Liste des acronymes

UML Unified Modeling Language

SysML System Modeling Language

IE Ingénierie des Exigences

SE System Engineering

INCOSE International Concil on Systems Engineering

IDM Inginierie Dirigée par les Modèles

MDE Model Driven Engineering

DSML Domain Specific Modeling Languages

M2T Model to Text

M2M Model to Model

RHS Right Hand Side

LHS Left Hand Side

JMI Java Metadata Interface

HDL Hardware Description Language

CTL Computation Tree Logic

TCTL Timed Computation Tree Logic

BDD Binary Decision Diagram

DBM Difference Bound Matrices

TAN Timed Automata Network

SD Sequence Diagram

ATM Automated Teller Machine

PIN Personal Identification Number

CVV Card Verification Value

TABLE DES FIGURES

2.1 Taxonomie des diagrammes UML 12
2.2 Taxonomie des diagrammes SysML 14
2.3 Les quatre piliers de SysML 15
2.4 La relation entre UML et SysML 16
2.5 Syntaxe du diagramme de séquence 17
2.6 Méta-modèle du TAN 18
2.7 Les couches de Métamodelisation 20
2.8 Concepts de base de transformation de modèles 20
2.9 Principe de vérification formelle avec Model Checker 24

3.1 Vue générale sur notre approche 28
3.2 Méta-modèle du diagramme de sequence SD_{2D} 29
3.3 Structure du TAN_E 32
3.4 Formules TCTL 34

4.1 Architecture de Formal-Sequence 37
4.2 Capture d'écran du plug-in Formal-Sequence 38

4.3 Diagramme de comportement User/ATM 42
4.4 Automate modélisant le comportement de l'utilisateur 43
4.5 Automate modélisant le comportement de l'ATM 43
4.6 Simulation automatique avec UPPAL 44
4.7 Différentes intéractions de l'utilisateur et de l'ATM dans un seul chemin . . 45
4.8 Vérification des propriétés avec UPPAAL 46

Liste des tables

2.1 Les approches de transformation de modèles 21

3.1 Description sémantique des fragments dans SD_{2D} 30

3.2 Règles de mappage de SD_{2D} à TAN 33

4.1 Relations de mappage de SD_{2D} à TAN_e 44

SOMMAIRE

1 Introduction 9
- 1.1 Contexte 9
- 1.2 Problématique 9
- 1.3 Objectifs 10
- 1.4 Organisation du mémoire 10

2 État de l'art 11
- 2.1 Langages de modélisation 11
  - 2.1.1 UML 12
    - 2.1.1.1 Architecture UML 12
    - 2.1.1.2 Profil UML 13
  - 2.1.2 SysML 13
    - 2.1.2.1 Architecture SysML 14
    - 2.1.2.2 Les quatre piliers de SysML 15
    - 2.1.2.3 La relation entre UML et SysML 16
    - 2.1.2.4 Diagramme de Séquence 16
  - 2.1.3 Automates temporisés 17
- 2.2 Ingénierie dirigée par les modèles - IDM 19
  - 2.2.1 Définitions 19
  - 2.2.2 La métamodélisation 19
  - 2.2.3 Transformation de modèles 20
    - 2.2.3.1 Pourquoi la transformation de modèles 21
    - 2.2.3.2 Approches de transformation de modèles 21
    - 2.2.3.3 Transformation Model to Text - M2T 21
    - 2.2.3.4 Transformation Model to Model - M2M 22
- 2.3 Vérification et Validation 23
  - 2.3.1 Test et Simulation (Validation) 23
  - 2.3.2 Méthodes Formelles (Vérification) 23
    - 2.3.2.1 Model-checking 24

Rapport-Gratuit.com

2.3.2.2 Theorem Proving 25

3 Vers une Vérification Formelle des Modèles SysML 27

3.1 Vue générale de notre approche 27
3.2 Des diagrammes de séquence aux automates temporisés 29
3.2.1 Diagramme de Séquence Simplifié SD_{2D} 29
3.2.1.1 Syntaxe du Diagramme de Séquence SD_{2D} 29
3.2.1.2 Sémantique de Diagramme de séquence 30
3.2.2 Automate Temporisé Étendu TAN_E 31
3.2.3 Syntaxe et Définition de TAN_E 31
3.2.4 Automate Temporisé et Model Checker 32
3.3 Règles de transformation de SD_{2D} 33
3.4 Simulation et Vérification par TAN_E 33
3.4.1 Simulation 34
3.4.2 Vérification formelle 34

4 Formal-Sequence : un Plug-in Génération d'automates 37

4.1 De SysML vers UPPAAL 37
4.2 Algorithmes développés 38
4.2.1 Extraction des données 39
4.2.2 Construction de TAN_E 40
4.3 Étude de cas 40
4.3.1 Description du système ATM 41
4.3.2 Modélisation du comportement de l'ATM 41
4.3.3 Modélisation formelle de l'ATM 41
4.3.4 Vérification de l'ATM 44
4.3.4.1 Simulation avec l'outil Uppaal 44
4.3.4.2 Vérification avec l'outil Uppaal 45

5 Conclusion Générale 47

5.1 Conclusion 47
5.2 Perspectives 48

1

INTRODUCTION

Les systèmes critiques temps réel deviennent de plus en plus complexes et exigent un niveau de sûreté et de fiabilité très élevé. Cela nécessite l'utilisation des méthodes de spécification et de vérification dans la phase de conception pendant le cycle de développement de ces systèmes afin de réduire la notion d'erreur avant même de commencer l'implémentation ce qui permet le développement d'un système fiable et de réduire les coûts et les délais.

1.1/ CONTEXTE

Ce sujet est situé dans un contexte de recherche et de développement pour la modélisation et la vérification de systèmes hétérogènes (par exemple des systèmes répartis sur des plateformes diverses ou des logiciels embarqués sur divers matériels).

La recomposition souple de divers composants d'un modèle semi-formel (SysML) est une voie pour modéliser et construire de tels systèmes. Cependant afin de les analyser et garantir leur correction vis à vis des exigences, il faut formaliser les modèles SysML et les vérifier au plus tôt.

1.2/ PROBLÉMATIQUE

Les approches descendantes *(top-down)* sont régulièrement utilisées pour permettre la construction des systèmes logiciels complexes, étape par étape, à partir de spécifications abstraites de haut niveau. Cependant, ces approches ne fournissent pas forcément de garantie sur la sûreté *(safety)* des systèmes ainsi conçus. S'assurer qu'un système est sûr est, avant tout, s'assurer qu'il répond bien aux exigences exprimées dans le cahier de charge, d'où l'importance de l'activité de l'ingénierie des exigences.

Or, la complexité croissante des systèmes rend cette activité à la fois plus importante et plus difficile à maîtriser. Quand cette activité n'est pas prise convenablement en charge, divers dysfonctionnements peuvent survenir et peuvent avoir des répercussions bien plus graves qu'un simple désagrément pour l'utilisateur, allant dans certains cas jusqu'à des accidents graves avec des dégâts catastrophiques en terme d'économie ou de vie humaine.

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

Par conséquent, la vérification formelle de tels systèmes (complexes et critiques) de-

vient extrêmement importante afin de garantir leur bon fonctionnement. Néanmoins, son intégration dans le processus de développement rend encore plus complexe la communication entre les différentes parties prenantes et nécessite ainsi des efforts de conception et de vérification conséquents et complexes, qui vont à l'encontre des objectifs industriels de réduction de coûts et de délais de production.

1.3/ Objectifs

Les objectifs de ce projet sont de :

- simplifier le diagramme de séquence afin de répondre à nos besoins ;
- définir les règles de transformation entre le diagramme de séquence simplifié et l'automate temporisé ;
- proposer des algorithmes de transformation basés sur les règles de transformation prédéfinies ;
- et développer un outil pour la vérification formelle d'un diagramme de séquence afin d'assurer la fiabilité du comportement des logiciels et l'intégrer à Papyrus.

1.4/ Organisation du mémoire

- Dans le chapitre 2, nous aborderons les différents langages de modélisation, UML et SysML, leur points communs et leur différences. Nous parlerons également de l'ingénierie dirigée par les modèles, puis conclurons ce chapitre sur la définition de la Vérification et la Validation et en utilisant Model Checking comme exemple.
- Le chapitre 3 sera consacré à notre approche, nous présenterons le modèle simplifié du diagramme de séquence que nous avons proposé ainsi que l'automate temporisé étendu. Les règles de transformation pour le passage du diagramme de séquence à un automate temporisé serons également expliquées dans ce chapitre.
- Enfin, nous présenterons dans la chapitre 4 notre plug-in Formal-Sequence ainsi que les algorithmes de transformations. Nous appliquerons notre plug-in sur une étude cas d'ATM et exposerons les résultats obtenus.

Une conclusion générale ainsi que quelques perspectives sont données à la fin de ce manuscrit.

2

ÉTAT DE L'ART

2.1/ LANGAGES DE MODÉLISATION

Les langages de modélisation sont couramment utilisés pour spécifier, visualiser, stocker, documenter, et échanger des modèles de conception. Ils sont spécifiques au domaine et contiennent toutes les informations syntaxiques, sémantiques et de présentation concernant une application donnée dans un domaine.

Différents langages de modélisation ont été définis par des organisations et des entreprises afin de cibler différents domaines tels que le développement web (WebML [20]), télécommunications (TeD [2]), matériel (HDL [1]), logiciels, et plus récemment, les systèmes (UML [14]). D'autres langages tels que IDEF [19] ont été conçus pour un large éventail d'utilisations, y compris la modélisation fonctionnelle, la modélisation des données et la modélisation de résaux.

Bien que l'ingénierie des systèmes existe depuis plus de cinq décennies, jusqu'à récemment, il n'y avait pas de langage de modélisation dédié à cette discipline. Traditionnellement, les ingénieurs systèmes ont beaucoup travaillé sur la documentation pour exprimer les exigences système et, en l'absence d'une langue standard spécifique, ont utilisé différents langages de modélisation pour exprimer une solution de conception complète.

Cette diversité de techniques et d'approches a limité le travail coopératif et l'échange d'information. Parmi les langages de modélisation existants qui ont été utilisés par les systèmes ingénieurs, on peut citer HDL, IDEF et EFFBD. Afin de fournir une solution à ce problème, OMG et INCOSE [10], avec un certain nombre d'experts dans le domaine de l'ingénierie système, ont collaboré pour la construction d'un langage de modélisation standard. UML, étant le langage de modélisation par excellence pour l'ingénierie logicielle, était le langage de choix destiné à la personnalisation à l'égard des besoins des ingénieurs systèmes.

Cependant, la version UML 1.x s'est révélée inadéquate pour une telle utilisation et donc la révision en évolution de UML (c'est-à-dire, UML 2.0) a été publiée, avec des fonctionnalités spéciales pour les ingénieurs systèmes. En avril 2006, une proposition pour un langage de modélisation standard pour la modélisation des systèmes, à savoir SysML [11], a été soumis à l'OMG, avec l'objectif d'aboutir à un processus de normalisation final [24].

2.1.1/ UML

Le langage de modélisation unifié (UML) est une modélisation visuelle à usage général dont la maintenance a été assumée par l'OMG depuis 1997. C'est le résultat de la fusion de trois notations majeures : la méthodologie de Grady Booch [3] pour décrire un ensemble d'objets et leurs relations, la technique de modélisation des objets de James Rumbaugh (OMT [16]), et la méthodologie de cas d'utilisation d'Ivar Jacobson.

Bien que UML ait été conçu à l'origine pour spécifier, visualiser et documenter systèmes logiciels, il peut également être appliqué à divers autres domaines tels que l'organisation des sociétés et processus métier. UML a de nombreux avantages, il est largement accepté par de nombreux leaders de l'industrie, est non exclusif et extensible et est également commercialement soutenu par de nombreux outils et manuels. La norme UML a été révisée plusieurs fois et de nombreuses versions ont été publiées jusqu'à la version 2.5.1 publiée en 2018 [24].

2.1.1.1/ Architecture UML

Les diagrammes UML sont classés en deux catégories principales : structurelle et comportementale. Cette dernière catégorie comprend une catégorie de sous-ensembles appelée diagrammes d'interaction. La taxonomie des diagrammes est illustrée à la Fig 2.1.

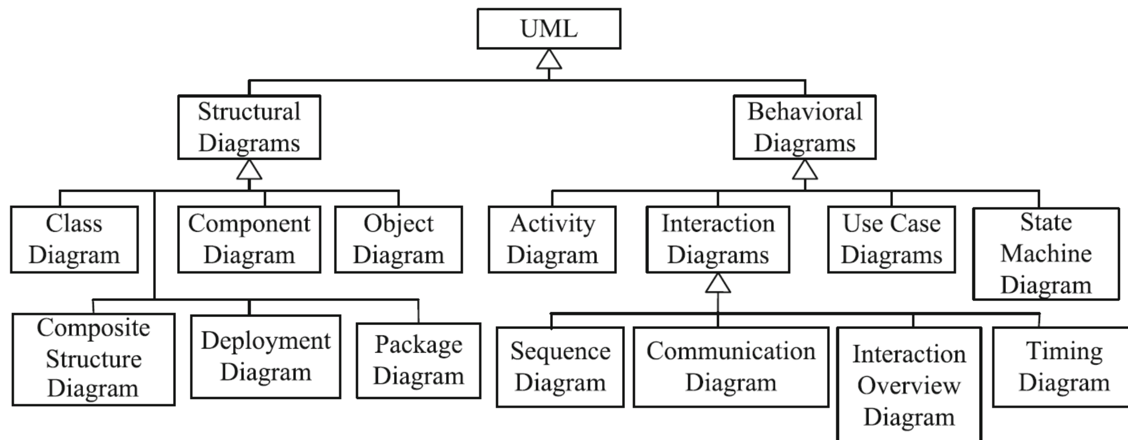


Figure 2.1 – Taxonomie des diagrammes UML

Les diagrammes structurels modélisent les aspects statiques et les caractéristiques structurelles du système, tandis que les diagrammes comportementaux décrivent le comportement dynamique du système. Les diagrammes structurels incluent le diagramme de classe, le diagramme de composants, le diagramme de structure composite, le diagramme de déploiement, le diagramme d'objet, et le diagramme de packages.

Les diagrammes comportementaux incluent le diagramme d'activité, le diagramme de cas d'utilisation et le diagramme d'état machines. Les diagrammes d'interaction font par-

tie de de la catégorie des diagrammes comportementaux car ils soulignent l'interaction entre les composants modélisés. Ils comprennent le diagramme de communication, le diagramme global d'interactions, le diagramme de séquence et le diagramme de temps.

