TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

LISTE DES ALGORITHMES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

DSML	Domain Specific Modeling Langage
ÉTS	École de Technologie Supérieure
EFSM	Extended Finite State Machine
CEFSM	Communicating Extended Finite State Machine
IDM	Ingénierie Dirigée par les Modèles
OMG	Object Management Group
MDA	Model Driven Architecture
RTCA	Radio Technical Commission for Aeronautics
DAL	Design Assurance Level
HLR	High Level Requirement
LLR	Low Level Requirement
MOF	Meta Object Facility
OCL	Object Constraint Language
UML	Unified Modeling Language
STEF	State machine To EFsm
EMF	Eclipse Modeling Framework
QVT	Query/View/Transformation

INTRODUCTION

Contexte de recherche

L'ingénierie dirigée par les modèles (IDM) est une méthode d'ingénierie des systèmes utilisant des modèles pour représenter à la fois un problème posé et sa solution. Le modèle en question est une abstraction d'un élément réel représentant un point de vue particulier. La philosophie de l'IDM considère que "tout est modèle" (Bézivin, 2004), c'est à dire que n'importe quel élément de la réalité peut être représenté par un modèle. L'IDM s'applique à différents domaines : dans le domaine d'affaires ou des logiciels, et à différents niveaux : modèle d'analyse ou modèle de conception. En ce qui concerne le domaine du logiciel, l'IDM a pris de l'essor avec l'introduction par l'Object Management Group (OMG) de l'approche Model Driven Architecture (MDA) (OMG, 2000). L'OMG propose un ensemble de standards, permettant de spécifier des modèles depuis le niveau métier, jusqu'à la plateforme de développement. Cette démarche, introduite au début des années 2000, est largement répandue dans l'industrie des systèmes embarqués critiques (Huhn & Hungar, 2007).

En effet les industriels du développement de systèmes embarqués ont réalisé l'intérêt de l'intégration d'une démarche dirigée par les modèles, notamment en ce qui concerne les systèmes critiques et complexes. En effet, les modèles permettent de traiter plus simplement cette complexité par la représentation graphique. De plus, comme on attend de ces systèmes un fonctionnement fiable, il faut les tester de façon approfondie. Pour cela les modèles peuvent être utilisés pour simuler, tester ou encore générer du code. C'est dans ce contexte que des langages comme Simulink/Stateflow (Mat, 2019) et Scade (Ansys, 2018) ont vu le jour.

Pour ces systèmes critiques, des agences gouvernementales se chargent de certifier les produits avant qu'ils ne soient disponibles sur le marché. Pour cela ces agences fournissent les directives nécessaires pour l'obtention de la certification. Nous nous intéressons dans le contexte de notre projet aux systèmes de l'aéronautique civile. En ce qui concerne le domaine de l'aéronautique

civile, l'organisation Radio Technical Commission for Aeronautics (RTCA) propose le document de certification DO-178 (RTCA, 2011a) qui a été très largement adopté par les agences gouvernementales. Ce document fournit les objectifs que le produit logiciel doit satisfaire, ainsi que les documents requis pour la certification.

De façon un peu plus détaillée, la certification DO-178 propose un ensemble d'objectifs et de documents pour chaque étape du cycle de vie du logiciel. Ce cycle de vie débute lors de la spécification des exigences systèmes relatives au logiciel, jusqu'à l'assurance de la qualité de celui-ci. Les produits logiciels sont classés selon différents niveaux nommés Design Assurance Level (DAL), reflétant leur niveau de criticité. Les niveaux allant de E : "défaut logiciel sans impact la sécurité", à A : "défaut logiciel ayant un impact catastrophique sur la sécurité de l'avion (conduisant à la perte de l'appareil)".

Constatant l'intérêt grandissant des entreprises pour l'IDM, RTCA a proposé dans sa dernière révision de DO-178 (DO-178C), un supplément concernant l'utilisation de modèles : DO-331 (RTCA, 2011b). Celui-ci introduit de nouveaux objectifs, notamment en ce qui concerne l'étape de test du logiciel, par exemple, lors de l'utilisation d'artefacts issus de simulations. Les cas de tests peuvent être générés depuis les modèles, et vérifiés ensuite en utilisant des propriétés extraites de ces mêmes modèles. De nombreuses techniques de test basées sur les modèles sont disponibles dans la littérature, et dans des outils commercialisés. Notre projet de recherche s'inscrit dans ce contexte là et il fait partie d'un projet de plus grande envergure, appelé projet CRIAQ AVIO604.

Le projet CRIAQ AVIO604 est issu du partenariat de deux entreprises de production de logiciels aéronautiques, et de deux universités : l'université Concordia et l'ÉTS. L'objectif de ce projet est de développer des méthodes et outils facilitant la spécification et les tests des logiciels avioniques en tirant profit des avancées réalisées dans l'IDM. La portée du travail de l'ÉTS dans ce projet est illustrée par la Figure 0.1. Les objectifs de l'équipe de l'ÉTS sont :

1. Développer des langages pour supporter la spécification des exigences systèmes, des exigences de haut niveau (HLRs) et de bas niveau (LLRs : modèles de conception) et la traçabilité entre ces exigences. Le document de certification DO-331 utilise les termes *modèles de spécification* pour les HLRs et *modèles de conception* pour les LLRs.
2. Proposer et implémenter une approche pour maintenir/renforcer la cohérence entre les modèles de conception décrits dans différents langages ; les partenaires industriels utilisent notamment les langages UML et Simulink/Stateflow pour spécifier les modèles de conception.
3. Proposer et implémenter une approche qui permet de transformer les modèles de conception (partie comportementale) en une représentation intermédiaire unifiée qui pourrait supporter la génération des tests. Le formalisme visé par la transformation est un langage de machines à états finies étendues et communicantes (CEFSM).

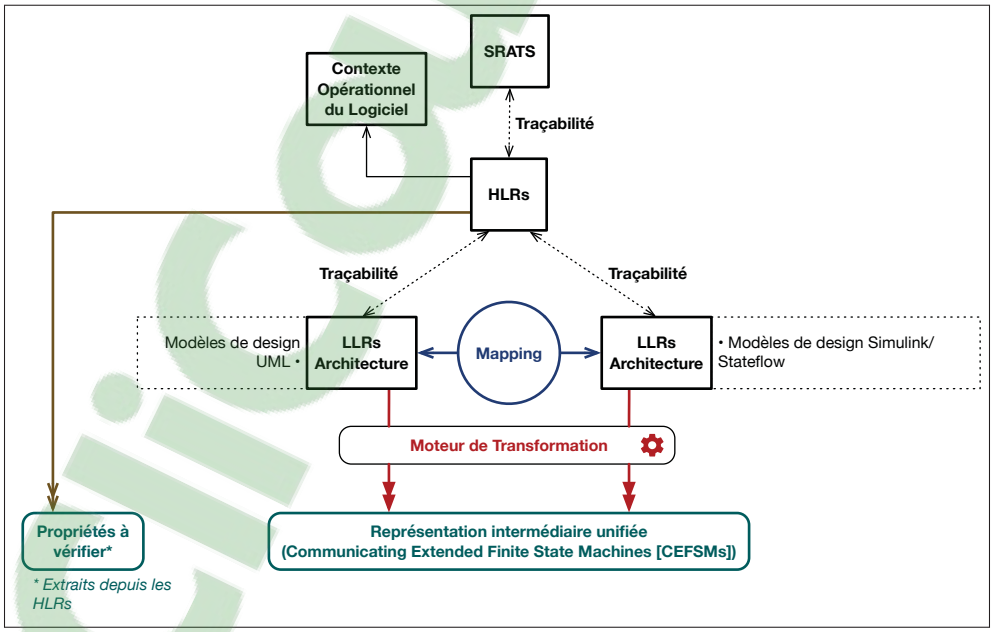


Figure 0.1 Le travail de l'équipe de l'ÉTS dans le projet CRIAQ AVIO604

Notre projet de recherche couvre le dernier objectif à savoir supporter la transformation de modèles de conception vers les CEFSM. En particulier, nous visons la transformation de modèles exprimés dans les langages utilisés par les partenaires industriels pour spécifier le comportement de leurs systèmes : les machines à états UML et Stateflow.

Problème de recherche

D'une part, même si les EFSM et les machines à états UML et Stateflow ont pour racine commune les Statecharts de D. Harel (Harel, 1987), leurs formalismes diffèrent de façon importante. Par exemple les EFSM ne permettent pas de définir de hiérarchie, contrairement à UML et Stateflow. D'autre part, il existe des travaux (Kim *et al.*, 1999; Yao & Shatz, 2006; Minas & Hoffmann, 2008) qui ont traité des transformations de machines à états vers d'autres formalismes. Cependant, très peu de travaux traitent le formalisme Stateflow. De plus la majorité des travaux ne proposent qu'une couverture partielle des concepts des langages en se focalisant sur certains concepts plutôt que d'offrir une solution générale. Enfin pratiquement aucun des travaux relevés ne proposent d'implémentation des transformations.

Objectifs

L'objectif de notre travail de recherche est de développer et implémenter une approche générique pour transformer un modèle de conception exprimé en utilisant un formalisme à base de machines à états vers un EFSM. L'approche proposée doit donc satisfaire aux exigences suivantes :

- Offrir une transformation couvrant l'ensemble des concepts de chacun des langages, plus spécifiquement l'ensemble des concepts utilisés par les partenaires industriels ;
- Offrir une transformation qui préserve le comportement des machines à états ;
- Définir un ensemble de bonnes pratiques de modélisation pour éviter les erreurs.

Méthodologie

Pour atteindre nos objectifs de recherche, nous avons adopté la méthodologie suivante :

- **Familiarisation avec les concepts clé** : Nous avons notamment étudié le standard DO-178 et l'ingénierie dirigée par les modèles ainsi que les technologies relatives. Plus particulièrement nous nous sommes intéressés aux transformations de modèles dans le cadre de l'IDM ainsi que les techniques existantes. Nous avons également étudié les techniques de test basés sur les modèles. Enfin nous avons étudié en profondeur les formalismes ciblés par le projet
- **Revue de littérature** : Ensuite, nous avons fait une revue des approches existantes de transformation des formalismes sources de notre projet (UML et Stateflow) vers le formalisme EFSM, ou un formalisme très proche. Nous avons relevé les points intéressants ainsi que les lacunes de ces travaux.
- **Proposition et application de l'approche de transformation générique** : Nous avons proposé une approche générique de transformation de formalismes de machines à états vers des EFSM. Cette approche comprenant une partie analyse de formalisme et une partie transformation. Cette approche a ensuite été appliquée aux deux langages sources du projet (UML et Stateflow) afin de valider l'approche.
- **Implémentation et étude de cas** : Enfin, nous avons implémenté l'application de l'approche pour les langages UML et Stateflow dans un outil open source à travers deux plugins. Nous avons ensuite mis en place une étude de cas pour l'implémentation de l'application de l'approche pour UML, afin de vérifier les résultats fournis par l'outil, et de prouver l'utilité de l'outil.

Plan

Le plan de ce mémoire est organisé de la façon suivante. Le Chapitre 1 présente les concepts nécessaires à la compréhension de l'approche proposée. Notamment en ce qui concerne les transformations de modèle et les formalismes de machines à états. Ensuite le Chapitre 2 fait une revue des méthodes de transformation de machines à états existantes, en s'efforçant de relever les éléments intéressants, ainsi que les lacunes des travaux existants. Le Chapitre 3 présente l'approche générale proposée pour la transformation des formalismes de machines à états en EFSM, ainsi que son application aux formalismes UML et Stateflow. Le Chapitre 4 présente l'implémentation de l'approche pour chaque langage dans un outil open source. Ce chapitre présente également une étude de cas menée pour l'un des deux langages, utilisant des modèles représentant des systèmes réels. Enfin nous concluons en résumant le travail accompli et en discutant des travaux futurs.

CHAPITRE 1

CONCEPTS

Ce chapitre aborde les concepts nécessaires à la compréhension de l'approche proposée, et présente les travaux pertinents pour chacun de ces concepts. Entre autres, nous nous attacherons à présenter l'ingénierie dirigée par les modèles, et dans notre contexte nous mettrons l'emphase sur les techniques de transformation de modèles. Suite à cela nous ferons une présentation détaillée des formalismes d'entrée et de sortie de notre transformation. Enfin nous verrons comment le formalisme cible de la transformation peut être utilisé dans le cadre des tests basés sur les modèles.

1.1 L'Ingénierie Dirigée par les Modèles

L'IDM est une méthode de développement de logiciel, s'appuyant sur la création et l'utilisation de modèles spécifiques à un domaine. L'utilisation de ces modèles permet de s'affranchir du choix d'une plateforme spécifique de développement. L'IDM a engendré un changement important dans les pratiques du génie logiciel. En effet nous sommes passés d'une pratique de développement centrée sur la notion d'objet à une pratique centrée sur la notion de *modèle* (Bézivin, 2004). Dans cette philosophie, tout est modèle, et les modèles exprimés pour décrire un domaine d'affaire sont transformés pour finalement obtenir des modèles spécifiques à la plateforme, en l'occurrence du code. La proximité de la représentation avec le monde réel plutôt qu'avec la plateforme de développement permettent une portabilité bien meilleure qu'avec les langages de programmation classiques. Ainsi dans le processus de spécification du logiciel, l'ingénieur peut s'affranchir des limitations imposées par le choix d'une plateforme.

Pour appréhender la complexité du logiciel, tout en utilisant une pratique de spécification de très haut niveau, l'IDM se base sur deux mécanismes (Schmidt, 2006) : I) La création de *langages de modélisation spécifiques au domaine*, II) L'utilisation de *moteurs de transformation et de générateurs*.

1.1.1 Les langages de modélisation spécifiques au domaine

Le premier mécanisme est la création de langages de modélisation spécifiques au domaine (DSML). Un DSML est un langage de modélisation créé pour modéliser un domaine particulier, pour satisfaire un ou plusieurs objectifs. Citons par exemple le langage *AADL* (Feiler, 2014), créé afin de modéliser à la fois la partie logicielle et matérielle d'un système temps réel aéronautique. Les modèles peuvent ensuite être utilisés pour de l'analyse (vérification de satisfaction des contraintes temporelles) ou de la génération de code. Un DSML vient généralement avec un ensemble d'outil permettant de satisfaire les objectif visés.

1.1.2 Les transformations de modèles

En plus de la création de DSML, on utilise les transformations de modèles. L'acte de transformation de modèles décrit le passage d'un modèle vers un autre type de modèle (de même formalisme ou non), ou vers du code. Une transformation est définie comme un ensemble de règles ou d'algorithmes. Une règle transforme un fragment particulier du modèle source en un fragment du modèle cible. La Figure 1.1 présente la transformation d'un diagramme de classe UML en base de données relationnelle. Cette transformation contient 2 règles : une pour la génération de la table de la base de données depuis la classe (ici la classe `Crayon`), et une autre permettant de générer les colonnes de la table depuis les attributs.

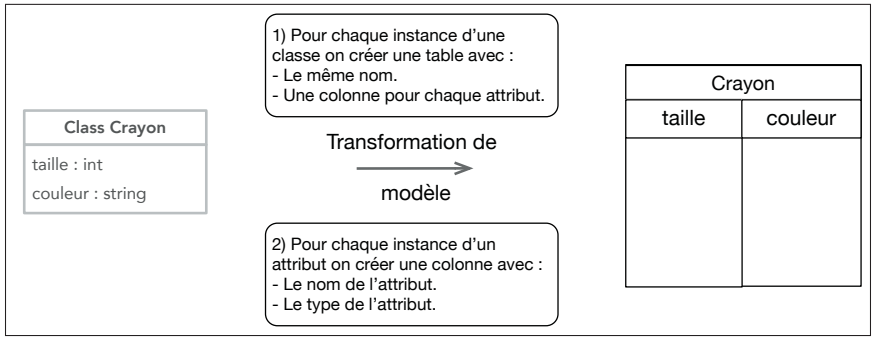


Figure 1.1 Exemples de transformations d'un diagramme de classe UML en base de données relationnelle

l'utilisation des transformations sera évoquée plus en détail dans la section 1.2.

1.1.3 Un exemple : Model-Driven Architecture

Au début des années 2000, l'OMG a proposé une implémentation de la démarche d'ingénierie dirigée par les modèles appelée *Model Driven Architecture* (MDA) (OMG, 2000). On retrouve au coeur de MDA un langage standardisé appelé UML (OMG, 2017), offrant des outils de modélisation s'appliquant de façon générale à un ensemble de domaines (à l'opposé d'un DSML). La démarche proposée dans le standard MDA reprend les fondements de l'ingénierie dirigée par les modèles, à savoir la création de DSML et les transformations de modèles. En ce qui concerne la création de DSML, le méta modèle UML supporte le langage mais permet également d'étendre ses possibilités. Ce mécanisme d'extension est appelé *profil*. Il permet de mettre à profit le méta modèle UML et de créer nos propres langages de modélisation, basés sur les besoins de notre domaine. Il existe des profils célèbres, parmi lesquels MARTE, pour modéliser les systèmes temps réel, ou SysML pour modéliser les exigences spécifiées. Pour aller plus loin, en plus du mécanisme de profil, MDA propose un langage pour spécifier les méta modèles, appelé MOF (OMG, 2016). MOF définit un standard permettant d'exprimer des méta modèles. On peut par exemple définir le méta modèle EFSM en utilisant MOF. L'architecture proposée par MDA est donc telle que présentée dans la Figure 1.2. Le premier niveau (M0) constitue le système réel modélisé par le langage utilisé (par exemple le langage UML) qui se trouve un niveau au dessus (M1). Ce langage doit se conformer à son méta modèle, qui va fixer ses règles de construction et d'utilisation (M2). Enfin, l'approche MDA offre la possibilité de définir ses propres méta modèles en utilisant le standard MOF (M3).

1.2 Les Transformations de Modèles

Comme précisé auparavant, le second pilier de l'ingénierie dirigée par les modèles est l'utilisation de moteurs de transformation et de génération. Dans MDA, les transformations de modèles ont une grande importance. Ils permettent en outre de changer de niveau d'abstraction, par exemple de passer d'un modèle indépendant de la plateforme de développement : PIM

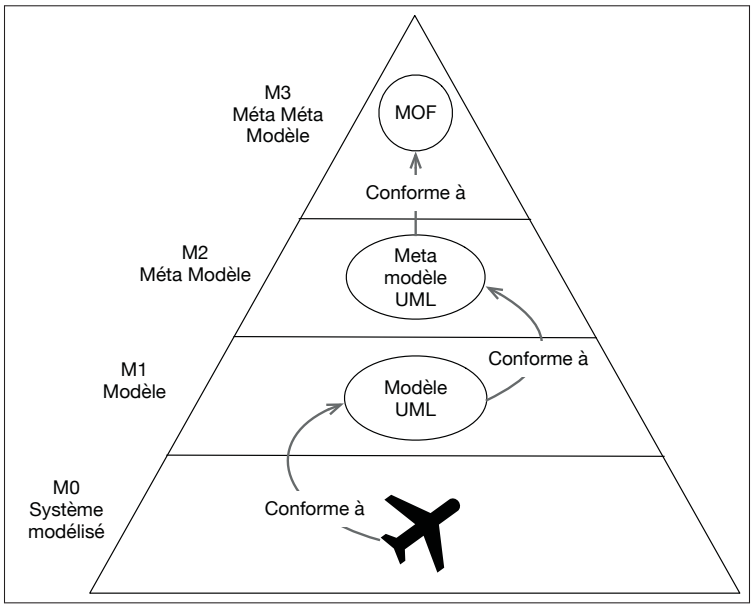


Figure 1.2 L'architecture en 4 couches proposée dans MDA

à un à un modèle dépendant de la plateforme : PSM. Ou encore ils permettent de passer d'un formalisme à un autre. Dans un objectif de gain de temps et d'optimisation des coûts, les transformations peuvent être automatisées. L'approche de MDA préconise que la transformation elle-même soit modélisée, mais elle peut être également développée dans un langage orienté objet par exemple. Dans une démarche de transformation dans le cadre de MDA, celle-ci doit s'intégrer dans l'architecture quatre couches présentée précédemment.

1.2.1 Un exemple de langage de transformation : le langage QVT

Query/View/Transformation (QVT) est un standard défini par l'OMG pour spécifier les transformations entre modèles, dont le méta modèle satisfait au standard MOF. Il comprend une partie déclarative et une partie impérative. La partie déclarative se compose de deux parties : une partie réalisant la correspondance entre les deux modèles exprimés dans le standard MOF nommée QVTr (relations), et une partie qui permet d'évaluer des conditions sur les éléments de nos modèles pour les faire correspondre, nommée QVTc (Core). Ces deux parties se servent du langage OCL (Object Constraint Language) (OMG, 2014) pour définir les règles

de correspondance. OCL est un langage formel standardisé par l'OMG permettant de spécifier des contraintes logicielles. La partie impérative, constituée de QVTo (opérationnel), permet d'étendre le langage déclaratif. Des constructions telles que les boucles *for* ou des conditions *if* y sont offertes. QVTo introduit également l'utilisation de règles OCL impératives.

1.2.2 Exigences sur les transformations de modèles

Pour supporter la productivité et diminuer les coûts, les transformations doivent satisfaire certaines exigences. En se basant sur les travaux de Mens & Van Gorp (2006), Sendall & Kozaczynski (2003), Kolahdouz-Rahimi *et al.* (2014) et sur la norme ISO révélant les critères nécessaires à la production d'un logiciel de qualité (ISO, 2017) nous considérons que les principales exigences sont les suivantes :

L'utilisabilité : L'utilisateur final doit pouvoir utiliser la transformation, l'intégrer à ses outils sans que cela n'amène trop de complexité, ou que l'ensemble de son processus de travail ne soit changé.

L'efficacité : La transformation doit s'exécuter dans un temps convenable. Elle ne doit pas non plus consommer une quantité excessive de ressources, quelque soit le modèle fournit en entrée.

La maintenabilité : Comme tout autre artefact logiciel, une transformation est amenée à évoluer ; i.e., certains concepts du langage changent et d'autres sont nouvellement introduits. La transformation doit être facile à mettre à jour en conséquence. La transformation doit donc être écrite (ou modélisée) de façon claire et précise.

La fiabilité : La transformation doit donner le résultat attendu pour un modèle d'entrée donné à chaque exécution.

L'automatisabilité : La transformation doit automatiser le plus de règles possibles. L'idéal étant qu'il n'y ai aucune intervention humaine lors de la transformation, c'est à dire que pour un modèle d'entrée donné on obtient le modèle de sortie, et de préférence sous la forme que l'on doit traiter.

La vérifiabilité : La transformation doit pouvoir être vérifiée et validée.

L'interopérabilité : La transformation doit pouvoir s'intégrer dans le processus de l'utilisateur. Notamment, les points de connexions (modèles d'entrée et de sortie) doivent être formalisés de façon à ce que le reste des étapes du processus ne soient pas impactées.

La technique utilisée pour la transformation, le langage de la transformation, la façon dont la transformation va être implémentée ainsi que l'outil qui va être utilisé auront un impact sur la satisfaction de ces exigences. Les formalisme des modèles en entrée et en sortie orienteront également sur les différentes techniques, langages et outils disponibles.

1.2.3 Les différents types de transformation de modèles

Les travaux Mens & Van Gorp (2006), Kolahdouz-Rahimi *et al.* (2014), Czarnecki & Helsen (2006) classifient les transformations selon différents critères. On distinguera par exemple les transformations unidirectionnelles s'exécutant dans un seul sens (source vers cible) et les transformations bidirectionnelles. Dans le cas où les langages source et cible sont les mêmes, on parle de transformation *endogène*. Plusieurs exemples de ce type sont donnés dans Mens & Van Gorp (2006) :

- **Optimisation** du modèle, pour le rendre plus performant tout en conservant son comportement.
- **Refactoring** du modèle, pour changer la structure sans changer le comportement.
- **Simplification** du modèle.

Dans le cas où le langage des modèles cibles et sources sont différents on parle de transformation *exogène*. Des exemples de transformation sont :

- **Génération de code**. Par exemple passer d'un modèle de conception UML à du code source Java.
- **Rétro ingénierie** correspondant à la transformation inverse à celle mentionnée au dessus.
- **Migration** d'un langage vers un autre tout en restant au même niveau d'abstraction.

En plus de ça, une transformation peut être horizontale (si elle reste au même niveau d'abstraction) ou verticale si on change de niveau. On retrouve des exemples de transformations verticales dans la Figure 1.3, lors du passage de PIM à PSM, et de PSM au code source. Le changement de niveau d'abstraction est la base de la démarche de l'IDM. A savoir, partir d'un modèle proche du domaine modélisé, pour finir par obtenir du code source implémentable. Dans la démarche MDA, on parle d'un passage d'un modèle indépendant de la plateforme, à un modèle spécifique pour enfin obtenir du code source. La Figure 1.3 illustre ces différents types de transformation.

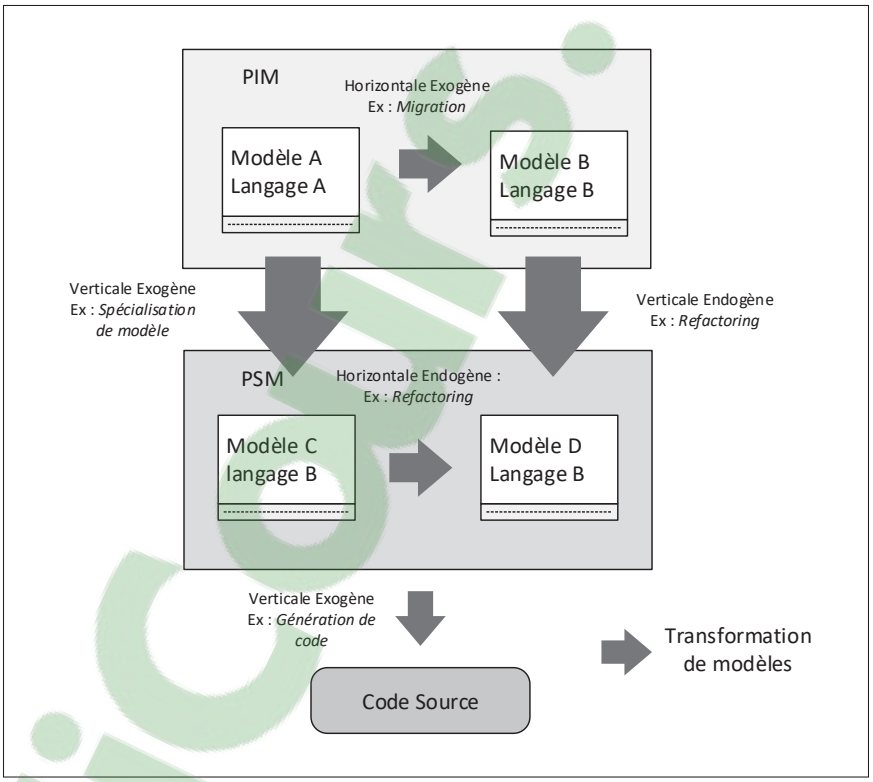


Figure 1.3 Exemples de transformations de modèles dans le cadre de MDA

1.2.4 Les techniques de transformation

Pour spécifier et implémenter correctement une transformation, il faut comprendre la syntaxe et la sémantique des langages source et cible. Il existe différentes techniques pour spécifier ces

transformations selon les traitements réalisés sur les modèles. Le travail de Sendall & Kozaczynski (2003) classifie les techniques de transformation en trois catégories différentes.

- **La manipulation directe de modèles** : Cette technique utilise la représentation du modèle dans son langage source, et la manipule directement à travers une API.
- **L'utilisation d'une représentation intermédiaire** : Cette fois on exporte le modèle vers un langage standardisé, qui est supporté par un ensemble d'outil, et qui permet une manipulation plus simple.
- **La spécification de la transformation en utilisant un langage** : Un langage de transformation permet de définir des transformations de modèles de façon complète, de façon indépendante des langages source et cible. Cette transformation pourra être générique et réutilisable.

Czarnecki & Helsen (2003) différencie les transformations de modèle-à-code et de modèle-à-modèle. Pour la mise en oeuvre de transformation de modèle-à-modèle, ils évoquent les techniques suivantes :

- **Les approches basées sur la manipulation directe de modèles** : On manipule le modèle directement à travers une API. C'est notamment cette méthode que l'on retrouve dans la plupart des outils commerciaux qui ont défini leur propre formalisme. Cette méthode restreint néanmoins la transformation à un langage spécifique, et nécessite des connaissances poussées de la structure du langage.
- **Les approches relationnelles** : Cette approche propose de définir une simple relation entre un (ou plusieurs) élément du modèle source et un (ou plusieurs) élément du modèle cible. Cette méthode est celle que l'on retrouve dans la partie relationnelle du langage QVT, permettant de réaliser du "pattern matching" entre éléments de méta modèles satisfaisant au standard MOF.
- **Les approches basées sur les transformations de graphes** : Représenter les modèles sous forme de graphes et appliquer la transformation sous forme de règles. Plusieurs travaux utilisent cette technique (e.g., Kuske, 2001; Geiß *et al.*, 2006; Grønmo *et al.*, 2009).

- **Les approches dirigées par la structure** : elles sont constituées de deux phases : d'abord on crée la structure hiérarchique du modèle cible à partir d'informations du modèle source, puis on crée des règles de transformation pour les éléments du modèle source devant être modifiés. Cette technique est très peu utilisée. Un des exemple célèbres est *OptimalJ*, qui n'est plus maintenu depuis 2008.
- **Les approches hybrides** : Mélange d'une approche déclarative (approche relationnelle), et d'une approche impérative permettant de traiter de façon extensive les éléments du modèle source. On retrouve ici les implémentations de QVT, prenant en compte la partie relationnelle, ainsi que la partie impérative du langage. Un des outils célèbres est le langage *ATL*, dont nous parlerons dans la prochaine section.
- **Autres** : Difficilement classifiable, l'utilisation du standard CWM de l'OMG par exemple ou encore la manipulation de représentations XML avec XSLT. L'utilisation de XSLT est un mélange de l'utilisation d'un langage de transformation (XSLT) et d'une représentation intermédiaire (XML).

Les approches existantes se différencient par les techniques utilisées. En plus, elles offrent des outils avec un degré variable d'automatisation de la transformation. L'article de Gomes *et al.* (2014), propose une classification des outils selon différents critères objectifs. L'un des critères est le taux d'automatisation de la transformation. Une technique manuelle impose à l'utilisateur de développer la transformation pratiquement "from scratch". Alors qu'un outil automatique ne demande que très peu d'intervention.

Sur l'ensemble de ces techniques, on peut identifier certaines plus adaptées que d'autre selon la situation. L'utilisation d'une approche basée sur la manipulation directe de modèles va restreindre l'utilisation à un langage particulier. Qui plus est il est nécessaire de connaître parfaitement le langage et ses constructions. C'est pour cela qu'elle est privilégiée dans les outils commerciaux. Une approche relationnelle sera très efficace lorsqu'il s'agira de réaliser un simple *refactoring* par exemple, mais sera limitée si on a besoin de réaliser des modifications profondes depuis le modèle source. Les approches dirigées par les structures sont plutôt complexes, et nécessitent d'avoir un outil à disposition qui permet de bâtir la structure du modèle

source depuis le modèle cible. L'approche hybride et l'approche basée sur les graphes sont les plus complètes, s'affranchissant des formalismes source et cible tout en offrant la possibilité de faire des manipulations complexes du modèle source. Néanmoins la transformation de graphes est une tâche qui peut être très complexe. Dans le cadre de MDA, QVT est la solution proposant une approche hybride.

1.2.5 Les implémentations de QVT.

Depuis que l'OMG a proposé le standard QVT pour l'expression des transformations, plusieurs implémentations ont émergé. Les trois principales sont ATL (Jouault *et al.*, 2008), QVT opérationnel (QVTo) (Eclipse, 2019c) et QVT Relationnel (QVTd) (Eclipse, 2019b). Il y a également VIATRA (Eclipse, 2019d), un outil basé sur les transformations de graphes, qui implémente certains concepts de QVT. Ces langages sont implémentés dans l'IDE Eclipse en utilisant le cadre de développement Eclipse Modeling Framework (EMF). EMF est un plug-in qui permet de modéliser et de manipuler les modèles. EMF laisse la possibilité de modéliser ses propres méta modèles (en respectant le standard MOF) et ainsi définir les langages source et cible des transformations.

QVTd implémente la partie relationnelle du langage, ainsi que QVTCore. Ce plug-in permet de faire correspondre des éléments du modèle source, avec des éléments du modèle cible. Les règles permettant de conditionner cette correspondance sont exprimées en utilisant le langage OCL. Alors que le plug-in **QVTo** implémente la partie impérative du langage. Celui-ci permet de définir des relations quand une expression purement déclarative ne suffit pas, en s'appuyant cette fois sur des règles OCL impératives. Ces règles enrichissent les règles basiques avec des constructions plus complexes (e.g., une boucle while), tout en gardant l'expressivité du langage. Ces deux plug-ins offrent tout un ensemble d'outils utiles à leur déploiement sur Eclipse (API Java, debugger, editeur spécifique, etc ..).

VIATRA2 implémente à la fois la partie déclarative et la partie impérative du langage. Cette implémentation est basée sur deux formalismes : les transformations de graphes et les ma-

chines à états abstraites. D'un coté, VIATRA2 ne satisfait pas au standard MOF : il offre un outil de d'expression de méta modèles plus expressif que MOF appelé *VPM model space* et le langage de transformation utilisé n'est pas QVT. Cependant, il offre un support permettant de traduire la spécification de QVT vers les deux formalismes utilisés. VIATRA2 offre un environnement complet intégré à Eclipse, proposant également des mécanismes de vérification des transformations produites.

