Contributions à la modélisation et à la validation des systèmes complexes

Ahmed Hammad

École doctorale SPIM : Sciences Pour l'Ingénieur et Microtechniques

HABILITATION À DIRIGER DES RECHERCHES
DE L'ÉTABLISSEMENT UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ
PRÉPARÉE À L'UNIVERSITÉ DE FRANCHE-COMTÉ

Spécialité : Informatique

Présentée par
Ahmed HAMMAD

Contributions à la modélisation et à la validation des systèmes complexes

Soutenue à Besançon le 21 septembre 2018 devant le jury composé de :

Rapporteurs

Yamine AIT-AMEUR	Professeur INPT-ENSEEIHT, Toulouse.
Christian ATTIOGBÉ	Professeur Université de Nantes.
Khalil DRIRA	Directeur de recherche LAAS-CNRS, Toulouse.

Examinateurs

Fabrice BOUQUET	Professeur Université de Bourgogne Franche-Comté.
Pierre-Cyrille HÉAM	Professeur Université de Bourgogne Franche-Comté.
Hassan MOUNTASSIR	Professeur Université de Bourgogne Franche-Comté.

Remerciements

Je remercie vivement Yamine Ait-Ameur, Christian Attiogbé et Khalil Drira d'avoir accepté d'être rapporteurs de ce mémoire. La lecture en profondeur qu'ils ont réalisée est remarquable et mérite bien plus que mes remerciements. Leurs remarques et leurs suggestions ont ainsi pu faire progresser significativement ce document.

Mes travaux sont les fruits de collaborations avec de nombreux chercheurs, que je tiens à les remercier collectivement ici. Je remercie particulièrement Hassan Mountassir qui a favorisé la préparation de cette habilitation en acceptant de m'associer à ses travaux par des co-directions de thèses. Depuis mon arrivée au LIFC, il m'a apporté son soutien constant pour développer de nouveaux projets de recherche.

Fabrice Bouquet m'a soutenu dans une période cruciale de ma carrière en me permettant de co-encadrer la thèse de J-M. Gauthier. Je le remercie vivement pour la confiance qu'il m'a accordée, pour sa rigueur et sa constance tout au long de notre collaboration.

Ma reconnaissance va aussi à Pierre-Cyrille Héam qui m'a beaucoup aidé ces dernières années. Il m'a accompagné dans la préparation de cette habilitation ; les précieux conseils qu'il m'a prodigués et ses nombreuses lectures de ce manuscrit ont largement contribués à l'amélioration de ce document. Je le remercie infiniment.

Je remercie chaleureusement Malika Ioualalen de l'USTHB d'Alger pour sa précieuse collaboration et son soutien indéfectible. Elle m'a permis d'encadrer plusieurs doctorants avec elle.

J'adresse mes remerciements à tous les membres du DISC (Département d'Informatique des Systèmes complexes), aux collègues du MN2S et de AS2M de l'institut FEMTO-ST avec qui j'ai collaboré dans des projets de recherche pluri-disciplinaires.

Je tiens à remercier les collègues avec qui j'ai travaillé depuis mon arrivée au LIFC : Bruno Tatibouët pour la partie "Utilisation conjointe de B et UML", Jacques Julliand sur la thématique du "Raffinement d'automates", Hervé Guyennet pour "les réseaux de capteurs" et Samir Chouali pour les "systèmes à base de composants".

Tous les co-auteurs cités dans ce manuscrit méritent également ma reconnaissance : Fabrice Bouquet, Samir Chouali, Bruno Tatibouët, Jacques Julliand, Fabien Peureux, Hassan Mountassir, Alain Giorgetti, Olga Kouchnarencko, Malika Ioualalen, Messaoud Rahim, Jean Marie Gauthier, Mohammed Al Alchhab, Isabelle Jacques, Jean Michel Hufflen, Abbas AbdulHameed, Hervé Guyennet, Régine Laleau, Hamida Bouaziz. Toutes mes excuses à celles et ceux que je n'ai pas cités.

Je remercie également mes collègues du département d'enseignement LEA (Langues Etrangères Appliquées) de l'UFR SLHS (Sciences du Langage, de l'Homme et de la société) pour avoir allégé mes responsabilités collectives lors de la rédaction de ce manuscrit.

J'exprime ma gratitude aux nombreuses personnes qui ont jalonné ces 25 années de recherche, les remercier pour les échanges scientifiques que nous avons eus et qui m'ont permis de progresser dans les différentes thématiques de recherche mais aussi pour leurs encouragements, leurs sourires et leurs critiques constructives. Cela concerne non seulement mes collègues mais également les doctorants, stagiaires et personnels techniques et administratifs du laboratoire.

Enfin, rien n'aurait été possible sans le soutien de mon épouse Nalia et de mes filles Kahina, Soraya et Nadia. Je voudrais ici leur témoigner toute ma reconnaissance et mon amour.

A la mémoire de mes parents.

Sommaire

Table des figures

Liste des tableaux

Glossaire

AcTRL	Activity Temporal Requirement Language
AH	Automate Hiérarchique
AI	Automate d'interface
BDD	Blocks Definition Diagrams
CTL	Computation Tree Logic
DAB	Distributeur Automatique de Billets
DET	Diagramme d'Etats-Transitions
IBD	Internal Blocks Diagrams
IDM	Ingénierie Dirigée par les Modéles
INCOSE	International Council on Systems Engineering
LTL	Linear Temporal Logic
MARTE	Modeling and Analysis of Real Time and Embedded systems
MEMS	MicroElectroMechanical Systems
OCL	Object Constraints Language
OMG	Object Management Group
PNML	Petri Net Markup Language
PSL	Property Specification Language
RdPHC	Réseaux de Petri Hiérarchique et Coloré
UML	Unified Modeling Language
SK	Structure de Kripke
SysML	Systems Modeling Language
VHDL-AMS	Very High Speed Integrated Circuit Hardware Description Language-Analog and Mixed Signal

Première partie

Introduction

1

Introduction

Plan du chapitre

1.1 Problématique, contexte et motivations

Avec l'évolution de la technologie, le développement informatique des systèmes au carrefour de différentes disciplines (aéronautique, automobile...) devient de plus en plus complexe. Ces derniers requièrent plusieurs approches pour la conception ou pour le développement, ce qui nécessite une collaboration entre les développeurs et les concepteurs. En raison de l'augmentation continue de la complexité de ces systèmes, il est nécessaire de mettre en place un environnement qui permet de les modéliser en éliminant les détails de bas niveau et en se focalisant sur l'aspect général, structurel et comportemental. L'identification des besoins, la construction d'architectures organiques et fonctionnelles, la spécification ou la vérification d'un comportement, le partage d'informations entre les parties prenantes nécessitent l'emploi de modèles. Ceux-ci permettent en effet d'appréhender et, dans une certaine mesure, de maîtriser la complexité des systèmes à réaliser. De l'identification des problèmes à la construction de solutions, les modèles employés sont généralement d'un niveau d'abstraction décroissant.

Mes travaux de recherche concernent les domaines de modélisation et de validation des systèmes complexes. Nous proposons des démarches complémentaires pour la spécification et la vérification de ces systèmes. Ces démarches reposent sur l'utilisation conjointe des langages de modélisation semi-formels, des méthodes formelles de vérification et des environnements de simulation. L'approche abordée est orientée modèle. Au lieu de construire le système réel directement, on conçoit d'abord un modèle, qui est analysé et validé. Un modèle doit être une représentation la plus proche possible du système réel et en même temps ne pas être trop complexe afin de ne pas être difficile à comprendre et/ou à analyser. Plus précisément, la modélisation est le processus permettant de produire un modèle. Le but principal d'un modèle est de permettre à l'analyste de prévoir l'effet de changements du système [1]. La modélisation permet de décrire les exigences du système à concevoir, sa structure et ses comportements. Elle est aussi utile pour vérifier si les comportements des modèles du système sont conformes à leurs exigences.

Plusieurs langages et formalismes peuvent être utilisés pour modéliser des systèmes. Ces langages peuvent être classés en trois catégories : les langages naturels, semi-formels et formels. Les langages naturels sont faciles à utiliser par les différentes parties prenantes impliquées dans le développement du système. Cependant, le langage naturel reste souvent ambigu, non précis et très souvent différentes interprétations peuvent être déduites à partir d'une spécification. Il n'est donc pas aisé de spécifier dans un langage naturel un système complexe et un ensemble complet et cohérent d'exigences. Les langages semi-formels sont des langages qui reposent généralement sur des notations graphiques. Ces langages sont très utiles pour décrire des systèmes complexes, car ils sont graphiques, donc faciles à apprendre et à utiliser. Cependant, l'absence d'une sémantique précise peut parfois donner lieu à une certaine ambiguïté lorsqu'il s'agit d'interpréter un modèle construit avec ces langages. L'utilisation et la mise en oeuvre des spécifications formelles permettent de détecter les erreurs assez tôt durant le processus de développement, automatiser l'étape de vérification et aider dans la compréhension du système. Malgré tous ces avantages, les langages formels restent peu utilisés dans l'industrie, car leur utilisation nécessite des compétences théoriques, ainsi que la maîtrise des fondements mathématiques et d'une variété de notations spécifiques.

Dans ce qui suit, nous exprimons une spécification dans les trois types de langage définis précédemment, nous considérons par exemple le cas d'une machine à café à jetons. Exemple :

1. en langage naturel : Si on insère un jeton dans la machine, elle nous délivrera un café,
2. en langage semi-formel, nous aurons le diagramme décrit par la figure 1.1, où PlacerGobelet et RécupérerCafé sont respectivement des actions d'entrée et de sortie de l'état EnService.
3. En langage formel, Nous définissons le modèle de données (variables d'état) suivant :
   - $Jeton\_inséré \in \{vrai, faux\}$
   - $Receptacle\_café \in \{vide, plein\}$
   - En LTL [1] : $\Box(Jeton\_inséré = vrai) \Rightarrow \Diamond(Receptacle\_café = plein)$

Un de nos objectifs est de proposer des mécanismes pour tirer profit à la fois de la facilité d'utilisation des langages semi-formels et en même temps de la précision des langages formels ainsi que leurs avantages pour les phases de validation et de vérification, comme représenté sur la figure 1.2. Dans ce manuscrit, je décris mes travaux, qui peuvent être structurés en trois parties distinctes mais complémentaires.

1. logique temporelle linéaire

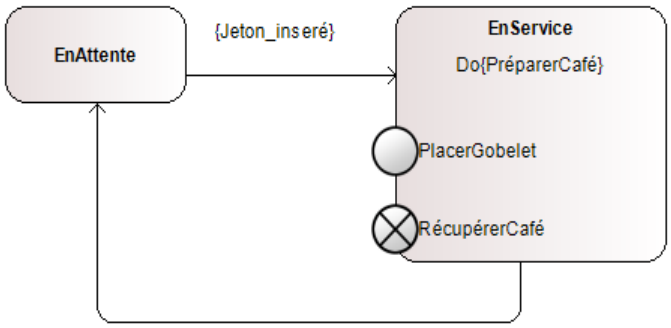


FIGURE 1.1 – Diagramme d'états-transitions de la machine à café

La première partie de mes travaux consiste à extraire des vues semi-formelles à partir des vues formelles afin de valider des spécifications formelles au regard du cahier des charges. Ce travail aide à la compréhension du modèle formel par les non-spécialistes et contribue donc à la validation du cahier des charges.
La seconde partie propose une démarche incrémentale de spécification par raffinement de systèmes en préservant les propriétés des niveaux antérieurs. L'objectif principal de cette partie est de proposer une démarche pour faire face au problème de l'explosion combinatoire du nombre d'états lié à la taille et à la complexité des systèmes.
La troisième partie, qui constitue ma contribution majeure, concerne la proposition d'une démarche de spécification et de validation des systèmes complexes. L'idée est de proposer une telle démarche qui permet à l'utilisateur de produire facilement des spécifications (utilisation du langage semi-formel SysML [2, 3]) tout en pensant, d'une part à la vérification (possibilité de passage automatique à des spécifications formelles, automatisation de la vérification et son applicabilité sur les spécifications formelles dérivées) et d'autre part à la validation par simulation (transformation de modèles semi-formels vers des environnements de simulation).

Dans ce qui suit, je ferai souvent référence aux transformations de modèles utilisant les techniques de l'ingénierie dirigée par les modèles. La modélisation et la validation du système par rapport aux exigences du système constituent le coeur de ces travaux. Dans ce qui suit, avant d'introduire mes contributions, je décrirai brièvement les notions de systèmes complexes.

1.2 Systèmes complexes

Un système complexe est un système représentant un grand nombre de composants interagissant (des agents, des processus, etc) dont l'activité globale est non-linéaire (non dérivable des additions de l'activité de composants individuels). Les systèmes que nous prenons en considération dans ces travaux sont les systèmes intégrant fortement le logiciel avec d'autres composants. Nous pouvons citer les exemples suivants utilisés dans nos projets de recherche :

— Un système micro-robotique distribué pour le convoyage, le positionnement et le tri de micro-pièces à l'échelle mésoscopique, utilisé dans le cadre du projet Smart Surface (ANR 06 ROBO 0009) du Programme Systèmes Interactifs et Robotique PSIROB 2006 [2]

— Un système de tri et de convoyage de micro pièces fragiles au moyen d'un système distribué reconfigurable dynamiquement constitué de MEMS, utilisé dans la projet Smart Block (projet ANR-11-BS03-0005) [3] qui est la continuité du projet Smart Surface.

2. `http://smartsurface.free.fr/`
3. `http://smartblocks.univ-fcomte.fr/`

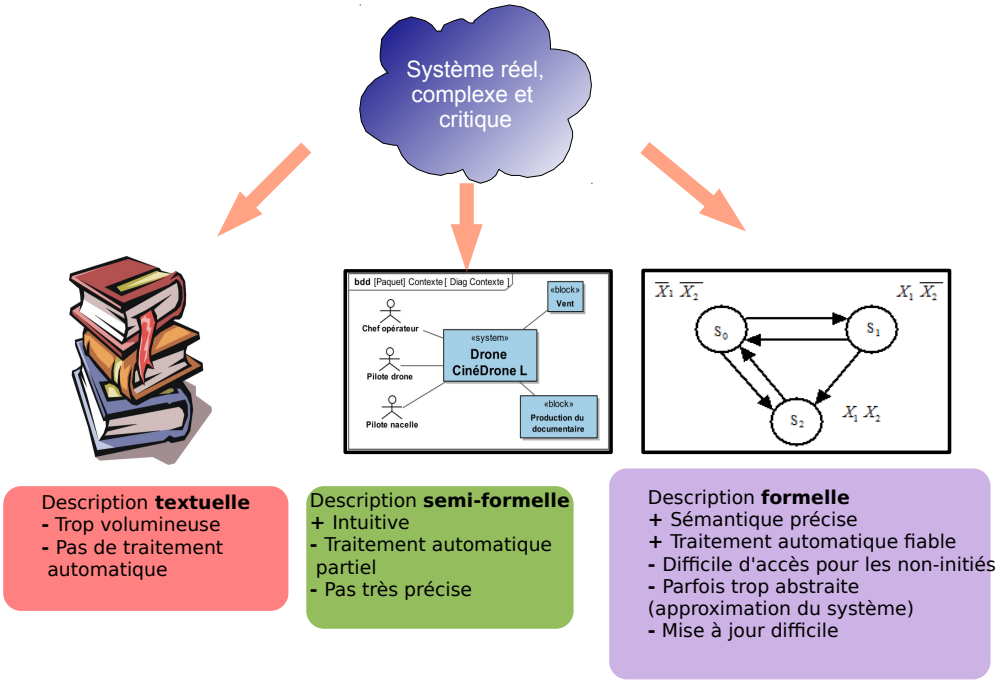


FIGURE 1.2 – Modèles des systèmes réels

— Étude d'un écoulement autour du corps d'Ahmed [4] dans le projet région SyVAD[4]. Nous nous sommes intéressés notamment aux générateurs de micro-jets distribués.

1.3 Contributions

1.3.1 Combinaison des langages formels et semi-formels

Dans le domaine industriel, les langages de modélisation les plus utilisés sont les langages dits semi-formels de type UML [5, 6], SysML ou MARTE [7, 8]. Ils présentent un avantage majeur qui est leur facilité d'utilisation. Les modèles sont décrits sous forme de diagrammes en utilisant des syntaxes graphiques facilement comprises par les utilisateurs et les concepteurs de systèmes. Cet atout a largement répandu leur adoption comme langages de modélisation dans le milieu industriel. Leur principal inconvénient est le manque d'une sémantique précise qui les rend ambigus. Cependant, la complexité grandissante des logiciels est de plus en plus évidente et la maîtrise des risques inhérents à leur utilisation devient impérative et appelle, de ce fait, une rigueur et une précision accrues lors de leur élaboration. Aussi, l'adoption de langages de modélisation, reposants sur des fondements mathématiques, s'impose-t-elle, désormais, en tant que garant de la sûreté. Ces langages, dits formels, ont été élaborés afin d'assurer un niveau aussi élevé que possible en matière de précision et de cohérence. Leur avantage majeur réside, en somme, dans le fait qu'ils sont basés sur les mathématiques, ce qui permet d'une part de pallier aux risques d'ambiguïté et d'incertitude, et d'autre part, de parvenir à un produit fini qui répond aux spécifications requises. Cependant, ces méthodes utilisent des notations et des concepts spécifiques qui génèrent souvent une faible lisibilité et une difficulté d'intégration dans les processus de développement et de certification. Notre objectif est de tirer profit des avantages de ces deux catégories de langages. Nous proposons une utilisation conjointe de ces deux concepts en vue de valider le cahier des charges à la fin de l'étape spécification/conception. En effet, avant de passer à l'étape déploiement, il est nécessaire de vérifier la conformité des spécifications définies par les futurs utilisateurs avec les comportements définis par les concepteurs du système. Notre démarche consiste donc à extraire des vues semi-formelles à partir des modèles formels. Dans ce travail, nous

4. http://syvad.univ-fcomte.fr/

nous sommes intéressés à l'extraction de diagrammes UML à partir de spécifications écrites en langage *B* [9, 10]. Ce travail aide à la compréhension du modèle formel par les non-spécialistes et contribue donc à la validation du cahier des charges. Ce travail a été mené en étroite collaboration avec Bruno Tatibouet et a donné lieu à une soutenance de thèse (Jean-Christophe Voisinet) [11] et aux communications suivantes : [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22].

1.3.2 Abstraction et raffinement de modèles

L'écriture d'une spécification devient une tâche ardue quant il s'agit de spécifier des systèmes de grande taille, complexes et hétérogènes. L'abstraction est un des fondements de la modélisation des systèmes. Elle permet d'identifier les caractéristiques les plus importantes du système en vue d'une utilisation précise. Suivant l'étape de modélisation du système, certaines caractéristiques sont moins pertinentes à ce moment là, donc peuvent être simplifiées pour se consacrer aux autres qui sont plus importantes à ce stade de la modélisation. L'abstraction consiste donc à ne pas prendre en compte certaines fonctionnalités du système au cours de la phase de spécification. L'objectif principal de cette partie est de proposer une démarche pour faire face au problème de l'explosion combinatoire du nombre d'états lié à la taille et à la complexité des systèmes. Nous proposons d'utiliser la démarche de spécification par raffinement comme moyen. Le raffinement est un processus de conception qui décrit la manière de passer d'une spécification abstraite à une spécification plus détaillée en remplaçant dans la spécification abstraite les éléments de haut niveau par des constructions plus proches du système réel. Il est aussi considéré comme un processus de vérification. En effet, il est nécessaire de s'assurer que les transformations apportées par le raffinement préservent la correction des nouvelles spécifications vis-à-vis de la spécification initiale. Dans la première partie de ce travail, nous avons choisi d'utiliser les automates hiérarchiques comme modèle intermédiaire. Ces automates ont été proposés pour la première fois dans [23]. Cette représentation garantit la préservation de certaines notions comme la hiérarchie du modèle, la priorité entre les transitions et les transitions inter-niveaux. Dans le cadre de ce travail, M. Al Achhab a soutenu sa thèse en 2006 [24]. Nous avons publié les articles et communications suivants : [25, 26, 27, 28, 29, 30, 31, 32].
Dans la seconde partie, nous nous sommes intéressés à la modélisation et à la vérification par *model-checking* des systèmes temps réel. Ces systèmes sont modélisés par les automates temporisés proposés dans [33]. Nous utilisons le concept de raffinement pour pallier au problème combinatoire du nombre d'états dû principalement à la continuité de la variable temps. Les articles suivants ont été publiés dans le cadre de ce travail : [34, 35, 36, 37, 38, 39].

1.3.3 Modélisation et validation des systèmes complexes

Afin d'assurer la correction des systèmes complexes et critiques, dès les premières phases de leur développement, il est essentiel de définir un processus de développement qui inclut les phases d'analyse, de spécification, de conception, de validation et de vérification. Ces phases doivent être étroitement liées et doivent se combiner de façon itérative. Nous visons à tirer profit des concepts émergents portés par l'ingénierie des systèmes basée sur les modèles semi-formels, combinés à des méthodes de validation et de vérification formelles, pour aider les concepteurs à développer des systèmes qui répondent pleinement aux exigences qui y sont relatives. Nous proposons une méthodologie basée sur SysML s'appuyant sur une description précise des exigences et l'exploitation des relations entre exigences et éléments du modèle SysML. Le langage SysML est bien adapté aux systèmes complexes hétérogènes, en plus de la modélisation hiérarchique des blocs matériels et logiciels, il permet de modéliser graphiquement les équations mathématiques définissant le comportement physique de ceux-ci. Ce langage a été adopté récemment par l'OMG [40] comme un standard dans l'ingénierie système. Néanmoins, malgré les différents avantages de SysML, il reste un langage semi-formel sans possibilité d'exécuter directement ou de simuler les modèles décrits, donc difficile à valider. Cette démarche intégrera la traçabilité des exigences et les techniques du Model Driven Engineering (MDE) pour la transformation des modèles SysML vers des environnements de simulation (modelica [41], VHDL-AMS [42], SystemC [43],...) et de vérification (Uppaal [44], SPIN [45], TINA [46],...).
Les exigences sur les systèmes complexes peuvent être fonctionnelles ou non. Les exigences non fonctionnelles peuvent être validées par simulation en transformant les diagrammes SysML, notamment le dia-

gramme paramétrique, vers des environnements de simulation. Les exigences fonctionnelles seront validées par vérification formelle. Nous proposons un langage temporel proche de l'environnement SysML pour préciser les exigences fonctionnelles, exprimables sur les diagrammes SysML et transformer celles-ci ainsi que les modèles SysML vers des environnements de vérification.

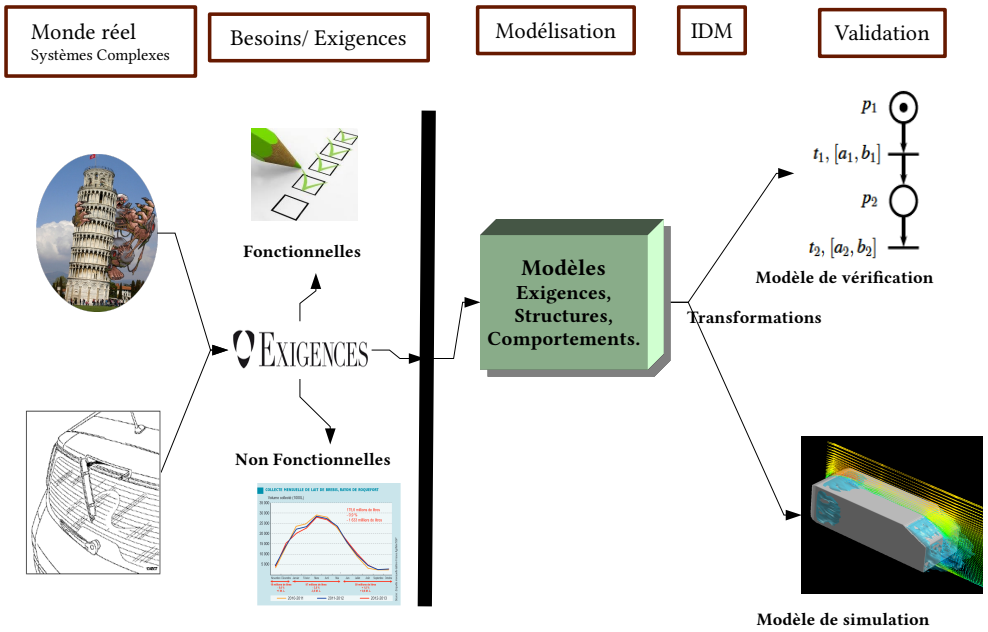


FIGURE 1.3 – Démarche de modélisation et de validation

Notre objectif est donc de proposer une démarche (voir Figure 1.3) basée sur le langage SysML et d'exploiter par la suite les techniques et méthodes d'ingénierie des modèles pour transformer les modèles SysML vers des environnements de simulation et de vérification. Ces transformations sont nécessaires pour évaluer la conformité entre les besoins et la conception. Nous avons réalisé les travaux suivants :

— Dans le cadre du projet SyVAD, nous avons modélisé l'écoulement de l'air autour du corps d'Ahmed [4] avec SysML, exprimé des exigences avec le langage PSL (Property Specification Language) et généré une spécification VHDL-AMS en vue de la simulation, [47, 48].

— Dans la cadre de la thèse de J.M. Gauthier et du projet Smart Blocks, nous avons collaboré avec l'OMG pour la validation des règles de transformation entre SysML et Modelica. Nous avons notamment modélisé avec SysML et simulé avec l'environnement Modelica un convoyeur à jets d'air [49, 50, 51, 52, 53, 54, 55]. La description de ces travaux n'est pas incluse dans ce manuscrit.

— Les travaux de thèse de A. Abbas nous ont permis de proposer des transformations des modèles SysML vers SystemC en vue de valider les exigences non fonctionnelles et de générer du code Promela/Spin en vue de vérifier des exigences fonctionnelles exprimées en LTL [56, 57, 58, 59].

— Dans le cadre du partenariat Hubert Curien (PHC) Tassili (projet SysVEP) et des travaux de thèse de M. Rahim, nous avons proposé un langage temporel pour la spécification des exigences fonctionnelles et proposé des règles de transformation du diagramme d'activités SysML vers les réseaux de Petri (RdP) en vue de les vérifier avec l'outil CPNTools [60, 61, 62, 63, 64, 65, 66].

— Les travaux de H. Bouaziz, concernent la correction de la compatibilité des blocs SysML pris comme étant des composants. Nous avons notamment généré des automates d'interfaces à partir des diagrammes de séquences en vue de vérifier l'assemblage des composants blocs-SysML [67, 68, 69, 70, 71, 72, 73, 74].

1.4 Chronologie des travaux

Mes travaux de recherche sont décrits chronologiquement comme suit :

De 1993 à 1998 : lors de mon recrutement, j'ai intégré le laboratoire de Mathématiques, Informatique et Statistique (M.I.S) de l'UFR Sciences du Langage, de l'Homme et de la Société (SLHS), UFR de mon affectation. J'ai notamment travaillé sur les problématiques liées aux thématiques de recherche de cette UFR, notamment l'Analyse de Données (A.D) et l'apprentissage des langues. Dans la première partie, j'ai contribué à la conception et à la réalisation d'un outil de dépouillement d'enquêtes et de traitement statistique de données e-pragma [75, 76]. Ce logiciel offre à l'utilisateur, la possibilité de saisie en ligne de questionnaires, de fonctions statistiques de base et des fonctions d'analyse de données plus élaborées comme l'analyse factorielle. Dans la deuxième partie, je me suis particulièrement intéressé à l'apprentissage du français dans le contexte d'une langue seconde. J'ai notamment contribué à la conception d'un outil intégrant un test adaptatif informatisé pour l'apprentissage du français langue étrangère dans le cadre du projet TAFIC [77]. La description de ces travaux n'est pas incluse dans ce manuscrit.

De 1999 à 2002 : j'ai intégré le LIFC [5] et je me suis intéressé aux questions sur la combinaison des langages formels et semi-formels en vue de la spécification et de la vérification de systèmes. Je me suis intéressé plus particulièrement à l'extraction des spécifications UML à partir des modèles B.

De 2002 à 2009 : j'ai orienté mes travaux vers le raffinement, tout en gardant comme sujet les descriptions formelles proches de celles des descriptions semi-formelles, afin d'avoir une cohérence dans la démarche globale. Ces travaux concernent plus particulièrement l'apport du raffinement pour la spécification et la vérification des systèmes.

Depuis 2009 : mes recherches restent dans le cadre des problématiques précédentes, mais ont évoluées pour tenir compte de nouvelles formes d'applications issues de problématiques ayant émergées avec le rapprochement, puis l'intégration, du LIFC dans l'institut FEMTO-ST [6]. Ces travaux portent particulièrement sur l'utilisation du langage SysML dans le cadre de la modélisation et de la validation des systèmes complexes.

5. Laboratoire d'Informatique de l'université de Franche-Comté.

6. L'institut FEMTO-ST (Franche-Comté Electronique Mécanique Thermique et Optique - Sciences et Technologies, UMR 6174), est une fusion de sept laboratoires francs-comtois. Son champ thématique couvre en effet l'optique, l'acoustique, les micro nanosciences et systèmes, le temps-fréquence, l'automatique, l'informatique, la mécatronique, en même temps que la mécanique et les matériaux, l'énergétique et le génie électrique

Deuxième partie

IDM et la vérification des systèmes

2

Extraction de diagrammes UML à partir d'une spécification B

Plan du chapitre

2.1 Introduction

La modélisation est une représentation d'un problème dans un formalisme afin de le rendre compréhensible. Le modèle produit ne devrait pas être si complexe qu'il est impossible de le comprendre et de l'expérimenter. Les modèles peuvent décrire le comportement et/ou la structure du système conçu, ils peuvent être utilisés pour valider les caractéristiques de certaines parties ou de l'ensemble du système conçu, par exemple, sa fonctionnalité ou ses performances [1]. Plusieurs méthodes de spécifications ont été élaborées. Parmi ces méthodes, on distingue deux grandes familles : la première dite formelle, se basant sur des notations issues des mathématiques, la deuxième dite semi-formelle se basant principalement sur des notations graphiques.

Les méthodes formelles sont aujourd'hui la manière la plus rigoureuse de produire des logiciels. Elles fournissent des techniques pour assurer la cohérence d'une spécification et pour garantir que quelques parties de code implémentent bien une spécification donnée. Plusieurs entreprises industrielles où la sécurité est critique, comme l'industrie ferroviaire, ont perçu les avantages de telles approches. Des développements importants comme le métro Paris Meteor [78] ont été partiellement réalisés en utilisant des méthodes formelles [79]. Des sociétés comme Siemens Transport [80], Clearsy [81] et Gemalto[82] ont utilisé B [9, 10, 83] comme la méthode centrale pour les développements de code ou de modèle.
La méthode B a du succès dans le domaine ferroviaire notamment les systèmes de contrôle embarqués dans les équipements roulants ou plus récemment dans le système de contrôle de portes palières [84] La méthode B est de la famille des méthodes de développement formel basées sur les techniques de raffinement. la méthode B est bien outillée avec des plate-formes industrielles (telles que Atelier-B [1], B-Toolkit [2], Rodin [3]). Pourtant, bien que les méthodes formelles offrent des solutions au problème de la vérification (s'assurer que le système est correct par rapport à des propriétés), le problème de validation (la validation consiste à se demander si le texte formel "traduit" bien la demande informelle faite par celui qui commande le logiciel, en d'autres termes s'assurer que le système est correct par rapport aux spécifications) reste un défi majeur pour les ingénieurs de méthodes formelles. L'une des principales difficultés avec les spécifications formelles est la complexité des notations utilisées et donc leur lisibilité. Malheureusement, une mauvaise lisibilité empêche l'utilisateur de participer à la validation de ces spécifications et donc ne favorise pas la communication entre les différents membres d'un projet. Le manque d'implication et de contribution des membres d'un projet est traditionnellement la première cause d'échec de celui-ci.

La modélisation d'un cahier des charges à l'aide d'un langage semi-formel offre de multiples avantages. La plus grande souplesse d'expression d'un tel langage par rapport aux langages formels utilisés surtout s'il s'appuie sur un formalisme graphique, favorise la compréhension de la spécification. Dans un tel cadre de modélisation semi-formelle, la spontanéité n'est pas bloquée par les notations mathématiques complexes. La charge cognitive est alors davantage dévolue à l'activité de schématisation plutôt qu'à celle de formalisation. De plus, par rapport à un langage strictement formel, l'usage d'un langage semi-formel offre la possibilité :

— d'élargir le nombre des personnes aptes à contribuer aux spécifications, sans l'aide d'un ingénieur spécialiste des méthodes formelles ;

— D'économiser du temps pour l'élicitation des connaissances expertes dans un contexte de gestion des connaissances dans les organisations, libérant ainsi plus rapidement les experts pour la réalisation de leur travail habituel ;

— De réaliser un gain de qualité dans la représentation des connaissances, dans la mesure où le langage semi-formel possède une plus grande expressivité, permettant par exemple de représenter des connaissances procédurales, ce que ne permettent pas la plupart des langages formels.

La figure 2.1 montre les avantages et les inconvénients des différentes méthodes de modélisation. Les descriptions semi-formelles, type UML, doivent permettre une modélisation compréhensible et intuitive des différents éléments du système afin de faciliter notamment le dialogue des différents intervenants d'un projet. Ces descriptions manquent d'une sémantique précise, leur utilisation pour une analyse automatique est délicate. Les deux questions principales qui se posent alors sont (1) de développer des techniques facilitant

1. http ://www.atelierb.eu/
2. https ://github.com/edwardcrichton/BToolkit
3. http ://www.event-b.org/

le passage des descriptions semi-formelles vers les descriptions formelles à l'aide d'outils d'ingénierie dirigée par les modèles et (2) de développer des techniques et d'outils d'analyse pour les descriptions formelles utilisées. Nous pouvons donc affirmer que La famille des méthodes formelles est rigoureuse mais peu uti-

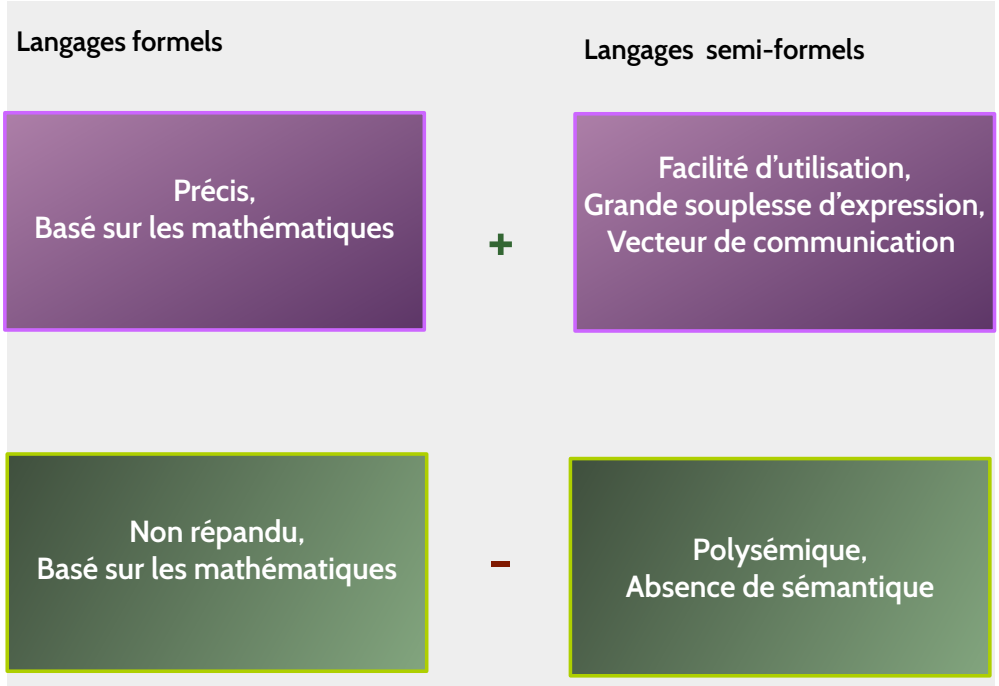


FIGURE 2.1 – Avantages et inconvénients des langages formels et semi-formels

lisée à cause de la complexité des notations utilisées. La famille des méthodes semi-formelles représente le système de manière intuitive et synthétique mais avec l'absence d'une sémantique précise des diverses notations utilisées et l'impossibilité de prouver la cohérence du système.

Nous nous intéressons particulièrement à l'utilisation conjointe des modélisations semi-formelles et formelles, et au passage de l'une vers l'autre, en vue de spécifier et de valider des systèmes complexes et critiques. Nous traitons dans ce chapitre l'utilisation conjointe de deux formalismes pour spécifier un problème. Le but est de mettre en correspondance ces deux spécifications afin de voir l'apport de chacune des méthodes à la résolution du problème. Cet intérêt est justifié par les aspects complémentaires et les apports croisés de ces deux techniques [85]. Le cas étudié est le robot typé.

Notre objectif est de rendre les notations formelles plus facilement compréhensibles. Nous avons donc extrait un diagramme de classes pour capturer la vue statique du système et un diagramme d'états transitions pour la vue dynamique. Le choix de la méthode B est motivé par l'existence d'outils permettant de prouver automatiquement les propriétés du système. Le langage UML[5] est un standard reconnu par l'OMG [4] et supporté par de nombreux outils de modélisation. UML intègre le langage OCL (Object Constraint Language) permettant d'exprimer des invariants, des pré-conditions, des gardes.
De nombreux travaux [86, 87, 88, 89, 90] ont été menés ces dernières années sur la génération (dérivation) de spécifications B à partir de diagrammes OMT ou UML soit à partir du diagramme de classe, soit à partir de diagrammes d'états. Les avantages d'une telle approche sont évidents : la modélisation est faite en dehors du cadre de la spécification B et elle permet aux concepteurs d'ignorer celle-ci. Les inconvénients sont dus à deux causes principales. Premièrement la notation B n'est pas orientée objet et par conséquent la spécification abstraite engendrée à partir d'une modélisation à objets est éloignée de celle que l'on aurait écrite directement. Deuxièmement, la spécification B n'est pas complète et il faut ajouter les éléments manquants manuellement, faire les preuves, puis poursuivre le processus avec l'écriture des raffinements. Les inconvénients nous paraissant l'emporter sur les avantages. Dans ce travail, nous nous sommes intéressés aux

4. Object Management Group

analogies utilisées dans UML et B. Pour cela nous avons commencé par comparer ces deux formalismes sur plusieurs études de cas dont le robot typé que nous traitons ci-après.

Après un bref résumé du cahier des charges du robot typé, nous présentons la spécification B, puis les diagrammes des classes et d'états transitions de la modélisation UML avec des contraintes exprimées à l'aide du langage OCL [91].

2.2 Exemple d'illustration

2.2.1 Présentation

Le système physique à piloter est schématisé dans la figure 2.2. Il est composé de six dispositifs :

— Deux dispositifs d'arrivée des pièces appelés *DA_A et DA_B*,

— Trois dispositifs d'évacuation des pièces appelés *DE_A* ,*DE_B et DE_C*,

— Un dispositif de transport des pièces, appelé DT, qui est un bras muni d'une pince ; le bras dispose de trois mouvements (vertical, horizontal et en profondeur) ; la pince a également un mouvement d'ouverture et de fermeture.

Trois types de pièces (T1, T2 et T3) sont prises en charge par ce robot avec les contraintes suivantes :

— Les deux dispositifs d'arrivée *DA_A et DA_B* et le dispositif de transport DT peuvent indifféremment supporter les trois types de pièces,

— Les pièces de type T1 ne peuvent être déchargées que par les dispositifs *DE_A et DE_C*,

— Les pièces de type T2 ne peuvent être déchargées que par les dispositifs *DE_A et DE_B*,

— Les pièces de type T3 ne peuvent être déchargées que par le dispositif *DE_C*.

Aucun des six dispositifs ne peut contenir plus d'une pièce à la fois.

2.2.2 Principes de fonctionnement

2.2.2.1 Transport et déchargement

— *DT* ne peut charger une pièce que si *DT* est libre,

— *DT* ne peut charger une pièce sur un dispositif d'évacuation que si ce dispositif est libre,

— Le dispositif d'évacuation *DE_A* ne peut recevoir que des pièces de type T1 et T2,

— *DE_B* ne peut recevoir que des pièces de type T2,

— *DE_C* ne peut recevoir que des pièces de type T1 et T3.

2.2.2.2 Chargement

Une pièce ne peut arriver sur un dispositif d'arrivée que si celui ci est libre

2.2.2.3 Mouvements

— *DT* décharge en position haute,

— *DT* charge en position basse,

— *DT* doit tenir une pièce pour monter,

— *DT* doit être vide pour descendre.

Dans un souci de simplification, nous ne présentons pas la modélisation des mouvements de la pince.

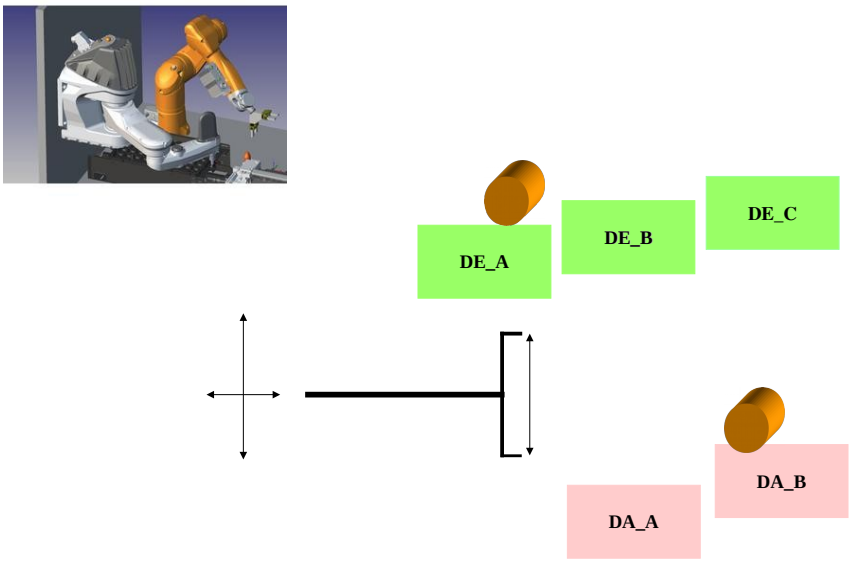


FIGURE 2.2 – Le robot typé

2.3 La spécification B

2.3.1 Présentation rapide de la méthode B

La méthode B, développée par Jean-Raymond Abrial, est une méthode formelle basée modèle qui permet un processus incrémental de développement (aussi appelée processus de raffinement) qui démarre à partir d'une spécification abstraite jusqu'à la génération automatique de code. La notion de machine abstraite est similaire à la notion de module et/ou d'objet que l'on retrouve dans les langages de programmation classique. Le concept central étant l'encapsulation, l'évolution de l'état d'une machine abstraite ne doit se faire qu'au travers des services/opérations de celle-ci. Les machines abstraites se répartissent en trois niveaux, les machines qui décrivent le niveau de spécification le plus abstrait, les *refinements* qui décrivent les étapes intermédiaires entre la spécification et le code et enfin, les *implementations* qui définissent le codage. Les machines abstraites sont composées de trois parties : la partie déclarative, la partie composition de machines et la partie opérationnelle.

La partie déclarative permet de décrire l'état d'une machine abstraite au travers de variables, de constantes, d'ensembles, et surtout de propriétés que doit toujours vérifier l'état de la machine. Cette partie est basée sur la théorie des ensembles et les prédicats du premier ordre.

Les clauses de composition (*SEES*, *INCLUDES*, *IMPORTS* et *EXTENDS*) permettent de décrire les différents liens entre machines abstraites, chaque clause introduisant des règles de visibilité sur les variables et les opérations des machines abstraites concernées. La partie opérationnelle, quant à elle, contient l'initialisation et les opérations de la machine abstraite, elle est basée sur le langage des substitutions généralisées.

2.3.2 Notations utilisées

B utilise les notations classiques sur les prédicats et sur les ensembles mais aussi des fonctions et opérateurs spécifiques. Nous définissons ci-dessous ceux qui sont nécessaires pour la compréhension de la spécification :

+->	Fonction partielle
<|	Restriction à une relation
<->	Relation
dom	Domaine
ran	Codomaine
|->	maplet(couple)

2.3.3 La spécification B du robot

```
MACHINE
Robot
SETS DE = {DE_A,DE_B,DE_C};
DA = {DA_A, DA_B};
DT = {DT_A};
TypePieces = {T1,T2,T3}
```

Des ensembles énumérés (`DE`) pour les dispositifs d'évacuation, `DA` pour les dispositifs d'arrivée, `DT` pour les dispositifs de transport et `TypePieces` permettent de décrire les différents éléments intervenant dans le cahier des charges du Robot.

```
VARIABLES transport, evacuation, arrivee,
autorisationEvacuation
INVARIANT
evacuation ∈ DE +-> TypePieces ∧
arrivee ∈ DA +-> TypePieces ∧
transport ∈ DT +-> TypePieces ∧
autorisationEvacuation
∈ DE <-> TypePieces ∧
∀(xx,yy).((xx |-> yy) ∈ evacuation ⇒ ((xx |-> yy) ∈
autorisationEvacuation)
```

La variable `evacuation` est ici une fonction partielle entre l'ensemble des dispositifs d'évacuation et des types de pièces : c'est donc un ensemble de couples ou le premier élément est un dispositif d'évacuation et le deuxième le type de pièces qui occupe ce dispositif. Les variables `arrivee` et `transport` se définissent de manière similaire.

Il faut aussi savoir quel type de pièces peut évacuer un dispositif de sortie. C'est le but de la relation `autorisationEvacuation` associée à la propriété qui la suit. Cette propriété assure que si un dispositif d'évacuation `xx` est occupé par une pièce de type `yy` alors ce type de pièce est autorisé à être évacué par le dispositif `xx`. La relation `autorisationEvacuation` est donc constituée de couples dont le premier élément indique le dispositif d'évacuation et le deuxième le type de pièce que peut évacuer ce dispositif.

```
INITIALISATION
autorisationEvacuation := {(DE_A |-> T1), (DE_B |-> T2),
(DE_C |-> T3),
(DE_C |-> T1),(DE_A |-> T2)} ||
evacuation = {} || transport = {} ||
arrivee = {}
```

Initialement aucun dispositif n'est occupé par une pièce. L'invariant est devenu vrai après l'initialisation. Le || exprime la simultanéité : le séquencement n'apparaît pas au niveau abstrait.

```
OPERATIONS
Arriver (yy) =
PRE  yy ∈ TypePieces ∧
card(dom(arrivee)) < card(DA)
THEN ANY da WHERE da ∈ DA − dom(arrivee)
THEN arrivee := arrivee ∪ {da |−> yy}
END
END;
```

Dans la clause **OPERATIONS** on trouve l'ensemble des opérations. La première opération est ici `Arriver` qui correspond à l'arrivée d'une pièce sur un dispositif d'arrivée. Cette opération est soumise à une précondition qui doit être vérifiée. Cette précondition permet d'attribuer un type au paramètre puis de vérifier qu'il existe au moins un dispositif libre. **ANY** permet de choisir n'importe quel dispositif *da* libre pour y placer la pièce : **ANY** est indéterministe. Au niveau le plus abstrait, l'indéterminisme est souvent utilisé puis progressivement abandonné pour des solutions déterministes lors des raffinements. L'implémentation qui constitue le dernier niveau de raffinement à partir duquel le code exécutable est généré est complètement déterministe.
Les opérations dans le langage *B* sont exprimées avec des substitutions généralisées comme **ANY**, :=, ||, ou **PRE**. Cette notion « opérationnelle » permet d'exprimer ce que font les opérations. De plus les substitutions peuvent être manipulées par le prouveur pour vérifier que l'invariant reste vrai après « l'exécution » de l'opération. Les opérations qui suivent sont basées sur le même principe.

```
Charger =
PRE
arrivee ≠ {} ∧ transport = {} /*DTestlibre*/
THEN ANY da,ty
WHERE (da |−> ty) ∈ arrivee ∧ ty ∈ TypePieces
THEN transport := {DT_A |−> ty} ||
arrivee := arrivee − {da |−> ty}
END
END;
```

```
Decharger =
PRE
ran(transport) ∩ ran(DE − dom(evacuation))
<| (autorisationEvacuation) ≠ {}
THEN ANY ty,de WHERE
ty ∈ TypePieces ∧ (DT_A |−> ty) ∈ transport ∧
de ∈ DE − dom(evacuation) ∧ (de |−> ty) ∈ autorisationEvacuation
THEN transport := {} || evacuation := evacuation ∪ de |−> ty
END
END;
```

```
Evacuer = /* libération d'un DE */
PRE
evacuation ≠ {} ∧
THEN ANY ev WHERE ev ∈ evacuation
THEN evacuation := evacuation − {ev}
END
END
END
```

2.4 Extraction des diagrammes UML à partir du modèle B

2.4.1 Introduction à UML

UML (Unified Modeling Language) est un langage de modélisation orientée objet développé en réponse à l'appel à propositions lancé par l'OMG (Object Management Group) dans le but de définir la notation standard pour la modélisation des applications construites à l'aide d'objets. Standardisé et encensé, UML a reçu le consensus de l'industrie logicielle et son utilisation est désormais incontournable pour toute démarche d'analyse et de conception objet. Dans la spécification, le modèle décrit les classes et les cas d'utilisation vus de l'utilisateur final du logiciel. Le modèle produit par une conception orientée objet est en général une extension du modèle issu de la spécification. Il enrichit ce dernier de classes, dites techniques, qui n'intéressent pas l'utilisateur final du logiciel mais seulement ses concepteurs. Il comprend les modèles des classes, des états et d'interaction. UML est également utilisée dans les phases terminales du développement avec les modèles de réalisation et de déploiement. UML est un langage utilisant une représentation graphique. Il est nécessaire de préciser qu'un langage tel que UML ne suffit pas à produire un développement de logiciel de qualité à lui seul. En effet, UML n'est qu'un formalisme, ou plutôt un ensemble de formalismes permettant d'appréhender un problème ou un domaine et de le modéliser, ni plus ni moins. Un formalisme n'est qu'un outil. Le succès du développement du logiciel dépend évidemment de la bonne utilisation d'un langage comme UML mais surtout de la façon dont on utilise ce langage à l'intérieur du cycle de développement du logiciel. Dans UML, il existe plusieurs formalismes, modèles ou bien diagrammes, comme celui des classes, des cas-d'utilisation (use-case), d'états-transitions et d'interactions,..
Le modèle des classes est le plus utilisé. C'est un formalisme pour représenter les concepts usuels de l'orienté objet. Le modèle des états-transitions et le modèle d'interaction permettent de représenter la dynamique des objets. Le modèle des cas d'utilisation permet de décrire les besoins de l'utilisateur final du logiciel. Dans nos travaux, nous nous sommes intéressés particulièrement à l'extraction des diagrammes de classes et d'états transitions.

2.4.2 Extraction du diagramme de classes

Le diagramme des classes

Ce modèle constitue la vue logique du système. Il montre une collection d'éléments statiques (classes), leur contenu et les relations entre eux.