Les nouveaux diagrammes proposés dans UML 2.x [15] sont le diagramme de structure composite, le diagramme global d'interactions et le diagramme de temps. En outre, d'autres diagrammes ont été étendus et modifiés depuis UML 1.x, à savoir, le diagramme d'activité, le diagramme d'état machine, le diagramme de séquence et le diagramme de communication. [15]

2.1.1.2/ Profil UML

UML définit un concept de profil spécifique qui fournit un mécanisme d'extension générique pour construire des modèles UML dans un domaine particulier. Le profilage UML permet un affinage de la sémantique des éléments UML standard d'une manière strictement additive et sans introduire de contradictions sémantique [24].

Les principaux mécanismes de profilage UML sont les stéréotypes et les valeurs marquées. Ces mécanismes sont appliqués à des éléments de modèle spécifiques tels que les classes, les attributs, les opérations et les activités afin de les adapter à différentes fins. Les mécanismes sont définis comme suit :

- Les stéréotypes UML sont utilisés pour étendre le vocabulaire UML en créant de nouveaux éléments de modèle dérivés des existants. Les stéréotypes ajoutent des propriétés spécifiques adapté à un domaine de problème spécifique et sont utilisés pour la classification ou le marquage des blocs de construction UML.
- Les annotations sont utilisées pour spécifier des valeurs de mot-clé. Elles permettent l'extension des propriétés d'un bloc de construction UML et la création de nouvelles informations dans la spécification d'un élément défini pour des éléments de modèle existants ou pour stéréotypes individuels. De plus, des valeurs balisées peuvent être utilisées pour spécifier des propriétés qui sont pertinentes pour la génération de code ou la gestion de la configuration. Divers profils UML ont été émis par l'OMG afin de personnaliser UML pour des domaines particuliers (par exemple, l'aérospatiale, la santé, la finance, les transports, l'ingénierie des système) ou des plates-formes (par exemple, J2EE, .NET) [24].

2.1.2/ SysML

Le langage de modélisation des systèmes (SysML [11]) est un langage dédié aux applications d'ingénierie des systèmes. C'est un profil UML qui non seulement réutilise un sous-ensemble d'UML 2.1.1 mais fournit également des extensions supplémentaires pour mieux s'adapter aux besoins spécifiques de l'ingénierie des systèmes.

Il est destiné à aider à spécifier et structurer des systèmes complexes et de leurs composants afin de permettre leur analyse, conception, vérification et validation. Ces systèmes peuvent être constitués de composants hétérogènes tels que du matériel, des logiciels, des informations, des processus, du personnel et des installations.

SysML englobe des capacités de modélisation qui permettent de représenter les systèmes et leurs composants en utilisant :

- La composition, classification et interconnexion des composants structuraux ;
- Les comportements, notamment les flux d'activité, les scénarios d'interaction et la transmission de messages ainsi que les comportements réactifs dépendants de l'état ;
- L'allocation d'un élément d'un modèle à un autre, comme les fonctions aux composants, composants logique aux composants physiques et logiciels au matériel ;
- Les contraintes sur les valeurs de propriété du système telles que la performance, la fiabilité et les propriétés physiques ;
- Les hiérarchies et les dérivations ainsi que leurs relations avec d'autres éléments du modèle [12].

2.1.2.1/ Architecture SysML

Les diagrammes SysML [11] définissent une syntaxe concrète qui décrit comment les concepts SysML sont visualisé graphiquement ou textuellement. Dans la spécification SysML, cette notation est décrite dans des tableaux qui montrent la cartographie des concepts de langage en symboles graphique sur les diagrammes [24]. La taxonomie des diagrammes est illustrée à la Fig 2.2.

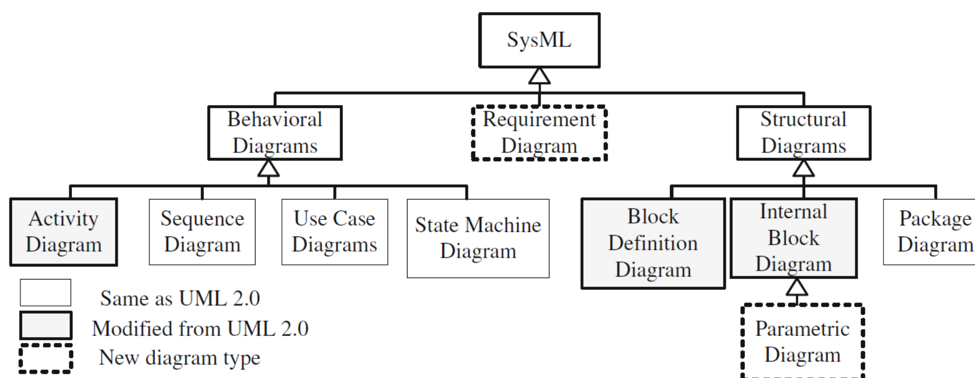


FIGURE 2.2 – Taxonomie des diagrammes SysML

SysML réutilise complètement certains diagrammes de UML 2.1 [15] :

- diagramme de cas d'utilisation ;
- diagramme de séquence ;
- diagramme d'état machine ;
- diagramme de package ;

En outre, deux nouveaux diagrammes sont ajoutés :

- diagramme d'exigences ;
- diagramme paramétrique ;

De plus, d'autres diagrammes UML sont réutilisés sous forme étendue :

- diagramme d'activité (étend le diagramme d'activité UML) ;
- diagramme de définition de bloc (étend le diagramme de classes UML) ;
- diagramme de bloc interne (extension du diagramme de structure composite UML) ;

2.1.2.2/ Les quatre piliers de SysML

Sur le plan des concepts, le langage SysML repose sur quatre ensembles fondamentaux, qui constituent les quatre piliers de SysML illustrés ci-dessous (Fig 2.3) :

- La notion de structure. Pour représenter la structure, on utilise la notion de bloc, qui permet de modéliser tout élément concret : système, sous systèmes, composants logiciels ou matériels, mais aussi acteurs, système externes, ou encore éléments de matière, d'énergie ou d'information échangés. Les diagrammes correspondants sont les diagrammes de définition de bloc et le diagramme de bloc interne ;
- La notion de comportement, pour la modélisation dynamique ou comportementale du système ou d'un constituant. Les diagrammes correspondants sont les diagrammes d'activités et le diagramme d'états (et au besoin de séquence) ;
- La notion d'exigence. Le diagramme correspondant est le diagramme d'exigences ;
- Les équations et contraintes régissant le système ou ses constituants. Le diagramme correspondant est le diagramme paramétrique [28].

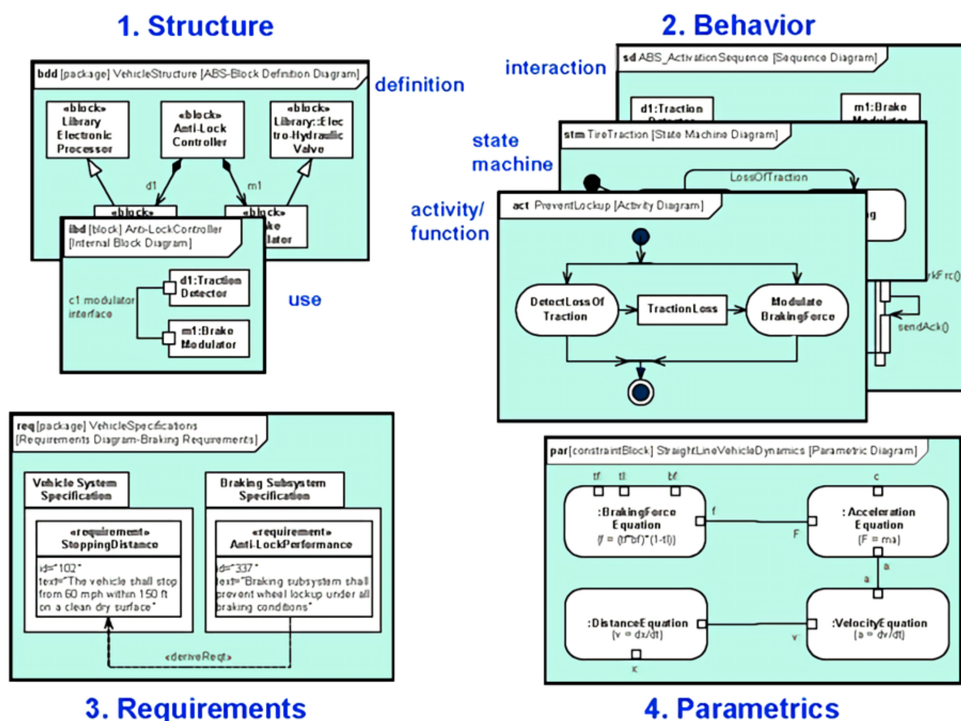


FIGURE 2.3 – Les quatre piliers de SysML

2.1.2.3/ LA RELATION ENTRE UML ET SYSML

SysML réutilise un sous-ensemble de UML 2.1 [15], appelé "UML4SysML", qui représente environ la moitié du langage UML. Une partie importante de l'UML des concepts ont été écartés car ils n'étaient pas considérés comme pertinents pour les besoins du modèle de SE. Certains des diagrammes réutilisés ont été inclus comme dans UML 2.1.1 [15].

Ceux-ci incluent la machine d'état, la séquence et les diagrammes de cas d'utilisation. D'autres diagrammes ont été étendus, tels que les diagrammes d'activité, afin de répondre aux besoins spécifiques des SE. De plus, SysML a omis certains diagrammes UML, à savoir d'objet, de composant, de déploiement,de communication, de temps et d'interactions.

Les diagrammes de structure dont diagramme de Classe et composition ont été fondamentalement modifiés et remplacés par des diagrammes de définition de blocs et diagrammes de blocs internes. Ces extensions sont basées sur le mécanisme de profilage UML standard, qui inclut les stéréotypes UML, extensions de diagramme UML et la bibliothèques de modèles. Le mécanisme de profilage a été choisi sur d'autres mécanismes d'extension afin de tirer parti des outils existants basés sur UML pour la modélisation de systèmes.

En outre, SysML ajoute deux nouveaux diagrammes, ceux-ci étant diagrammes d'exigences et diagrammes paramétriques, et intègre de nouvelles capacités de spécification liens tels que l'allocation. La relation entre UML et SysML est illustrée dans 2.4.

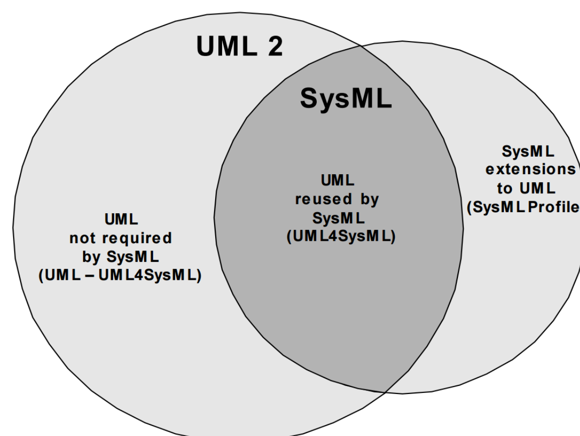


FIGURE 2.4 – La relation entre UML et SysML

2.1.2.4/ DIAGRAMME DE SÉQUENCE

Les diagrammes de séquence décrivent les interactions au sein d'un système en utilisant la communication entre des entités représentées par un rectangle à partir duquel les lignes de vie descendent. Une interaction est une communication basée sur l'échange de messages sous la forme d'appels d'opérations ou signaux disposés dans un ordre temporel. Les objets assument le rôle d'entités dans la communication. Le corps d'une ligne de vie représente le cycle de vie de son objet correspondant.

Un message est utilisé pour transmettre des informations et il peut être échangé entre deux lignes de vie dans deux modes possibles : synchrone et asynchrone. Il existe quatre types de messages : appel d'opération, signal, réponse et création/destruction d'objet. Le message est symbolisé par une flèche étiquetée pointant de l'expéditeur vers le récepteur. De plus, le diagramme peut contenir des exigences de temps, placées à gauche du diagramme, ce qui permet de spécifier combien de temps une opération a besoin pour terminer son exécution.

En plus des lignes de vie et des messages, les diagrammes de séquence peuvent définir d'autres constructions afin d'organiser les interactions modélisées. L'abstraction de l'unité d'interaction générale est appelée "fragment d'interaction", qui représente une généralisation d'un "fragment combiné". Ce dernier définit une expression du "fragment d'interaction" et consiste en un opérateur d'interaction et des opérandes d'interaction. Les fragments combinés permettent une illustration compacte des traces de messages échangés. Ils sont représentés par un cadre rectangulaire avec des lignes pleines et un petit pentagone le coin supérieur gauche montrant l'opérateur avec des lignes pointillées séparant les opérandes.

D'une part, un opérande d'interaction contient un ensemble ordonné de fragments d'interactions. D'autre part, les opérateurs d'interaction incluent, mais ne sont pas limités à, des opérateurs d'exécution conditionnelle (désigné par "alt"), des opérateurs de bouclage (désigné par "loop") et des opérateurs d'exécution parallèle (désigné par "par"). La Figure 2.5 illustre un sous-ensemble de la syntaxe des diagrammes de séquence [24].

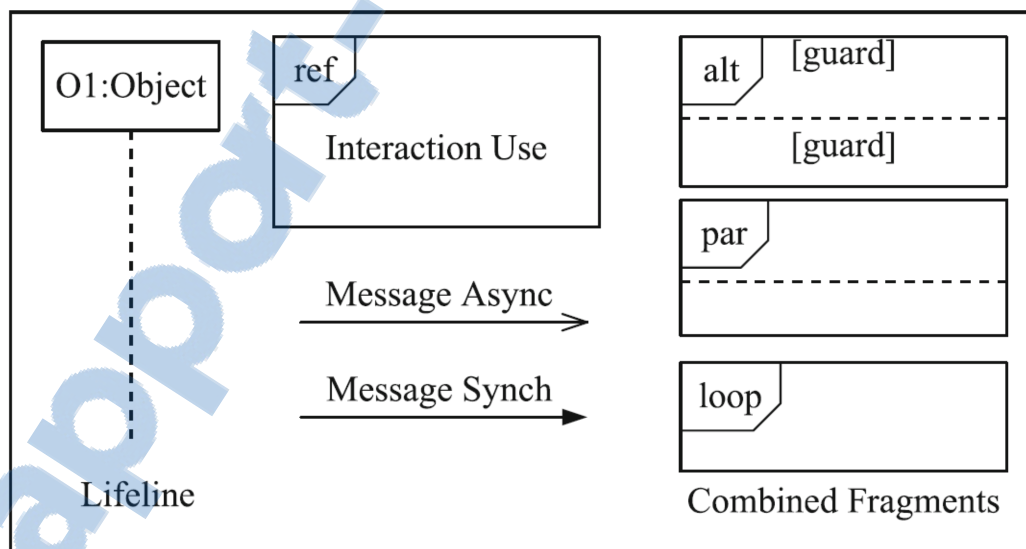


FIGURE 2.5 – Syntaxe du diagramme de séquence

2.1.3/ AUTOMATES TEMPORISÉS

Les automates temporisés reprennent le formalisme des automates auxquels s'ajoutent des horloges. Dans le formalisme des automates, un système est représenté par un ensemble de sommets décrivant les états du système, ces sommets étant reliés par des

arcs étiquetés par des événements.