ATL implémente à la fois la partie déclarative et la partie impérative du langage. Il a été développé en réponse à la demande de l'OMG de fournir une implémentation de l'approche définie dans QVT. Le langage ATL fournit un moyen simple de définir des transformations de modèles depuis et vers des modèles définis en accord avec le standard MOF. De la même façon que pour VIATRA, tout un ensemble d'outil est fourni dans l'environnement Eclipse pour supporter son utilisation. Nous verrons plus en détail l'utilisation du langage ATL dans le Chapitre 4.

VIATRA n'étant pas purement basé sur QVT, nous nous intéresserons plutôt aux trois autres langages. Plusieurs paramètres sont à considérer lors du choix d'une de ces implémentations. Le premier paramètre est la performance de chacun, dans un contexte où la taille des modèles peut considérablement impacter le temps d'exécution de la transformation. Le travail de van Amstel *et al.* (2011) s'attache à comparer ces trois langages en termes de performance. Pour cela il s'appuie sur deux types de modèles en entrée (un de type graphe et un de type arbre) et incrémente le nombre d'éléments de chacun tout en vérifiant le temps d'exécution des transformations. Les résultats des travaux montrent que ATL est plus performant que QVTo et QVTd, quelque soit le type de modèle en entrée. Ils montrent également que les résultats ne sont pas grandement impactés par l'utilisation d'une implémentation déclarative plutôt qu'impérative d'ATL.

1.3 Langage de modélisation source de notre transformation : les machines à états UML

Le standard UML offre un guide pour l'utilisation de modèles adaptés à chaque étape de l'analyse et de la conception du logiciel, avec une proposition de représentation graphique pour

chaque modèle. Malgré la présence d'une spécification détaillée, le standard offre une assez grande liberté d'interprétation des concepts proposés.

Les machines à états permettent de présenter le comportement du logiciel a travers une succession d'états. Le comportement du logiciel en réponse à l'occurrence de certains événements est représenté par une séquence de chemins à travers la machine. Une machine à états contient une ou plusieurs régions. La présence de multiples régions peut représenter deux comportements différents :

- Si elles sont orthogonales elles représentent des fragments concurrent pouvant s'exécuter en parallèle.
- Imbriquées elles permettent de définir des niveaux hiérarchiques dans la machine à états.

Une région contient un ensemble de *noeuds*, correspondant à des *états* ou des *pseudo-états*, connectés entre eux par des transitions. Par exemple, dans la Figure 1.4, la machine à états `Light` contient une seule région contenant les états `Off` et `On`. Il existe un type particulier de machine à états appelé *sous machine à états*. Cette construction permet la réutilisabilité de la machine : une sous machine à état peut être appelé par d'autres machines.

Enfin, les machines à états UML peuvent communiquer entre elles, à travers un mécanisme de signaux : une machine peut envoyer un signal à une ou plusieurs autres machines. Le signal peut alors déclencher l'exécution d'une transition d'un état vers un autre.

1.3.1 Les états

Un état est soit actif soit inactif. Il représente l'état du logiciel à un instant donné. Plusieurs états peuvent être actifs au même instant, en existant dans des régions orthogonales différentes. Les états sont connectés entre eux par des transitions, définissant les conditions de passage de changement d'état du logiciel. Dans la Figure 1.4, le passage de l'état `Off` à l'état `On` est conditionné par l'événement `SWITCH` et la condition `powerRunning=true`.

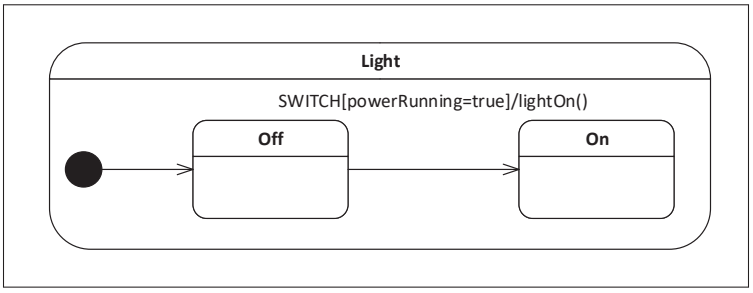


Figure 1.4 Les états UML

Les états peuvent être soit de type simple, soit de type composite (ou contenir une sous machine à états). Un état simple, comme son nom l'indique, ne contient pas de sous états. Un état composite peut contenir une ou plusieurs régions. Comme précisé précédemment les régions contiennent des noeuds et des transitions. Il existe deux types d'états composites :

- **Le Or-state** : qui est un état composite avec une et une seule région. L'activation de cet état entraîne l'activation d'un et un seul de ses sous états à un instant donné.
- **Le And-state** : qui est un état composite avec deux régions ou plus. Quand cet état est actif, il y a un sous état par région qui est actif. Donc plusieurs de ses sous états sont actifs à un instant donné. Par exemple dans l'état `LightOn` dans la Figure 1.5, à un instant t la lampe principale et la lampe secondaire peuvent être toutes deux dans l'état `Running` dans chaque région.

Enfin, les états peuvent contenir trois types d'*activités internes*. D'abord l'activité *entry*. Celle-ci déclenche une action aussitôt que l'état est activé, après que la transition entrante ait été exécutée. Ensuite l'activité *do*, qui définit une action qui sera exécutée tout le temps où l'état est actif. Enfin une activité *exit*, définissant une action déclenchée lorsque l'état est désactivé, juste avant que la transition sortante soit exécutée. On retrouve un exemple de ces activités dans la Figure 1.5.

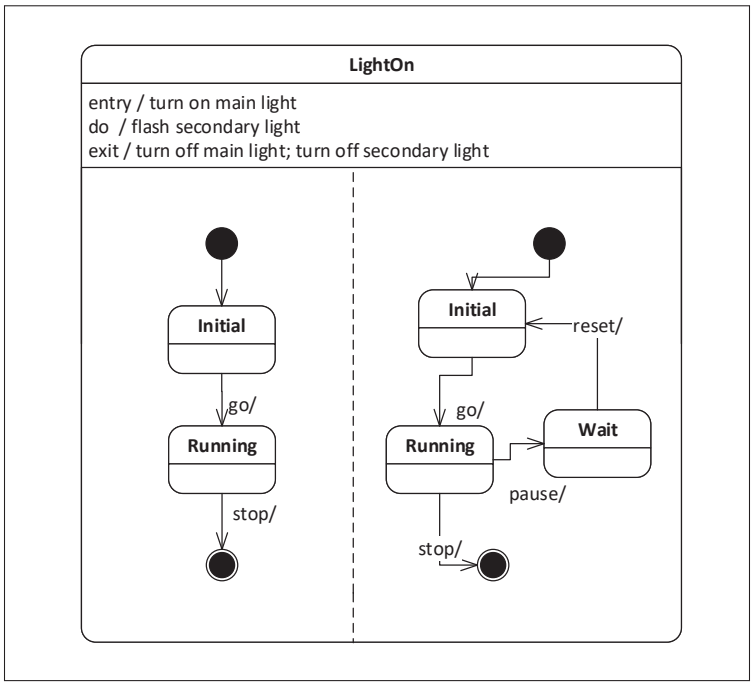


Figure 1.5 Exemple d'état composite

1.3.2 Les transitions

Une transition est une connexion entre deux noeuds (état ou pseudo-état). Elle possède une unique source et une unique cible. Lors de son activation, une transition est d'abord dite *atteinte*, quand son noeud source est actif. Elle est ensuite *exécutée* puis *complétée* une fois que son noeud cible est atteint.

Il existe différents types de transition. La plus courante est la transition *externe*, dont l'exécution amène l'inactivation du vertex source. Ensuite la transition *locale*, qui n'existe que dans les états composites. Elle consiste en une transition entre deux états appartenant tous les deux au même état composite. Enfin, il y a les transitions *internes*. Celles-ci sont des transitions de l'état vers lui-même.

Une transition est définie par trois caractéristiques optionnelles, que nous appellerons dans la suite "items" :

- *Des évènements "Triggers" :* qui définissent les événements nécessaires au déclenchement d'une transition. Le trigger de la transition entre Off et On de la Figure 1.4 est l'événement SWITCH.
- *Une condition "Guard" :* C'est une condition qui doit être évaluée vraie pour que la transition puisse être déclenchée. Dans la Figure 1.4, la transition de l'état Off à On ne peut être déclenchée que sous la condition powerRunning=true.
- *Une action "Effect" :* qui correspond à une ou plusieurs actions qui seront exécutées si la transition est exécutée. L'action lightOn() est déclenchée par l'exécution de la transition dans la Figure 1.4.

1.3.3 Les pseudo-états

D'après la spécification UML, un pseudo-état est une "abstraction qui englobe différents types de noeuds transitoires dans la machine à états". Ceci signifie qu'un pseudo-état permet d'exprimer un comportement particulier, sans pour autant être un noeud qui sera considéré actif à un instant t, à la différence d'un état. Il existe plusieurs types de pseudo-états, exprimant des comportements différents :

- Le pseudo-état *initial* : celui-ci sert à démarrer l'exécution d'une région. Il est obligatoire pour chaque région. Dans la Figure 1.6, le pseudo-état initial indique que le premier état actif sera l'état A au démarrage de la machine.
- Les noeuds d'histoire *deep* et *shallow* permettant de conserver une trace de l'exécution passée de la machine. Ainsi si un noeud d'histoire est placé dans un état composite, on connaîtra l'historique d'exécution de ses sous états. Par exemple le dernier sous état actif sera prit comme point de départ si l'état composite est réactivé.
- Les noeuds *join* et *fork* : ceux-ci permettent de fusionner ou séparer un ensemble de transitions respectivement. Pour sortir du fork ou du join, chacune des transitions entrantes devra avoir été exécutée. Dans la Figure 1.6, L'état C est connecté aux états G et H par un fork. Lorsque l'événement E5 arrive, les deux états G et H sont activés. Ensuite, pour que l'on

sorte du noeud join et que l'état final soit actif, il faut que les deux événements E6 et E7 arrivent. Si seulement l'un d'eux est déclenché, on attend le second avant de sortir du noeud.

- le noeud de *jonction* : Celui-ci permet de simplifier l'écriture de multiples transitions sortantes ou entrantes des états. La Figure 1.6 contient un noeud de jonction liant les états A avec B et C. L'évaluation des conditions des transitions sortantes de ce noeud est réalisée de manière statique (sans tenir compte d'éventuelles actions sur les transitions entrantes).
- Le noeud de *décision* : Le comportement est similaire à celui du noeud de jonction, mais celui là permet d'assurer que les conditions des transitions sortantes seront évaluées en fonction de l'effet de l'action des transitions entrantes (évaluation dynamique). On l'utilise fréquemment pour exprimer le comportement *if-then-else*. Par exemple le noeud de décision qui relie les états A, D et E dans la Figure 1.6 assure que l'action A2 soit exécutée avant d'évaluer les conditions des transitions sortantes.
- Le point d'*entrée* et de *sortie* : permettent d'offrir une alternative à l'entrée ou la sortie d'une région. Un exemple est présenté à la Figure 1.6 dans l'état composite E.
- Le point *terminate* : permettant de terminer l'exécution d'une machine à états de façon prématurée. À ne pas confondre avec un état final, qui lui n'est pas un pseudo-état, mais un état exprimant la terminaison de l'exécution d'une région. Dans la Figure 1.6, si l'événement E4 arrive alors que l'on est dans l'état D, alors on arrête l'exécution de la machine.

Les pseudo-états sont une particularité du langage UML, nous ne les retrouvons pas dans les autres langages.

1.4 Langage de modélisation source de notre transformation : les machines à états Stateflow

L'entreprise MathWorks propose un langage de modélisation en complément du langage de développement Matlab : le langage Simulink. C'est un langage graphique de modélisation définissant un ensemble de blocs paramétrables incluant différentes fonctions mathématiques de base.

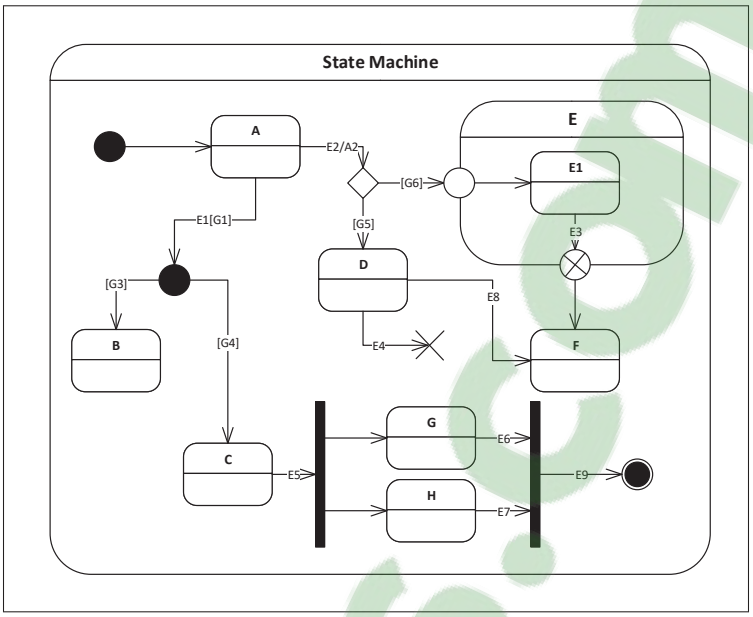


Figure 1.6 Les pseudo-états dans le langage UML

Les machines Stateflow étendent le langage Simulink en offrant un mécanisme similaire aux machines à états d'Harel. Une machine Stateflow s'intégre dans un diagramme Simulink, et s'interface avec les blocs à travers des ports d'entrées et de sorties. Stateflow est utilisé pour représenter la partie contrôle du logiciel. Contrairement au langage UML, Stateflow ne laisse aucune place à l'interprétation.

Dans les sections suivantes nous présenterons les concepts du langage Stateflow. Les figures illustrant les concepts de Stateflow sont adaptées du guide utilisateur (Mathworks, 2019).

1.4.1 Les états

Un état est soit actif ou inactif. Un état actif correspond à une situation stable du système. L'évolution du système à travers les différents états se fait en réponse à l'occurrence d'événements. La spécification Stateflow introduit la possibilité de définir une hiérarchie dans les états en utilisant des états composites. Dans cette situation, un état composite devient actif, suivi par ses sous-états. Quand tous ses sous états deviennent inactifs, l'état composite devient inactif. Les états sont liés entre eux par des transitions, permettant de déterminer les conditions d'acti-

vation et d'inactivation de ceux-ci. Les états `A` et `B` de la Figure 1.7 sont par exemple liés par une transition conditionnée par l'occurrence de l'événement `E1`.

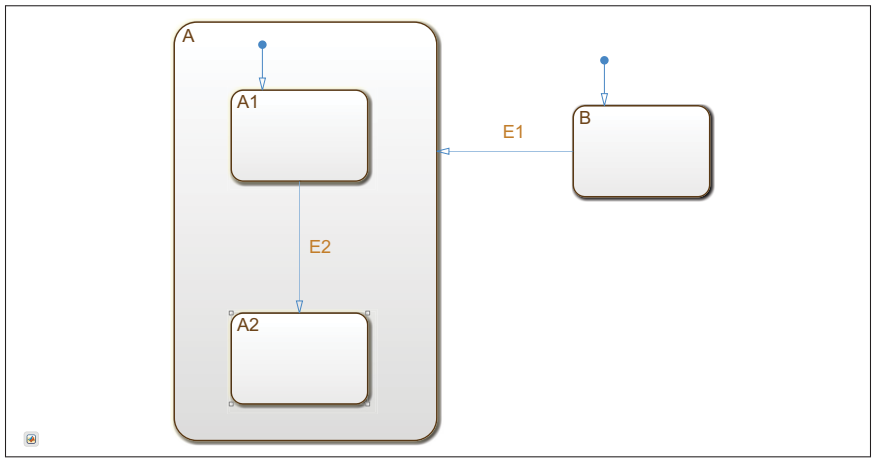


Figure 1.7 Les états dans Stateflow

Un état composite peut contenir un ensemble séquentiel de sous états. Ce type est nommé *Or-state*. Dans ce cas un seul des sous états peut être actif à un instant donné. C'est par exemple le cas de l'état `A` de la Figure 1.7, étant un Or-state contenant deux sous états `A1` et `A2`. Un état composite peut aussi contenir un ensemble parallèle de sous états. Cet état est appelé *And-state*. Cette fois plusieurs sous états peuvent être actifs au même moment (c'est le cas des états `gearState` et `selectionState` dans la Figure 1.8). Pour éviter toute situation non déterministe, un ordre sera attribué (manuellement ou automatiquement) à chaque état parallèle. L'état `gearState` a la priorité 1, et l'état `selectionState` a la priorité 2.

Un état (simple ou composite) peut contenir des *actions internes*. Une action interne peut être de type *entrée*, dans ce cas elle est exécutée aussitôt que l'état la contenant devient actif. Une action peut être de type *during*, dans ce cas elle est exécutée durant la période où l'état est actif. Enfin une action de *sortie* peut être définie. Celle-ci est exécutée aussitôt que l'état devient inactif.

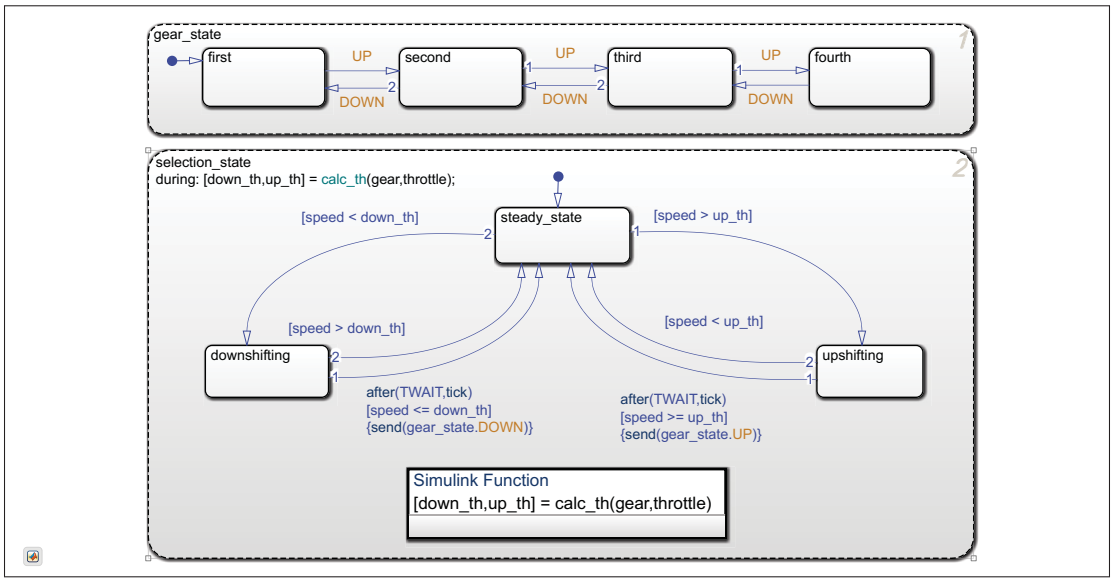


Figure 1.8 États parallèles - Machine à états Stateflow extraite du logiciel Matlab

1.4.2 Les transitions

Une transition correspond à un arc représentant le passage d'un état à un autre. Les séquences de transitions décrivent les différents comportements adoptés par le système. Une transition sera empruntée lors de la sortie de son état source en réponse à l'occurrence d'un événement. Il existe différents types de transitions. La transition classique, où l'état source et cible sont différents, et au même niveau hiérarchique. Une transition *self-loop*, ou l'état source et cible sont les mêmes. Les transitions entre niveaux, pour lesquelles la source et la cible appartiennent à des niveaux de hiérarchie différents. Et la transition par défaut, équivalente à un pseudo-état initial pour UML, sans état source, qui permet d'indiquer l'état s'activant à l'activation d'un thread local (e.g., le sous état d'un état composite). La transition entrante dans l'état `A1` dans la Figure 1.7 en est un exemple.

Les transitions sont caractérisées par les éléments suivants :

- Des *événements* : l'occurrence d'un des événements déclenche l'évaluation de la transition. On retrouve par exemple l'événement `E` sur la Figure 1.9 entre les états `On` et `Off`.

- Des *conditions* : Les valeurs de certaines variables sont évaluées. La condition [offCount==0] de la Figure 1.9 évalue la valeur de la variable offCount.
- Des *conditions actions* : Si les conditions sont évaluées *vraies*, ces actions sont exécutées. C'est le cas de offCount = offcount + 1 dans la Figure 1.9. Cette construction n'apparaît cependant plus dans la version 2019 de la spécification.
- Des *actions* : Si la transition est finalement empruntée pour transiter vers l'état cible, ces actions sont exécutées. La fonction lightOff() de la transition entre les états On et Off de la Figure 1.9 est un exemple d'action.

Afin d'éviter toute situation non déterministe, les transitions possèdent également des priorités. Un états possédant plusieurs transitions sortantes doit définir un ordre de priorité pour chacune des transitions. Ainsi, les conditions sur les variables des transitions sont évaluées dans l'ordre selon leurs priorités.

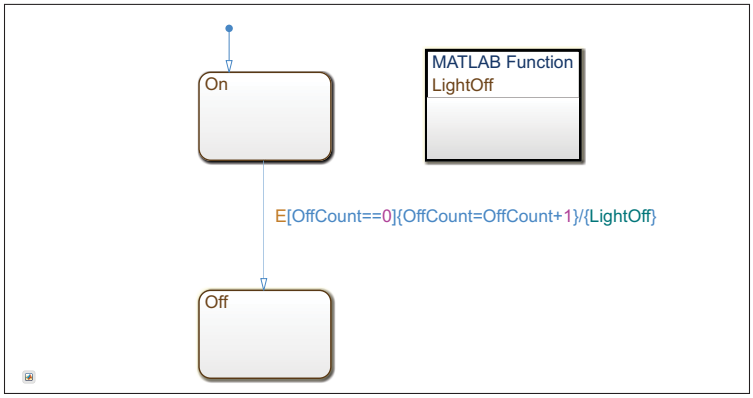


Figure 1.9 Les transitions dans le langage Stateflow

Un événement peut être *explicite* s'il n'est pas produit par la machine, par exemple si il est produit par l'environnement. Dans la Figure 1.8, l'événement after(TWAIT, tick) est indépendant du déroulement de la machine. Ou alors les états eux mêmes peuvent produire des événements, qui sont dans ce cas *implicites*. Ils peuvent les partager comme c'est le cas pour les événements UP et DOWN dans la Figure 1.8. Durant leurs exécutions, à travers des constructions telles que *send(événement)*, une transition peut produire un événement à travers

son action, et cet événement sera reçu par une autre transition pour évaluation. Cela permet par exemple de synchroniser des états parallèles.

1.4.3 Les jonctions

Il existe deux types de jonctions : la *jonction d'histoire* et la *jonction connective*. Une jonction d'histoire placée dans un état composite permet de conserver l'historique de l'activation des sous états. Par exemple, si on réactive l'état composite, le dernier sous état actif est connu et est pris comme point de départ.

Le cas classique d'utilisation d'une jonction connective est de fusionner et découper des transitions. Fusionner plusieurs transitions de sources multiples vers une seule cible. Découper une transition depuis une source vers de multiples cibles. Fusionner plusieurs transitions depuis une source vers une destination basée sur le même événement. La jonction entre les états `D`, `E` et `F` dans la Figure 1.10 permet de lier les transition basées sur l'événement commun `EOne`, tandis que la jonction entre `A` et `B`, `C` sépare les transitions déclenchées par deux événements différents. Cette utilisation est essentiellement une simplification graphique.

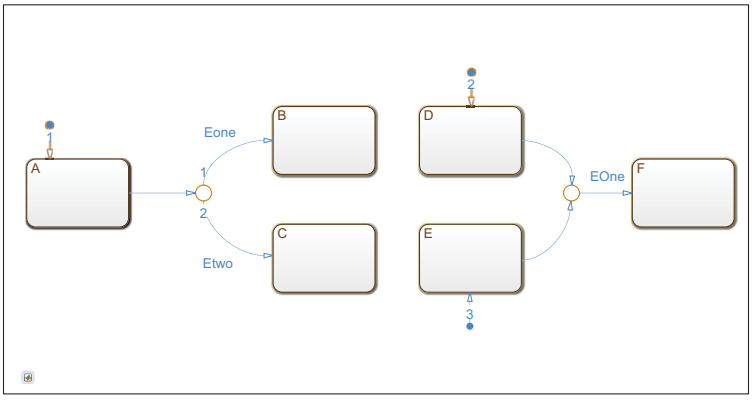


Figure 1.10 Réunir ou séparer des transitions avec une jonction connective

Une jonction de connexion permet aussi d'offrir différentes alternatives à une seule transition selon le type de construction. Celle-ci permet de représenter les comportements suivants à travers des constructions particulières :

Le comportement d'un *if-then-else*

Celui-ci se construit avec un noeud de jonction, comprenant plusieurs transitions sortantes, possédant des guards différents. Pour éviter de tomber dans une situation non déterministe, une des transitions sortantes (celle la moins prioritaire), ne comporte pas de guard. Celle-ci correspond à la branche *else*. Dans la Figure 1.11, si la condition `x>2` n'est pas vérifiée une fois la jonction atteinte, on emprunte la transition avec le priorité 2 connecté à l'état `D`.

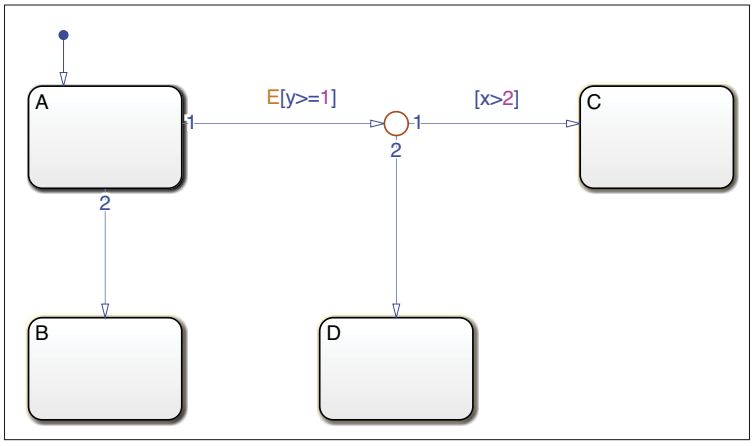


Figure 1.11 Utilisation d'une jonction connective pour exprimer un comportement *if-then-else*

Le *Backtracking*

Ce cas se construit comme une chaîne de plusieurs jonctions connectives. Entre chacune des jonctions les transitions contiennent optionnellement des conditions et des actions, qui sont évaluées dans l'ordre de priorité définie. Il se peut qu'après avoir exécuté un certain nombre de transitions, aucune des transitions sortantes de la jonction atteinte ne puisse être exécutée. Dans ce cas on revient à l'état de départ de la chaîne de jonction, et on évalue les transitions moins prioritaires si il y a lieu. Dans la Figure 1.12, la condition `y>=1` de la transition avec la priorité

1 est évaluée, ensuite on tombe dans le premier point de jonction, et on évalue d'abord si `x>2`, si ce n'est pas le cas on évalue `x>=1`. Si ce dernier guard est vrai on exécute la condition action `y=0` et on évalue si `z>5`. Si celui-ci est faux on retourne à l'état `A` et on exécute la transition avec la priorité 2.

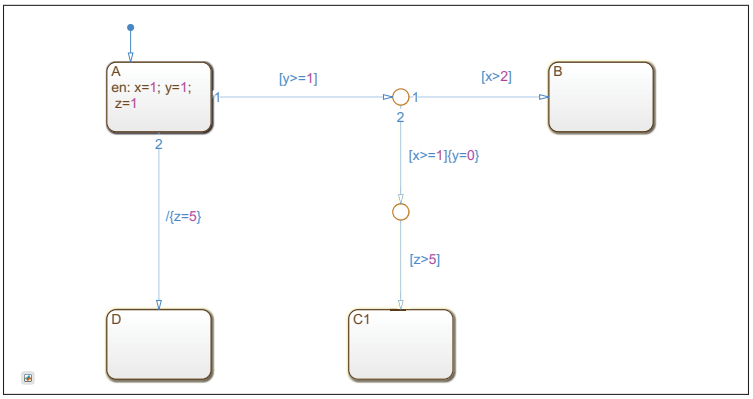


Figure 1.12 Cas de Backtracking utilisant des jonction connectives

Le comportement d'une boucle *for*

Ce comportement se construit comme une transition de la jonction vers elle même avec les composants d'une boucle for classique (e.g., *for(int i=0, i<10, i++)*). Cette transition, ayant une priorité plus élevée que les transitions sortantes de la jonction, s'exécute tant qu'on a pas atteint la limite de la boucle. Dans la Figure 1.13, la jonction connective entre les états `A` et `B` assure que la fonction `func1()` est exécutée 10 fois.

1.5 Langage de modélisation cible de notre transformation : Les machines à états finis étendues

Les machines à états finis étendues (EFSM) ont été choisies comme langage unificateur dans le contexte du projet AVIO604. C'est donc le langage cible de notre transformation. En effet, les EFSM ont l'avantage de s'appuyer sur le formalisme des machines à état finis, permettant de représenter la partie flot de contrôle de notre système, tout en apportant une partie flot de données à travers l'utilisation et la manipulation de variables. Son formalisme est plus simple

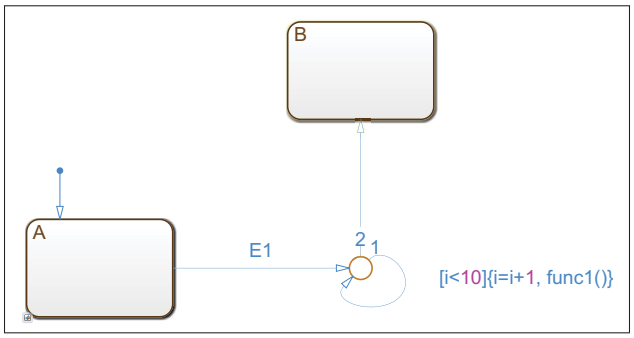


Figure 1.13 Utilisation d'une jonction connective pour exprimer un comportement boucle *for*

que celui des machines Stateflow et UML : il n'inclut pas le concept de jonction, de pseudo-états, d'événement et d'action interne. Ayant pour objectif de générer des tests, dans le projet AVIO604 nous avons considéré des EFSM sans hiérarchie ; i.e., les états sont tous simples. Un EFSM se présente sous la forme d'une succession d'états et de transitions relatant le comportement d'un système. Les chemins depuis un état initial jusqu'à un état final représentent les scénarios comportementaux du système, tout en explicitant l'enchaînement d'événements et de conditions qui les contraignent.

1.5.1 Les états et transitions

Un état dans le formalisme EFSM a la même signification qu'un état dans le langage Stateflow ou dans le langage des machines à états UML. Un état est soit actif, soit inactif. Cependant, plusieurs états ne peuvent pas être actifs au même moment au sein de la même machine. Les états sont actifs de manière séquentielle. À la différence des autres langages, un EFSM ne définit pas d'activités ou d'actions internes.

Une transition a la même signification que dans les langages précités. Elle permet lors de la satisfaction des exigences qu'elle définit, de changer d'état. Par exemple dans le cas de la machine de la Figure 1.14, si l'état Off est actif, que l'événement SWITCH est reçu et que le

guard `PowerRunning=true` est évalué à "vrai", alors l'action `lightOn()` est exécutée, l'état `Off` marqué inactif, et l'état `On` marqué actif.

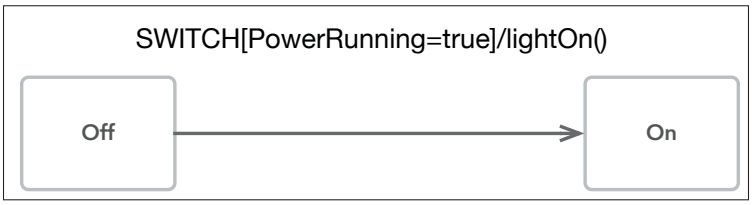


Figure 1.14 États et transitions des EFSM

1.5.2 Définition formelle d'une machine EFSM

Dans le contexte de notre projet, nous adoptons la définition suivante pour nos EFSM, adaptée de Alagar & Periyasamy (2011). Un EFSM est un tuple M = (S, S_0, E, A, T) où :

- S est un ensemble fini non vide d'états atomiques.
- $S_0 \in$ S est l'état initial.
- E est un ensemble fini d'*événements*.
- A un ensemble fini d'actions.
- T est un ensemble fini de transitions. Une transition t $\in$ T se présente sous la forme t $\xrightarrow{e[g]/a}$ t', où e $\subset$ E, g est une condition appelée *guard*, et a $\in$ A est une *action*.

1.5.3 Extension des EFSM pour supporter la communication

A l'instar des machines à états Stateflow et UML, certains EFSM peuvent communiquer. C'est le cas des CEFSM : EFSM communicants. Un système de CEFSM (Bourhfir *et al.*, 2001) se compose d'un couple (EFSM, Channel), constitué d'un ensemble d'EFSM et de *Channels*. Un Channel est un tuple C = (M_{source}, M_{cible}, Signal) dont la définition est la suivante.

- M_{source} étant la machine depuis laquelle le signal est envoyé.

- M_{cible} étant la machine vers laquelle le signal est envoyé.
- *Signal* étant le signal envoyé. Le signal est envoyé par une action, et reçue à travers un événement.