Les classes

Dans notre système, nous distinguons deux classes de base qui sont `Dispositif` (2.3) et `Piece` (2.4). La classe `Piece` se spécialise en pièces de type `T1`, `T2` ou `T3`. Ces trois sous-classes héritent des attributs et méthodes de la classes `Piece` et peuvent posséder leurs propres attributs et méthodes. La classe `Dispositif` se décompose en trois sous-classes qui sont les dispositifs de transport `DT`, d'évacuation `DE` et d'arrivée `DA`. `DE` se spécialise en 3 sous classes correspondant aux différents types de dispositifs d'évacuation.

Les relations binaires

Les relations suivantes représentées sur la figure 2.5 :

— `Piece-DT` : Association Transport de type 0..1,0..1 ;`DT` peut transporter, à la fois, au plus une pièce.

— `Piece-DA` : Association Arrivée de type 0..1,0..1 ; Au plus, à la fois, une pièce peut arriver sur un dispositif `DA`.

— `Piece-DE` : Association Evacuation de type 0..1,0..1 ; Au plus, à la fois, une pièce peut être évacuée sur un dispositif d'évacuation.

— `DT-DA` : Association Chargement, dérivée des associations Transport et Arrivée.

— `DT-DE` : Association Déchargement, dérivée des associations Transport et Evacution

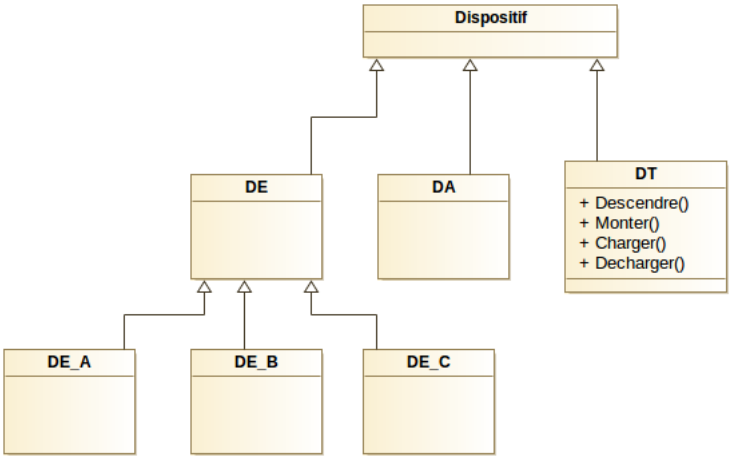


FIGURE 2.3 – Les types de dispositif

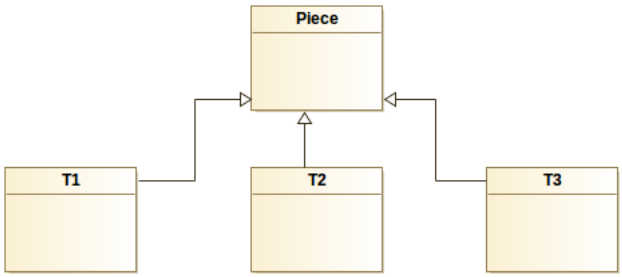


FIGURE 2.4 – Les types de pièces

Les contraintes

Les quelques contraintes en OCL que nous exprimons ci-après ne sont pas exprimables sur le diagramme des classes.

— **C1** : *Une pièce ne peut arriver sur un dispositif d'arrivée que si celui-ci est libre.*

context `DA : :` Arriver(T : Piece)
pre : Piece $\rightarrow$ *size* $= 0$
post : *Piece* $\rightarrow$ *size* $= 1$

— **C2** : *`DT` ne peut charger une pièce que s'il est libre et qu'il y a une pièce sur le dispositif d'arrivée.*

context `DT : :` Charger(T : Piece)
pre : Piece$\rightarrow$ *size* $= 0$**and**
T.DA $\rightarrow$ *size* $= 1$
post : *Piece* $\rightarrow$ *size* $= 1$**and**
T.DA $\rightarrow$ *size* $= 0$

— **C3** : *Le dispositif d'évacuation `DE_A` ne peut recevoir que des pièces du type `T1` et `T2` et le dispositif d'évacuation `DE_B` ne peut recevoir que des pièces du type T2.*

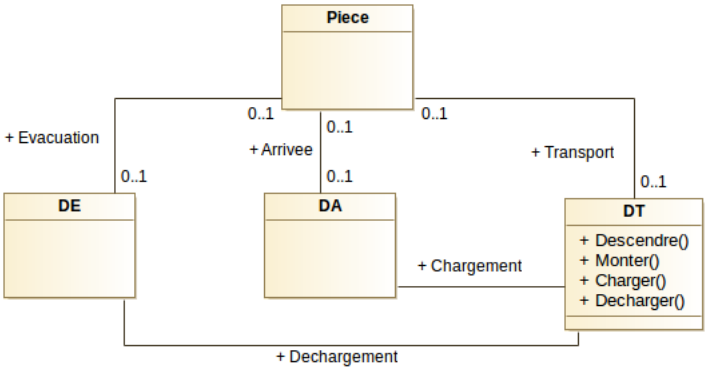


FIGURE 2.5 – Les associations

```
context DE inv :
Piece → forall(t : Piece | (self.oclIsTypeOf(DE_A) implies
(t.oclIsTypeOf(T2)ort.oclIsTypeOf(T1)))
or
(self.oclIsTypeOf(DE_B) implies t.oclIsTypeOf(T2))
or
(self.oclIsTypeOf(DE_C) implies (t.oclIsTypeOf(T1)
or
t.oclIsTypeOf(T3))))
```

— **C4** : *DT ne peut charger une pièce sur un dispositif d'évacuation que si ce dispositif est libre et si le dispositif d'évacuation est prévu pour ce type de pièces.*

```
context :: Decharger (e : DE)
pre :
Piece → size = 1 and e.Piece → size = 0
and
Piece → forall(t : Pièce |
(e.oclIsTypeOf(DE_A) implies (t.oclIsTypeOf(T2)
or
t.oclIsTypeOf(T1)))
or
(e.oclIsTypeOf(DE_B) implies t.oclIsTypeOf(T2))
or
(e.oclIsTypeOf(DE_C) implies (t.oclIsTypeOf(T1)
or
t.oclIsTypeOf(T3))))
post :
Piece → size = 0and
e.Piece → size = 1
```

Les contraintes portent sur les classes et sont donc des invariants (**C3**) ou portent sur des opérations d'une classe et permettent d'exprimer des pré-conditions et des post-conditions (**C1**, **C2**, **C4**).

2.4.3 Comparaison entre les 2 spécifications B et UML

Sur cet exemple simple qui a été retravaillé de façon itérative entre la spécification B et la modélisation UML, il est possible de montrer que les ensembles B et les valeurs énumérées des ensembles sont devenus des classes, les fonctions partielles `evacuation`, `arrivee` et `transport` des associations. Pour les pro-

priétés plus complexes énoncées en B (mais évidemment les plus intéressantes), il est nécessaire de faire intervenir les contraintes pour les exprimer en UML. La contrainte **C3** reflètent la relation `autorisationEvacuation` ainsi que la propriété faisant intervenir cette relation dans l'invariant de la spécification B. La pré-condition des contraintes **C1**, **C2** et **C4** est plus difficile à identifier dans la spécification B car on a adopté dans celle-ci un style d'opération sans paramètres. On peut d'ailleurs noter que la pré-condition de l'opération `Decharger` est aussi complexe dans les deux spécifications.
Le langage *ALF* (Action Language for Foundational UML) [92], dont la syntaxe est très proche de celle de Java, permet de définir facilement le comportement des opérations de classe. Sa syntaxe étant proche des langages de programmation traditionnels, il devient possible d'ecrire en *ALF* les substitutions généralisées de B qui permettraient de donner un corps aux opérations.

2.4.4 Extraction du diagramme d'Etats-Transition

Introduction

Cette partie concerne la représentation graphique de la partie dynamique d'une spécification B. Des travaux de couplage des parties comportementales entre UML et B ont été proposés, nous allons nous focaliser en particulier sur les travaux de traduction des Diagrammes d'Etats-Transitions (*DET*) d'UML pour représenter les aspects comportementaux des spécifications B. L'un des premiers travaux pour la combinaison des *DET* et B avait pour finalité de concevoir, de manière graphique, des systèmes réactifs [93]. Dans cette approche, la conception des systèmes réactifs est initialement réalisée au moyen de *StateCharts*, généralement utilisés pour leur forme graphique expressive, qui sont traduits, par la suite, en une machine abstraite B. Cette traduction permet, dans une certaine mesure, d'attribuer une sémantique formelle B aux *StateCharts* de D. Harel [94]. Différents aspects de ce travail nous paraissent intéressants à mettre en évidence tels que la prise en compte des notions d'états hiérarchiques et concurrents, ainsi que la notion d'états communicants. Bien que les règles proposées par [93] soient assez précises et couvrent la quasi-totalité des spécificités des diagrammes d'états, elles ne prennent pas en compte l'aspect statique, ou l'existence d'un lien avec un diagramme de classes. Les travaux de [95, 96] sont, entre autres, des tentatives de formalisation des *DET* d'UML ciblant la prise en compte des aspects statiques.

Les diagrammes d'états-transitions

Les diagrammes d'états-transitions héritent des *StateCharts* de Harel [94] et permettent la description des changements d'un état en réponse aux interactions avec d'autres états. Ils décrivent le comportement interne d'un objet à l'aide d'un automate à états finis. Ils présentent les séquences possibles d'états et d'actions qu'une instance de classe peut traiter au cours de son cycle de vie en réaction à des événements discrets (de type signaux, invocations de méthode). Ils spécifient habituellement le comportement d'une instance de classeur (classe ou composant), mais parfois aussi le comportement interne d'autres éléments tels que les cas d'utilisation, les sous-systèmes, les méthodes. Le *DET* est le seul diagramme, de la norme UML, à offrir une vision complète de l'ensemble des comportements de l'élément auquel il est attaché. En effet, un diagramme d'interaction n'offre qu'une vue partielle correspondant à un scénario sans spécifier comment les différents scénarii interagissent entre eux.

Approche adoptée

Dans notre démarche, à chaque machine B est associée un package UML et à chaque variable B est associée une classe UML. Dans le méta-modèle UML défini par l'OMG, chaque *DET* est attaché à une classe. Nous avons donc poursuivi cette conception en associant à chaque variable un *DET*. L'idée de base est présentée dans la figure 2.6. Les machines abstraites B sont représentées par un super état composite dont le nom est celui de la machine en question.

Chaque variable est représentée par un état composite inclus dans le super état composite de la machine abstraite. Les états correspondants aux variables sont concurrents et donc liés par la conjonction ET. Le *DET* correspondant à la machine raffinée inclut le *DET* des variables utilisées dans la machine source. Pour

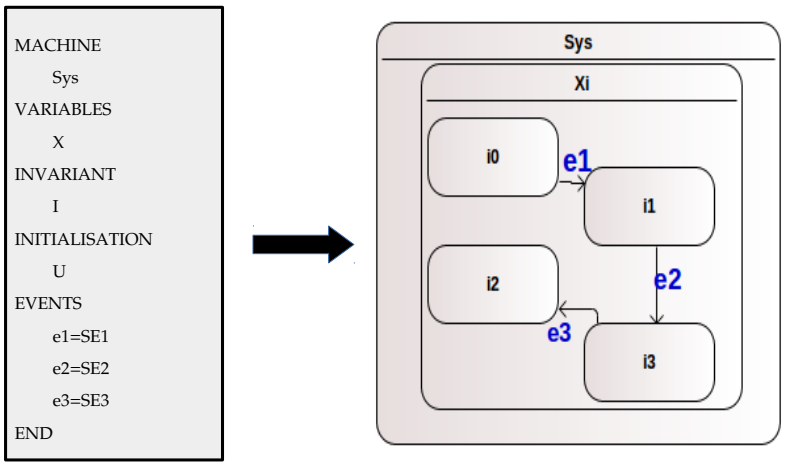


FIGURE 2.6 – L'idée de base

```
MACHINE Robot0
VARIABLES DT0
INVARIANT
DT0 ∈ {vid,occ}
INITIALISATION
DT0:=vid
EVENTS
Chgt ≙ SELECT DT0:=vid THEN DT0:=occ END
Dchgt ≙ SELECT DT0:=occ THEN DT0:=vid END
END
```

FIGURE 2.7 – Machine abstraite

chaque machine, nous utilisons les clauses VARIABLES et SETS pour sélectionner les variables qui ont un comportement dynamique susceptible d'être représenté par un *DET*.

Extraction du *DET* à partir de la machine abstraite

La machine abstraite représentée par la figure 2.7 décrit le dispositif *DT*. Dans la clause EVENTS, nous trouvons les événements *Chgt* et *Dchgt* qui correspondent respectivement aux chargement et déchargement des pièces par le dispositif de transport *DT*. L'événement *Chgt* fait passer le dispositif *DT* de libre ou vide (*vid*) à occupé (*occ*) et *Dchgt* le fait passer d'occupé (*occ*) à vide (*vid*). Le diagramme *DET* extrait (Fig.2.8) comme indiqué dans notre approche, consiste en un super état composite *Robot*0 qui décrit la machine *Robot*0 et l'état *DT*0 qui décrit la variable *DT*0. Nous notons que cette variable est constituée de deux états (*vid*) et (*occ*) qui correspondent respectivement à l'état vide (donc libre) et occupé. Les événements *Chgt* et *Dchgt* font passer la variable respectivement de (*vid*) à (*occ*) et inversement.

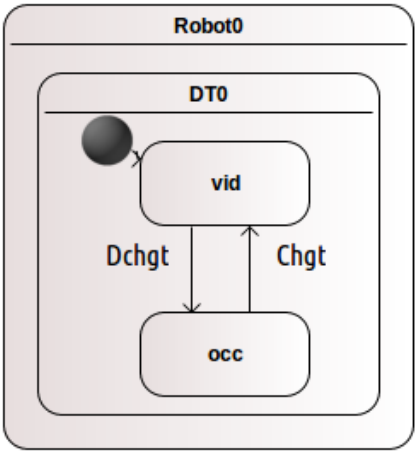


FIGURE 2.8 – DET du robot0

```
MACHINE Robot1 REFINES Robot0
VARIABLES DT1, DA1
INVARIANT
DT1 := DT0  DA1 ∈ {vid,occ}
INITIALISATION
DT1, DA1:=vid, vid
EVENTS
Chgt ≙ SELECT DA1:=occ  DT1:=vid THEN DA1, DT1:=vid, occ
END
Dchgt ≙ SELECT DT1:=occ THEN DT1:=vid END
arr_p ≙ SELECT DA1:=vid THEN DA1:=occ END
END
```

FIGURE 2.9 – Premier raffinement

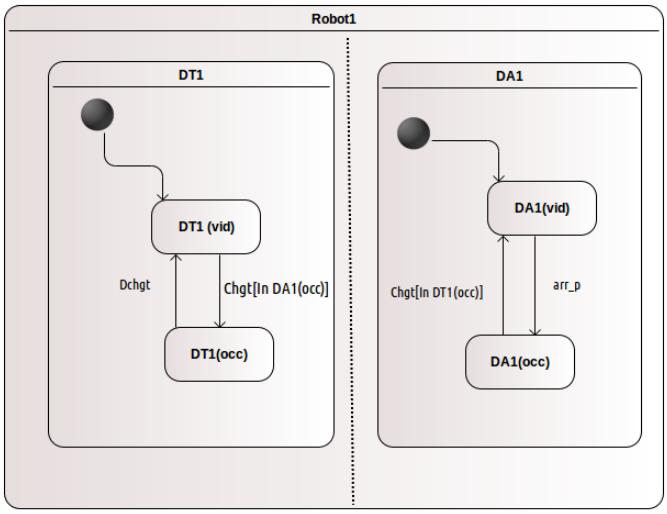


FIGURE 2.10 – DET du robot1

Extraction du *DET* à partir de la machine raffinée

Premier Raffinement.

Dans le premier raffinement (Fig. 2.10), nous introduisons le dispositif d'arrivée formalisé par la variable DA1. Cette variable a deux états : (*vid*) quand il n'y a aucune pièce qui arrive sur le dispositif DA1 et

```
MACHINE Robot2 REFINES Robot1
VARIABLES DT2, DA2, DE2
INVARIANT
DT2:=DT1  DA2 :=DA1  DE2 ∈ {vid,occ}
INITIALIASTION
DT2, DA2, DE2:=vid, vid, vid
EVENTS
Chgt ≙ SELECT DA2:=occ  DT2:=vid THEN DA2, DT2:=vid, occ
END
Dchgt ≙ SELECT DT2:=occ  DE2 := vid THEN DT, DE2:=vid,
occ END
arr_a ≙ SELECT DA2:=vid THEN DA2:=occ END
evac ≙ SELECT DE2:=occ THEN DE2:= vid END
END
```

FIGURE 2.11 – second raffinement

(*occ*) quand une pièce est arrivée. Cet événement est décrit dans la clause EVENTS par *arr_p*. L'événement *Chgt* exprime la propriété suivante : "*La pince du robot peut charger une pièce sur DA" uniquement si le dispositif de transport DT est libre (vid)*". Il décrit le changement d'états simultanés des variables DA1 (de *occ* à *vid*) si DT1 est dans l'état (*vid*) et de DT1 (de *vid* à *occ*) si DA1 est dans l'état (*occ*). L'événement *Dchgt* reste inchangé par rapport à la machine abstraite. Le *DET* extrait Fig.2.10, consiste en un super état composite $Robot_1$, composé de deux états composites et concurrents DA1 et DT1. Pour DT1, le changement d'état entre (*vid*) à (*occ*), activé par l'événement *Chgt*, et est conditionné par la satisfaction de la garde entre états [In $DA_1(occ)$], qui exprime que le dispositif DA1 soit dans l'état (*occ*). L'événement *Dchgt*" fait passer la variable DT1 de l'état (*occ*) à (*vid*). Pour DA1, L'événement *arr_p* fait passer la variable DA1 de l'état (*vid*) à (*occ*). L'événement *Chgt*, la fait passer de l'état (*occ*) à (*vid*) conditionné par la satisfaction de la garde entre états [In $DT_1(vid)$].

Second Raffinement.

Nous introduisons la variable DE2 qui modélise le dispositif d'évacuation. L'événement *evac* permet de passer la variable DE2 de l'état (*occ*) à (*vid*). L'événement *Dchgt* permet de spécifier cette propriété : "*La pince du robot ne peut décharger uniquement que sur un disposition d'évacuation libre*". Les événements *Chgt* et *arr_p* restent inchangés. La machine B du second raffinement est illustrée par la Fig.2.11 et le *DET* extrait est représenté par le diagramme de la figure 2.12.

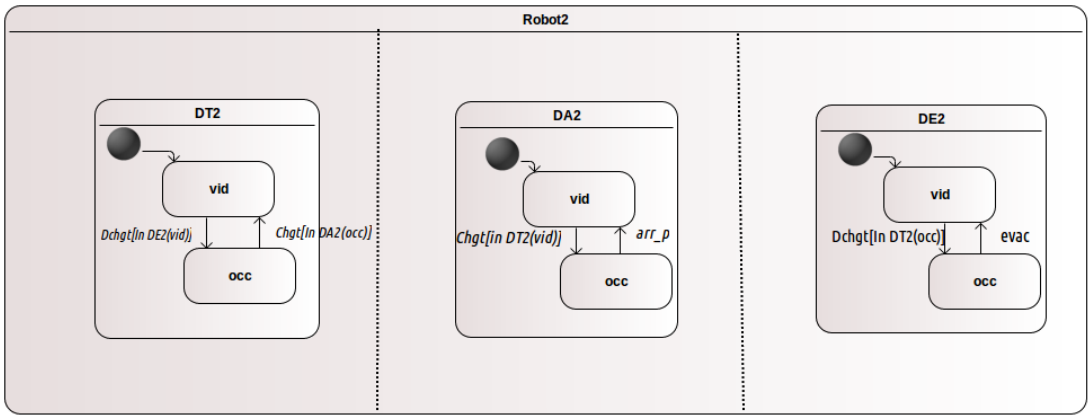


FIGURE 2.12 – DET du robot2

2.5 Travaux connexes

On peut distinguer trois classes d'utilisation d'UML avec des modèles formels :

1. formalisation d'UML,
2. dérivation d'un langage formel à partir d'UML,
3. extraction de diagrammes UML à partir de modèles formels.

En utilisant le langage OCL pour compléter le diagramme des classes on obtient une spécification assez comparable à celle d'un langage formel et probablement plus simple. Si UML a l'avantage d'être plus attrayant par son aspect visuel et par le grand nombre de diagrammes permettant d'avoir une vue complète du système à modéliser. Un langage formel comme *B* possède l'atout majeur de n'être pas simplement déclaratif mais de donner une substance aux opérations et à l'initialisation ce qui permet d'utiliser des outils (prouveurs, animateurs) permettant la vérification ou l'exécution de la spécification. Bien que des travaux de qualité [97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108] soient menés pour dériver des spécifications formelles (Z, Object Z, B, VDM,...) depuis une modélisation OMT ou UML, nous pensons que l'avenir réside dans l'intégration des mécanismes formels dans UML afin d'avoir le même paradigme orienté objet et de couvrir l'ensemble des aspects de la modélisation sans avoir à maîtriser deux langages de modélisation.

2.6 Bilan

Ce travail a été mené en étroite collaboration avec Bruno Tatibouet et a donné lieu :

1. **Projet** : Projet RNTL BOM [5] (Traduction du langage formel B optimisant la consommation mémoire)
2. **Thèse de doctorat en Automatique et en informatique- UFC** [11].

Année	Candidat	Titre
2004	J.C. Voisinet	*Contribution aux processus de développement d'applications spécifiées à l'aide de la méthode B par validation utilisant des vues UML et traduction vers des langages objet.*

TABLE 2.1 – Thèse sur B et UML

3. **DEA et Master 2 recherche**

Année	Candidat	Titre
2000	P. Clemente	*Vers le couplage entre UML et B à travers l'étude de cas du système de contrôle d'accès aux bâtiments.*
2001	M. Hariti	*Conception d'une base de données temps réel pour un système de productique à l'aide du langage UML.*
2002	R. Ait Saadi	*Modélisation comportementale avec UML d'un système de télé-maintenance.*
2003	S.A Khodja	*Modélisation en UML d'une phase de maintenance*

TABLE 2.2 – DEA et Master recherche B et UML

4. **Communications suivantes :** . [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22].

5. http ://vasco.imag.fr/Projets/bom.html

3

IDM et raffinement d'automates

Plan du chapitre

3.1 Contexte, problématique et motivations

La vérification des logiciels critiques est un domaine de recherche en plein essor, qui s'appuie sur une demande industrielle certes récente mais en forte croissance. Très souvent, ces logiciels critiques ont pour objectif le contrôle d'un processus qui communique avec son environnement par l'intermédiaire de capteurs, thermomètres, signaux, etc... Ces logiciels n'ont pas pour but de calculer un résultat, mais d'assurer le fonctionnement permanent du processus contrôlé, on les appelle des systèmes réactifs. Dans ce cadre, un logiciel défaillant ou une erreur de programmation peut avoir des coûts économiques très importants, ou en vies humaines. Il convient donc, lors de la phase de développement de telles applications, de s'assurer qu'elles satisfont un certain nombre de *propriétés*, notamment des propriétés de sûreté. Il y a pour cela deux grandes familles de techniques : les méthodes de preuve, où l'ingénieur utilise un *assistant de preuve* afin de prouver que le système étudié satisfait ses spécifications. La vérification formelle qui consiste en la création d'un modèle mathématique du programme, à la description par un langage formel des propriétés de celui-ci, et à l'utilisation des méthodes de vérification (en anglais, *model-checking*) afin de montrer la satisfaction de propriétés sur le modèle. Ainsi, la vérification formelle montre que tous les comportements du programme satisfont les propriétés.

Le principal problème réside dans la complexité des procédures de décision lors de la vérification de propriétés. En effet, les modèles engendrés sont de grande taille. En pratique, la taille de l'automate est déterminant pour la faisabilité du *model-checking*. Notre objectif est d'exploiter les concepts de raffinement pour atténuer cette complexité au moins en pratique. L'idée est d'introduire pas à pas des détails supplémentaires sur le système. Cette approche permet de décrire des spécifications abstraites de plus petite taille sur lesquelles on vérifie les propriétés exprimables au niveau de détail considéré. Puis le système est raffiné par introduction de nouveaux détails. Cette démarche ne résout le problème qu'à la condition que l'opération de raffinement préserve, sur le système raffiné, les propriétés vérifiées sur le système abstrait.

Notre premier travail s'articule autour de la spécification et de la vérification des systèmes hiérarchiques issus des travaux sur les *StateCharts* et Unified Modeling Language (UML). Diverses méthodes et techniques ont été introduites afin d'effectuer la vérification de propriétés par *model-checking* sur les modèles hiérarchiques [109, 110]. Dans [111], les auteurs montrent comment les *StateCharts* peuvent être traduits en *Promela* (le langage de spécification qu'utilise SPIN [112]) en utilisant *les automates hiérarchiques* comme format intermédiaire. Les techniques de vérification utilisées sont algorithmiques et sont souvent basées sur la détection de cycles pour montrer la satisfaction ou non d'une propriété. Les propriétés sont exprimées à l'aide d'une logique comme la *LTL* (*Linear Temporal Logic*) [113]. Une comparaison de différentes approches pour vérifier les *StateCharts* par *model-checking* peut être trouvée dans [114].

Le raffinement consiste à introduire pas à pas des détails supplémentaires sur le système en éclatant *les états de base* [1] du système abstrait. Cette approche permet de décrire des spécifications abstraites de plus petite taille sur lesquelles on vérifie les propriétés exprimables au niveau des détails considérés. Puis le système est raffiné par introduction de nouveaux détails qui peuvent être des automates ou un ensemble d'automates parallèles.

La contribution de ce travail est de définir un raffinement entre automates hiérarchiques de telle sorte qu'il garantisse le raffinement de leurs modèles qui sont des *structures de Kripke*. Cette démarche ne résout le problème qu'à condition que l'opération de raffinement préserve sur le système raffiné les propriétés vérifiées sur le système abstrait. Dans [115], les auteurs ont démontré que le raffinement de systèmes de transitions préserve les propriétés de logique temporelle linéaire. Donc, si une propriété *LTL* est vérifiée au niveau abstrait alors elle l'est également au niveau raffiné.

Cette modélisation possède des caractéristiques telles que le raffinement d'états, des transitions qui ont plusieurs états de départ et plusieurs états d'arrivée (Interlevel Transitions), la priorité entre les transitions et l'exécution simultanée des transitions.

Le second travail concerne la conception des systèmes temps réel sûrs. La qualité et la sûreté des systèmes temps réel dépendent non seulement des capacités des outils qui les valident, mais aussi de la manière dont ils sont modélisés et spécifiés. Dans ce travail, notre objectif est double. D'une part, nous examinons l'expression des spécifications des systèmes temps réels en B. Pour ce faire, et compte tenu que la mé-

1. Nous appelons état de base, l'état qui n'a pas une structure interne.

thode B ne permet pas de spécifier des variables réelles (les horloges), nous discrétisons le temps en tenant compte des remises à zéro et des contraintes d'horloges utilisées dans les automates temporisés. L'évolution de ces variables est modélisée par une opération *tic* qui fait évoluer le temps. D'autre part, nous étendons le concept de raffinement utilisé pour la spécification des systèmes réactifs, au modèle des systèmes temps réel. Enfin, nous proposons une démarche de vérification des systèmes temporisés par raffinement. L'objectif est de définir des méthodes de vérification de propriétés fondées sur le raffinement qui permettent de répondre au problème d'explosion combinatoire de l'exploration du graphe d'états.

Dans ce chapitre, j'ai choisi de présenter uniquement le raffinement d'automates hierarchiques. Ces travaux ont donné lieu à la :

— thèse de M. Al Achhab [24] et

— aux publications suivantes : [25, 30, 32, 26, 27, 28, 29]

Dans ce qui suit nous présentons les concepts de base : automate séquentiel, automate hiérarchique, sémantique d'automate hiérarchique sous forme de structure de *Kripke* et conditions de raffinement entre automates hiérarchiques. Dans la section 3.6, nous présentons le résultat principal sur la préservation de propriétés LTL ainsi qu'un outil qui permet de vérifier automatiquement la correction des conditions de raffinement, et ensuite nous dressons le bilan de ce travail.

Nous illustrons nos propos par l'exemple de Distributeur Automatique de Billets (DAB).

3.2 Préliminaires

Dans cette section nous rappellons la notion d'automate hiérarchique. Ce modèle a été proposé par Mikk et al. [23] pour représenter les *StateCharts* d'une manière plus formelle. Nous donnons également sa sémantique à l'aide des structures de *Kripke*. La sémantique de la LTL est donnée dans la section 3.2.6.
L'automate hiérarchique est un formalisme qui permet de modéliser un système réactif. Il se compose d'un ensemble d'automates séquentiels qui sont liés entre eux d'une manière parallèle et/ou hiérarchique. La hiérarchie consiste à lier un état d'un automate séquentiel et un autre automate ou un ensemble d'automates parallèles. Les automates parallèles exécutent leurs transitions d'une manière synchrone ou asynchrone. Nous commençons par définir la notion d'automate séquentiel.

3.2.1 Automate séquentiel

Un automate séquentiel est constitué d'un ensemble d'états, d'un ensemble de transitions et d'un alphabet permettant l'étiquetage des transitions.

Nous souhaitons retrouver, au niveau des automates, les variables du système modélisé. Les transitions correspondront à des changements de valeurs des variables considérées. Pour cela, nous introduisons les décors des états.

Soit $V = \{x_0, \ldots x_n\}$, un ensemble de variables. Soit $Dom(x_i)$ le domaine de la variable x_i, qui est un ensemble fini. Nous appelons $AP_V = \{ap, ap_0, ap_1, \ldots\}$ l'ensemble des propositions atomiques sur l'ensemble des variables V où une proposition atomique ap est de la forme $x_i = d_j, x_i \in V$ et $d_j \in Dom(x_i)$.

Définition 3.2.1. *(Automate séquentiel). Un automate séquentiel est un 5-uplet de la forme* $\langle S, s_0, \Sigma, \longrightarrow, L\rangle$ *où :*

— *S est un ensemble fini d'états,*

— $s_0 \in S$ *est l'état initial,*

— Σ *est un alphabet fini de noms d'actions,*

— $\longrightarrow \subseteq S \times \Sigma \times S$ *est l'ensemble des transitions,*

— $L : S \rightarrow 2^{AP_V}$ *est la fonction qui associe à chaque état de l'automate un ensemble de propositions atomiques.*

Dans tout le document, on note S_A l'ensemble des états de l'automate A, Σ_A l'ensemble d'actions de A, s_{0A} l'état initial de l'automate A. $L(s)$ définit l'ensemble des propositions atomiques de l'état s et $\longrightarrow_A$ désigne la relation de transition de A.

Une exécution σ de A est une séquence, finie ou infinie, d'états et d'actions $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots s_i \xrightarrow{a_i} \dots$ telle que s_0 est l'état initial et pour chaque $i \geq 0$, nous avons $s_i \xrightarrow{a_i} s_{i+1} \in \longrightarrow_A$. On note $Exec(A)$ l'ensemble des exécutions de l'automate A et $tr(\sigma) \stackrel{def}{=} a_0 a_1 \dots a_i \dots$ la trace de l'exécution σ.

Définition 3.2.2. *(Cycle) :*
On appelle cycle d'un automate $A = \langle S, s_0, \Sigma, \longrightarrow, L\rangle$ *une séquence finie d'états et d'actions* $Cy \stackrel{def}{=} s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} s_n$, *telle que* $s_n = s_0$, *et il existe une transition* $s_i \xrightarrow{a_i} s_{i+1} \in \longrightarrow_A$ *pour tout i, où* $0 \leq i < n$.

3.2.2 Automate hiérarchique

Un automate hiérarchique, noté AH, est un ensemble d'automates séquentiels liés entre eux par une fonction de composition. La fonction de composition établit le lien entre un état s d'un automate séquentiel et un ensemble d'automates $\{A_1, \dots A_i\}$. Nous appelons *s l'état père* de l'automate $A_{j_1 \leq j \leq i}$ et A_j *l'automate fils* de l'état s.

Soit $F = \{A_1, \dots A_n\}$ un ensemble d'automates séquentiels de AH. Nous supposons que les automates séquentiels, qui appartiennent à l'automate hiérarchique, ont un ensemble d'états distincts, c'est à dire qu'on ne peut pas trouver un état s qui appartient à deux automates différents.

Définition 3.2.3. *(Fonction de composition).*
Soit $F = \{A_1, \dots A_n\}$ *un ensemble d'automates séquentiels avec des ensembles d'états distincts. Nous appelons* $\gamma : \bigcup_{A \in F} S_A \longrightarrow 2^F$ *une fonction de composition sur F si :*

1. $\exists! A_{racine} \notin \bigcup rang(\gamma)$;
2. $(\bigcup rang(\gamma) = F \backslash A_{racine}) \wedge (\forall A.(A \in \bigcup rang(\gamma) \Rightarrow \exists s.(s \in \bigcup_{A' \in F \backslash \{A\}} S_{A'} \wedge A \in \gamma(s))))$;
3. $\forall S.(S \subseteq \bigcup_{A \in F} S_A \Rightarrow \exists s.(s \in S \wedge S \cap \bigcup_{A \in \gamma(s)} S_A = \emptyset))$.

La clause 1 énonce que dans l'ensemble F, il y a un seul automate racine A_{racine} qui n'est fils d'aucun état. La clause 2 énonce que l'image de la fonction γ est l'ensemble $F \backslash A_{racine}$ et que chaque automate $A \in F \backslash A_{racine}$ a un état père. Ces deux conditions sont nécessaires pour que l'automate racine soit l'ancêtre de tous les autres automates. La clause 3 énonce que la fonction de composition ne doit pas contenir de cycles.

Soit $|\gamma(s)|$ le nombre d'automates dans $\gamma(s)$. Un état s d'un automate $A \in F$ peut être raffiné soit par un seul automate $|\gamma(s)| = 1$, soit par des composants parallèles qui s'exécutent simultanément $|\gamma(s)| > 1$ sinon s est un état de base non raffiné $|\gamma(s)| = 0$, noté $Basic(s)$.

Définition 3.2.4. *(Automate hiérarchique) AH est un triplet* $\langle F, E, \gamma\rangle$ *où :*

- $F = \{A_1, \dots A_n\}$ *est un ensemble d'automates ayant des ensembles d'états distincts,*
- $E = \cup_{A \in F} \Sigma_A$ *est un alphabet fini de noms d'actions et*
- $\gamma : \bigcup_{A \in F} S_A \longrightarrow 2^F$ *est une fonction de composition sur F.*

3.2.3 Fonctions de successeurs et sous-automate hiérarchique

La fonction de composition γ sur un ensemble $F = \{A_0, \dots A_n\}$ constitue un lien entre un état et ses automates fils. Dans cette section, nous définissons des fonctions de successeurs induites par γ et liant un état et les états des automates fils.

On note χ l'application qui lie un état raffiné avec les états d'un automate fils : $s' \in \chi(s)$ si s' est un état d'un automate fils de s. χ^+ est l'application qui lie un état raffiné avec ses descendants. χ^+ est définie comme la fermeture transitive non-réflexive de χ. Aussi, nous définissons χ^* comme la fermeture transitive réflexive de χ. La fonction d'ancêtre, notée χ^{-1}, lie un état s et l'état ancêtre de l'automate père.

Proposition. *Le décor de l'état fils implique celui de son ascendant :* $\forall s' \in \chi(s) \Longrightarrow (L(s') \Longrightarrow L(s))$.

Soit $AH = \langle F, E, \gamma\rangle$ un automate hiérarchique et $\mathscr{S} = \cup_{A \in F} S_A$ l'ensemble des états de AH. Nous notons $\gamma' : \mathscr{S} \longrightarrow F$ l'application qui constitue un lien entre un état $s \in \mathscr{S}$ et son automate père. Nous notons $\gamma^{-*} : \mathscr{S} \longrightarrow F$ l'application qui constitue le lien entre un état s et son ancêtre A.

Dans l'exemple du DAB représenté dans la figure 3.1 $\chi(s_0) = \{s_2, s_3\}$, $\chi^{-1}(s_3) = \{s_0\}$ et $\gamma'(s_2) = \{DAB\}$.

Définition 3.2.5. *(Sous-automate hiérarchique)*
Soit $AH = \langle F, E, \gamma \rangle$ un automate hiérarchique, la restriction de la fonction de composition γ aux états d'un automate $A \in F$ nous permet de définir le sous-automate hiérarchique $AH_A = \langle F_A, E_A, \gamma_A \rangle$ tels que :

— $F_A = F \backslash \{A_i | S_{A_i} \cap \chi^*(S_A) = \emptyset\}$,

— $E_A = E$,

— $\gamma_A = \gamma | \chi^*(S_A)$. *(On considère A comme l'automate racine de AH_A).*

F_A est l'ensemble des automates fils descendants des états de A, E_A est l'ensemble d'actions et γ_A est la restriction de la fonction de composition à l'ensemble des états S_A.

3.2.4 Notion de configuration

Une configuration d'un automate hiérarchique permet de décrire l'état global de celui-ci à un instant précis. Donc, elle définit les états des automates séquentiels de l'automate hiérarchique activés simultanément.

Définition 3.2.6. *(Configuration) : Soit $S_t = \cup_{A \in F} S_A$ l'ensemble des états de $AH = \langle F, E, \gamma \rangle$ et soit C un ensemble d'états dans S_t. C est une configuration si :*

- *l'automate racine participe par un seul état à la configuration C : $\exists! s.(s \in S_{A_{racine}} \wedge s \in C)$ et*
- *la fermeture en bas : pour chaque état s dans C si s est raffiné par un automate A, alors, A participe aussi par un seul état à C : $\forall s, A.(s \in C \wedge A \in \gamma(s) \Rightarrow \exists! s'.(s' \in S_A \wedge s' \in C))$*

On note $Conf_A$ l'ensemble des configurations de l'automate hiérarchique AH_A où A est l'automate racine de AH_A.

Dans l'automate hiérarchique de la figure 3.1, les ensembles $\{s_0, s_2\}$, $\{s_0, s_3\}$ et $\{s_1\}$ sont des configurations. (s_0, s_2) est la configuration initiale.

Proposition. *Chaque automate $A \in F$ participe, avec au plus un seul état, à une configuration.*

3.2.5 Étiquetage des transitions dans l'automate hiérarchique

Pour représenter les diagrammes *StateCharts* à l'aide des automates hiérarchiques, *Mikk et al* [23, 111] ont ajouté deux gardes, *sr(source restriction)* et *td(target determinator)* aux transitions d'automates séquentiels. L'objectif est de préserver les informations des transitions qui ont plusieurs états de départ et plusieurs états d'arrivée ($Interlevel\ Transitions$).

L'étiquetage d'une transition t dans un automate $A \in F$ est défini par le triplet $t = s \xrightarrow{sr,a,td} s'$ tel que : $source(t) = s$, $but(t) = s'$, $action(t) = a$, $sr \subseteq \cup\chi^+(s)$ est une configuration sur le sous-automate hiérarchique $AH_{\gamma(s)}$ et $td \subseteq \cup\chi^+(s')$ est une configuration sur le sous-automate hiérarchique $AH_{\gamma(s')}$.

Dans l'exemple de la figure 3.1, la transition $(s_0 \xrightarrow{carteInseree} s_1)$ est définie comme suit :
$(s_0 \xrightarrow{carteInseree} s_1) \stackrel{def}{=} (s_0 \xrightarrow{\{s_2\},carteInseree,\emptyset} s_1)$.

Remarque. *Soient s et s' deux états d'un automate $A \in F$ et $t = s \xrightarrow{sr,a,td} s'$ une transition dans $\longrightarrow_A$. sr est employée pour déterminer dans quelle configuration t est activable et td est employée pour déterminer les états d'arrivée de t. Si s est raffiné par un ou plusieurs automates et $sr = \emptyset$ alors t est activable de n'importe quel état des automates fils de s. Si s' est raffiné par un ou plusieurs automates et $td = \emptyset$ alors les états d'arrivée de t sont les états initiaux des automates fils de s'.*

3.2.6 Logique temporelle linéaire

Soit p une proposition atomique de l'ensemble AP_V. Une formule temporelle φ de LTL est définie par la grammaire suivante : $\varphi ::= p | \neg\varphi | \varphi \wedge \varphi | \bigcirc \varphi | \varphi \mathscr{U} \varphi$.

Soient φ, φ_1, φ_2 des formules temporelles. Soit $\sigma = C_0, C_1, \ldots C_i, \ldots$ une exécution d'une structure de *Kripke SK*. Nous définissons que "φ est satisfaite à la configuration $C_{i_{i\geq 0}}$ de l'exécution σ", noté $\sigma(i) \models \varphi$, comme suit :

— $\sigma(i) \models p$ si et seulement si $p \in LK(C_i)$,

— $\sigma(i) \models \neg\varphi$ si et seulement si il n'est pas vrai que $\sigma(i) \models \varphi$,

— $\sigma(i) \models \varphi_1 \wedge \varphi_2$ si et seulement si $\sigma(i) \models \varphi_1 \wedge \sigma(i) \models \varphi_1$,

— $\sigma(i) \models \bigcirc\varphi$ si et seulement si $\sigma(i+1) \models \varphi$,

— $\sigma(i) \models \varphi_1 \mathscr{U} \varphi_2$ si et seulement si $\exists k.(k \geq i \wedge \sigma(k) \models \varphi_2 \wedge \forall j.(i \leq j < k \Rightarrow \sigma(j) \models \varphi_1))$.

Les autres opérateurs temporels se redéfinissent comme suit : $\Diamond\varphi$ peut être redéfini par $true \mathscr{U} \varphi$, $\Box\varphi$ peut être redéfini par $\neg\Diamond\neg\varphi$.

3.3 Exemple du distributeur automatique de billets

Nous allons illustrer les différentes notions citées par l'exemple d'un automate hiérarchique qui modélise le fonctionnement d'un Distributeur Automatique de Billets (DAB).

Le DAB est une machine automatique permettant aux clients d'une banque de faire des retraits d'argent. Les clients doivent posséder une carte bancaire valide. L'appareil est en sommeil (*Inactif*) ou en transaction (*Actif*). Il passe de l'état *Inactif* à *Actif* lorsqu'une carte est introduite dans la fente. Lorsqu'une carte est détectée, elle est ingérée et lue. Si la carte est reconnue, on passe à la phase de reconnaissance du client. Durant cette phase, le client peut interrompre la transaction par appui sur la touche «annulation». Le client choisit de retirer des billets de banque (le montant est contrôlé) puis imprimer un reçu. Lorsque l'opération demandée est effectuée alors la transaction est finie. L'automate hiérarchique modélisant le DAB est représenté dans la figure 3.1. Au départ, le DAB est inactif. Dans cet état, il peut être soit en état d'attente d'insertion d'une carte, soit en maintenance. Après insertion d'une carte, le DAB devient actif et authentifie certaines informations.

L'automate hiérarchique est composé de deux automates : $F = \{DAB, Inactif\}$. Au premier niveau d'observation, dans l'automate racine *DAB*, le distributeur peut être dans deux états s_1 (Actif) ou s_0 (Inactif) l'état initial. L'état s_0 est raffiné par l'automate Inactif pouvant, lui aussi, se trouver dans deux états s_2 (Attente) qui est son état initial ou s_3 (Maintenance). Quand le distributeur est dans l'état s_0 l'automate Inactif est activé.

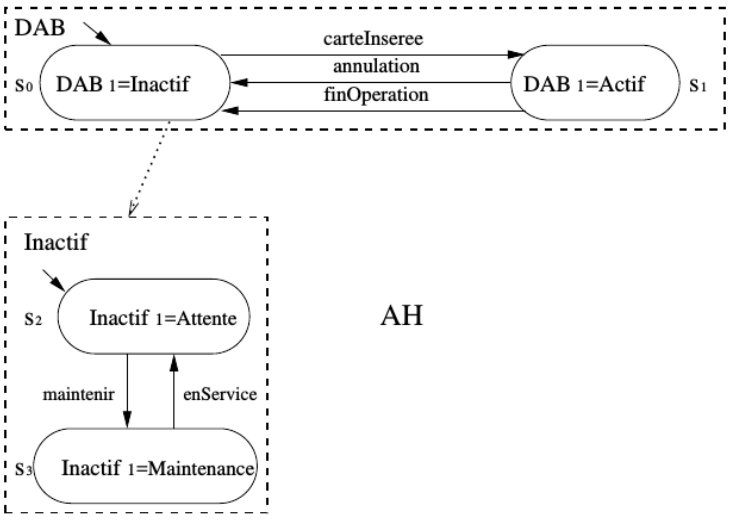


FIGURE 3.1 – Exemple d'automate hiérarchique

La fonction de composition γ est définie comme suit :
$\gamma = \{s_0 \longrightarrow \{Inactif\}\} \cup \{s \longrightarrow \emptyset | s \in \{s_1, s_2, s_3\}\}$.

3.4 Sémantique des automates hiérarchiques

3.4.1 Introduction

Les automates hiérarchiques étendus contiennent des concepts puissants, au niveau de leur spécification, comme la hiérarchie, le parallélisme, les gardes sur les transitions pour indiquer les états source et les états d'arrivée des transitions inter-niveaux. Ces concepts rendent difficile l'application directe des algorithmes de vérification des méthodes formelles. Dans [23], les auteurs ont défini une sémantique des automates hiérarchiques étendus sous la forme de structures de *Kripke* en prenant en considération la notion de priorité entre les transitions et l'exécution simultanée des transitions.

Pour définir cette sémantique, nous avons besoin de certaines notions comme la transition activable, la priorité entre les transitions et l'ensemble maximal des transitions activables sans conflit.

Définition 3.4.1. *(Transition activable)*
Soient $AH = \langle F, E, \gamma \rangle$ un automate hiérarchique, A un automate dans F, C une configuration dans $Conf_A$ et $e \subseteq E$ un ensemble d'actions. On dit que la transition $t \stackrel{def}{=} s \stackrel{sr,a,td}{\longrightarrow} s'$ de l'automate A est activable à partir de la configuration C, notée $activable_{(C|t)}$ et sur l'ensemble e si l'état source est actif ($s \in C$), sr est une configuration active ($sr \subseteq C$) et l'action a est présente ($a \in e$).

Définition 3.4.2. *(Priorité entre les transitions)*
Soient t_1 et t_2 deux transitions activables à partir de C|e. On dit que t_2 est prioritaire par rapport à t_1 si : $(source(t_1) \in \chi^+(source(t_2)))$.

Remarque. *La priorité entre les transitions dans les automates hiérarchiques, comme dans les StateCharts, consiste à exécuter les transitions activées de l'automate père. Puis, si aucune transition n'est activable à partir de l'automate père, les transitions des automates fils sont exécutées.*

Définition 3.4.3. *(Ensemble maximal des transitions sans conflit). Soit $ET_{(C|e)}$ l'ensemble des transitions étiquetées par l'ensemble des actions $e \subseteq E$ activables à partir de la configuration C. $trs \subseteq ET_{(C|e)}$ est appelé l'ensemble maximal des transitions sans conflit activables à partir de $(C|e)$ si :*

- *chaque automate participe avec au plus une seule transition : $\forall A \in F.(|trs \cap \longrightarrow_A| \leq 1)$;*
- *trs ne contient que des transitions activables ayant un même niveau de priorité : $\forall t \in ET_{(C|e)}.(t \in trs)$ s'il n'y a pas de transition $t' \in ET_{(C|e)}$ avec une priorité élevée.*

3.4.2 Structure de Kripke

Nous nous intéressons à des systèmes réactifs interagissant avec l'environnement. La sémantique de *AH* consiste à définir, à partir d'un ensemble d'actions venant de l'environnement, et de la configuration actuelle de *AH*, l'ensemble maximal des transitions sans conflit.

Définition 3.4.4. *(Structure de Kripke) : La sémantique de $AH = \langle F, E, \gamma \rangle$ est une structure de Kripke étiquetée $SK = \langle Conf, C_0, \longrightarrow_K, E, LK \rangle$ où :*

- *Conf est l'ensemble des configurations de AH,*
- *C_0 est la configuration initiale : $C_0 = (\chi|_{\mathscr{S}_0})^*(s_{0racine})$,*
- *E est l'ensemble d'actions,*
- *$LK : Conf \longrightarrow 2^{AP_V}$ telle que $LK(C) = \bigwedge_{s_i \in C} L(s_i)$,*
- *$\longrightarrow_K \subseteq Conf \times 2^E \times Conf$ est la relation de transitions de SK.*

$LK(C)$ est la conjonction de toutes les propositions atomiques des états qui constituent la configuration C.

Une transition $t \in \longrightarrow_K$ est un ensemble de transitions activables sans conflit définie par les règles suivantes :

— **Règle de progrès :** Cette règle s'applique à un automate A si l'un de ses états s est dans la configuration C. Si l'une des transitions partant de s est activable, alors l'une de ces transitions est exécutée.

$$\{s\} = C \cap S_A$$

$$\frac{\exists t \in \longrightarrow_A .(activable_{(C|e)}(t)) \wedge t = (s \xrightarrow{sr,a,td} s')}{A :: C \xrightarrow{e}_K (\{s'\} \cup td)}$$

— **Règle de composition :** Cette règle explique comment un automate délègue sa transition à ses automates fils : elle s'applique à un automate A ayant un état s dans la configuration C, telles que toutes les transitions partant de s ne sont pas activables et que l'état s est raffiné par un ou plusieurs automates. Les automates fils peuvent donc exécuter leurs transitions. Si l'état actif est raffiné par des automates parallèles, alors ces automates exécutent leurs transitions simultanément.

$$\{s\} = C \cap S_A$$

$$\forall t.(t \in \longrightarrow_A .(t = (s \xrightarrow{sr,a,td} s') \Rightarrow \neg activable_{(C|e)}(t)))$$

$$\gamma(s) = \{A_1, \ldots A_m\} \neq \emptyset$$

$$A_1 :: C \xrightarrow{e}_K C'_1$$

$$\frac{A_m :: C \xrightarrow{e}_K C'_m}{A :: C \xrightarrow{e} \{\{s\} \cup C'_1 \cup \cdots \cup C'_m\}}$$

— **Règle de bégaiement :** Cette règle est appliquée à un automate A ayant un état s dans la configuration C mais toutes les transitions partantes de s ne sont pas activables et s n'est pas raffiné par un autre automate.

$$\{s\} = C \cap S_A$$

$$Basic(s)$$

$$\frac{\forall t \in \longrightarrow_A .(t = (s \xrightarrow{sr,a,td} s')) \Rightarrow \neg activable_{(C|e)}(t)}{A :: C \xrightarrow{e}_K \{s\}}$$

La structure de *Kripke SK* associée à l'automate hiérarchique du DAB est représentée dans la figure 3.2. Normalement, nous ne pouvons pas représenter, sous forme graphique toutes les informations définies par la sémantique de AH. Nous avons représenté que les configurations et les transitions étiquetées par une seule action.

