glaces ou la colonne de direction de véhicule. Sur ce dernier exemple, nous avons eu l'opportunité d'exécuter les tests sur un banc de test physique. Ces cas d'étude ont permis de valider chacune des étapes de l'approche proposée.

Mots-clés: Validation de systèmes embarqués, Génération de tests à partir de modèles, UML/SysML, Critères de couverture, Transformation de modèles, Chaîne outillée