Dans la Figure 1.15, la machine M1 envoie le signal `E2` à travers l'action `SEND(E2)` sur la transition entre les états `A` et `B`. La machine M2, attend cet événement `E2` pour exécuter la transition entre `C` et `D`. On retrouve ici le *channel* C = (M1, M2, E2)

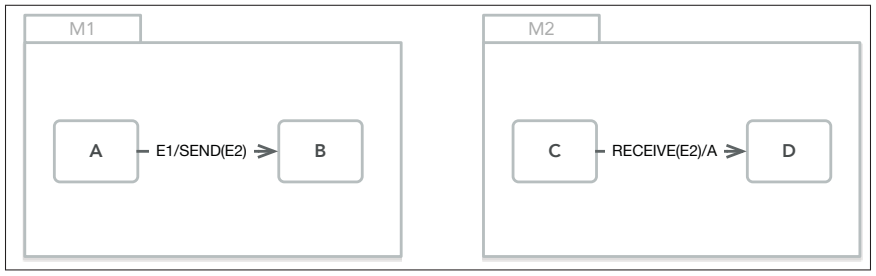


Figure 1.15 Deux EFSM M1 et M2 qui communiquent

1.6 Les Tests Basés sur les Modèles

L'étape de test d'un logiciel, surtout dans le domaine du logiciel des systèmes embarqués critiques, est une étape déterminante et primordiale. C'est pour cela qu'elle peut représenter jusqu'à 50% (Boehm *et al.*, 1981) des ressources utilisées dans la conception du logiciel. Une des approches qui vise à faciliter la phase de tests est l'approche de tests basée sur les modèles. Le travail de Shafique & Labiche (2015) définit les tests basés sur les modèles comme une approche qui utilise le ou les modèles d'un système que l'on doit tester afin d'en extraire des cas de test abstraits. Ces cas de tests abstraits sont ensuite raffinés en cas concrets afin d'être utilisés sur une plateforme spécifique. Les modèles permettent également d'extraire les oracles de tests. Dans la démarche de l'IDM, le test basé sur les modèles est une approche intéressante permettant d'exploiter au maximum la puissance de la modélisation. L'objectif de ce travail est d'obtenir un modèle formel et unificateur supportant la génération de test. Dans le cadre du projet AVIO604 les modèles utilisés pour générer les tests sont des CEFSM. En effet il existe

un nombre élevé de travaux utilisant ce type de formalisme (Li & Wong, 2002; Guglielmo *et al.*, 2011; Kalaji *et al.*, 2011; Bourhfir *et al.*, 2001).

1.6.1 Techniques de génération de tests basées sur les EFSM

Dans un premier temps, afin de générer des cas de tests utilisables, il faut identifier l'ensemble des "chemins faisables" à travers la machine. Un chemin faisable est un ensemble de transitions démarrant de l'état initial S_0 menant à un état S où chaque condition sur les variables de chaque transition est satisfaite. Cela peut impliquer une séquence particulière, dans un ordre particulier.

Ainsi, dans les travaux abordant la génération de tests depuis des EFSM, la première étape consiste à trier les séquences de transitions, afin de ne retenir que les séquences faisables. Pour cela, différentes méthodes sont proposées. Certaines méthodes présupposent certaines propriétés sur les EFSM. Par exemple les travaux de Duale & Uyar (2004) présupposent : 1) l'absence de communication entre les EFSM (impossible donc dans le cadre de l'utilisation de CEFSM), 2) l'absence de fonctions récursives, et 3) le fait que les conditions et les actions soient linéaires.

Duale & Uyar (2004) proposent d'éliminer les séquences infaisables entre S_0 et S, en résolvant les conflits sur les variables dans les actions et dans les conditions. Pour cela ils proposent d'utiliser des matrices qui représentent les effets de chaque action sur les variables à travers la machine pour toutes les paires d'états incluant S. Si dans ces matrices certaines variables ont été modifiées de différentes manières, il y a potentiellement des conflits entre les actions. En utilisant la valeur symbolique (Coward, 1991) de chaque variable à l'état *S*, on détermine la faisabilité de chaque condition impliquant les variables modifiées. Et pour chaque conflit, on sépare la machine en sous-machines éliminant ces conflits.

Une fois que tous les conflits ont été résolus, un EFSM est dit *cohérent* et les techniques de génération de test à partir des EFSM s'appliquent. Il existe de nombreux travaux utilisant des techniques de génération variées (model-checking, utilisation de technique d'optimisation telles que les algorithmes génétiques, etc...). On peut notamment identifier les travaux de Ernits

et al. (2006) utilisant les techniques de model-checking afin d'identifier les chemins faisables et d'obtenir une couverture de test convenable. Les travaux de Kalaji *et al.* (2011) utilisent un algorithme génétique afin d'extraire à la fois les chemins faisables et les cas de tests, en limitant la possible explosion de leur nombre. Enfin pour répondre à cette problématique d'explosion tout en maximisant la couverture du modèle, Di Guglielmo *et al.* (2011) proposent une utilisation à la fois d'exécution symbolique et concrète. L'exécution symbolique va déterminer quelles sont les entrées à injecter dans le modèle lors de l'exécution concrète afin d'obtenir la couverture souhaitée. Cela permet de limiter l'exploration du modèle et donc l'explosion du nombre de cas de test.

Une autre méthode permettant la génération de cas de tests est l'extraction des *graphes de flot de contrôle et de données* depuis les machines EFSM/CEFSM (Li & Wong, 2002). Les travaux de Naik & Tripathy (2008) définissent ces graphes de la façon suivante. Le *graphe de flot de contrôle* est une représentation graphique d'une unité d'un programme. Il permet de visualiser tout les chemins possibles dans un programme. C'est-à-dire toute les séquences de conditions et de déclarations qui peuvent avoir lieu dans notre programme. Le *graphe de flot de données* représente la manipulation des variables à travers le programme. Ainsi une variable peut être soit *définie*, soit *utilisée*. Elle peut être utilisée soit dans une condition, soit dans une action. La détermination des noeuds de définition et d'utilisation pour chaque variable permet de construire le graphe de flot de données pour chacune d'elle. Une fois les graphes construits, il sont exploités selon la couverture de test souhaitée.

1.6.2 Couverture de test

Pour assurer la qualité des tests, il faut analyser leur degrés couverture, c'est-à-dire la partie du code couverte par ces tests. Cela permet de voir si le code est testé de manière adéquate selon le contexte. On peut considérer différents critères de couverture, incluant la couverture de conditions, d'instructions, des fonctions etc. Dans notre cas, par exemple, la norme DO-178 impose que les tests des logiciels aéronautiques de DAL A assurent une couverture Modified Condition / Decision Coverage (MC/DC). La norme définie cette couverture de la façon sui-

vante : chaque décision doit avoir pris toutes les valeurs possibles, et chaque condition de cette décision doit avoir été évaluée en ayant un impact sur la décision.

Considérons la décision suivante :

$$if((A||B)\&\&C)\{...\}else\{...\} \tag{1.1}$$

Il faudra trouver un jeu de tests permettant de vérifier les conditions suivantes :

A = faux, B = faux, C = vrai : La décision est évaluée à faux

A = faux, B = vrai, C = vrai : La décision est évaluée à vraie

A = faux, B = vrai, C = faux : La décision est évaluée à faux

A = vrai, B = faux, C = vrai : La décision est évaluée à vraie

Les travaux de Frankl & Weyuker (1988) présentent les critères de couverture plus communément utilisés en ce qui concerne les tests sur les variables. Ces critères s'appliquent sur les graphes de flot de données. En identifiant les étapes auxquelles une variable est définie et utilisée, par une condition ou une action et en considérant les séquences de définition/d'utilisation, certains critères comme *All-Defs* peuvent être vérifiés. Le critère All-Defs définit un chemin (une séquence états-transitions) au cours duquel la variable n'est pas redéfinie entre le noeud de départ et celui d'arrivée, et la variable est utilisée au noeud d'arrivée. Évidemment, ces différents critères sont à utiliser selon la couverture souhaitée et selon le contexte.

CHAPITRE 2

REVUE DE LITTÉRATURE

2.1 Les transformations de machines à états

Cette revue a été menée dans l'objectif d'identifier et d'analyser les techniques existantes de transformation de machines à états. Afin de réduire le champ d'étude aux aspects qui sont reliés à notre travail, la revue se limite aux techniques suivantes :

- Nous nous sommes concentrés sur les transformations de machines à états UML et Stateflow.
- Le modèle cible est, dans la majorité des travaux, un formalisme de machines à états étendues (ou un formalisme très proche).
- L'objectif de la plupart de ces travaux est de fournir un modèle cible formel utilisable à des fins de vérification.

2.1.1 Les transformations de modèles UML

Différents travaux se sont intéressés à la transformation des machines à états UML vers des formalismes plus simples à analyser et exploiter. L'article de Devroey *et al.* (2015) fait un état de l'art concernant les techniques de suppression de la hiérarchie des machines à états, que nous appellerons *Flattening*. Cet article recense une dizaine de travaux dont le modèle d'entrée de la transformation correspond à une machine à états UML. Ces travaux ont différents objectifs et utilisent différents formalismes cibles. Certains visent la génération de code, d'autres les tests basés sur les modèles, ou encore la vérification de modèles. Mais l'objectif principal est bien souvent d'obtenir un modèle plus formel sur lequel des analyses pourront être menées.

Les travaux de Kim *et al.* (1999) et Hong *et al.* (2000), abordent la transformation de machines à états UML vers des machines à états finies étendues (EFSM) en définissant un ensemble de

règles présentées de manière formelle. L'objectif de la transformation dans ces travaux est la génération de tests basés sur les modèles. Ces travaux présentent le concept de *configuration*. Une configuration correspond à l'ensemble des états dans lesquels le système se trouve à un instant donné. Par exemple, pour un état composite actif contenant deux régions parallèles comme l'état `ON` dans la Figure 2.1, trois états sont actifs en même temps : l'état composite (`ON`), et un état dans chacune des régions (e.g.,`IDLE` dans la région `COFFEE` et `EMPTY` dans la région `MONEY`). L'une des configurations obtenues sera donc `ON, COFFEE, IDLE, MONEY, EMPTY`.

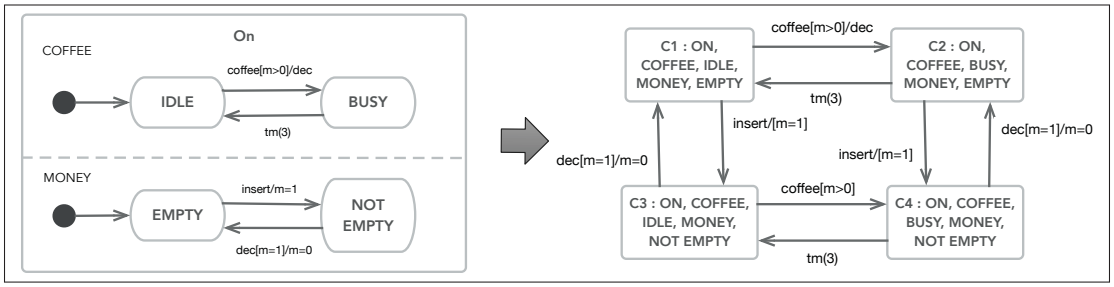


Figure 2.1 Les configurations, adapté de Kim *et al.* (1999)

Le travail de Hong *et al.* (2000) propose une optimisation du nombre de transitions produites lors de la transformation, et défini le concept de *Normal Form Specification*, décrivant un EFSM préservant le comportement de la machine à état UML dont il est issu. Ce travail propose de conserver cette forme normale tout en réduisant le nombre de transitions, éliminant celles introduisant des conflits, notamment en vérifiant manuellement les enchaînements d'événements possibles et impossibles et les dépendances entre les variables. Dans la Figure 2.1, on remarque qu'un conflit apparaît : la condition de la transition de la configuration C1 vers C2 impose que `m>0` pour être exécutée. Hors la variable `m` est seulement modifiée par la transition de `C1` vers `C3`, donc la transition de `C1` vers `C2` ne peut jamais être empruntée. Néanmoins les techniques proposées dans ces travaux s'appliquent à l'ancienne version de UML (1.1), et n'aborde pas la transformation de certains concepts UML tels que les pseudo-états. De plus ces techniques obligent la personne modélisant le système à intervenir dans la transformation pour l'optimisation du nombre de transitions, ce qui limite l'automatisation.

Des travaux tels que ceux de Kuske (2001) ou plus récemment de Pradhan *et al.* (2019) utilisent les techniques de transformation de graphes afin d'obtenir un modèle aisément utilisable pour la génération de test. Le travail de Pradhan *et al.* (2019) génère simplement un "graphe d'accessibilité" en exécutant du code généré depuis le modèle. Cette méthode ne peut cependant s'appliquer qu'à des machines à états UML simple, sans hiérarchie, sous peine de devenir très complexe.

Le travail de Kuske (2001) s'attaque à la hiérarchie des machines à états et présente la formalisation de la structure d'une machine à états UML sous la forme d'un graphe. Pour chaque élément structurel de la machine (e.g., état composite, état simple), un noeud particulier est défini. Une unité de transformation appliquée à ce noeud permet de l'aplatir. Chaque unité se compose d'un ensemble de règles, spécifiant chacune l'élément à transformer et l'élément cible de la transformation, ainsi que les conditions de la transformation. La Figure 2.2 présente la règle de transformation de graphe *r2*, permettant de raffiner de façon récursive un noeud *cs* correspondant à un état composite `T` contenu dans une machine à états `STM` en un noeud correspondant à un état simple séquentiel. Il contient un noeud *SorC*, contenant le reste de la hiérarchie sous-jacente. Le booléen *isC* permet de préciser si le noeud composite est un And-state (isC = T) ou un Or-State (isC = F). En appliquant ces règles dans un ordre précis à chaque niveau de hiérarchie on obtient finalement une machine plate.

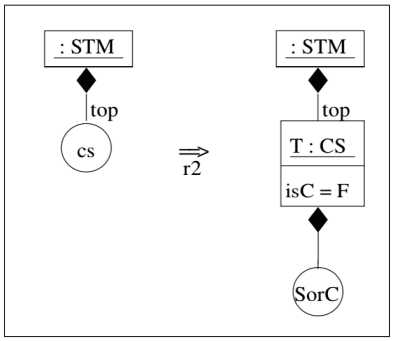


Figure 2.2 Règle de transformation d'un état composite, extrait de Kuske (2001)

Dans le même esprit, le travail de Minas & Hoffmann (2008) présente un algorithme permettant de supprimer la hiérarchie des machines à états UML, en utilisant la technique de transformation de graphes *Double Pushout Graph Rewriting*. L'Algorithme 2.1 propose les étapes suivante : 1) il transforme les And-states en Or-states, et 2) il aplatit les Or-states.

Algorithme 2.1 Algorithme de Flattening, extrait de Minas & Hoffmann (2008)

```
1  while the graph contains And State or Or State nodes do
2      //remove a bottom level and-state
3      if possible mark_bottom_level_and;
4      for all matches and_create_cross_product;
5      if possible and_create_init;
6      for all matches and_create_trans;
7      as long as possible and_clean_up;
8      //remove all or-state
9      as long as possible or_move_outgoing_trans;
10     as long as possible or_remove;
11 end
```

Les lignes 3 à 7 opèrent sur les And-states. D'abord, la fonction *mark_bottom_level_and* identifie le niveau le plus profond du And-state considéré, car l'algorithme s'applique du niveau le plus profond vers le plus haut. Ensuite la fonction *and_create_cross_product* permet de réaliser le produit cartésien des régions contenue par les noeud états composite, ce qui s'apparente aux configurations que le travail de Kim *et al.* (1999) propose. La fonction *and_create_init* permet de créer le nouveau noeud pseudo-état initial qui servira de point de départ pour les sous états du nouvel Or-state qui émanera de cette transformation. La fonction *and_create_trans* s'occupe de la gestion des transitions internes, en reportant toute transition pré-existante entre deux états contenus dans la configuration, tandis que la fonction *and_clean_up* permet de remplacer le noeud Or-state par un noeud And-state. Les deux dernières fonctions (lignes 9 et 10) concernent le flattening des Or-states. La première *or_move_outgoing_trans* permet de connecter les transitions qui étaient précédemment connectées au Or-state aux sous états, tandis que *or_remove* supprime le Or-state. La Figure 2.3 présente l'algorithme appliqué sur le And-state `B` dans la figure (a). La figure (b) présente le résultat obtenu à la fin de l'application

de la première étape, et la figure (c) le résultat obtenu à la fin de l'application de l'algorithme. Cet algorithme ne permet cependant pas de traiter les pseudo-états ou les activités internes.

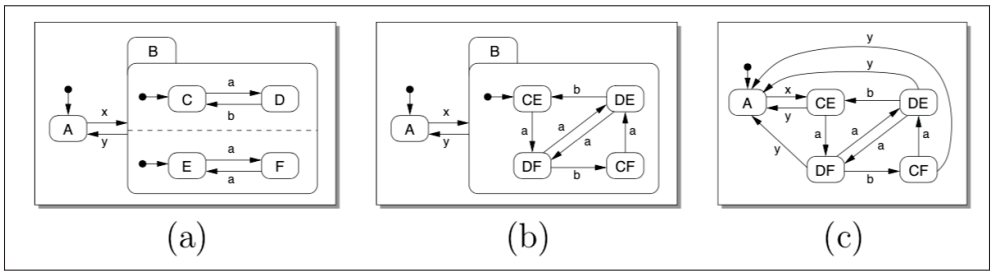


Figure 2.3 L'algorithme appliqué à une machine à états UML, extrait de Minas & Hoffmann (2008)

D'autres travaux proposent de transformer les concepts des machines à états UML vers d'autres formalismes, par exemple vers du code (Niaz & Tanaka, 2004), ou encore des réseaux de Petri (Kumar *et al.*, 2017). Les travaux de André *et al.* (2014) et Yao & Shatz (2006) proposent une transformation vers des réseaux de Petri colorés étendus (ECPN). Ces réseaux présentent deux types de nœuds : des nœuds états et des nœuds transitions, liés entre eux par des arcs. L'extension fournie par l'utilisation de couleurs permet de typer les nœuds. Les travaux de Yao & Shatz (2006) proposent un algorithme de transformation en deux étapes : 1) la machine est transformée en un modèle intermédiaire plat séquentiel appelé *Extended Hierarchical Automata*, et ensuite 2) ce modèle intermédiaire est tranformé en ECPN. La Figure 2.4 présente les deux étapes de l'algorithme appliqué sur la machine de la figure (a). D'abord, on transforme la machine en Extended Hierarchical Automata en isolant la partie composite de la machine du reste. A cette étape les transitions ne peuvent pas traverser un niveau de hiérarchie, donc dans cet exemple pour conserver les transitions telles que la transition `T3` sortante de l'état `C` vers l'état `H` dans la figure (b), on crée un ensemble nommé *source restriction* gardant l'information. De façon similaire, pour conserver l'information sur une transition telle que `T8` entrante dans l'état `B`, on crée un ensemble nommé *target determination*.

Dans la seconde étape de l'algorithme, la partie composite est transformée en une *composite place*. Celle-ci contient des noeuds états obtenus de la même façon que l'on obtient les confi-

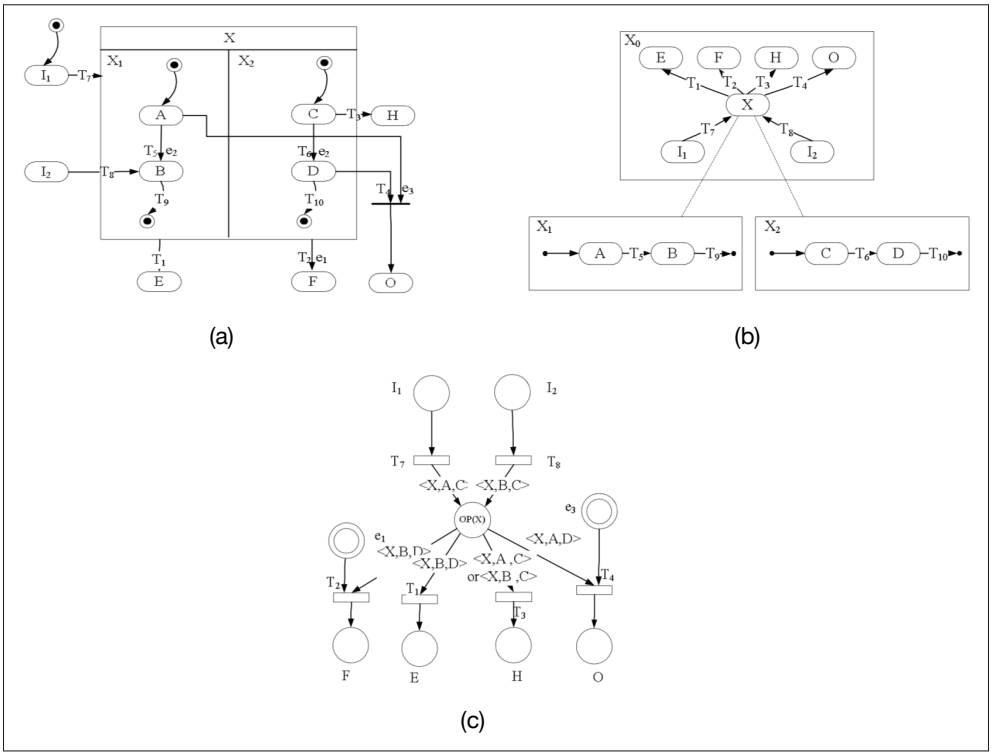


Figure 2.4 L'algorithme de transformation des machines à états UML en réseaux de Petri colorés étendus, extrait de Minas & Hoffmann (2008)

gurations dans les exemples précédents. Dans la figure (c), on a *OP(X)*=<*X,A,C*>, <*X,A,D*>, <*X,B,C*>, <*X,B,D*>. La connexion avec les transitions externes se fait, d'une part avec les informations contenues dans les ensemble source restriction et target determination, et d'autre part en consultant quel élément contenu dans OP(X) contient les états précédemment liés aux pseudo-états initiaux (ici <X,A,C>). Cependant ces travaux n'abordent pas la transformation des pseudo-états ou des activités internes, et n'explique pas comment sont gérées les transitions internes des composite places.

2.1.2 Les transformations de modèles Stateflow

Certains travaux se sont intéressés à transformer les machines Stateflow vers des formalismes plus simples à traiter et analyser. Pour de nombreux travaux, l'objectif est d'obtenir un modèle

utilisable pour de la vérification ou de la génération de test. Les langages cibles (e.g., code, automata) et les techniques (e.g., transformation de graphes, manipulation directe de modèles) sont divers.

Le travail de Li & Kumar (2011) présente par exemple une transformation de Stateflow vers un formalisme proche de celui que nous avons adopté. Ce formalisme, *Input/Output Extended Finite Automata* (I/O-EFA), repose sur le principe de *location*, pouvant être initiale, normale ou finale, et sur des transitions contenant des événements entrants et sortants, des conditions et des actions. Comme pour les EFSM, l'existence d'une hiérarchie n'est pas permise. Ce travail présente un algorithme de transformation du langage Stateflow vers I/O-EFA en deux principales étapes : 1) les actions internes dans les états sont transformées, et 2) les états composites (Or et And) sont aplatis. La première étape traite les états en tenant compte des actions *entry*, *during* et *exit* qu'ils peuvent contenir. Le modèle I/O-EFA résultant pour un état *s* est présenté dans la Figure 2.5. Un état *s* est découpé en trois locations : initiale (l_0), normale (l_i) et finale (l_m). Les transitions entre l_0 et l_i permettent de traduire le comportement des actions internes entry `en` et during `du`, tandis que la transition entre l_i et l_m traduit l'action exit `ex`. La condition de valeur sur la variable d_a permet de déterminer si l'état est actif(d_a=0)/inactif(d_a=1)/nouvellement actif(d_a=2), tandis que la condition sur d_l nous donne le sens d'exécution, si l'état s est un état composite. On détermine ainsi si l'exécution de cet état amène à changer de niveau hiérarchique vers le bas(d_l=-1)/vers le haut(d_l=1) ou si il reste au même niveau(d_l=0).

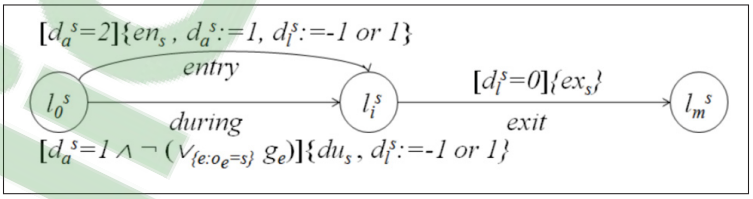


Figure 2.5 I/O-EFA correspondant à un état Stateflow, extrait de Li & Kumar (2011)

La hiérarchie est ensuite traitée selon si l'état composite est un Or-state ou un And-state. Pour le Or-state, la location normale du super état est connectée à l'ensemble de ses sous états, qui

auront leurs trois locations actives de manière séquentielle. Tandis que pour un And-state, les locations des sous états sont actives en parallèle pour un état par région. Bien qu'offrant une solution au traitement des actions internes, ces travaux n'abordent pas l'ensemble des concepts existants dans le langage (par exemple les jonctions) et le formalisme conduit à un modèle relativement complexe et pas facile à utiliser comme support pour la vérification.

Agrawal *et al.* (2004) proposent une transformation d'un sous ensemble de Simulink et Stateflow vers un formalisme appelé *Hybrid Automata*. Ce formalisme spécifie un ensemble fini d'états, où chaque état définit une équation algébrique associée. L'algorithme est implémenté avec *GReAT*, un langage basé sur la grammaire des graphes. L'algorithme de transformation proposé se découpe de la façon suivante : 1) la hiérarchie est supprimée par une fonction (non présentée), 2) les états sont découpés de façon à obtenir des états "discrets", ne contenant plus aucune variable. Enfin, 3) les états inatteignables sont retirés en parcourant le modèle. Ce travail se focalise sur la partie 2) de l'algorithme : la découpe des états selon la valeur des variables qu'ils contiennent. La Figure 2.6 présente les étapes de cette partie de l'algorithme, appliquée à un exemple contenant des états contenant des actions internes entry. Le diagramme Stateflow dans cet exemple est connecté à un diagramme Simulink en entrée. En simulant le diagramme Simulink, il est possible de réaliser des inférences sur les variables impliquées dans l'action interne. Dans l'exemple de la Figure 2.6, les inférences sur le modèle permettent de déduire que la variable `V3` dans l'état `Low`, et les variable `V1` et `V3` dans l'état `Too_High` peuvent prendre les valeurs 0 ou 1. La variable V2 dans l'état *High* ne peut prendre que la valeur 1. Suite à cette étape d'inférence, on découpe les états selon les différentes valeurs possibles des variables, on conservant les transitions. Maintenant que les variables ont des valeurs constantes, on peut obtenir un Hybrid Automata, ne contenant que des états, équivalents à l'ensemble des cas possibles pour les variables impliquées dans leurs actions internes, et des transitions. Ainsi, ce travail se focalise seulement sur la partie de découpage des états, sans présenter de technique de flattening, ou de gestion des jonctions dans Stateflow.

Le mémoire de Kratochvlová (2015) présente SF2DVE, une transformation de Stateflow vers un langage utilisable par le *model checker DiVinE* (Barnat *et al.*, 2006). La première étape de la

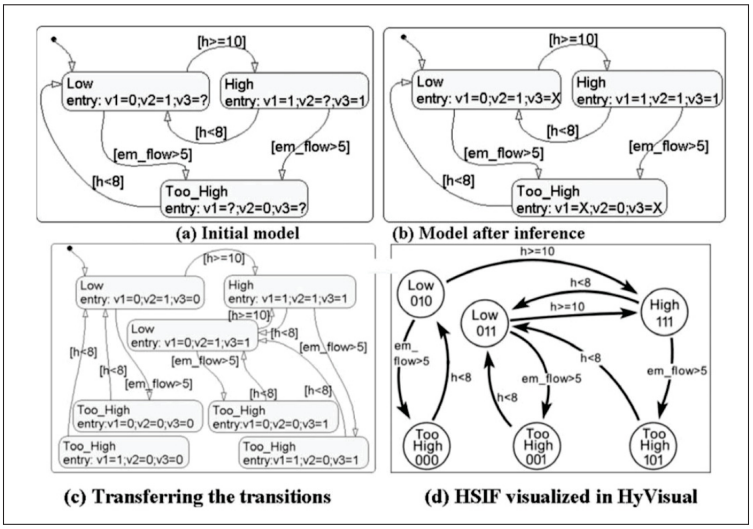


Figure 2.6 Étapes du découpage des états, extrait de Agrawal *et al.* (2004)

transformation est un algorithme permettant de transformer les éléments du langage Stateflow complexe à traiter de manière formelle (e.g., hiérarchie, actions internes). La transformation des états composites présentée se propose de traiter les Or-state de façon simple. Les états composites sont éliminés et on ne garde que les sous états. Pour ne pas perdre les actions internes des super états, celles-ci sont transposés sur les transitions de la manière présentée dans la Figure 2.7. Une action during contenue dans le super état telle que `duA` est placée sur les transitions sortantes du sous état dans la condition action. Le fait de la placer dans les conditions actions permet de s'assurer qu'elle s'exécute avant l'action interne exit qui est placée dans l'action. Ensuite les actions internes exit de l'état composite et du sous état source de la transition (`exA` et `exC`) sont placés dans l'action de la transition, de la même manière que les action interne entry de l'état composite et du sous état source de la transition (`enB` et `enD`). Le résultat de l'algorithme est donc une machine à états Stateflow plate, ne contenant que des états sans actions internes et des transitions.

Le résultat de l'algorithme est ensuite représenté au format XML, pour obtenir un fichier compatible avec DiVinE. Cependant les concepts de Stateflow ne sont pas tous traités, et l'ordre d'exécution n'est pas forcément conservé, notamment lors de la transformation des actions

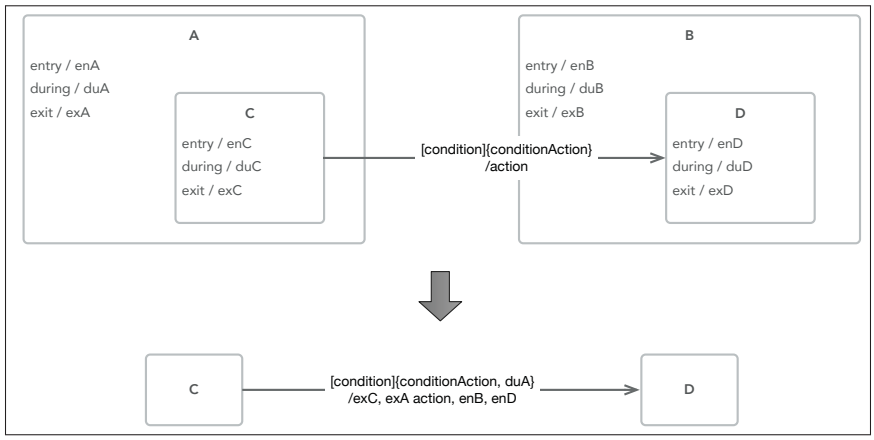


Figure 2.7 Transformation des actions internes, adapté de Kratochvlová (2015)

internes qui sont simplement transposées sur les transitions entrantes et sortantes dans l'état contenant les actions. Le flattening n'est présenté que pour le cas des Or-states.

Le travail de Scaife *et al.* (2004) propose une transformation d'un sous ensemble de Stateflow vers du code *Lustre*. Le langage Lustre est utilisé pour faire la vérification de modèles. Les auteurs définissent un sous ensemble de Stateflow comme un ensemble "Safe", dans le sens ou le sous-ensemble évite les erreurs possibles et les comportements imprévus. Par exemple le cas de *backtracking*, où une suite de jonctions peut amener un retour à l'état source, est éludé. L'algorithme de transformation vers du code Lustre propose par exemple de traduire les états comme des booléens : un booléen de valeur vrai correspond à un état actif, un booléen de valeur faux correspond à un état inactif. Ce travail propose de traduire les actions internes de la manière suivante : si un état devient actif (booléen de valeur vrai) on exécute son action d'entrée en appelant une fonction dans le code, puis on déclenche une fonction correspondante à son action during. Dès que l'état devient inactif (booléen de valeur faux), on déclenche une fonction équivalente à son action exit. De cette manière, le comportement de la machine Stateflow est traduit précisément dans le code. Il est intéressant de noter que les erreurs dans la traduction du comportement peuvent se propager dans une transformation d'un formalisme vers un autre, d'où l'intérêt de définir des règles de modélisation limitant les erreurs possibles.

Toujours dans l'objectif d'obtenir du code pour faire de la vérification formelle, les travaux de Zou *et al.* (2015) proposent une transformation de Simulink/Stateflow vers Hybrid Communicating Sequential Process (HCSP). HCSP permet de décrire des systèmes concurrents hybrides (comme Simulink/Stateflow). Ce travail permet de vérifier un modèle Simulink/Stateflow en tenant compte des constructions particulières (e.g., hiérarchie, utilisation de noeud d'histoire). Par exemple, un noeud d'histoire placé dans un Or-state est traduit comme une variable *history*, qui contient le dernier sous état actif lorsque l'état composite est devenu inactif. L'utilisation de HCSP dans ce contexte permet de retenir dans le modèle cible un aspect important des diagrammes Stateflow : la communication inter et intra diagrammes.

2.2 Discussion

À la lumière de la revue de littérature réalisée, nous avons constaté qu'il y a très peu d'approches traitant la transformation des machines Stateflow comparé aux machines à états UML. Ceci peut s'expliquer par le fait que UML et les outils le supportant sont accessibles aux chercheurs et aux praticiens. Les travaux étudiés génèrent différentes représentations formelles avec des langages divers. Toutefois, la majorité de ces travaux ne traitent pas les versions actuelles des langages UML et Stateflow.

La majorité des travaux se focalisent sur le processus de flattening de la machine à états. Dans le cas d'UML, les travaux utilisent relativement la même technique basée sur l'identification des configurations. Cependant, l'ensemble des travaux étudiés offrent une couverture partielle des langages UML et Stateflow. En effet, peu de travaux traitent certains concepts tels que les activités internes et les pseudo-états dans le cas d'UML, et les jonctions dans le cas de Stateflow.