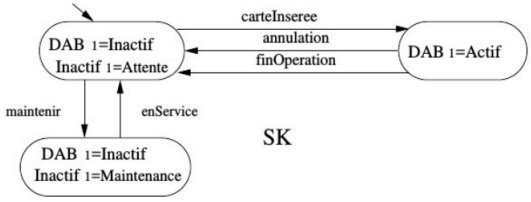


FIGURE 3.2 – *SK* associée à AH de la figure 3.1

Une exécution σ de *SK* est une séquence, finie ou infinie, d'états et d'actions $C_0 \xrightarrow{e_0} C_1 \xrightarrow{e_1} \cdots \xrightarrow{e_i} C_i \xrightarrow{e_{i+1}} \ldots$

Remarque. *Nous supposons que les états des automates hiérarchiques sont atteignables depuis l'état initial. Il se peut que certaines configurations du système ne soient pas atteignables. Cette hypothèse est réaliste car elle se situe au niveau de la spécification des automates hiérarchiques et peut être interprétée comme une anomalie dans les spécifications d'origine.*

3.5 Raffinement d'automates hiérarchiques

3.5.1 Introduction

Nous donnons dans cette section, les conditions de raffinement entre automates hiérarchiques en vue de préserver des propriétés temporelles sur des automates hiérarchiques. Ensuite, nous présentons une relation de raffinement entre automates hiérarchiques. Ce raffinement consiste à éclater les états de base de l'automate abstrait en un ensemble d'automates. Cette opération nous permet de détailler le comportement du système obtenu à partir d'un ou plusieurs états de base.

Dans toute la suite AH_1 et SK_1 dénotent la spécification de système abstrait sur l'ensemble des variables V_1 et AH_2 et SK_2 dénotent leurs spécifications raffinées sur l'ensemble des variables V_2. $A_1 = \langle S_1, s_{01}, \Sigma_1, \longrightarrow_1, L_1\rangle$ dénote un automate de AH_1 et $A_2 = \langle S_2, s_{02}, \Sigma_2, \longrightarrow_2, L_2\rangle$ dénote un automate de AH_2.

Le raffinement consiste à éclater les états de base de systèmes abstraits. Ces états sont remplacés par un ou plusieurs automates, ces automates sont notés A_τ. Les transitions de A_τ sont désignées par τ.

3.5.2 Relation de raffinement d'automates hiérarchiques

Nous définissons d'abord *la relation d'invariant de collage* sur V_1 et V_2 et *la relation de collage* μ entre les états de système raffiné et ceux du système abstrait.

Définition 3.5.1. *Un invariant de collage, notée I_{12}, est une relation entre les variables d'état du système abstrait $(\in V_1)$ et celles du système raffiné $(\in V_2)$ définie par une proposition de la forme suivante : $q ::= ap_1 | ap_2 | x_1 = x_2 | q \wedge q | \neg q$ où : $ap_1 \in AP_{V_1}, ap_2 \in AP_{V_2}, x_1 \in V_1$ et $x_2 \in V_2$.*

Définition 3.5.2. *La relation de collage entre les états de AH_1 et AH_2, notée μ, est une relation binaire $\mu \subseteq \cup_{A_2 \in F_2} S_{A_2} \times \cup_{A_1 \in F_1} S_{A_1}$ qui exprime le fait que les états sont collés si leurs propriétés d'états satisfont le collage.*

Autrement dit, les états $s_2 \in \cup_{A_2 \in F_2} S_{A_2}$ et $s_1 \in \cup_{A_1 \in F_1} S_{A_1}$ sont collés par la relation μ, notée $s_2 \,\mu\, s_1$, si et seulement si

$$\left(\bigwedge_{ap_2 \in L_2(s_2)} ap_2 \wedge \bigwedge_{ap_2 \notin L_2(s_2)} \neg ap_2 \wedge I_{12} \right) \Longrightarrow \left(\bigwedge_{ap_1 \in L_1(s_1)} ap_1 \wedge \bigwedge_{ap_1 \notin L_1(s_1)} \neg ap_1 \right)$$

3.5.3 Relation ρ proposée

Définition 3.5.3. *(Relation ρ). Soient $\mathscr{S}_1 = \cup_{A \in F_1} S_A$ et $\mathscr{S}_2 = \cup_{A \in F_2} S_A$ respectivement l'ensemble des états de AH_1 et AH_2. Soit I_{12} l'invariant de collage entre les variables de AH_1 et AH_2. Soit $\mu \subseteq \mathscr{S}_2 \times \mathscr{S}_1$ la relation de collage entre les états de $\mathscr{S}_2$ et ceux de $\mathscr{S}_1$. Nous définissons la relation de raffinement ρ comme la plus grande relation binaire incluse dans μ qui satisfait les conditions suivantes :*

1. *Raffinement de transitions :*
   (a) *les anciennes transitions sont raffinées : (voir la figure 3.3)*

$$s_2 \,\rho\, s_1 \wedge s_2 \xrightarrow{sr_2, a, td_2} s_2' \in \longrightarrow_2 \Longrightarrow \exists s_1' . (s_1 \xrightarrow{sr_1, a, td_1} s_1' \in \longrightarrow_1 \wedge s_2' \,\rho\, s_1' \wedge$$
$$(sr_1 \neq \emptyset \Longrightarrow sr_1 \rho sr_2) \wedge (sr_1 = \emptyset \Longrightarrow sr_2 = \emptyset \vee \exists A_\tau . (A_\tau \in \gamma(s_2))) \wedge$$
$$(td_1 \neq \emptyset \Longrightarrow td_1 \rho td_2)))) \wedge (td_1 = \emptyset \Longrightarrow td_2 = \emptyset \vee \exists A_\tau . (A_\tau \in \gamma(s_2'))))$$

   (b) *les τ-transitions bégaient : (voir A_τ la figure 3.3)*

$$s_2 \,\rho\, s_1 \wedge s_2 \xrightarrow{\tau} s_2' \in \longrightarrow_2 \Longrightarrow s_2 \xrightarrow{\tau} s_2' \in \longrightarrow_{A_\tau} \wedge s_2' \,\rho\, s_1$$

   (c) *pour les états terminaux*[2] *:*

$$s_2 \,\rho\, s_1 \wedge s_2 \not\longrightarrow \Longrightarrow s_1 \not\longrightarrow \vee$$
$$(s_2 \in S_{A_\tau} \wedge \exists t . (t \in \longrightarrow_A \wedge A \in \gamma^{-*}(s_2) \wedge (s_2 \in sr(t) \vee sr(t) = \emptyset)))$$

2. s est dit terminal s'il n'est l'état source d'aucune transition, on le note $s \not\longrightarrow$.

(d) pour les nouveaux cycles : (voir A_τ dans la figure 3.3)

$$cy \stackrel{def}{=} s_i \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_n} s_n \in Exec(A_\tau) \Longrightarrow$$

$$\exists t, j.(t \in \longrightarrow_A \wedge A \in \gamma^{-*}(s_i) \wedge i \leq j < n \wedge (s_j \in sr(t) \vee sr(t) = \emptyset))$$

2. *Préservation de la hiérarchie entre les états : (voir la figure 3.3)*

   (a) dans le système abstrait :

$$s_2 \rho\, s_1 \wedge A_1 \in \gamma(s_1) \Longrightarrow \exists A_2.(A_2 \in \gamma(s_2) \wedge s_{0_{A_2}} \rho\, s_{0_{A_1}})$$

   (b) dans le système raffiné :

$$s_2 \rho s_1 \wedge A_2 \in \gamma(s_2) \Longrightarrow s_{0_{A_2}} \rho s_1 \vee$$
$$(\exists t.A_1(t \in \longrightarrow_{A_2} \wedge action(t) \in E_1 \wedge A_1 \in \gamma(s_1) \wedge s_{0_{A_2}} \rho s_{0_{A_1}})$$

La figure 3.3 illustre les conditions de raffinement d'automates hiérarchiques. L'automate abstrait est représenté par les deux automates A_1 et A_2. Nous avons éclaté l'état s_2' en automate A_τ.

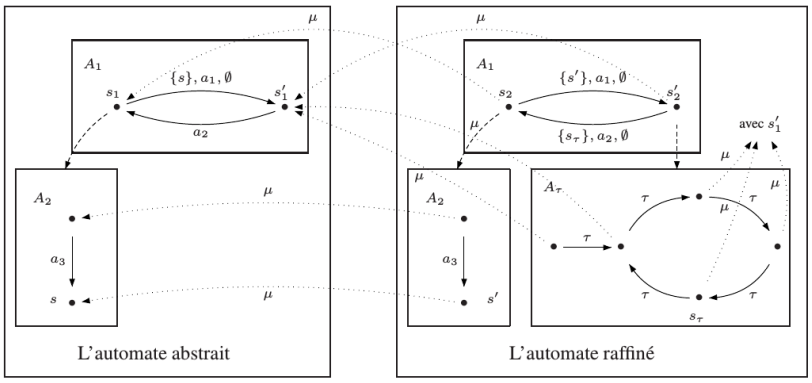


FIGURE 3.3 – Éclatement de l'état s_2'

La clause 1.*a* (voir la transition $s_1 \xrightarrow{\{s\},a_1,\emptyset} s_1'$ de la figure 3.3) énonce qu'il y a raffinement strict des transitions. La condition sur *sr* énonce que la transition $t_2 \stackrel{def}{=} s_2 \xrightarrow{sr_2,a,td_2} s_2'$ ne doit être activable qu'à partir de la configuration qui est liée à la configuration sr_1 de la transition $t_1 \stackrel{def}{=} s_1 \xrightarrow{sr_1,a,td_1} s_1'$. Cette condition est nécessaire pour que la transition t_2 dans le modèle raffiné soit activable à partir de la même configuration que la transition t_1. La condition sur *td* énonce que la configuration d'arrivée de la transition $s_2 \xrightarrow{sr_2,a,td_2} s_2'$ doit être collée à la configuration td_1 de la transition $s_1 \xrightarrow{sr_1,a,td_1} s_1'$. Cette condition est nécessaire pour que l'état d'arrivée de la transition dans le modèle raffiné soit la même que dans le modèle abstrait.

La clause 1.*b* énonce que toutes les nouvelles transitions doivent appartenir à des nouveaux automates et doivent également "bégayer" (voir les τ-transitions dans l'automate A_τ de la figure 3.3).

La clause 1.*c* énonce que le raffinement n'introduit pas de nouveaux deadlocks. Nous avons ajouté une condition sur les nouveaux états s_τ de l'automate A_τ qui ne sont états source d'aucune transition dans A_τ. Ces nouveaux états peuvent signaler de faux deadlocks. En effet, il peut exister une transition dans un automate ancêtre de s_τ qui est activable à partir de ces états.

La clause $1.d$ est utilisée pour interdire les τ-exécutions [3] infinies. Nous avons ajouté une condition, sur les cycles des nouveaux automates A_τ, qui énonce que pour chaque cycle des τ-transitions cy_τ, il existe au moins une transition, appartenant à un automate ancêtre de A_τ, sortante du cycle cy_τ. Au niveau sémantique des automates hiérarchiques, ce sont les transitions des automates ancêtres qui sont prioritaires par rapport à celles des automates fils. Donc, si la sortie du cycle est activée infiniment souvent, alors elle sera exécutée infiniment souvent. Ceci nous permet d'interdire les τ-exécutions infinies.

Les clauses $2.a$ et $2.b$ servent à préserver la hiérarchie entre les états dans les deux systèmes. Cela permet de préserver, dans le système raffiné, l'ordre de priorité entre les transitions, la décomposition parallèle des automates et la hiérarchie des états.

La clause $2.a$ énonce que dans le système abstrait, si un état s_1 est raffiné par un automate A_1, tel que s_1 est lié à s_2, alors s_2 est aussi raffiné par un automate A_2 et l'état initial de A_2 est lié à l'état initial de A_1 par la relation ρ.

La clause $2.b$ énonce que dans le système raffiné, si l'état s_2 est raffiné par un automate A_2, tel que s_2 est lié à s_1, alors soit l'état initial de A_2 est collé à s_1, soit dans A_2 il y a une transition étiquetée par une ancienne action, dans ce cas s_1 est aussi raffiné par un automate A_1 et l'état initial de A_2 est lié à l'état initial de A_1. Cette condition est nécessaire pour garder tous les automates fils abstraits.

Proposition. *Soit $s_2 \rho s_1$, si s_2 est un état de base alors s_1 l'est aussi.*

Preuve : Pour démontrer cette propriété, nous utilisons la preuve par contraposée, c'est à dire, pour montrer que (s_2 est un état de base) $\Longrightarrow$ (s_1 l'est aussi), il suffit de montrer qu'on a (s_1 n'est pas un état de base) $\Longrightarrow$ (s_2 ne l'est pas non plus).
s_1 n'est pas un état de base, donc $\gamma(s_1) \neq \emptyset$. Comme $s_2 \rho s_1$ et d'après la clause $2.a$ de la définition de raffinement $\gamma(s_2) \neq \emptyset$. Donc, s_2 n'est pas non plus un état de base.

Proposition. *Soit $s_2 \rho s_1$, si $A_\tau \in \gamma(s_2)$ alors tous les états de A_τ sont collés, avec la relation ρ, à l'état s_1.*

Preuve : D'après la clause $2.b$, l'état initial $s_{0\tau}$ de l'automate A_τ est collé avec l'état s_1. A_τ ne contient que les τ-transitions et d'après la clause $1.b$ $\forall s.(s \in S_{A_\tau} \wedge s \rho s_1)$.

Définition 3.5.4. *(Raffinement de AH) Soient s_{01} l'état initial de l'automate racine de AH_1 et s_{02} l'état initial de l'automate racine de AH_2. Nous disons que AH_1 est raffiné par AH_2, en accord avec un invariant de collage I_{12} si $s_{02} \rho s_{01}$.*

La définition de raffinement d'automates hiérarchiques signifie que l'état initial s_{02} de l'automate racine de AH_2 est lié par la relation ρ à l'état initial s_{01} de l'automate racine de AH_1. Il est suffisant que l'ensemble des états accessibles des deux systèmes soient reliés par la relation ρ. Ainsi une vérification algorithmique du raffinement des systèmes à nombre d'états fini, peut être faite par l'exploration conjointe des espaces d'états accessibles des automates hiérarchiques.

Une vérification algorithmique du raffinement est présentée dans [24].

La complexité est de l'ordre $O((|\mathscr{S}_1| + |\longrightarrow_1|).(|\mathscr{S}_2| + |\longrightarrow_2|))$ où $\mathscr{S}_1$, $\mathscr{S}_2$ sont les ensembles d'états de AH_1 et AH_2, et $\longrightarrow_1$, $\longrightarrow_2$ sont les ensembles des transitions.

3.5.4 Illustration sur l'exemple du DAB

Dans cette section, nous donnons deux exemples de raffinement de l'automate hiérarchique de la figure 3.1.

Premier raffinement

Dans le premier niveau de raffinement, nous allons détailler l'état actif du DAB. Lorsque le client insère sa carte, le DAB vérifie certaines informations concernant sa validité. Après validation, le client sélectionne une opération (retrait, ou juste imprimer un relevé de compte) et le DAB la traite. Ensuite, le client peut demander l'impression d'un reçu.

3. σ est une τ-exécution si la trace de σ n'est définie que par les actions τ.

Pour ce faire, au niveau spécification nous éclatons l'état s'_1 par l'automate Actif. La figure 3.4 représente l'automate hiérarchique raffiné de DAB abstrait.

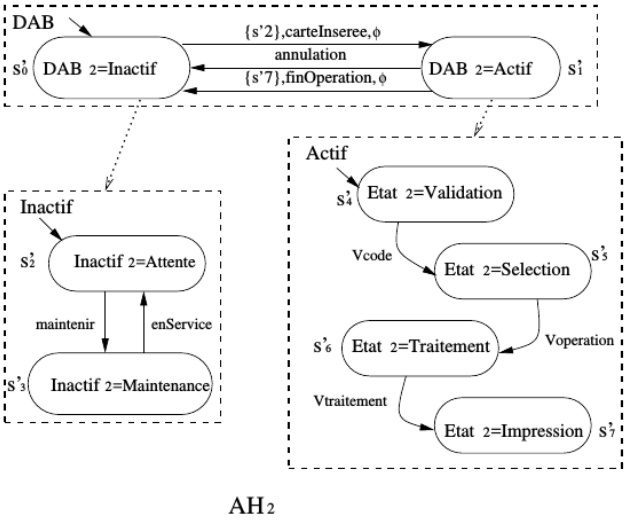


FIGURE 3.4 – Raffinement de niveau 1 de DAB

La variable *Etat* permet de désigner l'état du DAB à l'état actif, $Etat \in \{Validation, Selection, Traitement, Impression\}$. Les nouvelles actions de ce premier niveau de raffinement sont : $Vcose$: validation du code, $Voperation$: validation de la sélection, $Vtraitement$: validation du traitement.

La transition $s'_1 \overset{annulation}{\longrightarrow} s'_0$ est activable à partir de tous les états de l'automate Actif, par contre, la transition $s'_1 \overset{finOperation}{\longrightarrow} s'_0$ n'est activable qu'à partir de l'état s'_7 (*Impression*).

L'invariant de collage entre les variables de la spécification raffinée (DAB_2, $Inactif_2$, $Etat_2$) et de la spécification abstraite (DAB_1, $Inactif_1$) est défini comme suit :

$$I_{12} \overset{def}{=} (DAB_2 = DAB_1 \wedge Inactif_2 = Inactif_1 \wedge$$

$$DAB_2 = Actif \Rightarrow Etat_2 \in \{Validation, Selection, Traitement, Impression\})$$

Avec l'invariant de collage I_{12}, l'ensemble des couples de la relation μ est défini comme suit : $\mu \overset{def}{=} \{(s'_0, s_0), (s'_1, s_1), (s'_2, s_2), (s'_3, s_3), (s'_4, s_1), (s'_5, s_1), (s'_6, s_1), (s'_7, s_1)\}$.

D'après les deux automates hiérarchiques AH_1 (voir la figure 3.1) et AH_2 (voir la figure 3.4) et l'invariant de collage, AH_1 est raffiné par AH_2 car la relation de collage μ satisfait les clauses de raffinement entre automates hiérarchiques.

Deuxième raffinement

Nous allons détailler la spécification des deux états *Validation* et *Impression* de l'étape précédente. Concernant la validation de la carte, le client peut saisir deux fois le code. Si la première saisie est bonne, il peut traiter son opération. Sinon, il a droit de saisir son code une deuxième fois. En cas d'erreur, la carte peut être éjectée ou annulée.

Dans ce niveau de raffinement, cela revient à éclater l'état s'_4 par l'automate *TestCode* et l'état s''_7 par les trois automates (*Ticket*, *Carte*, *Billets*). La figure 3.5 représente le deuxième niveau de raffinement de l'automate hiérarchique raffiné de DAB abstrait.

La variable *Code* permet de désigner l'état de test du code, $TestCode \in \{Entre1, Entre2, Test1, Test2, Cbon\}$. La variable T permet de désigner l'état de l'impression de ticket, $T \in \{Impression, FinImpression, \}$. La variable C permet d'indiquer la position de la carte, $C \in \{In, Out\}$. La variable B permet d'indiquer l'état de la distribution des billets, $B \in \{Distribution, FinDistribution\}$.

Les nouvelles actions de ce deuxième niveau de raffinement sont : *TestCode* : pour tester le code, $Essai2$: le deuxième essai de code, *CodeV* : pour valider le code, $Erreur$: si erreur après les deux essais, $Imprime$: pour imprimer le ticket, $ejectCarte$: pour éjecter la carte, $distrBillets$: distribution de billets.

La transition $s'_1 \overset{annulation}{\longrightarrow} s'_0$ est activable à partir de tous les états des automates $Ticket$, $Carte$, $Billets$ et *TestCode*, par contre, la transition $s'_1 \overset{finOperation}{\longrightarrow} s'_0$ n'est activable qu'à partir de la configuration $(s''_7, s''_9, s''_{11}, s''_{12})$. La transition $s''_4 \overset{Vcode}{\longrightarrow} s''_5$ n'est activable qu'à partir de l'état s''_{17}.

L'invariant de collage entre les variables du deuxième niveau de raffinement et celles du premier niveau de raffinement est défini comme suit :

$$I_{12} \overset{def}{=} (DAB_3 = DAB_2 \wedge Inactif_3 = Inactif_2 \wedge Etat_3 = Etat_2 \wedge$$

$$Etat_3 = Validation \Rightarrow (Code \in \{Entre1, Entre2, Test1, Test2, Cbon, Erreur\}) \wedge$$

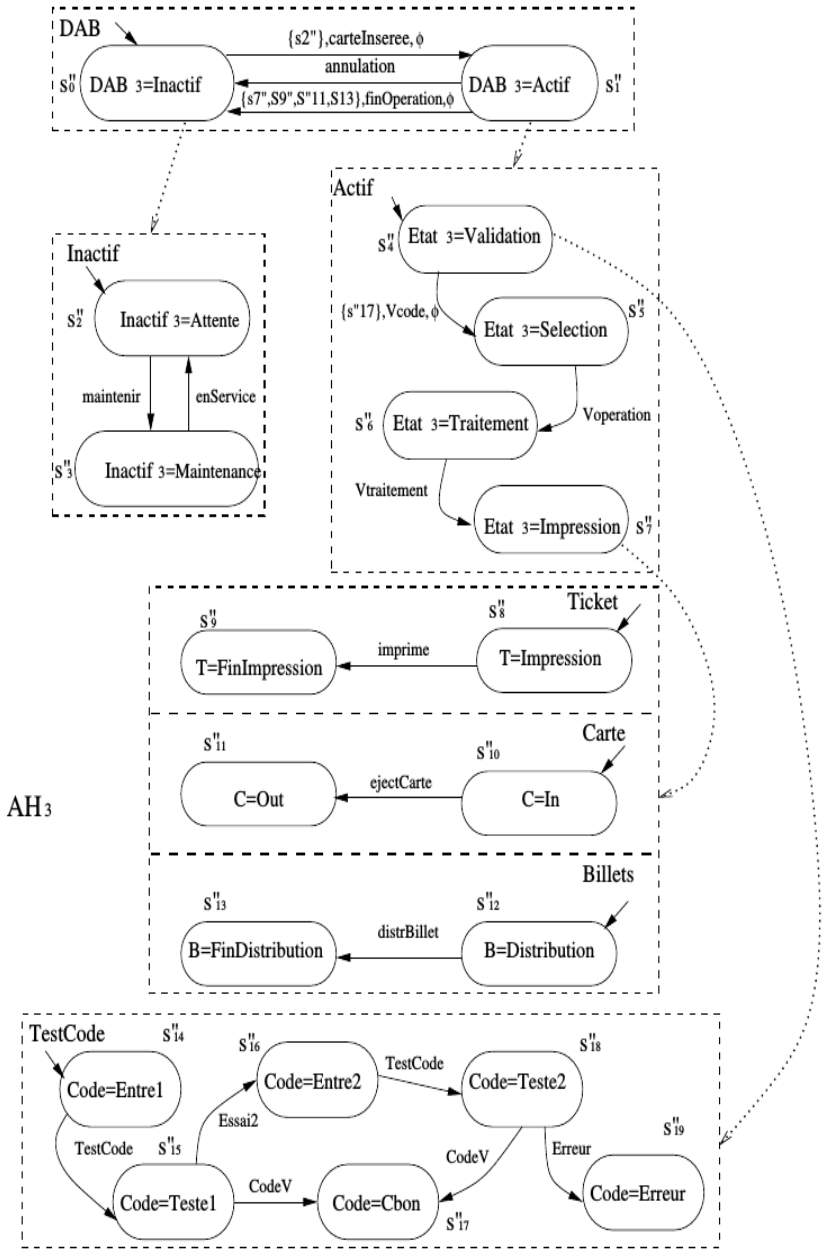


FIGURE 3.5 – Raffinement de niveau 2 de DAB

$$Etat_3 = Impression \Rightarrow (T \in \{Impression, FinImpression\} \wedge C \in \{In, Out\} \wedge$$
$$B \in \{Distribution, FinDistribution\}))$$

L'ensemble des couples de la relation μ est défini comme suit :

$$\mu \stackrel{def}{=} \{(s''_0, s'_0), (s''_1, s'_1), (s''_2, s'_2), (s''_3, s'_3), (s''_4, s'_4), (s''_5, s'_5), (s''_6, s'_6), (s''_7, s'_7)$$
$$(s''_8, s'_7), (s''_9, s'_7), (s''_{10}, s'_7), (s''_{11}, s'_7), (s''_{12}, s'_7), (s''_{13}, s'_7),$$
$$(s''_{14}, s''_4), (s''_{15}, s''_4), (s''_{16}, s''_4), (s''_{17}, s''_4), (s''_{18}, s''_4), (s''_{19}, s''_4)\}$$

La relation de collage μ satisfait les clauses de raffinement entre les deux automates hiérarchiques AH_2(voir la figure 3.4) et AH_3 (voir la figure 3.5).

3.6 Vérification par préservation

Nous avons défini les conditions de raffinement entre les automates hiérarchiques de telle manière qu'elles préservent les propriétés LTL sans l'opérateur $\bigcirc$. A partir d'un modèle abstrait AH_1, on vérifie la propriété φ sur sa sémantique SK_1 par la technique de *model-checking*. Pour montrer que φ est satisfaite sur AH_2, il suffit de vérifier les conditions de raffinement de la section précédente. Ces conditions permettent effectivement d'éviter les deadlocks sur SK_2 par la clause *c* et l'absence de livelocks par la clause *d*.

Théorème 3.6.1. *Soient φ une formule LTL, AH_1 et AH_2 deux automates hiérarchiques auxquels sont associées respectivement SK_1 et SK_2. Si AH_1 est raffiné par AH_2 et SK_1 satisfait φ alors SK_2 satisfait φ.*

Nous présentons quelques exemples de formules LTL vérifiées sur la structure de *Kripke* abstraite (voir la figure 3.2), associée à l'automate hiérarchique qui modélise le DAB au niveau abstrait (voir la figure 3.1).

— **Propriété 1 :** $\varphi_1 \stackrel{def}{=} \Box((DAB_1 = Actif) \Rightarrow \Diamond(DAB_1 = Inactif))$ cette propriété exprime que si le distributeur est actif, il finira par être inactif.

— **Propriété 2 :** $\varphi_2 \stackrel{def}{=} \Box((Inactif_1 = Maintenance) \Rightarrow \Diamond(DAB_1 = Actif))$ cette propriété exprime que si le DAB est en maintenance, alors il finira inévitablement par être dans l'état actif.

Ces deux propriétés sont satisfaites sur la structure de *Kripke* abstraite. Les clauses de raffinement entre automates hiérarchiques sont satisfaites entre AH_1 (voir la figure 3.1) et AH_2 (voir la figure 3.4),on en déduit alors que la structure de *Kripke* associée à AH_2 satisfait les propriétés φ_1 et φ_2 par préservation.

3.7 L'outil *MARhS*

Nous avons réalisé le logiciel *MARhS* (Modelisation and Analysis of Refined hierarchical Systems) qui est un prototype permettant principalement de vérifier la correction des conditions de raffinement.

MARhS est un début d'outillage développé en Java. Il permet de spécifier les automates hiérarchiques et de vérifier si les conditions de raffinement sont satisfaites.

La figure 3.6 présente ses fonctionnalités. Il prend en entrée trois fichiers : un automate hiérarchique abstrait, un automate hiérarchique raffiné et un invariant de collage. Ces trois fichiers sont décrits textuellement par une grammaire simplifiée.

L'outil *MARhS* peut aussi convertir un automate hiérarchique en langage Promela pour qu'il puisse être utilisé dans l'environnement `Spin`.

En utilisant l'invariant de collage, *MARhS* commence par définir les couples liés avec la relation de collage μ, dans un premier temps, il vérifie une propriété *LTL* sur le modèle *Promela* généré à partir de l'automate hiérarchique abstrait à l'aide de *Spin*. Après, il teste si les six conditions de raffinement sont vérifiées, si c'est le cas, on peut déduire que le modèle raffiné satisfait le propriété, sinon il fournit un diagnostic sur les conditions non vérifiées. *MARhS* permet également la génération de modèles Promela à partir des *SK* et de les visualiser en *.Dot*

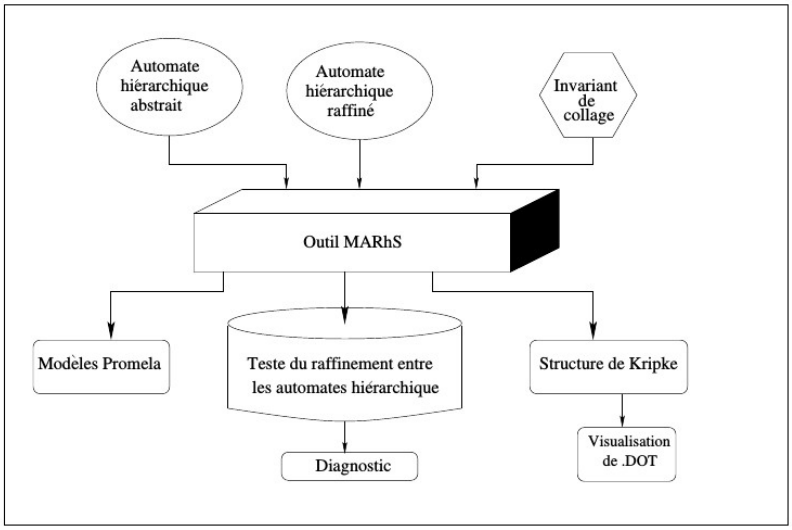


FIGURE 3.6 – Fonctionnement de l'outil MARhS.

3.8 Bilan

Ce travail a été mené en étroite collaboration avec Hassan Mountassir et a donné lieu :

1. **Projet** :
   — 2007-2011 : Projet PHC Volubilis [4] *TENEMO Télé-NEuroScience sur une plateforme collaborative Mobile sur Internet*. Coopération avec L'Université de Rabat (Maroc). Ma contribution concernait la modélisation et la vérification de propriétés d'un système collaboratif en neurosciences.
2. **Thèse de doctorat en Automatique et en informatique- UFC** [24]. Voir tableau 3.1.

Année	Candidat	Titre
2006	M. Al Achhab	*Vérification des systèmes hiérarchiques et préservation de propriétés linéaires par raffinement.*

TABLE 3.1 – Thèse sur le raffinement d'automates hiérarchiques

3. **DEA-Master 2 recherche**. Voir tableau 3.2
4. **Communications suivantes :** . [25, 26, 27, 28, 29, 30, 31, 32, 34, 35, 36, 37, 38, 39, 116, 117, 118]

4. http ://homepages.laas.fr/khalil/page/index.php ?n=PROJETS.TENEMO

Année	Candidats	Titre
2002	D. Okalas	*Contribution à l'expression en B et au raffinement des systèmes réactifs temps-réel.*
2003	M. Al A chhab J.B Rota Grasiozi	*Contribution à la définition des conditions du raffinement des automates temporisés* *Contribution à la préservation des proporiétés MITL par le raffinement des automates temporisés*

TABLE 3.2 – DEA et Master recherche sur le raffinement d'automates

4

Démarche de spécification et de validation des systèmes complexes

Plan du chapitre

4.1 Introduction

Ces dernières années, les systèmes sont devenus de plus en plus complexes en contenant plusieurs constituants hétérogènes et en combinant plusieurs aspects de différentes technologies. Ils ont des comportements plus élaborés et plus difficiles à prévoir, ont un nombre plus important de constituants en interaction et/ou réalisent des fonctions de plus haut niveau. Parallèlement à cette notion de complexité, s'ajoute également une notion de criticité qui peut être liée à la qualité des systèmes, notamment lorsque ceux-ci mettent en jeu un risque de vies humaines ou un risque financier important. En effet, elle peut être liée à la compétitivité du marché mondial qui impose aux développeurs de systèmes des contraintes de coût et de délai de plus en plus strictes.
La conception de ces systèmes, qualifiés de complexes, nécessite une approche rigoureuse qui permet de répondre aux exigences et de satisfaire les diverses contraintes de développement (coût, délai, qualité, fonctionnement, etc). C'est afin de formaliser et d'appréhender la conception de ce type de systèmes qu'est apparue l'Ingénierie Système (IS) qui définit une démarche méthodologique, pour maîtriser la conception des systèmes et des produits complexes [1]. Traditionnellement, l'ingénierie des systèmes s'appuyait exclusivement sur les documents. Ceux-ci (quel qu'en soit le support : papier, fichiers ou bases de données électroniques) constituent le moyen principal pour décrire, concevoir et analyser un système, pour communiquer et pour partager les informations entre les différentes parties prenantes de son développement. Cette ingénierie s'appuyant sur des documents a montré son efficacité mais a aussi des limites, surtout avec la complexité croissante des systèmes. Ces limites sont relatives d'une part à la difficulté de développer des systèmes complexes avec des équipes de plus en plus importantes, réparties sur des sites distants et d'autre part à la gestion et à l'exploitation difficile des différents documents produits.
L'INCOSE [2] suggère l'utilisation d'une approche s'appuyant sur les modèles MBSE (Model Based System Engineering) et propose un langage semi-formel (graphique) qui est un profil UML dédié à la modélisation des systèmes complexes (logiciel et matériel). Ainsi, l'INCOSE a développé une vision à court terme, dans laquelle, il considère que l'ingénierie des systèmes sera supportée par des modèles avec des pratiques d'ingénierie plus rigoureuses et plus formalisées. L'ingénierie dirigée par les modèles (IDM), ou Model Driven Engineering (MDE) en anglais, a permis plusieurs améliorations significatives dans le développement de systèmes complexes en permettant de se concentrer sur une préoccupation plus abstraite que la réalisation pratique.
Spécifier un système consiste à spécifier ces exigences, sa structure et ses comportements. Spécifier les exigences relatives à un système consiste à définir des capacités, des propriétés ou des contraintes qui doivent être satisfaites par le système. Les exigences peuvent être des fonctions que le système devra réaliser ou des conditions de performances, de fiabilité ou de sécurité. En conséquence, elles peuvent être donc classées en au moins deux catégories : fonctionnelles et non fonctionnelles.
La validation des systèmes dès les premières phases de conception est nécessaire pour assurer la correction des modèles conceptuels. Dans ce cadre, des travaux ont été proposés, notamment la méthode AVATAR [119] qui est un environnent qui inclut une méthode outillée, adaptée aux systèmes temps réel et distribués, assistée par l'outil Ttool [120]. Le langage AVATAR est un profil de SysML. Il étend SysML en proposant le langage TEPE pour l'expression des propriétés. Cette méthodologie concerne seulement la vérification des propriétés par *model-checking*. La traçabilité des exigences et la validation des exigences non fonctionnelles ne sont pas prises en compte dans cet environnement. L'approche OMEGA2 [121, 122] inclut un profil UML/SysML exécutable dédié pour la spécification et la validation formelle des systèmes critiques temps réel. Les modèles OMEGA2 utilisent l'outil IFx [123, 124] pour la simulation et la vérification des propriétés. Dans cette approche la simulation des contraintes physiques entre blocs matériels et la traçabilité des exigences ne sont pas considérées. Cependant, valider un système hétérogène complexe nécessite une démarche méthodologique permettant de considérer tous les aspects à savoir exigences fonctionnelles et non fonctionnelles et permettre leur validation automatique par vérification formelle et/ou par simulation. Cette démarche doit se baser sur la modélisation des exigences, de leur traçabilité, de la structure et du comportement du système. Dans [125], l'auteur a décrit certains des principaux modèles méthodologiques basées sur l'ingénierie des systèmes (MBSE) utilisés dans l'industrie aujourd'hui. Dans ce rapport, une méthodologie est définie comme une collection de processus, de méthodes et d'outils. Une méthodologie MBSE peut être caractérisée comme la collection de processus connexes, de méthodes et des outils utilisés

1. www.afis.fr
2. INternational COuncil on Systems Engineering

pour soutenir cette méthodologie. Le but de ce survey, qui est assez complet, est de fournir les différentes méthodologies MBSE qui sont disponibles dans le commerce ou candidates pour l'être. Ces méthodologies ne se basent pas sur SysML.

4.2 Démarche de validation

4.2.1 Introduction

La validation des systèmes complexes, très tôt dans le cycle de développement nécessite une démarche de spécification permettant la simulation des exigences non fonctionnelles et la vérification des exigences fonctionnelles. Notre approche introduit la validation dès les premières phases du cycle de développement. De plus, elle tient compte des aspects relatifs à la vérification et à la simulation lors de la phase de spécification. En effet, notre approche propose l'utilisation du langage SysML comme langage de spécification et les techniques d'ingénierie système en vue de transformer des modèles SysML vers des environnements de simulation et/ou de vérification. Avant de présenter notre approche, nous motivons le choix des langages utilisés.

4.2.2 Langage de spécification

SysML est un langage graphique suffisamment riche pour modéliser des systèmes critiques et complexes. SysML permet de décrire les exigences, la structure et les comportements d'un système et de modéliser ses exigences dans un modèle permettant de les décrire et de les lier avec les autres éléments de la modélisation. A travers le diagramme d'exigences, SysML fournit un support efficace pour la spécification des exigences dans les phases d'analyse et de conception. Pour ces avantages offerts par SysML, en plus de son adoption par l'OMG comme un standard, nous l'avons choisi comme langage de spécification. D'autres langages de modélisation existent, comme par exemple MARTE (Modeling and Analysis of Real Time and Embedded Systems) [7] ou AADL (Architecture Analysis and Description Language) [126]. MARTE est aussi un profil d'UML 2 pour l'analyse et la modélisation de systèmes temps-réel et embarqués. AADL est un langage de conception d'architecture standardisé par SAE (Society of Automative Engineers) destiné, comme MARTE, à la conception et à l'analyse de systèmes embarqués complexes et temps-réel. La différence entre ces langages de modélisation et SysML est le niveau d'abstraction des modèles réalisés. De ce point de vue nous estimons que SysML est plus générique et peut être utilisé pour la modélisation d'un système avec un niveau d'abstraction plus élevé que MARTE et AADL qui sont orientés métiers et spécifiques au domaine d'application des systèmes temps-réel et embarqués. De plus, l'avantage de SysML par rapport à ses langages est le fait de pouvoir modéliser les exigences et les lier aux modèles de conception. Il est à noter que ces langages peuvent coexister dans le même projet (ex. SysML et MARTE) pour spécifier un même système à des niveaux de détails différents.

4.2.3 Environnements de validation

Les langages semi-formels reposent généralement sur des notations graphiques. Ils sont très utiles pour décrire des systèmes complexes, car ils sont faciles à apprendre et donc à utiliser. Malgré qu'ils sont définis selon une syntaxe, ils souffrent souvent de l'absence d'une sémantique précise. Le langage SysML fait partie de cette classe de langages. Le passage vers des modèles formels permettant la vérification et/ou la simulation est donc indispensable en vue de valider les systèmes modélisés en SysML. Plusieurs langages de simulation et de vérification ont été utilisés dans cette approche. Le choix de ces langages est guidé par la nature de l'application dans le cadre des projets de recherche. Dans le cadre des environnements de simulation, le langage VHDL-AMS a été utilisé dans le cadre du projet ANR Smart-surface, le langage Modelica dans les projets ANR smart-blocs et Région SyvAD. Pour les environnements de vérification, les réseaux de Petri dans le cadre du projet TASSILI (avec USTHB, Algerie), les automates d'interfaces et le langage B dans le cadre du projet ANR TACOS.

4.3 Présentation de l'approche

Notre approche de validation est centrée sur les modèles. La première phase dans la démarche proposée est l'identification des besoins et l'élicitation des exigences. Cette phase permet la modélisation des exigences, la définition des relations entre exigences et la classification de celles-ci (fonctionnelles, non fonctionnelles). Le diagramme d'exigences SysML sert de support dans cette phase. La deuxième phase concerne la modélisation du système en mettant l'accent sur sa structure et son comportement. L'objet de celle-ci est de représenter le système sous forme de blocs. Puis, de décrire les interactions et les contraintes physiques entre ces blocs, ainsi que leurs comportements. Dans cette phase, des liens de traçabilité, de satisfaction et de vérification sont définis entre les exigences et les éléments du modèles. Les diagrammes SysML considérés sont les diagrammes de blocs, de blocs internes, paramétrique pour la structure et les diagrammes de séquence et d'activité pour le comportement. La première et la deuxième phase constituent l'étape de modélisation du système. L'étape suivante de l'approche utilise les techniques d'ingénierie dirigée par les modèles pour transformer les diagrammes SysML vers des environnements de simulation ou de vérification en fonction de la catégorie des exigences(fonctionnelles et non fonctionnelles). L'objectif de cette étape est de produire des modèles prêts à être validés par vérification ou par simulation. La dernière étape concerne l'utilisation des environnements de simulation (VHDL-AMS, Modelica, ..) et de vérification (Model-checker CPN-Tools, SPIN, B, ..) en vue de confronter les modèles de conception aux modèles d'exigences. La figure 4.1 décrit les étapes de notre approche.

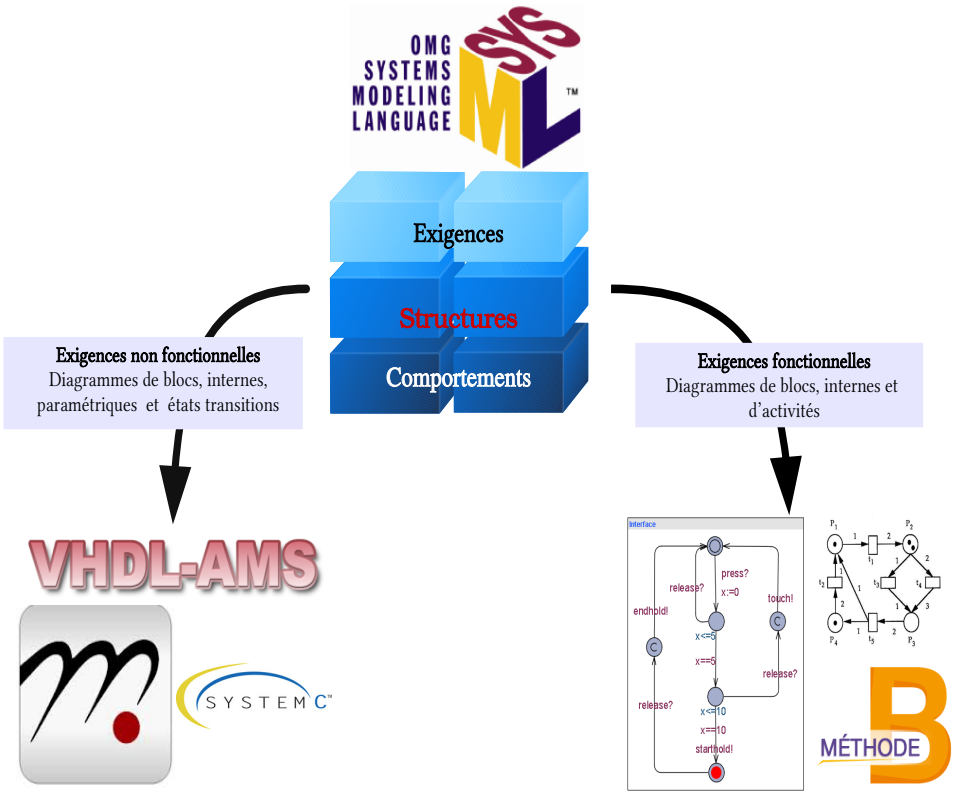


FIGURE 4.1 – Méthodologie de spécification et de validation

Dans ce chapitre, nous avons présenté brièvement notre approche de validation des systèmes hétérogènes et complexes. Les différentes étapes de cette approche seront détaillées dans les chapitres suivants en présentant nos démarches de validation par vérification (chapitre 5) et par simulation (chapitre 6).

5

Démarche de validation par vérification formelle

Plan du chapitre

5.1 Introduction

Dans le cycle de développement des systèmes complexes et critiques, la phase de vérification est essentielle pour s'assurer que les exigences soient satisfaites. Le langage SysML a été proposé pour modéliser des systèmes complexes. Malgré ses avantages et sa richesse pour décrire ce type de systèmes, il reste un langage semi-formel et ne propose pas de mécanismes de vérification. Ainsi, l'utilisation conjointe de SysML avec des langages formels s'avère bénéfique pour tirer profit des possibilités de spécification offertes par SysML et pour pouvoir assurer que les modèles SysML vérifient les exigences qui y sont relatives.
Nous commençons ce chapitre par la présentation de la démarche de spécification et de vérification proposée. Ensuite, afin d'illustrer notre approche, nous proposons son application pour un distributeur automatique de titres de transport (Ticket Vending Machine : TVM). Nous présentons par la suite, les limites de l'approche, notamment par rapport au problème d'explosion combinatoire de l'espace d'états et nous proposons une solution basée sur le découpage de modèles pour pallier ce problème. Avant de tracer le bilan de ce travail, nous mettrons en évidence les avantages de l'approche proposée par rapport aux travaux connexes, combinant SysML et les méthodes formelles.

5.2 Méthodologie de vérification

Dans cette section, nous présentons notre démarche pour spécifier et vérifier des systèmes complexes. Les différentes étapes de la démarche sont illustrées par la figure 5.1.

— La première étape consiste à spécifier le système et ses exigences selon la démarche proposée dans le chapitre 4. Cette étape permet de décrire les exigences, la structure et le comportement du système tout en pensant à la phase de vérification. La spécification d'exigences se fait en se basant sur le diagramme d'exigences SysML. Les exigences fonctionnelles (dans notre cas, celles à vérifier sur les diagrammes d'activités) doivent être formalisées en les exprimant en *AcTRL* (Activity Temporel Requirement Language)[64]. La formalisation des exigences vise à lever toute ambigüité sur leur sémantique afin de pouvoir les vérifier formellement. La spécification du comportement et de la structure du système se base aussi sur SysML. En particulier, nous utilisons le diagramme de définition de blocs pour décrire la structure du système et le diagramme d'activité pour décrire ses comportements.

— La deuxième étape consiste à considérer les relations entre exigences et éléments du modèle en vue de déterminer les diagrammes à prendre en compte lors de la phase de vérification. Cette traçabilité guide par la suite la procédure de vérification des exigences sur les modèles de conception. Ensuite, nous transformons les diagrammes SysML considérés vers des spécifications formelles en vue d'exprimer leur sémantique d'exécution et d'utiliser des outils de vérification formelle. Cependant, la définition d'une telle transformation reste difficile. Elle nécessite de bien connaître les domaines source et cible, de manière à trouver des équivalences structurelles et sémantiques entre les éléments sources et cibles. Par leur similarité avec les diagrammes d'activité, notre choix s'est porté sur les réseaux de Petri hiérarchiques colorés (*RdPHCs*). Le formalisme RdPHCs permet de capturer la sémantique de synchronisation et de parallélisme et offre la possibilité de modéliser des flux de contrôle et des flux de données. De plus, plusieurs outils supportent la vérification des RdPHCs et implémentent des model-checkers pour vérifier des propriétés exprimées par des formules de logique temporelle comme LTL et CTL.

— Une fois la transformation en spécifications formelles effectuée, la vérification des exigences SysML est entamée en se basant sur la technique du *model-checking*. Celle-ci est basée sur le *model-checking*, complètement automatique et permet de vérifier la satisfaction des propriétés écrites en logique temporelle. Elle fournit aussi des contre-exemples lorsque la propriété à vérifier n'est pas satisfaite. Il s'agit du scénario (trace d'exécution) qui a mené à l'erreur. Ces contre-exemples sont très utiles, car ils fournissent des indications importantes pour la correction du modèle sous la vérification.

— L'interprétation des résultats de la vérification sur la spécification de haut niveau (spécification SysML) constitue la dernière étape de notre démarche. Cette étape d'interprétation des résultats de vérification repose sur l'expertise humaine. Elle doit permettre de détecter l'origine de l'erreur dans la spécification SysML (erreur dans le modèle d'exigences ou dans les modèles de conception).

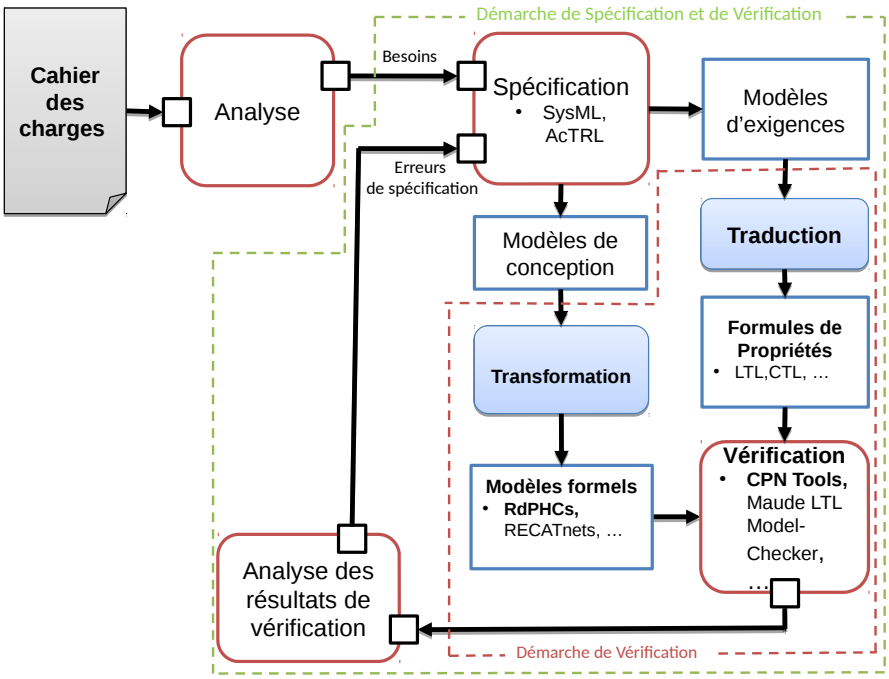


FIGURE 5.1 – Démarche de Spécification et de Vérification.

5.2.1 Formalisation des exigences

Notre proposition pour la formalisation des exigences concernent des comportements modélisés par des diagrammes d'activités. Cette particularité est prise en compte dans la définition du langage *AcTRL* [64]. Ce langage a été conçu pour être utilisé par des praticiens SysML. Il permet de formaliser, tout en restant à un niveau d'abstraction proche de SysML, les exigences décrites par des textes naturels, en les exprimant comme des propriétés temporelles concernant les éléments des diagrammes d'activités.
Dans le diagramme d'exigences SysML, le concepteur peut lier une exigence *R* à une activité SysML A, par une relation « verify » afin d'exprimer que l'activité A vérifie l'exigence R. Si cette exigence est fonctionnelle et concerne un comportement du système modélisé par un diagramme d'activités, elle peut être sûrement exprimée en fonction des états des éléments des activités SysML. Donc, une formalisation possible (au niveau SysML) des exigences liées aux activités consiste à les exprimer en fonction des états des éléments du diagramme d'activités. C'est en partant de cette idée, que nous avons proposé le langage *AcTRL*.
Le langage *AcTRL* est donc destiné aux concepteurs SysML pour décrire les exigences écrites en langage naturel en un langage plus formel mais proche de SysML. Il ne concerne que les exigences liées aux activités SysML. Il permet d'exprimer une exigence comme une propriété concernant les éléments de l'activité sur laquelle cette exigence sera vérifiée. Afin de définir ce langage, nous nous sommes basés sur la définition d'une représentation de haut niveau de la sémantique opérationnelle d'une activité SysML. Comme un système d'états/transitions, cette représentation décrit tous les états possibles induits par l'exécution de l'activité. Après la définition de cette représentation, nous avons défini des prédicats pour caractériser les états des éléments du diagramme d'activité et exprimer leurs propriétés. Dans cette proposition, nous avons utilisé OCL [127] pour exprimer les propriétés des objets et des noeuds d'objets. Par la suite, nous nous sommes basés sur le système de patrons de spécification proposé dans [128] pour doter le langage d'opérateurs temporels (comme présenté dans la figure 5.2). Pour des raisons de traçabilité, nous proposons de représenter les exigences exprimées en langage *AcTRL* dans le diagramme d'exigences SysML et de les lier aux exigences exprimées en langage naturel. Ceci est illustré par la figure 5.3. Nous remarquons qu'après la formalisation de l'exigence *Req1*, qui est liée par une relation "verify" avec un élément SysML stéréotypé par "test case" représentant une activité SysML, on génère une autre exigence *DReq1* exprimée en *AcTRL*. Cette dernière sera liée à l'exigence *Req1* par la relation "DeriveReq" et la relation "verify" sera entre l'exigence *DReq1* exprimée en *AcTRL* et l'élément "test case".