Dans un automate temporisé, la dynamique temporelle est décrite grâce aux horloges qui permettent la définition de contraintes temporelles associées aux sommets ou aux arcs. Une contrainte temporelle associée à un sommet est appelée son invariant, celle associée à une transition est nommée une garde. Il est possible de rester dans un état aussi longtemps que son invariant est vrai. Une transition peut être déclenchée dès que sa garde est vraie. Lors de l'exécution d'une transition, la remise à zéro des horloges est possible [35].

On note $\phi(\mathcal{X})$ l'ensemble des contraintes d'horloges. Un automate temporisé est défini par un n-uplet $< \mathcal{S}, \mathcal{X}, \mathcal{L}, \mathcal{E}, \mathcal{I} >$ où :

- $\mathcal{S}$ est un ensemble fini de sommets,$s_0 \in \mathcal{S}$ étant le sommet initial ;
- $\mathcal{X}$ est un ensemble fini d'horloges ;
- $\mathcal{L}$ est un ensemble fini d'étiquettes ;
- $\mathcal{E}$ est un ensemble fini d'arcs ; chaque arc e est un n-uplet $(s, l, \varphi, \delta, s')$ tel que e relie $s \in \mathcal{S}$, état source, à $s' \in \mathcal{S}$, état destination, et est étiqueté par l'événement l ; la condition d'activation que doit satisfaire les horloges pour franchir la transition est représentée par la contrainte temporelle $\varphi \in \phi(\mathcal{X})$,$\delta \subseteq \mathcal{X}$ correspond à l'ensemble des horloges réinitialisées lorsque la transition est tirée ;
- $\mathcal{I} : \mathcal{S} \rightarrow \phi(\mathcal{X})$ associe à chaque sommet de l'automate temporisé une contrainte temporelle appelée l'invariant du sommet [35].
  Le méta-modèle de TAN est représenté sur la figure 2.6.

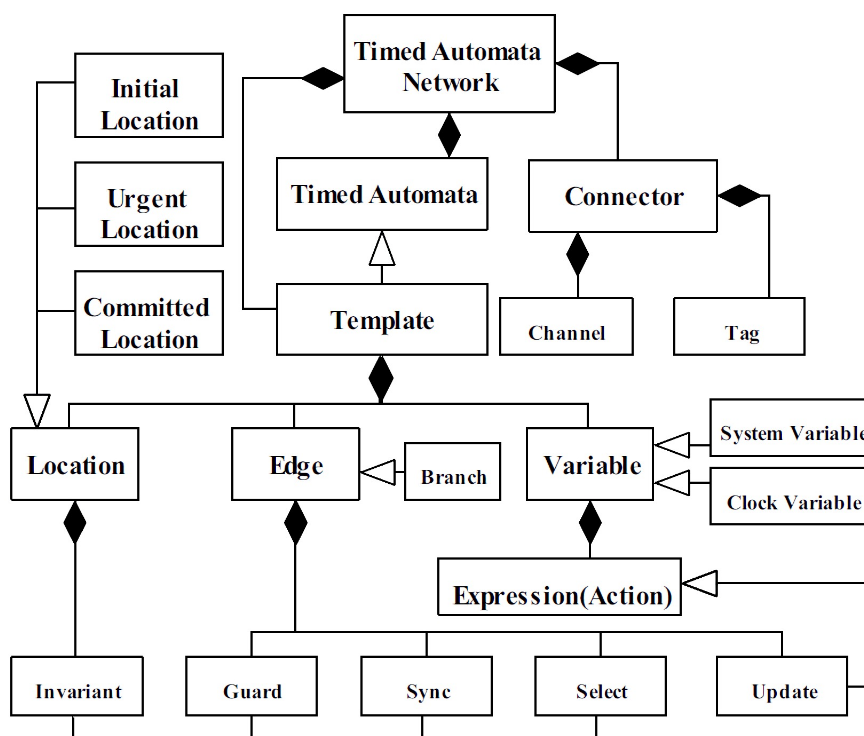


FIGURE 2.6 – Méta-modèle du TAN

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

La théorie des automates temporisés permet de décrire un système complexe comme un produit d'automates, ce qui autorise la construction modulaire du système global. Chaque automate décrit un sous-système et la synchronisation entre les automates se fait par le biais d'événements de synchronisation.

Lors de la définition des composants, il est alors nécessaire de distinguer les événements internes décrivant la dynamique propre et asynchrone du sous-système, des événements externes qui provoquent l'évolution simultanée de tous les sous-systèmes partageant ces événements [35].

2.2/ Ingénierie dirigée par les modèles - IDM

2.2.1/ Définitions

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 programmation classique. Il s'agit d'une forme d'ingénierie générative dans laquelle tout ou partie d'une application est engendrée à partir de modèles.

Un modèle est une abstraction, une simplification d'un système qui est suffisante pour comprendre le système modélisé et répondre aux questions que l'on se pose sur lui. Un système peut être décrit par différents modèles liés les uns aux autres.

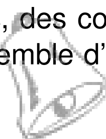
L'idée phare est d'utiliser autant de langages de modélisation différents (Domain Specific Modeling Languages – DSML) que les aspects chronologiques ou technologiques du développement du système le nécessitent. La définition de ces DSML, appelée métamodélisation, est donc une problématique clé de cette nouvelle ingénierie.

Par ailleurs, afin de rendre opérationnels les modèles (pour la génération de code, de documentation et de test, la validation, la vérification, l'exécution, etc.), une autre problématique clé est celle de la transformation de modèle.

En résumé, IDM couvre le cycle de vie complet de l'application , elle est basée sur deux principaux concepts : la métamodélisation et la transformation de modèles qui sont toujours des domaines de recherche [27].

2.2.2/ La métamodélisation

La métamodélisation est une activité qui consiste à définir le métamodèle d'un langage de modélisation. Il s'agit non seulement de produire des métamodèles mais aussi de définir la sémantique du langage, de mettre en œuvre des analyseurs, des compilateurs, des générateurs de code et plus généralement, à construire un ensemble d'outils exploitant les métamodèles [31].

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES


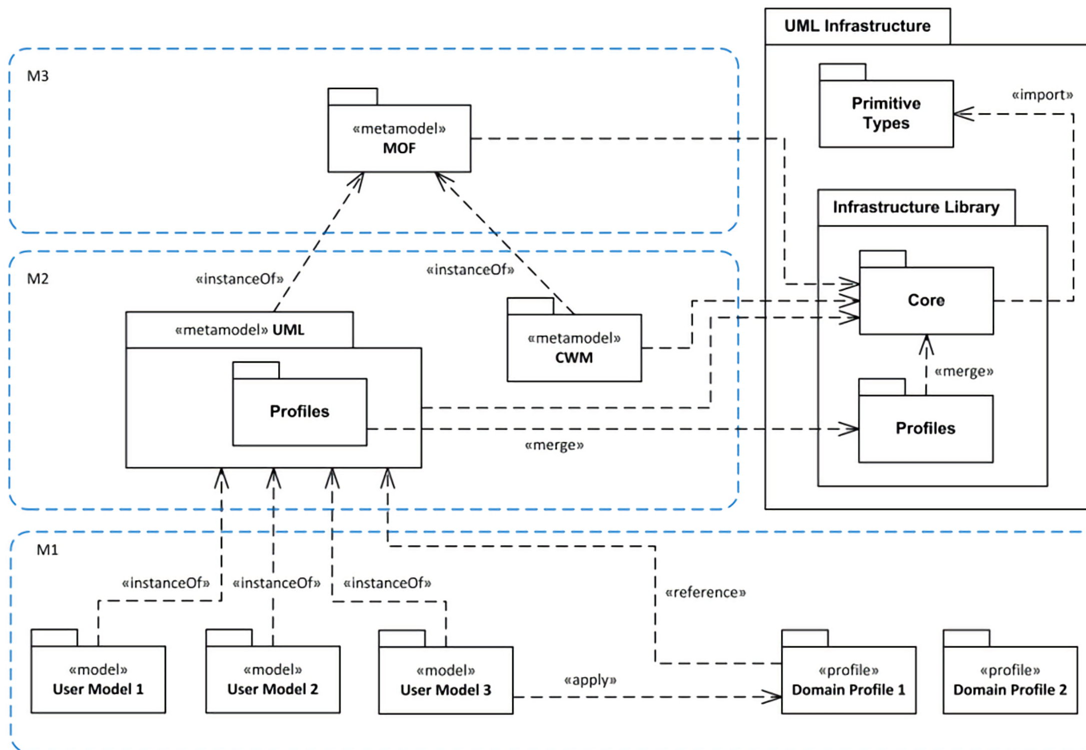


FIGURE 2.7 – Les couches de Métamodelisation

2.2.3/ TRANSFORMATION DE MODÈLES

La transformation de modèles est au cœur de l'IDM et y joue un rôle important. Elle peut être définie comme une génération d'un modèle cible à partir d'un modèle source, suivant une définition de transformation qui est un ensemble de règles de transformation qui décrivent comment un modèle dans un langage source peut être transformé en un modèle dans un langage cible.

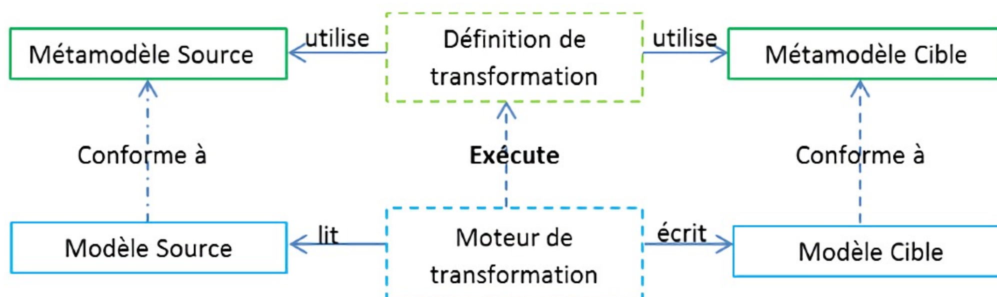


FIGURE 2.8 – Concepts de base de transformation de modèles

Une règle de transformation est une description de la manière d'une ou de plusieurs structures du langage source peuvent être transformées à une ou à plusieurs structures du langage cible.

Pour que la transformation soit appliquée à maintes reprises, elle doit être appliquée indépendamment du modèle source [31]. Les concepts de base de transformation sont présentés dans la Fig 2.8.

2.2.3.1/ Pourquoi la transformation de modèles

L'objectif principal et final de la transformation de modèles dans le contexte de l'IDM est la génération de code, mais la transformation de modèles couvre plusieurs activités parmi eux :

Vérification de modèle La vérification d'un modèle est le synonyme de modèle correct, c'est à dire le vérifier syntaxiquement et sémantiquement. L'analyse syntaxique vérifie si le modèle est conforme à un métamodèle et s'il respecte les conditions "contraintes" imposées par le langage tandis que l'analyse sémantique vérifie les propriétés dynamiques d'un domaine.

Validation de modèle La validation est de vérifier si un modèle répond aux exigences, ou d'une autre façon de vérifier si un modèle est fiable. La simulation et le test de modèle sont utilisés pour vérifier si un modèle est valide.

2.2.3.2/ Approches de transformation de modèles

Les approches de transformation de modèles sont classées en deux grandes familles :

les approches de modèle au code (Model to Text M2T) où le code est considéré comme un modèle spécifique et les approches de modèle à modèle (Model to Model M2M) [31]. Le tableau 2.1 ci-dessous, illustre les approches de transformation de modèles.

Approches de transformation de modèles	
M2M	*M2T*
Dirigée par la structure Manipulation directe	Parcours de modèles (programmation)
Approche relationnelle Transformation des graphes Hybride	Template

TABLE 2.1 – Les approches de transformation de modèles

2.2.3.3/ Transformation Model to Text - M2T

Cette classe est caractérisée par la méthode de génération du code source qui se base soit sur le parcours du modèle soit sur l'identification de canevas (templates).

Génération de code par parcours de modèle C'est l'approche de génération de code la plus basique qui consiste à parcourir la représentation du modèle en entrée et a écrire un code en sortie. L'exemple le plus connu avec cette approche est l'environnement orienté objet Jamda dont les modèles UML sont représentés par des classes où une API manipule les modèles et un outil est utilisé pour la génération de code.

Génération de code par template C'est l'une des approches la plus utilisée pour la génération de code. Cette approche consiste à définir des canevas de modèles cibles. Ces canevas sont des modèles cibles paramétrés ou modèles templates. L'exécution d'une transformation consiste à prendre un modèle template et à remplacer ses paramètres par les valeurs d'un modèle source. D'une autre façon , elle consiste à utiliser comme RHS "Right Hand Side" (le côté droit de la règle) un texte fixe dans lequel certaines parties sont variables : elles sont renseignées en fonction des informations récupérées dans le modèle source LHS "Left Hand Side" (le côté gauche de la règle), et peuvent notamment faire l'objet d'itérations. Parmi les langages les plus utilisées, on a Xpand [7], Acceleo [5] [31].

2.2.3.4/ TRANSFORMATION MODEL TO MODEL - M2M

Cette catégorie porte sur la transformation de modèle à modèle. Les méthodes utilisées sont diverses et variés, parmi elles nous allons présenter les deux qui suivent :

Approche par manipulation directe Cette approche offre une représentation interne d'un modèle plus une API pour le manipuler. ils sont généralement implémentés dans un environnement orienté objet, qui peut également fournir une infrastructure minimale pour organiser les transformations. Mais les utilisateurs doivent implémenter les règles de transformations à partir de zéro en utilisant un langage de programmation comme Java. Des exemples de cette approche comprennent Jamda [4] et JMI (Java Metadata Interface est une API basée sur le MOF). L'approche par manipulation directe est de type impératif, elle est située à un bas niveau d'abstraction.