Finalement nous avons observé que peu de travaux offrent un outil implémentant l'approche proposée.

En se basant sur ces observations, nous proposons dans ce mémoire une approche satisfaisant les exigences suivantes :

- Offrir une transformation couvrant l'ensemble des concepts de chacun des langages ;
- Offrir une transformation qui préserve le comportement des machines à états ;
- Offrir une transformation générique, c'est à dire pouvant s'appliquer à n'importe quel formalisme de machine à états ;
- Définir un ensemble de bonnes pratiques de modélisation pour éviter les erreurs ;
- Et fournir une implémentation de cette approche dans un outil open source.

CHAPITRE 3

APPROCHE

Dans ce chapitre, nous introduisons notre approche générique, appelée STEF (State machine To EFsm), pour transformer des machines à états vers un modèle formel plus simple supportant la vérification. Nous présentons ensuite l'application de l'approche aux machines à états UML et Stateflow.

3.1 Approche proposée de transformation

Nous avons analysé un ensemble de formalismes de machines à états incluant UML, Stateflow et Scade (Ansys, 2018). Cette analyse nous a permis d'identifier les concepts récurrents dans les différents formalismes et qui ne sont pas présents dans le formalisme cible (EFSM). Elle nous a aussi permis d'identifier les différentes étapes nécessaires pour générer un EFSM à partir d'une machine à états. Nous avons donc construit une approche générale de transformation pouvant s'appliquer à n'importe quel formalisme de machines à états. Comme illustré dans la Figure 3.1, notre approche comprend deux processus : analyse et transformation.

Le processus d'analyse est appliqué une seule fois pour tout nouveau formalisme de machines à états. Il consiste à :

1. Identifier les concepts de base du langage (états et transitions) et leurs sémantiques, principalement la sémantique d'exécution et les règles d'utilisation.
2. Identifier les autres concepts (autre que les simples états et transitions) ainsi que leurs sémantiques. De la même manière, il faut comprendre les règles d'utilisation et analyser les exemples fournis.
3. Identifier les concepts hiérarchiques particuliers autorisés par le langage (e.g., hiérarchie, concurrence), leurs sémantiques ainsi que les recommandations d'usage si elles sont données par la spécification.

Le processus de transformation est composé de trois étapes :

1. Pré-traitement (*Preprocessing*) : cette étape permet de traiter l'ensemble des concepts qui n'existent pas dans le formalisme cible. Ces concepts sont l'ensemble des concepts autre que les états et les transitions. Des exemples de tels concepts sont les pseudo-états et les activités internes dans UML, ainsi que les jonctions et les actions internes dans Stateflow.
2. Aplatissement (*Flattening*) : afin de rendre les machines plus expressives et compréhensibles, les différents formalismes sources permettent de définir les hiérarchies d'états (i.e., composition) et des états parallèles. Cette étape permet de supprimer la hiérarchie dans une machine à états pré-traitée tout en préservant le comportement de la machine à états.
3. Mise en correspondance (*Mapping*) : cette étape fait la correspondance entre la machine à états source (pré-traitée et aplatie) et une machine à états EFSM.

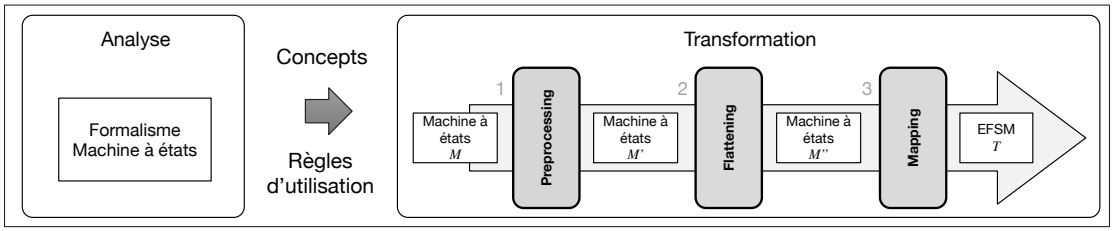


Figure 3.1 Approche générale proposée

Chaque étape est appliquée au niveau le plus bas de la hiérarchie des états puis aux niveaux supérieurs. Autrement dit, une étape est appliquée aux états les plus atomiques et ensuite propagée vers les états composites selon l'ordre de composition.

3.2 L'approche appliquée aux machines à états UML

Dans cette section, nous présentons le sommaire des résultats du processus d'analyse appliqué à UML, suivi des règles de transformation de chacune des étapes du processus de transformation dans le cas des machines à états UML.

3.2.1 Résumé des résultats du processus d'analyse des machines à états UML

Suite à l'analyse du formalisme des machines à états UML, nous avons identifié les concepts suivants (ces concepts ont été présentés dans la Section 1.3) :

1. Les états contiennent optionnellement des activités internes, et sont soit actifs soit inactifs au cours de la durée de vie de la machine. Les transitions contiennent optionnellement un événement, une condition et une action.
2. Le formalisme UML introduit le concept de pseudo-état, exprimant certains comportements particuliers.
3. Enfin, les états dans UML peuvent être composites ou simples. Il existe deux catégories d'états composites : Or-state (séquentiel) ou And-state (concurrent).

3.2.2 Preprocessing

Selon le résultat de l'analyse, nous avons identifié deux grandes familles de concepts différents des états et des transitions : les *pseudo-états* et les *activités internes*.

3.2.2.1 Les pseudo-états

Cette étape s'attache à remplacer les pseudo-états (excepté le pseudo-état initial qui sera remplacé dans les étapes suivantes afin d'identifier le point de départ de chaque région) par des concepts simples des machines à états UML tout en préservant le comportement de la machine à états traitée. La Figure 3.2 présente un ensemble de pseudo-états traités par notre approche. On y trouve, par exemple, des points d'entrées et de sorties dans l'état composite E, un noeud de jonction et un noeud de décision.

Les pseudo-états point d'entrée/point de sortie Les premiers éléments qu'il est nécessaire de traduire sont les points d'entrées et de sorties. Ceux-là se retrouvent à l'interface d'une machine à état ou d'un état composite. Les points d'entrées sont une alternative à un pseudo-état initial pour pénétrer une région en ciblant un sous état particulier. De même les points de

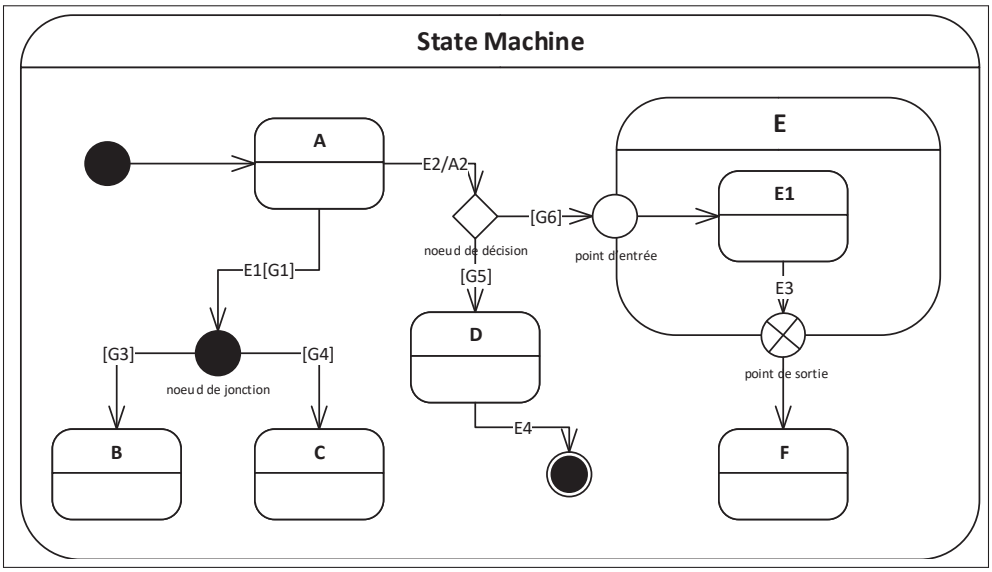


Figure 3.2 Machine à états contenant des pseudo-états

sorties permettent de définir une sortie particulière d'un état composite. Ces pseudo-états sont une pure abstraction graphique, ils permettent de facilement identifier les entrées et sorties particulières d'un état composite. La construction équivalente proposée est simplement une élimination de l'abstraction : connecter la source de la transition entrante dans le point (entrée ou sortie) avec la cible de la transition sortante. Seule la transition entrante peut posséder des items (événement, condition et action).

La Figure 3.3 présente le résultat de la transformation de la machine de la Figure 3.2. Par exemple, l'état `E1` est maintenant directement lié à l'état `F`. Lors de ces transformations, l'ordre d'exécution des états et des transitions est bien conservé. En effet, ici l'événement et la condition de la transition entrante seront d'abord évalués, puis l'action exécutée. Ensuite si l'état composite cible contient une activité entry, elle sera exécutée. Il faut ajouter que, même si la spécification reste vague sur ce point, il ne faut placer des items que sur la transition entrante dans les points d'entrées et de sorties.

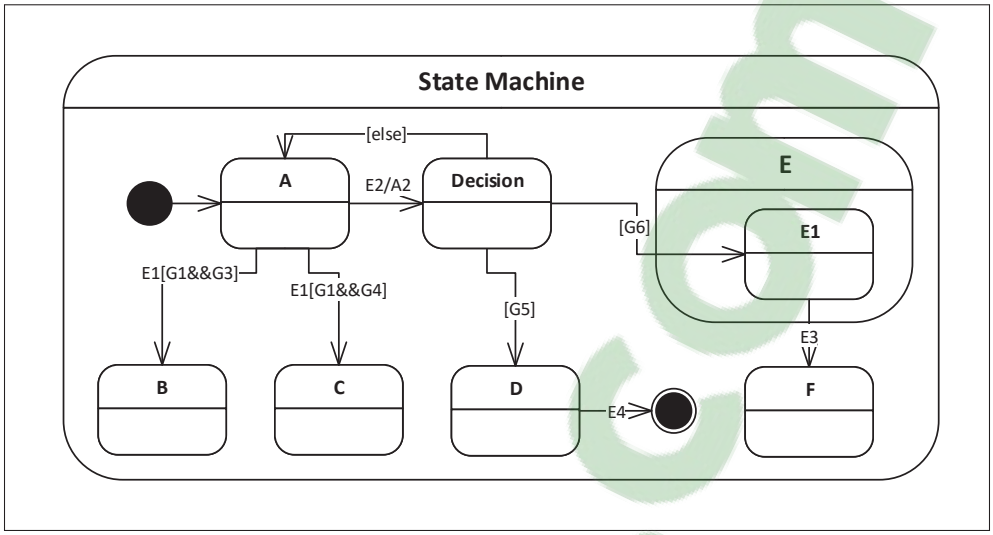


Figure 3.3 Machine à états sans pseudo-états

Les pseudo-états nœud de jonction/nœud de décision

Ces concepts servent à connecter les transitions entre les états, afin de fusionner un ensemble de transitions ou de démarrer un ensemble de transitions à partir d'une transition. À titre d'exemple, on peut s'en servir pour fusionner plusieurs transitions entrantes en une seule sortante et inversement. La différence entre ces deux pseudo-états réside dans l'évaluation des items de leurs transitions sortantes. Le nœud de décision, contrairement au nœud de jonction, permet d'évaluer toutes les transitions sortantes du nœud de façon dynamique. Aussi une action sur les variables réalisée en amont du nœud de décision pourrait avoir un impact sur les variables qui seront évaluées dans les conditions des transitions sortantes du noeud. Ceci n'est pas vrai pour le point de jonction où l'évaluation est statique, c'est à dire que les actions sur les transitions entrantes dans le noeud ne seront pas exécutées avant d'évaluer les conditions des transitions sortantes. Il est donc primordial de les transformer différemment, et en ce qui concerne la transformation du nœud de décision, il est nécessaire d'introduire un état entre les transitions entrantes et sortantes. Cela permet de s'assurer que l'action en amont est bien exécutée avant l'évaluation des conditions des transitions sortantes. Il est intéressant de rappeler que la spécification UML ne permet pas de placer un événement sur les transitions sortantes du noeud de jonction ou de décision.

La transformation proposée pour le nœud jonction est la suivante : le nœud est retiré, et on connecte chaque transition en entrée avec chaque transition en sortie. Ainsi, pour *n* transitions entrantes et *m* transitions sortantes on obtient *n*m* transitions. Chaque état étant lié au nœud en entrée se retrouve lié par une transition à chaque état en sortie du nœud. L'événement, la condition et l'action d'une transition sont calculés à partir de ceux des transitions qui ont été fusionnées pour former cette transition. En effet, ici l'évaluation est statique, donc l'évaluation est réalisée au même instant pour les transitions entrantes et sortantes.

Pour le nœud de décision il faut introduire un état pour remplacer le pseudo-état de décision. Les transitions entrantes ciblent maintenant ce nouvel état et les sortantes possèdent cet état comme source. La Figure 3.3 présente le résultat de la transformation des pseudo-états nœud de jonction et nœud de décision. Le noeud de décision sera remplacé par un état nommé `Décision`. Pour éviter d'être bloqué dans ce nouvel état dans le cas où aucune des conditions sur les transitions sortantes n'est satisfaite, on crée également une transition de l'état Décision vers l'état dont la transition va vers le noeud de décision, contenant la condition *else*. Cette condition étant vraie quand aucune des conditions des transitions sortantes du noeud de décision n'est vraie.

L'Algorithme 3.1 résume le traitement des pseudo-états réalisé lors de l'étape de preprocessing.

3.2.2.2 Les activités internes

Les activités *entry*, *do*, *exit* que l'on retrouve dans les états des machines UML (par exemple dans l'état `B` de la Figure 3.4) doivent être traduits. Pour préserver le comportement tout en utilisant une construction existante dans les EFSM, la solution proposée par l'article de Kim *et al.* (1999) est de transposer ces activités vers les actions des transitions entrantes (pour les activités *entry*) et des transitions sortantes (pour les *exit*). Cependant cette méthode ne permet pas de conserver l'ordre d'exécution des actions. Par exemple, dans la Figure 3.4, on doit s'assurer que l'action `A1` dans la transition de l'état `A` vers l'état `B` s'exécute avant l'activité d'entrée `en()` de l'état `B`.

Algorithme 3.1 Preprocessing : transformation des pseudo-états

```
1  Entrée : Machine à états UML 𝓜 avec pseudo-états
2  Sortie : Machine à états UML 𝓜 sans pseudo-états
3  for all point_entree p_e ∈ 𝓜 do
4  |   p_e.transitionEntrante().cible ← p_e.transitionSortante().cible
5  end
6  for all point_sortie p_s ∈ 𝓜 do
7  |   p_s.transitionEntrante().cible ← p_s.transitionSortante().cible
8  end
9  for all noeud_jonction j_c ∈ 𝓜 do
10 |   for all transition t_c where cible = j_c do
11 |   |   for all transition t_s where source = j_c do
12 |   |   |   t_i = creer_transition()
13 |   |   |   t_i.source ← t_c.source
14 |   |   |   t_i.cible ← t_s.cible
15 |   |   |   t_i.items ← t_s.items + t_c.items
16 |   |   end
17 |   end
18 end
19 for all noeud_decision d_e ∈ 𝓜 do
20 |   e_i = creer_etat()
21 |   d_e.transitionEntrantes().cible ← e_i
22 |   d_e.transitionSortantes().source ← e_i
23 |   for all transition t where cible = d_e do
24 |   |   t_i = creer_transition()
25 |   |   t_i.source ← t.source
26 |   |   t_i.cible ← d_e
27 |   |   t_i.condition ← else
28 |   end
29 end
```

Notre solution pour les activités entry et exit est de décomposer l'état les contenant. D'abord, pour une activité entry un état sera créé. Les transitions qui ciblent l'état contenant l'activité d'entrée vont alors cibler l'état nouvellement créé (les items de la transition ne changent pas). Une transition entre ce nouvel état et l'état qui contenait l'activité *entry* est créée, avec une condition de valeur *true* et une action équivalente à l'activité *entry*. La condition permet de s'assurer que la transition est empruntée. La Figure 3.5 illustre le résultat de cette transformation appliquée à la machine de le Figure 3.4. On peut voir le nouvel état appelé `B-entry`. La

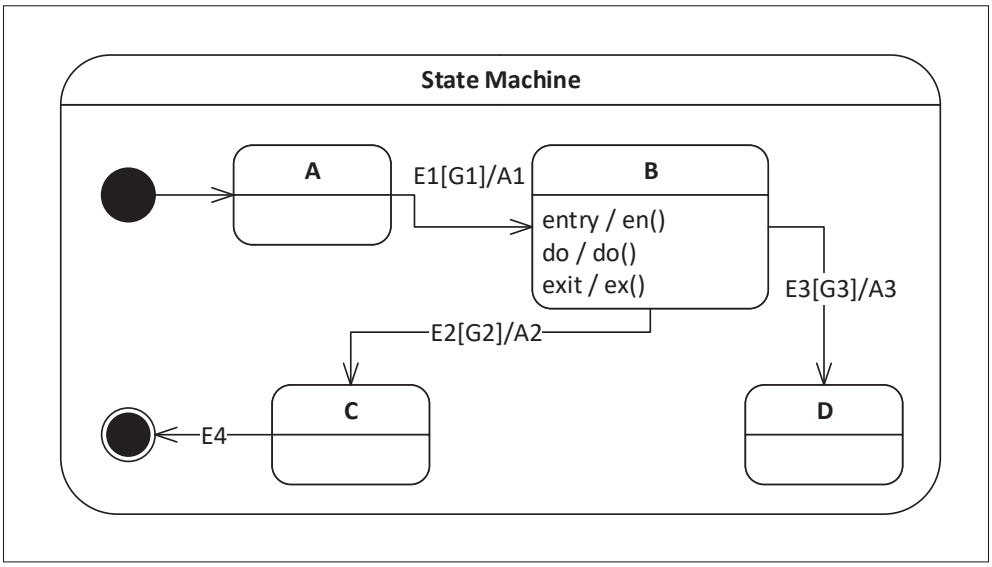


Figure 3.4 Machine à états avec état à activités internes

transition de A vers B se fait maintenant de A vers B-entry. Une fois que B-entry est atteint, une transition est déclenchée de B-entry vers B où l'action en() est exécutée.

Pour les activités *exit*, la méthode est similaire à celle adoptée pour l'activité *entry*. En revanche, cette fois-ci un nouvel état sera créé pour chaque transition sortante. Les transitions sortantes de l'état contenant l'activité de sortie ciblent maintenant les nouveaux états, et l'action de ces transitions prend maintenant la valeur de l'activité de sortie. Pour chacun de ces nouveaux états, une transition de ce nouvel état vers l'état cible de la transition sortante de l'état qui contenait l'activité est créée. Celle-ci possède une condition de valeur *true* et une action équivalente à l'action de la transition sortante de l'état contenant l'activité. Par exemple, la transition de l'état B vers l'état C dans la Figure 3.4 a été remplacée par la transition de l'état B vers le nouvel état B-exit1 ayant une action exit ex() suivie de la transition de B-exit1 vers C. De la même façon, la transition de B vers D a été remplacée par la transition de B vers le nouvel état B-exit2 suivie de la transition de B-exit2 vers D.

Pour les activités internes *do*, nous utiliserons la méthode proposée dans les travaux de Kim *et al.* (1999). À savoir une activité *do* interne à un état sera transformée en une transition de l'état qui la contient vers lui-même, avec comme événement l'événement *null*, une condition

de valeur *true* et une action correspondante à l'activité interne do. Ici un événement *null* signifie qu'aucun des événements déclenchant une inactivation de l'état n'arrive durant la période d'exécution de cette transition. Ceci est illustré dans la Figure 3.5 par la transition de l'état `B` vers lui même ; transition ayant l'action `do`

Le cas particulier d'une activité *do* sur un état composite doit être traité différemment. En effet lors de l'étape de flattening des états composites, les transitions émanant de la transformation d'une activité *do* seront reportées sur chacun des états découlant de la transformation.

Cette partie de la transformation permet de s'affranchir des activités internes, tout en conservant le comportement de ces activités. La différence est que les actions générées pour ces activités sont supposées s'exécuter après la transition entrante (avant les sortantes pour les activités de sorties). Or, ici elles s'effectueront en dehors de l'état. Néanmoins l'ordre d'exécution imposé par la spécification sera préservé.

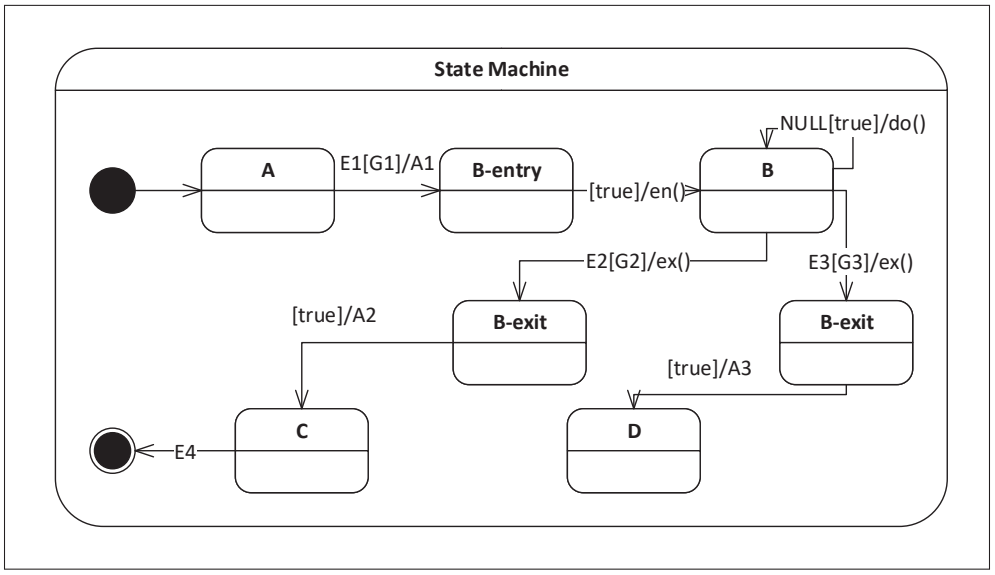


Figure 3.5 Machine à états sans état à activités internes

L'Algorithme 3.2 résume le traitement des activités internes réalisé lors de l'étape de preprocessing.

Algorithme 3.2 Preprocessing : transformation des activités internes

```
1  Entrée : Machine à états UML 𝓜 sans pseudo-états
2  Sortie : Machine à états UML 𝓜' sans pseudo-états ni activité internes
3  for all etat S ∈ 𝓜 do
4      for all activite_entry e_n ∈ S do
5          s_en = creer_etat()
6          t_en = creer_transition()
7          t_en.source ← s_en
8          t_en.cible ← S
9          t_en.condition = true
10         t_en.action = e_n
11         for all S.transition_entrante t_ie do
12             t_ie.cible ← s_en
13         end
14     end
15     for all activite_exit e_x ∈ S do
16         for all S.transition_sortante t_s do
17             s_ex = creer_etat()
18             t_ex = creer_transition()
19             t_ex.condition = true
20             t_ex.action = t_s.action
21             t_ex.source ← s_ex
22             t_ex.cible ← t_s.cible
23             t_s.action = e_x
24             t_s.cible ← s_ex
25         end
26     end
27     for all activite_do d_o ∈ S do
28         t_d = creer_transition()
29         t_d.source ← S
30         t_d.cible ← S
31         t_d.evenement = null
32         t_d.condition = true
33         t_d.action = d_o
34     end
35 end
```

3.2.3 Flattening

Le flattening consiste en la suppression des structures hiérarchiques et concurrentes. En effet, les états peuvent être composites, et contenir eux même des états composites. En découle une complexité importante lorsque l'on essaye d'analyser les différentes possibilités de séquences d'états et de transitions possibles. Pour simplifier cette analyse, il faut obtenir un diagramme équivalent sur un seul niveau de hiérarchie. Pour cela notre solution s'applique d'abord au niveau de hiérarchie le plus profond, pour ensuite remonter jusqu'au niveau le plus haut. En effet, chaque transformation d'état composite est impactée par les états qu'il contient.

3.2.3.1 Transformation des états composites à plusieurs régions concurrentes (And-states)

Les *And-states* correspondent à des états composites contenant deux régions ou plus. En opposition à un *Or-state*, être dans un And-state implique être dans plusieurs états simultanément (un dans chaque région). Le concept de hiérarchie n'existant pas dans les EFSM il est nécessaire d'« aplatir » nos machines. Pour ce faire, nous nous basons sur la méthode proposée par Kim *et al.* (1999). Cette méthode utilisée dans la majorité des travaux sur le flattening propose le passage par des configurations représentant « l'ensemble des états dans lesquels on peut être simultanément ». En réalisant le produit cartésien des sous états contenus dans chaque région, on obtient nos configurations, représentant l'ensemble des états dans lesquels le système est à un instant *t* lors de la période d'activation de l'état composite. Par exemple, la Figure 3.6 montre un état composite *Z* avec deux régions parallèles. Une région contenant les états `A` et `B`, et l'autre contenant les états `C` et `D`. Les configurations construites à partir de l'état composite Z sont montrées dans la Figure 3.7 et elles correspondent aux états `A C`, `A D`, `B C` et `B D`.

Les transitions locales à l'état composite doivent également être transformées. Comme discuté dans la revue de littérature, plusieurs travaux proposent de garder toutes les transitions préexistantes. Cette approche à le désavantage de faire exploser le nombre de transitions et de créer

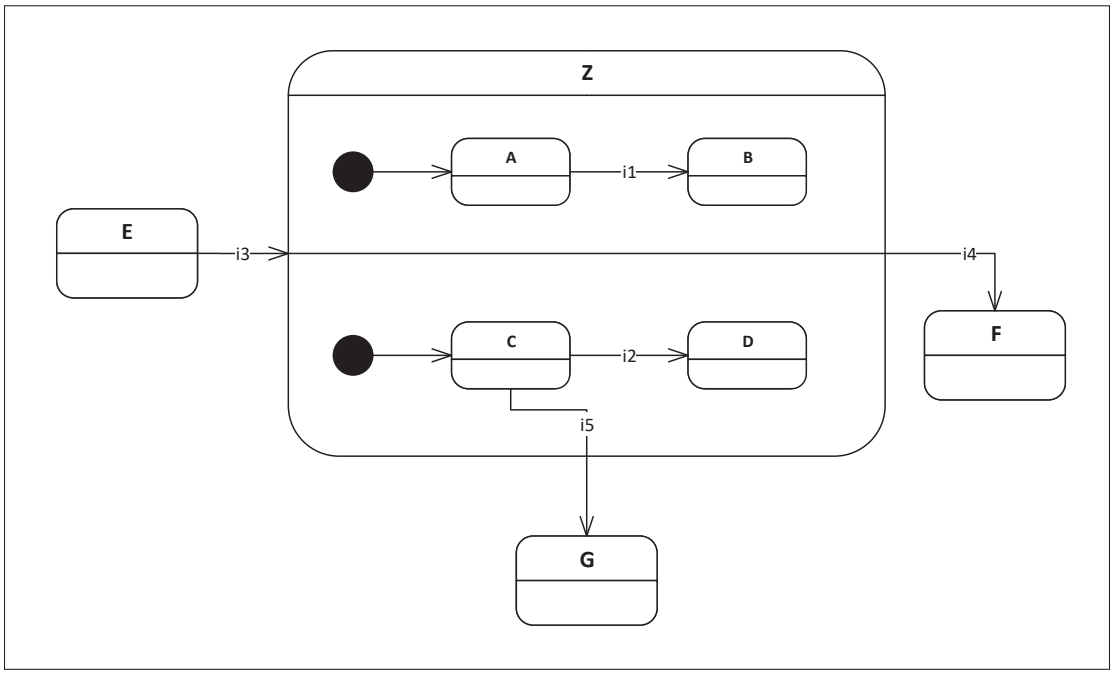


Figure 3.6 État composite Z à régions concurrentes : And-states

des transitions incohérentes. Pour remédier à ce problème , nous proposons une approche de transformation de ces transitions basée sur deux règles.

La première règle découle de la spécification UML qui stipule qu'un seul événement peut être évalué à la fois. Cela signifie que pour les configurations générées à partir d'un état composite, une transition locale reflétera l'évaluation d'un événement et un seul. Autrement dit une transition locale entraîne la transition d'une configuration vers une autre qui différent d'un seul état. Dans l'exemple de la Figure 3.6, une transition émanant de la transition entre A et B se rapporte entre les configurations A C et B C, mais pas entre A C et B D dans la Figure 3.7. Cela permet de limiter le nombre de transitions créées et d'éviter les incohérences (comme la création de transitions entre B C et A D qui conduirait à passer de l'état D à l'état C à travers une transition de A vers B par exemple).

Une seconde règle permet de traiter le cas particulier où une même paire événement/condition existe entre deux transitions ou plus dans des régions différentes. Dans ce cas une transition est créée entre la configuration qui contient les états sources des transitions contenant cette

même paire, et la configuration qui contient les états cibles. Cette transition est créée à la place des transitions entre les configurations contenant seulement une partie des états sources et une partie des états cibles.

Aussi dans un état composite, chaque région contient un pseudo-état initial lié à un et un seul état. Cet ensemble de pseudo-états initiaux va être remplacé par un seul pseudo-état initial, lié à la *configuration de départ*. La configuration de départ correspond à celle contenant tout les états post-initiaux (états possédant précédemment une transition entrante dont la source est un pseudo-état initial).

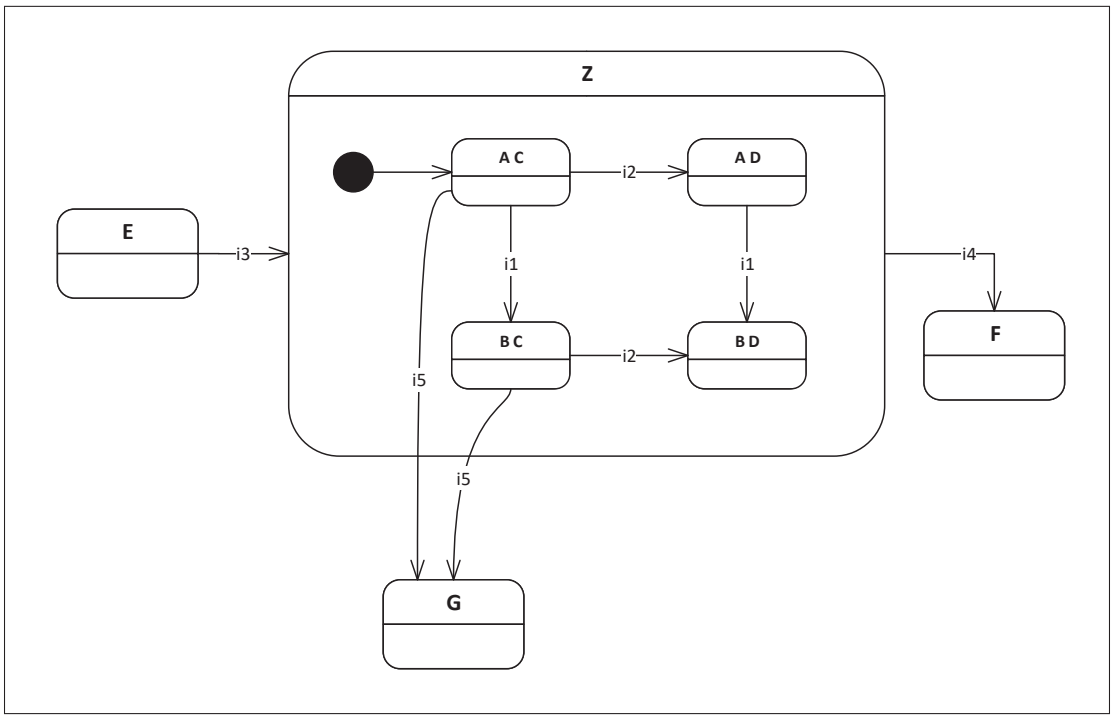


Figure 3.7 Transformation d'un And-state

Une fois qu'on a transformé les états et transitions locales de l'état composite And, on obtient un Or-state (une seule région dans l'état composite). Les transitions "traversants" l'état composite (comme la transition entre C et G dans la Figure 3.6) sont simplement traduites de la façon suivante : on crée une transition par configuration contenant l'état qui était impliqué dans la transition traversante (qu'elle soit entrante ou sortante).

3.2.3.2 Transformation des états composites à une seule région (Or-states)

Un Or-state est un état composite ne contenant qu'une seule région, en conséquence ses sous états s'exécutent de façon séquentielle lors de son activation. L'aplatissement d'un Or-state est beaucoup plus simple.

Ainsi, les états qui existaient dans la machine UML ne sont pas modifiés, ni les transitions qui existent entre eux. Le ou les états finaux sont transformés en états classiques tandis que le pseudo-état initial est retiré. Les transitions entrantes dans l'état composite gardent la même source, tandis que la cible devient l'état qui été connecté au pseudo-état initial. Pour chaque transition sortante de l'état composite, on va créer une nouvelle transition pour chaque état dans l'état composite. La source de cette transition sera l'état obtenu depuis la transformation du Or-state, tandis que la cible sera la cible de la transition sortante.

Chaque état intègre également l'état composite dans lequel il est contenu (à travers son nom). La Figure 3.8 présente la transformation du Or-state de la Figure 3.7.