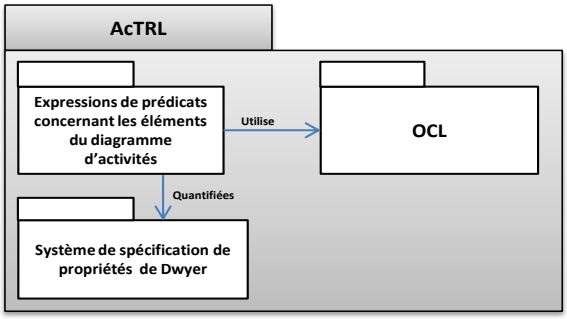


FIGURE 5.2 – Définition du langage *AcTRL*.

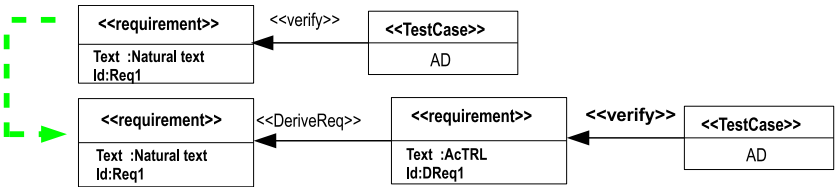


FIGURE 5.3 – Représentation des exigences exprimées en *AcTRL*.

5.2.2 Transformation en spécifications formelles

Notre approche se base sur une transformation des diagrammes d'activités en réseaux de Petri hiérarchiques colorés (*RdPHCs*). Avant de détailler cette transformation, nous présentons les *RdPHCs*.

5.2.2.1 Réseaux de Petri Hiérarchiques Colorés

Les réseaux de Petri hiérarchiques colorés (*RdPHC*) [129] sont une classe des réseaux de Petri de haut niveau proposée pour permettre la modélisation compositionnelle des systèmes.
Un modèle *RdPHC* est construit à partir de plusieurs réseaux de Petri colorés reliés les uns aux autres. Dans les *RdPHC*, le concept de transition de substitution permet de faire référence à un réseau de Petri coloré appelé subnet. Ce dernier est décrit dans une page séparée et donne une description plus détaillée de la partie du système modélisée par la transition de substitution.
La notion de transition de substitution est utilisée pour permettre de structurer le *RdPHC* en plusieurs pages (plusieurs sous-réseaux). Une transition de substitution a des places en entrée et des places en sortie qu'on appelle places sockets. La relation entre une transition de substitution et son sous-réseau est donnée par la spécification qui lie les places du sous-réseau avec les places sockets de la transition de substitution. Ces concepts sont illustrés par la figure 5.4. Plus formellement, un *RdPHC* peut être défini comme suit :

Définition 5.2.1. *(Réseau de Petri hiérarchique coloré)*
Un Réseau de Petri Hiérarchique Coloré *(RdPHC) est un tuple* $RdPHC = \langle S, P, T, A, \Sigma, St, SA, P_{por}, P_{soc}, PS \rangle$ *tel que :*

- S *est un ensemble fini de pages (subnets) où chaque page représente un RdP coloré.*
- P *est un ensemble fini de places.*
- T, *telle que* $P \cap T = \emptyset$ *et* $P \cup T \neq \emptyset$, *est un ensemble fini de transitions.*
- $A \subseteq (P \times (T \cup St)) \cup ((T \cup St) \times P)$, *est un ensemble fini d'arcs.*
- Σ *est un ensemble non vide de types, appelé aussi ensemble de couleurs.*
- St *est un ensemble fini de transitions de substitution.*
- $SA : St \rightarrow S$ *est une fonction qui associe à chaque transition de substitution une page (un subnet).*
- $P_{por} \subseteq P$ *est un ensemble fini de places portes.*
- $P_{soc} \subseteq P$ *est un ensemble fini de places sockets.*
- $PS : P_{por} \rightarrow P_{soc}$ *est une fonction qui associe les places portes aux places sockets.*

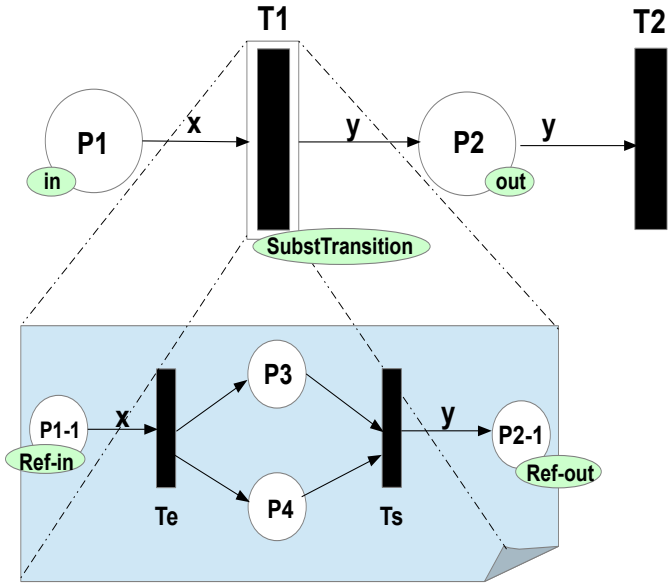


FIGURE 5.4 – Exemple de réseau de Petri hiérarchique.

PNML (Petri Net Markup Language) est un format d'échange normalisé pour les réseaux de Petri. Il est actuellement standardisé par ISO/IEC JTC1/SC7 WG 19 comme partie 2 du standard ISO/IEC 15909 [130]. L'objectif principal de PNML est d'aboutir à l'interopérabilité des outils basés sur les réseaux de Petri.
Dans cette approche, se basant sur les *RdPHCs*, au lieu de générer un réseau de Petri dans un format spécifique à un outil particulier, nous proposons de passer tout d'abord à une représentation PNML, c'est-à-dire un format intermédiaire indépendant de tout outil permettant la représentation des réseaux de Petri obtenus à partir des diagrammes d'activités. Ce choix est justifié par le fait que PNML est un standard indépendant de tout outil de manipulation des réseaux de Petri. De plus, des méta-modèles pour PNML ont été définis et implémentés avec l'outil PNML framework [131], ce qui nous permettra de réutiliser ces méta-modèles pour implémenter la transformation.
CPN Tools [132] est l'un des outils les plus utilisés pour l'analyse et la vérification des réseaux de Petri. A l'origine, il a été développé par le Groupe CPN à l'Université d'Aarhus de 2000 à 2010. Les principaux architectes de l'outil sont Kurt Jensen, Søren Christensen, Lars M.Kristensen, et Michael Westergaard. Depuis l'automne 2010, l'outil *CPN Tools* a été transféré au groupe AIS à l'université de technologie d'Eindhoven, aux Pays-Bas. *CPN Tools* est un outil permettant l'édition, l'analyse, la simulation et la vérification des réseaux de Petri. Il supporte les réseaux de Petri ordinaires, les réseaux de Petri colorés et les RdPHCs. Il dispose d'un simulateur, un outil d'analyse de l'espace d'états et intègre un modèle-checker pour la logique temporelle *ASK-CTL* [133], une variante de la logique CTL.

5.2.2.2 Transformation des diagrammes d'activités en RdPHCs

Plusieurs transformations des diagrammes d'activités vers des réseaux de Petri ont été proposées [134, 135, 136, 137, 138, 139]. Chacune de ces transformations est réalisée pour atteindre un objectif bien déterminé. L'objectif de notre transformation est de pouvoir vérifier des exigences sur les diagrammes d'activités via leur transformation **automatique** en une spécification formelle basée sur les RdPHCs. Dans le cadre de ce travail, en plus des constituants basiques des diagrammes d'activités, nous avons traité **la composition des activités** en considérant les actions d'appel d'activité. En particulier, nous pensons qu'il est essentiel de préserver la structure et la sémantique (comportement) des diagrammes d'activités dans les modèles *RdPHCs* résultants.

La transformation des diagrammes d'activités en *RdPHC* est réalisée en effectuant deux transformations. La première est une transformation M2M (Modèle à Modèle). Elle se base sur le langage ATL [140] pour transformer un digramme d'activités en un réseau de Petri en format PNML. La deuxième est une transformation M2T (Modèle à Texte). Elle génère un *RdPHC* en format *CPN Tools* à partir de la représentation PNML. Comme nous pouvons le constater, la transformation proposée suit une démarche MDA. La figure 5.5 montre les différentes étapes de cette transformation.

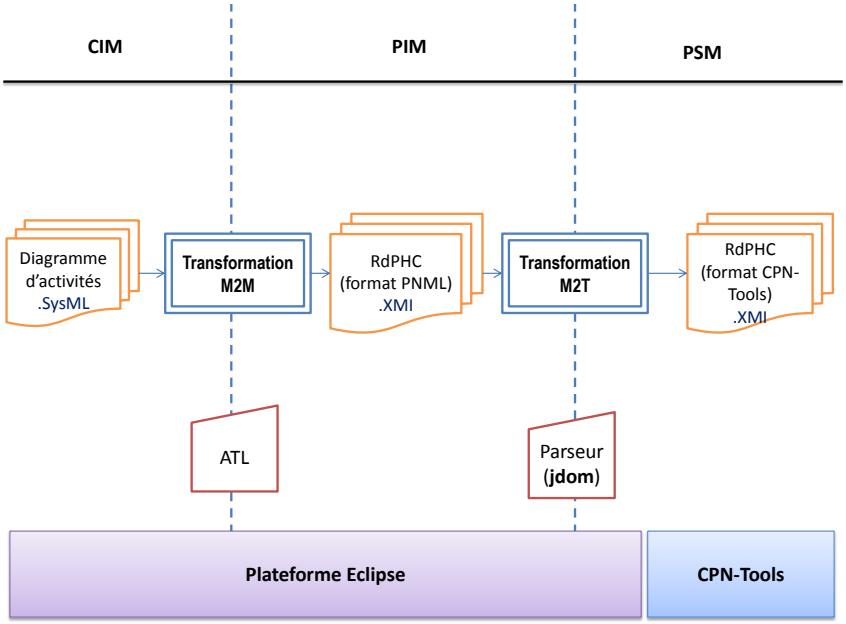


FIGURE 5.5 – Transformation des diagrammes d'activités en *RdPHC*.

Pour automatiser la transformation, nous avons utilisé ATL [140] comme outil de transformation de modèles. Le processus de transformation des diagrammes d'activités vers les RdPHC (en format PNML) à travers l'outil ATL est représenté dans la figure 5.6. Dans le but de préserver la structure composite des dia-

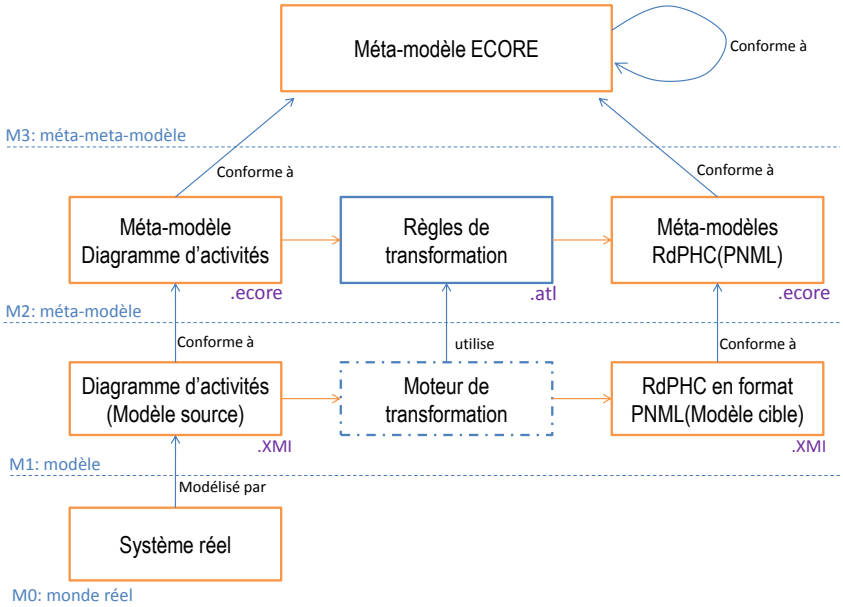


FIGURE 5.6 – Processus de transformation des diagrammes d'activités en RdPHC (format PNML).

grammes d'activités dans le modèle cible, nous proposons de transformer un diagramme d'activités incluant des actions d'appel d'activité en un RdPHC composé de plusieurs sous-réseaux (pages). La figure 5.7 représente la correspondance entre la structure du diagramme d'activités et la structure du RdPHC résultant

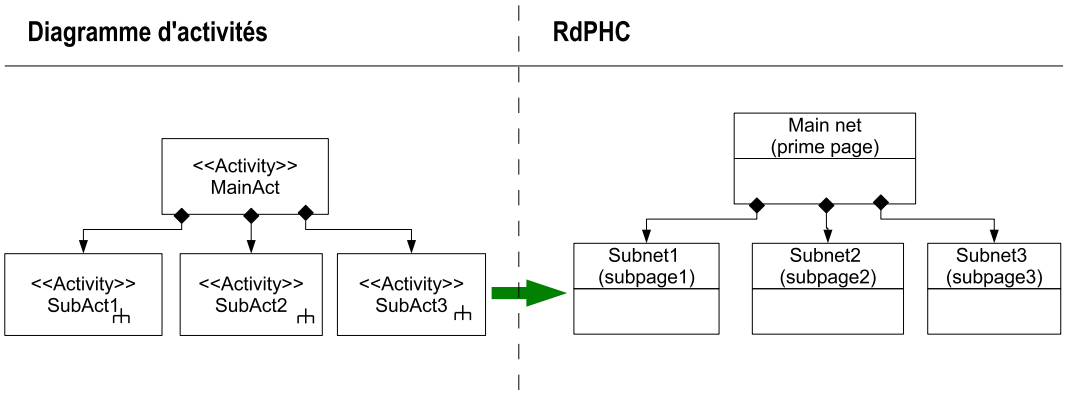


FIGURE 5.7 – Transformation de la structure.

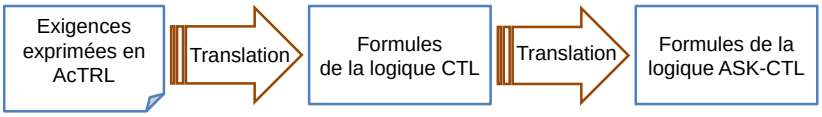


FIGURE 5.8 – Transforamation des expressions *AcTRL* vers *ASK-CTL*.

de la transformation. Chaque sous-activité sera transformée en un sous-réseau du RdPHC représenté dans une page séparée. La page primaire du RdPHC représente l'activité principale du diagramme d'activités.

5.2.2.3 Traduction des exigences en formules de logique temporelle

Le langage *AcTRL* permet une spécification de haut niveau des exigences pour permettre leur vérification par différentes approches selon les outils de vérification à utiliser.
Afin de pouvoir vérifier les exigences SysML exprimées en *AcTRL* sur le RdPHC obtenu à partir d'un diagramme d'activités, nous translatons ces exigences en langage de logique temporelle. Comme nous avons choisi d'utiliser l'outil *CPN Tools* pour la vérification, une traduction des exigences exprimées en *AcTRL* vers la logique temporelle *ASK-CTL* doit être effectuée.
La logique *ASK-CTL* est une variante de la logique CTL, supportée par l'outil *CPN Tools*. Le passage de *AcTRL* à *ASK-CTL* est réalisé en deux étapes. La première étape se base sur la librairie proposée en [141] pour passer de *AcTRL* à CTL. La deuxième étape se base sur des équivalences entre CTL et ASK-CTL pour passer de CTL à ASK-CTL. L'interprétation des expressions de prédicats *AcTRL* sur le RdPHC est définie selon la transformation des diagrammes d'activités en RdPHC.
La figure 5.8 montre les étapes de traduction des expressions *AcTRL* en formules de logique *ASK-CTL*.

5.2.3 Vérification des exigences SysML

Une fois les spécifications SysML (diagramme d'activités et exigences SysML) sont transformées en spécifications formelles (*RdPHC* en format *CPN Tools* et formules ASK-CTL), l'étape de vérification peut être débutée en utilisant l'outil *CPN Tools*. La figure 5.9 schématise la vérification des exigences SysML sur les diagrammes d'activités par l'outil *CPN Tools*.
La vérification des formules *ASK-CTL* par l'outil *CPN Tools* se fait en suivant les étapes suivantes :

- Génération de l'espace d'états,
- Invocation de la bibliothèque *ASK-CTL* via la commande suivante :

```
use ( ogpath ^ "ASKCTL/ASKCTLloader.sml")
```

- Ouverture du fichier contenant les formules *ASK-CTL*,
- Évaluation des formules *ASK-CTL*.

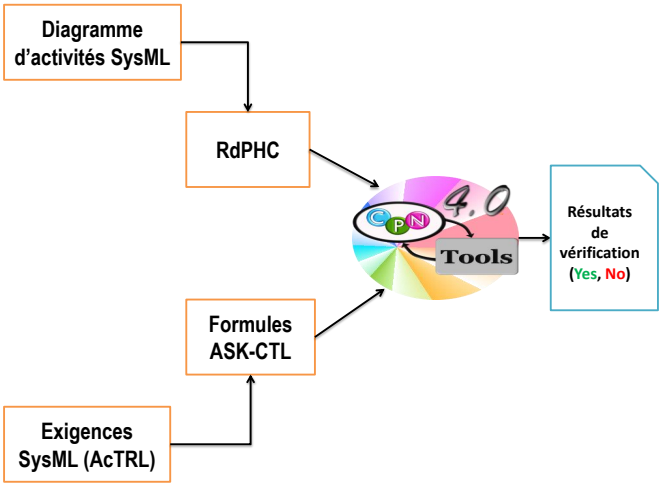
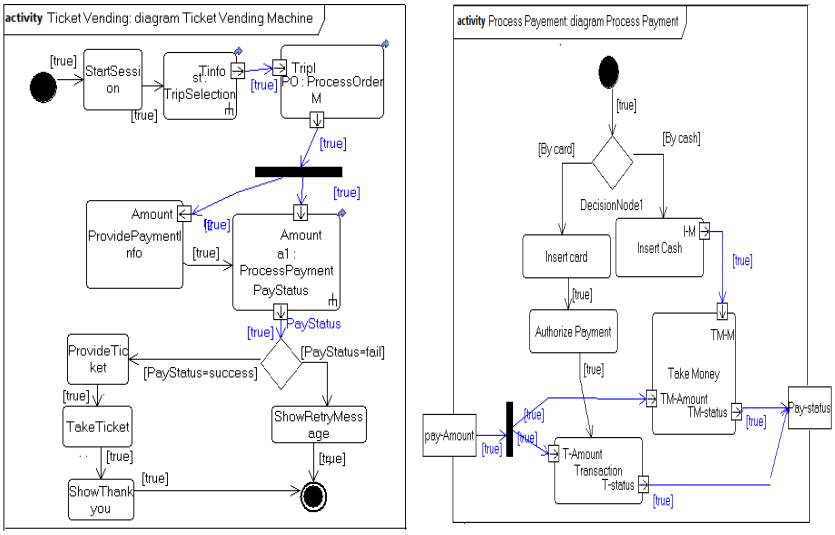


FIGURE 5.9 – Vérification des exigences avec l'outil *CPN Tools*.

5.2.4 Distributeur automatique de titres de transport

Afin d'illustrer l'approche utilisant les RdPHC, nous proposons de l'illuster par un distributeur automatique de titres de transport (Ticket Vending Machine : TVM). Une TVM est un système utilisé pour vendre des titres de transport, dans les stations de trains par exemple. Actionnée par un passager qui veut acheter un titre de transport, la TVM demande les informations concernant le voyage à effectuer (date, départ, destination, ...). Une fois ces informations introduites par le passager, la TVM calcule et affiche le montant total du titre de transport demandé. Puis, elle demande au passager de choisir une option de payement. Le payement peut se faire en cache ou par carte de crédit. Le passager doit spécifier son choix et procéder au payement. Si ce dernier se passe sans anomalie, la TVM délivre au passager son titre de transport.
Le comportement de la TVM est spécifié par le diagramme d'activités présenté dans la figure 5.10. Le diagramme décrit une activité composite (figure 5.10(a)), qui fait appel à d'autres sous-activités. A titre d'exemple, la sous-activité "payment process" est présentée dans la figure 5.10(b). Nous spécifions deux



((a)) Activité principale. ((b)) Sous-activité "payment process".

FIGURE 5.10 – Diagramme d'activités modélisant le comportement de la TVM.

exigences à vérifier sur le diagramme d'activités dans l'extrait du diagramme d'exigences présenté dans la figure 5.11 :

1. La première exigence REQ1 (Si l'opération de payement échoue, la TVM ne doit pas délivrer un titre de transport) est à vérifier sur l'activité principale.

2. La deuxième exigence REQ2 (Les passagers peuvent utiliser du cache ou des cartes pour le payement) est à vérifier sur la sous-activité "payment process".

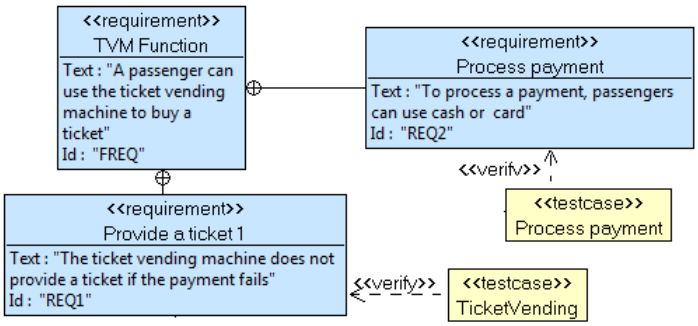


FIGURE 5.11 – Exemple de deux exigences de la TVM.

Dans la figure 5.12, nous présentons la formalisation des deux exigences REQ1 et REQ2. Les exigences DREQ1 (resp. DREQ2) formalisent et sont dérivées à partir de REQ1 (resp.REQ2). Elle sont exprimées en *AcTRL*.

La transformation du diagramme d'activités modélisant le comportement de la TVM en RdPHC est présentée dans la figure 5.13. la page principale du *RdPHC* résultant de la transformation est présentée. Nous remarquons que cette page contient trois transitions de substitution, chacune correspond à une action d'appel d'activité et fait référence à une page correspondante à une sous-activité.
Les exigences exprimées en *AcTRL* sont traduites en *ASK-CTL*. A titre d'exemple, le listing suivant représente la traduction de l'exigence DREQ1 présentée dans le diagramme de la figure 5.12.

```
fun testRunProvideTicket n = (Mark.ActTV'Running_provide_ticket
1 n=[1]);
val isProvideTicketActive=NF("ProvideTicket Running",
testRunProvideTicket);
fun testpayfail n = (Mark.ActTV'Pay_status 1 n=[0] );
val payfail= NF("Pay status", testpayfail );
val askCTL_DREQ1= INV(OR(NOT(payfail),
INV(NOT(isProvideTicketActivee))));
```

La vérification de cette exigence en utilisant l'outil *CPN Tools* est illustrée par la figure 5.14. La commande eval_node askCTL_DREQ1 est utilisée pour tester la satisfaction de l'exigence dans l'espace d'états généré à partir du modèle *RdPHC*.

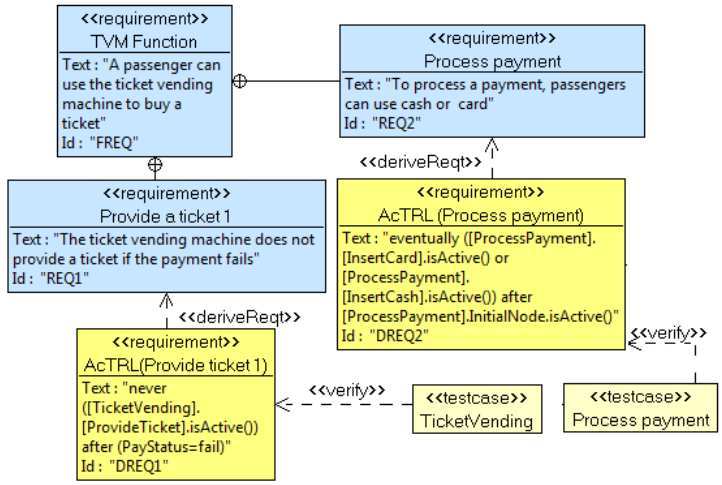


FIGURE 5.12 – Formalisation des exigences en *AcTRL*.

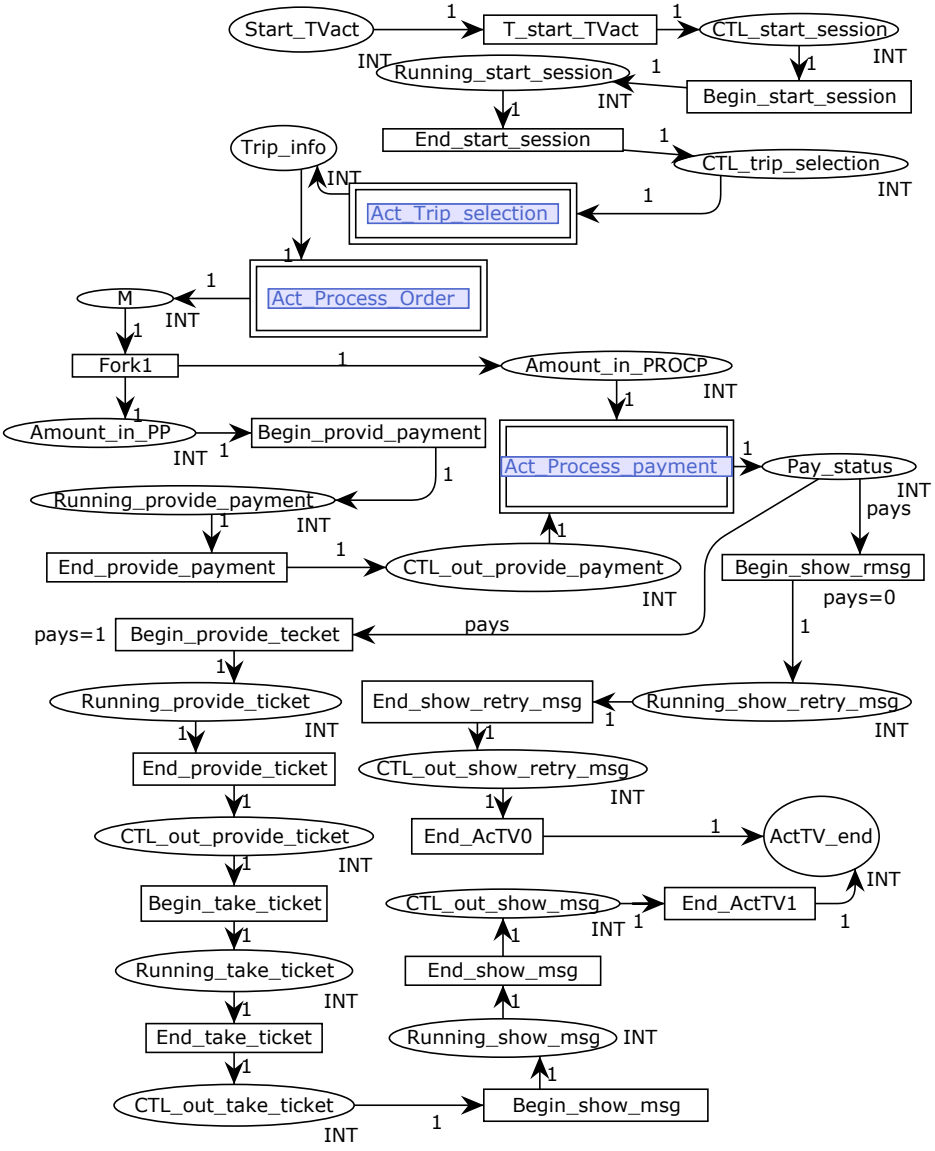


FIGURE 5.13 – RdPHC résultant de la transformation (page principale).

```
val testProvideTicketIsActive = fn : Node -> bool
val isProvideTicketActive = NF ("ProvideTicket Running",fn) : A
val testpayfail = fn : Node -> bool
val payfail = NF ("Pay status",fn) : A
val askCTL_DREQ1 =
  NOT
    (EXIST_UNTIL
      (TT,
       NOT
         (OR
           (NOT (NF ("Pay status",fn)),
            NOT
              (EXIST_UNTIL (TT,NOT (NOT (NF ("ProvideTicket Running",fn))))))))))
  : A
val it = true : bool
```

```
fun testProvideTicketIsActive n = (Mark.ActTV'Running_provide_ticket 1 n=[1]);
val isProvideTicketActive =NF("ProvideTicket Running", testProvideTicketIsActive);
fun testpayfail n = (Mark.ActTV'Pay_status 1 n=[0] );
val payfail= NF("Pay status", testpayfail );
val askCTL_DREQ1= INV(OR(NOT(payfail), INV(NOT(isProvideTicketActive))));
eval_node askCTL_DREQ1 1;
```

FIGURE 5.14 – Vérification d'une exigence (formule ASK-CTL).

5.3 Vérification avec découpage de modèles

Notre démarche de vérification se base sur le *model-checking*, elle hérite en conséquence des limites de cette technique de vérification. En particulier, le problème d'explosion combinatoire de l'espace d'états limite l'application de notre démarche sur des systèmes complexes de grandes taille.
Notre démarche pallie en partie ce problème en mettant à profit les avantages de la traçabilité des exigences. Cependant, ce problème persiste quand des diagrammes d'activité modélisant des comportements complexes sont concernés par la vérification. Pour cela, nous proposons un découpage de modèles.
Le découpage de modèle (**slicing**) est une technique qui permet de décomposer un grand modèle en modèles plus petits qui peuvent être vérifiés, analysés et/ou testés. Plusieurs travaux ont proposé des approches basées sur le découpage de modèles [142, 143, 144, 145]. Cependant, très peu proposent de découper des diagrammes d'activités dans l'objectif d'effectuer une vérification formelle.
Dans le but de réduire l'effet du problème d'explosion combinatoire de l'espace d'états pour vérifier des systèmes complexes, nous proposons d'exploiter les relations entre les éléments de modélisation SysML pour guider le processus de vérification. En particulier, les relations du diagramme d'exigences, à savoir, la relation "verify" qui permet de lier une exigence à un "test case" (une activité SysML dans notre cas) et la relation "satisfy" qui permet de lier une exigence avec un bloc. Aussi, nous exploitons la relation d'allocation implicite, représentée dans un diagramme d'activités avec des partitions (swimlanes), qui permet d'allouer des noeuds d'activités aux blocs SysML.
L'idée sur laquelle nous nous sommes basés est expliquée à travers la figure 5.15. L'objectif est de vérifier l'exigence "E" sur l'activité spécifiée par le test case "A". Le bloc qui satisfait l'exigence "E" est "B". Supposons que le diagramme d'activités est structuré en swimlanes (contient plusieurs partitions d'activités), le bloc "B" qui satisfait l'exigence "E" est parmi ces partitions d'activités. La question à poser est la suivante : *Est-il possible d'exploiter ces informations pour considérer qu'un sous-ensemble du diagramme d'activités "A" lors de la vérification de l'exigence "E" ?* Les différentes étapes de cette approche sont :

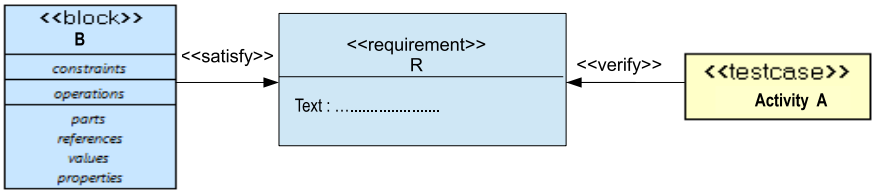


FIGURE 5.15 – Relations entre exigences, blocs et activités.

1. Transformation du diagramme d'activités en *RDPHC* en prenant en compte les partitions d'activités,
2. Détermination, pour chaque exigence fonctionnelle, le bloc et l'activité qui sont liés à celle-ci respectivement par les relations "satisfy" et "verify". Ces informations seront exploitées par la suite, pour diviser le *RdPHC* obtenu à partir du diagramme d'activités.
3. Réalisation du découpage au niveau du *RdPHC* afin d'éviter des transformations multiples qui doivent être réalisées dans le cas où le découpage est réalisé au niveau du diagramme d'activités.
4. Vérification des exigences dans la partie du *RdPHC* résultante de l'étape du découpage

Ces étapes sont illustrées dans la figure 5.16 et détaillées dans la suite de cette section. Pour prendre en compte les partitions d'activités (swimlanes) dans la phase de transformation des diagrammes d'activités vers les *RdPHC*, la règle suivante est appliquée : chaque partition est transformée en une transition de substitution et un sous-réseau *RdPHC* (une page *RdPHC* référencée par la transition de substitution).
Dans la figure 5.17, on illustre cette règle de transformation par un exemple. Elle présente la transformation d'un diagramme d'activités avec quatre (04) partitions d'activité : B1, B2, B3 et B4. Le RdPHC résultant de la transformation est constitué d'une page principale contenant quatre transitions de substitution : TB1, TB2, TB3 et TB4. Chacune de ces transitions représente une partition d'activité. Les éléments du diagramme d'activités appartenant à une même partition d'activité sont transformés en éléments RdPHC et regroupés dans une page référencée par la transition de substitution correspondante à cette partition d'activité. Enfin, chaque flux de contrôle ou d'objets liant deux partitions d'activité est transformé par une place de sortie à la transition de substitution dérivée de la première partition et une place d'entrée pour la transition de substitution dérivée de la deuxième partition.

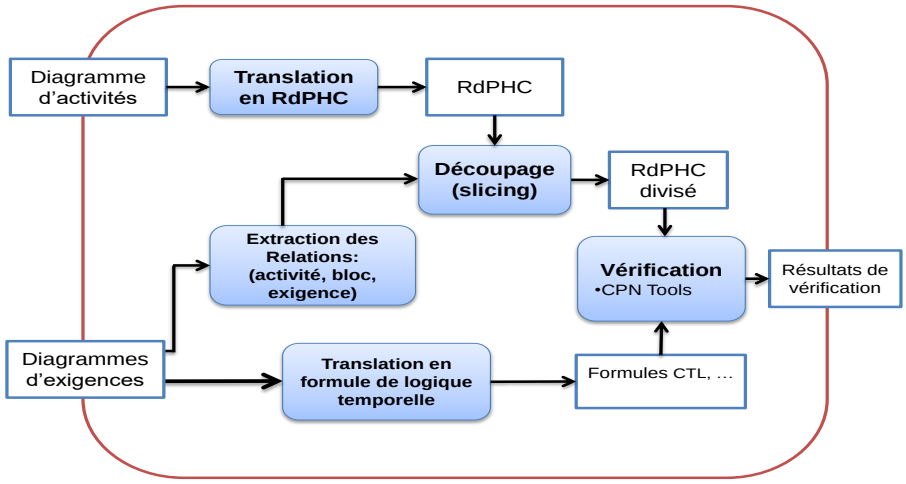


FIGURE 5.16 – Étapes de l'approche de vérification basée sur le découpage de modèles.

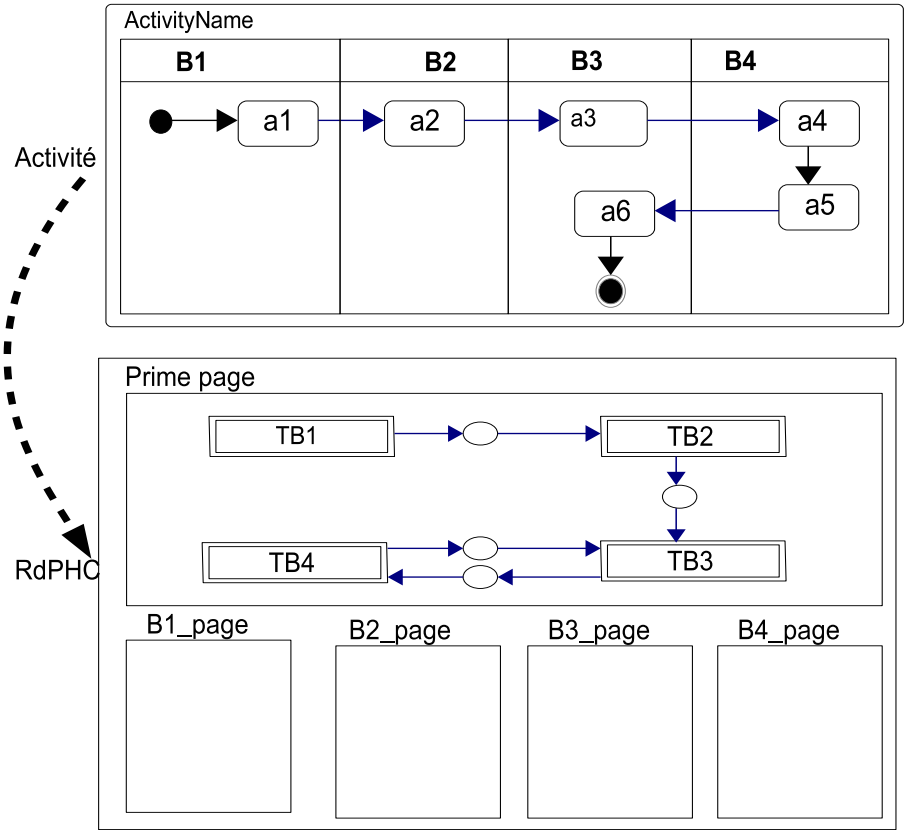


FIGURE 5.17 – Transformation des partitions d'activité.

5.3.1 Découpage (slicing)

Soit R une exigence à vérifier sur l'activité Act partitionnée en plusieurs partitions d'activité $Part = \{B_1, ...B_n\}$ et R est liée au bloc $B_i \in Part$ par la relation "satisfy". La division proposée repose sur la détermination des partitions d'activité nécessaires à la vérification d'une exigence. Si l'exigence R est satisfaite par B_i, B_i est parmi les partitions nécessaire à la vérification de R. Cependant, dans certains cas, la prise en compte seulement de B_i ne permet pas la vérification de R. Ceci est due à la dépendance entre les actions allouées à des partitions d'activité différentes. Une action a est dépendante d'une action b si l'action a demande en entrée un flux sortant de l'action b. Pour déterminer les partitions nécessaires à la vérification de l'exigence R, nous devons calculer toutes les partitions avec lesquels la partition B_i a une dépendance directe ou indirecte. Soit la fonction $dep(B_i, Act)$ permettant de calculer l'ensemble des dépendances de B_i, chaque partition d'activité $B_j \notin dep(B_i, Act)$ n'est pas nécessaire à la vérification de l'exigence R. Le découpage est fait sur le $RdPHC$ en supprimant pour chaque partition $B_j \notin dep(B_i, Act)$, sa transition de substitution correspondante. Ceci permet de réduire le modèle de vérification et ainsi pallier au problème de l'explosion combinatoire de l'espace d'états. Pour calculer l'ensemble des dépendances directs et indirects d'une partition d'activité, nous proposons de construire un graphe de dépendances entre partitions d'activité (Dependency Graph Between Activity Partitions).

Définition 5.3.1. *(Graphe de dépendances entre partition d'activités)*
Un graphe de dépendances entre partitions d'activité (Dependency Graph Between Activity Partitions) $DGBAP = \langle Part, A \rangle$*, où :*

> *DGBAP est un graphe orienté ayant les éléments de Part comme des sommets et les éléments de* $A \subset Part \times Part$ *comme des arêtes orientées (*$(B_i, B_j) \in A$ **si** B_j *a au moins un lien d'entrée à partir de* B_i*). Le calcul de* $dep(B_i, Act)$ *s'effectue alors par la fonction* $compute\_dep(B_i, DGBAP)$ *présentée par l'algorithme 1.*

Algorithm 1 $compute\_dep(B_i, DGBAP)$
Inputs : (B_i : activity partition, $DGBAP = \langle Part, A \rangle$)
Output : (DL : a list of activity partitions)

```
1: DL ← { } ;Stack P ; Push(P, Bi) ;
2: while (Not empty(P)) do
3:    Bj ← Pop(P) ; DL ← DL ∪ {Bj} ;
4:    for all (Bk, Bj) ∈ A) do
5:       if (Bk ∉ DL) then
6:          Push(P, Bk) ;
7:       end if
8:    end for
9: end while
```

5.4 Travaux connexes

Assurer la validité des systèmes critiques et complexes nécessite des approches de validation et de vérification qui aident à garantir la correction de leurs modèles de conception.
Dans cette section, nous allons présenter des travaux qui proposent la co-utilisation des langages semi-formels UML et SysML avec des méthodes formelles de vérification. En particulier, nous focalisons sur ceux qui concernent SysML et les méthodes de vérification basées sur les modèles.
Crée et standardisé par l'OMG pour l'ingénierie de logiciels, le langage UML (Unified Modelling language) est actuellement l'un des langages de modélisation le plus utilisé. Comme UML est un langage et n'est pas une méthodologie de modélisation ou de développement, il existe plusieurs travaux dédiés à l'intégration d'une phase de vérification dans le processus de conception d'un système avec UML. L'idée générale est de permettre aux concepteurs de bénéficier des avantages offertes par les méthodes de vérification pour déceler les erreurs et corriger la modélisation.
L'intégration des méthodes de vérification avec UML repose souvent sur la formalisation des diagrammes UML via leurs transformation en spécification formelle supportant une vérification automatique. L'objectif

étant d'aboutir à des outils permettant d'obtenir des spécifications formelles à partir des diagrammes UML sans intervention humaine, puis, le plutôt possible, réaliser une vérification automatique.
Nous pouvons classer les résultats obtenus dans ce cadre selon les langages formels utilisés pour la vérification.

Le langage Promela (outil Spin)

La plupart des travaux concernant la vérification des diagrammes UML ont été développés pour utiliser l'outil Spin. Les principales contributions dans ce domaine sont :

— **vUML** [146]. vUML est un outil qui permet une vérification automatique des modèles UML. Il focalise sur le diagramme d'états. Il est facile à utiliser et permet de générer automatiquement une spécification Promela aui peut être vérifiée automatiquement par Spin. La vérification automatique concerne des propriétés d'accessibilité.

— Latella, Majzik et Massink [147]. Ce travail concerne les diagrammes d'états d'UML encodés comme automates hiérarchiques à partir desquels des spécifications exprimées avec Promela peuvent être générées et vérifiées par Spin. La transformation n'est pas automatisée.

— HUGO [148]. Ce projet inclut un ensemble d'outils pour vérifier des diagrammes d'états et de collaborations. Le passage des diagrammes UML vers Spin est complètement automatique.

— Le travail en [149] propose une démarche pour la vérification des diagrammes d'états UML en utilisant l'outil Spin.

.

Le langage SMV

Des travaux existant visent à vérifier des diagrammes UML en utilisant l'outil SMV[150] et NuSMV [151]. Les plus importants sont :

— VeriUML [152] est un ensemble d'outils intégrés développés au niveau de l'université de Michigan permettant de vérifier des diagrammes d'états UML.

— TABU (Tool for the Active Behaviour of UML) [153] est un outil permettant la vérification des diagrammes d'états et d'activité UML en se basant sur l'outil SMV. L'outil se base sur XMI et permet de transformer automatiquement les diagrammes UML en langage SMV.

Réseaux de Petri

Plusieurs travaux concernent la co-utilisation d'UML avec des réseaux de Petri. Nous pouvons citer :

— UML-PNO [154, 155]. L'approche UML/PNO (Unified Modelling Language with Petri Net Objects) a été élaborée afin de proposer un support sémantique précis aux diagrammes dynamiques d'UML, de manière à mettre en oeuvre efficacement ce langage pour l'utiliser dans des spécifications de systèmes temps-réel.
L'approche UML/PNO propose d'enrichir la description semi-formelle UML du système par une modélisation formelle basée sur les réseaux de Petri. Elle consiste à décrire certains aspects comportementaux des objets (tels que les spécifications ou les contraintes temporelles) à l'aide des réseaux de Petri. L'approche de validation propose des traductions semi-automatiques des diagrammes UML en réseaux de Petri. Cette méthode permet ainsi d'obtenir des résultats intéressants en terme de vérification, mais reste complexe à mettre en oeuvre. De plus, elle ne permet pas par exemple de prendre en compte directement les problèmes de synchronisation d'horloge ou l'indéterminisme dû à des conflits.

— Les travaux de T.Bouabana-Tebibel et M. Belmesk [156, 157], les auteurs proposent de transformer les diagrammes UML en réseaux de Petri et leurs extensions pour réaliser la vérification.

La formalisation de la sémantique des diagrammes d'états a été adressée en [158, 159, 160]. Les diagrammes d'activité ont été traités en [139, 161, 162, 135, 163] et les travaux [164, 165, 166, 167] concernent les diagrammes de séquence.
Comme SysML est un profil d'UML, dédié à l'ingénierie systèmes. Plusieurs travaux, dont le but est d'étudier ou de définir de telles approches pour des systèmes à spécifier en SysML, ont été proposés. Dans [168], les auteurs proposent de dériver à partir de chaque diagramme SysML un modèle formel décrivant ses caractéristiques. Dans ce travail, les exigences du système à vérifier sont décrites par des formules de logique

temporelle exprimées sur le modèle formel dérivé. Cette tâche n'est pas facile pour des concepteurs (praticiens SysML) non familiarisés avec les langages de logiques temporelles. Linhares et al.[169], proposent un processus de développement de systèmes basés sur SysML et les réseaux de Petri. Les auteurs suggèrent d'utiliser le diagramme de block (BDD) pour définir la structure du système et les réseaux de Petri pour décrire ses aspects comportementaux. L'outil Tina a été utilisé pour la vérification des propriétés. Ce travail impose aussi l'expression des exigences en logique temporelle. l'idée d'utiliser conjointement SysML avec des modèles formels est à la base du formalisme HiLeS (High Level System Designer). Ce formalisme prend en compte les systèmes hybrides en permettant la modélisation conjointe d'une structure de contrôle sous la forme d'un réseau de Petri, et d'un ensemble de modèles VHDL-AMS associés aux composants élémentaires du système. Ce type de modèle combine les intérêts d'un modèle formel et d'un modèle analytique, au prix d'une modélisation parfois complexe. Ce formalisme a été notamment employé dans le domaine des systèmes aéronautiques et des microsystèmes [170].
Dans le travail de [171], les classes stéréotypées « test cases » on été assimilées à des dispositifs de vérification, alors que la spécification OMG de SysML, un test case est défini comme un modèle de comportement décrit par un diagramme d'activité, un diagramme de séquence ou un diagramme d'états. Dans [119], Knorreck et al. proposent TEPE, un éditeur graphique pour l'expression des propriétés temporelles en se basant sur le diagramme paramétrique de SysML. Ainsi, les propriétés ont été décrites sous forme de prédicats quantifié par des opérateurs temporels. L' outil UPPAAL [42] est utilisé pour la vérification. Ce travail se base sur la construction d'observateurs et adresse la vérification des diagrammes d'états SysML. Les approches basées sur la construction d'observateurs posent le problème de placement des observateurs dans le modèle à vérifier, sans perturber le fonctionnement normale du système.
Des travaux académiques traitant de SysML tels que ceux de R. Cloutier [172], défendent le principe selon lequel une utilisation rationnelle de SysML, appuyée sur une méthodologie et adaptée au contexte, aux contraintes des entreprises et des projets, favorise une bonne efficacité de la modélisation.
De nombreux autres travaux récents se sont intéressés à ces problématiques de vérification et de validation formalisées sur UML et les langages dérivés, citons en particulier [173, 174]. Le premier article décrit la méthode employée pour valider des modèles définis sur un dialecte UML permettant d'exprimer des contraintes temporelles grâce à un outil développé au laboratoire Verimag et s'appuyant sur des automates temporisés. Dans le second article, les auteurs proposent une démarche basée sur des méthodes formelles comme le *model-checking* pour la vérification et la validation de modèles d'ingénierie système ou logiciels, décrits au moyen d'UML 2.0 ou de SysML. Par ailleurs, il convient de noter que la version préliminaire de la spécification SysML1.0 mentionne que les problématiques de vérification et de validation (non encore traitées dans la version 0.9) sont partiellement gérées et qu'elles devraient être totalement implémentées à partir de la version 2.0. Certains travaux de recherche ont porté sur la spécialisation de SysML, ou son utilisation dans un contexte particulier. B Fontan [175] a par exemple proposé une méthode permettant d'améliorer la prise en compte des exigences temps réels, de leur formulation, jusqu'à leur vérification sur des modèles exécutables au moyen d'observateurs générés automatiquement à partir des exigences initiales. Cette approche associé aux modèles d'exigences SysML, un profil UML préalablement défini et dédié au domaine des systèmes temps réel.

5.5 Bilan

Ce travail a donné lieu :

1. **Projet** :
   - 2011-2014 Programme National de recherche (PNR Algérie) : *MoSPEV (Modélisation de Haut niveau, Spécification et Vérification des systèmes)*. Coopération avec l'USTHB Alger (Algérie).
   - 2013-2016 : Projet PHC Tassili *SYSVAP (Spécifier en vue de Vérifier et d'évaluer les performance des systèmes complexes*, Coopération avec l'USTHB Alger (Algérie). Ma contribution concerne la définition d'une logique temporelle des exigences s'appuyant sur le diagramme d'activités SysML, et la transformation des diagrammes d'activités vers les réseaux de Petri (RdPs).
2. **Thèse de doctorat USTHB - Alger** [176]. Voir tableau 5.1

Année	Candidat	Titre
2017	M. Rahim	*Vérification distribuée.*

TABLE 5.1 – Thèse démarche de de validation par vérification formelle

3. **DEA-Master 2 recherche**. Voir tableau 5.2.