Approche relationnelle Elle se base sur les transformations de type déclaratif reposant sur les relations mathématiques où l'idée de base est d'indiquer les types d'éléments de la source et de la cible dans une relation et de les spécifier à l'aide des contraintes. Par définition, une telle transformation n'est pas exécutable telle quelle, mais son exécution est possible par l'adjonction d'une sémantique exécutable, par exemple au moyen de la programmation logique. Il n'y a pas d'effets de bord avec l'approche relationnelle, contrairement à l'approche par manipulation directe.

La plupart des transformations basées sur l'approche relationnelle permettent la multidirectionnalité des règles, et fournissent des liens de traçabilité. Elles n'autorisent pas la transformation sur place (source et cible doivent être distinctes). Des exemples sur cette approche sont RFP [13], MOF 2.0 [18], QVT [17] [31].

2.3/ Vérification et Validation

2.3.1/ Test et Simulation (Validation)

Test Le test consiste à effectuer des actions spécifiques afin de vérifier l'opérabilité,capacité de support ou capacité de performance d'un article lorsqu'il est soumis à des conditions contrôlées réelles ou simulées. Les résultats obtenus sont ensuite comparés à ceux prévus ou attendus. Cela implique souvent l'utilisation d'un équipement d'essai spécial ou instrumentation afin d'obtenir des données quantitatives précises pour l'analyse. bien que les tests soient bien connus et aient été largement utilisés dans le génie logiciel,ils permettent seulement la découverte tardive des erreurs, et certains types d'erreurs peuvent effectivement rester transparent [24].

Simulation La simulation est la méthode traditionnelle de vérification. Comme son nom l'indique, elle essaye de tester le bon fonctionnement d'un composant en le soumettant à un système réel d'évaluation. Clairement le succès de cette technique compte beaucoup sur la détermination et l'attachement de l'individu qui fait le contrôle. Aussi chaque fois qu'un changement est fait au code HDL (Hardware Description Language) on a besoin du même effort pour vérifier la simulation résultante. Mais, bien que la technique standard de vérification soit le test direct, cette méthode permet d'afficher la présence d'erreurs mais n'est pas capable de prouver avec certitude leur absence. Cette conclusion découle de plusieurs arguments dont figure principalement l'impossibilité d'énumérer toutes les entrées possibles [25].

2.3.2/ Méthodes Formelles (Vérification)

La vérification formelle est un processus qui permet de prouver qu'un système se comporterait en parfait accord avec sa spécification. Cela revient à utiliser des approches mathématiques qui permettent de montrer que l'implémentation satisfait la spécification. Dans ce cas une considération de tous les cas possibles est implicite. Ces dernières années, les méthodes formelles ne sont plus uniquement des thèmes de recherche, mais plutôt un concurrent potentiel, si ce n'est pas une solution de remplacement, de la vérification par simulation. Ces points de vue partent d'une correcte énumération des caractéristiques des méthodes formelles.

Les avantages des méthodes de vérification formelle comparées à la vérification par simulation sont nombreux. En effet, ces méthodes considèrent toutes les entrées possibles au système, vérifient la validité des propriétés du système mathématiquement, ne nécessitent pas une spécification des sorties du système prévue de plus elles permettent, pour certains outils, d'identifier les traces des erreurs s'il y a lieu.

La vérification formelle possède, elle aussi, certains inconvénients. En effet, elle demande un effort supplémentaire pour parvenir à une description complète et simple du système à vérifier. C'est qu'il est nécessaire de définir une spécification, d'une part, tenant en considération tous les détails du système, et d'autre part, assez simple à manipuler dans la phase de vérification. D'autre part c'est difficile de faire un raffinement sans perte des propriétés du système.

2.3.2.1/ MODEL-CHECKING

Issu du domaine des méthodes formelles, le model-checking est utilisé pour la vérification automatique de systèmes complexes représentés par des systèmes à événements discrets.

Lorsque le modèle d'un système est décrit sous la forme d'un automate temporisé, les propriétés sont exprimées à l'aide d'une logique temporelle.

Le problème du model-checking peut alors se résumer de la façon suivante : étant donné M un modèle du système et φ une propriété à vérifier, est-ce que M satisfait φ ? Les programmes de model-checking retournent une réponse binaire (la propriété φ est vérifiée ou ne l'est pas).

La logique la plus répandue pour la vérification d'automates temporisés est la logique TCTL (Timed Computation Tree Logic [22]) qui est une extension de la logique CTL [29] autorisant l'expression de contraintes temporelles dans les propriétés.

Les formules TCTL sont définies de manière inductive par la grammaire suivante définie dans :

$$f ::= p \mid x \in \mathcal{I} \mid \neg f \mid f_1 \vee f_2 \mid \exists \diamond_I f \mid \forall \diamond_I f$$

avec $p \in P$ un prédicat de base,$x \in X$ une horloge et I un intervalle de temps.

On trouve les abréviations $\forall \square f$ pour $\neg \exists \diamond \neg f$ et $\exists \square f$ pour $\neg \forall \diamond \neg f$. Intuitivement l'opérateur $\exists \diamond_I f$ signifie qu'il existe une exécution menant à un état satisfaisant la propriété f au temps $t \in I$. $\forall \diamond_I f$ signifie que chaque exécution possède un état où la propriété f est valide au temps $t \in I$. $\forall \square f$ signifie que tous les états sur tous les chemins d'exécution satisfont la propriété f. $\exists \square f$ signifie qu'il existe une exécution telle que tous les états sur le chemin d'exécution satisfont la propriété f.

Un automate temporisé ayant la particularité de posséder un nombre infini de comportements liés à la valuation de ses horloges, on se trouve confronté au problème de l'explosion combinatoire. Afin de résoudre ce problème, le model-checking symbolique représente des ensembles d'états à l'aide de structures de données efficaces comme les diagrammes de décision binaire (encore appelés BDD pour Binary Decision Diagram), ou les matrices de bornes (DBM pour Difference Bound Matrices).

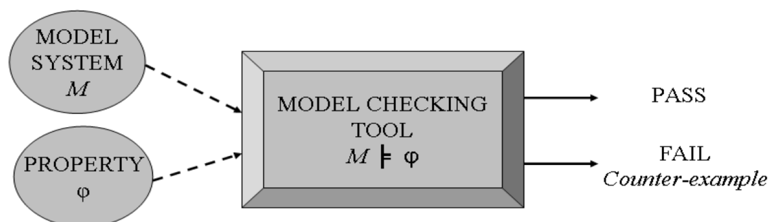


FIGURE 2.9 – Principe de vérification formelle avec Model Checker

Les algorithmes de model-checking symboliques travaillent sur ces ensembles d'états et déterminent, en fonction des propriétés à vérifier, soit l'espace atteignable, soit l'en-

semble des états satisfaisant une formule TCTL. la figure 2.9 représente le principe du modèle checker [35].

Langage de description UPPAAL Les outils les plus connus pour le model-checking symbolique sur les automates temporisés sont KRONOS [8] et UPPAAL [9]. Ce dernier régulièrement mis à jour par de nouvelles fonctionnalités et disposant d'une interface graphique agréable a été utilisé pour la réalisation de cette étude. UPPAAL étend le formalisme des automates temporisés présenté précédemment par de nouvelles caractéristiques et des notations particulières. La première extension concerne la définition possible de variables entières pouvant être associées à des états et modifiées lors du déclenchement de transitions. La seconde évolution touche à la synchronisation d'automates réalisée par le biais de canaux de synchronisation. Par exemple avec UPPAAL les notations a ! et a ? représentent respectivement des événements émis et reçus sur le canal a. Dans le système global, produit synchronisé de tous les automates, à chaque événement reçu doit correspondre un événement émis alors que les transitions non synchronisées correspondent à une évolution asynchrone du sous-système.

UPPAAL est composé d'un éditeur graphique, d'un simulateur graphique et d'un vérificateur de propriétés.

2.3.2.2/ THEOREM PROVING

Theorem Proving implique de vérifier la véracité des théorèmes mathématiques qui sont postulés ou inférés tout au long de la conception en utilisant un langage de spécification formelle. La procédure suivie pour prouver de tels théorèmes implique généralement deux composants principaux, à savoir un vérificateur de preuve et un moteur d'inférence. Cependant, alors que le premier peut être complètement automatisé dans la plupart des cas, le second peut exiger un guidage humain occasionnellement, empêchant ainsi l'automatisation de l'ensemble du processus.

De plus, il peut y avoir de rares cas où en raison du formalisme impliqué (par exemple, des références cachés circulaires conduisant à un paradoxe logique),la conjecture d'un théorème donné ne peut être ni prouvé ni réfuté. Les problèmes mentionnés ci-dessus représentent certaines des principales raisons pour lesquelles il n'est actuellement pas largement adopté pour effectuer la vérification et la validation [24].

3

VERS UNE VÉRIFICATION FORMELLE DES MODÈLES SYSML

3.1/ VUE GÉNÉRALE DE NOTRE APPROCHE

Dans cette partie, nous décrivons notre approche, la description du comportement et la vérification formelle du logiciel, qui est destinée à s'attaquer aux lacunes ci-dessous.

Premièrement, nous discutons des exigences de cette approche. Ensuite, nous fournissons un aperçu de cette dernière. Nous identifions les exigences suivantes pour l'approche de description du comportement et de la vérification formelle du logiciel :

- EX1 : définition formelle stricte. Pour éviter l'ambiguïté et les incohérences, l'approche devrait être formellement redéfinie avec une syntaxe et une sémantique précises.
- EX2 : description du comportement explicite. Afin d'améliorer la lisibilité et la compréhensibilité, l'approche devrait fournir une description explicite des comportements logiciels.
- EX3 : transformation de modèle souple. Afin de combler l'écart entre la modélisation semi-formelle et la modélisation formelle de logiciels, une méthode de transformation de modèle souple est nécessaire.
- EX4 : vérification formelle systématique. Pour assurer la fiabilité des logiciels, un mécanisme automatisé d'analyse et de vérification des logiciels est essentiel.

Autrement dit, nous recherchons une approche qui prend en charge la description explicite des comportements logiciels et fournit un mécanisme de vérification automatisé. L'approche doit intégrer les mérites de la modélisation semi-formelle et la modélisation formelle et combler l'écart entre eux. Nous proposons une approche pour répondre aux exigences ci-dessus. Comme le montre la figure 3.1, cette approche dépeint et vérifie les comportements logiciels en suivant trois étapes :

1. Analyse des exigences et modélisation du comportement :

   Dans cette étape, les concepteurs construisent des systèmes logiciels pour répondre aux exigences extraites du diagramme d'exigence SysML. Pendant ce temps, les concepteurs utilisent notre diagramme de séquence pour illustrer les comportements des systèmes logiciels anticipés.

Rapport-Gratuit.com

Notre diagramme de séquence est un modèle de description de comportement bidimensionnel. SD_{2D} hérite du méta-modèle des diagrammes de séquence SysML traditionnels en simplifiant sa syntaxe et sa sémantique. Ainsi, il peut représenter des interactions dynamiques entre deux objets avec une sémantique précise.

2. Transformation de modèles :

   Après avoir terminé la modélisation du comportement, nous devons transformer le modèle semi-formel du logiciel en un modèle formel pour une analyse plus approfondie et une vérification formelle. À cette fin, notre approche fournit un mécanisme de transformation de modèle souple.

   Cette approche de transformation de modèle établit une correspondance complète entre SD_{2D} et TAN (Timed Automata Network). Sur cette base, des algorithmes de transformation de modèle sont créés, réalisant la transformation automatisée de SD_{2D} en TAN pour combler l'écart entre la modélisation semi-formelle et la modélisation formelle des logiciels.

3. Vérification formelle :

   Sur la base du modèle TAN établi ci-dessus, nous pouvons simuler et analyser les comportements logiciels sur la plate-forme d'UPPAAL. De plus, il est possible de vérifier les spécifications des exigences décrites avec TCTL. Le modèle de comportement primaire du logiciel peut ainsi être raffiné et révisé en fonction des résultats de la vérification.

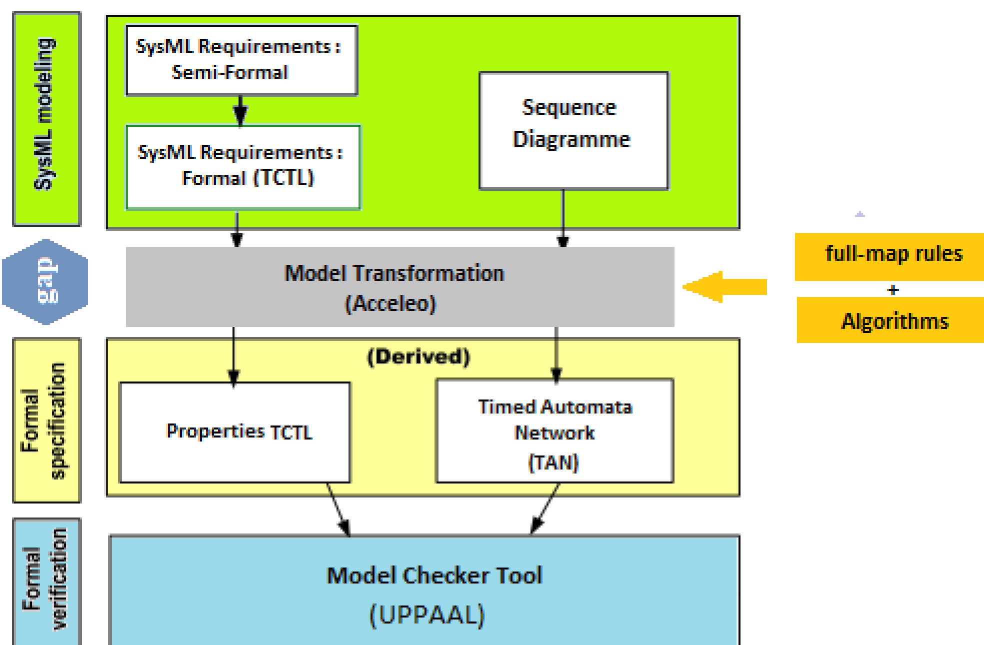


FIGURE 3.1 – Vue générale sur notre approche

3.2/ Des diagrammes de séquence aux automates temporisés

3.2.1/ Diagramme de Séquence Simplifié SD_{2D}

3.2.1.1/ Syntaxe du Diagramme de Séquence SD_{2D}

L'analyse et la vérification des modèles conçus est difficile en raison du manque de définitions formelles strictes dans les diagrammes comportementaux SysML. À cette fin, le diagramme de séquence est formellement redéfini dans un diagramme bidimensionnel appelé SD_{2D} dans cette approche.