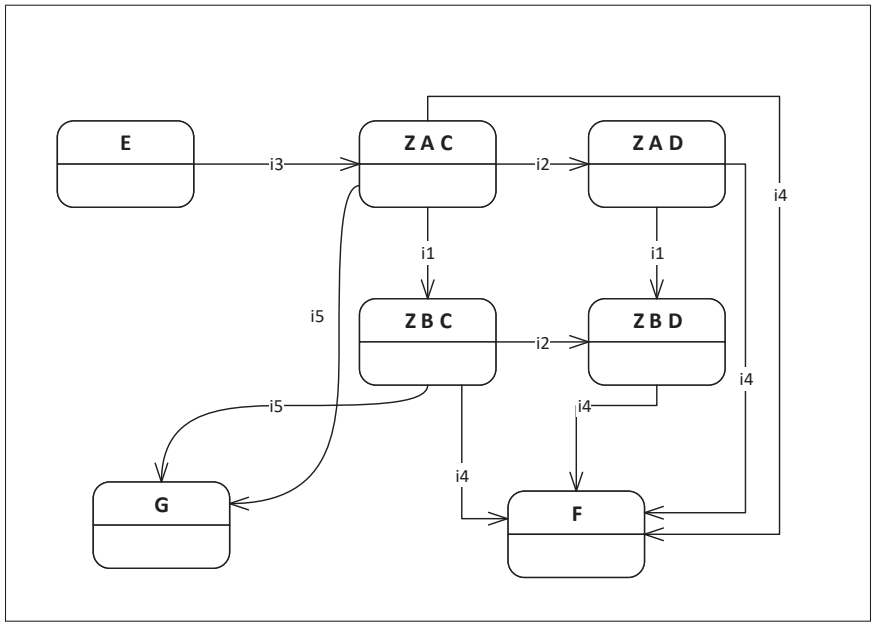


Figure 3.8 Résultat de la transformation d'un Or-state

L'Algorithme 3.3 référence la transformation des And-states des lignes 3 à 14, et la transformation des Or-states des lignes 15 à 27.

Algorithme 3.3 Flattening

```
1  Entrée : Machine à états UML M' pré-traitée
2  Sortie : Machine à états UML M'' aplatie
3  for all And_state S_as ∈ M' do
4      listConfigurations = produitCartesienRegions(S_as)
5      listTransitionsInternes = produitCartesienRegions(S_as)
6      R_e = creer_region //remplacer les régions de S_as par une seule région
7      S_as.region ← R_e
8      for all Configuration C_i ∈ listConfigurations do
9          s_ci = creer_etat
10         s_ci.name = C_i.name
11         R_e.liste_sous_etats.append(s_ci)
12     end
13     S_as.construireTransitionsInternes(listTransitionsInternes, listConfiguration)
14 end
15 for all Or_state S_os ∈ M' do
16     for all transition t_en where t_en.cible = S_os do
17         t_en.cible ← S_os.region.sous_etats.etat_connecte_pseudo_etat_initial()
18     end
19     for all transition t_ex where t_ex.source = S_os do
20         for all etat s_ei ∈ S_os.region.sous_etats do
21             t_ei = creer_transition
22             t_ei.source ← s_ei
23             t_ei.cible ← t_ex.cible
24             t_ei.items ← t_ex.items
25         end
26     end
27 end
```

3.2.4 Mapping

La dernière étape du processus de transformation consiste à passer du formalisme UML vers celui des EFSM. Cette étape de mapping prend comme entrée une machine à états UML pré-traitée et aplatie. Cette machine à états ne contient donc que des états et des transitions, et un unique pseudo-état initial. Comme les transitions EFSM possèdent les mêmes items que

les transitions des machines UML, une simple transposition suffit. Les états étant maintenant simples et sans activité interne, il suffit donc de passer vers des états d'EFSM. Le pseudo-état initial restant est supprimé et l'état auquel il est connecté est indiqué comme étant l'état initial de l'EFSM (`initial = true`). Les états finaux sont transformés en états EFSM étant indiqués comme états finaux (`final = true`).

Ensuite, il faut traiter la communication entre les machines à états UML quand il y en a. Autrement dit, nous avons besoin de récupérer les dépendances entre machines à états UML. Une dépendance entre deux machines à états se manifeste par l'existence d'une transition dans une machine, laquelle est activée à la réception d'un événement envoyé par l'autre machine. L'envoi, dans la spécification UML, ne peut se faire qu'à travers une action, et la réception à travers un événement.

Notre approche consiste à extraire les signaux envoyés et reçus par chaque machine à états UML. Nous créerons une liste, appelée *Lsend*, pour les signaux envoyés par une machine et une liste, appelé *Lreceive*, pour les signaux reçus par une machine. Pour faciliter l'identification des ces signaux, nous avons mis en place une directive à suivre lors de leur spécification.

- L'action d'envoyer doit être spécifiée par un *SEND(signal)* sur la transition dans la machine à états UML.
- L'événement reçu doit être spécifié à travers un *RECEIVE(signal)* sur la transition dans la machine à états UML.

L'Algorithme 3.4 présente l'étape du mapping.

3.3 L'approche appliquée aux machines à états Stateflow

De la même façon que pour le langage UML, nous avons appliqué notre approche aux machines à états Stateflow. Donc nous avons analysé le langage Stateflow pour identifier les concepts à traiter et comprendre leur sémantique. Nous avons ensuite appliqué les trois étapes de notre processus de transformation.

Algorithme 3.4 Mapping

```
1  Entrée : Machine à états UML M'' pré-traitée et aplatie
2  Sortie : EFSM T
3  L_send = creer_liste
4  L_receive = creer_liste
5  for all etat S ∈ M'' do
6  |   S_efsm = creer_etat_EFSM()
7  |   S_efsm.name = S.name
8  |   if S.isFinal() == true then
9  |   |   S_efsm.final = true
10 |   end
11 |   if pseudo_etat_initial ⊂ S.transitionsEntrantes() then
12 |   |   S_efsm.initial = true
13 |   end
14 end
15 for all transition T ∈ M'' do
16 |   T_esfm = creer_transition_EFSM()
17 |   T_efsm.source = T.source
18 |   T_efsm.cible = T.cible
19 |   T_efsm.evenement = T.evenement
20 |   if T.evenement.startWith(SEND) then
21 |   |   L_send.append(T.evenement)
22 |   end
23 |   if T.evenement.startWith(RECEIVE) then
24 |   |   L_receive.append(T.evenement)
25 |   end
26 |   T_efsm.condition = T.condition
27 |   T_efsm.action = T.action
28 end
```

3.3.1 Résumé du processus d'analyse des machines à états Stateflow

Suite à l'analyse du formalisme des machines à états Stateflow, nous avons identifié les concepts suivants (ces concepts ont été présentés en détail dans la section 1.4) :

1. Les états contiennent optionnellement des actions internes, et sont soit actifs soit inactifs au cours de la durée de vie de la machine. Aucun état n'est déclaré comme étant final. Les

transitions contiennent optionnellement un événement, une condition et une action (et une condition action dans l'ancienne version de la spécification).

2. Le formalisme Stateflow introduit le concept de jonction, exprimant certains comportements particuliers selon comment la jonction est utilisée. on distingue :
   - Les jonctions pour fusionner ou diviser des transitions ; celles-ci sont similaires aux noeuds de jonction dans UML.
   - Les jonctions avec des boucle "for", i.e., la jonction a une transition vers elle même conditionnée par une boucle "for".
   - Les jonctions avec des "if-then-else" ; la jonction dans ce cas divise la transition en deux ou plusieurs transitions en spécifiant les conditions sur certaines ou toute les transitions sortantes de la jonction.
3. Enfin, les états dans Stateflow peuvent être composites ou simples. Comme dans UML il existe deux catégories d'états composites : Or-state (séquentiel) ou And-state (concurrent).

3.3.2 Preprocessing

Comme pour le langage UML, Stateflow définit des concepts qui ne sont pas définis dans les EFSM et qu'il faut donc pré-traiter. Ces concepts comprennent les points de jonctions et les actions internes.

3.3.2.1 Les points de jonctions

Les points de jonctions, appelés jonctions connectives, comprennent les jonctions avec boucle "for", les jonctions "if-then-else" et les jonctions fusion et division.

La jonction avec une boucle *for*

Un exemple de jonction avec boucle "for" est présenté à la Figure 3.9. Dans cet exemple, une fois que l'on sort de l'état `A`, on va simplement exécuter 10 fois la fonction `func1()`, pour ensuite entrer dans l'état `B`. La transition sortante d'une jonction avec une boucle "for" n'a pas de

conditions. La transformation proposée consiste à éliminer cette jonction, et à considérer cette boucle for comme une *action*. Comme cette boucle est exécutée pour ensuite atteindre l'état suivant, on place l'action dans la transition allant de l'état source de la transition entrante dans la jonction, à l'état cible de la transition sortante de la jonction. L'événement et la condition seront récupérés depuis la transition entrante. Le résultat de la transformation de la machine de la Figure 3.9 est présenté à la Figure 3.10. Ici on définit une fonction Matlab (ou autre type de fonction) représentant la boucle for en question.

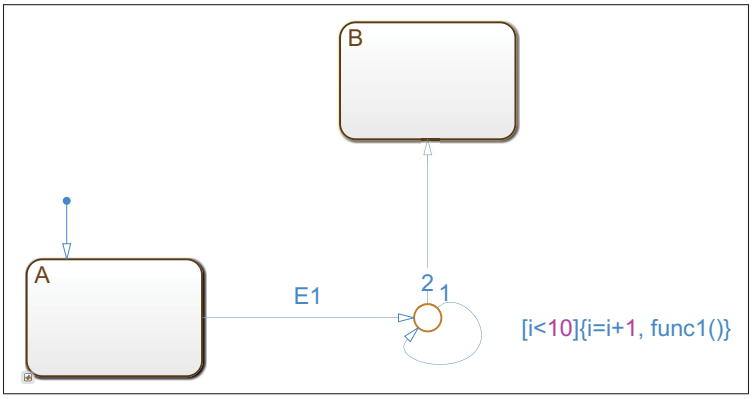


Figure 3.9 Construction d'une boucle for dans le langage Stateflow

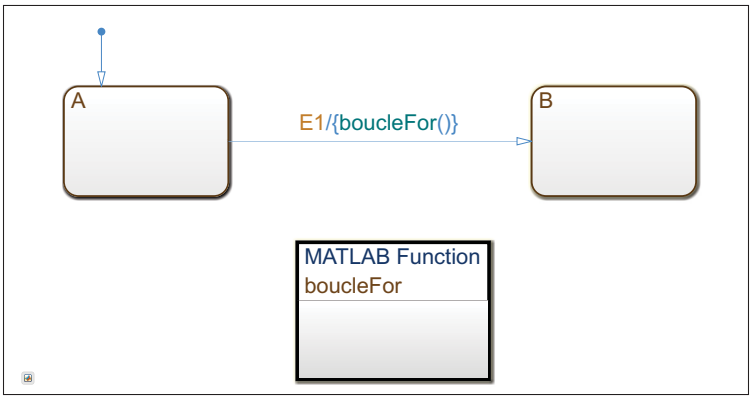


Figure 3.10 Résultat de la transformation d'une boucle for dans le langage Stateflow

La jonction *if-then-else*

Cette construction correspond au cas classique de programmation où l'on vérifie une condition, si celle-ci est vraie on exécute la branche, sinon on vérifie et exécute l'une des autres branches (s'il en existe plusieurs). Cette situation correspond dans UML au cas où on a une évaluation statique des conditions sur les transitions sortantes d'un noeud de jonction. La Figure 3.11 présente un exemple de jonction "if-then-else". Dans ce cas, l'action de la transition de A vers la jonction ne sera pas exécutée avant d'évaluer les conditions sur les transitions sortantes de la jonction. Le résultat de la transformation de cette jonction est présenté à la Figure 3.12. La transformation est similaire à celle réalisée pour UML dans le cas d'une évaluation statique d'un noeud de jonction ; il suffit de connecter l'état source et l'état cible en additionnant les conditions.

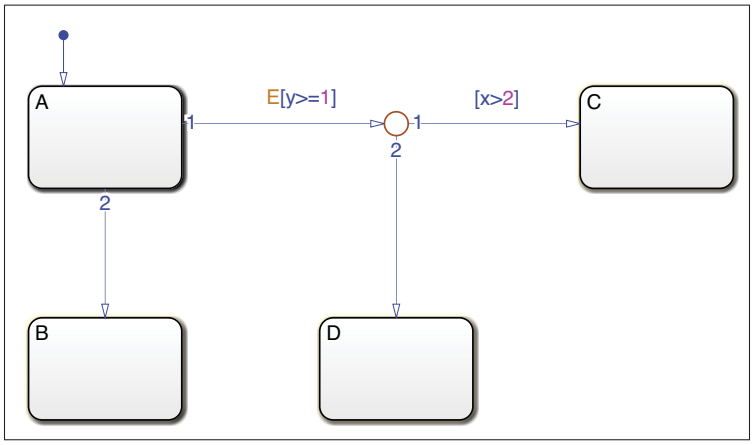


Figure 3.11 Construction d'un comportement if-then-else dans le langage Stateflow

La jonction pour fusionner/diviser

De la même façon que dans le langage UML, une jonction peut être utilisée pour limiter le nombre de transitions en les fusionnant ou les divisant. Ici la transformation sera donc la même, à savoir retirer la jonction, et connecter l'ensemble des états ciblés par les transitions sortantes de la jonction avec les états sources des transitions entrantes à la jonction. La Figure 3.14 présente le résultat de la transformation des machines présentées dans la Figure 3.13.

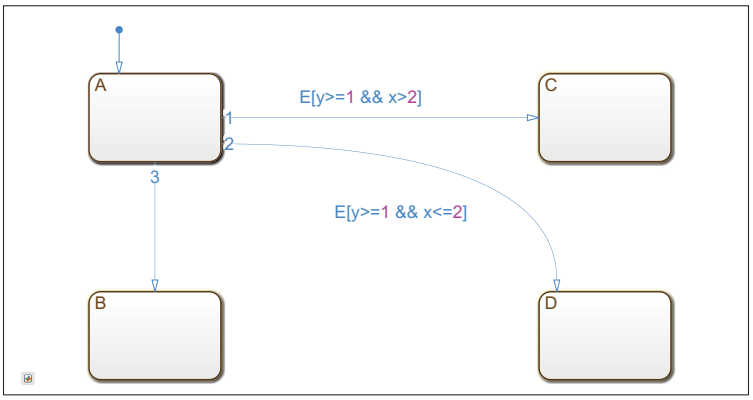


Figure 3.12 Résultat de la transformation d'un comportement if-then-else dans le langage Stateflow

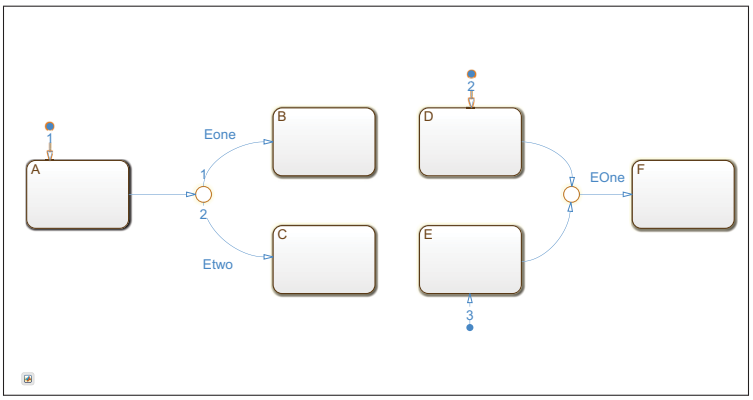


Figure 3.13 Construction utilisant une jonction pour fusionner/diviser des transitions

3.3.2.2 Les actions internes

Les actions internes correspondent aux *entry/during/exit/onAction* que l'on peut retrouver dans les états Stateflow. Les actions *entry*, *during* et *exit* ont la même sémantique que les activités internes entry, do et exit que l'on retrouve dans le langage UML. La transformation sera donc similaire : on découpe l'état en deux pour les *entry* et *exit*, et on déplace les événements, conditions et actions afin de respecter l'ordre d'exécution. De la même façon l'action *during* devient une transition de l'état vers lui même, contenant une action équivalente à l'action during, une condition de valeur *true*, et l'événement *null* (correspondant à l'absence d'autre événement

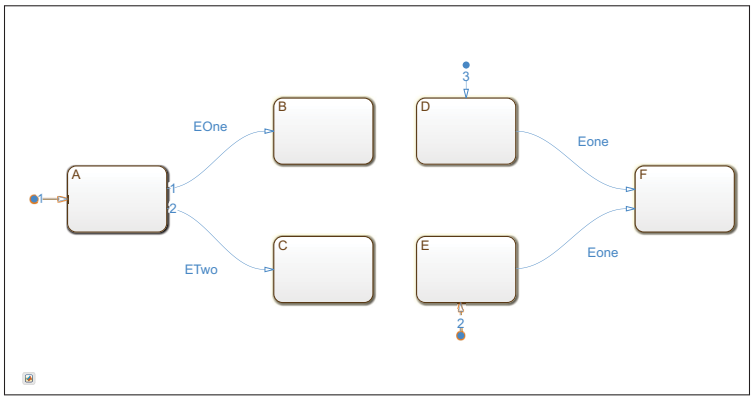


Figure 3.14 Résultat de la transformation d'une construction utilisant une jonction pour réunir/séparer des transitions

déclenchant la sortie de l'état). Le résultat de la transformations de l'action "*entry : onCount = 0*", l'action "*during : lightOn()*", et l'action "*exit : lightOff()*" de la Figure 3.15 est donné dans la Figure 3.16.

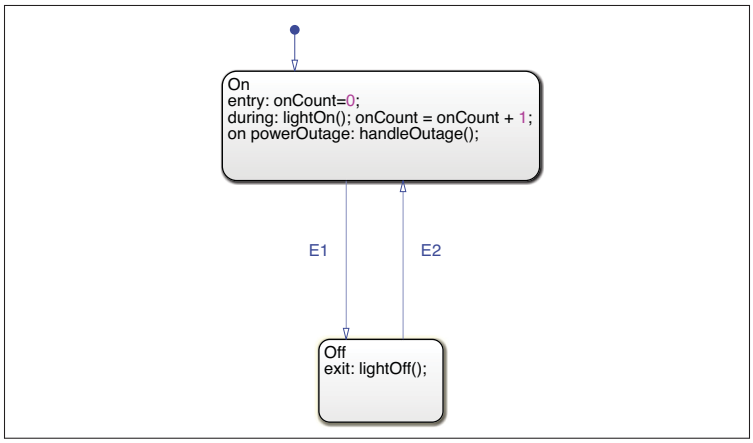


Figure 3.15 Machine Stateflow contenant des actions internes

En ce qui concerne l'action *onAction*, celle-ci correspond à une action exécutée dans le cas où un événement particulier arrive pendant que l'état est actif. Dans la Figure 3.15, par exemple, si l'événement `powerOutage` arrive alors que l'état `On` est actif, L'action `handleOutage()` est exécutée. La transformation est similaire à celle proposée pour l'action during. Mais cette

fois-ci un événement sera nécessaire à l'exécution de l'action, donc au lieu de l'événement *null*, on place l'événement correspondant sur la transition. Le résultat de la transformation de l'action interne *on powerOutage : handleOutage()* est présenté dans la Figure 3.16.

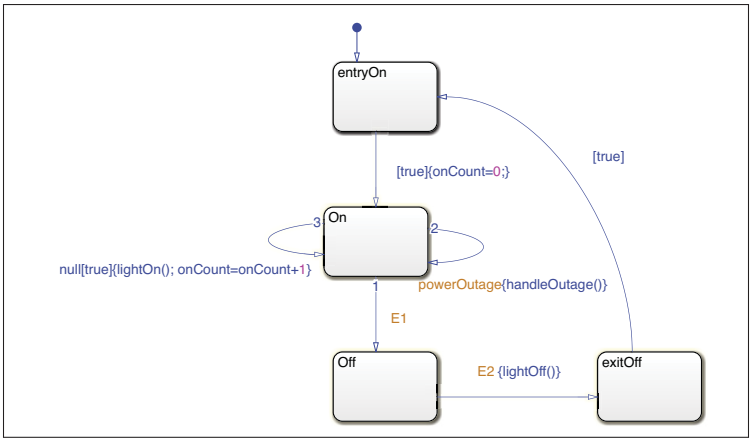


Figure 3.16 Résultat de la transformation des actions internes

3.3.3 Flattening

De la même façon que dans le langage UML, le langage Stateflow propose des constructions hiérarchiques (séquentielles ou concurrentes) qu'il faudra aplatir. On rappelle qu'on applique le flattening du niveau le plus bas vers le niveau le plus haut.

3.3.3.1 Les And-states

La transformation se fait de la même manière que pour le formalisme UML, sauf qu'ici les régions correspondent à des états (composites ou non) définis avec une décomposition *AND*. La première étape de la transformation consiste à créer les *configurations*. Pour rappel, celles-ci correspondent à l'ensemble des états concurrents pouvant être actifs simultanément. Par exemple, dans la Figure 3.17, aussitôt que l'état `CurrentOn` est actif, on se retrouve simultanément dans les états `Off` de `Light1`, `Off` de `Light2`, et `VerifyPower`. L'une des configurations sera donc : `Light1OffLight2OffVerifyPower`. Si un des états parallèles du And-state est un

état simple, il sera actif tant que le And-state est actif, donc contenu dans toutes les configurations. C'est le cas par exemple de l'état `VerifyPower`.

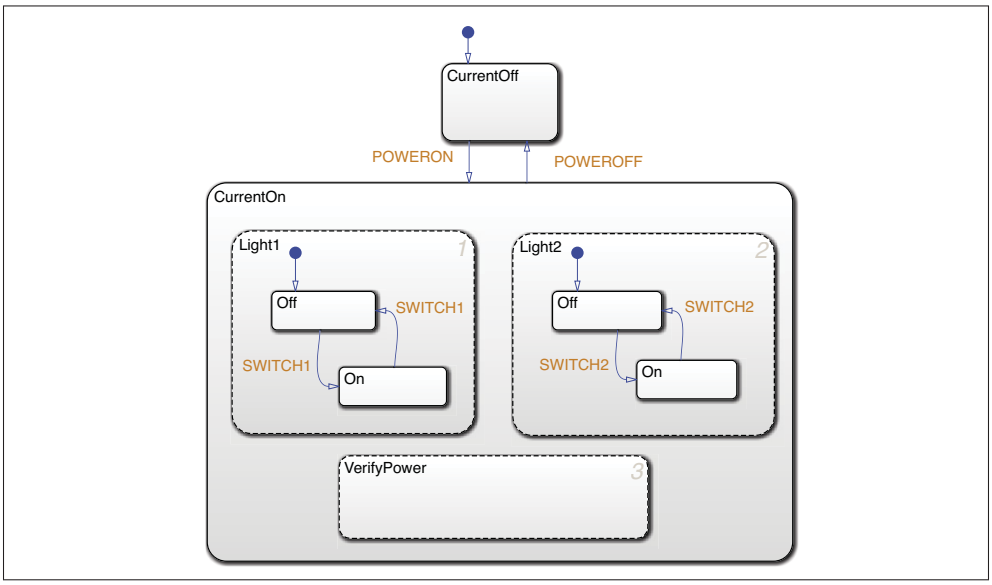


Figure 3.17 And-state Stateflow

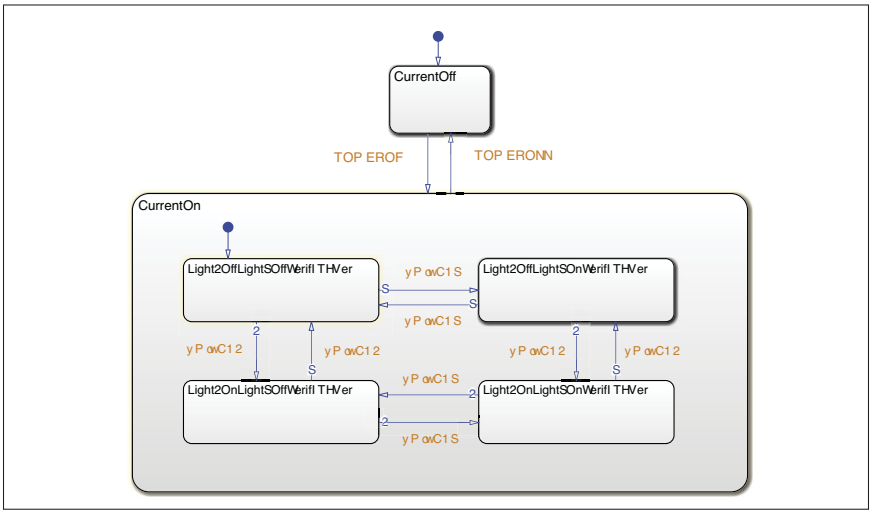


Figure 3.18 Résultat de la transformation d'un And-state Stateflow

La seconde étape consiste à transformer les transitions internes au And-state. Celles-ci sont transformées de la même manière que pour le formalisme UML. C'est à dire qu'une transition

entre deux configurations ne peut amener qu'au changement d'un seul état dans la configuration. Dans la Figure 3.18, la transition avec l'événement SWITCH1 n'amène qu'au changement de l'état Light1.Off à Light1.On.

Il faut également gérer les transitions par défaut. Pour cela, on applique la même règle que pour UML, à savoir déterminer la configuration "initiale" contenant uniquement des états post-initiaux. Cette configuration reçoit une transition par défaut comme transition entrante. À l'issue de cette transformation, on obtient un Or-state (ici le Or-state CurrentOn).

Pour le traitement des transitions internes, il existe deux cas particuliers qui nécessitent d'être pris en considération. Le premier est l'existence de couples évènement/condition communs qui déclenchent des transitions dans différents états parallèles (même cas particulier que UML), par exemple l'événement E_{two} dans la Figure 3.19. Pour ce cas particulier, on applique la même règle que pour UML (voir partie UML section 3.2.3.1). L'autre cas particulier est celui des signaux envoyés d'un état parallèle vers un ou plusieurs autres afin de les synchroniser. Dans la Figure 3.19, l'état composite A envoie un signal EOne à l'état composite B à travers l'action send(EOne,B).

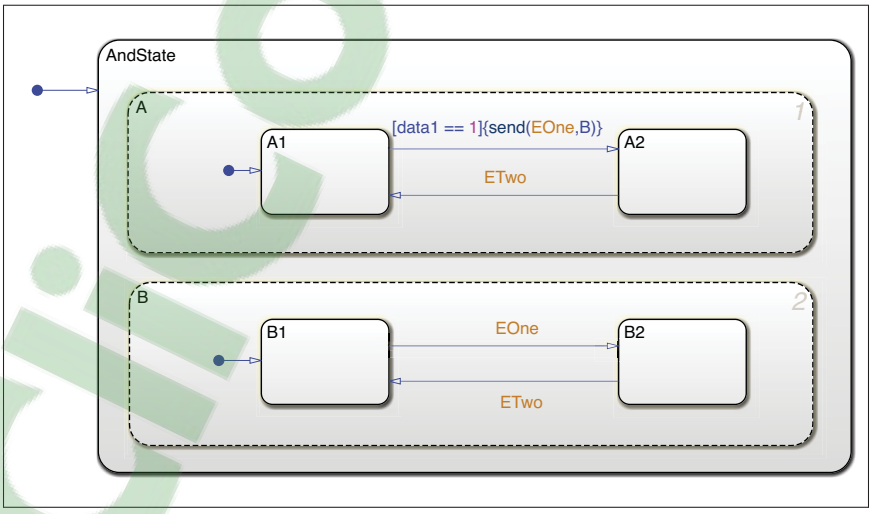


Figure 3.19 And-state Stateflow contenant un envoi de signal

Pour ce cas particulier, le signal envoyé par la transition de l'état `A1` vers `A2` dans la Figure 3.19 va automatiquement déclencher la transition de `B1` vers `B2`. Comme le traitement des événements est séquentiel, si la transition entre `A1` et `A2` est déclenchée, alors on attend la fin de l'exécution de cette transition, puis on exécute la transition entre `B1` et `B2`. La transformation des transitions locales dans ce cas génère deux transitions. La première entre la configuration qui contient l'état source de la transition qui envoi le signal (i.e.,`AndStateAA1BB1` dans la Figure 3.20) et la configuration qui contient la cible de cette transition et la source de la transition qui reçoit le signal (i.e.,`AndStateAA2BB1`). La transition créée possède les mêmes items : événement, condition et actions autres que l'envoi du signal, de la transition qui contient l'envoi du signal. La seconde transition créée a pour source la configuration contenant l'état cible de la transition qui envoi le signal et l'état source de la transition qui reçoit le signal (i.e.,`AndStateAA2BB1`), et pour cible la configuration qui contient l'état cible de la transition qui reçoit le signal (i.e.,`AndStateAA2BB2`). Les items de cette seconde transition sont un guard de valeur *true*, et une action égale à celle de la transition qui reçoit le signal. Le résultat de la transformation est présenté dans la Figure 3.20.

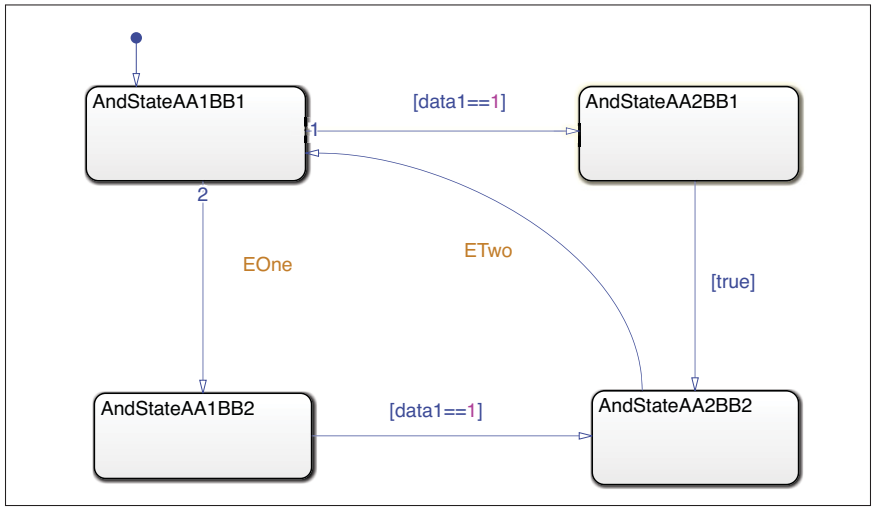


Figure 3.20 Résultat de la transformation d'un And-state contenant un envoi de signal

3.3.3.2 Les Or-states

La transformation est équivalente à UML. Un Or-state dans Stateflow est similaire à un Or-state dans UML. Il correspond à un état composite qui ne contient pas d'états concurrents. Donc deux de ses sous états ne peuvent pas être actifs au même instant. La transformation d'un Or-state Stateflow est donc similaire à celle de UML (voir Section 3.2.3.2). L'état composite est retiré, son nom est intégré aux noms de ses sous états, et les transitions entrantes et sortantes sont connectées de la même manière que pour la transformation du formalisme UML.

En ce qui concerne les transitions entrantes dans le Or-state, comme pour la transformation de UML celles-ci doivent être connectées à l'état suivant le "point initial". Ici le point initial correspond à la transition par défaut. Pour les transitions sortantes, comme pour le langage UML, il faut créer une nouvelle transition sortante par sous état, pour chaque transition sortante du Or-state.

3.3.4 Mapping

Ayant une machine Stateflow pré-traitée et aplatie, cette étape consiste è établir la correspondance entre les concepts de cette machine et celle d'une machine EFSM :

- Un état Stateflow devient un état EFSM.
- Une transition Stateflow devient une transition EFSM avec les mêmes items événement, condition et action.
- L'état cible de la transition par défaut restante, correspondant à un "point initial", devient un état EFSM balisé comme `initial`.

Comme dans UML, les machines permettent de communiquer par envoi/réception de signaux. Les signaux internes échangés entre sous-états d'un même état And-state ont été déjà traités dans le flattening. Cependant, les signaux échangés entre machines disjointes doivent être récupérés et retranscrits dans les EFSM. Pour ce faire, nous adoptons la même démarche que

pour UML : nous créons deux listes Lsend et Lreceive. Lsend contient les signaux envoyés par la machine, et Lreceive les signaux reçus.

3.4 Conclusion

Nous avons donc appliqué la démarche générale d'analyse et de transformation de machines à états à deux formalismes : le langage UML et le langage Stateflow. Il est à noter que, pour les deux langages, nous avons couvert l'ensemble des concepts utilisés par les partenaires industriels. Ces concepts sont les plus couramment utilisés. Toutefois, l'approche peut être appliquée à tous les concepts définis par les deux langages. Dans le prochain chapitre nous présenterons une implémentation du processus de transformation pour les deux langages, ainsi qu'une étude de cas réalisée pour le langage UML.

CHAPITRE 4

IMPLÉMENTATION ET EXPÉRIMENTATIONS

Dans ce chapitre, nous commençons par présenter de façon sommaire les outils que nous avons utilisé pour implémenter l'approche STEF. Nous présentons ensuite les détails d'implémentation pour les langages UML et Stateflow. Nous reportons aussi les résultats d'expérimentation à travers une étude de cas .

4.1 Choix technologiques

Eclipse Modeling Framework (EMF)

Pour l'implémentation de notre approche, nous avons choisi d'utiliser EMF (Eclipse, 2019a) qui est un cadre d'application ouvert qui supporte l'ingénierie dirigée par les modèles. EMF rassemble un ensemble d'outils et de plug-ins supportant l'IDM, et il est très utilisé que ce soit dans le monde académique ou industriel. Au coeur de EMF, on retrouve Ecore, permettant de créer des méta modèles satisfaisant au standard MOF. Pour UML, nous avons utilisé un plug-in basé sur EMF, appelé UML2, qui implémente le méta modèle UML. La représentation graphique des modèles étant un aspect important dans l'IDM, plusieurs plug-ins basés sur EMF sont disponibles. Nous avons utilisé le plug-in Papyrus qui permet de créer un modèle UML de façon simple à l'intérieur d'Eclipse. Pour les modèles EFSM résultant des transformations, nous avons implémenté un méta modèle en utilisant Ecore (voir Section 4.2). Pour la représentation graphique des EFSM, nous avons utilisé le plug-in Sirius qui offre la possibilité d'associer une représentation graphique aux éléments du méta modèle qui ont été créés en utilisant EMF.