Année	Candidat	Titre
2016	O. Bouyahiaoui L. Haddad	*Vérification d'exigences SysML.* *Transformation du diagramme d'activités SysML vers les réseaux de Pétri (RdP)*

TABLE 5.2 – DEA-Master 2 recherche chapitre 5

4. **Communications suivantes :** . [61, 62, 63, 64, 65, 66]

6

Démarche de spécification et de validation par Simulation

Plan du chapitre

6.1 Introduction

Dans ce chapitre, nous présentons une méthodologie pour la conception et la validation de systèmes complexes et critiques. Par systèmes complexes, nous entendons des systèmes intégrant des composants hétérogènes de plusieurs domaines (électronique, mécanique, hydraulique,...) parmi lesquels certains pourraient intégrer des logiciels. La complexité d'un système peut être évaluée par le nombre de ses composants et leurs interactions possibles. L'hétérogénéité des composants du système est donc un facteur majeur de complexité. Les composants des systèmes complexes peuvent être caractérisés par des contraintes qualitatives et/ou quantitatives. Les contraintes qualitatives (sûreté, vivacité, ..) peuvent être formalisées sous formes de propriétés exprimées en langages de logiques temporelles. Ces contraintes définissent des exigences fonctionnelles du système. Les contraintes quantitatives (consommation d'énergie, performance, ...) peuvent être formalisées par des équations mathématiques pour décrire des phénomènes physiques régissant le fonctionnement du système. Ces contraintes constituent les exigences non fonctionnelles du système. La vérification formelle n'est pas adaptée à la validation des exigences non fonctionnelles. En effet, ces exigences non fonctionnelles modélisées, par exemple, par des équations aux dérivées partielles ne peuvent pas être validées par des outils de vérification formelle.
La simulation informatique désigne l'exécution d'un logiciel informatique sur un ordinateur ou réseau en vue de simuler un phénomène physique réel et complexe (par exemple : déplacement d'un objet sur une surface, résistance d'une plate-forme pétrolière à la houle, fatigue d'un matériau sous sollicitation vibratoire, usure d'un roulement à billes). Les simulations numériques scientifiques reposent sur des équations mathématiques bien élaborées. L'utilisation de la simulation permet de surmonter ce problème, en explorant les phénomènes physiques à l'aide d'environnements informatiques de simulation afin d'étudier les différents comportements attendus. Cela évite des coûts très élevés et des risques importants qui peuvent être engendrés par des expérimentations sur des systèmes réels (ex : essais de véhicules ou d'avions), surtout dans les domaines aéronautique, spatial, nucléaire, ... La simulation peut donc offrir un aperçu sur le développement d'un système trop complexe en modélisant ses contraintes physiques par des formules mathématiques que l'on peut analyser.
SysML est adapté à modéliser des systèmes complexes. Il prend en compte non seulement les aspects logiciels mais aussi les aspects matériels. De plus, le diagramme paramétrique de SysML permet de représenter graphiquement les paramètres des contraintes modélisées par des équations mathématiques. Ce diagramme peut servir comme un point d'entrée dans les environnements de simulation. Notre démarche de validation par simulation consiste en l'utilisation de SysML pour spécifier ces systèmes et des techniques d'IDM en vue de transformer les modèles SysML vers des langages de simulation. Dans nos travaux, deux langages de simulation (VHDL-AMS et Modelica) ont été considérés. Ces choix ont été motivés par les domaines d'application étudiés dans le cadre des projets de recherche et de la thèse de J.M Gauthier[177]. Des travaux sur l'utilisation de SysML et SystemC ont débuté, ils ont fait l'objet de la thèse de M. Abbas [56].

Dans la suite de ce chapitre, nous présenterons les démarches de validation d'exigences non fonctionnelles par l'utilisation conjointe des ces langages de simulation avec SysML. Nous avons choisi de présenter brièvement les langages VHDL-AMS et Modelica, les traductions de SysML vers ces langages. Par la suite, nous présenterons deux projets (Projet région *SyVAD* et le projet ANR *Smart Surface*) validant la démarche de modélisation et de validation s'appuyant sur l'utilisation conjointe de SysML et VHDL-AMS. Pour SysML et Modelica, nous présenterons deux études de cas, la première concerne les réseaux de capteurs sans fils et la deuxième, les systèmes à base de MEMS [1].

6.2 Utilisation conjointe SysML et VHDL-AMS

La modélisation permet à un concepteur de gérer la complexité et mettre les différents collaborateurs sur une vue commune du système, en facilitant la conception et en permettant d'avoir un système sûr répondant aux exigences définies. La modélisation de ces systèmes représente un challenge pour l'industrie et pour l'académie dont l'écart entre le logiciel et le matériel devient de plus en plus étroit, parmi ces systèmes on peut citer les SOCs[178] (Systems on chip), les systèmes embarqués, dont le logiciel et le matériel interagissent de façon continue et nécessitent une collaboration entre l'équipe de développement logiciel,

1. MicroElectroMechanical Systems

l'équipe de développement matériel et le client afin de définir les besoins et les exigences et éliminer les ambiguïtés potentielles dès la phase de spécification pour mieux répondre aux exigences.
Les méthodes de l'Ingénierie Système (IS) [179] reposent sur des approches de modélisation, vérification et de simulation pour valider les exigences, vérifier ou évaluer le système pour avoir une meilleure qualité et d'assurer une fiabilité des systèmes critiques. Pour la modélisation des systèmes complexes, les langages de modélisation graphique se présentent comme une solution puissante et appropriée de modélisation.
UML permet de décrire les systèmes selon différents points de vue en offrant un ensemble de diagrammes pour la spécification des besoins, de la structure et du comportement en facilitant la compréhension et la communication entre les partenaires de la conception avant la réalisation du système. Bien que UML permet par son caractère à usage général d'adresser de nombreux besoins pour l'IS, il est nécessaire d'adapter ce langage de modélisation par la définition de profils UML [180]. Plusieurs projets ont été menés pour réaliser des profils UML sur des domaines spécifiques de l'IS, nous pouvons citer MARTE [7] qui est un profil UML pour des systèmes embarqués en temps-réel et le profil UML-SOC [181] pour les systèmes sur puce.
Le langage SysML est conçu pour fournir un vocabulaire simple et efficace adapté à l'ingénierie des systèmes. Avec SysML comme une solution de modélisation, les systèmes doivent être concrétisés et les fonctionnalités décrites dans la spécification sont implémentées dans des environnements de développement différents qui prennent la modélisation comme un premier support et référence afin de les traduire en langages de développement qui permettent l'exécution et la validation des systèmes pour assurer la cohérence entre la modélisation et l'implémentation.
Pour le logiciel, on peut citer *Java* ou *C++* comme les langages cibles les plus utilisés et pour le matériel les langages utilisés sont les langages de description de matériel appelé HDL (Hardware Description Language). Il s'agit de langages informatiques permettant une description formelle des circuits électroniques. Ils peuvent décrire les opérations effectuées par le circuit (fonctionnalités), mais aussi son organisation (structure), et tester le circuit et le vérifier par le biais de la simulation. Parmi ces langages le VHDL (VHSIC Hardware Description Language) [182], SystemC [43] et Verilog [183] qui sont parmi les HDL les plus utilisés.
Les besoins en simulation s'élargissant aux systèmes exigeant l'inclusion de différents domaines, par exemple les domaines analogique et le numérique. Ces domaines doivent interagir pour assurer le fonctionnement du système. Ceci représente une limite pour le VHDL, pour remédier à ce problème le VHDL-AMS a été proposé comme une solution. Il permet de modéliser des objets abstraits manipulant des signaux quantifiés à des moments discrets dans le temps et ajoute les instructions simultanées aux instructions concurrentes et séquentielles du VHDL, permettant de manipuler des valeurs à temps continu transportées par les objets " QUANTITY ".

6.2.1 Le langage VHDL-AMS

Le langage VHDL-AMS (VHSIC-Hardware Description Language - Analog and Mixed Systems) [184, 182] a été développé comme une extension du langage VHDL (IEEE 1076-1993). Il permet la modélisation et la simulation de circuits et de systèmes logiques, analogiques et mixtes. Il est basé sur le langage VHDL et la théorie des équations différentielles algébriques (EDAs). VHDL-AMS hérite de toutes les caractéristiques de VHDL, en particulier, la modularité, l'extensibilité, le typage fort des données, les possibilités de descriptions structurelle et fonctionnelle hiérarchiques et un ensemble d'instructions séquentielles et concurrentes. Le VHDL-AMS profite des capacités du langage VHDL à modéliser des objets abstraits manipulant des signaux quantifiés à des moments discrets dans le temps. Il ajoute les instructions simultanées permettant de manipuler des valeurs à temps continu. La grande force de ce langage est de permettre la simulation mixte en autorisant aussi bien les modélisations à temps continu (analogiques) qu'à événements discrets (logiques) ou mélangeant les deux. La description des systèmes analogiques, ou continus repose sur la théorie des EDAs. VHDL-AMS offre une notation pour les EDAs avec le temps comme variable indépendante. Soit des EDAs de la forme :

$$F(x, \frac{dx}{dt}, t) = 0 \tag{6.1}$$

avec
F : un vecteur d'expressions,

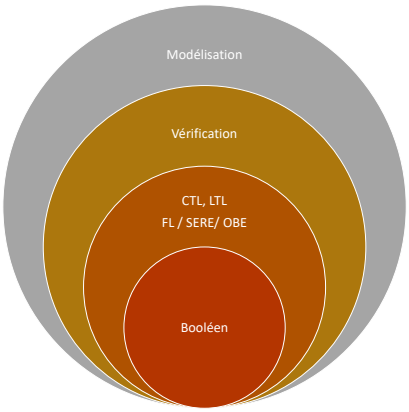


FIGURE 6.1 – Couches du langage PSL

x : un vecteur d'inconnues,
$\frac{dx}{dt}$: un vecteur de dérivation des inconnues par rapport au temps,
t : le temps.
Dans VHDL-AMS, la notation des EDAs est supportée par une nouvelle classe d'instructions appelées *instructions simultanées*. Les inconnues dans les équations EDAs sont représentées par une classe d'objets appelée *quantités*. Une *quantité* représente une fonction du temps à valeurs réelles. Le langage offre aussi une notation pour les dérivées, les intégrales et d'autres fonctions de transfert.
Un modèle VHDL-AMS a la structure suivante : l'entité spécifie l'interface du modèle(paramètres, ports) et l'architecture spécifie le contenu du modèle sous forme de comportement ou de structure. Plusieurs architectures peuvent être définies pour la même entité. Le comportement peut être dirigé par des événements discret ou continu. la structure est principalement une interconnexion de composants qui sont associés individuellement au moyen de configurations à des modèles VHDL-AMS pour la simulation.

6.2.1.1 Le langage PSL

Ces dernières années, la vérification à base d'assertions (ABV) [185] a fait son apparition dans les domaines académiques et industriels. Une assertion est une description concise d'un comportement complexe que le circuit doit satisfaire. Cette approche, qui consiste à supporter la vérification à l'aide d'assertions, a prouvé son efficacité, et la plupart des industriels ont aujourd'hui adopté des flots de conception basés sur l'ABV. Dans ce travail, nous considérons les assertions comme des propriétés logico-temporelles basées sur la CTL, la LTL, etc. Nous avons utilisé le langage PSL [186] (Property Specification Language) basée sur la LTL pour spécifier des exigences et les vérifier dans le simulateur VHDL-AMS. PSL est née sous le nom de *Sugar* [187] dans les laboratoires d'IBM au début des années 2001. Il a été standardisé par Accellera [2] en 2003, puis par IEEE en 2004. La dernière révision de la norme 2005 permet d'écrire des propriétés logico-temporelles complexes et adresse trois domaines liés à la vérification de circuits (cf figure 6.1) :

— spécification : fournit un langage structuré, doté d'une sémantique formellement définie, supprimant ainsi les ambiguïtés liées aux langages naturels.

— vérification formelle : model-checking.

— vérification classique : analyse des propriétés en parallèle de la simulation fonctionnelle HDL.

6.2.2 Démarche de modélisation et de validation

Notre objectif est donc de définir une approche méthodologique outillée (cf. figure 6.2) basée sur le formalisme SysML permettant :

2. http ://www.accellera.org/

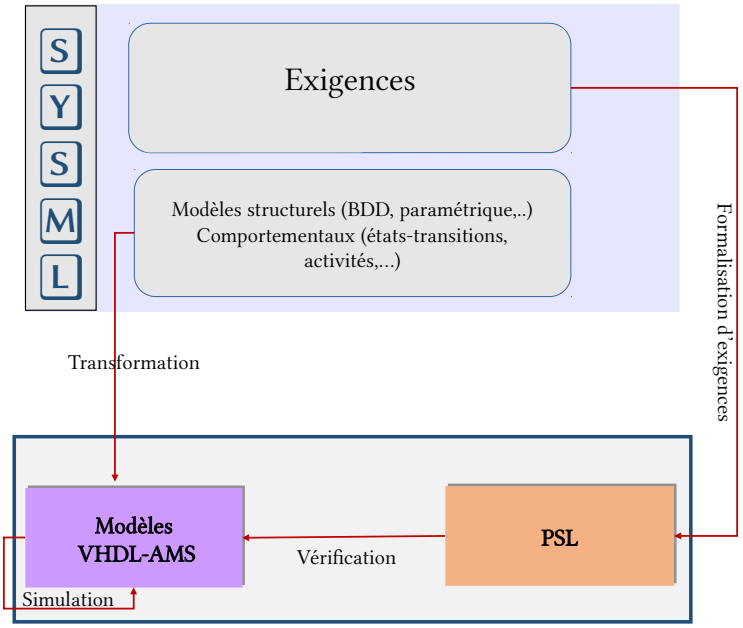


FIGURE 6.2 – Méthodologie de spécification et de validation (SysML et VHDL-AMS

1. L'identification, la formalisation et la structuration d'exigences relatives à un système au niveau du modèle SysML pour en assurer la traçabilité. L'idée est de pouvoir exprimer des exigences avec le langage PSL. Ainsi la transformation des exigences enrichies avec les éléments du modèle SysML donneront lieu à des propriétés PSL au niveau VHDL-AMS.
2. Définir des règles de transformations de modèles SysML en modèles VHDL-AMS. Ainsi, on prendra en compte l'ensemble des informations présentes dans les différents diagrammes SysML pour engendrer un modèle VHDL-AMS complet qui puisse être simulé directement. Cela prend en compte la description hiérarchique sous forme de blocs (composants matériel et/ou logiciel) du futur système, la modélisation du comportement pour chaque bloc et la description des contraintes physiques sous forme d'un diagramme paramétrique.
3. Définir une méthode de génération de cas de tests à partir du modèle SysML et des propriétés extraites des exigences pour servir d'entrée à la simulation du système sous sa forme VHDL-AMS.

La première étape de ce travail concerne la transformation des modèles SysML vers le langage VHDL-AMS. Plusieurs travaux sur la transformation de modèles décrits avec des langages graphiques basés sur UML [188, 189, 190, 191] vers VHDL ont été réalisés. Dans [188], les auteurs ont présenté un algorithme de transformation du diagramme de classes vers VHDL-AMS. Les travaux décrits dans [189], présentent une transformation des diagrammes de classes, d'objets et d'états transitions UML vers VHDL. Les travaux décrits dans [190, 191], présentent le passage des machines à états vers VHDL. Ces dernières années, nous trouvons des travaux sur le passage de SysML vers VHDL-AMS [192, 193], pour des besoins de simulation, et particulièrement la transformation des diagrammes de blocs vers VHDL-AMS. Ces transformations produisent généralement un squelette de code VHDL-AMS, le concepteur est contraint de compléter ce code pour pouvoir l'exploiter, ceci nécessite d'acquérir les compétences dans ce langage. Dans ce cas, le concepteur peut choisir de décrire directement son architecture avec VHDL-AMS sans passer par SysML. Nous proposons d'offrir la possibilité au concepteur de rester complètement au niveau SysML sans devoir intervenir sur le code VHDL-AMS [175]. Notre objectif est de générer le code complet VHDL- AMS correspondant à un modèle SysML. En plus de cette génération, nous comptons extraire, à partir du diagrammes d'exigences, des propriétés ré-écrites en PSL. Elles seront intégrées dans le code VHDL-AMS en vue de la vérification formelle statique et/ou dynamique des propriétés temporelles sur les circuits. Ceci permet de pouvoir vérifier les propriétés sur un modèle qui intègre les aspects matériels et logiciels. Ceci constituera la deuxième étape de ce travail.
L'utilisation de VHDL-AMS dans le cadre des MEMS a semblé prometteuse [194, 195, 196]. De plus, l'utilisation de ce langage pour la modélisation des systèmes microfluidiques est adaptée comme le montre les travaux de [197] et de façon plus générale pour les systèmes décrits par des équations différentielles [198].

6.2.3 Transformation de modèles SysML vers VHDL-AMS

La transformation de modèles est une activité prépondérante dans le processus de développement. Le principe de transformation de modèles a attiré beaucoup d'attention en devenant un sujet de recherche pour l'académie et l'industrie. Plusieurs termes ont été adoptés afin de définir le processus de transformation qui se décrit généralement par la transformation d'un modèle à certain niveau d'abstraction conforme à un méta-modèle vers un modèle cible à un autre niveau d'abstraction conforme à son méta-modèle dont le passage est décrit par des règles de transformation, celles-ci sont exécutées sur le modèle source afin de générer le modèle cible. Parmi les notions les plus utilisées, *M2M*(Model to Model) qui signifie la transformation d'un modèle en un autre modèle et *M2T*(Model to Text) qui signifie généralement la génération de code à partir de modèle. Pour mettre en œuvre ces différents concepts plusieurs langages ou méthodes permettent l'automatisation de la transformation. Parmi ces langages on peut citer *ATL* [199, 200], *QVT* [201], *kermeta* [202] pour la transformation *M2M* et *Acceleo* [203], pour la transformation *M2T*.
La différence au niveau d'abstraction entre le modèle (haut niveau d'abstraction) SysML et l'implémentation (bas niveau d'abstraction) VHDL-AMS rend le passage entre le modèle SysML et VHDL-AMS de plus en plus complexe. Pour pallier ce problème, nous avons défini un modèle intermédiaire appelé SysML4VHDL-AMS, qui permet de préciser SysML avec des mots clés de VHDL-AMS comme par exemple les déclarations de variables et l'intégration des bibliothèques. La figure 6.3 montre les étapes de transformation d'un modèle SysML vers un modèle VHDL-AMS.

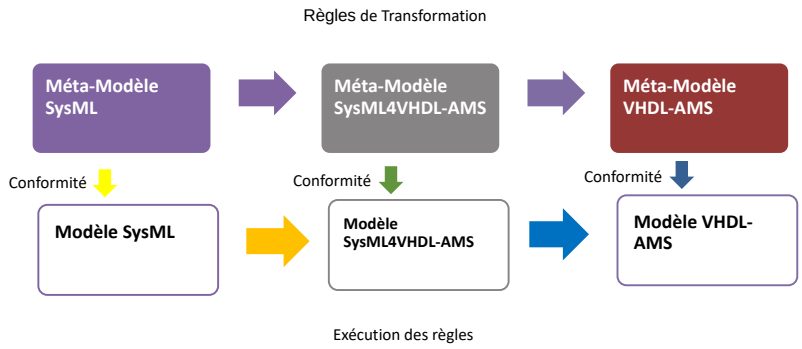


FIGURE 6.3 – Transformation de modèles SysML vers VHDL-AMS

Règles de transformation

Nous donnerons dans ce qui suit quelques règles de correspondances entre les éléments d'un sous ensemble SysML et ceux de VHDL-AMS. Cette synthèse est issue des différents travaux au sein de notre laboratoire [204, 47, 48]. Le pouvoir d'expression SysML pris en compte est le suivant :

- Diagramme de description de blocs BDD,
- Diagramme interne de blocs,
- Diagramme de bloc de contraintes ecrites en VHDL-AMS,
- Diagramme paramétriques,
- Diagramme d'activités

Le code VHDL-AMS généré est simulable dans un environnement de simulation tel que Smash [205].

Diagramme de blocs

Ce sont des unités modulaires dans la description d'un système. Le bloc SysML constitue la brique de base pour la modélisation d'un système. Il inclut les deux aspects structurels (statique, les propriétés) et comportementales (dynamique, les opérations) d'un système afin de représenter l'état du système et son

comportement. L'entité de conception (design entity) est l'abstraction de base en VHDL-AMS. Elle représente une portion d'un système matériel possédant une interface entrée-sortie et une fonction bien définies. Une entité de conception est constituée d'une déclaration d'entité et d'un corps d'architecture correspondant. Une entité de conception peut représenter un système matériel à plusieurs niveaux de complexité : un système entier, un sous-système, une carte, un circuit intégré,... D'après les travaux de passage des modèles SysML vers VHDL-AMS [188, 189, 190, 191, 188, 192, 193], nous pensons que le bloc SysML correspond à L'entité de conception de VHDL-AMS. Nous transformons donc un bloc SysML en ENTITY VHDL-AMS. Le nom de cette ENTITY sera le même que celui du bloc.

Diagramme de bloc interne

Décrit la structure interne d'un bloc sous forme de parts, de ports et de connectors. Il capture la structure interne d'un bloc y compris les points d'interaction à d'autres parties du système et montre la configuration de parts qui exécutent le comportement du bloc. Les parts sont traduites en components de VHDL-AMS.La figure 6.4 montre la correspondance entre un bloc et son IBD avec le code VHDL-AMS correspondant.

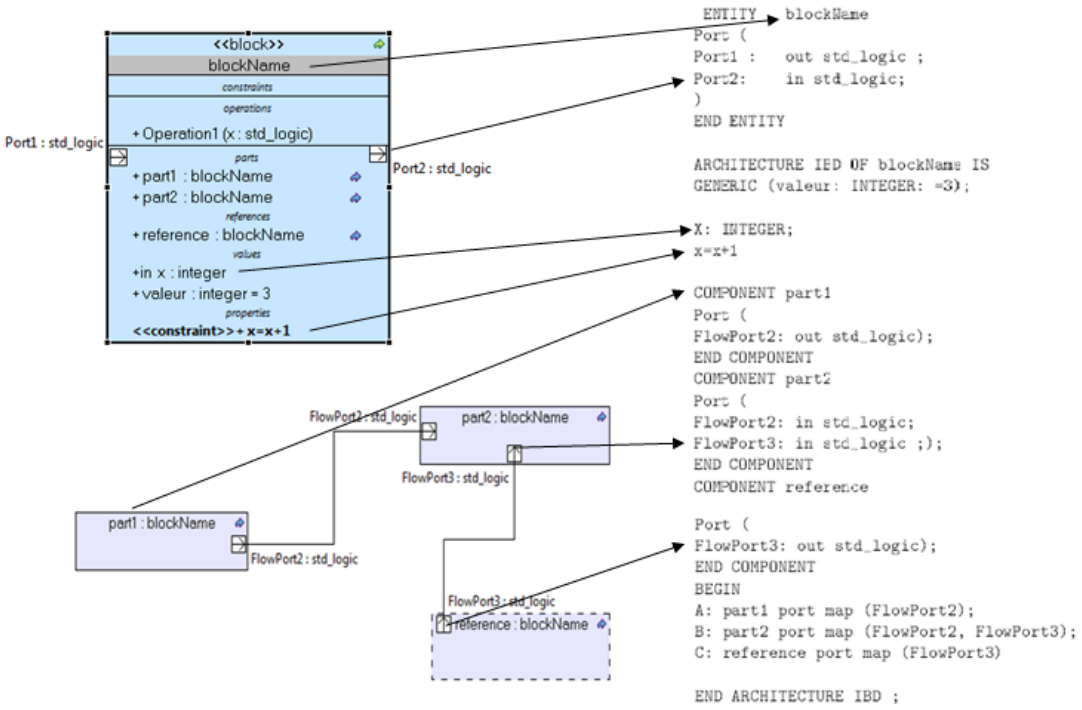


FIGURE 6.4 – Régles de transformation : BDD/IBD et VHDL-AMS

Régles de transformation du diagramme paramétrique

Bloc de contraintes

Les blocs de contraintes fournissent un mécanisme permettant d'intégrer des analyses d'ingénierie telles que les modèles de performance et de fiabilité avec d'autres modèles SysML. Les blocs de contraintes peuvent être utilisés pour spécifier un réseau de contraintes qui représentent des expressions mathématiques qui contraignent les propriétés physiques d'un système, telles que :

$$tp = whlpwr - (Cd * v) - (Cf * tw * v) \tag{6.2}$$

De telles contraintes peuvent également être utilisées pour identifier les paramètres de performance critiques et leurs relations avec d'autres paramètres, qui peuvent être suivis tout au long du cycle de vie du système. Un bloc de contraintes comprend des contraintes et leurs paramètres (tels que Cd, Cf et i). Il définit des formes génériques de contraintes pouvant être utilisées dans plusieurs contextes. Les contraintes sont d'abord définies dans les blocs de contraintes dans un diagramme de définition de bloc, puis on utilise un diagramme paramétrique pour lier les paramètres des contraintes aux propriétés.

La traduction en VHDL-AMS est comme suit :

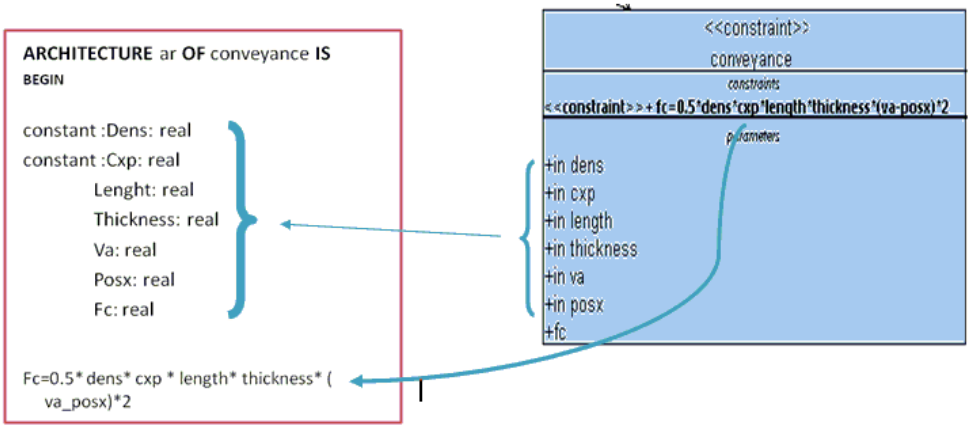


FIGURE 6.5 – Régles de transformation : ConstraintsBlock et VHDL-AMS

— Chaque contrainte et ces paramètres doivent être déclarés dans le corps de l'architecture de l'entité en tant qu'instruction concurrente ou simultanée selon la définition.

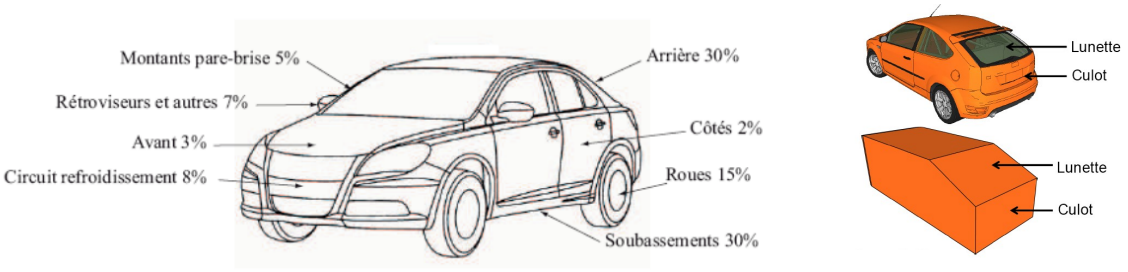
— Si le bloc de contraintes ne fait référence à aucun bloc du BDD alors une nouvelle architecture doit être déclaré sinon la contrainte doit étre ajouté à l'architecture issue de la transformation du bloc en VHDL-AMS. Ces règles sont représentées dans la figure 6.5. Le tableau 6.1 montre une synthèse des correspondances entre les diagrammes SysML et VHDL-AMS.

SysML	VHDL-AMS
BDD : Block	ENTITY
BDD : Block : name	ENTITY : name
BDD : Block : FlowPort	ENTITY : port (type Signal)
BDD : FlowPort : name	port : name
BDD : FlowPort : Direction	port : direction
BDD : FlowPort : Type	port : type
BDD : Block : Constraints	ARCHITECTURE : instructions
BDD : Block : Operation	ARCHITECTURE : operation
BDD : Block : Part	ARCHITECTURE : component
BDD :Block : Reference	ARCHITECTURE : component
BDD : Block : Property	ARCHITECTURE : variable
IBD : Part	ARCHITECTURE : component
IBD : FlowPort	ARCHITECTURE : port
PARAM : ConstraintBlock	ARCHITECTURE : instructions
PARAM : ConstraintBlock : Parameters	ARCHITECTURE : parametersDecl
ACTIVITY : Action	ARCHITECTURE : instructions

TABLE 6.1 – Correspondances SysML-VHDL-AMS

6.2.4 Projet région SyVAD

Ce projet avait pour vocation de permettre la mise en place d'un processus d'aide à la réalisation de micro-systèmes basés sur des actionneurs distribués. Ce processus a permis de réaliser les différentes étapes de conception permettant la modélisation, la vérification, la validation et la simulation de ces systèmes avant d'en réaliser la fabrication. Nous avons modélisé l'ensemble du système constitué des capteurs, actionneurs MEMS et du dispositif de contrôle à l'aide des langages SysML, VHDL-AMS et PSL. Notre étude de cas se situe le domaine automobile, nous nous sommes intéressé au contrôle de la couche limite à l'aide de générateurs distribués de jets inclinés d'une automobile roulant à vitesse variable, dans un environnement normal. Nous avons pris en compte à la fois des phénomènes



((a)) Répartition des pertes de SCxs ((b)) Le corps d'Ahmed

FIGURE 6.6 – Répartition des pertes de SCxs et le corps d'Ahmed

extérieurs en intégrant des capteurs (humidité, température, vitesse de l'écoulement, capteur d'effort, etc) et de la charge du véhicule. Dans le contexte de limitation des consommations énergétiques et des émissions de particules de gaz à effet de serre des véhicules, la maîtrise de la résistance à l'avancement des véhicules constitue l'un des enjeux majeurs de la recherche appliquée à l'automobile. les aérodynamiciens tentent de réduire au maximum les forces de traînée qui freinent la progression du véhicule. Les forces de traînée se décomposent principalement en traînée de pression (80 %) et en traînée de friction (20 %). Ces forces qui augmentent le coefficient de pénétration dans l'air (Cx) et la consommation de carburant interviennent en différents endroits du véhicule (voir figure 6.6(a)). Les surfaces alaires (qui "percent" le fluide à l'avant du véhicule) ne sont pas les principales causes de traînée mais les parties arrières du véhicule comme la lunette et le culot (voir figure 6.6(b)). Nous avons ainsi choisi d'étudier l'écoulement de l'air autour du corps d'Ahmed [4]. Cet objet représente une géométrie simplifiée d'une voiture. Il a aussi déjà fait l'objet de nombreuses études depuis de longues années puisqu'il est la première étape de l'étude de l'écoulement d'un fluide autour d'une voiture. C'est en effet avec ce corps d'Ahmed qu'on simule numériquement le fonctionnement de nouvelles technologies dont le but principal est bien souvent de réduire la traînée.

La modélisation SysML prenant en compte les aspects software, hardware incluant les équations différentielles sur les contraintes inter-composants sont décrits dans [204].

6.2.5 Projet ANR Smart Surface

Dans le cadre du projet Smart Surface (ANR 06 ROBO 0009) du Programme Systèmes Interactifs et Robotique PSIROB 2006 de l'ANR. Le projet Smart Surface a proposé un concept de micro manipulateur distribué et intégré fondé sur une matrice de plusieurs dizaines de micro modules intelligents à l'échelle de quelques centimètres. Chaque micro-module est composé d'un micro-capteur, d'une unité de traitement et d'un micro-actionneur (cf. Figure 6.7). La coopération des micro-modules permet de différencier efficacement les pièces et de commander les micro-actionneurs de manière distribuée au moyen de jets d'air afin de déplacer et positionner de manière précise les micro-pièces sur la smart surface. Nous avons utilisé le langage SysML pour la modélisation du la smart surface [47, 48] et nous avons générer le code VHDL-AMS correspondant à la vue structurelle définie dans SysML.

6.3 SysML et Modelica

6.3.1 Introduction

Nous allons abordé, l'utilisation conjointe de SysML et Modelica dans deux cadres différents. Le premier travail concerne la modélisation d'un réseau de capteurs en vue de simuler la consommation d'énergie. Le deuxième décrit une nouvelle approche de modélisation utilisant le langage (SysML), qui permet de modéliser le système sous test (SUT) et son environnement dans le contexte des processus de conception MIL (Model In-the-Loop). Il permet de valider des systèmes hybrides et critiques

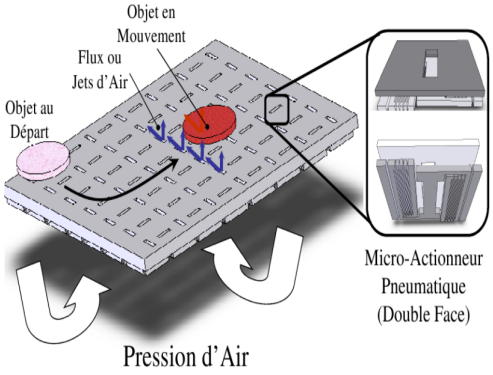


FIGURE 6.7 – Principe de la Smart Surface : ensemble de micro manipulateurs distribués

de sécurité multi-physiques à l'aide de simulations et de génération de tests fonctionnels automatiques à partir de modèles de spécifications écrits avec SysML. L'architecture et le comportement du système sont décrits par un modèle SysML qui combine des fonctions continues et discrètes pour les activités de simulation et de test. Cette approche a été testée et validée sur des études de cas des partenaires de recherche de Femto-ST [3]. Ces travaux ont été menées dans le cadre des projets SyVAD [4] qui est un projet de la région de Franche-Comté, Smart Blocks [5] qui est un projet (ANR) et le Labex Action [6].

Le langage Modelica se rapproche dans son utilisation aux langages VHDL-AMS et Verilog-AMS (issus de l'industrie électronique) par rapport à la façon de modélisation des systèmes (un modèle est représenté par un ensemble d'équations). Le simulateur associé à Modelica résout le système d'équations, d'un modèle, à chaque pas temporel.

Par ailleurs, dans l'article [206] les auteurs ont mené une étude comparative entre Modelica et VHDL-AMS. Ils mentionnent que Modelica permet une approche de modélisation orienté objet comme la définition de classes, la notion d'héritage, qui n'ont pas leurs équivalents en VHDL-AMS. Ils remarquent aussi que Modelica permet l'intégration des informations concernant la représentation graphique du modèle. En Addition, un point important signalé dans cette étude concerne la gestion de la simulation mixte (analogique-numérique). Dans le langage VHDL-AMS la simulation mixte met en jeu le couplage de deux simulateurs distincts dédiés respectivement à la partie discrète et à la partie continue, qui va se synchroniser à certaines dates T. Dans le cas du langage Modelica, il considère à tout instant les équations différentielles, algébriques et discrètes d'un modèle, comme un seul et unique système d'équation ce qui permet une synchronisation automatique entre les parties discrètes et continues.

6.3.2 Le langage Modelica

Modelica est un langage de modélisation orienté objet qui permet la modélisation des systèmes physiques, complexes et hétérogènes. Il peut être considéré comme un langage de modélisation multi-disciplinaires (thermique, mécanique, hydraulique, électronique, etc.) [207]. Modelica est un langage ouvert dont le développement et la promotion sont assurés par l'association Modelica [208]. Les modèles (en Modelica) sont décrits mathématiquement de façon acausale par le biais d'équations différentielles, algébriques et discrètes [209]. Les solveurs de Modelica contiennent des algorithmes de résolution des systèmes d'équations très efficaces qui permettent la manipulation des modèles complexes décrits par des milliers d'équations. Le point fort de Modelica est la réutilisation des composants du modèle ce qui permet la simplification de la tâche de modélisation [210].

3. http ://www.femto-st.fr
4. http ://syvad.univ-fcomte.fr/syvad/
5. http ://smartblocks.univ-fcomte.fr/
6. http ://www.labex-action.fr/fr

6.3.3 Techniques de vérification et validation sous Modelica

Le langage Modelica est un langage exécutable (simulable). Cependant, il permet la simulation et aussi la validation du système à travers des tests. Le travail de référence qui a illustré une démarche de vérification (vérification virtuelle - test) sous Modelica est celui de [211]. Cet article décrit une approche de vérification virtuelle *vVDR*[7] d'un modèle (sous Modelica) par rapport à un système d'exigence. Cette approche s'appuie sur le MBSE (Model Base Systems Engineering) qui définit le modèle, les exigences et les cas de test du système étudié. Ce modèle va être exécuté (simulé) et vérifié par rapport au système d'exigences dès les premières étapes du cycle de développement. Cependant, ces exigences doivent être d'une part standardisées par rapport aux qualités définis par Hull [212] (uniques, bien définis, claires, complètes, légales, abstraites, etc.) et d'autres part compréhensives et exécutables par l'ordinateur (le compilateur). En outre, dans l'approche MBSE les exigences sont reliées avec les éléments du modèle ce qui permet la traçabilité lors de la simulation et la vérification. Cette technique est très répandue dans le domaine industriel en vu de ces avantages par rapport aux méthodes formelles qui exigent une haute qualification (connaissances en mathématique) [213]. Dans cette technique de test, le système d'exigences est utilisé comme l'oracle du système sous test. Dans le cas où une des exigences est violée durant la simulation alors le test est échoué. Par ailleurs, la séparation entre le modèle et le système d'exigences assure la fidélité des résultats de tests. De plus, cette indépendance permet la réutilisabilité des exigences pour vérifier plusieurs modèles en parallèle. Le diagramme d'exigences comporte des exigences vérifiables et non vérifiables. Concernant celles qui sont vérifiables, chacune d'elle est décrite par un diagramme d'états-transitions dont nous trouvons des états qui représentent la violation de l'exigence. Le profil ModelicaML [214] intègre cette approche dans le but d'assurer la vérification des systèmes conçus par ce profil.

Toutefois, dans le travail de I. Groher et al. [215], les auteurs ont présenté une approche de vérification dynamique à travers un système de contraintes. Ils sont focalisés sur l'expression du modèle de contraintes sans celui du modèle d'exigences. De plus, ils ont exprimé ces contraintes en fonction des éléments (variables) du modèle conçu au contraire de la démarche présentée dans [211] qui se base sur la séparation entre le système d'exigences et le modèle conçu (du modèle réel).

Par ailleurs, dans l'article [216] les auteurs ont proposé une extension du langage Modelica à travers une librairie qui permet la vérification et la validation des modèles. Ils ont distingué trois composants de chaque modèle Modelica, notamment le modèle d'environnement, modèle de propriétés et le modèle de comportement. Le modèle de propriétés accède au modèle d'environnement en mode lecture contrairement au modèle de comportement qui y accède en mode écriture. Le modèle de propriétés permet d'exprimer toutes les exigences, les attentes et les limites du système à concevoir (système, sous-système et composants). Ce travail peut être vu comme complémentaire à ModelicaML [214]. Cette extension assure la vérification virtuelle proposée dans [211]. De plus, elle donne la possibilité de vérifier la cohérence du modèle de propriétés et aussi la cohérence entre modèle de propriétés et modèle de comportement.

En addition, dans l'article [217], les auteurs ont proposé une approche de modélisation et de vérification sous l'environnement SimulationX [218]. La modélisation se base sur les diagrammes d'états-transitions qui décrivent le comportement du système. La vérification des propriétés du système modélisé est assurée par des gardes annotant les transitions. Ces gardes représentent des exigences du modèle et l'activation des états, qui ont des transitions contenant ces gardes comme entrantes, indiquent la violation des propriétés (exigences) associées à ces gardes validées.

Dans notre approche, nous envisageons de se baser sur le diagramme d'exigences du SysML, nous utilisons l'approche *vVDR* pour valider un système de réseaux de capteurs sans fils. la partie formalisation des éxigences présentée dans la figure 6.8 représente le processus de la *vVDR*. Le designer de tests sélectionne les exigences vérifiables en associant à ces exigences des expressions booléennes (contraintes) qui représentent des invariants ou des contraintes de sureté. Dans un deuxième temps, nous utilisons la simulation via les transformations de modèles dans le cadre d'un système basé sur les MEMS.

7. the virtual Verification of system Design against system Requirements

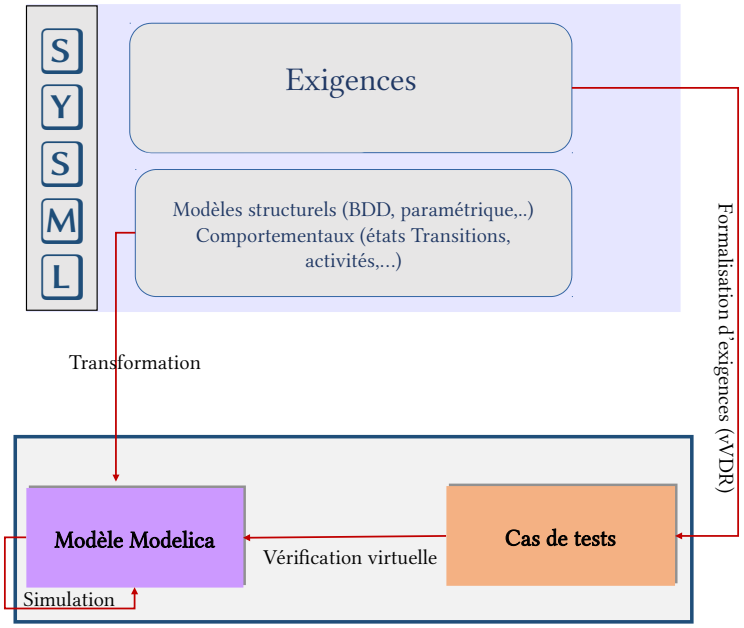


FIGURE 6.8 – Méthodologie de spécification et de validation (SysML et modelica)

6.3.4 Transformation de modèles SysML vers Modelica

Dans cette section nous présentons une démarche de transformation de modèle SysML vers un modèle Modelica dont nous définissons les équivalences et les synonymes (mappages) entre les concepts de ces deux langages.

Mappage entre les éléments de méta-modèles SysML et Modelica

Les langages SysML et Modelica sont basés sur le paradigme orienté objet. Par conséquent, leurs structures sont presque similaires. Par exemple, la structure de base en SysML est l'unité bloc (block) et son correspondant en Modelica est l'entité classe (class). Par ailleurs, il y a des concepts propre à chaque langage dont ils n'ont pas de correspondant dans l'autre langage. Par exemple, les propriétés standards et de de flux (standard property, flow property) de SysML n'ont pas de correspondances dans le langage Modelica. Afin de créer une correspondance entre ces propriétés, nous utilisons le mécanisme d'extension du SysML (Profils).
La démarche que nous avons adopté est celle définie par l'OMG [219] sur la spécification des transformations SysML vers Modelica.

Travaux de transformation de modèles SysML en modèles Modelica

Comme nous avons mentionné auparavant, SysML ne définit pas un langage d'action. En conséquence, les modèles SysML ne sont pas exécutables. Cependant pour simuler ces modèles, nous devons les traduire en un modèle exécutable (sous Modelica). Dans la littérature, nous trouvons plusieurs tentatives de définition des correspondances entre le langage SysML (ou UML2) et le langage Modelica. Ces travaux sont basés sur la norme ADM qui décrit la démarche à suivre afin de faire des transformations de modèles (modèle vers modèle).

Tout d'abord, en 2008 C. J. Paredis et T. A. Johnson [220, 221] ont proposé une démarche générique de transformation de modèle SysML vers un modèle Modelica en utilisant la méthode de transformation TGG (Triple Graph Grammars). Dans cet étude, les diagrammes structurels (statiques : le diagramme de définition de bloc et le diagramme interne de bloc) du SysML sont utilisés pour définir la structure du modèle correspondant sous Modelica (composants (model) et connexions (connector)). Concernant la partie dynamique du modèle Modelica, elle est définie à partir du diagramme paramétrique (du modèle SysML) qui représente un modèle mathématique du comportement du système réel.

D'autres part, R.Renier et al. [222] ont proposé une autre approche pour définir les correspondances entre un modèle SysML et un modèle Modelica en s'appuyant sur l'ingénierie des systèmes basée sur les modèles (Model-Based Systems Engineering - MBSE). Dans ce travail, les auteurs n'ont traité que la transformation de la partie statique (le BDD et l'IBD) du modèle SysML vers un modèle Modelica.

En outre, P. Vasaiely [223] a proposé une démarche de transformation de modèle SysML vers modèle Modelica. Il s'est basé sur les digrammes BBD et IBD pour décrire la partie structurelle du modèle Modelica et le diagramme paramétrique pour spécifier sa partie comportementale.

Par ailleurs, W. Schamai et al. [214] ont proposé un langage de notations graphiques (ModelicaML) qui permet de générer du code Modelica (exécutable). Ils ont crée un profil UML2 pour Modelica dont ils ont modifié la sémantique d'UML2 pour représenter les concepts présents dans Modelica. Dans ce travail, ils ont pris en considération les diagrammes structurels et dynamiques d'UML2 (diagramme de classe, diagramme structure composite, diagramme d'activité et diagramme d'états).

En résumé, les approches de transformation de modèle SysML vers Modelica proposées dans [220, 222, 223] se basent sur les diagrammes statiques du SysML pour décrire la structure globale du modèle cible (en Modelica). Concernant la partie dynamique du modèle Modelica, elle est décrite par le diagramme paramétrique du SysML. Nous pensons que ces approches sont incomplètes vu qu'elles n'ont pas pris en compte les diagrammes dynamiques du SysML qui permettent de décrire le comportement du système d'une façon fidèle et précise.

Dans les travaux présentés dans [214], les auteurs ont proposé un profil d'UML2 pour le langage Modelica en se basant sur ses diagrammes structurels et comportementaux. *ModelicaML* est une extension d'UML2 prenant en compte les spécifications du langage Modelica. Toutefois, nous pensons qu'il est préférable de conserver la sémantique initiale d'UML2 et d'éviter les profils afin de garder l'expressivité initiale du langage. Une autre remarque, le langage SysML est dédié à l'ingénierie des systèmes pas comme UML2 qui est orienté spécification et conception des logiciels. Pour cela, nous pensons que les auteurs n'ont pas tiré profit des nouveaux concepts et diagrammes du langage SysML (comme le diagramme paramétrique qui n'est pas utilisé dans leurs solution).

6.3.5 SysML, Modelica et les RCSF

Les évolutions technologiques enregistrées dans ces dernières décennies concernant les domaines de la microélectronique, de la micromécanique et des technologies de communication sans fil, ont donné naissance aux Réseaux de Capteurs Sans Fil (RCSF). Ces réseaux sont caractérisés par leur faible coût de production, leur capacité à s'autoconfigurer, leur faible consommation d'énergie, leur petite taille et leur déploiement aléatoire. Par conséquent, ils couvrent de nouveaux domaines d'applications très divers caractérisés, parfois, par leur hostilité. De nos jours, l'utilisation des RCSF est de plus en plus demandée pour la supervision et la sécurité. Pour cela, ces réseaux ont connu un fort succès au sein des communautés scientifiques et industrielles qui promeuvent continuellement leurs applications, à savoir militaires, médicales, domotiques, industrielles, etc. Ces systèmes sont contraints et soulèvent un intérêt grandissant de la part des industriels et d'organisations. Ces systèmes sont caractérisés par une forte interaction entre le matériel et le logiciel. Un capteur est un petit dispositif autonome capable d'effectuer des mesures sur son environnement.

Présentation du système technique

La densité du trafic automobile en zone urbaine impose la mise en place d'une signalisation réglementant la circulation. Ceci permet d'en améliorer la sécurité et la fluidité. Les problèmes les plus délicats se situent au niveau des carrefours. En effet, la priorité de passage associée aux éventuels changements de direction, risque de créer des embouteillages. La solution retenue par les responsables de la circulation est la signalisation à l'aide de feux tricolores et bicolores de croisement. Cependant, les changements des feux de signalisation sont gérés par un contrôleur qui les synchronise. La durée du feu de signalisation (jaune, rouge ou vert) dépend du nombre de véhicules en attente dans chaque voie de l'intersection. La figure 6.9 illustre l'environnement d'un système de gestion de carrefour.

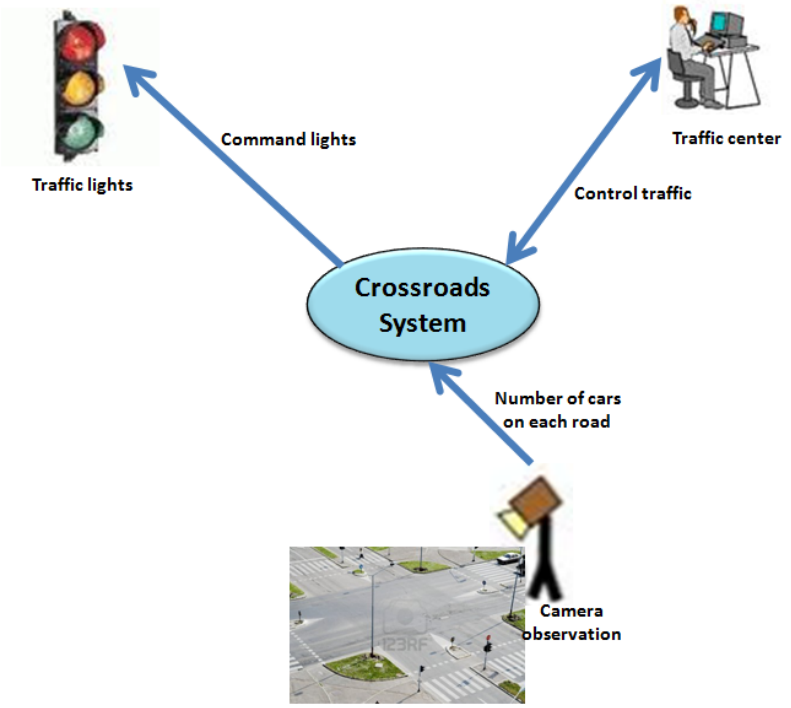


FIGURE 6.9 – L'environnement d'un système de gestion de carrefour

Les hypothèses

L'objectif principal de notre exemple est d'étudier la consommation d'énergie dans un RCSF. Pour cela, nous devons simplifier notre cas d'étude afin de se focaliser sur la propriété à étudier. Les hypothèses ci-dessous permettent de clarifier et simplifier notre cas d'étude.

- Les noeuds capteurs assurent le contrôle du trafic routier sur chaque voie et aussi la communication avec le contrôleur ;
- Les noeuds capteurs sont alimentés en énergie par des batteries ;
- Les feux de signalisation sont : jaune, rouge et vert ;
- La gestion des pannes n'est pas traitée dans ce cas d'étude ;
- La caméra (Additional Sensor Unit) et les feux tricolores (actuator) sont alimentés en énergie par le réseau d'électricité (EDF, pas en batterie) ;
- La caméra envoie au noeud capteur toutes les cinq secondes le nombre des véhicules sur la voie contrôlée ;
- La distance de détection qui limite le nombre maximal de véhicule détectable sur la voie par la caméra est fixée lors de l'installation ;
- la gestion du passage des piétons n'est pas prise en considération dans notre cas d'étude ;
- La communication entre le contrôleur et le centre de gestion de trafic de la ville n'est pas prise en considération dans notre cas d'étude ;

L'étude du système technique

On suppose que le carrefour de notre cas d'étude a deux routes perpendiculaires A et B à double sens. Les quatre composants de signalisation sont nommés comme suit : AN (route A, voie Nord), AS (route A, voie Sud), BO (route B, voie Ouest) et BE (route B, voie Est). La figure 6.10 illustre cette configuration.