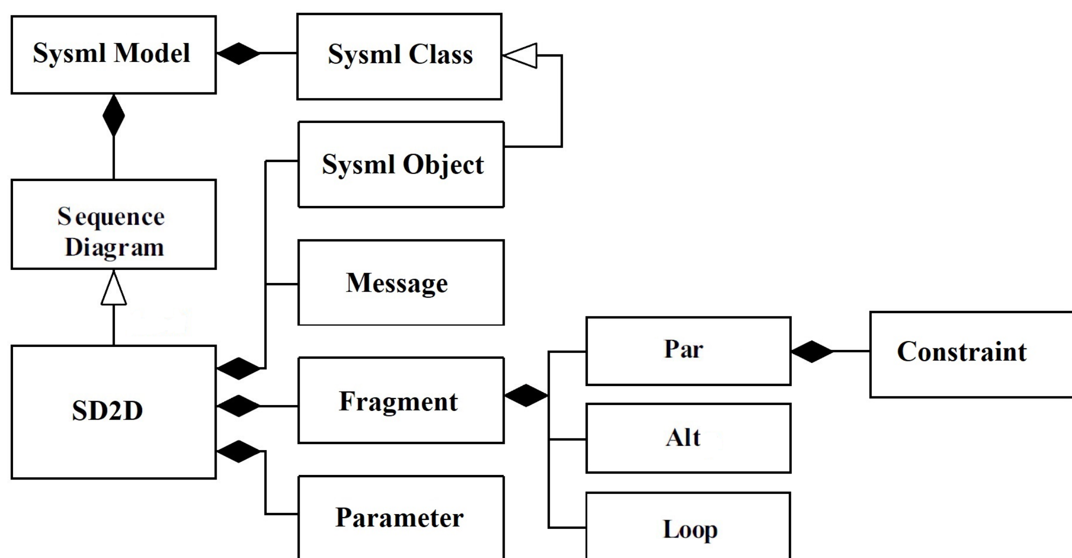


FIGURE 3.2 – Méta-modèle du diagramme de sequence SD_{2D}

SD_{2D} est défini comme un tuple :

$$SD_{2D} = (O, M, F, S, P)$$

où :

- O est un ensemble fini d'objets, qui sont des instances de classes dans SysML.
- M est un ensemble contenant tous les messages entre différents objets. Pour chaque message $m \in M_F$, m! représente l'événement d'envoi et m? représente l'événement de réception.
- F est un ensemble de fragments, où F = {alt, par, loop}. Un fragment est représenté sous la forme d'un ensemble avec une garde et contient tous les messages conditionnels sous cette garde (c'est-à-dire, FF = (garde, messages)). Le tableau 3.1 illustre des descriptions sémantiques des fragments ci-dessus.

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES
Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

- S est un ensemble contenant toutes les contraintes, Chaque fragment est attaché avec une contrainte ou plusieurs (gardes).
- P est un ensemble de paramètres décrivant les modifications du système logiciel et de son environnement d'exécution.

SD_{2D} décrit formellement le concept de fragments, qui est principalement introduit dans UML 2.0. Les fragments peuvent faciliter la description des chemins d'exécution des logiciels. Parmi les trois types de fragments ci-dessus, alt et par peuvent représenter les chemins d'exécution alternatifs tandis que loop peut décrire les chemins d'exécution de retour. Le concept de variable d'horloge est vu comme des éléments de première classe à représenter dans SD_{2D}, car la propriété temporelle est d'une importance vitale dans la plupart des systèmes logiciels.

Description sémantique des fragments dans SD_{2D}	
Type	*Description Sémantique*
alt	Le fragment alternatif (noté "alt") modélise le processus if .. then .. else.
par	Les fragments parallèles (notés "par")modélise le processus des modèles simultanés.
loop	Un fragment de boucle contient un processus de rétroaction de logiciel comportemental

TABLE 3.1 – Description sémantique des fragments dans SD_{2D}

3.2.1.2/ SÉMANTIQUE DE DIAGRAMME DE SÉQUENCE

Les difficultés rencontrées lors de la spécification de la dynamique d'un système avec SysML résident essentiellement dans :

- le manque de sémantique précise qui rend impossible la validation formelle du comportement des objets (par exemple, une situation de blocage, un état non accessible), ainsi que l'analyse des performances ;
- l'absence de moyen de vérification des incohérences entre les différents modèles. Par exemple, il serait intéressant de vérifier que le comportement défini au travers des diagrammes états/transition de chacun des objets d'un système implémente bien les interactions prévues dans les diagrammes de séquence de chaque cas d'utilisation dans lesquels ils interviennent ;
- chaque diagramme de séquence décrit un scénario, c'est à dire une séquence ou flot de messages déclenché par un stimulus et ne permet pas de représenter les contraintes d'interaction ou de synchronisation entre scénarios concurrents d'un même cas d'utilisation (ou de cas d'utilisation différents).

La génération automatique d'automate temporisé à partir d'un diagramme de séquence permet d'attribuer une sémantique opérationnelle à ces diagrammes ainsi que de

résoudre certains des problèmes rencontrés en SysML : validation de comportement (vérification de la cohérence des contraintes de synchronisation des messages reçus et émis par chaque objet), simulation, et la vérification formelle.

Le but recherché dans cette approche n'est donc pas d'étendre le pouvoir d'expression des Diagrammes séquence de SysML mais de leur attribuer une sémantique formelle.

3.2.2/ Automate Temporisé Étendu TAN_E

Le SD_{2D} et TAN sont tous deux des modèles pour représenter le comportement d'un logiciel. Ils possèdent de fortes similitudes dans leurs méta-modèles et leurs mécanismes de stockage et il est possible de convertir un SD_{2D} en TAN.

Premièrement, le SD_{2D} est très similaire à TAN dans le méta-modèle, une carte des correspondances complète peut donc être établie entre les deux modèles.

Deuxièmement, le SD_{2D} a le même mécanisme de stockage que TAN. Les deux types de modèles sont tous stockés dans le format de XML standard. Les similitudes ci-dessus permettent de transformer un SD_{2D} en TAN. Afin d'établir une correspondance complète de SD_{2D} à TAN, nous étendons et redéfinissons les automates temporisés et les TAN.

3.2.3/ Syntaxe et Définition de TAN_E

Un automate temporisé étendu (TAN_E) est défini comme suit :

$$TA_E = < L, l_0, S, A, E, I, B, V >$$

où :

- L est un ensemble d'emplacements ; l_0 est l'emplacement initial, $l_0 \in L$.
- S est un ensemble de contraintes attachées sur l'arête, $S \subseteq Guard$.
- A est un ensemble d'actions, y compris les actions d'entrée, les actions de sortie et les actions internes, $A = \{a!|a \in Chan\} \cup \{a?|a \in Chan\} \cup \{\tau\}$.
- E est un ensemble d'arêtes, $E \subseteq L \times A \times G(c, v) \times U(c, v) \times L, (l, a, g, u, l')$ représente la transition de l'emplacement l à l', qui est accompagné de garde g, affectation u et l'action a.
- I est un ensemble d'invariants, $I \subseteq Guard$ assigne des invariants aux emplacements.
- B est un ensemble de branches qui représente les transitions d'un emplacement à plusieurs emplacements selon différentes conditions.
- V est un ensemble de variables qui contient des variables système et des variables d'horloge.

De la même manière, plusieurs TA_E constituent le produit d'automates temporisés étendus, qui peuvent être formellement définis comme suit :

$$TAN_E \equiv <TA_{E1} \parallel TA_{E2} ... \parallel TA_{En}, C_\tau>$$

où, $TA_{E1} \parallel TA_{E2} ... \parallel TA_{En}$ représente le réseau d'automates temporisés étendus, C_τ est un ensemble de connecteurs joignant les automates interactifs.Il est composé de chaînes et de tags. Le premier est réalisé à l'aide de messages synchrones (syncs en abrégé) tandis que le second est un type de variables globales spécifiques. [26]

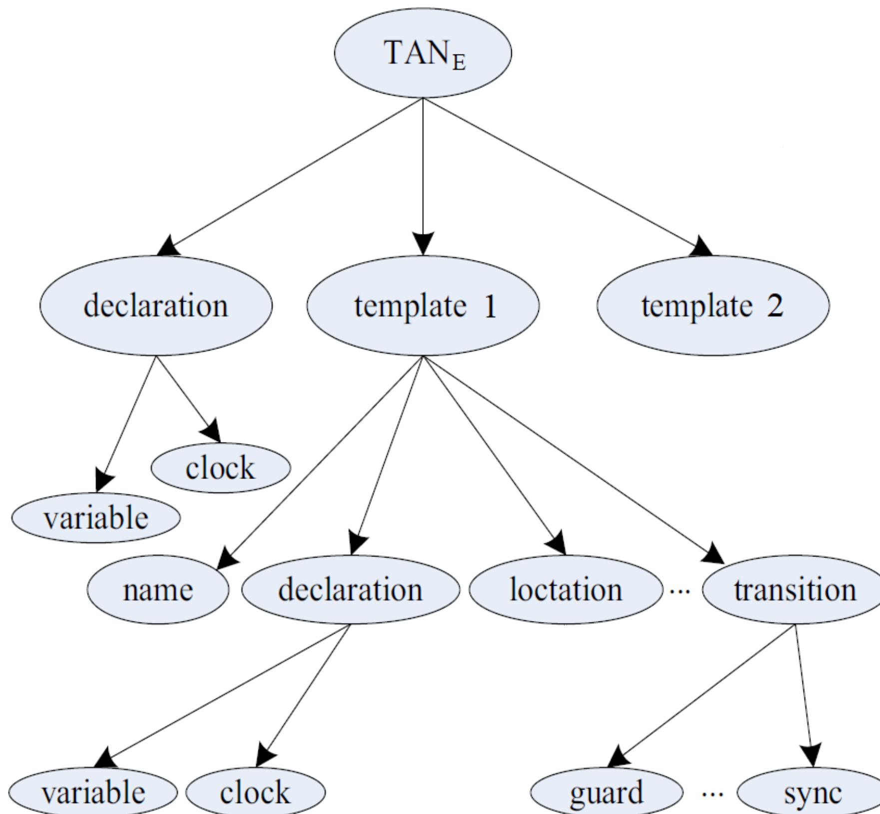


FIGURE 3.3 – Structure du TAN_E

3.2.4/ AUTOMATE TEMPORISÉ ET MODEL CHECKER

Dans le domaine de la modélisation logicielle, un automate temporisé est utilisé pour modéliser les comportements d'un seul objet. Cependant, un système logiciel est composé de divers objets en interaction.

Ainsi, un réseau d'automates temporisés (TAN) est proposé, ce qui est représenté comme un réseau de plusieurs automates temporisés en parallèle : TAN = TA1 || TA2 || ... TAn. Le modèle est en outre étendu avec des variables discrètes bornées, qui sont lues, écrites et soumises à des opérations arithmétiques communes. Différents automates interagissent à travers les canaux et les variables discrètes.

Afin d'analyser et de vérifier automatiquement les propriétés des systèmes logiciels, des vérificateurs de modèles automatisés sont créés, par exemple UPPAAL. Le but principal d'un vérificateur de modèle est de le vérifier par rapport à une spécification d'exigence.

Comme le modèle, la spécification d'exigence doit être exprimée dans un langage formellement bien défini et lisible par la machine, et UPPAAL utilise une version simplifiée de Timed Computed Logiques temporelles (TCTL)

3.3/ Règles de transformation de SD_{2D}

Afin de réaliser une transformation en douceur de SD_{2D} à TAN_E, nous avons étudié les correspondances entre leurs métamodèles et établi un ensemble de règles de mappage entre eux. Comme le montre le tableau 3.2, un SD_{2D} peut être mappé à un TAN_E. [26]

- Un objet dans SD_{2D} correspond à un template dans TAN_E ;
- Un état dans SD_{2D} correspond à un emplacement dans TAN_E ;
- Les messages interactifs dans SD_{2D} peuvent être mappés pour être des synchronisations dans TAN_E
- Les contraintes de SD_{2D} correspondent aux gardes dans TAN_E ;
- Les fragments de *Alt* et *Par* peuvent être décrits en utilisant des embranchements dans TAN_E ;
- Le fragment *Loop* peut être illustré avec un tag dans TAN_E ;
- Finalement, les paramètres dans SD_{2D} peuvent être mappés aux variables dans TAN_E.

Règles de mappage de SD_{2D} à TAN		
Règle	*Élément du SD_{2D}*	*Élément de TAN*
1	SD_{2D}	TAN
2	Objet	Template
3	État	Location
4	Message	Synchronisation
5	Contrainte	Guarde
6	Alt/Par	Embranchement
7	Loop	Étiquette/Tag
8	Paramètre	Variable

TABLE 3.2 – Règles de mappage de SD_{2D} à TAN

3.4/ Simulation et Vérification par TAN_E

Une fois que nous avons établi le modèle de TAN_E, nous pouvons maintenant vérifier les propriétés du système modélisé à l'aide du vérificateur UPPAAL. Le vérificateur de modèle UPPAAL décrit le logiciel système comme un modèle d'automate S, et formule les propriétés comme des expressions logiques F.

Ainsi, le problème de la vérification formelle est converti en un problème mathématique décrit comme S $\models$ F ?. Dans notre cadre de séquence formelle, nous utilisons les expressions logiques de TCTL pour spécifier les états et les formules de chemin.

Après avoir terminé la transformation du modèle de SD_{2D} à TAN_E et terminé la spécification formelle des propriétés du logiciel, nous pouvons maintenant analyser et vérifier les comportements du logiciel.

3.4.1/ SIMULATION

Le simulateur d'UPPAAL peut être utilisé pour simuler le modèle TAN_E pour valider s'il se comporte comme prévu. Il peut fonctionner manuellement ou de manière aléatoire pour voir comment certains états sont joignables ou pour découvrir une impasse.

3.4.2/ VÉRIFICATION FORMELLE

Nous avons divisé les propriétés en quatre groupes, respectivement accessibilité, sécurité, vivacité et contraintes temporisées.