ATL

L'étude des langages de transformation implémentant QVT réalisée dans la Section 1.2.5 montre qu'ATL présente plusieurs propriétés intéressantes pour l'implémentation d'une transformation. Il est performant, et ses développeurs sont actifs sur les forums de discussion. Il est

à la fois utilisé dans le domaine industriel et scientifique. De plus, une implémentation pour EMF est fournie. En conformité avec les recommandations des concepteurs d'ATL, nous avons utilisé la partie déclarative d'ATL pour implémenter notre approche.

Une transformation ATL est stockée dans un fichier ".atl". Elle peut être composée de plusieurs règles de transformation. La Figure 4.1 est un exemple d'extrait du fichier ".atl". ATL permet de générer des traces entre éléments sources et cibles lors de la transformation. Les lignes 1 et 2 dans la figure permettent de spécifier les adresses des méta modèles source et cible de la transformation. La ligne 3 donne le nom et spécifie le méta modèle source (model1) et le méta modèle cible (model2). Les *helpers* (lignes 6 et 7) permettent de définir des fonctions que l'on peut appeler dans les règles ; c'est un mécanisme favorisant la réutilisation. Ces fonctions sont spécifiées en utilisant le langage OCL. Par exemple, le helper dans la Figure 4.2 permet de transformer la première lettre d'une chaîne de caractères en une lettre minuscule.

```
1  --model1 = URI du meta modèle 1
2  --model2 = URI du meta modèle 2
3  module transformation
4  create OUT : model2 from IN : model1
5
6  helper context TypedEntree def : fonction1() : TypeDeRetour =
7  corps ;
8
9   rule ElementModel1ToElementModel2 {
10 from
11        s : Model1!Element1 (condition sur Element1)
12 to
13        t : Model2!Element2 (
14                  features1DeElement2<-s.features1DeElement1,
15                  features2DeElement2<-s.features2DeElement1,
16                  …)
17        }
```

Figure 4.1 Le langage ATL, adapté de Jouault *et al.* (2008))

Les lignes 9 à 17 de la Figure 4.1 présentent une règle de transformation. Une règle a un nom (ligne 9), et peut être spécifiée de manière déclarative ou impérative. Les règles déclaratives

```
helper context String def: firstToLower() : String =
        self.substring(1, 1).toLower() + self.substring(2, self.size());
```

Figure 4.2 Helper du langage ATL,
extrait de Jouault *et al.* (2008))

sont appelées *Matched rules*. Dans la Figure 4.1, la partie "from ..."" (lignes 10 à 11) spécifie l'élément source de la règle. La partie "to ..." (lignes 12 à 17) permet de spécifier l'élément cible de la transformation, ainsi que les valeurs de ses attributs. Il existe trois types de règles déclaratives :

- Les règles standards (*Rules*) qui sont appliquées une fois pour chaque élément source satisfaisant les conditions.
- Les règles *Lazy Rules* qui sont appelées par d'autres règles, et qui peuvent être appliquées plusieurs fois pour définir plusieurs éléments cibles.
- Les règles *Unique Lazy Rules* qui peuvent être appelées une seule fois pour un élément source donné pour produire un élément cible.

Dans notre implémentation, nous avons utilisé les règles standards uniquement.

4.2 Implémentation du méta modèle EFSM

Le langage cible de notre transformation étant EFSM, nous avons dû construire et implémenter le méta modèle EFSM. Pour construire le méta modèle nous avons analysé le langage EFSM (une présentation détaillée des EFSM a été donnée dans la Section 1.5). Nous avons implémenté le méta modèle en utilisant l'outil Ecore de EMF. La Figure 4.3 présente le méta modèle résultant. Une `ExtendedFiniteStateMachine` est composée d'états (`State`) et de transitions (`Transition`). Elle possède comme attributs un nom et deux listes `lsend` et `lreceive` permettant de stocker les signaux envoyés et reçus par la machine. Les états ont un nom et peuvent éventuellement être initiaux ou finaux. Ils sont connectés entre eux par des transitions, qui possèdent un nom, un évènement, une condition et une action.

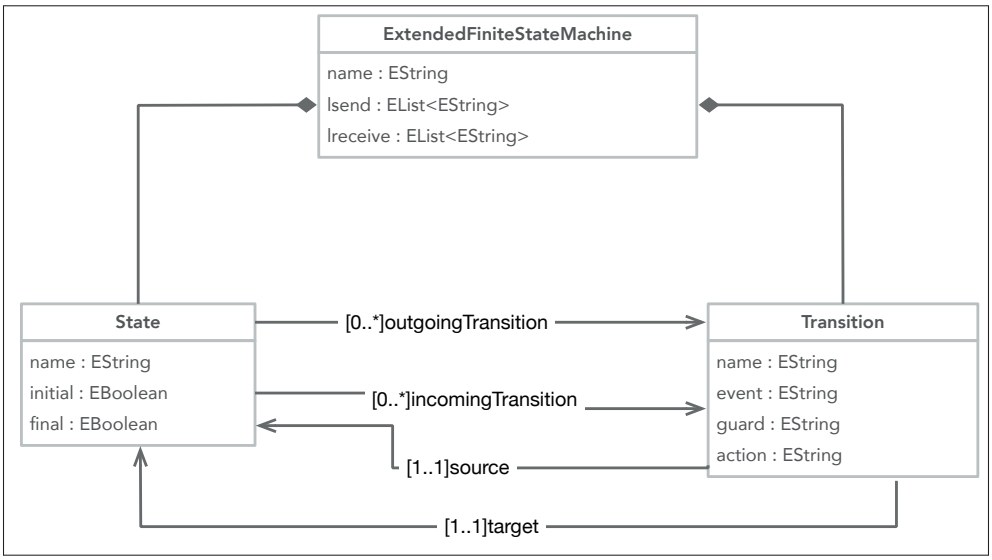


Figure 4.3 Méta modèle EFSM

4.3 Implémentation de l'approche STEF pour UML

L'implémentation de notre approche pour les machines à états UML a résulté en cinq transformations ATL (5 fichiers ".atl"). Deux transformations ont été implémentées pour supporter la phase de Preprocessing : une pour les pseudo-états et une pour les activités internes. Pour le Flattening, nous avons aussi implémenté deux transformations : une pour les And-states et une pour les Or-states. La cinquième transformation implémente le Mapping.

ATL traite ces transformations de façon séquentielle. De ce fait nous avons assigné un ordre aux cinq transformations. Comme ATL permet d'utiliser le plug-in *Apache Ant* pour spécifier une chaîne de transformations, nous avons créé un fichier "build.xml" où nous avons spécifié l'ordre de nos transformations. La Figure 4.4 présente une vue du contenu du fichier "build.xml". Ce fichier permet de charger le modèle source au format UML (e.g., `input.uml`), d'appliquer les fichiers ATL de transformation dans l'ordre (depuis `TransformPseudoState.atl` jusqu'à `Mapping.atl`) au modèle à transformer et aux modèles intermédiaires, et enfin de sauvegarder le modèle final obtenu qui est conforme au méta modèle EFSM (`result.efsm`).

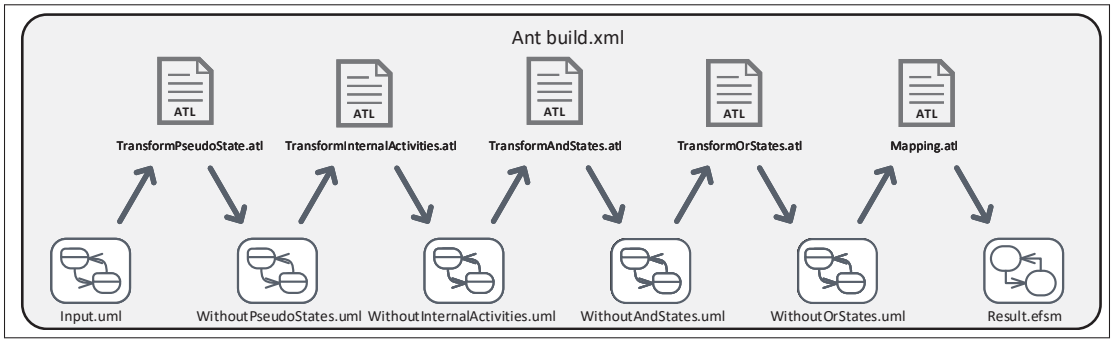


Figure 4.4 L'implémentation de STEF pour UML dans Eclipse

Durant l'implémentation de STEF pour UML nous avons dû définir un certain nombre de directives qu'un concepteur doit respecter lors de la création des machines à états UML. Certaines de ces directives reflètent les pratiques des partenaires industriels, par exemple la hiérarchie des états doit se limiter à trois niveaux. D'autres directives ont été déduites de la spécification UML. Ces directives rendent explicites des pratiques/contraintes implicites dans la spécification, par exemple, une région doit commencer par un (et un seul) pseudo-état initial.

4.3.1 La transformation des pseudo-états

Le fichier ATL de transformation des pseudo-états est présenté dans la Figure 4.5. Ce fichier est composé de six règles de transformation permettant de traiter les points d'entrées et de sorties, ainsi que les noeuds de jonction et de décision.

```
1 – @nsURI UML = http ://www.eclipse.org/uml2/4.0.0/UML
2 module transfoPseudoStates
3 create OUT : UML from IN : UML
4 rule TransitionReplacingEntryPoint{...}
5 rule TransitionReplacingExitPoint{...}
6 rule Junction2Transitions{...}
7 rule Choice2State{...}
8 rule IncomChoiceTransi2Transi{...}
9 rule OutgoChoiceTransi2Transi{...}
```

Figure 4.5 Fichier ATL de transformation des pseudo-états

Un des exemples de règle de transformation est présenté dans la Figure 4.6, lequel décrit la transformation d'une transition visant un point d'entrée (`TransitionReplacingEntryPoint`). Pour rappel, la transformation d'un point d'entrée passe par la transformation des transitions entrantes vers ce point d'entrée.

Les lignes 2 à 13 de la Figure 4.6 permettent de définir l'élément source sur lequel la règle s'appliquera, à savoir une transition entrante d'un pseudo-état de type point d'entrée. La seconde partie, de la ligne 14 à la ligne 31, crée une nouvelle transition avec les éléments adéquats : évènement, condition et action, tout en lui affectant la bonne source et la bonne cible. Pour rappel l'état cible est maintenant l'état que ciblait la transition sortante du point d'entrée.

4.3.2 La transformation des activités internes

Dans cette transformation, les activités internes *entry*, *do* et *exit* sont traitées. Le fichier ATL de transformation contient six règles de transformation. En effet, en plus de la transformation des trois types d'activités (3 règles), il faut traiter des cas particuliers à travers d'autres règles. Par exemple la transformation d'une activité exit dont une des transitions sortantes de l'état qui la contient a pour cible un état contenant une activité *entry*.

La Figure 4.7 présente la transformation de l'activité *entry* (le traitement des activités *exit* est basé sur une construction équivalente). Cette transformation passe par la transformation des transitions entrantes vers l'état, contenant l'activité *entry*. Les lignes 2 à 10 spécifient l'élément source de cette transformation, à savoir une transition entrante vers un état contenant une activité *entry*. Depuis cette transition on génère les différents éléments créés par la transformation : l'état intermédiaire *entryState* (lignes 12 à 14), la transition allant vers cet état contenant les éléments de la transition transformée (lignes 15 à 22), et la transition entre cet état et l'état qui contenait l'activité d'entrée (lignes 27 à 33). Comme une nouvelle transition est créée, il faut également créer les items de celle-ci (lignes 23 à 26 pour créer *guard* et *effect*).

```
1  rule TransitionReplacingEntryPoint {
2      from
3          s : UML !Transition (if s.target.oclIsTypeOf(UML !Pseudostate)
4              then
5                  if s.target.kind=#entryPoint
6                  then
7                      not s.source.oclIsTypeOf(UML !Pseudostate)
8                  else
9                      false
10                 endif
11                 else
12                     false
13                 endif)
14             to
15                 t1 : UML !Trigger(
16                     name<-s.getTriggerNames()),
17                 t2 : UML !Constraint(
18                     name<-s.getGuardNames()),
19                 t3 : UML !FunctionBehavior(
20                     name<-s.getActionNames()),
21                 t4 : UML !Transition(
22                     name<-s.name,
23                     source<-s.source,
24                     target<-s.target.getOutgoings()->first().target,
25                     trigger<-t1,
26                     guard<-t2,
27                     effect<-t3,
28                     container<-s.container)}
```

Figure 4.6 Règle de transformation d'un point d'entrée

4.3.3 La transformation des And-states

Ce fichier de transformation contient deux helpers et une règle s'assurant de la gestion complète d'un And-state. Cette transformation, qui s'intègre dans la partie flattening de notre algorithme, permet de traiter les états composites avec deux régions parallèles (r_1, r_2), fournissant en sortie un Or-state. La Figure 4.8 est un extrait de cette règle de transformation. Les lignes 2 à 8 spécifient l'élément source qui est un And-state. Dans la partie cible de la transformation, la première étape consiste à créer les configurations en appelant un helper *getConfigurationAndStates()* (ligne 12). Ce helper est détaillé dans la Figure 4.9. Cet helper récupère chaque état

```
1  rule EntryActivityToStateAndATransi{
2      from
3          s : UML !Transition(if not s.target.oclIsTypeOf(UML !Pseudostate)
4              then if not s.target.entry.oclIsUndefined()
5                  then if not s.source.oclIsTypeOf(UML !Pseudostate)
6                      then
7                          s.source.exit.oclIsUndefined()
8                      else false endif
9                  else false endif
10                 else false endif )
11             to
12                 t1 :UML !State(
13                     container<-s.container,
14                     name<-s.target.name+"-entryState"),
15                 t2 :UML !Transition(
16                     name<-s.name,
17                     source<-s.source,
18                     target<-t1,
19                     trigger<-s.trigger,
20                     guard<-s.guard,
21                     effect<-s.effect,
22                     container<-s.container),
23                 t3Guard :UML !Constraint(
24                     name<-"true"),
25                 t3Action :UML !FunctionBehavior(
26                     name<-s.target.entry.name),
27                 t3 :UML !Transition(
28                     name<-s.name+"bis",
29                     source<-t1,
30                     target<-s.target,
31                     container<-s.container,
32                     effect<-t3Action,
33                     guard<-t3Guard)}
```

Figure 4.7 Règle de transformation d'une activité interne *entry*

S_ir_1 de la région r_1, et pour chaque état S_jr_2 dans la région r_2 on crée une paire d'états (S_ir_1, S_jr_2). Cela revient à réaliser le produit cartésien des deux régions. La première boucle à la ligne 4 itère sur la première région, tandis que l'on itère sur la seconde région à la ligne 6. Ce helper appliqué sur un And-state retourne l'ensemble des configurations émanant des sous états de celui-ci.

```
1  rule Andstate2states{
2      from
3          s : UML!State (if s.isOrthogonal()
4              then
5                  true
6              else
7                  false
8              endif)
9          to
10             t : UML!State(
11                 name<-s.name)
12             t1 : distinct UML!State foreach(config in s.getConfigurationAndStates())(
13                 name<-config.asSequence()->iterate(states; acc : String = "" | acc +
                     states.name + " "),
14                 container<-t),
15             t2 : distinct UML!Transition foreach(trans in
                   s.createTransitionsForConfig())(
16                 /*gestion des transitions locales*/),
17             grd : distinct UML!Constraint foreach(trans in
                   s.createTransitionsForConfig())(
18                 /*...*/),
19             act : distinct UML!FunctionBehavior foreach (trans in
                   s.createTransitionsForConfig())(
20                 /*...*/),
21             trig : distinct UML!Trigger foreach (trans in
                   s.createTransitionsForConfig())(
22                 /*...*/),
23     }
```

Figure 4.8 Extrait de la règle de transformation d'un And-state

La règle de la Figure 4.8 transforme un And-state en Or-state, en modifiant l'état, en lui affectant les configurations créées à partir du helper et en gérant les transitions locales suivant les règles précisées dans l'approche. Sans rentrer dans les détails de chaque élément transformé, cette règle de transformation s'appuie sur le helper qui crée les configurations (Figure 4.9), et un helper qui gère les transitions locales (ligne 15). Une fois créé, chaque élément est affecté en suivant les consignes définies dans l'approche (par exemple ne changer que d'un état à travers une transition locale préexistante, entre deux configurations). Pour cette règle, la gestion des transitions locales s'est révélée être une tâche très complexe. En effet, l'attribution des sources et cibles de ces transitions lors de la transformation dépend du résultat de la création

des configurations. Il faut donc réaliser la gestion des transitions dans la même règle qui crée les configurations. Cependant plusieurs transitions doivent être créées pour une configuration donnée, et ATL n'est pas vraiment optimisé pour ce genre de tâche. Pour cela nous avons utilisé une boucle qui crée des transitions distinctes (ligne 15), prenant en paramètre la liste des configurations, en se basant sur les consignes précitées.

```
1 helper context UML!State def : getConfigurationAndStates() :
  Sequence(Sequence(UML!State)) =
2     self.region.at(1).subvertex->select(stat|stat.oclIsTypeOf(UML!State) or
        stat.oclIsTypeOf(UML!FinalState))
3     ->iterate(states; acc : Sequence(UML!State)=Sequence|
4     acc.union(self.region.at(2).subvertex->collect(st|st)
5         ->select(stat|stat.oclIsTypeOf(UML!State) or
            stat.oclIsTypeOf(UML!FinalState))
6         ->asSequence()->iterate(state; accu : Sequence(UML!State)=Sequence|
7         accu->append(Sequence->append(state)->append(states)))));
```

Figure 4.9 Helper créant les configurations provenant d'un And-state

4.3.4 La transformation des Or-states

Cette étape s'intègre également dans la partie flattening de l'approche. Le fichier de transformation contient un helper et quatre règles. Le helper, nommé `getFirstState()`, permet de récupérer l'état qui est est lié au pseudo-état initial dans le Or-state. Les règles permettent de traiter les Or-states, ainsi que les transitions qui y rentrent et qui en sortent. À cette étape tout les And-states sont devenus des Or-states. En premier lieu il est nécessaire de traiter les sous-états du Or-state. La règle `Substate2State` présentée dans la Figure 4.10 permet cela. Les lignes 3 à 18 permettent de s'assurer que nous traitons bien un sous état d'un état composite à une seule région. Ensuite les lignes 20 et 21 permettent de transformer cet état en un nouvel état dont le nom est la concaténation du nom du sous état et du nom de l'état composite.

Pour traiter les transitions externes à un Or-state, il y a trois règles. D'abord pour différencier transition entrante et sortante, mais aussi pour traiter le cas où la transition entrante d'un

```
1  rule Substate2State{
2      from
3          s : UML !State (if not s.container.oclIsUndefined()
4              then
5                  if not s.container.state.oclIsUndefined()
6                      then
7                          if s.container.state.isComposite()
8                      then
9                          not s.container.state.isOrthogonal()
10                     else
11                         false
12                     endif
13                     else
14                         false
15                     endif
16                     else
17                         false
18                     endif)
19         to
20             t : UML !State(
21                 name <-s.container.name + s.name,
22                 container<-s.container.state.container)}
```

Figure 4.10 Règle de transformation d'un Or-state

des états composites est la transition sortante d'un des autres. La Figure 4.11 présente la règle `IncomingOrTransi2Transi` qui permet par exemple de créer une transition à partir d'une transition entrante vers le Or-state. Les éléments des lignes 2 à 12 permettent de vérifier que l'on applique bien la règle à une transition allant d'un état simple vers un état composite (une autre règle gère le cas de deux états composites connectés entre eux). La seconde partie (lignes 13 à 27) permet de créer la nouvelle transition, en la connectant non plus à l'état composite mais au premier sous état (grâce au helper getFirstState() ligne 23).

4.3.5 Le mapping

Le fichier ATL de cette étape contient cinq règles et deux helpers. Ces derniers permettant de créer les listes *Lsend* et *Lreceive*. Le helper `getLsendValue()`, présenté dans la Figure 4.12 permet de créer la liste regroupant l'ensemble des événements envoyés par la machine,

```
1  rule IncomingOrTransi2Transi{
2      from
3          s :UML !Transition(if not s.target.oclIsUndefined()
4              then if s.target.isComposite()
5                  then if not s.source.oclIsUndefined()
6                      then if not s.source.oclIsTypeOf(UML !Pseudostate)
7                          then
8                              s.source.isSimple()
9                          else true endif
10                     else false endif
11                 else false endif
12             else false endif)
13         to
14             t1 : UML !Constraint(
15                 name<-s.getGuardNames()),
16             t2 : UML !Trigger(
17                 name<-s.trigger->first().name),
18             t3 : UML !FunctionBehavior(
19                 name<-s.getActionNames()),
20             t :UML !Transition(
21                 name<-s.name,
22                 source<-s.source,
23                 target<-s.target.getFirstState(),
24                 trigger<-t2,
25                 guard<-t1,
26                 effect<-t3,
27                 container<-s.container)}
```

Figure 4.11 Règle de transformation d'une transition entrante dans un Or-state

nécessaire pour construire notre réseau de CEFSM. La première partie du code, des lignes 2 à 12, collecte les événements (envoyés à travers des `effects`), et la seconde (ligne 13 et 14) les ajoute dans la liste en éliminant les doublons.

Les cinq règles implémentant le mapping sont assez simples. Trois règles permettent de transformer les états UML en états EFSM : une pour les états normaux, une pour l'état initial et une pour les états finaux. Une règle transforme les transitions UML en transitions EFSM, et la dernière règle transforme la machine à états UML en EFSM.

```
1  helper context UML !Region def : getLsendValues() : Sequence(String) =
2      self.transition->collect(tlt.effect)->select(efif not ef.oclIsUndefined()
3          then
4              if not ef.name.oclIsUndefined()
5              then
6                  ef.name.startsWith("SEND")
7              else
8                  false
9              endif
10             else
11                 false
12             endif)
13         ->iterate(effect ; accum : Sequence(String) = Sequence |
14         accum.append(effect.name)) ;
```

Figure 4.12 Helper créant la liste des signaux envoyés par la machine

4.3.6 Démonstration de STEF pour UML

STEF pour UML a été développé comme un plug-in Eclipse. La machine UML à transformer doit être créée en conformité avec le méta modèle UML implémenté dans le plug-in UML2. Il est possible de modéliser la machine directement en utilisant le format XMI supporté par EMF ou graphiquement, avec l'outil *Papyrus*.

A des fins de démonstration, nous avons choisi un système dont le comportement est communément représenté en utilisant une machine à états UML : un `ATM` (Baouya *et al.*, 2015; Felderer & Herrmann, 2015). La Figure 4.13 présente une machine à états modélisant le comportement d'un ATM. Cette machine regroupe différents concepts du langage UML, par exemple un état composite (`Serving Customer`), et des activités internes (comme `readCard()`). La Figure 4.14 présente une vue sous forme d'arbre du fichier XMI, correspondant à la machine UML de l'ATM, créée par l'outil Ecore.

Une fois la machine modélisée, il suffit de faire un clic droit sur le fichier contenant la machine et de choisir l'option `Transform Model` (voir Annexe II). Une fenêtre va s'ouvrir et va proposer de choisir le dossier cible dans lequel le résultat va être généré. Le résultat peut être

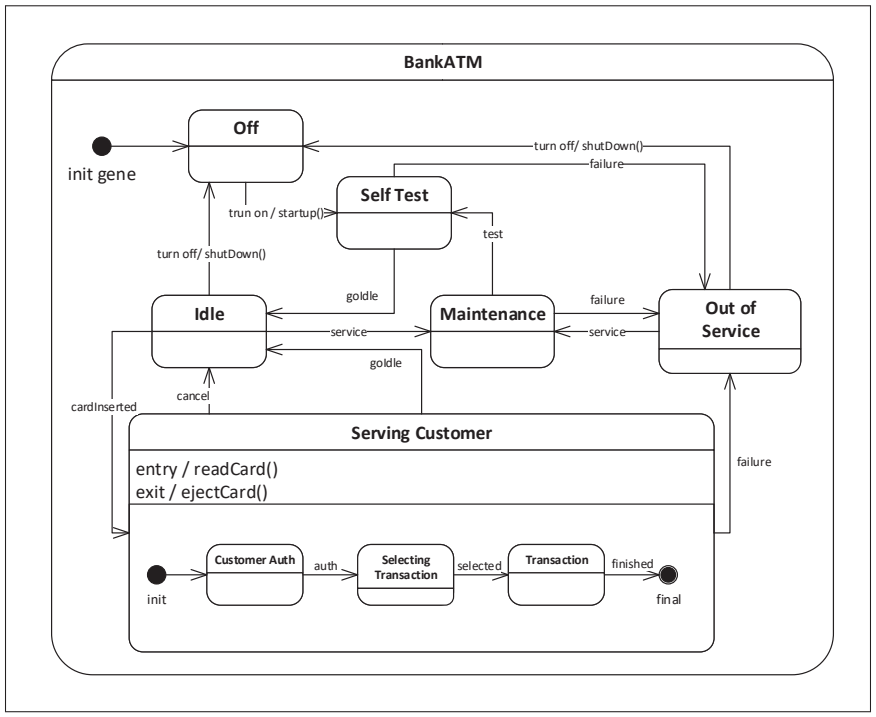


Figure 4.13 La machine à états `BankATM`

visualisé : le plug-in Sirius nous a permis de créer une représentation graphique des éléments du méta modèle EFSM. La Figure 4.15 présente le résultat de la transformation obtenu avec STEF pour UML.

4.4 Implémentation de l'approche STEF pour Stateflow

L'implémentation de STEF pour le langage Stateflow a résulté en cinq transformations ATL (5 fichiers ".atl"). Deux transformations ont été implémentées pour supporter la phase de Preprocessing : une pour les jonctions connectives et une pour les actions internes. Pour le Flattening, nous avons implémenté aussi deux transformations : une pour les And-states et une pour les Or-states. La cinquième transformation implémente le Mapping.

La Figure 4.16 présente une vue du contenu du fichier "build.xml". Ce fichier permet de charger le modèle source au format Stateflow (e.g., `input.simulink`), d'appliquer les fichiers ATL de transformation dans l'ordre (depuis `TransformJunction.atl` jusqu'à `Mapping.atl`) au modèle

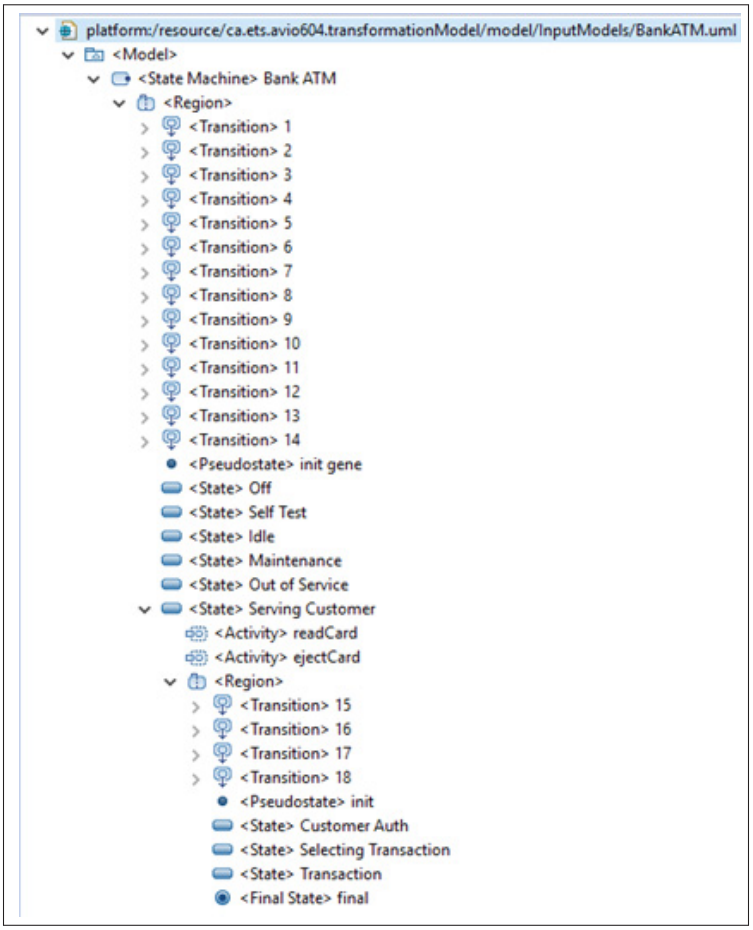


Figure 4.14 La machine `BankATM` modélisée avec l'outil Ecore

à transformer et aux modèles intermédiaires, et enfin de sauvegarder le modèle final obtenu qui est conforme au méta modèle EFSM (`result.efsm`).

Là encore, durant l'implémentation de STEF pour Stateflow nous avons dû définir un certain nombre de directives qu'un concepteur doit respecter lors de la création des machines à états Stateflow. Les directives qui relèvent des pratiques des partenaires industriels sont les mêmes (e.g., la hiérarchie des états doit se limiter à trois niveaux). D'autres directives ont été relevées depuis la spécification du langage Stateflow. Par exemple, la spécification déconseille l'utilisation du backtracking.

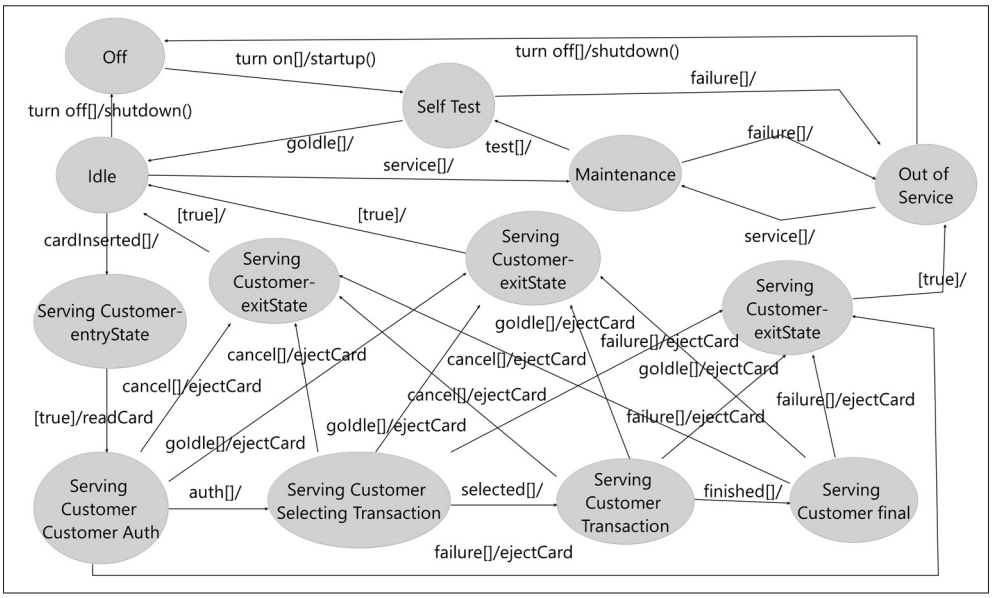


Figure 4.15 L'EFSM résultant de la transformation de la machine `BankATM`

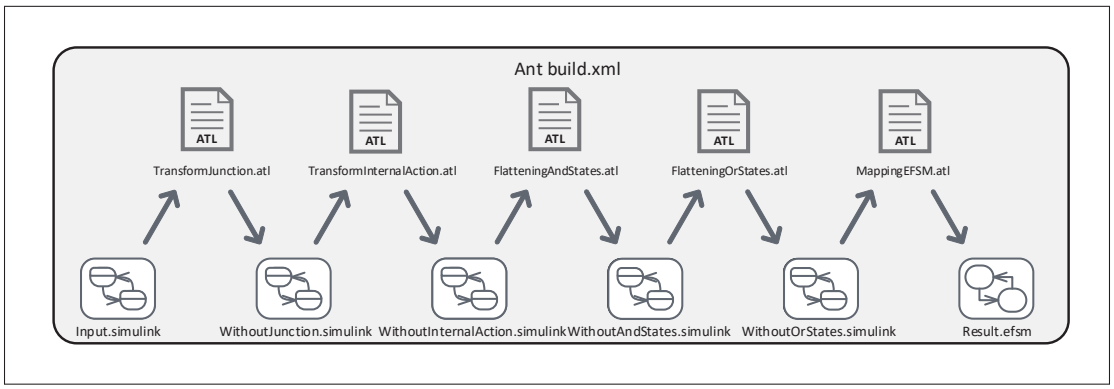


Figure 4.16 L'implémentation de STEF pour Stateflow dans Eclipse

4.4.1 Transformation des Jonctions Connectives

Cette étape de la transformation s'occupe de transformer les jonctions connectives existantes dans la machine. Cette étape contient quatre règles. Une règle qui traite la transformation de la jonction avec boucle for, deux règles qui traitent le cas de la fusion et de la séparation des transitions. Et enfin, une dernière règle traite le cas d'une jonction if-then-else. La Figure 4.17

présente la règle de transformation d'une jonction avec boucle for. Pour rappel cette jonction a une transition vers elle même, avec les paramètres d'une boucle. On s'assure donc d'appliquer la règle sur une transition allant d'une jonction à elle même (lignes 2 à 13). La règle crée une nouvelle transition (lignes 21 à 27) contenant une condition et un événement correspondant aux items évènement/condition de la transition entrante dans la jonction (lignes 15 à 18) ainsi qu'une action *boucleFor* contenant les items de la boucle for précédemment définie (ligne 19 à 20).