La spécification du système

- Le système doit assurer la sécurité des passagers :

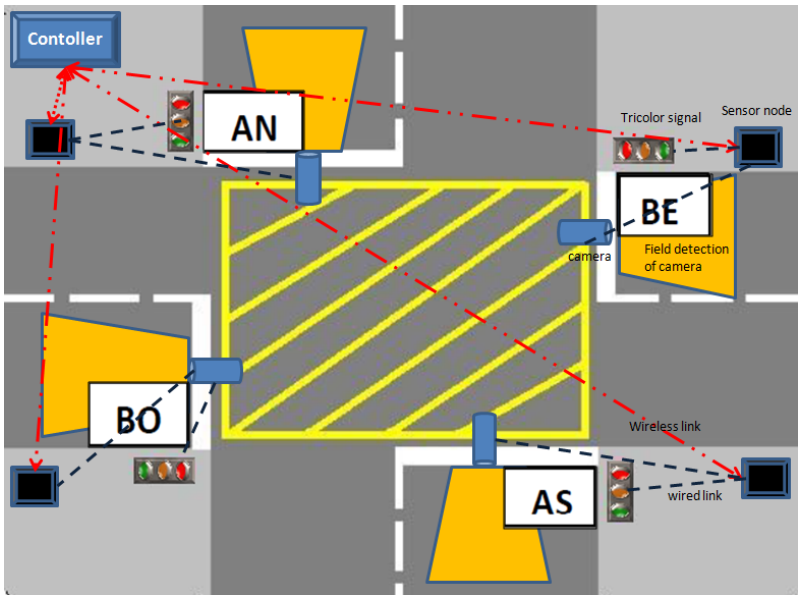


FIGURE 6.10 – L'exemple de carrefour à deux routes

- Les feux de signalisation sur les voies AN et AS sont de même couleur ; il en est de même pour les voies BO et BE ;
- Les feux de signalisation sur les voies AN et AS sont différents de ceux des voies BO et BE
- Le changement de signalisation dans l'intersection se fait d'une façon synchronisée

— Le système doit être efficace :

- Si le feu de signalisation dans une voie est de couleur verte, il ne change que s'il y a des véhicules en attente sur les voies opposées ;
- La transmission radio se fait via le protocole de communication ZIGBEE (IEEE 802.15.04) ;
- Tous les messages entre noeuds capteurs passent par le contrôleur ;
- Les feux de signalisation changent en fonction du nombre de véhicules en attente sur chaque voie.

— Le système doit être économique en consommation d'énergie : La consommation d'énergie doit être minimisée ;

— Le système doit être compatible avec les lois de la circulation routière.

La méthode de vérification des propriétés du système

La démarche utilisée est inspirée de celle décrite dans cet article [211]. Nous avons ajouté une couche qui intègre la modélisation du système en incluant les exigences avec le langage SysML.

L'approche que nous avons adoptée pour vérifier les propriétés de notre système s'appuie sur l'ingénierie des systèmes basée sur les modèles (MBSE). À cet effet, nous avons opté à une séparation entre le modèle conçu, les exigences et les cas de tests en se basant sur l'expressivité du langage SysML. Une fois cette étape est faite, nous avons sélectionné les exigences vérifiables.

Pour cela, nous avons annoté ces exigences par des contraintes qui les traduisent du langage naturel imprécis en code Modelica précis. De plus, dans le but de garantir la traçabilité de ces exigences, nous avons liées ces derniers à des blocs (éléments du modèle) avec des relations de type (**satisfy-by**). En d'autres mots, ces blocs vont satisfaire ces exigences.

Par ailleurs, la séparation entre le modèle de conception (structure et comportement) et le système d'exigences assure la fidélité des résultats de tests ainsi que la réutilisabilité des exigences pour vérifier plusieurs modèles en parallèle. Par conséquent, notre modèle sera simulable et vérifiable par

rapport au système d'exigences qui est utilisé comme l'oracle du système sous test, dès les premières étapes du cycle de développement.

Par exemple, dans la figure 6.11, l'exigence (Id=08) représente la longévité du système dont nous spécifions que le système modélisé ne doit pas y avoir une panne d'énergie au moins durant cinq jours. En d'autres termes, chaque neoud capteur ne doit pas consommer toute son énergie dans les cinq premiers jours du lancement du système. Le bloc qui va satisfaire cette exigence est le système de gestion de l'intersection (intersection monitoring system). En outre, nous avons traduit cette exigence qui est exprimée en langage naturel (dans le diagramme d'exigence du SysML) en code Modelica par le biais d'une contrainte liée à cette exigence.

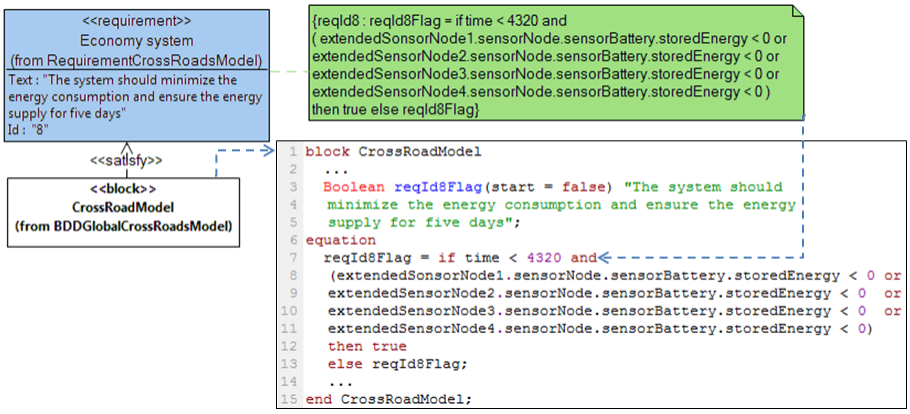


FIGURE 6.11 – L'annotation de l'exigence exprimant la longévité

Lors de la transformation de modèle Modelica vers le code Modelica, nous allons créer automatiquement une variable booléenne (ReqId08Flag) initialisé à faux qui représente l'état de l'exigence (faux : elle est toujours valide, vraie : elle est violée). En outre, cette variable est actualisée à chaque pas élémentaire de simulation du système modélisé. Elle reçoit le résultat de l'exécution de la propriété (contrainte) qui représente l'exigence en fonction de l'évolution de la simulation. Cependant, cette variable booléenne et cette propriété (contrainte) sont insérées respectivement dans la partie déclarative et la partie équation du bloc satisfaisant l'exigence étudiée.

Toutefois, dans la figure 6.12 , nous présentons un autre exemple exprimant une contrainte de sûreté. En d'autres termes, les feux de signalisation dans les deux routes qui forment l'intersection ne doivent pas y être dans le même état.

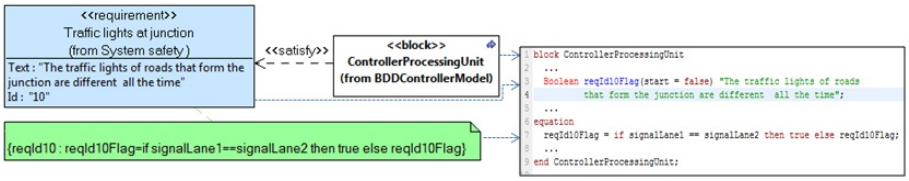


FIGURE 6.12 – L'annotation d'une exigence exprimant une propriété de sûreté

Les résultats obtenus

Dans l'approche de modélisation adoptée, nous avons conçu deux cas de tests, ces scénarios de fonctionnement, nous ont permis d'étudier le comportement du modèle conçu par rapport au système d'exigences. En outre, la comparaison entre les comportements du modèle relatifs à chaque cas de tests, nous a donné la possibilité de bien comprendre les dépendances entre les différents paramètres du modèle conçu.

Par ailleurs, ces cas de tests ne sont pas réalistes car la circulation change d'intensité selon le jour et la nuit, les heures de pointe et les heures normales, les jours de travail et les jours fériés, les saisons, etc. Par conséquent, les résultats fournis par le modèle lors de la simulation ne reflètent pas la réalité par rapport à la durée de vie du système.

D'autres part, ces cas des tests fournissent toujours des éléments pertinents à l'étude du modèle représentant l'application de RCSF. De plus, s'ils ne donnent pas une durée de vie réelle du système de gestion de carrefour, ils permettent d'estimer cette durée de vie en fonction du nombre de messages échangés entre le contrôleur et les noeuds capteurs.

Toutefois, il faut signaler que cette problématique est liée à l'application du RCSF (système de gestion de carrefour) choisie pour illustrer notre approche de modélisation et de vérification. En addition, nous avons repris les paramètres de simulation spécifiée dans [224].

6.3.6 SysML, Modelica et l'application aux MEMS

Dans le cadre de ce travail, nous nous sommes concentré sur trois aspects du processus In-the-Loop. La Figure 6.13 illustre simplement l'approche proposée de validation basée sur un modèle pour des systèmes complexes. Premièrement, nous proposons d'utiliser SysML pour modéliser et spécifier un système en temps réel et son environnement associé. L'utilisation de SysML est motivée par deux raisons : SysML est suffisamment abstrait (proche des exigences) et est efficace pour être le point d'entrée des activités de simulation et de test.

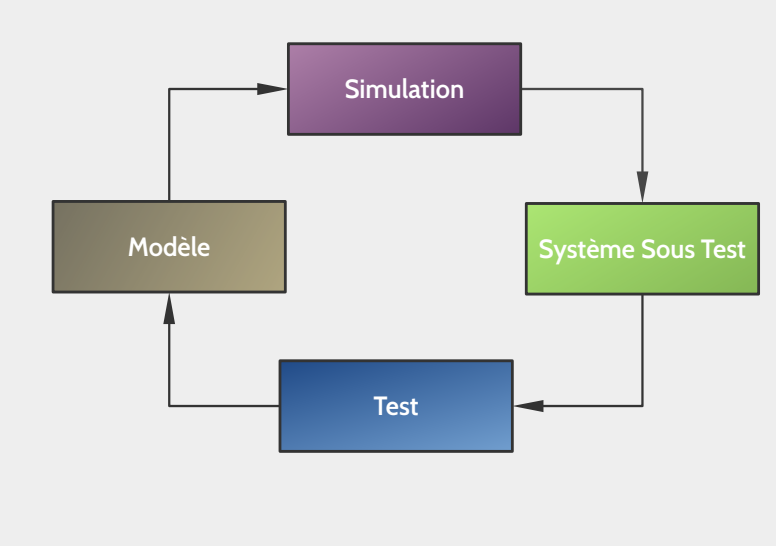


FIGURE 6.13 – Processus de validation basé-modèle pour des systèmes complexes

Deuxièmement, nous proposons une approche qui permet d'effectuer la vérification sur le modèle SysML en utilisant la simulation Modelica. Cette approche définit une contribution au MIL en permettant la génération automatique de prototypes Modelica et leur environnement de simulation associé à l'aide de techniques d'ingénierie dirigée par les modèles. Troisièmement, nous proposons de valider le modèle SysML du système en générant des scénarios de tests à partir du modèle de simulation. Ceci définit une contribution aux tests MIL et HIL (Hardware In-the-Loop). Les travaux de thèse de Jean Marie Gauthier [177] décrivent un cadre formel original basé sur SysML pour la simulation et le test de systèmes multi-physiques et critiques, qui comble l'écart entre la conception de haut niveau "modèle", point de départ des approches MBSE [8], et la plate-forme d'exécution en temps réel, clé de voûte des approches "In-the-Loop". De cette façon, ce cadre permet aux ingénieurs système de rester aussi près que possible des spécifications de conception initiale lors de la réalisation de toutes les étapes de développement. De plus, elle tire partie des deux approches en assurant un processus centré sur le modèle permettant la validation, la simulation et le test dès le début de la conception. Pour cela, l'architecture et le comportement discret du système sont décrit par un modèle SysML, qui est annoté avec du code OCL et Modelica pour spécifier ses caractéristiques discrètes et continues. Ce modèle est utilisé pour générer automatiquement un modèle de simulation en temps réel de Modelica, et des cas de tests sous forme de de boîte noire pour la validation. Les cas de test

8. Model-Based Software Engineering

générés peuvent être simulés à l'aide du modèle de simulation Modelica généré pour valider le modèle de conception et donc le système physique. Par conséquent, le cadre proposé peut contribuer à la fois au processus MIL et HIL . La mise en oeuvre de l'ensemble du processus a permis d'évaluer la pertinence de l'approche dans un contexte opérationnel.

Une illustration de ce travail est présentée sur une utilisation conjointe de SysML et Modelica dans un système à base de MEMS, en vue de le spécifier et de valider certaines exigences non fonctionnelles.

Etude de cas : Un convoyeur à jets d'air

Nous proposons une modélisation SysML et une simulation avec Modelica d'un convoyeur à jet d'air pour micro-objets. Nous nous sommes interessés principalement à la modélisation et la simulation des buses à jet d'air et sur leur influence sur un objet de taille millimétrique. Les résultats obtenus sont discuté et analysé pour obtenir des informations sur le convoyeur système. Ce travail fait partie du projet ANR Smart Blocks.

Le tri et le transport d'objets sont deux tâches principales nécessaires pour les lignes de production. Il existe beaucoup de méthodes différentes pour effectuer ces tâches, mais la plupart d'entre elles exigent un contact entre le convoyeur et les objets transportés. Pour cette raison, les petits objets fragiles peuvent être endommagés par des manipulations alors que les produits propres peuvent être contaminé par le contact avec le convoyeur (notamment dans les industries pharmaceutiques, microélectronique et alimentaire). Pour résoudre ces problèmes, le développement d'un convoyeur modulaire et auto-reconfigurable basés sur une technologie sans contact (technologie air-jets) est nécessaire. Ce convoyeur est composé de blocs de 2,5 centimètres qui seront reliés entre eux pour former la surface de transport. Chaque bloc comprend un ensemble d'actionneurs MEMS dans la face supérieure afin de déplacer les objets, des capteurs capables de détecter la position de l'objet, un micro-contrôleur et des ports de communication qui le relient à ses voisins.

La principale exigence fonctionnelle du système est de transporter un petit objet avec la technologie des jets d'air. Par conséquent, nous devons prédire et maîtriser le comportement d'un petit objet soumis à un flux d'air à grande vitesse. La modélisation, la simulation et les tests sont des moyens de valider une telle exigence.

La modélisation et la simulation des jets d'air sont des tâches importantes pour comprendre comment un petit objet peut se déplacer d'un point à un autre. De plus, la simulation pourrait nous donner des indices sur la disposition des blocs pour fournir des trajectoires non linéaires. Le premier objectif est de proposer une représentation mathématique de l'influence des jets d'air sur un un petit objet. Le deuxième objectif est de présenter un modèle SysML de ce système, s'appuyant sur le modèle mathématique précédemment défini. Enfin, le modèle SysML du convoyeur de bloc intelligent est automatiquement traduit en code de simulation Modelica pour effectuer des expériences numériques.

Modèles mathématiques

Pour modéliser le comportement de l'objet soumis à la propulsion de plusieurs jets d'air, nous considérons la force élémentaire d'un jet d'air en deux dimensions. En effet, nous considérons les hypothèses suivantes :

— La lévitation est fournie par les jets d'air sous l'objet mais nous ne considérons pas leur effet, ils ne sont pas évalués

— Les jets d'air sont indépendants, Il n'y a aucune interaction entre les jets d'air.

Ensuite, nous faisons la somme de chaque jet d'air avec leur degré d'influence. Cette influence dépend de la position du jet d'air par rapport à la position et à la distance avec l'objet. Globalement, l'objet est soumis à deux forces : une force motrice et une force de déplacement d'opposition (frottement d'air). Leur portée est représentée dans la figure 6.14.

$$\sum \overrightarrow{F} = \overrightarrow{F_d} + \overrightarrow{F_v} = m \cdot \ddot{x} \cdot \overrightarrow{u_n} \quad (1)$$

m : poids de l'objet, x : position de l'objet.

La force motrice $overrightarrow F_d$ résulte de la somme des forces de propulsion par jets d'air qui

agissent à proximité de l'objet.

$$\overrightarrow{F_d} = \sum_{i=1}^{M} \sum_{n=1}^{N} \Delta i,n \cdot \overrightarrow{f_{i,n}} \tag{2}$$

$\Delta i,n = 1$ si le jet d'air atteint la surface de l'objet.

La force élémentaire $\overrightarrow{f_{i,n}}$ (Figure 6.15) induite par chaque jet d'air est déterminée comme suit [225] :

$$\overrightarrow{f_{i,n}} = \frac{1}{2}\rho \cdot C_D \cdot s_{i,n} \cdot v_{i,n}^2 \cdot \overrightarrow{u_n} \tag{3}$$

$\rho = 1,3 kg/m^3$ si le jet d'air atteint la surface de l'objet
$C_D = 1,2$ coefficient de traînée pour un demi-cylindre
$s_{i,n}$: surface projetée au contact du jet d'air
$v_{i,n}$: la vitesse relative du jet d'air, définie comme :

$$v_{i,n} = \dot{x} - v_{air}(\delta_{i,n}) \tag{4}$$

v_{air} : vitesse absolue du jet d'air, à la sortie des buses d'air, qui peut prendre deux valeurs.

$$\delta_{i,j} > 0 \Rightarrow v_{air}(\delta_{i,n}) = 5500 \cdot \exp^{-\frac{\delta_{i,j}^2}{4}} \tag{5}$$

$$\delta_{i,j} = 0 \Rightarrow v_{air}(\delta_{i,n}) = 0 \tag{6}$$

$\delta_{i,j}$: distance entre la buse d'air et le point de contact de l'objet.

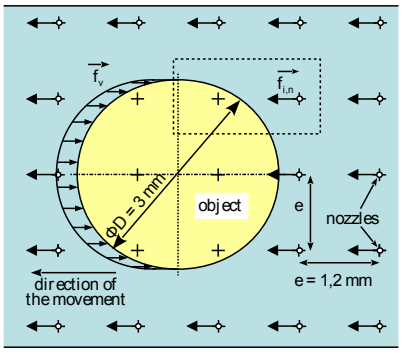


FIGURE 6.14 – Objet soumis au jets d'air

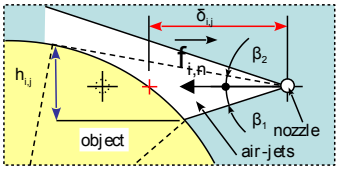


FIGURE 6.15 – Modèle d'un jet d'air

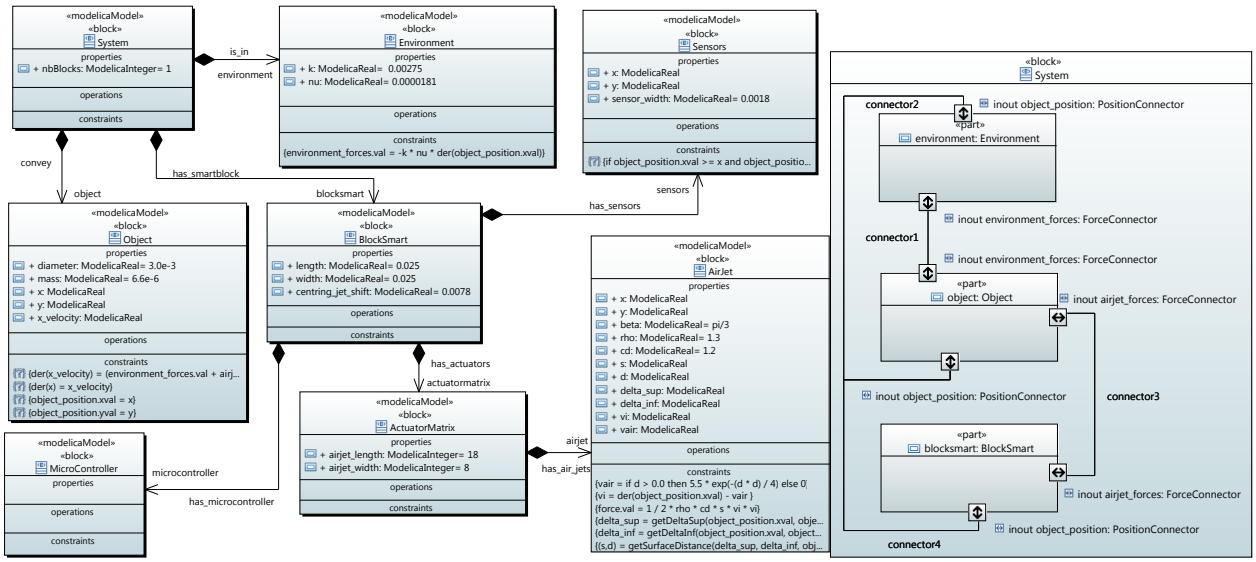
En opposition au déplacement, des forces de traînée visqueuses apparaissent $\overrightarrow{F_v} = \int_S \overrightarrow{f_v}$ représentées par :

$$\overrightarrow{F_v} = -K \cdot \eta \cdot \dot{x} \cdot \overrightarrow{u_n} \tag{7}$$

$K = 2,75 mm$: coefficient géométrique de la force de viscosité
$\eta = 1,81.10^{-}5$: Viscosité de l'air

La combinaison dynamique de ces fonctions n'est pas facile car l'objet est toujours en mouvement et la vitesse n'est pas continue. Le contrôle de la trajectoire dépend de la position de l'objet sur chaque bloc.

Cette sous-section a décrit le modèle analytique d'un objet soumis à des jets d'air. L'étape suivante consiste à modéliser le système en utilisant SysML afin de générer le code de Modelica simulable.



((a)) BDD du système ((b)) IBD du système

FIGURE 6.16 – Modèle SysML du système

Modélisation SysML

Le premier niveau de modélisation SysML est le diagramme BDD (Block Definition Diagram). La Figure 6.16(a) montre une vue structurelle du système. Il a été réalisé avec le logiciel de modélisation appelé Papyrus. Le bloc *System* est décomposé en 3 sous-blocs (*Environment*, *Object* et *BlockSmart*). Le bloc *Object* représente un objet à déplacer. Le bloc *BlockSmart* représente des blocs convoyeur composés d'une matrice d'actionneurs, d'un capteur et d'un microcontrôleur. Le bloc *ActuatorMatrix* est composé de jets d'air. Ces jets d'air sont au coeur de notre étude. Enfin, le bloc *Environment* représente l'air ambiant qui impose une force de frottement $\overrightarrow{F_v}$ à l'objet. De plus, chaque bloc du BDD contient les équations du modèle mathématique. Ces équations sont modelisées pour permettre leur interprétation et leur résolution à chaque pas de temps.

Le deuxième niveau de modélisation SysML est le diagramme de bloc interne (IBD). La Figure 6.16(b) montre la vue interne du système. Cette vue permet de représenter les interactions entre les composants du système, pour montrer les forces et la position de l'objet.
Comme l'objet est soumis à deux forces principales $\overrightarrow{F_v}$ et $\overrightarrow{F_d}$, son bloc est lié au bloc *Environment* et *BlockSmart*. De plus, chaque bloc convoyeur doit connaître la position de l'objet pour calculer la force des jets d'air. C'est pourquoi il y a un lien typé avec *PositionConnector* entre l'objet et le convoyeur.

Cette sous-section décrit le modèle du la smart-blocks au niveau du système. La dernière étape de notre travail consiste à simuler le modèle afin de le valider et d'étudier l'influence des jets d'air sur l'objet. Nous avons développé un plugin pour Papyrus qui peut traduire automatiquement le modèle SysML en code Modelica. Le processus de génération de code est décrit dans les travaux de thèse de Jean-Marie Gauthier [177]. Nous présentons des résultats de simulation directement à partir de l'exécution du modèle dans l'environnement OpenModelica.

Simulation

Les simulations ont été effectuées dans les conditions initiales suivantes :

— La masse de l'objet : 6.6e-6 kg

— La taille de l'objet : 0.003 m de diamétre

— Vitesse initiale de l'objet : 0.0 m/s

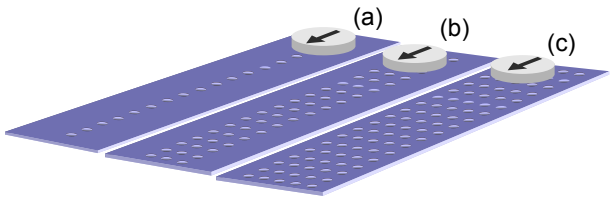
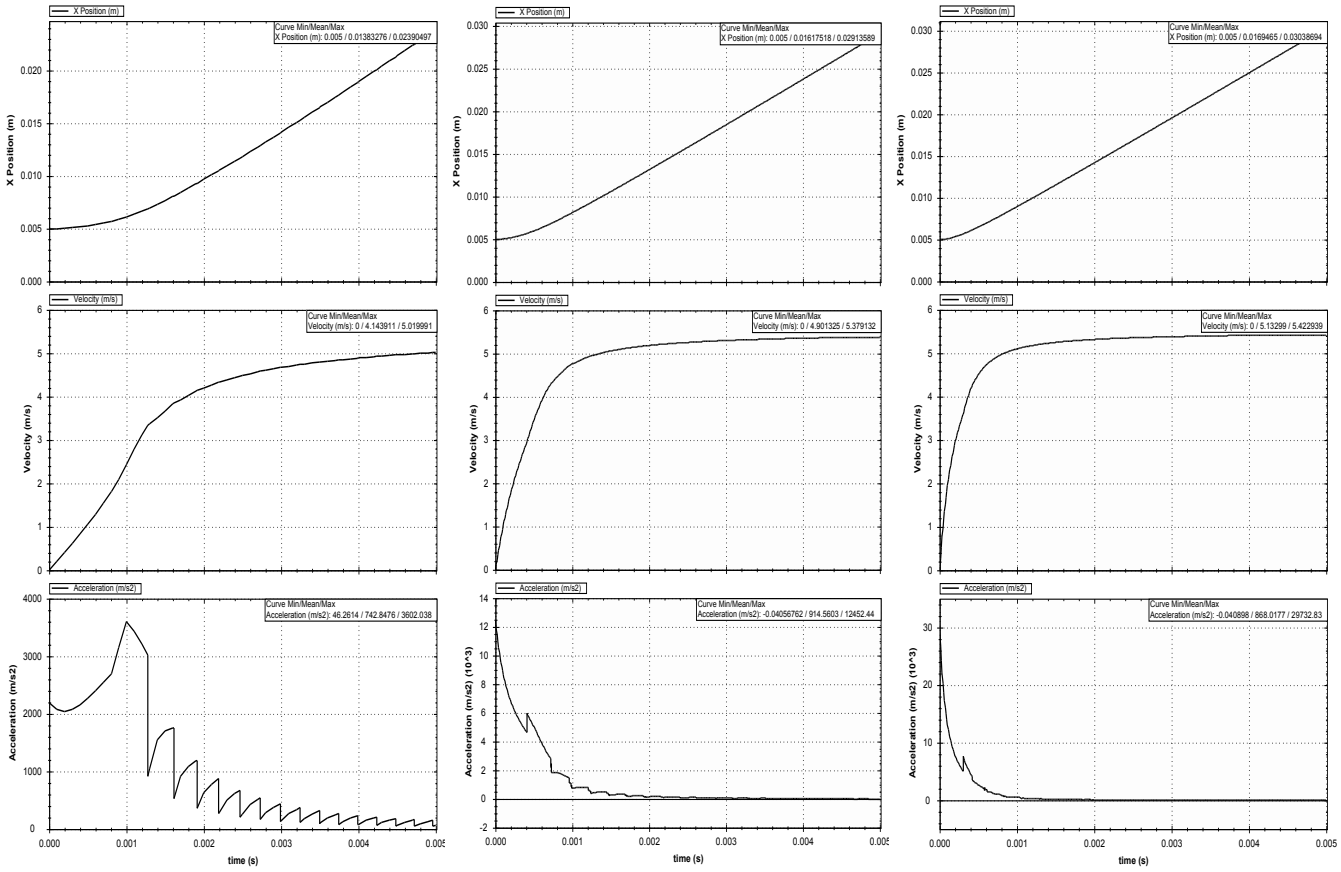


FIGURE 6.17 – Scenarios de simulation



((a)) Une ligne de 18 jets d'air ((b)) Trois lignes de 18 jets d'air ((c)) Cinq lignes de 18 jets d'air

FIGURE 6.18 – Résultats de la simulation Modelica

Pour rappel, un bloc est composé d'une surface de transport (appelée *ActuatorMatrix* dans le modèle SysML). Cette surface est une matrice de 18 jets d'air en longueur et de 8 jets d'air en largeur. Nous avons simulé trois scénarios pour comprendre l'influence des jets d'air sur l'objet. Tout d'abord, l'objet est soumis à une ligne de jets d'air, puis à trois lignes et enfin à cinq lignes de jets d'air. Pour chaque simulation, l'objet est centré sur la surface. Ces scénarios, illustrés sur la Figure 6.17, permettent de valider le modèle en ce qui concerne les comportements attendus de L'object. Par conséquent, il n'est pas nécessaire de simuler des scénarios plus importants car nous avons besoin d'un modèle de jet d'air correct qui pourrait être un bon point de départ pour des scénarios plus complexes.

Comme indiqué sur la Figure 6.18, les résultats de la simulation se concentrent sur la position, la vitesse et l'accélération de l'objet. Pendant que l'objet bouge, il perd successivement et gagne des forces motrices jets d'air.

Sur les résultats de simulation 6.18(a), seulement un ou deux jets d'air sont influents à la fois. Ceci résulte en une variation de forces de 100%. C'est pourquoi nous pouvons observer des secousses sur la courbe d'accélération. Sur les résultats de simulation 6.18(b), un lissage de courbe est observé. En effet, il y a successivement 3 à 4 et 4 à 6 jets d'air influents à la fois, ce qui implique une variation de 50% des forces.

Concernant le dernier scénario, dont les résultats sont illustrés sur la Figure 6.18(c), on peut observer un effet de lissage plus important sur la courbe d'accélération. Nous avons également simulé un objet soumis à 8 lignes de jets d'air. Mais, comme le modèle prend en compte la distance entre l'objet et les jets d'air, seuls les jets d'air les plus proches influencent la position x de l'objet. Par conséquent, les résultats de la simulation sont identiques pour cinq lignes ou plus de jets d'air.

Malgré de fortes hypothèses de départ, notamment l'absence d'influence d'un jet d'air sur les autres, nous considérons que le modèle est un bon point de départ pour d'autres scénarios. Par exemple, nous avons commencé à explorer les capacités de freinage d'une smart-blocks positionné dans la direction opposée à la trajectoire de l'objet. Nous avons observé qu'un petit nombre de jets d'air sont suffisants pour ralentir efficacement l'objet.

6.4 Bilan

Ce travail a donné lieu :

1. **Trois projets**
   — 2006-2010 *Programme Systèmes Interactifs et Robotique (ROBO), Projet Smart Surface* [9] : Le projet avait pour objectif la conception, le développement et le contrôle d'un système microrobotique distribué pour le convoyage, le positionnement et le tri de micropièces à l'échelle mésoscopique (mm au mm).
   Partenaires : LAAS (Université de Toulouse III), LIMMS (CNRS Japon), MN2S (FEMTO-ST)).
   Dans ce projet, j'ai principalement travaillé sur la partie modélisation et validation de la smart surface.
   — 2011-2015 : *Smart Blocks* [10] : ce projet combine des systèmes microtechniques et informatiques pour créer un convoyeur modulaire auto-reconfigurable basée sur une technologie sans contact. Ce convoyeur est composé de blocs de taille centimétrique (1-3 cm), appelés blocs intelligents, qui seront reliés entre eux pour former la surface de transport. Chaque bloc comprend un ensemble d'actionneur MEMS dans la face supérieure afin de déplacer les objets.
   Partenaires : LAAS (Université de Toulouse III), LIMMS (CNRS Japon),MN2S (FEMTO-ST))
   Dans ce projet, j'ai travaillé particulièrement sur la modélisation en SysML et la simulation dans l'environnement Modelica du convoyeur d'objets à jets d'air.
   — **Projet Région 2011-2014** : *SyVAD (Modélisation SysML pour la Validation et la Vérification de micro-systèmes : application aux générateurs de microjets distribués*. J'ai participé aux parties sur la définition des exigences, la modélisation du cas d'étude (corps d'Ahmed) et la définition des règles pour la génération du code VHDL-AMS.
2. **Thèses de doctorat UFC** [177, 56]. Voir tableau 6.2

Année	Candidat	Titre
2016	A. Abbas	*Combining SysML and SystemC to Simulate and Verify Complex Systems.*
2015	JM. Gauthier	*Combining Discrete and Continuous Domains for SysML-Based Simulation and Test Generation .*

TABLE 6.2 – Thèses modélisation et validation des systèmes complexes

3. **DEA-Master 2 recherche**. Voir tableau 6.3

4. **Communications suivantes :** . [204, 226, 227, 48, 50, 52, 55, 54, 53, 59, 57, 58, 228]

9. http ://smartsurface.free.fr/
10. http ://smartblocks.univ-fcomte.fr/

Année	Candidat	Titre
2010	M. Iraqui	*Modélisation SysML de la smart surface et génération de code VHDL-AMS*
2011	K. Ourfella	*Expression en PSL des exigences SysML*
2012	S. Berrani	*Combinaison des langages SysMl et Modelica pour spécifier et valider les réseaux de capteurs.*
	J.M. Gauthier	*Transformation des modèles SysML vers VHDL-AMS.*

TABLE 6.3 – DEA-Master 2 recherche chapitre 6

7

Vérification de l'assemblage des composants SysML

Plan du chapitre

7.1 Introduction

Dans notre vie quotidienne, nous utilisons de plus en plus de dispositifs électroniques pour le divertissement, la navigation GPS et la communication. Ces appareils, appelés des dispositifs embarqués, regroupent une grande variété de composants, ayant des fonctionnalités complexes. La demande de ce type de dispositifs dans le domaine de l'industrie connaît une croissance exponentielle. Du point de vue industriel, les besoins en terme de composants logiciels/matériels s'expriment plutôt par la nécessité de disposer de technologies permettant de composer rapidement des applications à partir d'éléments (logiciels et matériels) réutilisables. La conception, le développement et la garantie d'une haute fiabilité et de bonnes performances de tels systèmes sont devenus un défi.

Un des points clé actuel dans les recherches en génie logiciel est l'étude du développement d'applications par assemblage de composants. L'approche par composants vise la réutilisation par assemblage aisé et cohérent des composants. Mais l'obtention d'un assemblage de composants cohérent n'est pas un exercice facile. Nous proposons une approche qui combine les composants SysML et les automates d'interface afin d'assembler des composants et vérifier formellement leur compatibilité. Notre proposition vise à vérifier formellement l'assemblage de composants spécifiés avec SysML. L'architecture du système à base de composant est alors spécifiée par le diagramme de définition de blocs et les interactions entre blocs sont spécifiées par le diagramme de bloc interne. Les protocoles d'interaction sont spécifiés par des diagrammes de séquence. Ces derniers sont nécessaires pour exploiter le formalisme des automates d'interface. Ce formalisme se base sur des automates d'entrée/sortie (E/S) [229] pour spécifier la signature et les protocoles d'interaction des interfaces des composants. Nous proposons alors de formaliser les spécifications SysML afin de pouvoir exploiter les automates d'interface. Nous améliorons ainsi l'approche basée sur les automates d'interface par la considération de l'architecture du système spécifiée en SysML pour la vérification de l'assemblage de composants.

Les systèmes à base de composants sont constitués d'une collection d'entités, appelées composants. L'idée du CBSE (Component Based Software Engineering) est de développer des applications logicielles à partir de l'assemblage de diverses bibliothèques de composants [230], [231]. Cette approche de développement permet d'étendre les systèmes basés sur des composants via la réutilisation des composants prêts à l'emploi. Ceci a pour but d'économiser sur les coûts et le temps de développement. Un composant est une unité de composition avec des spécifications des interfaces à base de contrats et de dépendances explicites [230].

Une interface décrit les services offerts et requis par un composant sans divulguer la mise en oeuvre de celui-ci. C'est le seul accès aux informations sur le composant. Les interfaces peuvent décrire les informations sur les composants telles que la signature (noms de méthodes et leurs types), le comportement ou le protocole (ordonnancement des appels de méthodes), la sémantique des méthodes et la qualité des niveaux de services.

Le succès de l'application de l'approche par composants dépend de l'interopérabilité (nous disons aussi la compatibilité des composants) des composants connectés. L'interopérabilité peut être définie comme la capacité de deux entités ou plus à communiquer et coopérer malgré les différences de langages d'implémentation, d'environnement d'exécution ou du modèle abstraction, [232, 233]. Il y a interopérabilité entre les composants lorsque les interfaces sont compatibles.

Dans ce travail, nous nous concentrons sur la vérification de l'interopérabilité entre les composants dont leurs interfaces sont décrites par des automates d'interface. Cette approche a été proposée par L.Alfaro et T.Henzinger [234, 235, 236]. Ils ont proposé de spécifier l'interface des composants avec des automates, qui sont étiquetés par des actions d'entrées/sorties et internes. Ces automates permettent de décrire les informations sur les composants au niveau de la signature et du protocole. Une approche de vérification intéressante a également été proposée pour détecter les incompatibilités au niveau des signatures et des protocoles entre les interfaces de deux composants. La vérification est basée sur la composition des automates d'interfaces, ce qui est réalisé en synchronisant les actions des composants.

Notre objectif est de construire des modèles formels correspondant à un sous-ensemble de diagrammes SysML, qui décrivent des composants, afin de vérifier formellement leur assemblage (ou leur interopérabilité). Nous proposons une approche qui combine des modèles SysML de composants et d'automates d'interface afin d'assembler des composants et de vérifier formellement leur

interopérabilité. Nous spécifions l'architecture, basée sur les composants, avec les diagrammes de définition de blocs SysML et les liens de composition entre les composants avec des diagrammes de blocs internes. Les protocoles de composants sont spécifiés avec des diagrammes de séquence, qui sont plus compréhensibles que les automates d'interface. Ces diagrammes sont traduits en modèles formels afin d'exploiter l'approche des automates d'interface et aussi pour pallier un inconvénient de cette approche. Effectivement, le principal inconvénient de l'approche des automates d'interface est qu'elle ne considere pas en entrée l'architecture complète du système pour vérifier la compatibilité. En effet, cette approche prend en entrée que deux automates d'interface de composants, ce qui constitue une architecture partielle du système. Par conséquent, nous proposons d'enrichir et d'adapter cette approche en exploitant les modèles formels des diagrammes SysML qui spécifient l'architecture du système basée sur les composants, et considérer cette information dans la vérification de l'assemblage des composants. Nous proposons donc une solution pour vérifier la composition d'un ensemble de composants (plus de deux) selon l'architecture du système SysML.

Ce chapitre est structuré comme suit : Dans la section 7.2, nous présentons le concept d'automate d'interface. La spécification informelle du système CyCab et les modèles SysML sont présentés dans la section 7.3. Dans la section 7.4, nous présentons notre modélisation formelle des diagrammes SysML et notre approche pour vérifier l'assemblage des composants en considérant les diagrammes SysML. La section 7.5 porte sur les travaux connexes.

7.2 Automates d'interface

Les automates d'interface (IAs) ont été définis dans [234], pour modéliser le comportement temporel des interfaces de composants logiciels et matériels. Ces modèles ne sont pas activés [1] comme dans les automates d'E/S. Chaque interface de composant est décrite par un automate d'interface où les actions d'entrée (assignés par **?**) sont utilisées pour modéliser les méthodes qui peuvent être appelées, et la fin de la réception des messages de canaux de communication, ainsi que les valeurs de retour de ces appels. Les actions de sortie (assignées par **!**) sont utilisées pour modéliser les appels de méthodes, les transmissions de messages via des canaux de communication, et les exceptions qui se produisent pendant l'exécution de la méthode. Les actions masquées (assigné par **;**) décrivent les opérations locales du composant.

Définition 7.2.1. *(Automate d'interface).*
L'automate $A = \langle S_A, I_A, \Sigma_A^I, \Sigma_A^O, \Sigma_A^H, \delta_A \rangle$ *consiste en :*

- *Un ensemble fini S_A d'états ;*
- *Un sous-ensemble d'états initiaux $I_A \subseteq S_A$. Sa cardinalité $card(I_A) \geq 1$ et si $I_A = \emptyset$ alors A est appelé vide ;*
- *Trois ensembles disjoints Σ_A^I, Σ_A^O et Σ_A^H d'actions d'entrées, de sorties et masquées ;*
- *Un ensemble $\delta_A \subseteq S_A \times \Sigma_A \times S_A$ de transitions entre les états*

La composition de deux automates d'interfaces est possible *si et seulement si* leurs actions sont disjointes, sauf les actions d'entrées et de sorties partagées entre eux. Lorsque nous les composons, les actions partagées sont synchronisées et toutes les autres sont entrelacées de manière asynchrone. $Shared(A_1,A_2) = (\Sigma_{A_1}^I \cap \Sigma_{A_2}^O) \cup (\Sigma_{A_2}^I \cap \Sigma_{A_1}^O)$ est l'ensemble d'actions partagées entre A_1 et A_2. nous pouvons maintenant définir correctement le produit des automates $A_1 \otimes A_2$.

Définition 7.2.2. *(Produit synchronisé).*
Soient A_1 et A_2 deux automates d'interface composables. Le produit $A_1 \otimes A_2$ est défini par :

- $S_{A_1 \otimes A_2} = S_{A_1} \times S_{A_2}$ *et* $I_{A_1 \otimes A_2} = I_{A_1} \times I_{A_2}$;
- $\Sigma_{A_1 \otimes A_2}^I = (\Sigma_{A_1}^I \cup \Sigma_{A_2}^I) \setminus Shared(A_1,A_2)$;
- $\Sigma_{A_1 \otimes A_2}^O = (\Sigma_{A_1}^O \cup \Sigma_{A_2}^O) \setminus Shared(A_1,A_2)$;
- $\Sigma_{A_1 \otimes A_2}^H = \Sigma_{A_1}^H \cup \Sigma_{A_2}^H \cup Shared(A_1,A_2)$;

1. les actions d'entrées ne sont pas activées à chaque état d'un automate

— $((s_1,s_2),a,(s'_1,s'_2)) \in \delta_{A_1 \otimes A_2}$ if
  — $a \notin Shared(A_1,A_2) \wedge (s_1,a,s'_1) \in \delta_{A_1} \wedge s_2 = s'_2$
  — $a \notin Shared(A_1,A_2) \wedge (s_2,a,s'_2) \in \delta_{A_2} \wedge s_1 = s'_1$
  — $a \in Shared(A_1,A_2) \wedge (s_1,a,s'_1) \in \delta_{A_1} \wedge (s_2,a,s'_2) \in \delta_{A_2}$.

L'incompatibilité entre deux automates d'interfaces composables est dû à l'existence de certains états (s_1,s_2) dans le produit pour lequel un composant requiert une action partagée *sa* de l'état s_1 qui n'est pas fourni par l'état s_2 dans l'autre composant ou vice versa. Ces états sont appelés *états illégaux*.

Définition 7.2.3. *(États illégaux).*
Soient deux automates d'interfaces composables A_1 *et* A_2, *L'ensemble* Illegal *des états illégaux* $(A_1,A_2) \subseteq S_{A_1} \times S_{A_2}$ *de* $A_1 \otimes A_2$ *défini par* $\{(s_1,s_2) \in S_{A_1} \times S_{A_2} | \exists a \in Shared(A_1,A_2).$ $(a \in \Sigma^O_{A_1}(s_1) \wedge a \notin \Sigma^I_{A_2}(s_2)) \vee (a \in \Sigma^O_{A_2}(s_2) \wedge a \notin \Sigma^I_{A_1}(s_1))\}$.

L'accessibilité des états dans *Illegal* $\{A_1, A_2\}$ n'implique pas que A_1 et A_2 ne sont pas compatibles. L'existence d'un environnement E qui produit des entrées appropriées pour le produit $A_1 \otimes A_2$ assure que les états illégaux ne peuvent pas être pris en compte et ensuite A_1 et A_2 peuvent être utilisés ensemble. Les états compatibles, indiqués par *Comp* (A_1, A_2), sont des états à partir desquels l'environnement peut empêcher l'entrée dans les états illégaux. La compatibilité peut être définie différemment, A_1 et A_2 sont compatibles *si et seulement si* leur état initial est compatible

Définition 7.2.4. *(Composition).*
Soient deux automates d'interfaces compatibles A_1 et A_2. La composition $A_1 \parallel A_2$ est un automate d'interface défini par *:* (*i*) $S_{A_1 \parallel A_2} = Comp(A_1,A_2)$, (*ii*) *l'état initial est* $I_{A_1 \parallel A_2} = I_{A_1 \otimes A_2} \cap Comp(A_1,A_2)$, (*iii*) $\Sigma_{A_1 \parallel A_2} = \Sigma_{A_1 \otimes A_2}$, *et* (*iv*) *l'ensemble des transitions est* $\delta_{A_1 \parallel A_2} = \delta_{A_1 \otimes A_2} \cap (Comp(A_1,A_2) \times \Sigma_{A_1 \parallel A_2} \times Comp(A_1,A_2)$.

La vérification de la compatibilité entre le composant C_1 et le composant C_2 est obtenue en vérifiant la compatibilité entre leurs automates d'interfaces A_1 et A_2. Les étapes de vérification de la compatibilité entre A_1 et A_2 sont décrites par l'algorithme 2.

Algorithm 2 $Compos(A_1,A_2)$

Input : les automates d'interfaces A_1, A_2.
Ouput : $A_1 \parallel A_2$.
Begin

1. Vérifier que A_1 et A_2 sont compatibles,
2. Calculer le produit $A_1 \times A_2$,
3. Déterminer l'ensemble *Illegal* $A_1 \times A_2$,
4. Déterminer l'ensemble des états incompatibles dans $A_1 \times A_2$: les états accessibles à partir des états illégaux en utilisant seulement les actions internes et de sorties (on suppose l'existence d'un environnement utile),
5. Calculer la composition $A_1 \parallel A_2$ en éliminant dans l'automate $A_1 \times A_2$, les états illégaux, les états incompatibles, et les états inaccessibles à partir de l'état initial,
6. Si $A_1 \parallel A_2$ est vide alors A_1 et A_2 sont compatibles, donc C_1 et C_2 peuvent être assemblés correctement dans n'importe quel environnement.

End

La complexité de cette approche est linéaire dans le temps sur $|A_1|$ et $|A_2|$ [234]. Les étapes de vérification de cette approche peuvent être effectuées par l'outil *Ticc* [237].

7.3 Modélisation du CyCab en SysML

Dans cette section, nous présentons une modélisation SysML d'un exemple de système à base de composants.

7.3.1 Description du CyCab

A titre d'exemple, nous considérons une voiture CyCab qui est un système basé sur les composants ([238]), développé par l'INRIA [2]. Le CyCab est un petit véhicule routier, électrique et automatique, utilisé comme moyen de transport conçus essentiellement pour le transport autonome. Aucun humain à ses commandes, et pourtant le CyCab peut voir la route, détecter et éviter les obstacles, recevoir et échanger des informations avec d'autres véhicules ou avec une centrale de gestion, et prendre des décisions en temps réel en s'adaptant à un environnement extérieur qui évolue en permanence.

Des services permettant aux utilisateurs de se déplacer via un ensemble de stations pré-installées. Le Cycab (voir Figure 7.1) a une longueur qui n'excède pas celle d'un vélo, c'est-à-dire 1,9 mètre, destiné au transport de deux personnes (largeur : 1,2 mètre) dans des environnements urbains et péri-urbains encombrés : centres-villes, centres commerciaux, parcs d'attractions, aéroports, etc. N'étant pas prévu pour de longs trajets, sa vitesse est limitée à 30 kilomètre à l'heure.

FIGURE 7.1 – Le véhicule Cycab

7.3.2 Diagramme de définition de blocs

Un diagramme de définition de blocs (*BDD*) décrit la structure du système, il est basé sur le diagramme composite d'UML, qui étend le diagramme de classes d'UML. Le rôle d'un *BDD* est de décrire les relations entre blocs, qui sont des éléments structurels de base visant à spécifier la hiérarchie et les interconnexions du système à modéliser. Les interfaces requises (relation uses) et offertes (relation implements) des composants sont également décrites. Le bloc ou "Block" en SysML est l'équivalent de la classe en UML, il est l'élément structurel de base en SysML.

La Figure 7.2 montre un exemple de *BDD* avec huit blocs. C'est le premier niveau de modélisation du *CyCab*. Le bloc nommé ***CyCab* System** représente le système entier. Il est décomposé en deux sous-blocs (**Vehicle**, et **Station**), ils sont liés au système *CyCab* par une relation de composition. Le composant **Vehicle** est divisé en trois sous-composants qui sont **Starter**, **Véhicule Core (VC)** et **Arrêt d'urgence (EH)**. **Station** est décomposée en deux sous-composants qui sont **Sensor** et **Unité d'ordinateur (CU)**. Nous utilisons un *BDD* pour spécifier formellement l'architecture du système et exploiter cette spécification dans l'assemblage des composants.

7.3.3 Le diagramme de bloc interne

Le *diagramme de bloc interne* (*IBD*) permet au concepteur d'affiner l'aspect structurel du modèle. L'*IBD* est l'équivalent dans SysML du diagramme de structure composite d'*UML*. Dans l'*IBD*, les parties sont des éléments de base assemblés pour définir comment elles collaborent pour réaliser la structure en blocs et/ou comportement. Une *partie* dans SysML correspond à un objet dans *UML*.

2. Institut National de Recherche en Informatique et Automatique

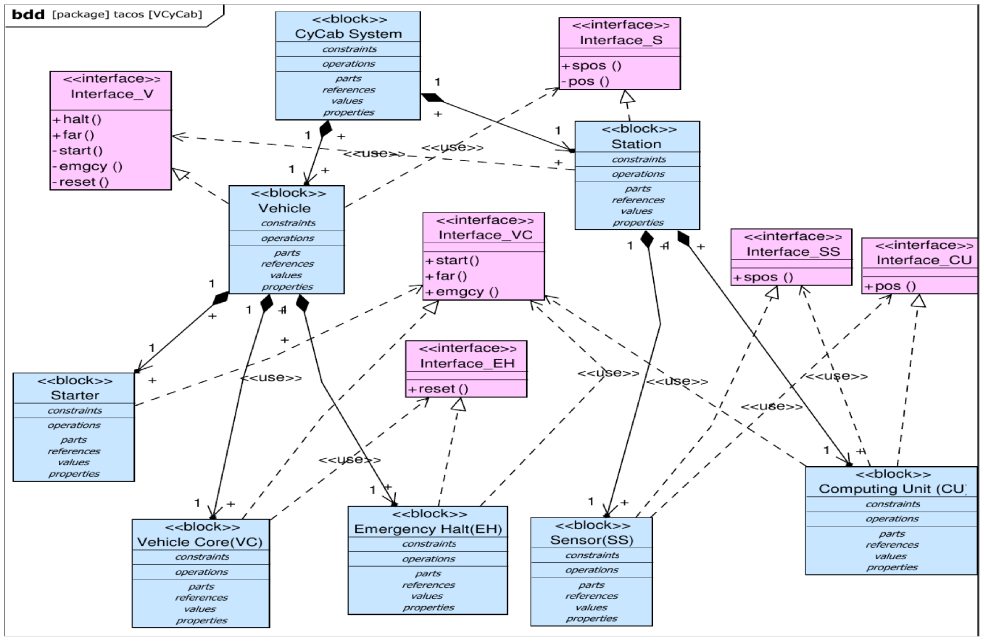


FIGURE 7.2 – Diagramme de définition de blocs du CyCab

Les *part* représentent les composants physiques du bloc alors que les ports de flux représentent les interfaces du bloc, à travers lequel il communique avec d'autres blocs. Deux types de ports sont disponibles dans *SysML ver 1.2* :

— Flow ports : Spécifie ce qui peut circuler entre les blocs

— Standard ports : Spécifie un ensemble d'interfaces requises ou fournies.