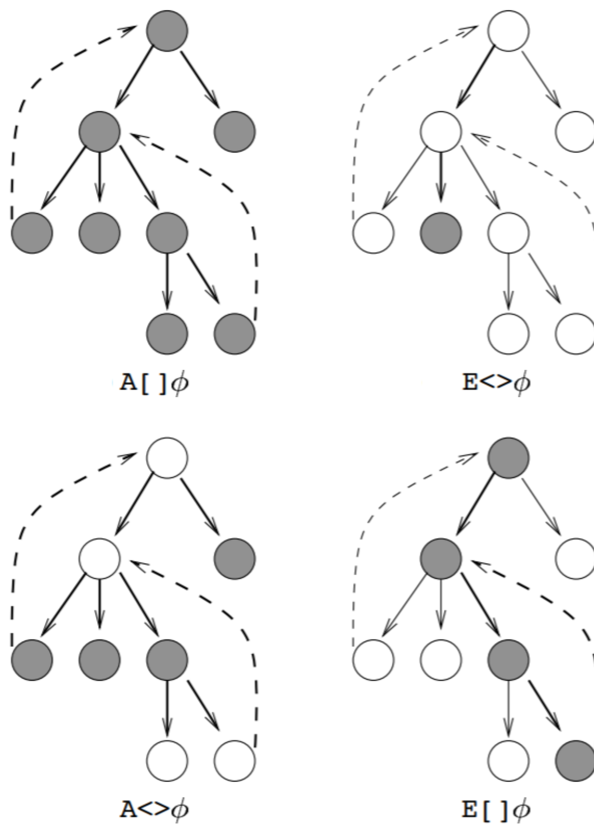


FIGURE 3.4 – Formules TCTL

- Les propriétés de sécurité sont sous la forme : "Quelque chose de mauvais n'arrivera jamais ", ce qui peut être positivement formulé comme "Quelque chose de bien est invariant". Ainsi, cette propriété peut être exprimé par "not E <> not ϕ" ou "A [] ϕ". Représentant de cette propriété est la vérification de l'impasse, qui peut être décrit comme "A [] pas deadlock".

- Les propriétés d'accessibilité sont la forme la plus simple des propriétés, qui demandent si une formule d'état donné ϕ, peut être satisfait par tout état accessible. Cette propriété peut être exprimé en utilisant la syntaxe E $<>$ ϕ, ce qui signifie qu'il existe éventuellement un chemin qui satisfait la formule d'état ϕ.
- Les propriétés de vivacité sont de la forme : "quelque chose finira par arriver ", qui peut être exprimé comme A $<>$ ϕ, ce qui signifie que ϕ est éventuellement satisfait. Cette propriété peut également être exprimé sous la forme de conduire à, par exemple, $\phi - > \psi$
- Les contraintes temporisées sont proposées pour décrire les exigences en temps réel lors de l'exécution du logiciel. L'expression de contraintes temporisées est composée de formules conventionnelles et d'une variable horloge, telle que, A [] template.location impliquent x ≧ 1 && x ≦ 2, montre que la transition est prise après un délai entre 1 et 2.

  Le vérificateur de UPPAAL peut être utilisé pour vérifier si le modèle TAN_E est correct par rapport aux propriétés précisées auparavant, et il fournit également les traces de roulement correspondantes pour une analyse plus poussée.

4

Formal-Sequence : un Plug-in Génération d'automates de vérification

4.1/ De SysML vers UPPAAL

Étant donné la complexité du réseau d'automates qui augmente exponentiellement en fonction de nombre d'objet et du nombre d'interaction, la construction manuelle du modèle devient une tâche plus difficile.

Formal-Sequence est un plug-in sous la plateforme Eclipse complétant l'outil de modélisation Papyrus [6] permettant la construction automatique des automates et la définition Formelle du diagramme de séquence sur lequel on souhaite appliquer notre plug-in.

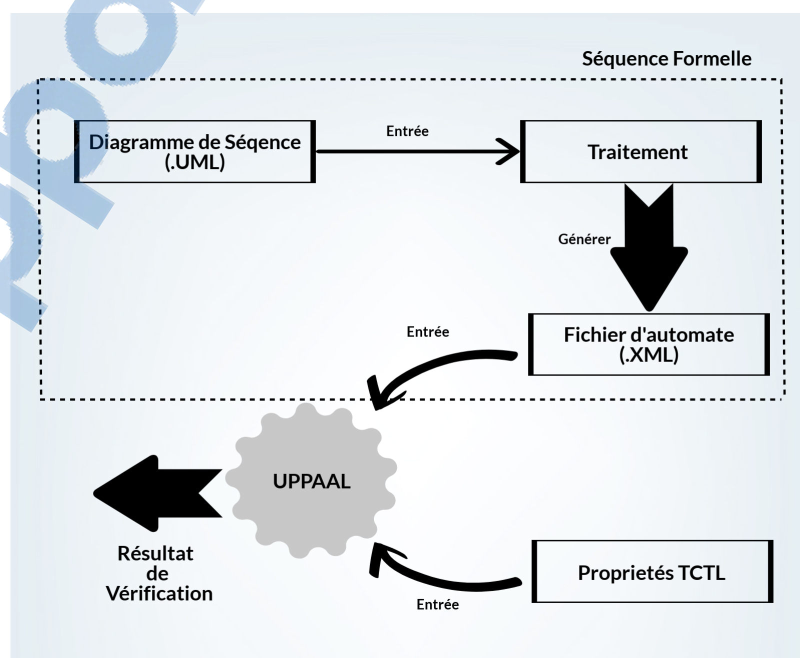


Figure 4.1 – Architecture de Formal-Sequence

Rapport-Gratuit.com

Une fois le modèle d'automates construit, la simulation et la vérification formelle du modèle peuvent être déclencher à l'aide d'une interface graphique dans l'outil Uppaal Cette section présente une vue globale du plug-in Formal-Sequence et son utilisation sur l'exemple simplifié du système ATM.

Le plug-in Formal-Sequence prend un modèle de diagramme de séquence avec l'extension (.UML) comme entrée, récupère les données à partir du modèle grâce au template Acceleo, puis applique les algorithmes de transformation, le résultat est un fichier Xml contenant la structure du réseau d'automate temporisés convenable au modèle d'entré. Comme indiquer dans la figure 4.2

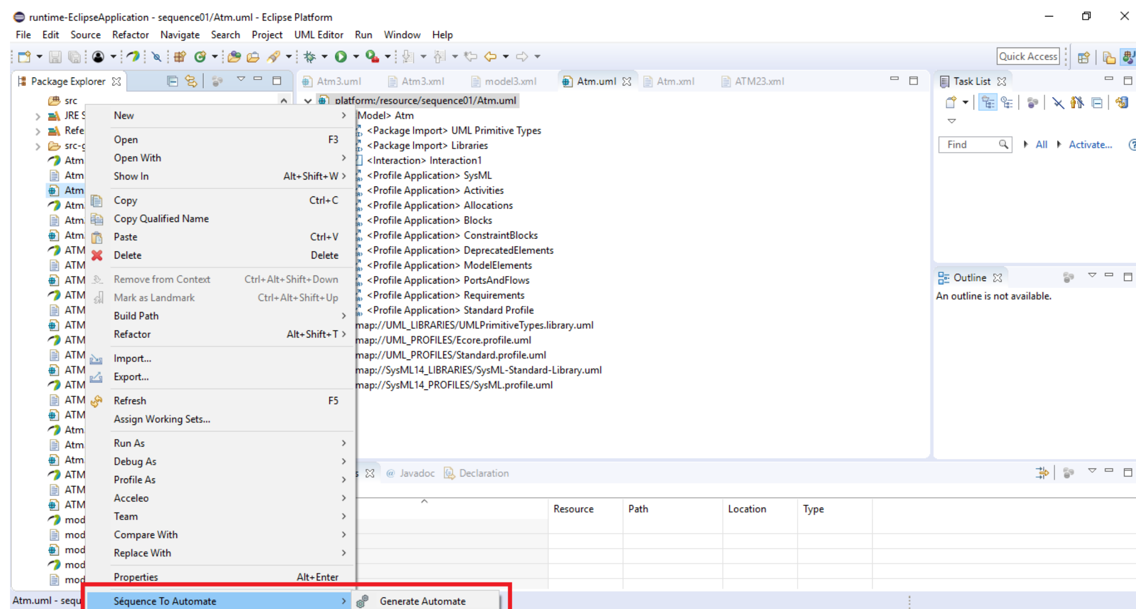


FIGURE 4.2 – Capture d'écran du plug-in Formal-Sequence

4.2/ ALGORITHMES DÉVELOPPÉS

Puisque chaque élément de SD_{2D} a été mappé à un élément de TAN_E, le SD_{2D} peut être mappé à un TAN_E. Sur la base de la carte complète des correspondances, cette section présentera l'approche automatisée de la transformation du modèle, en réalisant la transformation de SD_{2D} en TAN_E. Cette approche fonctionne suivant deux étapes : extraction d'informations, construction de l'automate.

4.2.1/ EXTRACTION DES DONNÉES

Le modèle SysML est prit directement depuis le schéma sur Papyrus. Nous devons d'abord extraire les éléments d'information depuis le SD_{2D}, tels que les objets, états, messages, etc à l'aide du template Acceleo. L'algorithme d'extraction de l'information est montré ci-dessous.

Premièrement nous traversons tout le SD_{2D} et récupérons tous les messages qui sont stockés dans une structure de donnée de type Array List.

Un deuxième passage sera fait pour récupérer les fragments, les messages qu'ils contiennent et les gardes pour chaque fragment, le type de fragment et les messages inclus seront stockés dans des Array List différente de celle pour les gardes.

Une fois cette étape terminée, nous pouvons ensuite passer à la construction des automates. Une description de l'algorithme d'extraction d'information est montrée ci-dessous.

Algorithme 1 : Extraction de l'information depuis SD_{2D}

Input : SD_{2D}.*uml*

Output : ArrayLists contenants les informations du SD_{2D}

1. **begin**
2. **Open** SD_{2D}.*uml* ; // Ouvre le diagramme SD_{2D}
3. **scan** (Element of SD_{2D}) ; // Parcours le diagramme SD_{2D}
4. **foreach** *Lifeline* in SD_{2D}.*uml*
5. ArrayList *Lifeline* ← *Lifeline*
6. **endfor**
7. **foreach** *Message* in SD_{2D}.*uml*
8. ArrayList *Message* ← *Message*
9. **endfor**
10. **foreach** *Constraint* in SD_{2D}.*uml*
11. ArrayList *Guard* ← *Guard*
12. **endfor**
13. **foreach** *Fragment* in SD_{2D}.*uml*
14. **foreach** *Message* in *Fragment*
15. ArrayList *Fragment_Message* ← *Message*
16. **endfor**
17. **endfor**
18. **foreach** *Parameters* in SD_{2D}.*uml*
19. ArrayList *Parameters* ← *Parameters*
20. **endfor**
21. **end**

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

4.2.2/ CONSTRUCTION DE TAN_E

Algorithm 2 : Construction de TAN_E

Input : ArrayLists resultants de l'algorithme 1

Output : TAN_E.*xml*

1. **begin**

2. System Declaration Global Variables ; // transformation des variables globales de SD_{2D} en déclaration en système TAN_E

3.Channel Declaration *M* ; // déclaration de la liste des synchronisation dans TAN_E à partir de la liste des messages de SD_{2D}

4. **foreach** *Lifeline* in ArrayList *Lifeline*

5. generateTemplate()

6. **foreach** *Message* in ArrayList *Message*

7. generateLocations()

8. generateTransition()

9. **endfor**

10. foreach *Message* in ArrayList *Fragment_Message*

11. generateLocations2()

12. **if** *Fragment_Message*.Type = "alt"

13. addGuard(ArrayList *Guard*)

14. generateTransition2()

15. **endif**

16. **if** *Fragment_Message*.Type = "par" or *Fragment_Message*.Type = "loop"

17. generateTransition3()

18. **endif**

19. **endfor**

20. **endfor**

21. Declare : template () // declaration des templates du système .

22. **end**

4.3/ ÉTUDE DE CAS

Dans les sections précédentes, nous avons présenté et décrit notre approche pour la modélisation explicite et vérification formelle du logiciel. Afin de valider cette approche, nous avons utilisé le cas d'étude ATM.

Nous voudrions utiliser notre plug-in Formal-Sequence pour modéliser et vérifier l'ATM. Plus concrètement, nous allons d'abord établir graphiquement le modèle de comportement de l'ATM en utilisant SD_{2D}, puis nous transférerons ce modèle semi-formel dans un modèle formel TAN_E.

Finalement, nous effectuerons une vérification formelle sur le TAN_E établi à partir du modèle de l'ATM et vérifierons ses propriétés spécifiques au domaine.

4.3.1/ Description du système ATM

Un guichet automatique (ATM) ou un guichet automatique bancaire (ABM) est un dispositif de télécommunication informatisé qui permet aux clients de l'établissement financier d'accéder à des transactions financières dans un espace public sans avoir besoin d'un employé ou d'un caissier de banque.

Sur la plupart des ATM modernes, le client est identifié en insérant une carte ATM en plastique avec une bande magnétique ou une carte à puce en plastique contenant un numéro unique et des informations de sécurité telles qu'une date d'expiration ou CVVC (CVV). L'authentification est fournie par le client en entrant un numéro d'identification personnel (PIN).

ATM est connu comme une machine automatique qui traite essentiellement des transactions entre une banque et son titulaire de compte, elle se connecte au serveur de la banque pour effectuer des transactions facilement et efficacement.

4.3.2/ Modélisation du comportement de l'ATM

Notre étude de cas présente les interactions entre un utilisateur et un ATM et montre les transitions d'état entre eux durant le processus de retrait d'argent.

Dans un premier temps, le client introduit sa carte et son numéro d'identification personnel (PIN), l'ATM vérifie la carte et le code PIN. Si la vérification échoue trois fois, la carte est saisie et l'opération prend fin. Sinon, l'opération continue.

Tout le processus de retrait de l'argent est représenté en utilisant le fragment de boucle et des actions alternatives de chaque objet est décrit en utilisant le fragment d'alt.

La figure 4.3 présente le comportement de ce système.

4.3.3/ Modélisation formelle de l'ATM

Dans cette section, nous allons transférer le modèle de comportement ci-dessus de l'ATM dans le modèle TAN_e en utilisant nos algorithmes de transformation introduits dans la section 4.2.

Pour commencer, nous allons extraire des informations essentielles (par exemple, les objets, les états, les messages et etc.) en utilisant l'algorithme d'extraction d'informations et les mapper aux éléments dans TAN_e. Le tableau 4.1 présente la relation de mappage du modèle SD_{2D} au modèle TAN_e de l'ATM.

Après avoir extrait des éléments ATM dans SD_{2D} et les avoir stockés, nous pouvons construire le modèle TAN_e de ATM utilisant l'algorithme de construction du modèle d'automate présenté à la section 4.2.