4.4.2 Transformation des actions internes

Cette seconde étape transforme les actions internes aux états du langage Stateflow. Les actions *entry*, *during*, *exit* et *onAction* y sont traitées. Le fichier ATL de transformation contient sept règles de transformation : quatre règles pour transformer les activités, et trois autres pour gérer les cas particuliers qui sont similaires à ceux du langage UML. Pour appliquer la règle adéquate, on identifie d'abord le type d'action contenue dans l'état. Par exemple, la Figure 4.18 présente la règle `duringAction2Transition` qui transforme l'action *during* en une transition de l'état vers lui-même. Les lignes 3 à 8 permettent de sélectionner l'action sur laquelle la règle est appliquée. La suite de la règle transforme cette action *during* en une transition en lui affectant les items adéquats : événement *null*, condition à *true* et action équivalente.

4.4.3 Transformation des And-states, Or-states et Mapping

Les trois fichiers de transformation pour ces trois étapes sont similaires à ceux implémentant STEF pour UML pour les mêmes étapes.

4.4.4 Démonstration de STEF pour Stateflow

STEF pour Stateflow a été développé de la même façon que STEF pour UML : c'est un plug-in Eclipse. D'abord la machine est modélisée en utilisant le langage Stateflow dans l'outil Matlab. Le modèle est ensuite importé sous forme de fichier Ecore grâce à un plug-in développé

```
1  rule junctionFor2ActionInTransition{
2      from
3          s : STF!Transition(if s.source.oclIsTypeOf(STF!Junction)
4              then
5                  if s.destination.oclIsTypeOf(STF!Junction)
6                  then
7                      s.source=s.destination
8                  else
9                      false
10                 endif
11             else
12                 false
13             endif)
14         to
15             tGuard : STF!SFWGuard(
16                 statement<-s.source.incomingTransitions->flatten()
                     ->select(trans|trans.source.oclIsTypeOf(STF!State))-
                     >first().guard.statement),
17             tTrigger : STF!SFWTrigger(
18                 statement<-s.source.incomingTransitions->flatten()
                     ->select(trans|trans.source.oclIsTypeOf(STF!State))-
                     >first().trigger.statement),
19             tAction : STF!Action(
20                 statement<-"boucleFor" + s.guard.statement + "," +
                     s.triggeredActions->first().statement + ""),
21             t : STF!Transition(
22                 parent<-s.parent,
23                 source<-s.source.incomingTransitions->flatten()
                     ->select(trans|trans.source.oclIsTypeOf(STF!State))->first().source,
24                 destination<-s.source.outgoingTransitions->flatten()
                     ->select(trans|trans.destination.oclIsTypeOf(STF!State))-
                     >first().destination,
25                 trigger<-tTrigger,
26                 guard<-tGuard,
27                 triggeredActions<-tAction)}
```

Figure 4.17 Règle de transformation d'une jonction avec boucle For

dans le cadre de l'étude de Paz & Boussaidi (2019). Ici nous avons transformé la machine `SequenceController` présentée dans Figure 4.19, extrait de Paz & Boussaidi (2018).

```
1  rule duringAction2Transition{
2      from
3          s :STF !Action(if not s.stateDuring.oclIsUndefined()
4              then
5                  true
6              else
7                  false
8              endif)
9          to
10             t1Trigger :STF !SFWTrigger(
11                 statement<-"null"),
12             t1Guard :STF !SFWGuard(
13                 statement<-"true"),
14             t1Action :STF !Action(
15                 statement<-s.statement),
16             t :STF !Transition(
17                 parent<-s.stateDuring.parent,
18                 source<-s.stateDuring,
19                 destination<-s.stateDuring,
20                 guard<-t1Guard,
21                 trigger<-t1Trigger
22                 triggeredActions<-t1Action)}
```

Figure 4.18 Règle de transformation d'une action interne *during*

Une fois la machine modélisée et importée au format Ecore, il suffit de faire un clic droit sur le fichier contenant la machine et de choisir l'option `Transform Model` (voir Annexe II). Une fenêtre va s'ouvrir et va proposer de choisir le dossier cible dans lequel le résultat va être généré. Le résultat peut être visualisé : le plug-in `Sirius` nous a permis de créer une représentation graphique des éléments du méta modèle EFSM. La Figure 4.20 présente le résultat de la transformation obtenu avec l'outil.

4.5 Étude de cas

L'application de l'approche STEF aux langages UML et Stateflow démontre la faisabilité de l'approche. Nous avons donc souhaité vérifier la correction des résultats donnés par les algorithmes de transformation que nous avons définis pour UML et Stateflow. Pour ce faire, nous avons procédé à une étude de cas impliquant différentes machines à états et des partici-

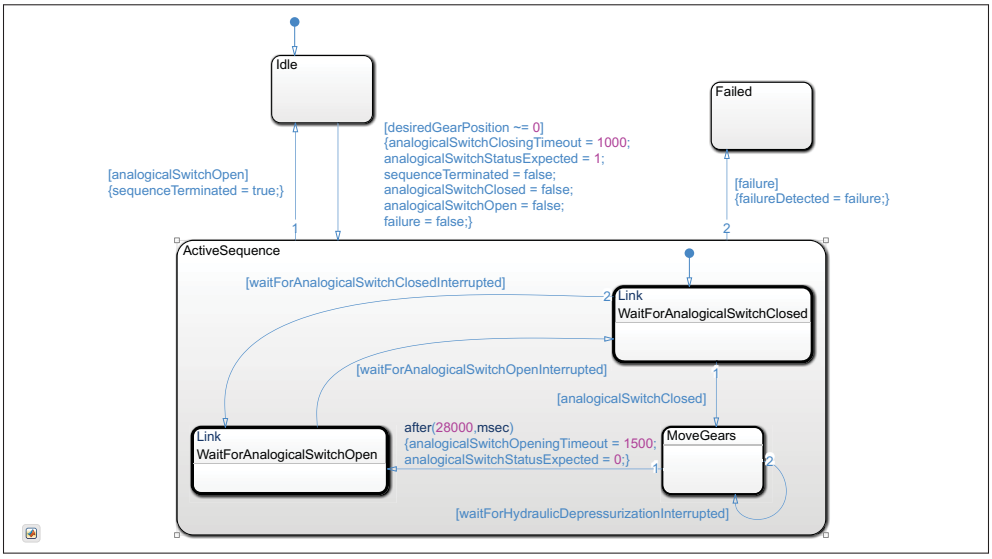


Figure 4.19 La machine à états Stateflow SequenceController, extrait de Paz & Boussaidi (2018)

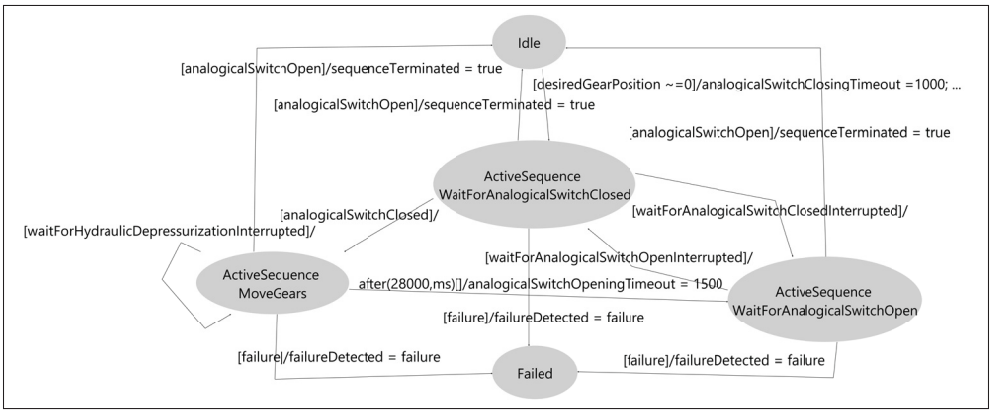


Figure 4.20 L'EFSM résultant de la transformation de la machine SequenceController

pants. Nous nous sommes concentrés sur le langage UML comme plusieurs algorithmes sont communs à UML et Stateflow. Dans ce qui suit, on décrit en premier notre configuration expérimentale. Ensuite nous présentons et discutons les résultats de notre expérimentation.

4.5.1 Objectifs et configuration de l'expérimentation

L'objectif de notre expérimentation avec STEF pour UML est de répondre aux deux questions suivantes :

1. Est-ce que STEF fournit des résultats corrects lorsque appliquée sur des machines à états UML bien formées ?
2. Comparée à une approche manuelle, est-ce que STEF permet d'améliorer la qualité des machines EFSM produites, et la productivité ?

Pour répondre à nos questions, nous avons recruté deux groupes de participants : un groupe de trois experts pour vérifier les résultats de notre outil et un groupe de trois utilisateurs pour comparer STEF à une approche manuelle. Nous avons recruté le groupe d'experts parmi les ingénieurs et chercheurs impliqués dans le projet AVIO604. Pour le groupe d'utilisateurs, nous avons recruté des étudiants de niveau maîtrise et doctorat en génie logiciel. Ces derniers ont alors participé à un court séminaire portant sur les machines à états UML et EFSM. Pour connaître le profil des participants nous avons réalisé un sondage auprès d'eux. Le sondage contenait les questions suivantes :

A) Votre domaine est-il lié à l'ingénierie dirigée par les modèles ?

B) A quel point estimez vous connaître le langage UML ?

C) A quel point estimez vous connaître les machines à états UML ?

D) A quel point estimez vous connaître d'autres formalismes de machines à états étendues ?

E) Connaissez vous les techniques de transformation de modèles ?

Les valeurs des réponses aux questions sont basées sur l'échelle de Likert avec cinq valeurs allant de `pas du tout` (score de 1) à `très bien` (score de 5). Les résultats du sondage sont compilés dans le Tableau 4.1. On remarque que le travail de tous les participants est relié à la pratique de l'IDM et qu'ils connaissent tous bien le langage UML. Cependant, l'expérience

dans d'autres formalismes de machines à états et dans les transformations de modèles, est beaucoup plus disparate (surtout pour le groupe des utilisateurs). Concernant les experts, ils ont été choisi pour leurs connaissances dans ces domaines, et c'est ce que reflètent les résultats du questionnaire.

Tableau 4.1 Profil des participants

Personne	**A**	**B**	**C**	**D**	**E**
Utilisateur 1	5	5	5	5	5
Utilisateur 2	5	5	4	2	2
Utilisateur 3	4	3	3	1	3
Expert 1	5	5	5	5	5
Expert 2	5	5	4	5	5
Expert 3	5	5	5	4	4

Nous avons aussi sélectionné un ensemble de six machines à états UML. Ces machines décrivent les comportements de différents systèmes qui sont tous du domaine de l'aéronautique. Ces machines ont été extraites et adaptées soit de l'étude de cas de Paz & Boussaidi (2018), ou des exemples fournis dans la documentation de l'outil Matlab (Mosterman & Ghidella, 2018). Chaque participant s'est vu attribuer deux machines à états UML différentes : une machine à états simple (e.g., la machine `RetractGears` de la Figure 4.21) et une machine complexe (e.g., la machine `ElevatorFailureDetection` de la Figure 4.22). Les autres machines utilisées pour l'étude de cas sont présentées dans l'Annexe I. Afin d'estimer la complexité des différentes machines, nous leur avons assigné un score de complexité, en se basant sur les règles suivantes :

- Chaque état et chaque transition augmente le score de complexité de 1.
- Chaque région augmente le score de 2.
- Chaque pseudo-état augmente le score de 2.
- Chaque activité interne augmente le score de 2.
- Chaque échange de signal (a travers SEND et RECEIVE) augmente le score de 2.

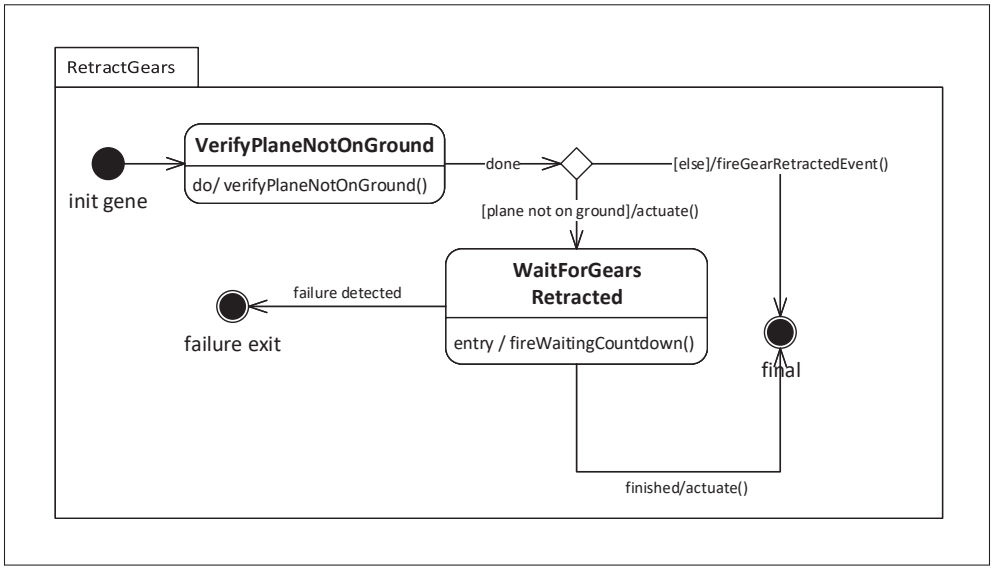


Figure 4.21 Machine à états UML RetractGears, adaptée de Paz & Boussaidi (2018)

Le Tableau 4.2 présente les machines avec leurs scores de complexité respectifs, ainsi que leur taille en nombre d'éléments. La taille est calculée en dénombrant le nombre d'éléments dans la machine (e.g., états, transitions, pseudo-états).

Tableau 4.2 Machines utilisées pour l'expérimentation

Machine à états	Score de complexité	Taille en nombre d'éléments
RetractGears	19	12
WaitForAnalogicalSwitchClosed	20	17
ExtendGears	21	13
ElevatorFailureDetection	47	23
WaitForHydraulicPressure	47	29
ValidSensorData	48	37

Ainsi pour répondre à la première question, nous avons transformé les six machines à états UML en utilisant le plug-in STEF pour UML. Nous avons ensuite soumis les résultats au groupe d'experts pour vérification ; chaque expert a reçu les deux EFSM correspondant aux deux machines UML qui lui ont été attribuées. Avant de vérifier la totalité d'une machine, les experts vérifient des extraits de celle-ci (par exemple le résultat EFSM obtenu depuis un

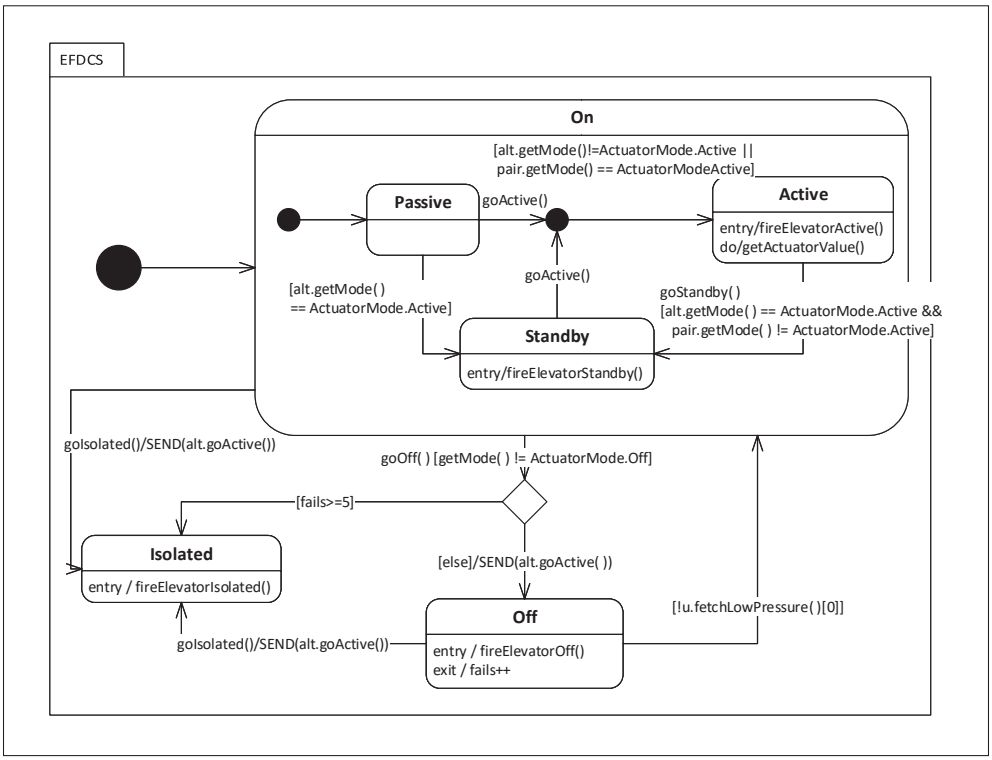


Figure 4.22 Machine à états UML ElevatorFailureDetection, adaptée de Mosterman & Ghidella (2018)

pseudo-état particulier dans une machine à états UML). Ceci permet de guider et faciliter la vérification. Pour répondre à la deuxième question, chaque membre du groupe des utilisateurs a transformé manuellement les deux machines UML qui lui ont été attribuées. Les EFSM résultants ont alors été vérifiées par nos soins.

Dans les deux cas, la vérification des EFSM s'est faite selon les critères suivants :

C1 Il n'existe plus de hiérarchie (état composite), de pseudo-état ni d'activité interne (activité interne aux états).

C2 Les états simples et les transitions entre eux dans les EFSM peuvent être traçés au modèle source.

C3 Les états composites transformés peuvent être traçés au modèle source.

C4 Les pseudo-états transformés peuvent être traçés au modèle source.

C5 Les activités internes transformées peuvent être traçées au modèle source.

C6 La sémantique d'exécution est préservée à travers la transformation de la machine.

4.5.2 Analyse des résultats pour la question 1

Les résultats de la vérification des EFSM produits par l'outil sont compilés dans le Tableau 4.3. Le taux de succès est calculé en comparant individuellement chaque élément de la machine à états UML source avec l'EFSM obtenu. Chaque élément qui n'est pas traduit ou qui est mal traduit diminue le taux de succès. le pourcentage est obtenu avec la formule suivante :

$$TauxSucces = \frac{NbElementBienTraduitVersEFSM}{TailleMachineSource} \tag{4.1}$$

L'outil fournit le résultat pour chaque machine en entrée en moins d'une seconde. Ceci améliore grandement la performance comparé à une transformation faite manuellement (voir sous section suivante). Après avoir été vérifiés par les experts, les EFSM résultant montrent un taux de succès moyen très élevé (**99,3%**). Une seule erreur a été relevée : c'est la non traduction d'une activité interne à un sous état de la machine `ElevatorFailureDetection`. Cette erreur a ensuite été corrigée. Les experts ont validé l'ensemble des machines, notamment en attestant du respect de l'ordre d'exécution dans la transformation pour l'ensemble des concepts transformés.

Les résultats montrent que l'ensemble des éléments transformés peuvent être reliés à la machine d'origine. À titre d'exemple, tous les pseudo-états ont pu être reliés à leurs constructions équivalentes dans la machine cible. Il en est de même pour tous les éléments de la hiérarchie. Enfin, l'ensemble des experts a validé la préservation de la sémantique d'exécution pour l'ensemble des machines.

Tableau 4.3 Résultats de la vérification des EFSM par les experts

Machine à états	Taux succès outil
RetractGears	100%
WaitForAnalogicalSwitchClosed	100%
ExtendGears	100%
ElevatorFailureDetection	96%
WaitForHydraulicPressure	100%
ValidSensorData	100%

4.5.3 Analyse des résultats pour la question 2

L'analyse des résultats du groupe d'utilisateur est présentée dans le Tableau 4.4. Le taux de succés est calculé selon la même formule que celle utilisée pour la vérification des EFSM obtenus avec l'outil STEF. Tous les utilisateurs ont réussi à réaliser la tâche en 50 minutes en moyenne. Le taux moyen de succès pour la transformation des machines à états en EFSM est de **72,5%**.

En effet, les participants ont tendance à avoir une compréhension biaisée de la sémantique de certains concepts. Le flattening des machines s'est révélé être particulièrement complexe. La compréhension de l'exécution d'un état composite est mauvaise, surtout dans le cas d'un And-state. Par exemple, les régions parallèles sont vues comme deux threads totalement séparés. Cependant, les pseudo-états dans l'ensemble ont été plutôt bien interprétés dans la traduction vers les EFSM. Cependant, nous avons relevé une erreur récurrente : les utilisateurs ont tendance à interpréter un noeud de décision et de jonction de la même manière. Cela fait perdre le caractère dynamique de l'évaluation des transitions sortantes. Enfin, les activités internes ont été un problème pour la plupart des participants : elles sont souvent traduites par un déplacement sur une transition entrante ou sortante. Toutes ces erreurs amènent à une perte plus ou moins importante dans la traduction de l'ordre d'exécution de la machine.

Ainsi, ces résultats nous amènent aux conclusions suivantes :

- Les résultats dépendent grandement de l'expérience du participant avec le formalisme.

- Les participants ont une compréhension biaisée de certains concepts des machines à états UML, notamment en ce qui concerne la hiérarchie.
- Même si on a insisté lors du séminaire donné aux participants sur l'importance de respecter l'ordre d'exécution de la machine lors de la transformation, les utilisateurs n'y ont pas prêté assez attention. C'est le problème le plus grave, car c'est un des aspects les plus importants pour la génération de cas de test.

Tableau 4.4 Résultats de la vérification des EFSM produits manuellement

Machine à états	**Taux succès utilisateur**
RetractGears	92%
WaitForAnalogicalSwitchClosed	54%
ExtendGears	63%
ElevatorFailureDetection	91%
WaitForHydraulicPressure	54%
ValidSensorData	84%

4.5.4 Menaces pour la validité

La validité des résultats et conclusions de notre étude de cas est limitée par certaines menaces que nous avons essayé d'atténuer. D'abord le nombre de participants pour l'étude de cas est assez limité (seulement 6). Bien qu'en nombre limité, les participants choisis ont tous une expérience significative en ingénierie dirigée par les modèles, et connaissent le langage UML. De plus, une formation a été donnée avant l'exercice afin d'uniformiser les profils. Ensuite, chaque participant a reçu deux machines : une simple et une complexe. La machine simple permettant de se familiariser avec la transformation avant de transformer la machine complexe.

Aussi le nombre de machines transformées est également faible. Mais bien qu'en nombre limité, ces machines sont extraites de cas réels, et couvrent l'ensemble des concepts que notre approche se propose de transformer. Chaque concept est représenté dans un nombre varié de constructions.

En ce qui concerne la vérification des résultats de STEF, là aussi le nombre limité de machines transformées rend difficile la généralisation de ces résultats. Pour limiter les biais, nous avons proposé un ensemble de règles de vérification strictes, et la vérification a été réalisée indépendamment, par un comité d'experts. Chaque expert s'est vu attribuer les deux machines à vérifier, ainsi que des extraits choisis des machines à états UML et de leurs traductions en EFSM, afin de guider la vérification.

CONCLUSION ET RECOMMANDATIONS

Contributions

Dans ce mémoire, nous avons proposé une approche générale permettant de transformer des machines à états décrites avec différents formalismes vers une représentation unifiée sous forme d'EFSM. Les EFSM sont exprimés dans un formalisme pour lequel les techniques de génération de test sont éprouvées. Nous avons appliqué notre approche avec succès à deux langages communément utilisés dans l'industrie aéronautique : *UML* et *Simulink/Stateflow*.

Notre approche possède les caractéristiques uniques suivantes :

- Notre approche est générique et peut donc s'appliquer à n'importe quel formalisme de machine à états. En effet le processus proposé par notre approche comprend 3 étapes génériques : Preprocessing, Flattening et Mapping. Selon le formalisme source le preprocessing et le flattening peuvent être optionnels.
- À travers le processus proposé, l'approche couvre l'ensemble des concepts et constructions que l'on peut trouver dans un formalisme de machine à états.
- C'est une approche à base de règles ce qui favorise la réutilisation. En effet, certaines règles de transformation sont communes à différents formalisme de machines à états. Par exemple, certaines règles spécifiant le processus de flattening et mapping sont réutilisable d'un formalisme à l'autre.
- L'approche peut être implémentée en utilisant différents outils et technologies de l'IDM. Ce qui la rend donc automatisable.
- L'approche se base sur des directives qu'un concepteur doit respecter lors de la création des machines à états. Certaines de ces directives reflètent les pratiques des partenaires industriels et visent à faciliter l'implémentation de l'approche.

Nous avons implémenté l'approche en utilisant l'environnement de développement Eclipse et cela sous la forme de plug-in pour UML et Stateflow. Afin de valider à la fois l'approche et l'outil développé pour UML, nous avons réalisé une étude de cas. Celle-ci s'appuie sur des exemples extraits de modèles représentant des systèmes aéronautiques réels. Cette étude de cas, faisant intervenir un ensemble d'experts du domaine pour la validation, montre d'excellents résultats pour la transformation faite par l'outil, en comparaison à une transformation réalisée manuellement (temps réduit de transformation, réduction des erreurs commises lors de la transformation).

Travaux futurs

Dans un futur proche, nous planifions de travailler sur les éléments suivants :

- **Étendre l'étude de cas** (court terme) : L'étude de cas proposée a porté sur un nombre limité de machines. Cela est grandement imputable au manque d'exemples disponibles pour des raison de sécurité et de confidentialité. L'outil a cependant été installé chez les partenaires industriels, et leurs retours nous permettront d'ajuster et/ou d'étendre l'outil.
- **Étendre le nombre de formalismes traités** (court à moyen terme) : l'approche a pour l'instant été appliquée à deux langages. Cependant, il existe d'autres formalismes utilisés dans le secteur aéronautique (e.g., Scade). L'application de l'approche à un nombre plus grand de formalismes permettrait de raffiner le processus proposé.
- **Intégrer la vérification des directives dans le processus de transformation** (moyen à long terme) : Dans l'état actuel, nous avons implémenté l'approche pour UML et Stateflow en tenant compte de directives reflétant les pratiques des partenaires industriels. Il serait possible d'encoder ces directives et de les vérifier de façon automatique. Cela permettrait de faire cette vérification avant de lancer le processus de transformation et de rendre l'implémentation de l'approche indépendante des directives.

ANNEXE I

LES AUTRES MACHINES DE L'ÉTUDE DE CAS

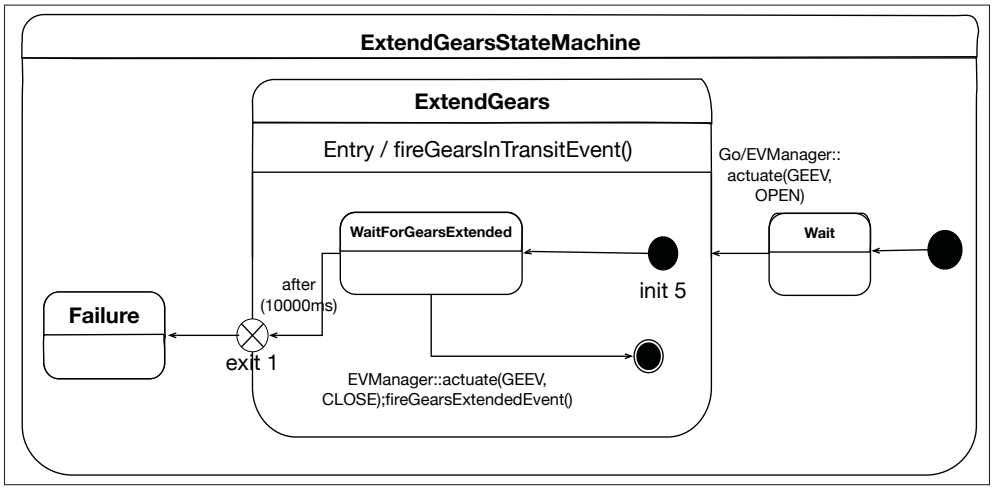


Figure-A I-1 Machine à états UML ExtendGears, adapté de Paz & Boussaidi (2018)

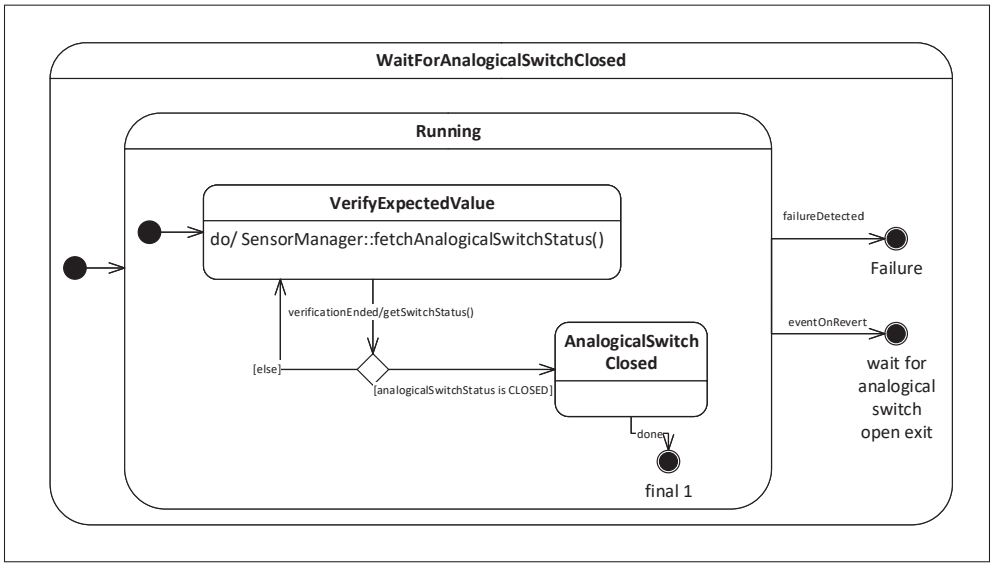


Figure-A I-2 Machine à états UML WaitForAnalogicalSwitchClosed, adapté de Paz & Boussaidi (2018)

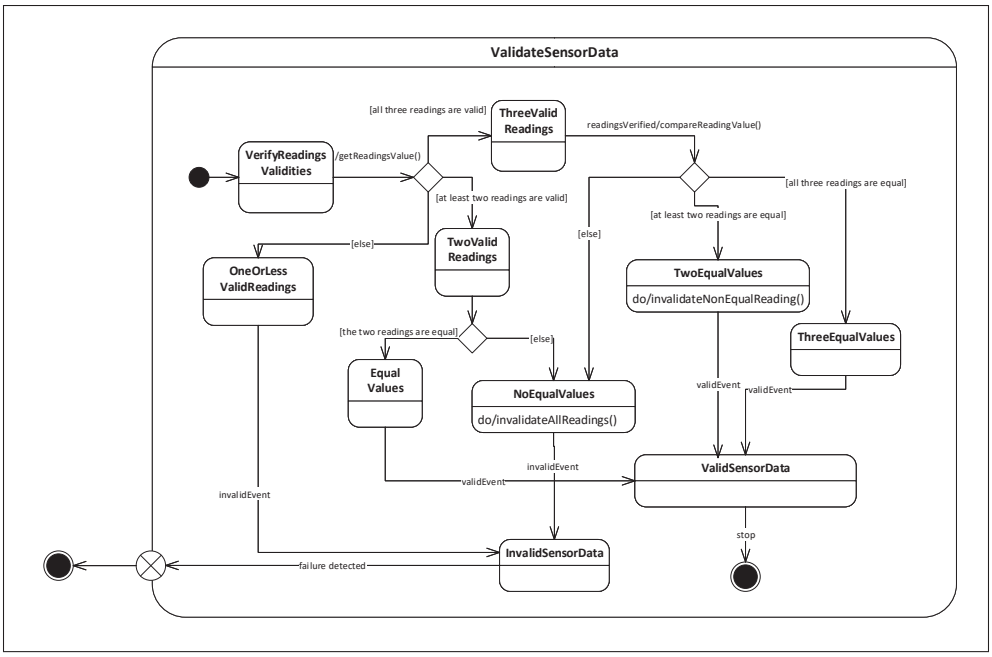


Figure-A I-3 Machine à états UML ValidateSensorData, adapté de Paz & Boussaidi (2018)