7.3.4 Le Diagramme de séquence

Le diagramme de séquence est utilisé pour représenter l'interaction entre les éléments structurels d'un bloc en tant que séquence d'échanges de messages. Dans le système CyCab, le *Vehicle* envoie des signaux *spos !* pour informer la prochaine station sur ses positions et il reçoit en retour les signaux (*far !* ou *halt !*) pour connaitre sa position (loin ou proche de la gare). Les deux composants *Sensor* (*Ss*) et *ComputingUnit* (*Cu*) sont les sous-composants de la station. Le *capteur* détecte un signal de position envoyé du véhicule et le convertit en coordonnées géographiques (*pos !*) qui seront utilisées par le *ComputingUnit* pour calculer la distance entre le véhicule et la station et décider si ils sont loin l'un de l'autre. Le véhicule est composé de trois composants primitifs : le *VehicleCore* (*Vc*), le *Starter* (*Sr*), et le composant embarqué *EmergencyHalt* (*Eh*).

Nous illustrons l'approche en ne considérant que les diagrammes de séquence correspondant au composant et aux sous-composants de la station. Les figures 7.3 et 7.4 montrent respectivement le diagramme de séquence des composants *Sensor* et *ComputingUnit*. Ces diagrammes spécifient les protocoles de composants, qui présentent l'interaction entre les composants et leur environnement. L'environnement représente les autres composants du système. Nous proposons de modéliser ces interactions de composants avec l'environnement afin d'adapter et de traduire le diagramme de séquence dans le formalisme des automates d'interface. En effet, un automate d'interface spécifie l'interaction d'un composant avec les environnements. Par conséquent, dans la section suivante, nous proposons un algorithme pour traduire ces diagrammes en automates d'interface afin de vérifier l'assemblage des composants.

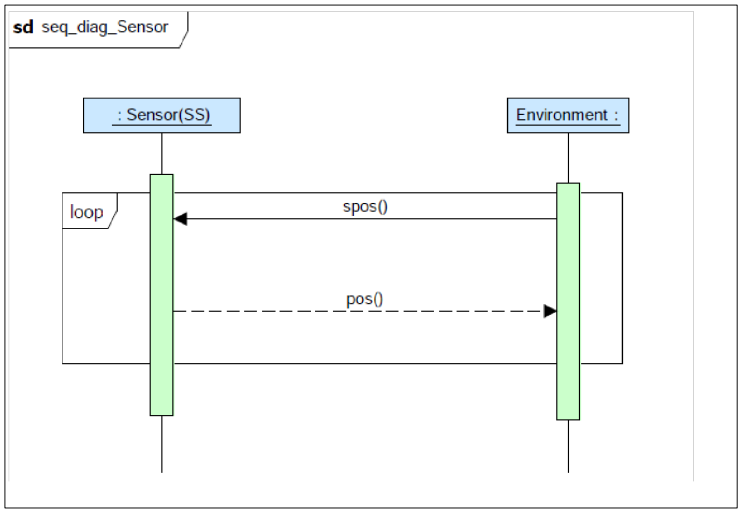


FIGURE 7.3 – Diagramme de séquence du *Sensor*

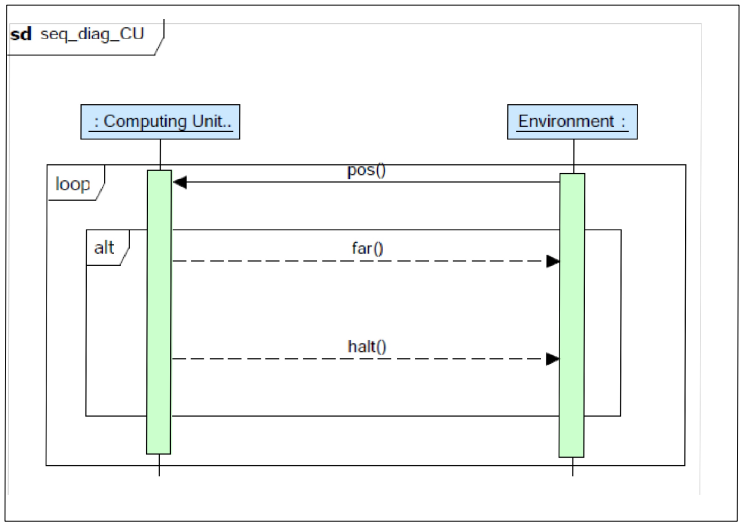


FIGURE 7.4 – Diagramme de séquence du *computingUnit*

7.4 Assemblage de composants

Dans cette section, nous proposons une approche pour générer des automates d'interface à partir de diagrammes de séquence en vue d'assembler des composants en tenant compte des spécifications SysML.

7.4.1 Dérivation d'automates d'interface à partir de diagrammes de séquence

Avant de présenter l'algorithme, nous présentons des définitions qui s'appuient sur la spécification formelle des diagrammes de séquence.

Définition 7.4.1. *(Message).*
Un message est un tuple *:*
$\langle comp_s, action, comp_f, \rangle$ *où :*

— $comp_s$ *est le composant source du message,*

— $comp_f$ *est le composant cible du message,*

— *action est la méthode invoquée.*

Nous considérons la condition suivante valide :
$\forall mes_i = \langle comp_{si}, action_i, comp_{fi}, \rangle \in Mes$ nous avons $comp_{si} = Environment$ ou $comp_{fi} = Environment$.
Cette condition est nécessaire pour translater le diagramme de séquences en automates d'interface. Dans la définition suivante, nous considérons seulement les fragments combinés *loop*, *alt* et *seq* qui sont suffisants pour la traduction en automates d'interface.

Définition 7.4.2. *(Modèle formel du diagramme de séquence).*
Le diagramme de séquence représentant le protocole d'un composant *A* est défini par *:*
$SD_A = \langle IM, Mes, Loop, Alt, Seq \rangle$,
- IM : le message initial,
- Mes : l'ensemble des messages,
- $Loop = \langle loop_1, ...loop_i, ...loop_n \rangle$*, est la liste des fragments combinés de la boucle,*
$loop_i = \langle obj_1, ..., obj_i, ..., obj_n \rangle$*,* obj_i *est un message ou un fragment, et* $card(loop_i) >= 1$
- $Alt = \langle alt_1, ...alt_i, ...alt_n \rangle$*, est la liste des fragments combinés alternatifs,*
$alt_i = \langle obj_1, ..., obj_i, ..., obj_n \rangle$*,* $card(alt_i) >= 2$
- $Seq = \langle seq_1, ...seq_i, ...seq_n \rangle$*, est la liste des fragments combinés séquentiels,*
$seq_i = \langle obj_1, ..., obj_i, ..., obj_n \rangle$*,* $card(seq_i) >= 2$

Dans ce qui suit, nous notons l'environnement spécifié dans le diagramme de séquence par *Env*, et l'unité de calcul par *Cu*. Le modèle formel correspondant au diagramme de séquence spécifiant le protocole de l'unité de calcul en bloc (voir Figure 7.4) est :
$SD_{cu} = \langle IM_{cu}, Mes_{cu}, Loop_{cu}, Alt_{cu}, Seq_{cu} \rangle$,
- IM_{cu} : Le message désigné par l'action $\langle Env, pos(), Cu \rangle$;
- Mes_{cu} : $\{\langle Env, pos(), Cu \rangle, \langle Cu, far(), Env \rangle, \langle Cu, halt(), Env \rangle\}$;
- $Loop_{cu} = \langle loop \rangle$, $loop = \langle \langle Env, pos(), Cu \rangle, alt \rangle$;
- $Alt_{cu} = \langle alt \rangle$, $alt = \langle \langle Cu, far(), Env \rangle, \langle Cu, halt(), Env \rangle \rangle$;
- $Seq_{cu} = \emptyset$.

7.4.1.1 Algorithme de dérivation

L'algorithme suivant nécessite comme entrée principale un diagramme de séquence SD_A, et aussi une liste *l* d'objets, qui sont des messages et des fragments appartenant à SD_A. L'algorithme fournit en sortie un automate d'interface *A*. Dans cet algorithme, nous utilisons les fonctions $first(l)$ et $last(l)$ qui renvoient respectivement le premier objet et le dernier objet de la liste *l*. La complexité de l'algorithme *SDtoIA* est linéaire sur la taille de l'ensemble des messages, *Mes*, dans la spécification du diagramme de séquence. En appliquant l'algorithme 3 sur le modèle formel du diagramme de séquence de l'unité de calcul ; SD_{cu}, on obtient l'automate d'interface $A_{cu} = \langle S_{A_{cu}}, I_{A_{cu}}, \Sigma^I_{A_{cu}}, \Sigma^O_{A_{cu}}, \Sigma^H_{A_{cu}}, \delta_{A_{cu}} \rangle$, décrit dans la Figure 7.5. $S_{A_{cu}} = \{1, 2\}$, $I_{A_{cu}} = \{1\}$, $\Sigma^I_{A_{cu}} = \{pos()?\}$,
$\Sigma^O_{A_{cu}} = \{halt()!, far()!\}$, $\Sigma^H_{A_{cu}} = \emptyset$,
$\delta_{A_{cu}} = \{(1, pos()!, 2), (2, halt()!, 1), (2, far()!, 1)\}$. Cet automate montre que lorsque le composant "unité de calcul" reçoit la position de CyCab, spécifiée par la transition étiquetée par l'action d'entrée $pos()?$, Elle envoie le message *halt* (transition étiquetée par l'action de sortie $halt()!$) ou far (transition marquée par l'action de sortie $far()!$) vers CyCab. L'automate d'interface du diagramme de séquence correspondant au composant de capteur est également décrit sur la Figure 7.5.

7.4.2 Modèles formels des diagrammes SysML

Dans cette section, nous proposons des modèles formels pour spécifier le diagramme de définition de blocs et le diagramme de blocs internes. Ces modèles spécifient l'architecture du système, donc nous les exploitons pour adapter l'approche des automates d'interface pour assembler les composants en fonction de l'architecture du système, spécifiée par le BDD, et les liens d'assemblage entre les composants spécifiés par l'IBD.

Algorithm 3 *SDtoIA*(SD_A, l, A)

— **ENTREES** :

— $SD_A = \langle IM, Mes, Loop, Alt, Seq \rangle$;

— l : Une liste d'objets dans SD_A,
$l = \langle obj_1, ..., obj_i, ..., obj_n \rangle$, obj_i est un message ou un fragment

— **SORTIES** : $A = \langle S, I, \Sigma^I, \Sigma^O, \Sigma^H, \delta \rangle$;

DEBUT
SI $(\Sigma^I = \emptyset \wedge \Sigma^O = \emptyset \wedge \Sigma^H = \emptyset)$ **Alors**
$I = act$ où act est une action dans le message initial,
$IM = \langle comp_i, act, comp_j \rangle$
$\Sigma^I = \{action_i | \exists mes_i \in Mes \wedge$
$mes = \langle comp_{si}, action_i, comp_{fi} \rangle \wedge comp_{fi} = Env\}$;
$\Sigma^O = \{action_i | \exists mes_i \in Mes \wedge$
$mes = \langle comp_{si}, action_i, comp_{fi} \rangle \wedge comp_{si} = Env\}$;
$\Sigma^H = \{action_i | \exists mes_i \in Mes \wedge$
$mes = \langle comp_{si}, action_i, comp_{fi} \rangle \wedge comp_{si} = comp_{si}\}$;
FIN SI
TANT QUE $(l \neq \emptyset)$ **FAIRE**
Soit obj le premier élément est dans l
SI $(obj \in Loop \cup Alt \cup seq)$
Soit l' une liste d'objets composant le fragment combiné obj
$SDtoIA(SD_A, l', A)$;
SINON
soit $t = (s, act, s')$ une transition est associée à obj ;
SI $(obj = last(loop_i))$
soit $t_f = (s_f, act_f, s'_f)$ une transition associée au premier message obj_f in $loop_i$
soit $t_p = (s_p, act_p, s'_p)$ une transition associée au message précédent obj_p in $loop_i$
$s' = s_f; s = s'_p$;
SINON SI $(obj \in \{first(loop_i), first(seq_i), first(alt_i)\})$ **ALORS**
soit $t_p = (s_p, act_p, s'_p)$ une transition associée au précédent message obj_p avant le fragment courant
$s = s'_p$;
SINON
soit $t_p = (s_p, act_p, s'_p)$ une transition associée au message précédent
$s = s'_p$;
FIN SI
$S = S \cup \{s, s'\}$; $\delta = \delta \cup t$;
FIN SI
$l = l - \{obj\}$; **FIN TANT QUE**
FIN SDtoIA

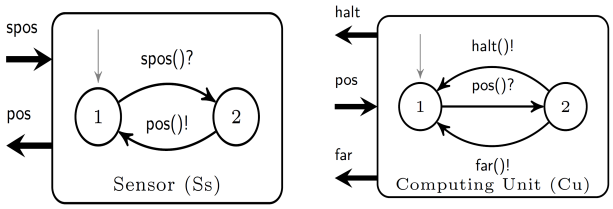


FIGURE 7.5 – L'automate d'interface du sous composant *Station*

Définition 7.4.3. *(Modèle formel des BDD).*
Un modèle formel pour le BDD d'un système S est : $BDD_s = \langle IB_s, SB_s, SubB_s \rangle$ *où :*

- IB_s *: est le bloc initial du système ;*
- SB_s *: est un ensemble de blocs ;*
- la fonction $SubB_s : SB \rightarrow 2^{SB_s}$ qui retourne pour chaque bloc un ensemble de sous-blocs.

Suivant la définition 7.4.3, le modèle formel du BDD défini dans la Figure 7.2 spécifiant le système CyCab est $BDD_{cycab} = \langle IB_{cycab}, SB_{cycab}, SubB_{cycab} \rangle$, où, $IB_{cycab} = CyCAB$, SB_{cycab} $= \{CyCab, Vehicle, Station, Starter, VehicleCore, EmergencyHalt, ComputingUnit, Sensor\}$, et nous utilisons cette fonction pour obtenir les sous-blocs *SubB*. Par exemple : $SubB_{cycab}(Vehicle) = \{VehicleCore, Emergency, Starter\}$.

Définition 7.4.4. *(Le modèle formel de l'IBD).*
Le modèle formel d'un IBD d'un système S est : $IBD_b = \langle PB_b, SI_b, L_b \rangle$ *où :*

- PB_b *: est l'ensemble des sous blocs ou (parts d'un bloc) qui composent le bloc b ;*
- SI_b *: l'ensemble des interfaces ;*
- L'ensemble $L_b = \{\{b_0, i_0, b_0'\}, ..., \{b_i, i_j, b_i'\}, ..., \{b_n, i_n, b_n'\}\}$, où $b_j \in PB_b$ et $i_j \in SI_b$, représentent les liens d'assemblage entre composants via des interfaces.

Par exemple le modèle formel de l'IBD *vehicle* est : $IBD_{vehicle} = \langle PB_{vehicle}, OI_{vehicle}, RI_{vehicle}, L_{vehicle} \rangle$ où :

- $PB_{vehicle} = \{VehicleCore, EmergencyHalt, Starter\}$,
- $L_{vehicle} = \{(VehicleCore, Interface_{vc}, Starter), (VehicleCore, interface_{EH}, EmergencyHalt)\}$,
- $SI_{vehicle} = \{Interface_{vc}, Interface_{EH}\}$.

Avec les définitions 7.4.3 et 7.4.4, nous obtenons la spécification formelle de l'architecture du système et les liens de composition entre composants (ou blocs). Et en appliquant l'algorithme *SDtoIA*, nous obtenons les automates d'interface correspondant aux diagrammes de séquence, donc, après cela nous pouvons exploiter les automates d'interface pour vérifier l'assemblage des composants du système. L'algorithme suivant, *SysCompos*, exploite les modèles formels et l'algorithme 3, pour vérifier la composition du système, spécifié par les diagrammes SysML. L'algorithme 4 exploite également l'approche des automates d'interface pour vérifier la compatibilité entre deux automates d'interface.
La complexité de l'algorithme *SysCompos* est linéaire sur la taille de l'ensemble des blocs dans le BDD, en considérant aussi la complexité de l'approche des automates d'interface (la fonction *Compos*). En appliquant l'algorithme 4 sur le bloc *Station*, avec les paramètres appropriés ($SysCompos(BDD_{cycab}, Station)$), on obtient l'automate d'interface décrit dans la figure 7.6. Celle-ci montre le protocole du composant composite obtenu *Station*, donc quand il reçoit le message $spos()?$ du véhicule en exécutant la transition étiquetée par $spos()?$, il calcule la position du véhicule, en exécutant l'action interne $pos()$ (obtenu par la synchronisation de $pos()!$ dans le composant capteur et $pos()?$ dans le composant unité de calcul), et envoie les messages *halt* ou far au véhicule composite. Pour vérifier la composition de l'ensemble du système, CyCab, il est nécessaire d'exécuter $SysCompos(BDD_{cycab}, IB_{cycab})$. Cette approche est évaluée dans la première étape sur l'exemple du système CyCab et également en fournissant la complexité des algorithmes de vérification. Dans la deuxième étape, nous prévoyons d'évaluer cette approche sur des études de cas plus réalistes en implémentant les algorithmes proposés, de manipuler les diagrammes SysML dans la vérification de l'assemblage de tout le système (composite), et aussi en exploitant l'outil *Ticc* pour vérifier la compatibilité entre les automates d'interface.

Algorithm 4 SysCompos(BDD,StartB)

— **ENTREES** :

— Le diagramme de définition de blocs (BDD),

— StartB est le premier bloc à analyser, a la valeur du bloc initial dans le BDD, IB, lorsque nous assemblons les composants de l'ensemble du système

— **SORTIE** : l'automate d'interface du composant composite si l'assemblage est correct, ou un automate d'interface vide sinon

DEBUT
Soit $IBD_s = \langle PB_s, SI_s, L_s \rangle$ le bloc interne de $StartB$
$setB = PB_s$; $SetL = L_s$;
REPETER
Soit $sb_i \in setB$,
SI $(SuB(sb_i) \neq \emptyset)$ **ALORS**
$IA_{sb_i} = SysCompos(BDD, sb_i)$;
SINON
soit SD_{sb_i} le diagramme de séquence de sb_i
nous appliquons la fonction $SDtoIA$ pour obtenir l'automate d'interface IA_{sb_i}
FIN SI
TANT QUE$(\exists\{sb_i, i, sb_i'\} \in SetL)$ **FAIRE**
SI $(SuB(sb_i') \neq \emptyset)$ **ALORS**
$IA_{sb_i'} = SysCompos(BDD, sb_i')$
SINON
soit $SD_{sb_i'}$ le diagramme de séquence de sb_i'
nous appliquons la fonction $SDtoIA$ pour obtenir l'automate d'interface $IA_{sb_i'}$
FIN SI
$IA_{sb_i} = Compos(IA_{sb_i}, IA_{sb_i'})$ //appliquer l'approche de l'automate d'interface
SI $(IA_{sb_i} = \emptyset)$ **ALORS** Exit() ;**FIN SI**
$SetL = SetL - \{sb_i, i, sb_i'\}$;
FIN TANT QUE ;
$setB = setB - sb_i$;
JUSQU'A $setB = \emptyset$;
RETOUR IA_{sb_i};
FIN

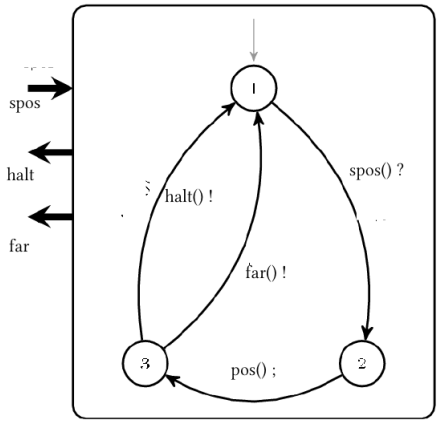


FIGURE 7.6 – L'automate d'interface du composant Station

7.5 Etat de l'art

Comme vu précédemment, plusieurs travaux existent déjà dans la dérivation de modèles formels à partir des modèles semi-formels comme UML. Notre travail est plus particulièrement orienté vers l'utilisation conjointe de SysML et des modèles formels en vue de vérifier l'assemblage des composants.

Dans [239], les auteurs proposent une syntaxe et une sémantique formelle pour les diagrammes d'activité SysML. La contribution principale consiste à définir un langage appelé Activity Calculus (AC), avec une sémantique opérationnelle formelle. De plus, la définition d'une sémantique formelle des diagrammes d'activité SysML permet une analyse rigoureuse de la conception et la découverte des erreurs de conception au début du processus de développement. Les auteurs ont l'intention, en tant que travaux futurs, de s'appuyer sur le formalisme actuel et de développer davantage la vérification probabiliste du modèle des diagrammes d'activité SysML.

Dans [240], les auteurs ont proposé une approche de modélisation de systèmes qui combine des méthodes semi-formelles basées sur les exigences SysML, des diagrammes de blocs et des méthodes formelles telle que la vérification du modèle pour prouver que le comportement local de chaque composant du système contribue à satisfaire les exigences du système. Ce travail est basé sur la mise en correspondance entre les modèles structurés des exigences à satisfaire, les fonctions à réaliser et le composant à mettre en oeuvre.

Dans [241], les auteurs décrivent un ensemble de règles de traduction de la spécification basée sur SysML vers des réseaux de Petri colorés (CPN) qui permettent l'analyse dynamique du système ainsi que la vérification formelle du comportement et de la fonctionnalité du design basé sur SysML.

Dans [242], les auteurs proposent une solution pour la modélisation et la vérification de systèmes embarqués en temps réel avec des contraintes d'énergie. Pour cela, ils combinent les fonctionnalités des modèles SysML et des annotations du profil MARTE avec les avantages d'utiliser l'outil Time Petri Net. Ce formalisme permet l'analyse et la vérification des besoins fonctionnels, temporels et énergétiques dans les premières phases du développement.

Dans [119], les auteurs présentent TEPE (TEmporal Property Expression), un langage graphique basé sur des diagrammes paramétriques SysML. Les propriétés sont construites sur des relations logiques et temporelles entre les attributs et les signaux des blocs. TEPE peut être intégré à un profil temps réel de SysML. De plus, AVATAR est associé à une méthode centralisée de vérification supportée par la boîte à outils open source TTool. Elle comprend un éditeur de diagramme, un générateur de code UPPAAL et une interface de bouton-poussoir pour la vérification formelle.

L'originalité de notre approche est la connexion formelle entre SysML et le formalisme des interfaces, pour exploiter et améliorer l'approche basée sur des automates d'interface pour vérifier l'interopérabilité dans les systèmes basés sur les composants.

7.6 Bilan

Ce travail a donné lieu :

1. **Projets**
   — 2006-2010 *TACOS (Trustworthy Assembling of Components : frOm requirements to Specification)*.
   Partenaires : LACL (Université Paris-Est Créteil), LORIA(University de Nancy), LAMIH (Université de Valenciennes).
   Dans ce projet, je me suis investi dans l'expression et la formalisation des exigences et le calcul du WCET dans le composant de localisation du véhicule.
2. **Thèse de doctorat UFC** [243].

Année	Candidat	Titre
2016	H. Bouaziz	*Adaptation of SysML Blocks and Verification of Temporal Properties.*

TABLE 7.1 – Thèse sur les composants

3. **Communications suivantes :** . [73, 72, 71, 70, 69, 68, 244, 245, 67]

Troisième partie

Synthèse, directions de recherche et perspectives

8

Synthèse, directions de recherche et perspectives

Plan du chapitre

8.1 Introduction

Traditionnellement, la phase de rédaction du rapport d'habilitation à diriger les recherches est l'occasion de prendre du recul par rapport au travail en cours pour faire une synthèse et une analyse sur tout ce qui a été réalisé dans le domaine de la recherche. Il me paraît évident que si je reproduisais la synthèse de tous mes travaux issus de mon parcours dans ce document, ceci apparaîtrait incohérent et son contenu éclaté. Ainsi, il ne me semble pas pertinent de faire cohabiter, dans un même document, mes travaux sur les environnements informatiques d'aide à l'apprentissage des langues, mes travaux sur les l'intégration des modèles de base de données et ceux de l'analyse des données d'une part et les modèles pour la spécification et la validation des systèmes réactifs d'autre part. En conséquence, dans ce document, j'ai présenté une synthèse de mes travaux uniquement depuis 1999, année de mon intégration au sein du laboratoire LIFC.

8.2 Utilisation conjointe de UML et B

8.2.1 Synthèse

Dans la première partie de ce document, j'ai présenté mes contributions concernant l'utilisation conjointe des langages formels et semi-formels. Ce travail, mérite d'y figurer car il fait partie des premières applications des techniques d'IDM, il ne concerne pas la génération d'un code formel à partir d'une spécification semi-formelle mais l'inverse, ceci dans le but de documenter et donc de rendre compréhensible la spécification formelle et l'aide à la validation externe de développements formels. Notre intérêt s'était porté précisément sur la méthode B qui est une méthode formelle utilisée pour modéliser des systèmes et prouver l'exactitude de leur conception par raffinements successifs. Mais les spécifications formelles sont difficiles à lire quand elles ne sont pas accompagnées d'une documentation. Cette lisibilité est essentielle pour une bonne compréhension de la spécification, notamment dans des phases de validation ou de certification. Aujourd'hui, en B, cette documentation est fournie sous forme de texte, avec, quelquefois, des schémas explicitant certaines caractéristiques du système. L'objectif de ce travail était de mettre en relation des spécifications en B avec des diagrammes UML, qui constituent un standard dans le monde industriel et dont le caractère graphique améliore la lisibilité. Nous avons axé notre processus d'extraction de diagrammes de classes à partir de spécifications B autour d'une technique d'ingénierie inverse guidée par un ensemble de correspondances structurelles et sémantiques spécifiées à un méta-niveau. L'extraction des diagrammes d'états-transitions (DET) est inspirée des travaux de D. Bert [246] sur la construction de systèmes de transition à éléments finis à partir de systèmes B abstraits. Le DET extrait prend en compte les raffinements successifs des machines B. Dans l'extraction du diagramme de classe, nous avons considéré la machine B abstraite comme une super classe et ses raffinements comme des classes qui héritent d'elle.

8.2.2 Directions de recherche et perspectives

Dans ce chapitre, nous avons abordé notre problématique de recherche dans une optique de complétude et d'homogénéité. Pour ce faire, nous avons exploré plusieurs voies orientées vers deux types de diagrammes : diagrammes structurels et diagrammes comportementaux. Nous avons présenté les correspondances entre les variables utilisées dans le langage B et les concepts utilisés dans les diagrammes de classes et d'états-transitions UML. Nous avons montré la possibilité d'extraire ses deux diagrammes UML en suivant la démarche de spécification de la méthode B avec les différents raffinements jusqu'à à l'implémentation. Nous pensons aussi que l'extraction d'autres diagrammes comme les diagrammes de séquence et d'activité est importante et permettrait d'avoir des vues complémentaires de celles précédemment définies. Une chaîne outillée pour supporter ces extractions est très intéressante. La différence des niveaux d'abstraction entre le modèle UML et le modèle B rend le passage entre le modèle B et UML de plus en plus complexe. Pour pallier ce problème, la création d'un modèle intermédiaire appelé B2UML sous forme de profil UML permettrait de prendre en compte des variables spécifiques B est fort utile. Dans ce cas, nous pensons dans un premier temps

extraire d'abord les diagrammes B2UML directement de la spécification B, pour préserver la sémantique de la spécification B. Une autre perspective serait la mise en relation entre les langages SysML et B, il serait intéressant d'intégrer ce travail à celui déjà amorcé dans [226] pour la prise en compte d'exigences SysML.

Je pense qu'il serait intéressant de dériver d'autres diagrammes notamment les diagrammes de séquence et d'activité pour avoir d'autres vues complémentaires du système. Je pense, à court terme, que la dérivation des diagrammes SysML à partir de B est fortement conseillée pour intégrer notre démarche méthodologique basée sur SysML.

Une autre piste consisterait à utiliser fUML[247], celui-ci produit un modèle UML exécutable dont la spécification comportementale est suffisamment détaillée pour pouvoir être exécutée efficacement en tant que programme. Cela peut être extrêmement utile pour valider le modèle extrait d'une spécification B, indépendamment de la ou des plates-formes de mise en oeuvre sur lesquelles le système en cours de modélisation sera finalement déployé.

8.3 Raffinement d'automates hiérarchiques

8.3.1 Synthèse

Dans la seconde partie, je me suis intéressé à la vérification de systèmes réactifs par raffinements. Ces systèmes sont modélisés à l'aide d'automates hiérarchiques définis dans [23]. Les techniques de vérification utilisées sont algorithmiques, et exploitent la notion de hiérarchie. Les propriétés sont exprimées à l'aide de la logique *LTL*. Nous avons proposé le raffinement d'automates hiérarchiques comme solution pour vérifier les systèmes hiérarchiques de grande taille. La démarche consiste à introduire pas à pas des détails supplémentaires sur le système en éclatant des états en d'autres comportements. Cette approche permet de décrire des spécifications abstraites de plus petite taille sur lesquelles on vérifie les propriétés exprimables au niveau des détails considérés. Puis le système est raffiné par introduction de nouveaux détails sous forme de composants qui peuvent être des automates ou un ensemble d'automates parallèles. Nous avons défini les conditions de raffinement entre automates hiérarchiques de telle sorte qu'elles garantissent le raffinement de leurs modèles sémantiques qui sont des structures de Kripke. Nous avons montré que ce raffinement préserve les propriétés temporelles satisfaites au niveau abstrait. Autrement dit, si une propriété *LTL* est vérifiée au niveau abstrait alors elle l'est également au niveau raffiné. Nous avons montré que l'opération de raffinement préserve sur le système raffiné les propriétés vérifiées sur le système abstrait.

Dans la thèse de M. Al Achhab [24], une relation de raffinement limitée à des nouveaux automates acycliques a été définie. Nous avons prouvé que ce raffinement permet de préserver les propriétés *LTL* vérifiées au niveau abstrait. L'hypothèse de l'absence de cycles dans les nouveaux automates est assez forte et permet de restreindre la classe des applications. Nous envisageons d'étendre la préservation de ces propriétés aux systèmes en présence de cycles.

8.3.2 Directions de recherche et perspectives

Nous pensons qu'il serait intéressant d'étendre la démarche, dans un premier temps, vers les StateCharts. L'approche consisterait, par exemple, à partir d'une spécification UML/SysML (notamment les StateCharts de produire une représentation équivalente en automates hiérarchiques étendus (tenant compte de l'aspect hiérarchie). Diverses méthodes et techniques ont été introduites afin d'effectuer la vérification de propriétés par *model-checking* sur les modèles hiérarchiques. Dans [111], les auteurs montrent comment les StateCharts peuvent être traduits en *Promela* (le langage de spécification qu'utilise SPIN [112]) en utilisant *les automates hiérarchiques* comme format intermédiaire.

Nous pensons qu'il serait intéressant d'utiliser la vérification locale des nouvelles propriétés définies sur les systèmes raffinées, liées seulement aux nouveaux automates introduits dans le processus de raffinement. Ces propriétés peuvent être vérifiées localement sur les nouveaux sous-automates introduits par l'opération de raffinement. Cette partie, peut être étendue par les relations d'exigences SysML.(Verify). L'adaptation de ces travaux aux modèles SysML est une nécessité, je pense définir une relation entre un sous-ensemble des diagrammes d'états transitions SysML et les automates

hiérarchiques, je pense également exploiter les liens entre exigences et éléments du modèle pour valider la vérification locale des nouvelles propriétés et le raffinement pour valider la correction des compositions des diagrammes de blocs SysML.

8.4 Démarche de validation par vérification formelle

8.4.1 Synthèse

Dans ce travail, nous avons proposé une démarche pour la spécification et la vérification de systèmes. Dans cette démarche, les approches proposées pour la vérification s'appliquent très tôt dans le processus de développement du système. Cette intervention précoce permet de limiter les coûts de correction d'éventuelles erreurs de spécification et de conception. La démarche proposée utilise le langage SysML pour spécifier le système et ses exigences et sur les méthodes formelles pour la phase de vérification. Le choix du langage SysML est justifié par le fait qu'il est le langage dédié à l'ingénierie système basée sur les modèles. C'est un langage graphique qui se base sur UML et hérite de ses avantages. De plus, il supporte l'allocation qui permet d'associer par exemple des éléments structurels à des éléments comportementaux et il offre la possibilité de modéliser les exigences et de les lier aux autres modèles de conception. Le choix de la vérification formelle est due à la rigueur des techniques proposées dans ce cadre pour vérifier et valider des systèmes critiques. Dans notre démarche basée SysML, le diagramme d'exigences est utilisé pour modéliser les exigences, la structure du système est décrite par des diagrammes de blocs et enfin, les diagrammes d'activités sont utilisés pour décrire le comportement du système. Comme le langage SysML ne propose aucune formalisation des exigences et propose seulement leur description par des textes écrits en langage naturel, nous avons proposé une étape de formalisation des exigences SysML. Celle-ci concerne les exigences fonctionnelles à vérifier sur des comportements modélisés par des diagrammes d'activités. La formalisation est faite en exprimant les exigences en ActRL, un langage de haut niveau que nous avons proposé pour permettre de décrire des exigences SysML en se basant sur les éléments du diagramme d'activités.
Nous avons proposé des passages automatiques à partir des diagrammes SysML aux réseaux de Petri. Le but étant d'obtenir des spécifications formelles, puis d'exploiter les possibilités et les mécanismes offerts par les méthodes de vérification basées sur les modèles pour vérifier les exigences SysML sur les diagrammes d'activités. Notre approche de vérification se base sur les réseaux de Petri hiérarchiques colorés, nous avons proposé une transformation automatique d'un sous- ensemble des éléments des digrammes d'activités vers les réseaux de Petri hiérarchiques colorés. Nous avons proposé aussi, une translation des exigences SysML, formalisées en ActRL, vers la logique ASK-CTL, une variante de la logique CTL implémentée dans l'outil CPN Tools. Ce dernier a été utilisé pour vérifier les propriétés ASK-CTL, issues de la translation des exigences SysML, sur les réseaux de Petri hiérarchiques colorés, issus de la translation des diagrammes d'activités.
La démarche de spécification et de vérification que nous avons proposée dans ce travail vise à aider les concepteurs à spécifier et à vérifier des systèmes critiques. Cependant, comme elle fait appel à la vérification basée sur les modèles, elle hérite de ses limites. En particulier, le problème de l'explosion combinatoire de l'espace d'états limite l'application de la démarche proposée pour vérifier des systèmes de grandes tailles. Dans le but d'élargir le champ d'application de la démarche de spécification et de vérification proposée, nous avons présenté une solution pour pallier le problème de l'explosion combinatoire de l'espace d'états, nous avons développé une idée permettant d'exploiter les relations entre exigences, blocs et activités, qui sont modélisés en SysML, pour découper le modèle et limiter l'effet du problème de l'explosion combinatoire de l'espace d'états. Pour une exigence donnée, la partie du modèle à considérer est celle qui est nécessaire à la vérification de cette exigence.

8.4.2 Directions de recherche et perspectives

8.4.2.1 Formalisation des exigences et Langage AcTRL

Nous avons montré l'utilité de formaliser les exigences pour automatiser leur vérification. Cependant, nous avons considéré seulement les exigences fonctionnelles concernant des comportements

modélisés par des diagrammes d'activités à travers la proposition du langage AcTRL. Comme perspective, nous proposons tout d'abord d'étendre le langage AcTRL pour couvrir tous les éléments du diagramme d'activités, comme les exceptions et les régions interruptibles, pour permettre d'exprimer des exigences relatives à ces éléments. Puis, l'étendre pour exprimer des exigences non fonctionnelles en considérant les probabilités introduites dans les diagrammes d'activités SysML. Ces extensions doivent être supportées par un éditeur d'exigences, intégré dans des plate-formes de modélisation SysML.

8.4.2.2 Approches de vérification

Dans ce travail, nous nous sommes focalisés sur la vérification des exigences par des diagrammes d'activités. Les approches proposées ne couvrent qu'un sous-ensemble des éléments du diagramme d'activités. Nous pensons aussi qu'il est important de considérer la vérification des exigences sur d'autres diagrammes SysML. Ceci va permettre la validation des différentes vues du système. Une autre perspective concerne l'interprétation des résultats de vérification sur la modélisation SysML, qui constitue une suite logique de ce travail.

8.4.2.3 Application de la démarche proposée

Les autres aspects à envisager concernent l'élargissement du champ d'application de la démarche proposée par l'approfondissement des propositions présentées dans le cadre de ce travail et leur combinaison avec d'autres solutions comme la vérification symbolique et les techniques de réduction. Une autre perspective assez importante concerne l'implémentation de la démarche de spécification et de vérification. Cet aspect a été considéré par le développement d'outils pour la transformation des diagrammes d'activités en réseaux de Petri hiérarchiques colorés. Il est intéressant d'améliorer ces outils et de les intégrer dans des plate-formes supportant le langage SysML afin de fournir des moyens d'assistance robustes et conviviaux aux concepteurs de systèmes critiques et complexes.

8.5 IDM et les systèmes complexes

Dans cette partie, La démarche que nous avons proposé définit une méthodologie qui englobe les phases d'analyse des exigences, de modélisation, de validation et de vérification formelle. Elle propose d'introduire ces pratiques (analyse, modélisation, validation et vérification des exigences) dès les premières phases de développement et de l'intégrer à celle-ci dès les premières étapes de conception (spécifier pour bien valider). L'objectif est d'aider à la découverte (et ainsi à la correction) des erreurs de spécification dès les premières phases de développement.
En ingénierie, la validation des exigences et de leurs évolutions sur tout le cycle de vie d'un projet matériel, logiciel ou mixte, permet d'élaborer une compréhension commune de tous les acteurs concernés par les exigences du système a développer. Les exigences sont la base de l'entente client-fournisseur, elles permettent de mesurer précisément la conformité du système ou du logiciel par rapport aux besoins ou aux contraintes. La première étape consiste à analyser les exigences et à déterminer celles qui portent sur la qualité du système (dites non fonctionnelles) et celles (fonctionnelles) sur les fonctions du système. Notre démarche mettra les exigences du système à concevoir au centre du processus de conception.
Notre ambition est de définir une démarche méthodologique pour la spécification et la validation des systèmes, tout en s'appuyant sur un langage de modélisation permettant la traçabilité entre les exigences et les éléments décrivant les différentes vues du système à développer. Nous avons proposé une méthodologie basée sur SysML s'appuyant sur une description précise des exigences et l'exploitation des relations entre exigences et éléments du modèle SysML. Elle intègre la traçabilité des exigences et les techniques du Model Driven Engineering (MDE) pour la transformation des modèles SysML vers des environnements de simulation (Modelica, VHDL-AMS, SystemC,..) et de vérification (Uppaal, SPIN, TINA,...).
Les exigences sur les systèmes complexes peuvent être fonctionnelles ou non. Les exigences non fonctionnelles peuvent être validées par simulation en transformant les diagrammes SysML, notamment le diagramme paramétrique, vers des environnement de simulation. Les exigences fonction-

nelles seront validées par vérification formelle, en exprimant les propriétés afférentes avec le langage PSL.

8.5.1 Directions de recherche et perspectives

Lorsqu'on développe une nouvelle démarche méthodologique, la construction d'outils logiciels pour la supporter se pose rapidement. Nous pensons que l'utilisation de notre démarche pour développer et valider cet outil constituera le premier test pour notre méthodologie. Pour compléter cette phase de tests, nous pensons solliciter nos collègues d'autres départements de l'institut pour participer à la modélisation et la validation d'applications critiques s'appuyant sur des composants MEMS ou sur des applications où l'exigence énergétique est critique. Dans notre introduction, nous avons exprimé le souhait de combiner les techniques de test et de simulation en vue de valider des systèmes. La vérification virtuelle utilisée dans l'exemple des réseaux de capteurs constitue la technique de test. Le test désigne une procédure de vérification partielle d'un système. Son objectif principal est d'identifier un nombre maximum de comportements problématiques du système afin d'en augmenter la qualité (si les problèmes identifiés lors des tests sont corrigés). Néanmoins, le test peut aussi avoir pour objectif d'apporter des informations quant à cette qualité afin de permettre la prise en décisions. Un test ressemble à une expérience scientifique. Il examine une hypothèse exprimée en fonction de trois éléments : les données en entrée, l'objet à tester et les observations attendues. Cet examen est effectué sous conditions contrôlées pour pouvoir tirer des conclusions. La coopération entre test et simulation consiste en l'utilisation par exemple des données de simulation comme entrée des batteries de tests et inversement.

8.6 Vérification de l'assemblage de composants SysML

8.6.1 Synthèse

Le travail présenté dans ce chapitre a lieu dans le domaine de développement basé sur les composants, il est une contribution à la spécification, l'adaptation et la vérification des systèmes à base de composants. Le but principal de ce travail est la proposition d'une approche formelle pour construire progressivement des systèmes complexes en assemblant et en adaptant un ensemble de composants, où leur structure et leur comportement sont modélisés à l'aide de diagrammes SysML. Dans la première étape, nous avons défini une approche basée sur la méta-modélisation et la transformation des modèles pour vérifier la compatibilité des blocs ayant leurs protocoles d'interaction modélisés à l'aide de diagrammes de séquence SysML. Pour vérifier leur compatibilité, nous effectuons une transformation en automates d'interface (IAs), et nous utilisons l'approche optimiste définie sur les IA. Cette approche considère que deux composants sont compatibles s'il existe un environnement approprié avec lequel ils peuvent interagir correctement. Dans ce travail, à l'aide d'un exemple, nous avons décrit comment *SysML* peut être utilisé pour définir des modèles qui peuvent être traduits en automates d'interface. Nous avons proposé une transformation du diagramme de séquence en automates d'interface, ce qui est réalisé par une construction algorithmique. Nous avons considéré une communication de composant avec l'environnement qui pourrait être le reste du système. Nous avons également proposé des modèles formels des diagrammes de définition de bloc et de bloc interne. Le diagramme de définition de bloc nous informe de l'architecture du système et du schéma fonctionnel interne, spécifie les liens d'assemblage entre les composants. Nous avons proposé un algorithme pour vérifier l'assemblage des composants en considérant l'architecture du système et en exploitant les automates d'interface.

Ce travail a servi comme point de départ de la thèse de H. Bouaziz [243], qui a étendu ce travail en considérant notamment la hiérarchie, qui peut être présente dans les modèles de protocole d'interaction des blocs, pour alléger la vérification de la compatibilité des blocs.

8.6.2 Directions de recherche et perspectives

Pour les travaux futurs, nous avons l'intention d'utiliser d'autres diagrammes *SysML* pour les différentes vues du système afin de prendre en compte les propriétés non-fonctionnelles telles que les

performances, les économies d'énergie lors de l'assemblage composants. Nous prévoyons également de considérer l'assemblage des composants matériels. De plus, nous prévoyons de concevoir et mettre en oeuvre un cadre pour la dérivation automatique des modèles sémantiques et les mapper dans les automates d'interface. Ce travail peut servir comme point de départ pour valider la cohérence des diagrammes SysML, notamment la compatibilité des blocs SysML. Un outil, intégré comme un plugin aux plate-formes de modélisation comme eclipse, vérifiant l'assemblage des blocs SysML serait d'un apport considérable pour les concepteurs.

8.7 Perspectives globales

Mes perspectives de recherche se concentrent sur les points suivants :

CASE pour les systèmes complexes : Afin d'assurer la correction des systèmes complexes et critiques, dès les premières phases de leur développement, il est essentiel de définir un processus de conception qui inclut les phases d'analyse, de spécification, de validation et de vérification. Ces phases doivent être étroitement liées et doivent se combiner de façon itérative. Nous visons à tirer profit des concepts émergents portés par l'ingénierie des systèmes basée sur les modèles semi-formels, combinée à des méthodes de validation et de vérification formelles. Lorsqu'on développe une nouvelle démarche méthodologique, la construction d'outils logiciels pour la supporter se pose rapidement. L'objectif est de mettre à la disposition des chercheurs de notre institut, un guide méthodologique outillé, pour la spécification, la conception et le développement de leurs systèmes. Nous pensons que l'utilisation de notre démarche pour développer et valider cet outil constituera le premier test pour notre méthodologie. Pour compléter cette phase de tests, nous comptons solliciter nos collègues d'autres départements de l'institut pour expérimenter notre démarche en contribuant à la modélisation et la validation d'applications critiques s'appuyant sur des composants MEMS ou sur des applications où les ressources énergétiques sont critiques.

Raffinement d'exigences : la première étape dans notre méthodologie est l'étape de spécification des exigences. Le rôle de cette étape est la définition et la modélisation des exigences après l'analyse des besoins décrits dans le cahier des charges. En ingénierie, la validation des exigences et de leurs évolutions sur tout le cycle de vie d'un projet matériel, logiciel ou mixte, permet d'élaborer une compréhension commune de tous les acteurs concernés par les exigences du système a développer. Les exigences permettent de mesurer précisément la conformité du système ou du logiciel par rapport aux besoins ou aux contraintes. Nous comptons enrichir notre démarche en analysant succinctement les exigences SysML. La première étape consistera à les formaliser pour les exprimer de la façon la plus précise pour enlever toute ambiguïté. La formalisation des exigences est une étape nécessaire pour leur vérification. Dans le chapitre 5, nous avons proposé le langage formel AcTRL pour formaliser les exigences qui sont liées par la relation *verify* avec les diagrammes d'activités. Il serait intéressant d'étendre ce langage pour prendre en compte les autres catégories d'exigences. Cette étape peut servir aussi à valider et/ou vérifier formellement les opérations sur les exigences notamment le raffinement. La relation *Refine* de SysML consiste à détailler une exigence par une autre ou par un élément de modélisation. Cette opération est très importante pour la validité du système car elle permet de garder la traçabilité entre les exigences et les éléments du modèle. La vérification formelle du raffinement d'exigences SysML n'est pas encore abordé dans la littérature. Je compte m'investir dans cette voie pour proposer une démarche de vérification de la correction de la relation *Refine*. Ensuite, la généralisation de cette démarche aux autres relations *Derive*, *Contains, satisfy,* .. est plus que nécessaire.

Cohérence des diagrammes SysML : La spécification des systèmes peut être abordée après analyse du cahier des charges et des échanges avec les parties prenantes concernées. Cependant, lors de la spécification des systèmes complexes et critiques, une attention particulière doit être accordée à la validité des modèles produits. La spécification d'un système consiste à créer à partir des besoins décrits dans le cahier des charges des modèles d'exigences et des modèles de conception. Les modèles formels liés aux modèles semi-formels peuvent contribuer à la validation de ces derniers. Comme exemple, nous pouvons valider la cohérence d'un diagramme

de blocs SysML en vérifiant la compatibilité des blocs/composants lors de leur assemblage . Les diagrammes d'états transitions peuvent être validés également en utilisant le raffinement d'automates hiérarchiques défini dans le chapitre 3.

Correction des spécifications SysML : L'interprétation des résultats de vérification sur la spécification de haut niveau (spécification SysML) constituera la dernière étape dans notre démarche de spécification et de vérification. La vérification des propriétés est réalisée dans un environnement incluant des techniques formelles difficilement exploitables par un utilisateur non initié. L'utilisation des règles d'extraction définies dans le chapitre 2, ou un reverse engineering vers le langage semi-formel (SysML) sera d'une grande utilité. Cette étape d'interprétation de résultats de vérification reposera sur l'expertise humaine. Elle doit permettre de détecter l'origine de l'erreur dans la spécification SysML (erreur dans le modèle d'exigences ou dans les modèles de conception).