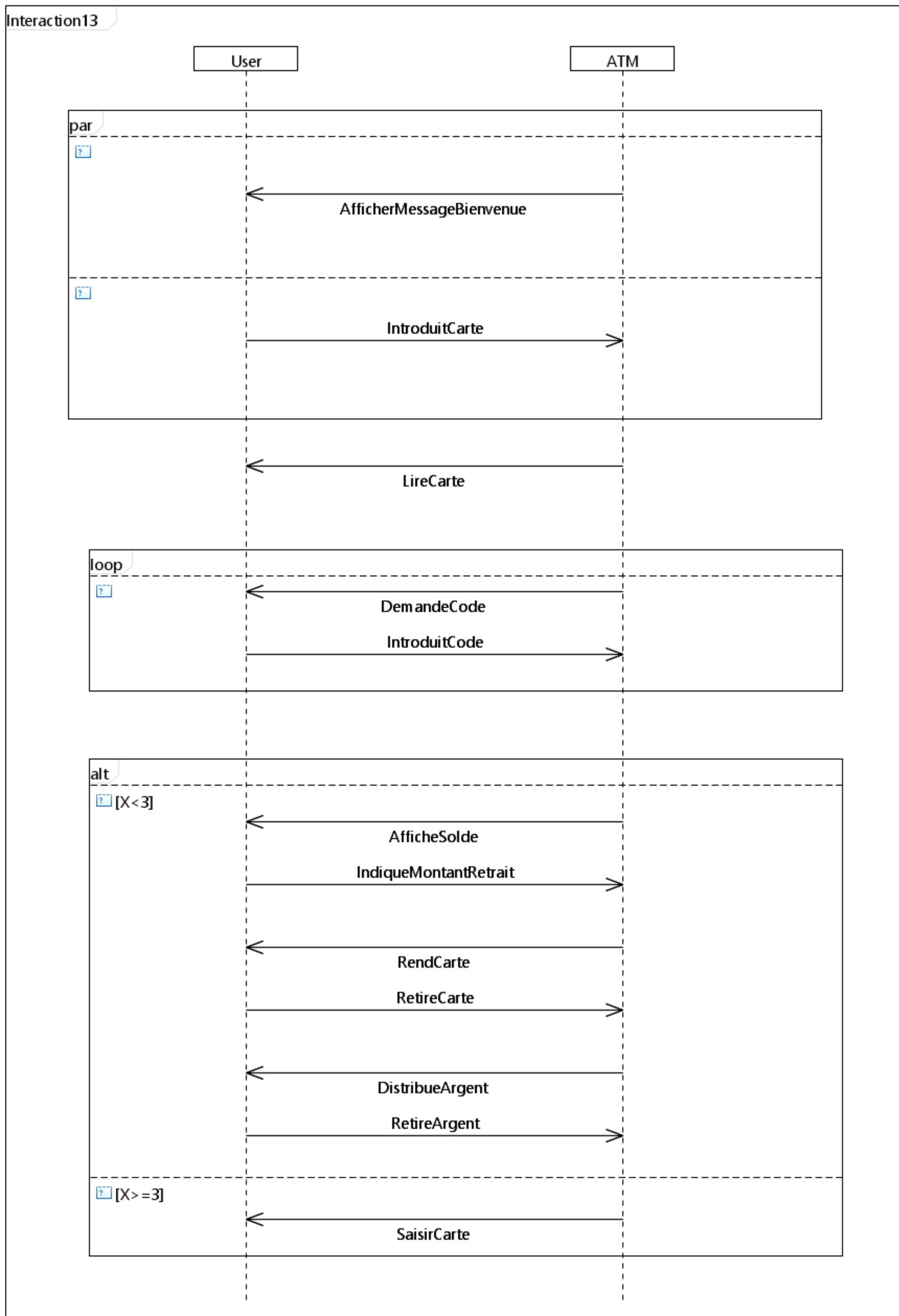


FIGURE 4.3 – Diagramme de comportement User/ATM

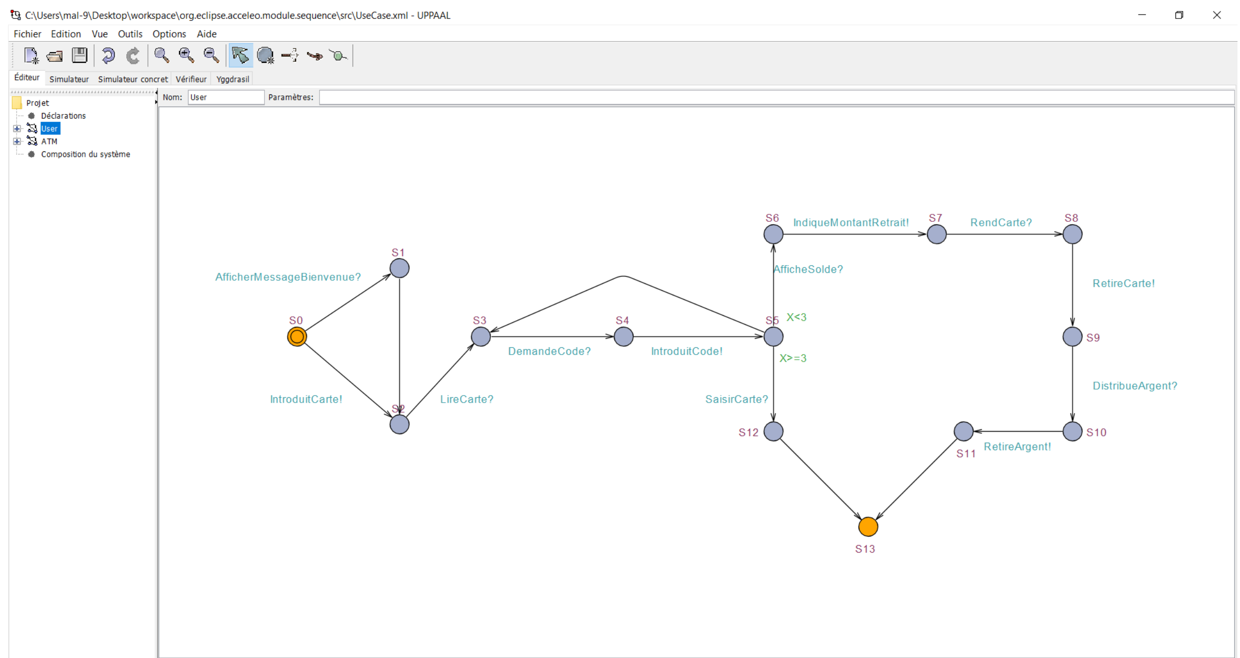


FIGURE 4.4 – Automate modélisant le comportement de l'utilisateur

Le modèle TAN_e nouvellement créé ne peut pas fonctionner directement sur la plate-forme de l'UPPAAL et nécessite d'autres la perfection. Certaines variables spécifiques doivent être initialisés et remis à zéro manuellement en utilisant la notion de "mise à jour" sur une syncronisation dans UPPAAL.

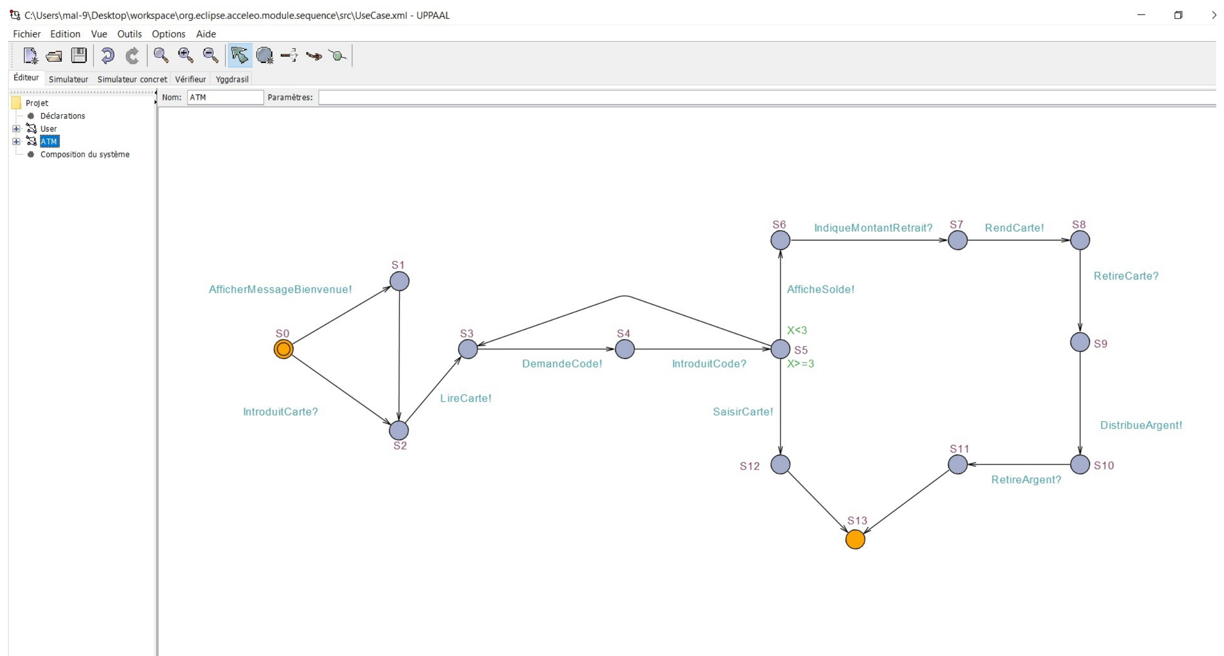


FIGURE 4.5 – Automate modélisant le comportement de l'ATM

Éléments du SD_{2D}	Éléments de TAN_e
Objets : User, ATM **États :** <S0,S1,S2,S3,S4,S5,S6,S7,S8, S9,S10,S11,S12,S13> ; <S0,S1,S2,S3,S4,S5,S6,S7,S8, S9,S10,S11,S12,S13> **Messages :** AfficherMessageBienvenue, IntroduitCarte, LireCarte, DemandeCode, IntroduitCode, AfficheSolde, IndiqueMontantRetrait, RendCarte, RetireCarte, DistribueArgent, RetireArgent, SaisirCarte ; **Paramètres :** X	**Templates :** User, ATM **Locations :** <S0,S1,S2,S3,S4,S5,S6,S7,S8, S9,S10,S11,S12,S13> ; <S0,S1,S2,S3,S4,S5,S6,S7,S8, S9,S10,S11,S12,S13> **Transitions :** AfficherMessageBienvenue, IntroduitCarte, LireCarte, DemandeCode, IntroduitCode, AfficheSolde, IndiqueMontantRetrait, RendCarte, RetireCarte, DistribueArgent, RetireArgent, SaisirCarte ; **Variables :** X

TABLE 4.1 – Relations de mappage de SD_{2D} à TAN_e

4.3.4/ VÉRIFICATION DE L'ATM

Sur la base de la modélisation formelle du système ATM, nous pouvons maintenant vérifier les propriétés du système ATM en utilisant le vérificateur UPPAAL.

4.3.4.1/ SIMULATION AVEC L'OUTIL UPPAAL

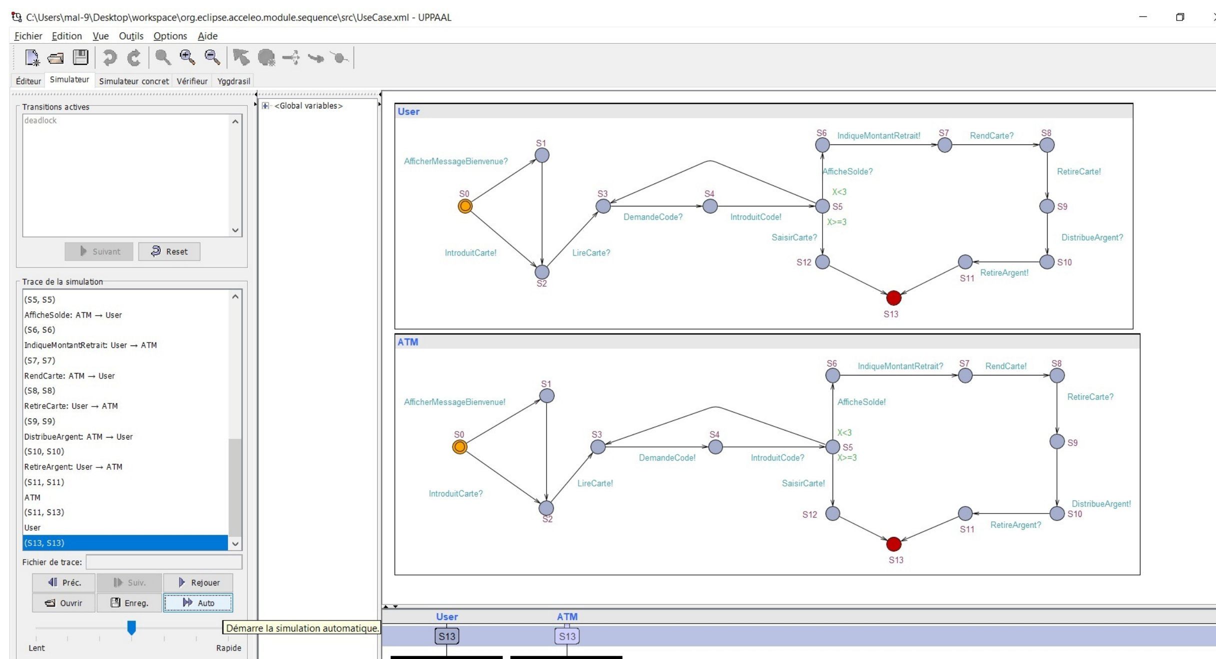


FIGURE 4.6 – Simulation automatique avec UPPAL

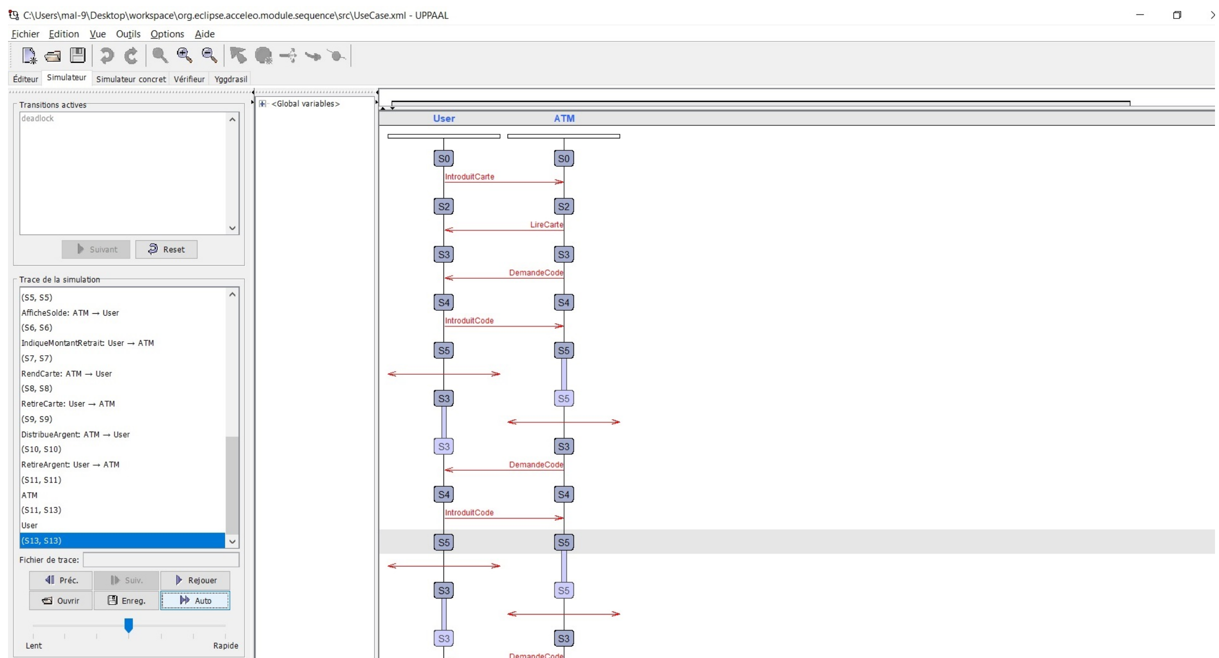


FIGURE 4.7 – Différentes intéractions de l'utilisateur et de l'ATM dans un seul chemin