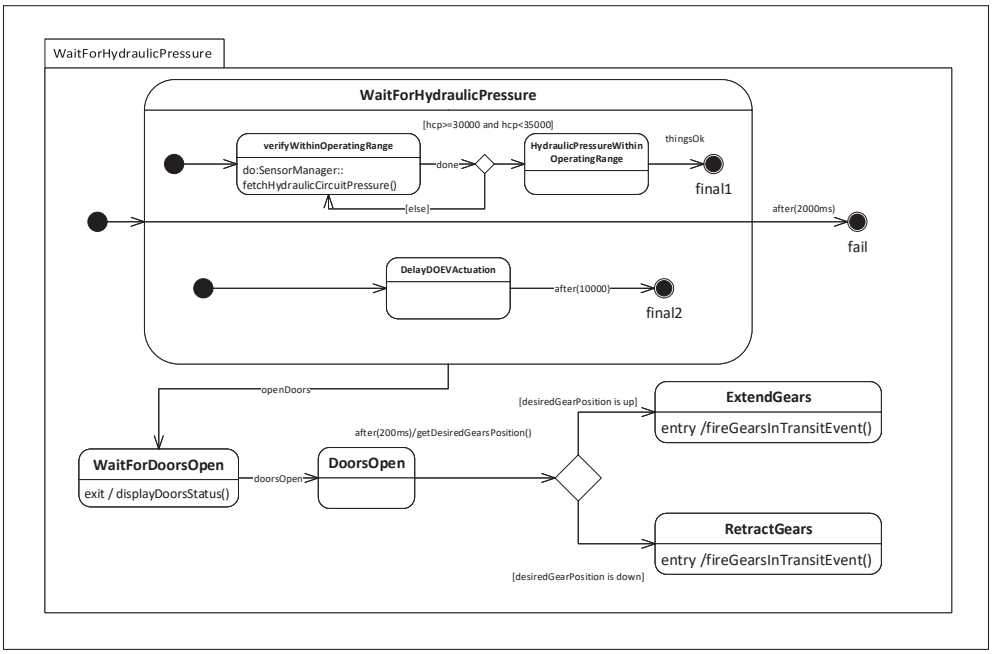


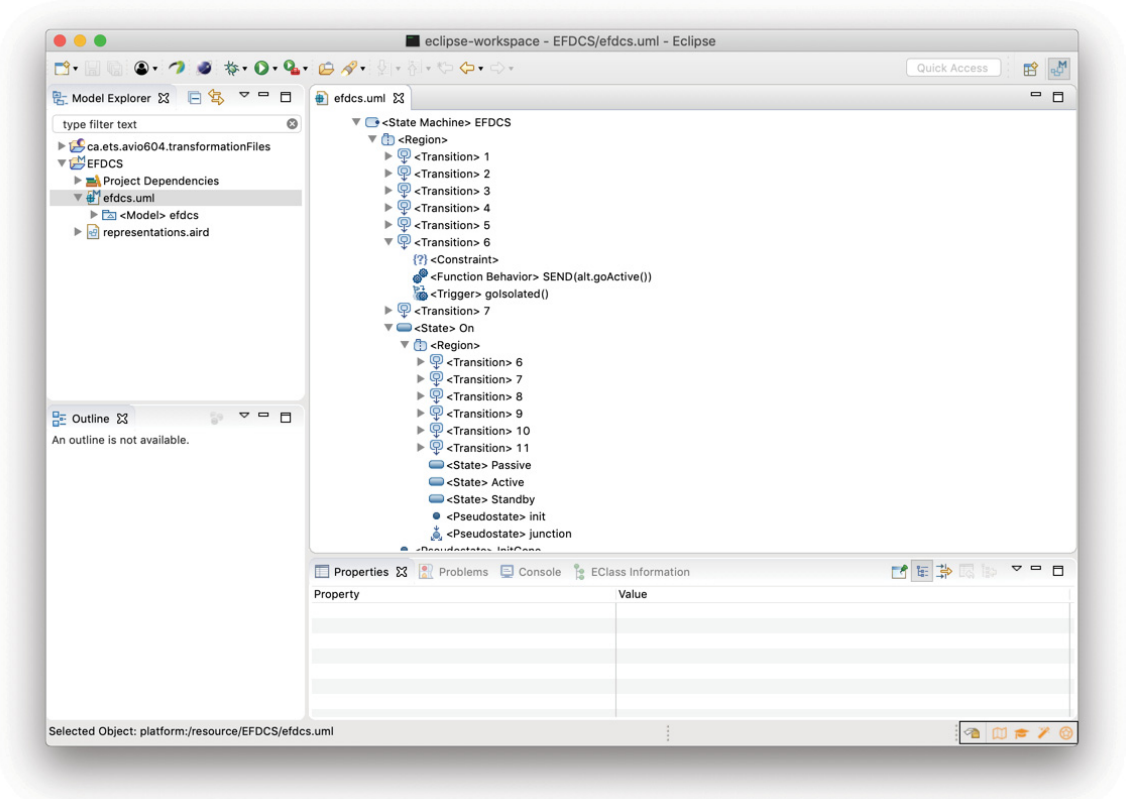
Figure-A I-4 Machine à états UML WaitForHydraulicPressure, adapté de Paz & Boussaidi (2018)

ANNEXE II

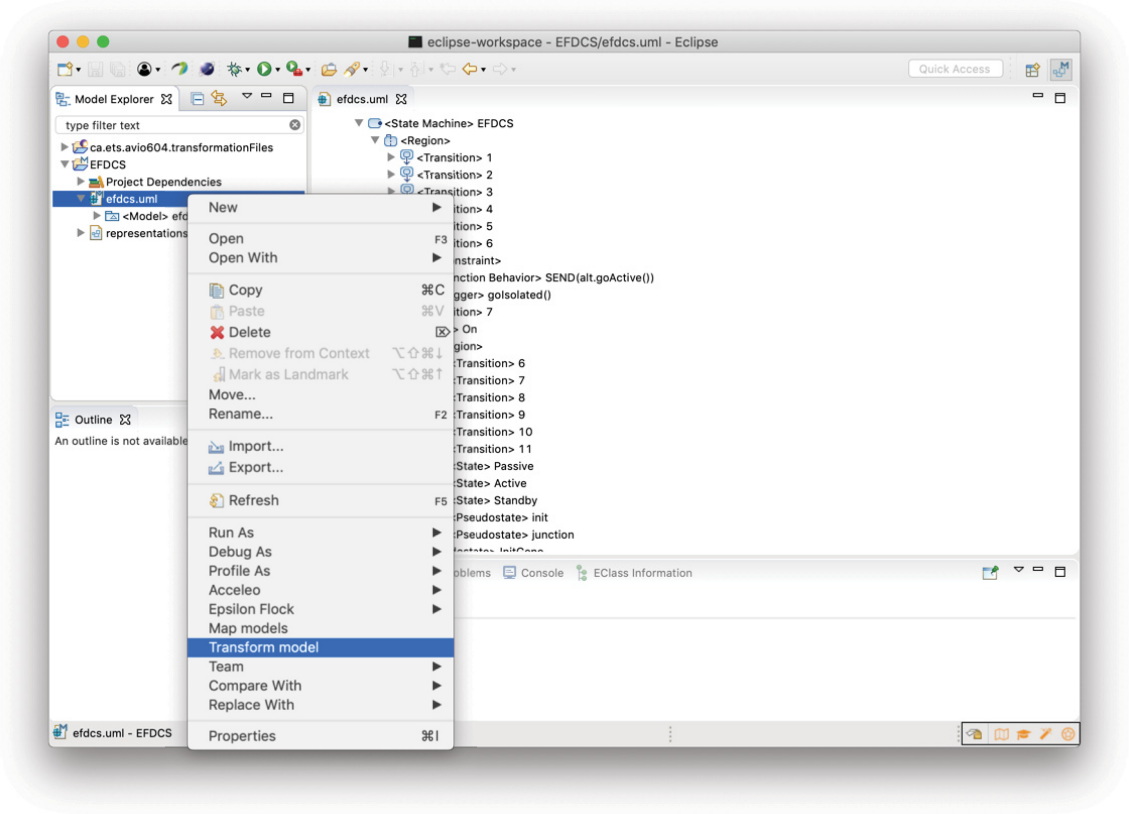
DESCRIPTION DE L'UTILISATION DU PLUG-IN *STEF*

Le guide suivant présente comment utiliser le plug-in STEF pour UML à l'intérieur de l'outil Eclipse. Le plug-in STEF pour Stateflow s'utilise exactement de la même manière. Dans ce qui suit on utilise l'exemple de la transformation de la machine `ElevatorFailureDetection (EFDCS)`

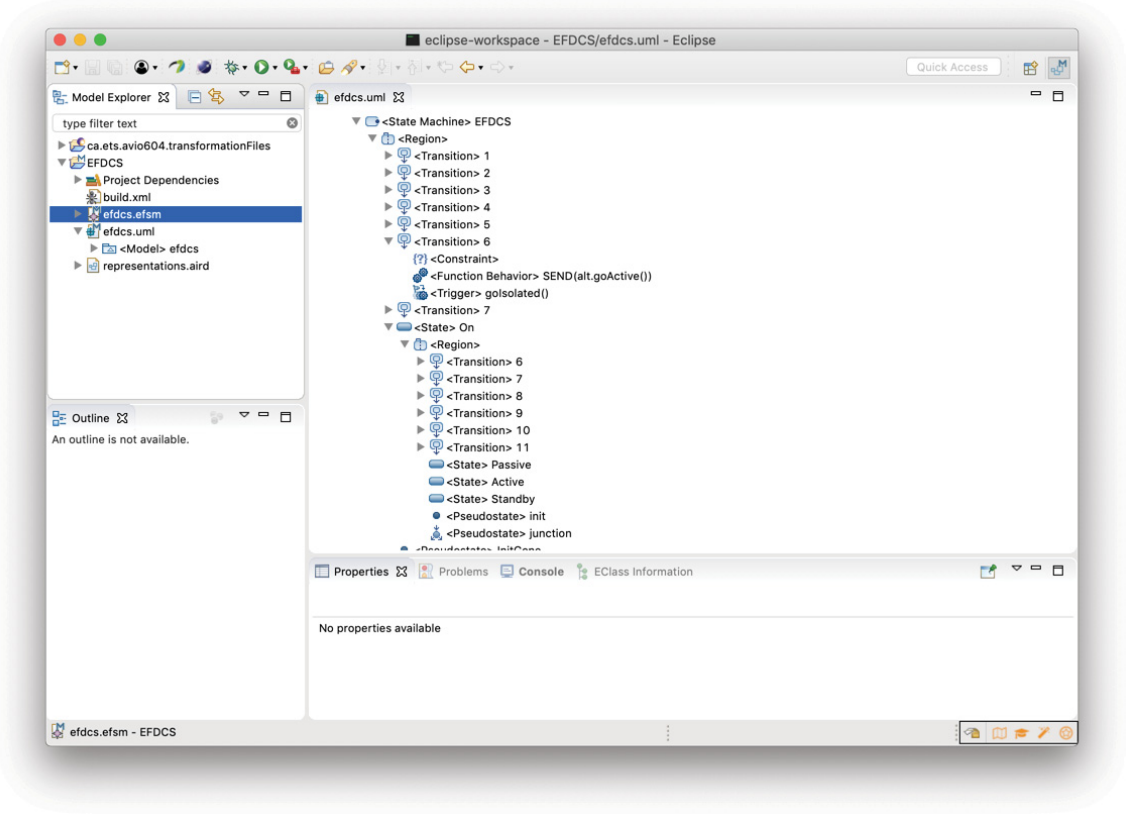
1. Trouver le fichier UML que vous souhaitez transformer. Dans notre cas `EFDCS.uml`. Celui-ci doit se trouver dans un projet de type *Modeling*. Il est également possible d'utiliser le plug-in **Papyrus for UML** afin de modéliser graphiquement la machine à états UML.



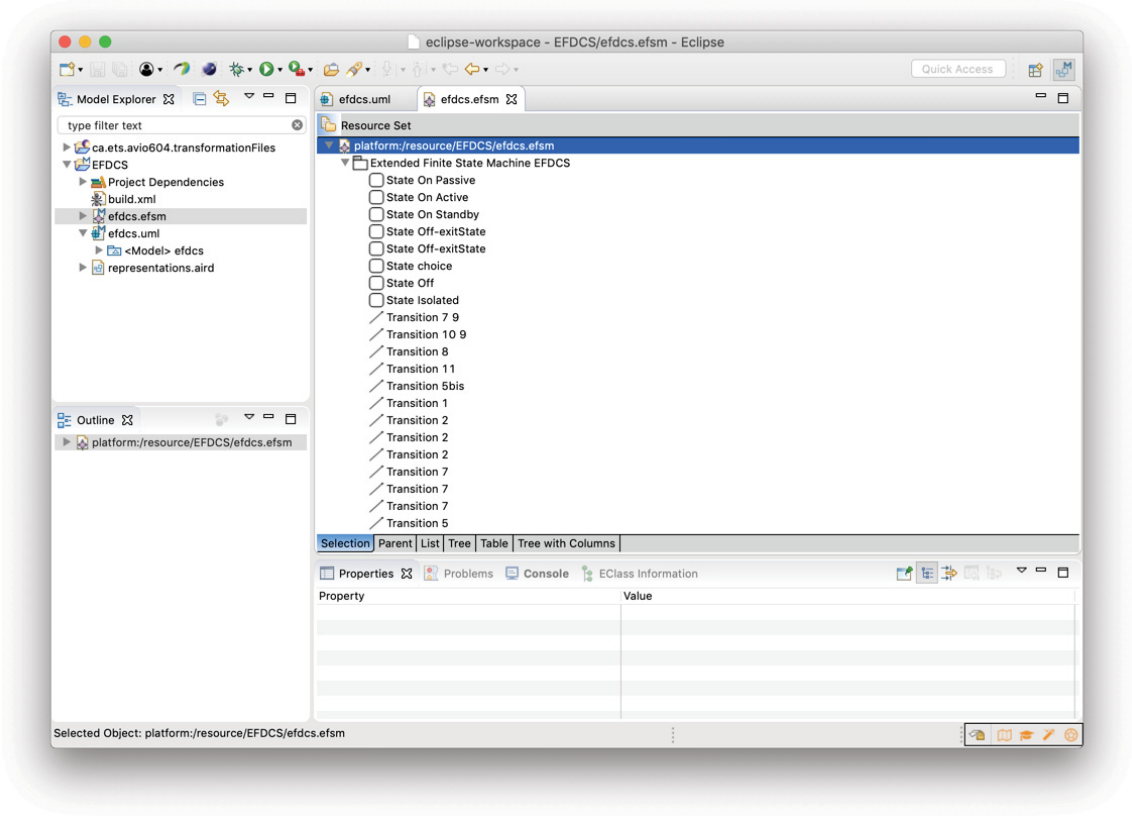
2. Ensuite, un clic droit sur le fichier UML en question permet de faire apparaître l'option **Transform model**. Cette option ne s'affiche que pour les fichiers UML.



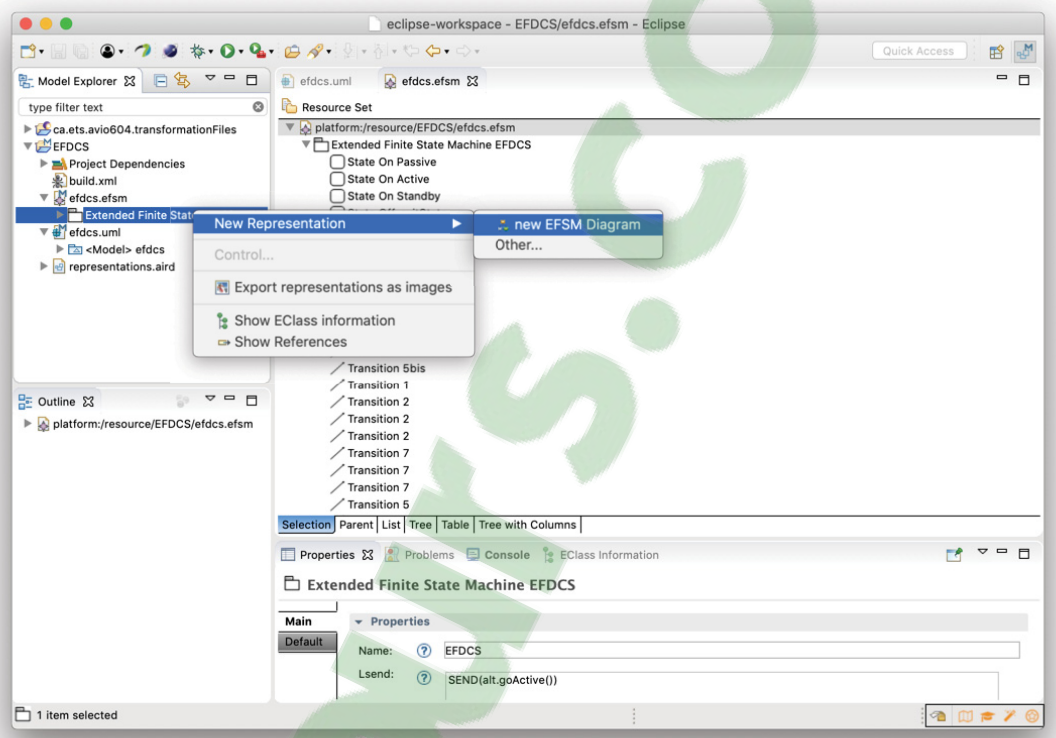
3. On obtient ainsi un fichier EFDCS.efsm correspondant au résultat de la transformation du modèle UML. Ce fichier se situe dans le même projet à la racine.



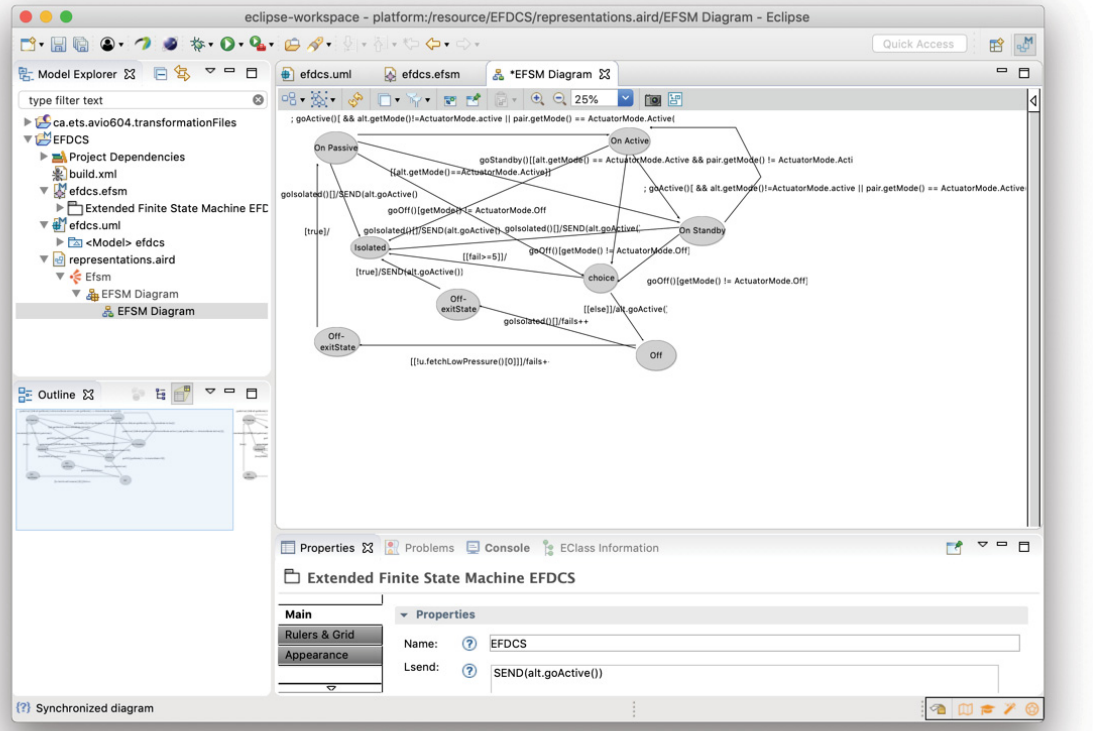
4. En ouvrant le fichier résultant on obtient une vue en arborescence de la représentation de la machine. Une icône est fournie pour chaque élément (état, transition, machine EFSM).



5. Il est également possible de créer une version graphique de la machine résultante. Pour cela, déroulez le fichier `EFDCS.efsm` dans l'onglet **Model Explorer** et faites un clic droit sur le premier élément (Extended Finite State Machine EFDCS). Choisissez l'option **New Representation** puis **new EFSM Diagram** et donnez un nom au diagramme.



6. Le diagramme est créé, et vous pouvez le réorganiser selon vos préférences.



BIBLIOGRAPHIE

Agrawal, A., Simon, G. & Karsai, G. (2004). Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. *Electronic Notes in Theoretical Computer Science*, 109, 43 - 56. Repéré à https://doi.org/10.1016/j.entcs.2004.02.055. Proceedings of the Workshop on Graph Transformation and Visual Modelling Techniques (GT-VMT 2004).

Alagar, V. & Periyasamy, K. (2011). Extended Finite State Machine. Dans *Specification of Software Systems* (éd. 2, pp. 105-128). Springer.

André, E., Benmoussa, M. & Choppy, C. (2014). Translating UML State Machines to Coloured Petri Nets Using Acceleo : A Report. *Electronic Proceedings in Theoretical Computer Science*, 150. doi : 10.4204/EPTCS.150.1.

Ansys. (2018). Scade Suite (Version 2019 R1) [Logiciel]. Ansys. Repéré à https://www.ansys.com/products/embedded-software/ansys-scade-suite.

Baouya, A., Bennouar, D., Ait Mohamed, O. & Ouchani, S. (2015, 04). A probabilistic and timed verification approach of SysML state machine diagram. pp. 1-9. doi : 10.1109/ISPS.2015.7245001.

Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P. & Šimeček, P. (2006). DiVinE - A Tool for Distributed Verification. *Computer Aided Verification*, pp. 278-281.

Bézivin, J. (2004). In search of a Basic Principle for Model-Driven Engineering. *Novatica – Special Issue on UML (Unified Modeling Language)*, 5(2), 21-24. Repéré à https://hal.archives-ouvertes.fr/hal-00442702.

Boehm, B. W. et al. (1981). *Software engineering economics*. Prentice-hall Englewood Cliffs (NJ).

Bourhfir, C., Aboulhamid, E., Dssouli, R. & Rico, N. (2001). A test case generation approach for conformance testing of SDL systems. *Computer Communications*, 24(3), 319 - 333. Repéré à https://doi.org/10.1016/S0140-3664(00)00220-6.

Coward, P. (1991). Symbolic execution and testing. *Information and Software Technology*, 33(1), 53 - 64. Repéré à https://doi.org/10.1016/0950-5849(91)90024-6.

Czarnecki, K. & Helsen, S. (2003). Classification of Model Transformation Approaches. *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 1-18.

Czarnecki, K. & Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45, 621-646.

Devroey, X., Cordy, M., Schobbens, P., Legay, A. & Heymans, P. (2015, April). State machine flattening, a mapping study and tools assessment. *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 1-8. doi : 10.1109/ICSTW.2015.7107408.

Di Guglielmo, G., Fujita, M., Fummi, F., Pravadelli, G. & Soffia, S. (2011, 07). EFSM-based model-driven approach to concolic testing of system-level design. *Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMPCODE2011)*, pp. 201-209. doi : 10.1109/MEMCOD.2011.5970527.

Duale, A. Y. & Uyar, M. U. (2004). A Method Enabling Feasible Conformance Test Sequence Generation for EFSM Models. *IEEE Trans. Comput.*, 53(5), 614–627. doi : 10.1109/TC.2004.1275300.

Eclipse. (2019a). Eclipse Modeling Framework (Version 2.13) [Logiciel]. The Eclipse Foundation. Repéré à https://www.eclipse.org/modeling/emf/.

Eclipse. (2019b). QVT Declarative (Version 0.19) [Logiciel]. The Eclipse Foundation. Repéré à https://projects.eclipse.org/projects/modeling.mmt.qvtd.

Eclipse. (2019c). QVT Operationnal (Version 3.9.2) [Logiciel]. The Eclipse Foundation. Repéré à https://wiki.eclipse.org/QVTo.

Eclipse. (2019d). Viatra2 (Version 2.2) [Logiciel]. The Eclipse Foundation. Repéré à https://www.eclipse.org/viatra/.

Ernits, J.-P., Kull, A., Raiend, K. & Vain, J. (2006). Generating Tests from EFSM Models Using Guided Model Checking and Iterated Search Refinement. *Formal Approaches to Software Testing and Runtime Verification*, pp. 85–99.

Feiler, P. H. (2014). *AADL and Model-based Engineering* (Rapport n°1). Pittsburgh, PA 15213 : Software Engineering Institute Carnegie Mellon University. Repéré à https://apps.dtic.mil/dtic/tr/fulltext/u2/a613948.pdf.

Felderer, M. & Herrmann, A. (2015). Manual Test Case Derivation from UML Activity Diagrams and State Machines : a Controlled Experiment. *Information and Software Technology*, 61. doi : 10.1016/j.infsof.2014.12.005.

Frankl, P. & Weyuker, E. (1988). Applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10), 1483–1498. doi : 10.1109/32.6194.

Geiß, R., Batz, G. V., Grund, D., Hack, S. & Szalkowski, A. (2006). GrGen : A Fast SPO-based Graph Rewriting Tool. *Proceedings of the Third International Conference on Graph Transformations*, (ICGT'06), 383–397. doi : 10.1007/11841883_27.

Gomes, C., Barroca, B. & Amaral, V. (2014). Classification of Model Transformation Tools : Pattern Matching Techniques. *Model-Driven Engineering Languages and Systems*, pp. 619–635.

Grønmo, R., Møller-Pedersen, B. & Olsen, G. K. (2009). Comparison of Three Model Transformation Languages. *Model Driven Architecture - Foundations and Applications*, pp. 2–17.

Guglielmo, G. D., Fujita, M., Fummi, F., Pravadelli, G. & Soffia, S. (2011, July). EFSM-based model-driven approach to concolic testing of system-level design. *Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMP-CODE2011)*, pp. 201-209. doi : 10.1109/MEMCOD.2011.5970527.

Harel, D. (1987). Statecharts : a visual formalism for complex systems. *Science of Computer Programming*, 8(3), 231 - 174. Repéré à https://doi.org/10.1016/0167-6423(87)90035-9.

Hong, H. S., Kim, Y. G., Cha, S. D., Bae, D. H. & Ural, H. (2000). A test sequence selection method for statecharts. *Software Testing, Verification and Reliability*, 10(4), 203-227. doi : 10.1002/1099-1689(200012)10:4<203::AID-STVR212>3.0.CO;2-2.

Huhn, M. & Hungar, H. (2007, 01). UML for Software Safety and Certification - Model-Based Development of Safety-Critical Software-Intensive Systems. pp. 201-237.

ISO. (2017). *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE).* (Rapport n°ISO/IEC 25010 :2011).

Jouault, F., Allilaire, F., Bézivin, J. & Kurtev, I. (2008). ATL : A model transformation tool. *Science of Computer Programming*, 72(1), 31 - 39. Repéré à https://doi.org/10.1016/j.scico.2007.08.002. Special Issue on Second issue of experimental software and toolkits (EST).

Kalaji, A. S., Hierons, R. M. & Swift, S. (2011). An integrated search-based approach for automatic testing from extended finite state machine (EFSM) models. *Information and Software Technology*, 53(12), 1297 - 1318. Repéré à https://doi.org/10.1016/j.infsof.2011.06.004.

Kim, Y. G., Hong, H. S., Bae, D. H. & Cha, S. D. (1999). Test cases generation from UML state diagrams. *IEE Proceedings - Software*, 146(4), 187-192. doi : 10.1049/ip-sen:19990602.

Kolahdouz-Rahimi, S., Lano, K., Pillay, S., Troya, J. & Gorp, P. V. (2014). Evaluation of model transformation approaches for model refactoring. *Science of Computer Programming*, 85, 5 - 40. Repéré à https://doi.org/10.1016/j.scico.2013.07.013. Special issue on Experimental Software Engineering in the Cloud(ESEiC).

Kratochvlová, P. (2015). *Modelling Stateflow Diagrams for Verification Purposes*. (Bachelor's thesis, Masarykova Univerzita Fakulta Informatiky).

Kumar, V., Singh, L. K. & Tripathi, A. K. (2017). Transformation of deterministic models into state space models for safety analysis of safety critical systems : A case study of NPP. *Annals of Nuclear Energy*, 105, 133 - 143. Repéré à https://doi.org/10.1016/j.anucene.2017.02.026.

Kuske, S. (2001). A Formal Semantics of UML State Machines Based on Structured Graph Transformation. *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, («UML» '01), 241-256. Repéré à http://dl.acm.org/citation.cfm?id=647245.719457.

Kuzniarz, L. & Staron, M. (2007). Two Techniques for UML Model Transformations. *International Journal of Computers and Applications*, 29(1), 10-17. doi : 10.1080/1206212X.2007.11441827.

Lano, K., Rahimi, S. K. & Clark, T. (2014). Language-independent model transformation verification. Dans Amrani, M., Syriani, E. & Wimmer, M. (Éds.), *VOLT 2014 : verification of model transformations : proceedings of the Third International Workshop on Verification of Model Transformations co-located with Software Technologies : Applications and Foundations (STAF 2014), York, Uk, July 21, 2014* (pp. 36–45). Tilburg University. Repéré à http://shura.shu.ac.uk/12068/.

Li, J. J. & Wong, W. E. (2002, April). Automatic test generation from communicating extended finite state machine (CEFSM)-based models. *Proceedings Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISIRC 2002*, pp. 181-185. doi : 10.1109/ISORC.2002.1003693.

Li, M. & Kumar, R. (2011, 07). Stateflow to Extended Finite Automata Translation. *Proceedings - International Computer Software and Applications Conference*, pp. 1-6. doi : 10.1109/COMPSACW.2011.11.

Marques, J., Yelisetty, S., Dias, L. & Cunha, A. (2012, 04). Using Model-Based Development as Software Low-Level Requirements to Achieve Airborne Software Certification. pp. 431-436. doi : 10.1109/ITNG.2012.102.

Mathworks. (2019). Simulink : Simulation and Model-Based Design (Version R2019a) [Logiciel]. Repéré à https://www.mathworks.com/products/simulink.html.

Mathworks. (2019). *Stateflow : User's Guide* (Rapport n°R2019a). Repéré à https://www.mathworks.com/help/pdf_doc/stateflow/sf_ug.pdf.

Mens, T. & Van Gorp, P. (2006). A Taxonomy of Model Transformation. *Electron. Notes Theor. Comput. Sci.*, 152, 125–142. doi : 10.1016/j.entcs.2005.10.021.

Minas, M. & Hoffmann, B. (2008). An Example of Cloning Graph Transformation Rules for Programming. *Electronic Notes in Theoretical Computer Science*, 211, 241 - 250. Repéré à https://doi.org/10.1016/j.entcs.2008.04.046. Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006).

Mosterman, P. & Ghidella, J. (2018). Fault Detection Control Logic in an Aircraft Elevator Control System. Accessed : 2018-11-13.

Naik, K. & Tripathy, P. (2008). Software Testing and Quality Assurance : Theory and Practice. 601-616. doi : 10.1002/9780470382844.index.

Niaz, I. A. & Tanaka, J. (2004). Mapping UML statecharts to java code. *IASTED Conf. on Software Engineering*.

OMG. (2000). *Model Driven Architecture*. Object Management Group (OMG). Repéré à https://www.omg.org/cgi-bin/doc?omg/00-11-05.

OMG. (2014). *Object Constraint Language 2.4*. Object Management Group (OMG). Repéré à https://www.omg.org/spec/OCL/2.4.

OMG. (2016). *Meta Object Facility 2.5*. Object Management Group (OMG). Repéré à https://www.omg.org/spec/MOF/2.5.1.

OMG. (2017). *Unified Modeling Language 2.5*. Object Management Group (OMG). Repéré à http://www.omg.org/spec/UML/2.5.1.

Paz, A. & Boussaidi, G. E. (2018). Building a Software Requirements Specification and Design for an Avionics System : An Experience Report. *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, (SAC '18), 1262–1271. doi : 10.1145/3167132.3167268.

Paz, A. & Boussaidi, G. E. (2019, 07). Supporting Consistency in the Heterogeneous Design of Safety-Critical Software (forthcoming). *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*.

Pradhan, S., Ray, M. & Swain, S. K. (2019). Transition coverage based test case generation from state chart diagram. *Journal of King Saud University - Computer and Information Sciences*. Repéré à https://doi.org/10.1016/j.jksuci.2019.05.005.

RTCA. (2011a). *DO-178C, Software Considerations in Airborne Systems and Equipment Certification* (Rapport n°DO-178C).

RTCA. (2011b). *DO-331, Model Based Development and Verification Supplement to DO-178C and DO-278A*. (Rapport n°DO-331).

Scaife, N., Sofronis, C., Caspi, P., Tripakis, S. & Maraninchi, F. (2004, 01). Defining and translating a safe subset of Simulink/Stateflow into Lustre. *EMSOFT 2004 - Fourth ACM International Conference on Embedded Software*, pp. 259-268. doi : 10.1145/1017753.1017795.

Schmidt, D. C. (2006). Guest Editor's Introduction : Model-Driven Engineering. *Computer*, 39(2), 25-31. doi : 10.1109/MC.2006.58.

Sendall, S. & Kozaczynski, W. (2003). Model transformation : the heart and soul of model-driven software development. *IEEE Software*, 20(5), 42-45. doi : 10.1109/MS.2003.1231150.

Shafique, M. & Labiche, Y. (2015). A systematic review of state-based test tools. *International Journal on Software Tools for Technology Transfer*, 17(1), 59–76. doi : 10.1007/s10009-013-0291-0.

Stahl, T., Voelter, M. & Czarnecki, K. (2006). *Model-Driven Software Development : Technology, Engineering, Management*. USA : John Wiley &#38 ; Sons, Inc.

van Amstel, M., Bosems, S., Kurtev, I. & Ferreira Pires, L. (2011). Performance in Model Transformations : Experiments with ATL and QVT. *Theory and Practice of Model Transformations*, pp. 198–212.

Visser, E. (2001). A Survey of Rewriting Strategies in Program Transformation Systems. *Electronic Notes in Theoretical Computer Science*, 57, 109 - 143. Repéré à https://doi.org/10.1016/S1571-0661(04)00270-1. WRS 2001, 1st International Workshop on Reduction Strategies in Rewriting and Programming.

Yao, S. & Shatz, S. (2006, 12). Consistency checking of UML dynamic models based on Petri Net techniques. *Proceedings - 15th International Conference on Computing, CIC 2006*, pp. 289–297. doi : 10.1109/CIC.2006.32.

Zou, L., Zhan, N., Wang, S. & Fränzle, M. (2015, 10). Formal Verification of Simulink/Stateflow Diagrams. *Automated Technology for Verification and Analysis : 13th International Symposium*, 9364, 464-481. doi : 10.1007/978-3-319-24953-7_33.