Bibliographie

[1] Anu Maria. Introduction to Modeling and Simulation. In *Proceedings of the 29th Conference on Winter Simulation*, WSC '97, pages 7–13, Washington, DC, USA, 1997. IEEE Computer Society. :1997 :IMS :268437.268440

[2] OMG systems Modeling Language (OMG SysML). `http://www.omgsysml.org/`. Accessed 2018-02-20. 5

[3] Friedenthal, S. and Moore, A. and Steiner, R. *A Practical Guide to SysML : The Systems Modeling Language*. Morgan Kaufmann, 2009. ISBN 9780123743794. 5

[4] Ahmed, S.R. and Ramm, G. and Faltin, G. Some Salient Features Of The Time-Averaged Ground Vehicle Wake. In *SAE Technical Paper*. SAE International, 02 1984. 6, 8, 73

[5] The Unified Modelling Language - UML. `http://www.uml.org`. Accessed : 2018-02-20. 6, 15

[6] J. Rumbaugh, I. Jacobson, and G. Booch. *"The Unified Modeling Language Reference Manual"*. Addison Wesley, 1998. ISBN 0-201-30998-X. 6

[7] Sebastien Gerard and Bran Selic. The UML MARTE standardized profile. In *Proceedings of the 17th IFAC World Congress*, volume 14, 2008. 6, 47, 67

[8] Object Management Group. *UML Profile for MARTE, draft revised submission*. Number number realtime/07-03-03L4.1. April 2007. 6

[9] Abrial, J.-R. *The B-book : Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996. :1996 :BAP :236705

[10] Jean-Raymond Abrial. On B. In Didier Bert, editor, *B'98 : Recent Advances in the Development and Use of the B Method, Second International B Conference, Montpellier, France, April 22-24, 1998, Proceedings*, volume 1393 of *Lecture Notes in Computer Science*, pages 1–8. Springer, 1998. :conf/b/Abrial98

[11] Jean-Christophe Voisinet. *Contribution aux processus de développement d'applications spécifiées à l'aide de la méthode B par validation utilisant des vues UML et traduction vers des langages objets*. PhD thesis, 2004. Thèse de doctorat dirigée par Julliand, Jacques Automatique et informatique Besançon 2004. 7, 28

[12] Bruno Tatibouet, A. Requet, Jean-Christophe Voisinet, and Ahmed Hammad. Java Card Code Generation from B Specifications. In *5th International Conference on Formal Engineering Methods (ICFEM'2003)*, volume 2885 of *Lecture Notes in Computer Science (LNCS)*, pages 306–318, Singapore, nov 2003. :ip

[13] Ahmed Hammad, Bruno Tatibouet, and Jean-Christophe Voisinet. De la spécification B vers les diagrammes d'états-transitions UML. In *7ème MCSEAI 2002 (Maghrebian Conference of Software Engineering and Intelligence Artificial)*, pages 133–143, Annaba, Algeria, may 2002. :np

[14] Ahmed Hammad, Bruno Tatibouet, Jean-Christophe Voisinet, and Wu Weiping. From a B Specification to UML Statechart Diagrams. In *4th International Conference on Formal Engineering Methods (ICFEM'2002)*, volume 2495 of *Lecture Notes in Computer Science (LNCS)*, pages 511–522, Shanghaï, China, oct 2002. :ip

[15] M. Hariti, Noureddine Zerhouni, and Ahmed Hammad. Conception d'une base de données temps réels pour un système de production à partir du langage UML. In *Actes des Journées d'Informatique pour l'Entreprise, JIEO1'02*, pages 233–241, Université de Blida, Algeria, mar 2002. :np

[16] Bruno Tatibouet and Ahmed Hammad. Génération de diagrammes de classes UML à partir de machines abstraites B. In *Actes des Journées d'Informatique pour l'Entreprise, JIEO1'02*, pages 6–17, Université de Blida, Algeria, mar 2002. :np

[17] Bruno Tatibouet, Ahmed Hammad, and Jean-Christophe Voisinet. From an abstract B specification to UML class diagrams. In *2nd IEEE International Symposium on Signal Processing and Information Technology (ISSPIT'2002)*, pages 5–10, Marrakech, Maroc, dec 2002. :ip

[18] Jean-Christophe Voisinet, Bruno Tatibouet, and Ahmed Hammad. jBTools : An experimental platform for the formal B method. In *Principles and Practice of Programming in Java (PPPJ'02)*, pages 137–140, Trinity College, Dublin, Ireland, jun 2002. :ip

[19] Ahmed Hammad and Bruno Tatibouet. Spécifications formelles et semi-formelles : l'exemple du robot typé. In *Fifth International Symposium on Programming and Systems (ISPS'2001)*, pages 229–240, Algiers, Algeria, may 2001. :np

[20] Ahmed Hammad and Bruno Tatibouet. Modélisation orientée objet et spécification formelle d'une application industrielle. In *Conférence Internationale sur la Productique, CIP'2001*, pages 50–54, Algiers, Algeria, jun 2001. :np

[21] Ahmed Hammad and Bruno Tatibouet. Formal and visual specification language. In *Information System Engineering (ISE'2001)*, pages 173–179, Las Vegas, United States, jun 2001. :ip

[22] Bruno Tatibouet and Ahmed Hammad. Une Utilisation conjointe de B et UML sur l'étude de cas d'un robot. In *2ème conférence d'ingénierie Système, AFIS 2001*, pages 285–290, Toulouse, France, jun 2001. :np

[23] Erich Mikk, Yassine Lakhnechi, and Michael Siegel. *Hierarchical automata as model for statecharts*, pages 181–196. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. 7, 31, 33, 35, 107

[24] Mohammed AL Achhab. *Vérification des systèmes hiérarchiques et préservation de propriétés linéaires par raffinement*. PhD thesis, Université de Franche Comté, 12 2006. 7, 31, 39, 43, 107

[25] Ahmed Hammad and Hassan Mountassir. Heuristics to verify LTL properties of hierarchical systems. In *VECoS'2008, 2nd Int. Workshop on Verification and Evaluation of Computer and Communication Systems*, eWiC - electronic Workshops in Computing, pages 1–10, Leeds, United Kingdom, jul 2008. :ip

[26] Mohammed Al'Achhab, Ahmed Hammad and Hassan Mountassir. Spécification et Validation d'un Contrôleur de Performances Sportives. In *E-Medisys 07, int. conf. on E-Medical Systems*, pages 173–178, Fez, Morocco, oct 2007. :np

[27] Mohammed Al'Achhab, Ahmed Hammad and Hassan Mountassir. Verifying LTL Properties on Hierarchical Systems : Application to Aircraft Autopilot. In *ISoLA 2006, 2nd Int. Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 19–26, Paphos, Cyprus, nov 2006. :ip

[28] Mohammed Al'Achhab, Ahmed Hammad, and Hassan Mountassir. Spécifications hiérarchiques et vérification des propriétés sous hypothèses d'équité. In M. Gourgaud and F. Riane, editors, *6ème Conf. Francophone de Modélisation et Simulation, MOSIM'06*, Rabat, Morocco, apr 2006. :np

[29] Mohammed Al'Achhab, Ahmed Hammad, and Hassan Mountassir. Vérification de systèmes hiérarchiques par raffinement. *Journal européen des systèmes automatisés (JESA)*, 39(1-3) :239–254, oct 2005. :nj

[30] Mohammed Al'Achhab, Ahmed Hammad, and Hassan Mountassir. Exploitation du raffinement pour vérifier les modèles hiérarchiques. In *Actes du RJCITR'05, 1ère Rencontres des Jeunes Chercheurs en Informatique Temps Réel*, Nancy, France, sep 2005. :onp

[31] Mohammed Al'Achhab, K. Cristiano, Ahmed Hammad, and Hassan Mountassir. Implementing Hierarchical automata in Promela/Spin. Technical Report RT2004-03, LIFC - Laboratoire d'Informatique de l'Université de Franche-Comté, apr 2004. :ir

[32] Mohammed Al'Achhab, Ahmed Hammad, and Hassan Mountassir. Refinement of Hierarchical Systems. Technical Report RR2004-03, LIFC - Laboratoire d'Informatique de l'Université de Franche-Comté, apr 2004. :ir

[33] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theor. Comput. Sci.*, 126(2) :183–235, 1994. 7

[34] Ahmed Hammad and Hassan Mountassir. Verification of real time systems : application to the transportation domain. In *SCLP'09, 2nd Int. Workshop on Service computing, Context-aware, Location-aware and Positioning techniques, joint to NTMS'09*, pages 719–727, Cairo, Egypt, dec 2009. :ip

[35] Abdelkader Bouamari, Mohammed Mostefai, and Ahmed Hammad. Modélisation UML des Systèmes Temps Réel. In *CIP 2005 - Conférence Internationale sur la Productique*, Tlemcen, Algeria, dec 2005. :np

[36] Ahmed Hammad. Apport du raffinement pour la vérification des systèmes temps réel. Conférence invitée aux Journées d'Informatique pour l'Entreprise (JIE2'04), Université de Blida, Algérie, oct 2004. :it

[37] Ahmed Hammad, Jacques Julliand, Hassan Mountassir, and Mohammed Al'Achhab. Conception et vérification de systèmes temps réel par raffinement. In *12th Conf. on Real Time and Embedded Systems, RTS EMBEDDED SYSTEMS'04*, pages 85–99, Paris, France, mar 2004. :np

[38] Ahmed Hammad, Hassan Mountassir, and Emilie Oudot. Préservation de propriétés MITL par raffinement temporisé. In Jacques Julliand, editor, *Congrès Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL'04*, pages 207–221, Besançon, France, jun 2004. :np

[39] Ahmed Hammad, Jacques Julliand, Hassan Mountassir, and D. Okalas. Expression en B et raffinement des systèmes réactifs temps réel. In *Congrès Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL'03*, pages 211–225, Rennes, France, jan 2003. :np

[40] About the SysML-Modelica transformation specification version 1.0. `https://www.omg.org/spec/SyM/1.0/`. Accessed : 2018-02-20. 7

[41] Modelica. `http://www.modelica.org`. accessed 2018-02-20. 7

[42] Kim G. Larsen, Paul Pettersson, and Wang Yi. *Int. Journal on Software Tools for Technology Transfer*, (1–2), Oct. :sttt97

[43] SystemC 2.0.1 Language Reference Manual Revision 1.0 . 2003. 7, 67

[44] Uppaal. `http://www.uppaal.org`. Accessed : 2018-02-20. 7

[45] Gerard J. Holzmann. The Model Checker SPIN. *Software Engineering*, 23(5) :279–295, 1997. 7

[46] LAAS. The TINA Tool Box. In *http ://homepages.laas.fr/bernard/tina/*, 2013. 7

[47] Jean-Marie Gauthier, Fabrice Bouquet, Ahmed Hammad, and Fabien Peureux. Transformation of SysML structure diagrams to VHDL-AMS. In Julien Bourgeois and Michel de Labachelerie, editors, *dMEMS 2012, Workshop on design, control and software implementation for distributed MEMS*, pages 74–81, Besançon, France, apr 2012. :ip

[48] Alain Giorgetti, Ahmed Hammad, and Bruno Tatibouet. Using SysML for Smart Surface Modeling. In *dMEMS'10, 1st workshop on design, control and software implementation for distributed MEMS*, pages 100–107, Besançon, France, jun 2010. :ip

[49] Ahmed Hammad, Hassan Mountassir, and Samir Chouali. Combining SysML and Modelica to Verify the Wireless Sensor Networks Energy Consumption. In *MODELSWARD 2013, 1st Int. Conf. on Model-Driven Engineering and Software Development*, pages 198–201, Barcelona, Spain, feb 2013. :oip

[50] Ahmed Hammad, Mountassir Hassan, and Samir Chouali. An approach combining SysML and modelica for modelling and validate wireless sensor networks. In *SESoS'13, 1st Int. Workshop on Software Engineering for Systems-of-Systems*, pages 5–12, Montpellier, France, jul 2013. Association for Computing Machinery (ACM). :ip

[51] Jean-Marie Gauthier, Fabrice Bouquet, Ahmed Hammad, and Fabien Peureux. Verification and Validation of Meta-Model Based Transformation from SysML to VHDL-AMS. In *MODELSWARD 2013, 1st Int. Conf. on Model-Driven Engineering and Software Development*, pages 123–128, Barcelona, Spain, feb 2013. :oip

[52] S. Berrani, Ahmed Hammad, and Hassan Mountassir. Mapping SysML to modelica to validate wireless sensor networks non-functional requirements. In *ISPS'13, 11th Int. Symposium on Programming and Systems*, pages 177–186, Algiers, Algeria, apr 2013. :ip

[53] Jean-Marie Gauthier, Dominique Gendreau, Ahmed Hammad, and Fabrice Bouquet. Modeling and Simulation of Modular Complex System : Application to Air-jet Conveyor. In *IEEE/ASME International Conference on Advanced Intelligent Mechatronics, AIM'2014.*, pages 1194–1199, Besançon, jul 2014. 8, 87

[54] Jean-Marie Gauthier, Fabrice Bouquet, Ahmed Hammad, and Fabien Peureux. Tooled Process for Early Validation of SysML Models using Modelica Simulation. In *FSEN'15, 6th IPM Int. Conf. on Fundamentals of Software Engineering*, volume 9392 of *Lecture Notes in Computer Science (LNCS)*, pages 230–237, Tehran, Iran, apr 2015. Springer. :ip

[55] Jean-Marie Gauthier, Fabrice Bouquet, Ahmed Hammad, and Fabien Peureux. A SysML Formal Framework to Combine Discrete and Continuous Simulation for Testing. In *17th International Conference on Formal Engineering Methods (ICFEM 2015)*, volume 9407 of *Lecture Notes in Computer Science (LNCS)*, pages 134–152, Paris, France, nov 2015. Springer. :ip

[56] Abbas Abdulhameed. *Combining SysML and SystemC to Simulate and Verify Complex Systems*. PhD thesis, Mars 2016. 8, 66, 87

[57] Abdulhameed Abbas, Ahmed Hammad, Bruno Tatibouet and Hassan Mountassir. An Approach to Verify SysML Functional Requirements Using Promela Spin. In *2015 12th International Symposium onProgramming and Systems (ISPS)*, pages 323 – 331, Algiers, Algeria, apr 2015. IEEE. :ip

[58] Abdulhameed Abbas, Ahmed Hammad, Hassan Mountassir and Bruno Tatibouet. An Approach based on SysML and SystemC to Simulate Complex Systems. In *MODELSWARD 2014, 2nd Int. Conf. on Model-Driven Engineering and Software Development*, pages 555–560, Lisbon, Portugal, jan 2014. :oip

[59] Abdulhameed Abbas, Ahmed Hammad, Hassan Mountassir and Bruno Tatibouet. An Approach Combining Simulation and Verification for SysML using SystemC and Uppaal. In *CAL 2014, 8ème conférence francophone sur les architectures logicielles*, page 9 pages, Paris, France, jun 2014. :onp

[60] M. Rahim, M. Boukala-Ioualalen and A. Hammad. Slicing based verification approach for the validation of SysML activity diagrams. In *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 1–6, Sept 2017. :ip

[61] Rahim Messaoud, Hammad Ahmed and Ioualalen Malika. A Methodology for Verifying SysML Requirements using Activity Diagrams. *Innovations in Systems and Software Engineering*, 13(1) :19–33, jun 2017. :ij

[62] Rahim Messaoud, Ahmed Hammad, and Malika Boukala-Ioualalen. Towards the Formal Verification of SysML Specifications : Translation of Activity Diagrams into Modular Petri Nets. In *SERA 2015, 13th IEEE/ACIS Int. Conf. on Software Engineering Research, Management and Applications*, pages 509–516, Hammamet, Tunisia, may 2015. IEEE. :ip

[63] Messaoud Rahim, Ahmed Kheldoun, Malika Ioualalen-Boukala, and Ahmed Hammad. Recursive ECATNets-Based Approach for Formally Verifying SysML Activity Diagrams. *IET Software*, 9(5) :119–128, oct 2015. :ij

[64] Messaoud Rahim, Malika Boukala-Ioualalen, and Ahmed Hammad. Petri Nets Based Approach for Modular Verification of SysML Requirements on Activity Diagrams. In *PNSE'14, A satellite event of Petri Nets 2014, 35th Int. Conf. on Application and Theory of Petri Nets and Concurrency and ACSD 2014, 14th Int. Conf. on Application of Concurrency to System Design*, pages 233–248, Tunis, Tunisia, jun 2014. :ip

[65] Rahim Messaoud, Malika Ioualalen, Mohamed Cherif Boukala, and Ahmed Hammad. Parallel Object-Based Load Balancing Strategy for Distributed State Space Construction : Application on Petri nets. In *INDS 2014, Int. Conf. on Advanced Networking Distributed Systems and Applications*, pages 41–46, Bejaia, Algeria, jun 2014. :ip

[66] Messaoud Rahim, Ahmed Hammad, and Malika Ioualalen. Modular and Distributed Verification of SysML Activity Diagrams. In *MODELSWARD 2013, 1st Int. Conf. on Model-Driven Engineering and Software Development*, pages 202–205, Barcelona, Spain, feb 2013. :oip

[67] Samir Chouali and Ahmed Hammad. Formal verification of components assembly based on SysML and interface automata. *Innovations in Systems and Software Engineering*, 7(4) :265 – 274, dec 2011. :ij

[68] Samir Chouali, Ahmed Hammad, and Hassan Mountassir. Assembling Components using SysML with Non-Functional Requirements. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 295 :31–47, may 2013. :ip

[69] Hamida Bouaziz, Samir Chouali, Ahmed Hammad, and Hassan Mountassir. Exploitation de la Hiérarchie pour la Vérification de la Compatibilité entre les Blocs SysML. In *CAL 2015, 9ème conférence francophone sur les architectures logicielles*, Hammamet, Tunisia, may 2015. :np

[70] Hamida Bouaziz, Samir Chouali, Ahmed Hammad, and Hassan Mountassir. Compatibility Verification of SysML Blocks Using Hierarchical Interface Automata. In *2015 12th International Symposium on Programming and Systems (ISPS)*, pages 313–322, Algiers, Algeria, apr 2015. IEEE. :ip

[71] Hamida Bouaziz, Samir Chouali, Ahmed Hammad, and Hassan Mountassir. SysML Blocks Adaptation. In Michael Butler, Sylvain Conchon et Fatiha Zaidi, editor, *Formal Methods and Software Engineering*, volume 9407 of *Lecture Notes in Computer Science (LNCS)*, pages 417–433, Paris, France, nov 2015. Springer. :ip

[72] Hamida Bouaziz, Samir Chouali, Ahmed Hammad, and Hassan Mountassir. Exploitation de la Hiérarchie pour la Véification de la Compatibilité des Blocs SysML. *Revue des Nouvelles Technologies de l'Information*, RNTI-L(8) :99–118, 2016. :nj

[73] Hamida Bouaziz, Samir Chouali, Ahmed Hammad, and Hassan Mountassir. A Model-Driven Approach to Adapt SysML Blocks. In *Information and Software Technologies*, volume 639 of *Communications in Computer and Information Science*, pages 255–268, Druskininkai, Lithuania, oct 2016. Springer. 8, 101

[74] Hamida Bouaziz, Samir Chouali, Ahmed Hammad, and Hassan Mountassir. SysML Model-Driven Approach to Verify Blocks Compatibility. *International Journal of Computer Aided Engineering and Technology (IJCAET)*, A venir :A venir, 2017. 8

[75] Catherine Caille-Cattin, Ahmed Hammad, Jean-Louis Poirey, and Rémi Thomas. Integration of the E&T database in the E-pragma system. In *Besançon 2008, Int. Conf. of Territorial Intelligence*, Besançon, France, oct 2008. :oip

[76] Hammad, Ahmed. Modèles UML de E-PRAGMA et intégration des bases de données de l'Observatoire Education et Territoires. In *L'enseignement scolaire en milieu rural et montagnard - Tome 5, Après le Collège*, Groupe d'intérêt scientifique 36, pages 41–56. 2010. :bc

[77] Thierry Chanier and Ahmed Hammad. Un test adaptatif informatisé pour le français : le projet TAFIC. In *Actes du Colloque FRAnche-Comté Traitement Automatique des Langues, (FRACTAL)*, Besançon, France, 1997. :np

[78] Métro de Paris-Ligne 14 - Projet METEOR. `http://www.systra.com/fr-projet/metro-de-paris-ligne-14-projet-meteor/`. 2018 (Accessed 2018-02-20). 14

[79] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Météor : A Successful Application of B in a Large Project. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, 1999, Proceedings, Volume I*, volume 1708 of *Lecture Notes in Computer Science*, pages 369–387. Springer, 1999. :conf/fm/BehmBFM99

[80] Siemens Transport. `http://w5.siemens.com/`, 2017. Accessed : 2017-03-06. 14

[81] Clearsy. `http://www.clearsy.com/`, 2017. Accessed : 2017-03-06. 14

[82] Gemalto. `http://www.gemalto.com//`, 2017. Accessed : 2017-03-06. 14

[83] K. Lano. *The B Language and Method. A Guide to Pratical Formal Development*. Springer, 1996. ISBN 3-540-76033-4. 14

[84] Patin, Florent and Pouzancre, Guilherm and Servat, Thierry. Approche formelle pour la réalisation d un système sécuritaire de contrôle commande de façades de quais. In *4ème conférence d'ingénierie Système, AFIS 2006*, pages 1–5, Toulouse, France, Mai 2006. 14

[85] Jean-Michel Bruel. Integrating formal and informal specification techniques. why ? how ? In *2nd Workshop on Industrial-Strength Formal Specification Techniques (WIFT '98), October 20-23, 1998, Boca Raton, FL, USA*, page 50. IEEE Computer Society, 1998. :conf/wift/-Bruel98

[86] Colin F. Snook and Michael J. Butler. UML-B : Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1) :92–122, 2006. :journals/tosem/SnookB06

[87] Regine Laleau and Fiona Polack. Coming and Going from UML to B : A Proposal to Support Traceability in Rigorous IS Development. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB 2002 : Formal Specification and Development in Z and B, 2nd International Conference of B and Z Users, Grenoble, France, January 23-25, 2002, Proceedings*, volume 2272 of *Lecture Notes in Computer Science*, pages 517–534. Springer, 2002. :conf/zum/LaleauP02

[88] Hung Ledang. Automatic Translation from UML Specifications to B. In *16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA*, page 436. IEEE Computer Society, 2001. :conf/kbse/-LeDang01

[89] Akram Idani, Yves Ledru, and Didier Bert. Derivation of UML Class Diagrams as Static Views of Formal B Developments. In Kung-Kiu Lau and Richard Banach, editors, *Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods, ICFEM 2005, Manchester, UK, November 1-4, 2005, Proceedings*, volume 3785 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 2005. :conf/icfem/IdaniLB05

[90] Yves Ledru, Régine Laleau, and Sylvie Vignes. Une tentative d'utilisation conjointe d'UML et d'une méthode formelle pour la modélisation de la sécurité des aéroports. In *Actes du XXVème Congrès INFORSID, Perros-Guirec, France, 22 au 25 mai 2007*, pages 155–170, 2007. :conf/inforsid/LedruLV07

[91] J. Warmer and A. Kleppe. *The Object Constraint Language : Precise Modeling with UML*. Addison-Wesley, 1999. ISBN 0-201-37940-6. 16

[92] About the Action Language for Foundational UML specification version 1.1. `https://www.omg.org/spec/ALF/About-ALF/`. Accessed : 2018-02-20. 23

[93] Emil Sekerinski. Graphical design of reactive systems. In *International Conference of B Users*, pages 182–197. Springer Berlin Heidelberg, 1998. 23

[94] D. Harel. Statecharts. A visual formalism for complex systems. *Science of computer programming*, (8) :231–274, 1987. 23

[95] E. Meyer. *Développements formels par objets : utilisation conjointe de B et d'UML*. PhD thesis, LORIA Université Nancy2, mars, 2001. 23

[96] Akram Idani and Yves Ledru. Dynamic graphical {UML} views from formal B specifications. *Information and Software Technology*, 48(3) :154–169, 2006. 23

[97] E. Meyer. *Développements formels par objets : utilisation conjointe de B et d'UML*. PhD thesis, Thèse de doctorat LORIA - Université Nancy 2, Nancy(F), Mars 2001. 27

[98] P. Facon, R. Laleau, and H.P. Nguyen. From OMT diagrams to B specifications. In M. Frappier and H. Habrias, editors, *An Overview Using a Case Study*, Software Specification Methods, pages 55–77. Springer, 2000. 27

[99] K. Lano, H. Houghton, and P. Wheeler. Integrating Formal and Structured Methods in Object-Oriented System Development. In *Formal Methods and Object Technology*. Springer-Verlag, 1996. ISBN 3-540-19977-2. 27

[100] C. Snook and M. Butler. *U2B : a tool for combining UML and B*, 2000. http ://www.ecs.soton.ac.ak/ cfs98r/U2Bdownnloads.htm. 27

[101] R. Laleau and A. Mammar. A Generic Process to Refine a B Specification into a Relationnal Database Implementation. In *ZB 2000 : Formal Specification and Development in Z and B, LNCS 1878, York (UK)*, August/September 2000. Springer - Verlag. 27

[102] R. Laleau and A. Mammar. An Overview of a Method and its support Tool for Generating B Specifications from UML Notations. In *The 15st IEEE Int. Conf. on Automated Software Engineering, Grenoble (F)*, September 11-15 2000. 27

[103] R. Marciano and N. Levy. Transformation d'annotations OCL en expressions B. In *AFADL'2001*, pages 39–49, 2001. 27

[104] H. Ledang and J. Souqieres. Integration of UML and B Specification Techniques : Systematic Transformation from OCL Expressions into B. In *APSEC 2002 : The* 9^{th} *Asia Pacific Software Engineering Conference. Gold Coast. Queesland (AU)*, Decembre 4-6 2002. 27

[105] H. Fekih, L. Jemni, and S. Merz. Transformation des Specifications B en des Diagrammes UML. In *AFADL'2004*, pages 131–148, Besançon, France, 2004. 27

[106] Jéremy Milhau, Akram Idani, Régine Laleau, Mohamed-Amine Labiadh, Yves Ledru, and Marc Frappier. Combining UML, ASTD and B for the formal specification of an access control filter. *ISSE*, 7(4) :303–313, 2011. :journals/isse/MilhauILLLF11

[107] Akram Idani, Mohamed-Amine Labiadh, and Yves Ledru. Infrastructure dirigée par les modèles pour une intégration adaptable et évolutive de UML et B. *Ingénierie des Systèmes d'Information*, 15(3) :87–112, 2010. :journals/isi/IdaniLL10

[108] Akram Idani and Yves Ledru. Dynamic graphical UML views from formal B specifications. *Information & Software Technology*, 48(3) :154–169, 2006. :journals/infsof/IdaniL06

[109] Rajeev Alur and Mihalis Yannakakis. Model Checking of Hierarchical State Machines. In *Foundations of Software Engineering*, pages 175–188, 1998. 30

[110] Diego Latella, István Majzik, and Mieke Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. *Formal Asp. Comput.*, 11(6) :637–664, 1999. 30

[111] Erich Mikk, Yassine Lakhnech, Michael Siegel, and Gerard J. Holzmann. Implementing Statecharts in Promela/SPIN. In *Proceedings of the 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques*, pages 90–101. IEEE Computer Society, October 1998. 30, 33, 107

[112] Edmund M. Clarke and Bernd-Holger Schlingloff. Model checking. In *Handbook of automated reasoning*, pages 1635–1790. Elsevier Science Publishers B. V., 2001. 30, 107

[113] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society. 30

[114] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992. 30

[115] C. Darlot, J. Julliand, and O. Kouchnarenko. Refinement Preserves PLTL Properties. In *Third International Conference of B and Z Users ZB'03 - Formal Specification and Development in Z and B*, volume 2651 of *LNCS*, pages 408–420, Turku, Finland, jun 2003. 30

[116] Nacéra Benaouda, Hervé Guyennet, Ahmed Hammad, and Mohamed Lehsaini. Design and Verification of a Self-organisation Algorithm for Sensor Networks. In Azizah Abd Manaf, Shamsul Sahibuddin, Rabiah Ahmad, Salwani Mohd Daud, and Eyas El-Qawasmeh, editors, *ICIEIS 2011, Int. Conf. on Informatics Engineering and Information Science*, volume 253 of *Communications in Computer and Information Science*, pages 530–543, Kuala Lumpur, Malaysia, 2011. :ip

[117] Bouamari Abdelkader, Mohammed Mostefai, Hassan Mountassir, and Ahmed Hammad. Efficient model checking with UML and specification patterns. In *ICWIT'10, the 3rd Int. Conf. on Web and Information Technologies*, pages 55–67, Marrakech, Morocco, jun 2010. :ip

[118] Nacéra Benaouda, Hervé Guyennet, Ahmed Hammad, and Mohammed Mostefai. A New Two Level Hierarchy Structuring for node Partitionning in Ad Hoc Networks. In *SAC'10, 25th ACM Symposium on Applied Computing*, pages 719–726, Zurich, Switzerland, mar 2010. :ip

[119] Daniel Knorreck, Ludovic Apvrille, and Pierre de Saqui-Sannes. TEPE : a SysML language for time-constrained property modeling and formal verification. *SIGSOFT Softw. Eng. Notes*, 36 :1–8, January 2011. 46, 63, 100

[120] Petri Nets Tool Database. `https://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html`. Accessed : 2018-02-20. 46

[121] Ahmad, Manzoor and Dragomir, Iulia and Bruel, Jean-Michel and Ober, Iulian and Belloir, Nicolas. Early Analysis of Ambient Systems SysML Properties using OMEGA2-IFx. In *SIMULTECH 2013*, Reykjavik, Iceland, Jul 2013. :hal-01085410

[122] Iulian Ober and Iulia Dragomir. OMEGA2 : A New Version of the Profile and the Tools. In *15th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2010, Oxford, United Kingdom, 22-26 March 2010*, pages 373–378, 2010. :conf/iceccs/OberD10

[123] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. *The IF Toolset*, pages 237–267. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. 46

[124] Iulian Ober, Susanne Graf, and Yuri Yushtein. Using an UML profile for timing analysis with the IF validation tool-set. In *Dagstuhl-Workshop MBEES : Modellbasierte Entwicklung eingebetteter Systeme II, Schloss Dagstuhl, Germany, 9.-13. Januar 2006, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*, pages 75–84, 2006. :conf/mbees/OberGY06

[125] A. Estefan, Jeff. Survey of Model-Based Systems Engineering (MBSE) Methodologies. 25 :, 01 2008. 46

[126] Peter H Feiler, David P Gluch, and John J Hudak. The architecture analysis & design language (AADL) : An introduction. Technical report, DTIC Document, 2006. 47

[127] OMG. OMG Object Constraint Language (OCL), Version 2.3.1. 2010. downloadable from http ://www.omg.org/spec/OCL/2.3.1/PDF. 51

[128] Matthew B Dwyer, George S Avrunin, and James C Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the International Conference on Software Engineering*, pages 411–420. IEEE, 1999. 51

[129] Peter Huber, Kurt Jensen, and Robert M Shapiro. Hierarchies in coloured Petri nets. In *Advances in Petri Nets 1990*, pages 313–341. Springer, 1989. 52

[130] Reference site for the implementation of Petri Net Markup Language (PNML). `http://www.pnml.org`. Accessed : 2018-02-20. 53

[131] Lom Messan Hillah, Fabrice Kordon, Laure Petrucci, and Nicolas Trèves. PNML Framework : an extendable reference implementation of the Petri Net Markup Language. In *31st International Conference on Petri Nets and Other Models of Concurrency (ICATPN 2010)*, volume 6128 of *Lecture Notes in Computer Science*, pages 318–327. Springer, June 2010. MoVe INT LIP6. 53

[132] Anne Vinter Ratzer, Lisa Wells, Henry Michael Lassen, Mads Laursen, Jacob Frank Qvortrup, Martin Stig Stissing, Michael Westergaard, Søren Christensen, and Kurt Jensen. CPN tools for editing, simulating, and analysing coloured Petri nets. In *Applications and Theory of Petri Nets 2003*, pages 450–462. Springer, 2003. 53

[133] Allan Cheng, Søren Christensen, and Kjeld Høyer Mortensen. Model checking Coloured Petri Nets-exploiting strongly connected components. *DAIMI Report Series*, 26(519), 1997. 53

[134] Albert Vincent J.C. Pascal Damien Foures. ACTIVITYDIAGRAM2PETRINET : Transformation-Based Model In Accordance With The OMG SySML Specifications. In *Proceedings of the Eurosis, The 2011 European Simulation and Modelling Conference*, pages 429–434, 2011. 53

[135] H. Storrle. Semantics of Control-Flow in UML 2.0 Activities. In *2004 IEEE Symposium on Visual Languages and Human Centric Computing, Rome, Italy*, pages 235–242, Sept 2004. 53, 62

[136] Damien Foures, Vincent Albert, Jean-Claude Pascal, and Alexandre Nketsa. Automation of SysML activity diagram simulation with model-driven engineering approach. In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium*, pages 11 :1–11 :6, San Diego, CA, USA, 2012. Society for Computer Simulation International. 53

[137] H. Yu H. Sun N. Yang and Z. Qian. Mapping UML Activity Diagrams to Analyzable Petri Net Models. In *Proc. of the 2010 10th Int. Conf. on Quality Software, IEEE, Zhangjiajie*, pages 369–372, 2010. 53

[138] E. Andrade, P. Macie, G. Callou, and B. Nogueira. A Methodology for Mapping SysML Activity Diagram to Time Petri Net for Requirement Validation of Embedded Real-Time Systems with Energy Constraints. In *Third International Conference on Digital Society, ICDS'09, Cancun, Mexico*, pages 266–271, 2009. 53

[139] Tony Spiteri Staines. Intuitive mapping of UML 2 activity diagrams into fundamental modeling concept Petri net diagrams and colored Petri nets. In *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*, pages 191–200. IEEE, 2008. 53, 62

[140] Freddy Allilaire, Jean Bézivin, Frederic Jouault, and Ivan Kurtev. ATL-eclipse support for model transformation. In *Proceedings of the Eclipse Technology eXchange workshop (eTX) at the ECOOP 2006 Conference, Nantes, France*, volume 66. Citeseer, 2006. 54

[141] Hamid Alavi, George Avrunin, James Corbett, Laura Dillon, Matt Dwyer, and Corina Pasareanu. Specification Patterns. url : http ://patterns.projects.cis.ksu.edu. 55

[142] Mitrabinda Ray, Soubhagya Sankar Barpanda, and Durga Prasad Mohapatra. Test case design using conditioned slicing of activity diagram. *International Journal of Recent Trends in Engineering (IJRTE)*, 1 :117–120, 2009. 59

[143] Rupinder Singh and Vinay Arora. Literature Analysis on Model based Slicing. *International Journal of Computer Applications*, 70(16) :45–51, 2013. 59

[144] Wang Ji, Dong Wei, and Qi Zhi-Chang. Slicing hierarchical automata for model checking UML statecharts. In *Formal Methods and Software Engineering*, pages 435–446. Springer, 2002. 59

[145] Philip Samuel and Rajib Mall. Slicing-based test case generation from UML activity diagrams. *ACM Sigsoft Software Engineering Notes*, 34 :1–14, 2009. 59

[146] Johan Lilius and Ivan Porres Paltor. vUML : A tool for verifying UML models. In *Automated Software Engineering, 1999. 14th IEEE International Conference on*, pages 255–258. IEEE, 1999. 62

[147] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal aspects of computing*, 11(6) :637–664, 1999. 62

[148] Timm Schafer, Alexander Knapp, and Stephan Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3) :357–369, 2001. 62

[149] Li Jing, Li Jinhua, and Zhang Fangning. Model Checking UML Activity Diagrams with SPIN. In *International Conference on Computational Intelligence and Software Engineering, 2009. CiSE 2009.*, pages 1–4, 2009. 62

[150] Kenneth L McMillan. The SMV system. In *Symbolic Model Checking*, pages 61–85. Springer, 1993. 62

[151] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, et al. Nusmv 2 : An opensource tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 359–364, London, UK, 2002. Springer-Verlag. 62

[152] Kevin Compton, Yuri Gurevich, James Huggins, and Wuwei Shen. An automatic verification tool for UML. *Univ. of Michigan, EECS Dept. Tech. Report CSE-TR-423*, 2000. 62

[153] M. Encarnacion "Beato, Manuel Barrio-Solorzano, Carlos E. Cuesta, and Pablo" de la Fuente. UML Automatic Verification Tool with Formal Methods. *Electronic Notes in Theoretical Computer Science*, 127(4) :3–16, 2005. 62

[154] Jérôme Delatour and Mario Paludetto. UML/PNO : A way to merge UML and Petri net objects for the analysis of real-time systems. In *Object-Oriented Technology : ECOOP'98 Workshop Reader*, pages 511–514. Springer, 1998. 62

[155] Jérôme Delatour. *Contribution à la spécification des systèmes temps réels : L'approche UML/PNO*. PhD thesis, Université Paul Sabatier-Toulouse III, 2003. 62

[156] Thouraya Bouabana-Tebibel and Mounira Belmesk. An object-oriented approach to formally analyze the UML 2.0 activity partitions. *Information and Software technology*, 49(9) :999–1016, 2007. 62

[157] Thouraya Bouabana-Tebibel and Mounira Belmesk. Formalization of UML object dynamics and behavior. In *Systems, Man and Cybernetics, 2004 IEEE International Conference on*, volume 5, pages 4971–4976. IEEE, 2004. 62

[158] M. von der Beeck. A structured operational semantics for UML-statecharts. *Software and Systems Modeling*, pages 130–141, 2002. 62

[159] A. Cavarra E. Borger and E. Riccobene. On formalizing UML state machines using ASMs. *Information and Software Technolog*, pages 287–292, 2004. 62

[160] H. Gronninger M. V. Cengarle and B. Rumpe. System model semantics of statecharts. *Informatik-Bericht*, 2008. 62

[161] Anthony Spiteri Staines. Simplified Bi-directional Transformation of UML Activities into Petri Nets. In *Proceedings of the 10th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems, Cambridge, UK*, SEPADS'11, pages 24–29. World Scientific and Engineering Academy and Society (WSEAS), 2011. 62

[162] Juan Pablo López-Grao, José Merseguer, and Javier Campos. From UML activity diagrams to Stochastic Petri nets : application to software performance engineering. *ACM SIGSOFT Software Engineering Notes*, 29(1) :25–36, 2004. 62

[163] Harald Storrle. Semantics and verification of data flow in UML 2.0 activities. *Electronic Notes in Theoretical Computer Science*, 127(4) :35–52, 2005. 62

[164] Z. Liu X. Li and J. He. A formal semantics of UML sequence diagram. In *Australian Software Engineering Conference*, pages 168–177. IEEE, 2004. 62

[165] H. Storrle. Semantics of interactions in UML 2.0. In *Symposium on Human Centric Computing Languages and Environments*, pages 129–136. IEEE, 2003. 62

[166] M. V. Cengarle and A. Knapp. UML 2.0 interactions : Semantics and reffinement. In *Critical Systems Development with UML*, pages 85–99. IEEE, 2004. 62

[167] Alexander Knapp and Jochen Wuttke. *Models in Software Engineering : Workshops and Symposia at MoDELS 2006, Genoa, Italy, October 1-6, 2006, Reports and Revised Selected Papers*, chapter Model Checking of UML 2.0 Interactions, pages 42–51. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. 62

[168] M. Debbabi, F. Hassaine, Y. Jarraya, A. Soeanu, and L. Alawneh. *Verification and Validation in Systems Engineering : Assessing UML/SysML Design Models*. Springer, 2010. Accessed January 2015. 62

[169] M.-V. Linhares, R.-S. de Oliveira, J.-M. Farines, and F. Vernadat. Introducing the Modeling and Verification process in SysML. In *12th IEEE Int. Conf. on Emerging Technologies and Factory Automation, ETFA'2007, Patras, Greece*, pages 344–351, 2007. 63

[170] Juan Carlos Hamon. *Méthodes et outils de la conception amont pour les systèmes et les microsystèmes*. Theses, Institut National Polytechnique de Toulouse - INPT, February 2005. :tel-00009523

[171] Pierre de SAQUI-SANNES. *Méthodologie de conception de systèmes temps réels et distribués en contexte UML/Sys,ML*. PhD thesis, université Toulouse III, 2008. 63

[172] Robert Cloutier. Introduction to this Special Edition on Model-based Systems Engineering. *INSIGHT*, 12(4) :7–8, 2009. :INST20091247

[173] Iulian Ober, Susanne Graf, and Ileana Ober. Validation of UML Models via a Mapping to Communicating Extended Timed Automata. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software*, pages 127–145, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. 63

[174] Luay Alawneh, Mourad Debbabi, Fawzi Hassaïne, and Andrei Soeanu. On the verification and validation of uml structural and behavioral diagrams. In *ACST*, 2006. 63

[175] Fontan, Benjamin. *Méthodologie de conception de systèmes temps réel et distribués en contexte UML/SysML*. Theses, Université Paul Sabatier - Toulouse III, Jan 2008. :tel-00258430

[176] Rahim Messaoud. *Proposition d'une démarche de spécification et de vérification des systèmes complexes*. PhD thesis, USTHB-Alger-Algérie, 2017. 64

[177] Jean-Marie Gauthier. *Combining Discrete and Continuous Domains for SysML-Based Simulation and Test Generation*. Theses, Université de Franche-Comté, November 2015. :tel-01248018

[178] System On Chip. `http://fr.wikipedia.org/wiki/SystemonChip`. Accessed : 2018-02-20. 66

[179] Systems engineering. `http://en.wikipedia.org/wiki/Systemsengineering`. accessed 2018-02-20. 67

[180] UML Profile category-specifications associated. `https://www.omg.org/spec/category/uml-profile`. Accessed : 2018-02-20. 67

[181] UML profile for SOC. `http://www.uml-sysml.org/documentation/uml-profile-for-soc-319-ko/view`. Accessed : 2018-02-20. 67

[182] *IEEE Standard VHDL Analog and Mixed-Signal Extensions Language Reference Manual V.1076.1-1999*, 1999. 67

[183] Verilog. `http://vol.verilog.com/`. Accessed : 2018-02-20. 67

[184] E. Christen and K. Bakalar. *VHDL-AMS* : a hardware description language for analog processing. In *IEEE Transactions on Circuits and Systems II : Analog and Digital Signal Processing (Volume : 46, Issue : 10)*, 1999. 67

[185] Claudionor Nunes Coelho and Harry D. Foster. *Assertion-Based Verification*, pages 167–204. Springer US, Boston, MA, 2004. 68

[186] The Accellera Systems Initiative. *Accellera Property Specification Language Reference Manual*. http ://www.accellera.org, 2004. dernière visite janvier 2018. 68

[187] Daniel Geist. The PSL/Sugar Specification Language A Language for all Seasons. In Daniel Geist and Enrico Tronci, editors, *Correct Hardware Design and Verification Methods*, page 3â3, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. 68

[188] C. T. Carr, T. M. McGinnity, and L. J. McDaid. Integration of UML and VHDL-AMS for analogue system modelling. *Formal Aspects of Computing*, 16 :80–94, 2004. 69, 71

[189] Medard Rieder, Rico Steiner, Cathy Berthouzoz, Francois Corthay, and Thomas Sterren. Synthesized UML, a Practical Approach to Map UML to VHDL. In springer verlag, editor, *RISE 2005 - Rapid Integration of Software Engineering techniques, LNCS*, volume LNCS 3943, pages 203–217, 2005. 69, 71

[190] Dag Bjorklund and Johan Lilius. From UML Behavioral Descriptions to Efficient Synthesizable VHDL. 2002. 69, 71

[191] D. H. Akehurst, O. Uzenkov, W. G. Howells, K. D. Mcdonald-Maier, and B. Bordbar. Compiling UML State Diagrams into VHDL : An Experiment in Using Model Driven Development. In *Using Model Driven Development. Forum on Specification and Design Languages (FDL'07)*, 2007. 69, 71

[192] David Guihal. *Modélisation en langage VHDL-AMS des systèmes pluridisciplinaires*. PhD thesis, Université Toulouse III, 2007. 69, 71

[193] Jean Verries. *Approche pour la conception de systèmes aéronautiques innovants en vue d'optimiser l'architecture : Application au système portes passagers*. PhD thesis, Université Toulouse III, 2010. 69, 71

[194] Y.A. Chapuis, L. Zhou, H. Fujita, and Y. HervÉ. Multi-domain simulation using VHDL-AMS for distributed MEMS in functional environment : Case of a 2-D air-jet micromanipulator. *Sensors and Actuators A : Physical* , 148 :pp.224–238, 2008. :hal-00350356

[195] D Muller and W Dotzel. Analyzing and simulation of MEMS in VHDL-AMS based on reduced order FE-models. 5 :866–871 Vol.2, 11 2003. 69

[196] Joachim Haase, Jens Bastian, and Sven Reitz. *VHDL-AMS in MEMS Design Flow*, pages 51–60. Springer US, Boston, MA, 2003. 69

[197] P. Voigt, G. Schrag, and G. Wachutka. Microfluidic system modeling using VHDL-AMS and circuit simulation. *Microelectronics Journal*, 29(11) :791–797, 1998. 69

[198] P. V. Nikitin and C.-J. R. Shi. VHDL-AMS based modeling and simulation of mixedtechnology microsystems : a tutorial, Integration. *the VLSI journal*, 40 :261–273, 2006. 69

[199] Bézivin, Jean and Breton, Erwan and Dupé, Grégoire and Valduriez, Patrick. The ATL transformation-based model management framework. Technical report, University of Nantes, Insititut de Recherche en Informatique de Nantes (IRIN), 2003. 70

[200] ATL. `http://www.eclipse.org/atl/`. Accessed : 2018-02-20. 70

[201] OMG Document Number : formal/2016 06-03. Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification. Technical report, DTIC Document, 2016. 70

[202] Kermeta3-Executable Meta-Modelling. `http://diverse-project.github.io/k3/index.html`. Accessed : 2018-02-20. 70

[203] Acceleo. `https://www.eclipse.org/acceleo/`. Accessed : 2018-02-20. 70

[204] Mahmoud Addouche, Fabrice Bouquet, Samir Chouali, Jean-Marie Gauthier, Ahmed Hammad, Isabelle Jacques, Jean-François Manceau, Hassan Mountassir, Fabien Peureux, Bruno Tatibouet, and Reda Yahiaoui. Transformation de modèles SysML vers VHDL-AMS. Technical report, December 2012. 70, 73, 87

[205] Smash. `https://www.dolphin-integration.com/`. Accessed : 2018-02-20. 70

[206] H. Elmqvist S. E. Mattsson M. Otter P. Schwarz C. Clauss. Mixed Domain Modeling in Modelica. In *Forum on Specification and Design Languages*, FDL'02, pages 23–27, Sep. 2002. 74

[207] P. Fritzson and P. Bunus. Modelica-A General Object-Oriented Language for Continuous and Discrete-Event System Modeling and Simulation. *Simulation Symposium, Annual*, 0 :0365, 2002. 74

[208] Modelica. `http://www.Modelica.org`. Accessed : 2018-02-20. 74

[209] G. Marguerie. De Modelica à XMLlab. *Ingénierie de Systèmes Informatiques Complexes ISICO, EISTI*, page 41, 2006. 74

[210] The Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling, Language Specification Version 3.0*, Sep. 2007. 74

[211] W. Schamai, P. Helle, P. Fritzson, and C. J. J. Paredis. Virtual verification of system designs against system requirements. In *Proceedings of the 2010 international conference on Models in software engineering*, MODELS'10, pages 75–89, Berlin, Heidelberg, 2011. Springer-Verlag. 75, 79

[212] M. Elizabeth C. Hull, Ken Jackson, and Jeremy Dick. *Requirements Engineering, Second Edition*. Springer, 2005. 75

[213] M. Luisa, F. Mariangela, and I. Pierluigi. Market research for requirements analysis using linguistic tools. *Requir. Eng.*, 9(1) :40–56, feb 2004. 75

[214] W. Schamai, P. Fritzson, Ch. J.J. Paredis, and A. Pop. Towards Unified System Modeling and Simulation with ModelicaML : Modeling of Executable Behavior Using Graphical Notations. In *Proc. 7th Modelica Conf.*, Sep. 2009. Como, Italy. 75, 77

[215] G. Iris, R. Alexander, and E. Alexander. Incremental Consistency Checking of Dynamic Constraints. In David S. Rosenblum and Gabriele Taentzer, editors, *Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6013 of *Lecture Notes in Computer Science*, pages 203–217. Springer, 2010. 75

[216] D. Bouskela T. Nguyen N. Ruel E. Thomas L. Chastanet R. Schoenig A. Jardin and S. Loembé. Modelling of System Properties in a Modelica Framework. In *Proceedings of the 8th International Modelica Conference, March 20th-22nd, Technical Univeristy, Dresden, Germany*, 2011. 75

[217] J. Haufe T. Blochwitz U. Donath and T. Neidhold. A new Approach for Modeling and Verification of Discrete Control Components within a Modelica Environment. In *6th International Modelica Conference*, 2008. 75

[218] Simulation. `http://www.simulationx.com`. Accessed : 2018-02-20. 75

[219] OMG SysML-Modelica Transformation specification V1.0. `http://www.omg.org/spec/SyM/1.0/`. Accessed : 2018-02-20. 76

[220] C. J. J. Paredis and T. Johnson. Using OMG's SysML to support simulation. In *Proceedings of the 40th Conference on Winter Simulation*, WSC '08, pages 2350–2352. Winter Simulation Conference, 2008. 76, 77

[221] T. A. Johnson. *Integrating models and simulations of continuous dynamic system behavior into SysML*. PhD thesis, Woodruff School of Mechanical Engineering, Georgia Institute of Technology. Atlanta, GA., 2008. 76

[222] R. Chenouard R. Renier. De SysML à Modelica : aide a la formalisation de modèles de simulation en conception préliminaire. In *12ème Colloque National AIP PRIMECA*, 2011. 77

[223] P. Vasaiely. Interactive Simulation of SysML Models using Modelica. Master's thesis, Hamburg University of Applied Sciences - Faculty of Engineering and Computer Science - Department Computer Science, August 2009. 77

[224] M. Zukerman M. N. Halgamuge and K. Ramamohanarao. An Estimation of Sensor Energy Consumption. *Progress In Electromagnetics Research B (PIERB)*, 12 :259–295, 2009. 81

[225] Laetitia Matignon, Guillaume J Laurent, Nadine Le Fort-Piat, and Yves-André Chapuis. Designing decentralized controllers for distributed-air-jet mems-based micromanipulators by reinforcement learning. *Journal of intelligent & robotic systems*, 59(2) :145–166, 2010. 83

[226] Régine Laleau, Farida Semmak, Abderrahman Matoussi, Dorian Petit, Ahmed Hammad, and Bruno Tatibouet. A first attempt to combine SysML requirements diagrams and B. *Innovations in Systems and Software Engineering*, 6(1) :47–54, mar 2010. :ij

[227] Ahmed Hammad, Hassan Mountassir, and Bruno Tatibouet. Using the profile UML4SoC for modeling a smart surface. In *ICEEDT'08, 2nd int. conf. on Electrical Engineering Design and Technology*, Hammamet, Tunisia, nov 2008. :ip

[228] J-M. Gauthier, F. Bouquet, A. Hammad, and F. Peureux. Transformation of SysML structure diagrams to VHDL-AMS. In *IEEE Workshop on design, control and software implementation for distributed MEMS (dMEMS'12)*, Besançon, France, 2012. IEEE CPS. 87

[229] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, pages 219–246, 1989. 90

[230] Szyperski, C. *Component Software*. ACM Press, Addison-Wesley, 1999. 90

[231] Heineman, G.T and Councill, W.T. *Component Based Software Engineering*. Addison Wesley, 2001. 90

[232] Konstantas, D. Interoperation of object oriented application. In Nierstrasz, Oscar and Tsichritzis, Dennis, editor, *Object-Oriented Software Composition*, pages 69–95. Prentice Hall, 1995. 90

[233] Wegner, P. Interoperability. *ACM Computing Survey*, 28(1) :285–287, 1996. 90

[234] Alfaro, L. and Henzinger, T. –A. Interface Automata. In *9th Annual Symposium on Foundations of Software Engineering, FSE*, pages 109–120. ACM Press, 2001. 90, 91, 92

[235] L. Alfaro and T. A. Henzinger. Interface-based Design. In *Engineering Theories of Software-intensive Systems*, volume 195 of *NATO Science Series : Mathematics, Physics, and Chemistry*, pages 83–104. Springer, M. Broy, J. Gruenbauer, D. Harel, and C.A.R. Hoare, 2005. 90

[236] L. Alfaro, T. A. Henzinger, and M. Stoelinga. Timed Interfaces. In *EMSOFT '02 : Proceedings of the Second International Conference on Embedded Software*, pages 108–122, London, UK, 2002. Springer-Verlag. 90

[237] B. Thomas Adler, Luca De Alfaro, Ro Dias Da Silva, Marco Faella, Axel Legay, Vishwanath Raman, and Pritam Roy. Ticc, a tool for interface compatibility and composition. In *In Proceedings 18th International Conference on Computer Aided Verification (CAV), volume 4144 of Lecture Notes in Computer Science*, pages 59–62. Springer, 2006. 92

[238] Baille, Gérard and Garnier, Philippe and Mathieu, Hervé and Pissard-Gibollet, Roger. *The INRIA Rhône-Alpes CyCab*. Technical report, INRIA , 1999. Describes the package natbib. 93

[239] Yosr Jarraya, Mourad Debbabi, and Jamal Bentahar. On the Meaning of SysML Activity Diagrams. In *Proceedings of the 2009 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, ECBS '09, pages 95–105, Washington, DC, USA, 2009. IEEE Computer Society. 100

[240] Jean-François Pétin, Dominique Evrot, Gérard Morel, and Pascal Lamy. Combining SysML and formal methods for safety requirements verification. In *22nd International Conference on Software & Systems Engineering and their Applications 22nd International Conference on Software & Systems Engineering and their Applications*, page CDROM, Paris France, 12 2010. 100

[241] Wang, Renzhong and Dagli, C. H. An Executable System Architecture Approach to Discrete Events System Modeling Using SysML in Conjunction with Colored Petri Net. In *Systems Conference, 2008, 2nd Annual IEEE*, 2008. 100

[242] Ermeson Carneiro, Paulo Maciel, Gustavo Callou, Eduardo Tavares, and Bruno Nogueira. Mapping SysML State Machine Diagram to Time Petri Net for Analysis and Verification of Embedded Real-Time Systems with Energy Constraints. *Electronics and Micro-electronics, International Conference on Advances in*, 0 :1–6, 2008. 100

[243] Bouaziz, Hamida. *Adaptation of SysML Blocks and Verification of Temporal Properties*. PhD thesis, 2016. Thèse de doctorat dirigée par Mountassir Hassan, Hammad Ahmed et Chouali Samir Informatique Besançon 2016. 101, 110

[244] Samir Chouali and Ahmed Hammad. Formal verification of components assembly based on SysML and interface automata. jun 2011. :ip

[245] Samir Chouali, Julien Dormoy, Ahmed Hammad, Jean-Michel Hufflen, Sebti Mouelhi, Olga Kouchnarenko, Hassan Mountassir, and Bruno Tatibouet. Assemblage des composants digne de confiance : de l'ingénierie des besoins aux spécifications formelles. *Génie Logiciel*, 95 :13–18, dec 2010. :nj

[246] F. Cave D. Bert. Construction of finite labelled transition systems from B abstract systems. In *IFM'2000*, number LNCS 1945, pages 235–254, November 2000. 106

[247] About the semantic of a Foundational subset for executable UML models specification version 1.3. `https://www.omg.org/spec/FUML/1.3/`. Accessed : 2018-02-20. 107