4.3.4.2/ VÉRIFICATION AVEC L'OUTIL UPPAAL

La première propriété traite le problème de la sécurité. Le système ATM ne doit pas être dans une impasse à tout moment. Un blocage peut par exemple se produire lorsque le client et ATM attendent l'autre et aucun d'entre eux est en mesure d'envoyer une réponse. De telles situations sont évidemment à éviter :

A[] not deadlock

La deuxième propriété indique que l'ATM saisi le cartes et arrête les opérations si le code PIN n'est pas valide :

A<>ATM.S12 imply X>=3

La troisième propriété traite de la facilité d'utilisation de l'ATM système, et tous les états sont accessibles :

A<>ATM.S0
A<>User.S0
E<>ATM.S12
E<>User.S12
A<>ATM.S11
A<>User.S11

```
Status
Pics dutilisation de la mémoire permanente/virtuelle: {0}KB / {1}KB.
La propriété est satisfaite.
A<>User.S0
Vérification/noyau/temps total écoulé: 0s / 0s / 0,015s.
Pics dutilisation de la mémoire permanente/virtuelle: {0}KB / {1}KB.
La propriété est satisfaite.
A<> User.S13
Vérification/noyau/temps total écoulé: 0s / 0s / 0,015s.
Pics dutilisation de la mémoire permanente/virtuelle: {0}KB / {1}KB.
La propriété est satisfaite.
A<>ATM.S0
Vérification/noyau/temps total écoulé: 0s / 0s / 0,015s.
Pics dutilisation de la mémoire permanente/virtuelle: {0}KB / {1}KB.
La propriété est satisfaite.
A<>User.S13
Vérification/noyau/temps total écoulé: 0s / 0s / 0,006s.
Pics dutilisation de la mémoire permanente/virtuelle: {0}KB / {1}KB.
La propriété n'est pas satisfaite.
```

FIGURE 4.8 – Vérification des propriétés avec UPPAAL

5

CONCLUSION GÉNÉRALE

5.1/ CONCLUSION

Le diagramme de séquence SysML a les mérites d'une visualisation plus élevée et une meilleure lisibilité, tandis que le modèle d'automates possède les mérites d'une définition plus stricte et d'une meilleure vérifiabilité.

Combinant les forces complémentaires des deux modèles, notre projet propose ainsi une approche de vérification formelle implémentée dans un plug-in appelé Formal-Sequence.

Tout d'abord, les diagrammes de séquence sont simplifiés et formellement définis dans un diagramme bidimensionnel appelé SD_{2D}. Ensuite, un ensemble de règles de transformation est établi entre SD_{2D} et le réseau d'automates temporisés (TAN_e). Enfin, sur la base des relations de mappage, la transformation des SD_{2D} à TAN_e est établie.

Une vérification formelle peut ensuite être effectuée pour vérifier les propriétés spécifiques au domaine à l'aide de vérificateurs de modèles automatisés tels que UPPAAL.

Notre approche comble le fossé entre la modélisation semi-formelle et la modélisation formelle des comportements logiciels. Les résultats que nous avons obtenus montrent que l'approche proposée est satisfaisante et prometteuse.

Il est intéressant de noter que plusieurs travaux antérieurs ont déjà tenté de formaliser [36, 32] et de vérifier [26, 30] des diagrammes de séquence. Ces travaux tentent de combiner le diagramme de séquence (semi-formel) avec les techniques formelles (comme les réseaux de Petri [37, 34], les automates [33] et d'autres techniques formelles [21, 23]), qui intègrent les mérites de la modélisation semi-formelle et de la modélisation formelle et comble le fossé entre eux. La majorité des approches proposées se concentrent d'abord sur la transformation du diagramme de séquence en un modèle formel, puis utilisent un vérificateur de modèle automatisé pour simuler et vérifier les propriétés écrites sous forme d'une logique linéaire temporelle (LTL).

Rapport-Gratuit.com

5.2/ PERSPECTIVES

Quant au travail futur, il est double :

- tout d'abord, nous aimerions travailler sur l'amélioration du plug-in Formal-Sequence pour ajouter les différents fragments restants, le traitement de feedback ainsi que la mise à jour des variables de l'automate.
- deuxièmement, nous souhaiterions déduire automatiquement les propriétés TCTL à partir du diagramme d'exigences SysML.

BIBLIOGRAPHIE

[1] D. agnew, l. j. m. claesen, and r. camposano, editors. computer hardware description languages and their applications, volume a-32 of ifip transactions. north-holland, amsterdam, 1993.

[2] D. m. nicol. special issue on the telecommunications description language. sigmetrics performance evaluation review, 25(4) :3, 1998.

[3] G. booch. object-oriented analysis and design with applications, second edition. addison-wesley, reading, ma, 1997.

[4] http ://jamda.sourceforge.net.

[5] https ://www.eclipse.org/acceleo/documentation/.

[6] https ://www.eclipse.org/papyrus/documentation.html.

[7] http ://wiki.eclipse.org//xpand.

[8] http ://www-verimag.imag.fr/temporised/kronos.

[9] http ://www.uppaal.org.

[10] International council on systems engineering (incose) website. http ://www.incose.org/. last visited : February 2010.

[11] Object management group. omg systems modeling language (omg sysml) specification v1.0, september 2007. omg available specification.

[12] Object management group. omg systems modeling language (omg sysml) specification v1.0, september 2007. omg available specification.

[13] Object management group. request for proposal : Mof 2.0 query/views/transformations rep. http ://www.omg.org/cgi-bin/doc ?ad/2002-4-10.

[14] Object management group. unified modeling language : Infrastructure version 2.1.1, february 2007.

[15] Object management group. unified modeling language : Infrastructure version 2.1.1, february 2007.

[16] Object management group. what is omg-uml and why is it important ? http ://www.omg.org/ news/pr97/umlprimer.html. last visited april 2006.

[17] Omg. meta object facility (mof) 2.0 query/view/transformation specification version 1.2. http ://www.omg.org/spec/qvt/1.0/pdf/, april 2008.

[18] Omg. meta object facility (mof) core specification version 2.0, 2006.

[19] R. j. mayer, m. k. painter, and p. s. dewitte. idef family of methods for concurrent engineering and business re-engineering applications. technical report, knowledge-based systems, inc., 1992.

[20] S. ceri, p. fraternali, and a. bongio. web modeling language (webml) : A modeling language for designing web sites. computer networks (amsterdam, netherlands : 1999), 33(1–6) :137–157, 2000.

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

[21] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to Alloy. pages 69–86, 2010.

[22] Finite Automata and Clock Constraints. Timed Automata and TCTL. pages 1–35.

[23] L I Dan, L I Xiaoshan, L I U Zhiming, and Volker Stolz. Automated transformations from UML behavior models to contracts. 57(December) :1–2, 2014.

[24] Systems Engineering. *Verification and Validation in Systems Engineering*.

[25] Momtez Haddouk and Riadh Robbana. Nouvelle méthode de spécification et de vérification de systèmes temps réel sur puce. 2007.

[26] Deshuai Han, Jianchun Xing, Qiliang Yang, Hongda Wang, and Xuewei Zhang. Formal Sequence : Extending UML Sequence Diagram for Behavior Description and Formal Verification. pages 474–481, 2016.

[27] H A L Id. Ingénierie Dirigée par les Modèles (IDM) – État de l ' art Benoît Combemale To cite this version : HAL Id : hal-00371565 Ingénierie Dirigée par les Modèles (IDM) État de l ' art. (Idm), 2009.

[28] H A L Id. durable des systèmes socio-environnementaux : To cite this version : l ' optimisation durable des systèmes socio - Ingénierie des systèmes de décision en univers. 2017.

[29] R Kapur, Maurice Lousberg, Tony Taylor, Brion Keller, Paul Reuter, and Douglas Kay. Ctl the language for describing core-based test. pages 131–139, 02 2001.

[30] V Lima, C Talhi, D Mouheb, M Debbabi, and L Wang. Formal Verification and Validation of UML 2 . 0 Sequence Diagrams using Source and Destination of Messages. *Electronic Notes in Theoretical Computer Science*, 254 :143–160, 2009.

[31] Memoire D E Magister and Mecheri Nacera. Remerciements. 2015.

[32] Zoltán Micskei and Hélène Waeselynck. The many meanings of UML 2 Sequence Diagrams : a survey. pages 489–490, 2011.

[33] Iulian Ober, Susanne Graf, and Ileana Ober. Validation of UML Models via a Mapping to Communicating Extended Timed Automata. pages 127–128, 2004.

[34] Tony Spiteri Staines. Transforming UML Sequence Diagrams into Petri Nets. 10 :72–81, 2013.

[35] Yulong Zhao, Christine Largouët, Marie-odile Cordier, Yulong Zhao, Christine Largouët, and Marie-odile Cordier Ecomata. EcoMata : Un logiciel d ' aide à la décision pour améliorer la gestion des écosystèmes To cite this version : HAL Id : hal-00644577 EcoMata : Un logiciel d ' aide à la décision pour améliorer la gestion des écosystèmes. 2011.

[36] Yu Zhou. Towards a Formal Semantics for UML / MARTE State Machines Based. 28(248864) :188–202, 2013.

[37] Lianzhang Zhu and Fansheng Kong. Research of Automatic Conversion from UML Sequence Diagram to CPN Based on Modular Conversion. pages 95–96, 2012.

Résumé :

SysML est un langage de modélisation des systèmes rapidement émergeant comme une norme de facto utilisée pour les spécifications logicielles, les diagrammes de séquence SysML fournissent une technique graphique pour modéliser et décrire les comportements logiciels. Cependant, les diagrammes de séquence ne permettent pas d'analyser et de vérifier automatiquement les comportements logiciels dû au manque de sémantique rigoureuse. Pour assurer la fiabilité des systèmes logiciels, une description du comportement et une approche de vérification formelle sont proposées dans ce projet, en utilisant le diagramme de séquence SysML et un modèle d'automate.
Premièrement, une relation complète est établie entre le diagramme de séquence et le réseau d'automates temporisés à partir de certaines règles de transformation.
Ensuite, suivant la base des règles prédéfinies, la transformation du modèle sera établie.
La vérification formelle peut être ensuite effectué pour vérifier les propriétés du domaine basées sur le langage TCTL comme étant une logique expressive non ambiguë avec un vérificateur de modèles automatisés (UPPAAL).
Notre proposition comble le fossé entre la modélisation semi-formelle et la modélisation formelle logiciel, afin d’avancer l’étape de vérification le plus tôt possible dans le cycle de développement des systèmes hétérogène.
Notre approche a été illustrée avec une étude de cas d’ATM.

Mots-clés : SysML, Diagramme de Séquence, Vérification Formelle, Automate Temporisé, transformation de Modèle, UPPAAL.

Abstract :

Since SysML is a rapidly emerging system modeling language as a de facto standard used for software specifications, SysML sequence diagrams provide a visual technique for modeling and describing software behaviors. However, sequence diagrams can not be used to automatically analyze and verify software behavior due to the lack of formal semantics. To ensure the reliability of the systems software, a description of the behavior and a formal verification approach are proposed in this project, using SysML sequence diagram and timed automata model.
First, a complete relationship is established between the sequence diagram and the timed automata network from defined transformation rules.
Then, the model transformation will be established using the predifined rules.
Finally formal verification can then be performed to verify TCTL-based domain properties as unambiguous expressive logic with an Automated Model Checker (UPPAAL).
Our proposal bridges the gap between semi-formal and formal software modeling, and advance the verification step as early as possible in the heterogeneous systems development cycle.
Our approach has been evaluated on an ATM simulation system. The case study shows that this proposed approach is effective and in the behavior, description and formal verification of the software.

Keywords : SysML, Sequence Diagram, Formal Verification, Timed Automata, Model Transformation,UPPAAL.

ملخص

SysMLهي لغة نمذجة الانظمة سريعة الظهور كمعيار واقعي يستخدم لمواصفات البرمجيات ، وتوفر مخططات تسلسل SysML تقنية بيانية لنمذجة ووصف سلوكيات البرامج .ومع ذلك، لا تسمح المخططات التسلسلية بالتحليل التلقائي والتحقق من سلوك البرامج بسبب عدم وجود دلالات صارمة .لضمان موثوقية الأنظمة البرمجية، تم اقتراح وصف للسلوك ونهج تحقق رسمي في هذا المشروع، باستخدام مخطط تسلسل SysML ونموذج الأوتوماتا.
أولاً، يتم تأسيس علاقة كاملة بين مخطط التسلسل وشبكة الأوتوماتا الموقوتة من قواعد التحويل المعرفة في هذه التقنية.
بعد ذلك، بناءً على القواعد المحددة مسبقًا، سيتم إنشاء نموذج التحويل من مخطط تسلسل SysML الى نموذج الأوتوماتا.
يمكن إجراء التحقق الرسمي بعد ذلك للتحقق من الخصائص الديناميكية المستندة إلى TCTL كمنطق تعبير لا لبس فيه باستخدام مدقق النماذج المؤقتة (UPPAAL) .
ان اقتراحنا يعمل على سد الفجوة بين النمذجة شبه الرسمي ونمذجة البرامج الرسمي، من أجل دفع خطوة التحقق في أقرب وقت ممكن في دورة تطوير النظم غير المتجانسة. وقد تم توضيح نهجنا مع دراسة حالة ATM كمثال.

الكلمات المفتاحية: SysML ، مخطط تسلسل، التحقق الرسمي، الأوتوماتا الموقوتة، تحويل النموذج،UPPAAL.

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES