الجمهورية الجزائرية الديمقراطية الشعبية
République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université 20 août 1955 – SKIKDA-

وزارة التعليم العالي و البحث العلمي

جامعة 20 أوت 1955 سكيكدة

N° d'ordre :

Faculté des Sciences

Département d'Informatique

THESE

Présentée en vue de l'obtention du diplôme de
Doctorat en sciences

Spécialité : Informatique

Intitulée :

UNE APPROCHE INTÉGRÉE POUR LA MODÉLISATION ET L'ANALYSE DES SYSTEMES DISTRIBUÉS

Par : Mme Houda HAMROUCHE (Épouse ALLEG)

Soutenue publiquement le 07/03/2024.

devant le jury composé de :

M. Mensoul ABDELHAK	MCA à l'Université 20 août 1955-Skikda	Président
M. Allaoua CHAOUI	Professeur à l'Université AbdelHamid Mehri, Constantine 2	Rapporteur
M. Smaine MAZOUZI	Professeur à l'Université 20 août 1955-Skikda	Co-Rapporteur
M. Yacine KISSOUM	MCA à l'Université 20 août 1955-Skikda	Examinateur
M. Mourad BOUZENADA	MCA à l'Université AbdelHamid Mehri, Constantine 2	Examinateur
M. Nabil BELALA	Professeur à l'Université AbdelHamid Mehri, Constantine 2	Examinateur

2023-2024

DEDICACES

A mes chers parents

A mon cher mari

A ma petite fille bayene el houda

A toute ma famille

REMERCIEMENTS

D’abord, je remercie Dieu le Tout-Puissant de m’avoir donné la force, la patience et la volonté d’arriver au terme de ce travail.

Je tiens à remercier vivement mon directeur de thèse, le Pr. Allaoua CHAOUI, pour sa guidance, son soutien et ses conseils tout au long de cette recherche.

Je souhaite également adresser mes sincères remerciements à mon co-directeur de thèse, le Pr. Smaine MAZOUZI, pour sa contribution précieuse à cette thèse et ses commentaires éclairés qui ont grandement enrichi mon travail.

Je voudrais exprimer ma reconnaissance envers le président de jury, le Dr. Mensoul ABDELHAK, d’avoir accepté de présider la soutenence.

Mes remerciements vont également à mes examinateurs, le Dr. Yacine KISSOUM, le Dr. Mourad BOUZENADA, et le Pr. Nabil BELALA, pour avoir consacré leur temps et leur expertise à l'évaluation de ce travail de recherche.

Enfin, je tiens à remercier tous ceux qui ont contribué d’une manière ou d’une autre à la réalisation de cette thèse.

Résumé

Cette thèse s'inscrit dans le domaine de l'Ingénierie Dirigée par les Modèles (IDM), qui révolutionne la manière dont les systèmes complexes sont conçus, développés et maintenus, en s'appuyant sur la création de modèles abstraits pour représenter les aspects essentiels d'un système. UML (Unified Modeling Language) offre un cadre graphique puissant pour représenter la structure et le comportement des systèmes logiciels. La vérification des diagrammes UML doit être effectuée dès les premières phases du processus de développement logiciel afin de garantir la production d'un système fiable. Cependant, les modèles UML manquent d'une sémantique formelle, ce qui rend leur vérification difficile, en particulier lorsque nous modélisons un système critique où l'automatisation de la vérification est nécessaire. Par conséquent, la recherche de solutions pour attribuer une sémantique formelle aux modèles UML est une préoccupation majeure dans le domaine de l'ingénierie système. Communicating Sequential Processes (CSP) est un langage de spécification formel adapté pour décrire les interactions dans les systèmes concurrents et possède de nombreux outils de vérification automatique. Dans la littérature, de nombreux travaux de formalisation d'UML dans CSP ont été présentés, cependant, ils manquent d'automatisation ou ne traitent que partiellement les composants du diagramme formalisé.

Dans cette thèse nous proposons une approche intégrée UML 2.0/CSP visant à modéliser et à vérifier les aspects statiques et dynamiques des systèmes distribués. Nous concentrons notre attention sur les Diagrammes de Classe (UML 2.0 CD), les Diagrammes de Séquence (UML 2.0 SD), et les Diagrammes d'États-Transitions (UML 2.0 STM). Cette approche a pour objectif d'améliorer les formalisations existantes tout en proposant des outils visuels pour la modélisation et l'automatisation de la transformation des modèles UML 2.0 en spécifications CSP. Cela permet aux vérificateurs de modèles CSP existants d'effectuer les vérifications souhaitées. Notre approche repose sur la transformation de graphes et utilise l'outil de méta-modélisation multi-formalismes AToM^3. Elle propose trois méta-modèles des diagrammes UML 2.0 ainsi que trois grammaires de graphes. Pour la vérification des propriétés comportementales du système modélisé, telles que les impasses, les blocages vivants et le déterminisme, nous faisons appel au vérificateur de modèle Failures-Divergence-Refinement (FDR4). Pour illustrer cette approche et ses outils, nous proposons une étude de cas.

Mots clés: UML 2.0, CSP, Meta-modélisation, Grammaires de graphes, Vérificateur de modèle, AToM^3.

ABSTRACT

This thesis is part of the field of Model-Driven Engineering (MDE), which is revolutionizing the way complex systems are designed, developed, and maintained, relying on the creation of abstract models to represent essential aspects of a system.

UML (Unified Modeling Language) provides a powerful graphical framework to represent the structure and behavior of software systems. UML models must be verified in the early stages of software development process to guarantee the production of a reliable system. However, these models lack formal semantics, which makes their verification difficult, especially if we are modeling a critical system where the automation of verification is necessary. Therefore, finding solutions to assign formal semantics to UML models is a major concern in the field of system engineering. Communicating Sequential Processes (CSP) is a formal specification language that is suited for describing interactions in concurrent systems and has many automatic verification tools. In the literature, several attempts to formalize UML in CSP have been presented; However, they lack automation or only partially process components of the formalized diagram.

In this thesis, we propose an integrated UML 2.0/CSP approach aimed at modeling and verifying both static and dynamic aspects of distributed systems. We focus on Class Diagrams (UML 2.0 CD), Sequence Diagrams (UML 2.0 SD), and State Machine Diagrams (UML 2.0 STM). This approach aims to enhance existing formalizations while providing a visual tools for modeling and automating the transformation of UML 2.0 models into CSP specifications, enabling existing CSP model checkers to perform the desired verifications. Our approach is based on graph transformation and uses the meta-modeling multi-formalism tool AToM^3. It introduces three meta-models for UML 2.0 diagrams and three graph grammars. To verify the behavioral properties of the modeled system, such as deadlocks, livelocks, and determinism, we use the Failures-Divergence-Refinement (FDR4) model checker. To illustrate this approach and its tools, we provide a case study.

Keywords : UML 2.0, CSP, Meta-modeling, Graph grammar, Model checker, AToM^3.

ملـخــص

تندرج هذه الأطروحة في مجال الهندسة المعتمدة على النماذج (IDM), والذي أحدث ثورة في طريقة تصميم الأنظمة المعقدة, تطويرها و صيانتها, من خلال الإعتماد على إنشاء نماذج مجردة لتمثيل الجوانب الأساسية للنظام.
توفر لغة النمدجة الموحدة (UML) إطارا رسوميا قويا لتمثيل بنية و سلوك أنظمة البرمجيات. ينبغي التحقق من نماذج UML في مرحلة مبكرة من عملية تطوير البرمجيات لضمان إنتاج نظام موثوق. غير أن هذه النماذج تفتقد الى صيغة رسمية، مما يجعل التحقق منها صعبا، خاصة عندما نقوم بتصميم نظام حرج حيث تكون أتمتة التحقق ضرورية . لذلك، فإن إيجاد حلول لإسناد صيغ رسمية لنماذج UML يعد إنشغالا كبيرا في مجال هندسة النظام.
Communicating Sequential Processes (CSP) هي لغة توصيف رسمية مكيفة لوصف التفاعلات في الأنظمة المتنافسة والتي تتمتع بالعديد من أدوات التحقق التلقائي. في الأدبيات, تم تقديم العديد من الأعمال التي أعطت صيغة رسمية لنماذج UML فيCSP ,غير أنها إما تفتقد للأتمتة أو تعالج جزئيا مكونات النموذج المراد صياغته رسميا. في هذه الأطروحة, نقترح نهجا متكاملا UML2.0/CSP يسمح بنمذجة بنية و سلوك الأنظمة الموزعة و التحقق منها. نركز اهتمامنا على مخططات Classe (UML 2.0 CD), États-Transitions (UML 2.0 STM) و (UML 2.0 SD) Séquence. يهدف هذا النهج إلى تحسين الصيغ الرسمية الحالية مع تقديم أدوات مرئية للنمذجة و أتمتة تحويل نماذج UML 2.0 إلى توصيفات CSP , مما يسمح لمدققات نماذج CSP الموجودة بإجراء عمليات التحقق المطلوبة. يعتمد نهجنا على تحويل الرسومات البيانية باستخدام الأداة $ATOM^3$, كما يقدم ثلاثة نماذج ميتا لمخططات UML 2.0 و ثلاث قواعد للرسوم البيانية. للتحقق من الخصائص السلوكية للنظام المعني, كالطرق المسدودة, الإنسدادات الحية و الحتمية, نستخدم مدقق النموذج FDR4. لتوضيح هذا النهج و أدواته المقترحة, نقدم دراسة حالة.

الكلمات المفتاحية: UML 2.0, CSP ,النمذجة ميتا, قواعد الرسوم البيانية, مدقق النموذج ,$ATOM^3$

TABLE DES MATIERES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ABREVIATIONS

AFIS	Association Française d'Ingénierie Système
AGG	Attributed Graph Grammar System
ARC	Adelaide Refinement Checker
ASM	Abstract State Machine
ATM	Automated Teller Machine
AToM³	A Tool for Multi-formalism Modeling and Meta-modelling
CIM	Computation Independent Model
CSP	Communicating Sequential Processes
CSPM	The Machine-readable dialect of CSP
CWM	Common Warhouse Metamodel
DB	DataBase
FDR	Failures-Divergences Refinement
GROOVE	GRaphs for Object-Oriented VErification
HOL	Higher Order Logic
IDM	Ingénierie Dirigée par les Modèles
ISO	International Organization for Standardization
LHS	Left Hand Side
LOC	Lines Of Code
LOTOS	Language Of Temporal Ordering Specification
LTL	Linear Temporal Logic
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MSDL	Modelling, Simulation and Design Lab
OCL	Object constraint language
OMG	Object Management Group
OMT	Object-Modeling Technique
OOSE	Object-Oriented Software Engineering
PAT	Process Analysis Toolkit
PCTL	Probabilistic Computation Tree Logic
PIM	Platform Independent Model
PROMELA	Process Meta Language

PSM	Platform Specific Model
PVS	Prototype Verification System
RHS	Right-Hand Side
SADT	Structured Analysis and Design Technique
SBVR	Semantics Of Business Vocabulary And Business Rules
SPEM	Software Process Engineering Meta-model
TGG	Triple Graph Grammar
UML	Unified Modeling Language
UML 2.0 CD	Diagrammes de Classe UML 2.0
UML 2.0 SD	Diagrammes de Séquence UML 2.0
UML 2.0 STM	Diagrammes d'Etats-Transitions (State Machine Diagram) UML 2.0
VDM	Vienna Development Method
XSLT	eXtensible Stylesheet Language Transformations

CONTEXTE

De nos jours, le champ d'application des logiciels s'étend de manière impressionnante, touchant pratiquement tous les aspects de notre vie. L'expansion des domaines d'application s'accompagne d'une complexité croissante des systèmes logiciels. La demande des fonctionnalités toujours plus avancées a conduit à la création de systèmes de plus en plus sophistiqués, avec des architectures interconnectées et des exigences de performance élevées. Parallèlement, les équipes de développement logiciel ont augmenté en taille pour répondre à ces défis. La convergence de ces facteurs souligne la nécessité d'approches innovantes pour la conception, le développement, et la gestion des systèmes logiciels dans le contexte moderne.

L'ingénierie dirigée par les modèles (IDM) est un paradigme puissant qui révolutionne la manière dont les systèmes complexes sont conçus, développés et maintenus. En s'appuyant sur la création de modèles abstraits pour représenter les aspects essentiels d'un système. La promesse d'IDM réside dans sa capacité à accélérer le cycle de développement, à améliorer la qualité des produits et à faciliter l'adaptation aux évolutions technologiques, faisant de lui un domaine de recherche essentiel pour répondre aux défis complexes de l'ingénierie contemporaine.

La phase de conception occupe une place centrale dans le processus de développement logiciel. C'est à ce stade que les idées abstraites prennent forme pour devenir des logiciels. Il est important de détecter les erreurs de conception dès les premières phases du processus de développement logiciel afin de garantir la production d'un système fiable. Cela permet de détecter et de corriger les erreurs et les incohérences dès le début, avant d'atteindre des étapes avancées du développement. Donc, La vérification précoce des modèles évite la multiplication des problèmes au fil du temps, ce qui peut rendre leur résolution coûteuse et beaucoup plus complexe.

Les langages de spécification varient en termes de formalisme, allant de la spécification informelle, qui utilise un langage naturel ou une notation simplifiée, à la spécification semi-formelle et formelle, chacun offrant des avantages et des inconvénients spécifiques. Les langages semi-formels, comme l'Unified Modeling Language (UML) (UML, 2017), offrent une vision graphique avec une certaine rigueur. Ils sont largement

utilisés pour la modélisation de systèmes logiciels, leurs avantages résident dans la clarté visuelle et la réduction des ambiguïtés. Cependant, ils manquent de la rigueur mathématique des langages formels, ce qui les rend moins adaptés à la vérification formelle.

Les langages de spécification formels, tels que LOTOS (Language Of Temporal Ordering Specification) (Bolognesi & Brinksma, 1987), Z (Ruhela, 2012), et CSP (Communicating Sequential Processes) (Hoare, 1978), offrent un haut degré de précision et de rigueur mathématique. Ils sont particulièrement adaptés à la vérification formelle de systèmes critiques où la sécurité est cruciale. Leur avantages incluent la possibilité de détection précoce d'erreurs et d'ambiguïtés, ainsi que la garantie de conformité aux spécifications. Cependant, leur utilisation nécessite une expertise technique considérable et leur formalismes peut rendre les spécifications plus complexes à rédiger.

Le choix du type de spécification dépend des besoins du projet, de la complexité du système, des compétences de l'équipe de développement et des contraintes de temps. Les projets critiques exigent souvent une spécification formelle pour garantir la fiabilité, tandis que les projets moins critiques peuvent utiliser des spécifications semi-formelles ou informelles pour accélérer le processus de développement.

L'intégration des langages de spécification semi-formels et formels présente un avantage considérable dans le domaine du développement logiciel. Les équipes de développement peuvent tirer parti de la clarté et de la compréhensibilité des spécifications semi-formelles tout en garantissant la correction formelle grâce aux spécifications formelles. Cette combinaison permet de réduire les erreurs de conception dès les premières phases du développement, ce qui se traduit par des systèmes plus fiables et un processus de développement plus efficace.

Plusieurs approches ont été introduites pour intégrer des langages de spécification semi-formels et des langages de spécification formels. Nous citons à titre d’exemples, les travaux (Bouarioua et al., 2011), (Mozaffari & Harounabadi, 2011), (Russo, 2011), (Cunha et al., 2011), (Zhao et al., 2006), (Lima et al., 2009), (Hlaoui et al., 2017), (Vidal-Silva et al., 2018), (Custódio Soares et al., 2018) , (Messaoudi et al., 2019), (Chabbat et al., 2020).

Cette thèse s'inscrit dans le contexte de l'intégration des langages de spécification formels et semi-formels, en appliquant les principes de l'IDM. Dans ce cadre, nous combinons les langages UML 2.0 et CSP pour proposer une approche intégrée UML 2.0/CSP en vue de la modélisation et de l'analyse des systèmes distribués.

PROBLÉMATIQUE ET OBJECTIF

Le génie logiciel moderne repose sur des outils et des méthodes avancés pour la conception et l'analyse des systèmes logiciels complexes. Une méthode consiste principalement en un langage de modélisation et en un processus. Le langage de modélisation est la notation utilisée par la méthode pour exprimer les conceptions, tandis que le processus décrit les étapes à suivre pour les réaliser (Fowler, 2003).

L'architecture dirigée par les modèles (MDA), est une variante particulière de l'IDM. Elle repose sur l'idée de séparer la spécification du fonctionnement d'un système des détails de son implémentation. Dans l'approche MDA, le processus de développement logiciel commence par définir le système de manière abstraite et indépendante de toute plateforme d'implémentation. Ensuite, il consiste à décrire les différentes plateformes possibles pour mettre en œuvre ce système logiciel. Enfin, une plate-forme spécifique est choisie, et la spécification du système est transformée en une spécification adaptée à cette plateforme sélectionnée. Un méta-modèle est un modèle décrivant un langage d'expression de modèles. Tout modèle doit être conforme à son méta-modèle. Cette relation de conformité constitue l'aspect distinctif de l'IDM, nous permettant de construire correctement les modèles pour leur appliquer des transformations automatisées. La transformation de modèle est effectuée en mappant un modèle source à un modèle cible équivalent. Les caractéristiques des modèles sont identifiées par leur méta-modèles. Ensuite, le mappage est défini comme une traduction entre les méta-modèles source et cible.

Actuellement, UML 2.0 est l'un des langages de modélisation les plus utilisés dans l'industrie et le milieu universitaire. C'est une norme de l'OMG (Object Management Group) (OMG, 2023) qui offre un cadre graphique puissant pour décrire la structure et le comportement des systèmes logiciels. UML 2.0 propose deux classes de notations graphiques pour modéliser les différentes vues d'un système, à savoir les diagrammes de structure et les diagrammes de comportement.

Les diagrammes de structure montrent la structure statique des objets dans un système, décrivant les éléments des spécifications d'une manière indépendante du temps. D'un autre côté, les diagrammes de comportement présentent la dynamique des objets comme une série de changements apportés au système au fil du temps (UML, 2017). La spécification de l'OMG pour les diagrammes d'UML 2.0 consiste en une syntaxe concrète qui définit la notation graphique, et une syntaxe abstraite fournie avec un méta-modèle qui décrit la relation entre les éléments du diagramme.

Les cas d'utilisation spécifient les exigences fonctionnelles du système. Les classes décrivent les différents types d'objets du système pour répondre à ces exigences. Un diagramme de classes (UML 2.0 CD) fait parti de la vue logique d'UML, il décrit les classes et les relations statiques entre eux, il montre également les propriétés et les opérations des classes, ainsi que les contraintes de connexion des objets (Miles & Hamilton, 2006).

Les diagrammes d'interaction décrivent comment des groupes d'objets collaborent dans certains comportements. UML définit plusieurs formes de diagramme d'interaction, dont la plus courante est le diagramme de séquence (UML 2.0 SD). Généralement, un diagramme de séquence capture le comportement d'un seul scénario, et il permet de représenter les échanges entre les différents objets et acteurs du système en fonction du temps (Fowler, 2003). Le fragment combiné est une partie du diagramme de séquence qui définit une combinaison de fragments d'interaction. Ce mécanisme permet à l'utilisateur de décrire un certain nombre de traces de manière compacte et concise.

Les diagrammes d'activité et les diagrammes d'interaction sont utiles pour décrire le comportement d'un système, mais parfois, l'état d'un objet ou d'un système est un facteur important dans son comportement. Dans de telles situations, il est utile de modéliser les états d'un objet et les événements provoquant des changements d'état. La vue logique d'UML décrit les descriptions abstraites des parties d'un système, y compris quand et comment ces parties peuvent être dans différents états à l'aide de diagrammes d'états-transitions (UML 2.0 STM). Ces derniers présentent une forme spécifique des automates à états finis basée sur une variante orientée objet du formalisme *statechart* de David Harel. Les diagrammes d'états-transitions sont largement utilisés dans la modélisation des systèmes temps-réels, les systèmes critiques et les appareils dédiés

dont le comportement est défini en termes d'état, tel que les distributeurs automatiques de billets (ATMs) (Miles & Hamilton, 2006).

UML, à l'instar des langages de spécification semi-formels, jouit d'une large utilisation en raison de la richesse et de la clarté de ses modèles. Toutefois, il souffre d'un déficit de rigueur mathématique, une caractéristique qui distingue les langages formels. Cette lacune le rend moins approprié pour la vérification formelle des systèmes logiciels.

CSP est un langage de spécification formel faisant partie de la famille de l'algèbre des processus, adapté pour décrire les modèles d'interaction dans les systèmes concurrents où il y a plus d'un processus à la fois. Il englobe une collection de modèles mathématiques et de méthodes de raisonnement permettant de comprendre et d'utiliser ce langage (Roscoe A. W., 1997). Dans CSP, les processus sont des entités indépendantes qui interagissent les uns avec les autres par le biais de la communication. Chaque processus peut exécuter des événements décrivant son comportement. Les interactions entre les processus ou avec leur environnement sont présentées dans un style algébrique à l'aide d'un ensemble d'opérateurs. CSP est doté de nombreux outils de vérification automatique, notamment FDR4 (Failures-Divergences Refinement) (FDR, 2020), PAT (Process Analysis Toolkit) (Sun J. et al., 2009) et ProB (Leuschel & Butler, 2003).

Dans la littérature, plusieurs études tentent à formaliser UML 2.0 en CSP, comme celles référencées dans les sources (Dan & Danning, 2010), (Kaizu et al., 2013), (Kaizu et al., 2015), (Xu et al., 2008), (Bisztray et al., 2007). Cependant, les approches proposées présentent des restrictions concernant les composants formalisés, elles ne traitent que partiellement les diagrammes formalisés ou elles manquent d'automatisation. Dans le cadre de cette thèse, nous nous somme inspirés des trois travaux suivants : (Jacobs & Simpson, 2014), (Ng & Butler, 2003) et (Ng & Butler, 2002).

L'objectif de cette thèse est de proposer une approche intégrée UML 2.0/CSP pour la modélisation et la vérification des aspects statiques et dynamiques des systèmes distribués.

Nous nous concentrons particulièrement sur UML 2.0 CD, UML 2.0 SD et UML 2.0 STM. Nous entamons notre démarche en initiant un essai visant à améliorer les formalisations

sélectionnées d'UML 2.0 en CSP. Par la suite, nous avançons vers l'automatisation de cette approche, en proposant des outils visuels dédiés à la modélisation et à la transformation automatique des modèles UML 2.0 en spécifications CSP. Cette avancée permet aux vérificateurs de modèles CSP existants d'effectuer les vérifications requises. Notre approche repose sur la transformation de graphes et utilise l'outil de méta-modélisation multi-formalismes AToM³ (A Tool for Multi-formalism Modeling and Meta-modelling) (Juan de Lara, 2002). Elle propose trois méta-modèles des diagrammes UML 2.0 sélectionnés ainsi que trois grammaires de graphes. Afin de vérifier les propriétés comportementales du système modélisé, telles que les impasses, les blocages et le déterminisme, nous utilisons le vérificateur de modèle FDR4. Un aperçu général de l'approche proposée est présenté dans la figure 1 ci-dessous.

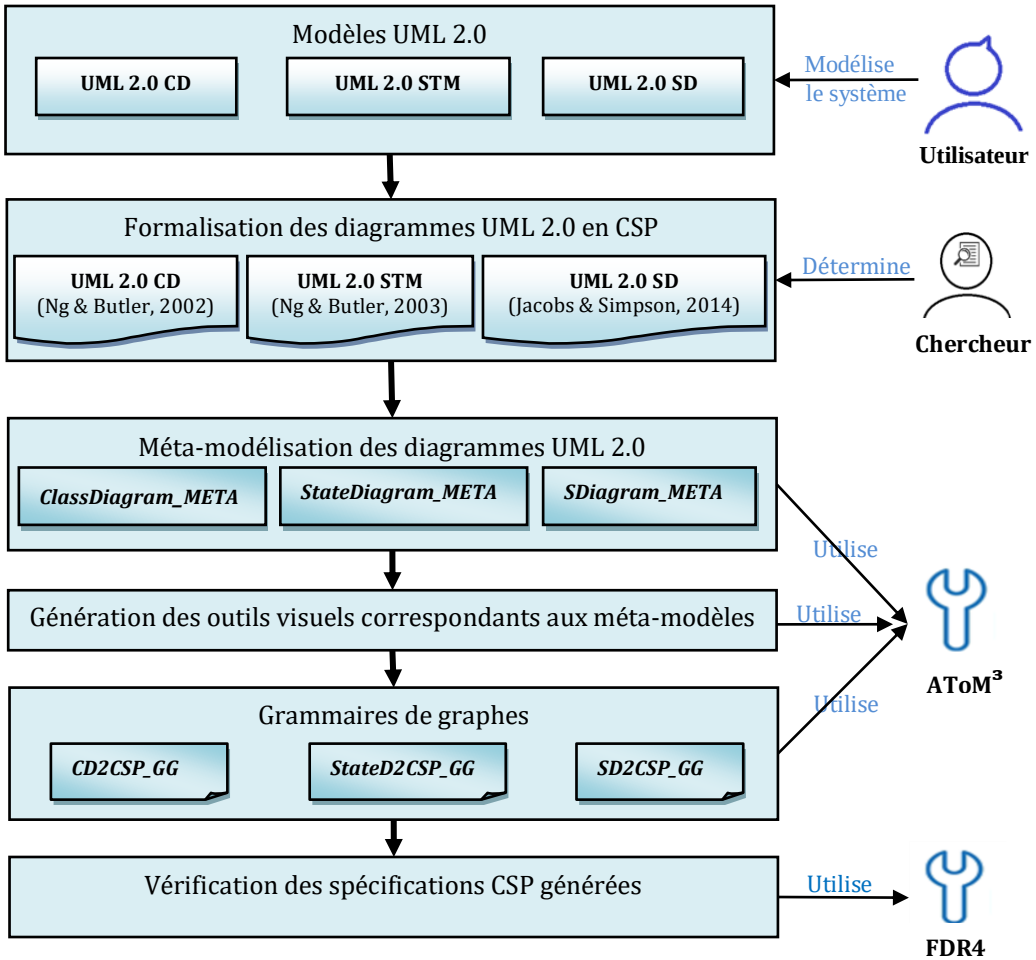


Figure 1. Aperçu de l'approche

CONTRIBUTIONS

Les apports principaux de cette thèse peuvent être synthétisés de la manière suivante :

• La formalisation de quatre pseudo-états d'UML 2.0 STM en CSP, à savoir les régions, la bifurcation (Fork), la jointure (Join) et la jonction (Junction).

• La proposition d'une approche automatique intégrée UML 2.0 / CSP pour la modélisation et la vérification des systèmes. Les étapes de réalisation de notre approche se décomposent comme suit :

1. Méta-modélisation des diagrammes UML 2.0

a- Définition de trois méta-modèles :

- SDiagram_META : le méta-modèle d'UML 2.0 SD.
- StateDiagram_META : le méta-modèle d'UML 2.0 STM.
- ClassDiagram_META : le méta-modèle d'UML 2.0 CD.

b- Génération des outils visuels correspondants aux méta-modèles proposés.

2. Transformation des diagrammes UML 2.0 en définissant trois grammaires de graphes pour transformer les diagrammes UML 2.0 en CSPM. Les grammaires développées sont les suivantes :

- SD2CSP_GG : pour transformer UML 2.0 SD.
- StateD2CSP_GG : pour transformer UML 2.0 STM.
- CD2CSP_GG : pour transformer UML 2.0 CD.

3. Vérification des spécifications CSPM générées

a- Définition des assertions CSPM.

b- Utilisation du model checker FDR4 pour vérifier le code CSPM généré en utilisant les assertions CSPM. Ensuite, génération du résultat de la vérification (passée si la propriété est satisfaite, échouée avec un contre-exemple sinon).

ORGANISATION DU DOCUMENT

Partie 1. Cadre théorique et état de l'art

Le premier chapitre présente les bases essentielles nécessaires à la compréhension de l'Ingénierie Dirigée par les Modèles et de l'approche MDA. Nous commencerons par une

exploration approfondie de l'IDM, décrivant sa philosophie et ses principes fondamentaux. Nous soulignerons l'importance de la séparation entre la conception et l'implémentation d'un système dans le processus de développement logiciel. Ensuite, nous examinerons l'approche MDA, en présentant ses notions de base, ainsi que son processus de transformation de modèles, depuis la création de modèles abstraits jusqu'à leur transformation en code exécutable. Nous présenterons également la transformation de modèles, les outils de transformations de graphes et en particulier l'outil AToM³ qui sera utilisé pour la réalisation de ce travail. Ce chapitre consacrera aussi une section au langage de modélisation UML 2.0 avec une attention particulière portée aux diagrammes de classe (UML 2.0 CD), aux diagrammes de séquence (UML 2.0 SD) et aux diagrammes d'états-transitions (UML 2.0 STM).

Le contenu du deuxième chapitre se concentre sur l'introduction du domaine de l'ingénierie système, des langages de spécification, et plus particulièrement des langages de spécification formels. Nous examinerons également les différentes méthodes de vérification, notament les tests, la simulation, le model-checking et le theorem-proving. Nous présenterons également le langage de spécification formel CSP en nous concentrant sur sa syntaxe, ses sémantiques, ses outils de vérification automatique, avec une attention particulière portée à FDR4.

Partie 2. Contributions

Le troisième chapitre se consacre à la présentation des méthodes formelles, des stratégies d'intégrations des langages formels et semi-formels, des travaux connexes dans ce domaine, ainsi que la formalisation des diagrammes UML 2.0 SD, UML 2.0 STM et UML 2.0 CD en CSP. Pour chaque diagramme, nous présenterons les correspondances entre les deux langages source et cible ainsi que le code CSP généré. Cette formalisation est inspirée des travaux précédents, et notre contribution consiste à l'amélioration de la formalisation d'UML 2.0 STM en CSP.

Le quatrième chapitre débute par une présentation globale de l'approche proposée, une approche intégrée UML 2.0/CSP pour la modélisation et la vérification des systèmes distibués. Ensuite, il détaille l'implémentation de cette approche automatique, basée sur l'utilisation combinée de la méta-modélisation et de la transformation de graphes. Le chapitre introduit les trois méta-modèles et les trois grammaires de graphes permettant la

modélisation des diagrammes de classe, de séquence et d'états-transitions , ainsi que leurs transformations en CSP pour effectuer les vérifications nécéssaires.

Dans le cinquième chapitre, nous exposons une étude de cas portant sur un distributeur automatique de billets (ATM) afin d'illustrer notre approche.

Enfin, la conclusion marque la fin de la thèse tout en ouvrant la voie vers de potentielles orientations de recherche futures.

Partie 1. Cadre théorique et état de l'art

Chapitre 1. Fondements de l'Ingénierie Dirigée par les Modèles et Modélisation de Systèmes

Chapitre 2. Le CSP et la Vérification Formelle

CHAPITRE

1

FONDEMENTS DE L'INGENIERIE DIRIGEE PAR LES MODELES ET MODELISATION DE SYSTEMES

1.1 INTRODUCTION

L'Ingénierie Dirigée par les Modèles (IDM) est un paradigme puissant qui modifie la manière dont les systèmes complexes sont conçus, développés et maintenus. Il met l'accent sur les modèles au sein du processus de développement logiciel. L'IDM offre des concepts, des outils et des langages pour créer et transformer des modèles, permettant ainsi de représenter les aspects essentiels d'un système. L'adoption de ce paradigme permet d'accélérer le cycle de développement, d'améliorer la qualité des produits et de faciliter l'adaptation aux évolutions technologiques.
La modélisation est le processus de spécification d'un modèle, impliquant la représentation simplifiée du système à travers des modèles qui font usage de concepts prédéfinis dans un langage de modélisation.

Au cours de ce chapitre, qui est une présentation générale du domaine de l'IDM et du langage de modélisation UML 2.0, nous essayons d'introduire les éléments nécessaires pour la compréhension des concepts de ce domaine. Nous allons tout d'abord, présenter les notions de base liées à l'IDM et à l'architecture dirigée par les modèles (MDA), notamment la notion de méta-modèle et celle de méta-méta-modèle. Puis, nous présentons la transformation de modèles, les outils de transformations de graphes et en particulier l'outil AToM³ qui est utilisé pour la réalisation de ce travail.
Ensuite, nous allons introduire UML 2.0 avec une attention particulière portée aux diagrammes de classe (UML 2.0 CD), aux diagrammes de séquence (UML 2.0 SD) et aux diagrammes d'états-transitions (UML 2.0 STM).

1.2 OBJECT MANAGEMENT GROUP

L'OMG est une association commerciale internationale constituée en tant que société à but non lucratif aux États-Unis en 1998, affiliée à des organisations du monde entier. Elle reçoit un financement sur la base de cotisations annuelles de ses membres diversifiés, composés de centaines entreprises, d'universités et d'organismes de normalisation. L'OMG se concentre sur les normes d'intégration et organise la plus grande exposition et conférence axées sur l'évolution de la technologie d'intégration.

L'OMG a été formé pour aider à réduire la complexité des systèmes, réduire les coûts et accélérer l'introduction de nouvelles applications logicielles. Ces objectifs sont atteints grâce à l'introduction de l'architecture dirigée par les modèles. Cette dernière est un cadre architectural avec des spécifications détaillées qui conduisent vers des composants logiciels interopérables, réutilisables et portables, et vers des modèles de données basés sur des modèles standards. MDA est une approche d'utilisation de modèles dans le développement de logiciels (Belaunde et al., 2003). Avant de détailler les objectifs, les modèles et le processus de transformation de modèles de l'approche MDA, examinons quelques concepts fondamentaux liés à l'IDM.

1.3 CONCEPTS DE BASE

Un système est un ensemble d'entités en interaction pour accomplir un objectif. Il peut comprendre un programme, un système informatique unique, une combinaison de parties de différents systèmes, des personnes, des entreprises, etc. Un système est représenté par un modèle, l'élément le plus important dans l'IDM.

1.3.1 Modèle

Un modèle est une abstraction d'un système. C'est une description simplifié du système étudié mettant l'accent sur un certain nombre d'aspects du système et ignorant d'autres. Il n'existe pas de définition cummune du concept de modèle. La déclaration suivante expose celle de Bézivin & Gerbé (Bézivin & Gerbé, 2001).

> **Modèle :** « *Un modèle est une simplification d'un système construit avec un objectif visé dans esprit. Le modèle doit être capable de répondre aux questions à la place du système actuel. Les réponses apportées par le modèle doivent être les même que ceux données par le système lui-même, à condition que les questions relèvent du domaine défini par l'objectif général du système* » (Bézivin & Gerbé, 2001)

1.3.2 Méta-modèle

Plusieurs définitions du méta-modèle existent. Il est parfois défini en tant que modèle de modèle ou en tant que modèle décrivant un langage de modélisation. Cependant, selon notre perspective, la définition la plus précise provient de Da Silva, comme exposé dans (Da Silva, 2015).

Méta-modèle : « *Un méta-modèle est un modèle qui définit la structure d'un langage de modélisation* » (Da Silva, 2015)

En se référant à la Figure 2, nous pouvons approfondir notre compréhension du concept méta-modèle en détaillant les relations qui le lient au modèle et au langage de modélisation.

- la relation *ElementDe* (entre Modèle et langage de modélisation) : un langage de modélisation est un ensemble de modèles ou un modèle est un élément d'un langage de modélisation.
- la relation *Définit* (entre méta-modèle et langage de modélisation) : un méta-modèle est un modèle d'une structure de langage de modélisation ou un langage de modélisation est défini par un méta-modèle.
- La relation d'héritage (entre méta-modèle et modèle) : un méta-modèle est un modèle d'un ensemble de modèles ou est un modèle de modèles.
- la relation *ConformeAvec* (entre méta-modèle et modèle) un modèle est conforme à un méta-modèle, ce qui signifie que le modèle devrait satisfaire aux règles définies au niveau de son méta-modèle.

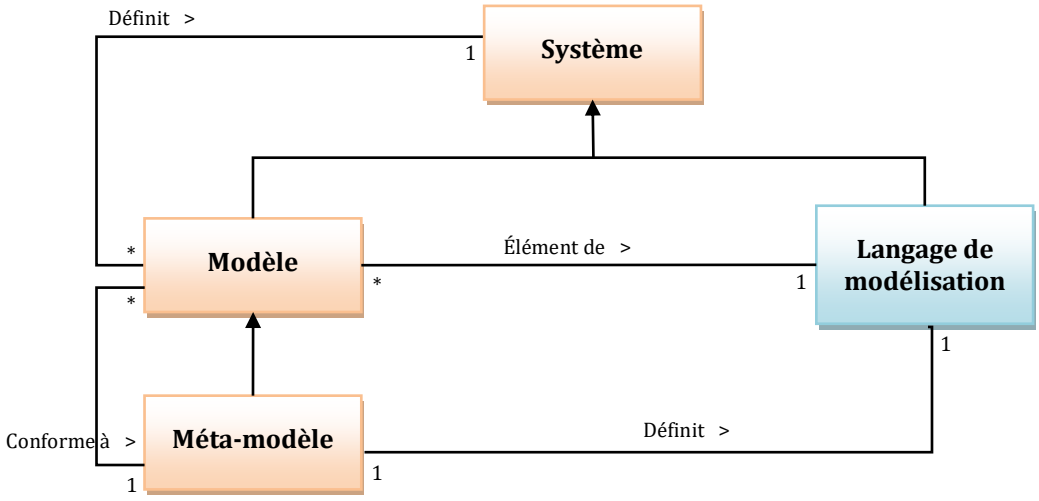


Figure 2. Définition du méta-modèle : relations entre méta-modèle et modèle (Da Silva, 2015)

1.3.3 Méta-méta-modèle

Le rôle joué par le concept de méta-modèle est important, mais n'est pas suffisant. Le développement d'une grande variété de méta-modèles différents et incompatibles (data warehouse, workflow, software process, etc.) rendait impératif la définition d'un cadre global d'intégration des méta-modèles dans le domaine de l'ingénierie logicielle, de l'ingénierie systèmes et de l'ingénierie de données. La solution consistait à introduire un langage de définition des méta-modèles, où chaque méta-modèle définissant un langage pour décrire un domaine d'intérêt spécifique (Bézivin & Briot, 2004). La Figure 3 présente les quatre niveaux de modélisation de l'OMG.

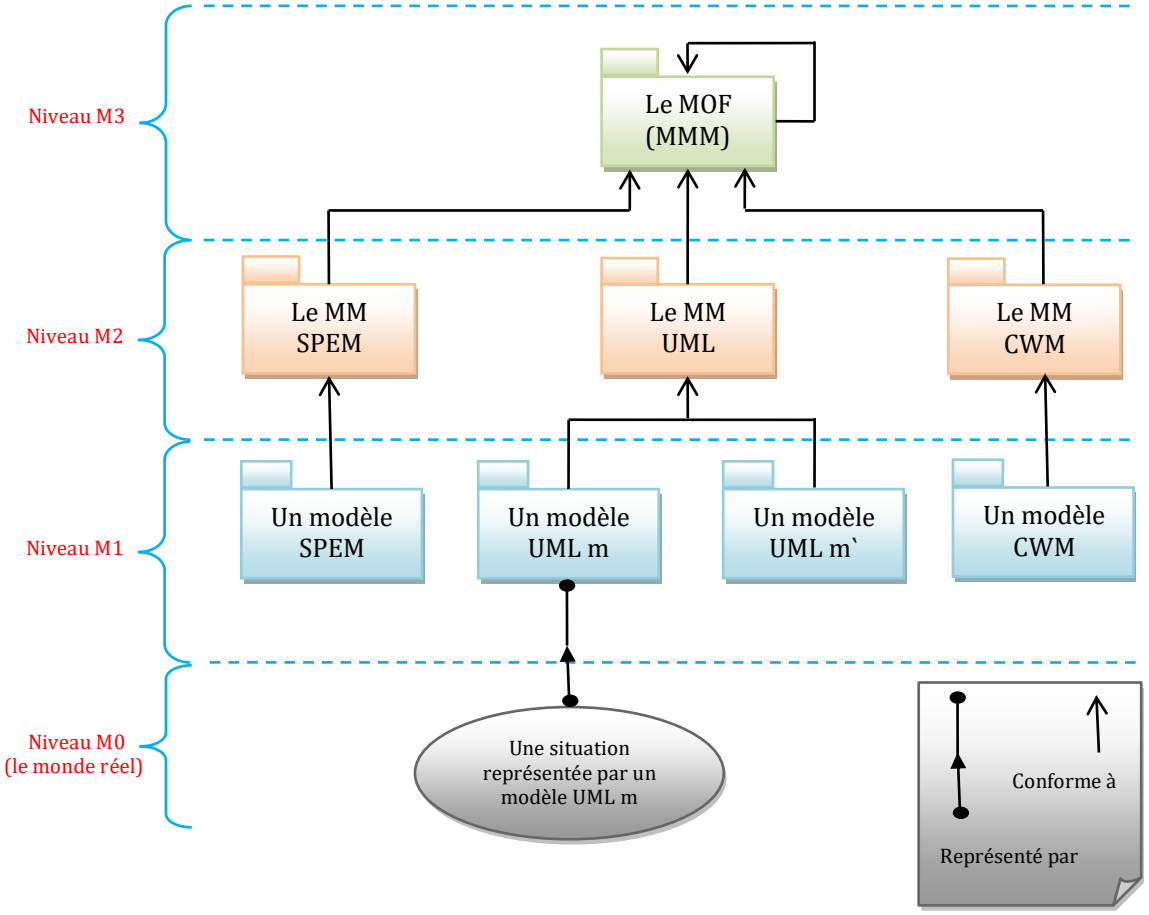


Figure 3. La modélisation multi-niveau de l'OMG (Bézivin & Briot, 2004)

1.4 MODEL DRIVEN ARCHITECTURE

L'architecture dirigée par les modèles repose sur l'idée de séparer la spécification du fonctionnement d'un système des détails de son implémentation. Dans l'approche MDA, le processus de développement logiciel commence par définir le système de manière abstraite et indépendante de toute plateforme d'implémentation. Ensuite, il consiste à décrire les différentes plateformes possibles pour mettre en œuvre ce système logiciel. Enfin, une plateforme spécifique est choisie, et la spécification du système est transformée en une spécification adaptée à cette plateforme sélectionnée.

Les trois principaux objectifs de MDA sont la portabilité, l'interopérabilité et la réutilisabilité, assurées par la séparation architecturale des préoccupations. MDA vise à permettre la création de modèles d'application et de données lisibles par machine, ce

qui permet d'apporter une flexibilité à long terme à différentes phases du cycle de développement logiciel. Lors de la phase d'implémentation, il est possible d'intégrer une nouvelle infrastructure d'implémentation ou de la cibler par des conceptions existantes. Si une conception est disponible au moment de l'intégration, nous pouvons automatiser la création de ponts d'intégration de données et réaliser la connexion à de nouvelles infrastructures d'intégration. La disponibilité de la conception sous une forme lisible par machine offre aux développeurs un accès direct aux spécifications du système, simplifiant ainsi considérablement la maintenance. De plus, étant donné que les modèles développés peuvent être utilisés pour générer du code, ils peuvent être validés par rapport aux exigences, testés sur différentes infrastructures et utilisés pour simuler directement le comportement du système en cours de conception (Belaunde et al., 2003).

La dénomination de l'approche MDA et de ses modèles peut être clarifiés en explorant les concepts suivants :

- **Architecture :** L'architecture d'un système est une spécification des parties et des connecteurs du système et les règles d'interaction entre eux. L'architecture dirigée par les modèles décrit certains types de modèles à utiliser, les relations entre eux et les manières pour les élaborés.
- **Dirigé par les modèles :** MDA renforce le rôle des modèles dans le processus de développement de systèmes. Elle est dirigée par les modèles car elle fournit un moyen d'utiliser des modèles pour guider la compréhension, la conception, la construction, le déploiement, l'exploitation, la maintenance et les modifications.
- **Plateforme :** Une plateforme est un ensemble de sous-systèmes et de technologies qui fournissent un ensemble cohérent de fonctionnalités via des interfaces et des modèles d'utilisation spécifiés. Toute application prise en charge par cette plateforme peut utiliser les fonctionnalités fournies par la plateforme sans se soucier des détails de ses implémentations. L'indépendance de la plateforme est une qualité qu'un modèle peut présenter. C'est la qualité que le modèle est indépendant des caractéristiques d'un type particulier d'une plateforme.

- **Point de vue :** Un point de vue sur un système est une technique d'abstraction qui fait appel à un ensemble sélectionné de concepts architecturaux et de règles de structuration, dans le but de mettre l'accent sur des aspects particuliers du système en question. Dans ce contexte, le terme "abstraction" est utilisé pour désigner le processus de suppression des détails sélectionnés pour établir un modèle simplifié. MDA spécifie trois points de vue sur un système, un point de vue indépendant du calcul, un point de vue indépendant de la plateforme et un point de vue spécifique à la plateforme.

1.4.1 Classes de modèles

MDA propose trois classes de modèles pour décrire le système modélisé, Chaque classe correspond à un niveau d'abstraction différent, et un modèle pour décrire la plateforme d'exécution du logiciel.

1.4.1.1 Computation Independent Model (CIM)

Le modèle d'exigences (CIM) modélise l'environnement et les exigences du système, sans inclure de détails sur sa structure ou son comportement. Un CIM est un modèle de système qui montre le système dans l'environnement dans lequel il fonctionnera. Ainsi, il aide à présenter exactement ce que le système est censé faire sans description de la manière de son implémentation (Belaunde et al., 2003). Dans une spécification MDA d'un système, les CIM doivent être traçables aux modèles PIM et PSM qui les implémentent. De plus , on peut également les considérer comme des éléments contractuels, servant de référence pour vérifier la conformité d'une application aux demandes du client. (Blanc & Salvatori, 2011)

1.4.1.2 Platform Independent Model (PIM)

Le modèle d'analyse et de conception abstraite (PIM) met l'accent sur le fonctionnement du système modélisé tout en dissimulant les particularités d'une plateforme spécifique. Il représente la partie de la spécification qui reste constante, indépendamment de la plateforme choisie (Belaunde et al., 2003). Dans l'élaboration des PIM, nous ne prenons en compte que la conception abstraite qui peut être réalisée sans avoir besoin de connaître les techniques d'implémentation. MDA préconise l'utilisation d'UML pour

réaliser les modèles PIM, mais il n'exclut pas l'utilisation d'autres langages de modélisation. (Blanc & Salvatori, 2011)

1.4.1.3 Platform Model (PM)

Un modèle de plateforme (PM) décrit la plateforme d'exécution d'un logiciel. Il inclut une série de concepts techniques qui représentent les composants d'une plateforme ainsi que les services offerts par cette dernière. De plus, il comporte des concepts qui décrivent comment une application utilise la plateforme.

1.4.1.4 Platform Specific Model (PSM)

Le modèle de code (PSM) combine les spécifications du PIM avec les détails de la plateforme cible (PM) grâce à la transformation de modèle. Il spécifie comment les fonctionnalités d'un PIM sont concrétisées sur une plateforme spécifique (Belaunde et al., 2003). Après la création du PSM, l'étape suivante du processus de développement logiciel implique la génération du code à partir du PSM et le déploiement du système dans un environnement spécifique.

1.4.2 Transformation de modèles dans l'approche MDA

Les transformations de modèles sont appliquées de manière séquentielle sur les modèles jusqu'à la génération du code. Dans le contexte de MDA, le code d'une application est une suite de lignes de texte, tandis qu'un modèle de code est une représentation structurée qui englobe des éléments tels que des boucles, des événements, des composants, des instructions, des conditions, etc. Par conséquent, la génération de code à partir d'un modèle de code est une opération relativement simple (Blanc & Salvatori, 2011). La Figure 4 offre une vision globale du processus de transformation de modèles dans l'approche MDA.

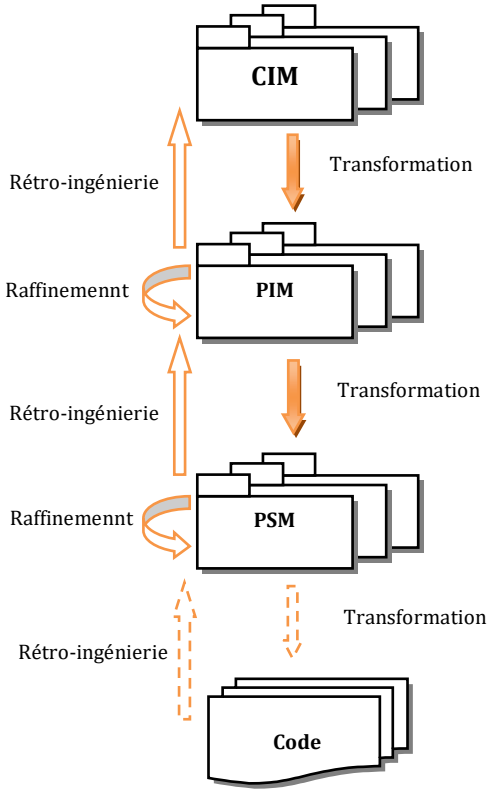


Figure 4. Processus de transformation de modèles de l'approche MDA

1.5 CLASSIFICATION DES TRANSFORMATIONS DE MODELES

Selon (Bézivin & Briot, 2004), la transformation de modèle s'effectue en générant un ou plusieurs modèles cibles à partir d'un ou de plusieurs modèles sources équivalents. Chaque modèle doit être conforme à un méta-modèle, qui identifie les caractéristiques de ce modèle. Ensuite, le mappage est défini comme une traduction entre les méta-modèles source et cible.

Si les modèles source et cible sont conformes au même méta-modèle, on parle de transformation endogène. Dans le cas contraire, on parle d'une transformation exogène.

Une transformation peut entraîner un changement de niveau d'abstraction. Elle est considérée comme verticale lorsqu'elle engage différents niveaux d'abstraction dans la transformation. À l'inverse, on parle de transformation horizontale lorsque les modèles source et cible impliqués sont au même niveau d'abstraction.

Le degré d'automatisation d'une transformation est lié aux informations fournies à cette transformation, appelées paramètres de la transformation. Si tous les paramètres sont

fournis à la transformation avant son exécution, on parle de transformation automatique. En revanche, si certains paramètres ne sont renseignés qu'au moment de l'exécution de la transformation, on parle de transformation semi-automatique (Mens & Van Gorp, 2006).

Cezarneki et ses collègues ont proposé, dans (Czarnecki et al., 2003), une classification des approches de transformation de modèles. Au niveau supérieur, ils ont distingué deux classes d'approches de transformation : les approches de transformation modèle vers code et modèle vers modèle. Selon les auteurs, la première classe peut être considérée comme un cas particulier de la seconde, car il suffit de fournir un méta-modèle pour le langage de programmation cible. Toutefois, pour des raisons pratiques de réutilisation de la technologie de compilateur existante, le code est souvent généré simplement sous forme de texte, qui est ensuite introduit dans un compilateur. Pour cette raison, ils ont distingué entre les deux classes de transformation.
Dans la catégorie modèle vers code, les auteurs ont distingué les approches basées sur les visiteurs et celles basées sur des modèles. Quant à la catégorie modèle vers modèle, ils ont identifié les approches de manipulation directe, les approches relationnelles, les approches basées sur la structure, les approches hybrides et les approches basées sur la transformation de graphes. La Figure 5 résume les approches de transformation de modèle proposées dans (Czarnecki et al., 2003).

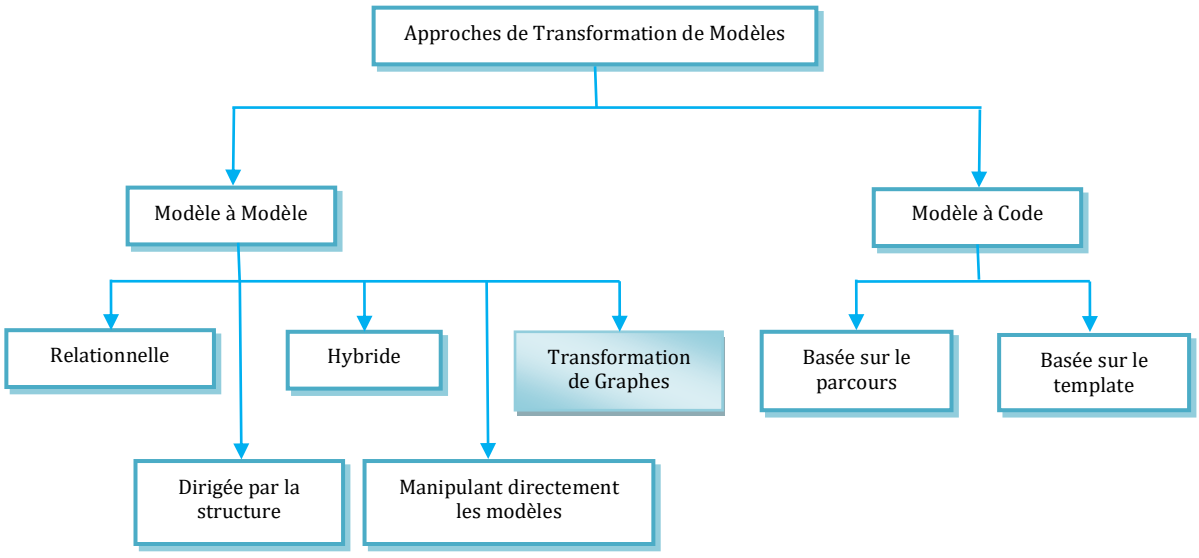


Figure 5. Classification des approches de transformation de modèles (Czarnecki et al., 2003)

Ci-dessous, nous exposons le principe des transformations de graphes, sur lequel se base l'approche que nous proposons dans notre thèse.

1.5.1 Transformation de Graphes

Cette catégorie d'approches de transformation de modèles fonctionne sur des graphes typés, attribués et étiquetés (Andries et al., 1999). Les grammaires de graphes (Ehrig et al., 1999) sont une généralisation des grammaires de Chomsky pour les graphes. Ces grammaires sont composées de règles ou productions avec des membres gauche et droit : *P = (LHS, RHS)*. L'application de chaque règle permet la transformation d'un graphe initial, appelé *host graph*, en remplaçant une de ses parties par un autre graphe. Le principal défi de ce remplacement est de savoir comment relier RHS au contexte du graphe cible. La Figure 6 illustre la modification des graphes basée sur des règles associée à l'architecture de transformation de modèle.

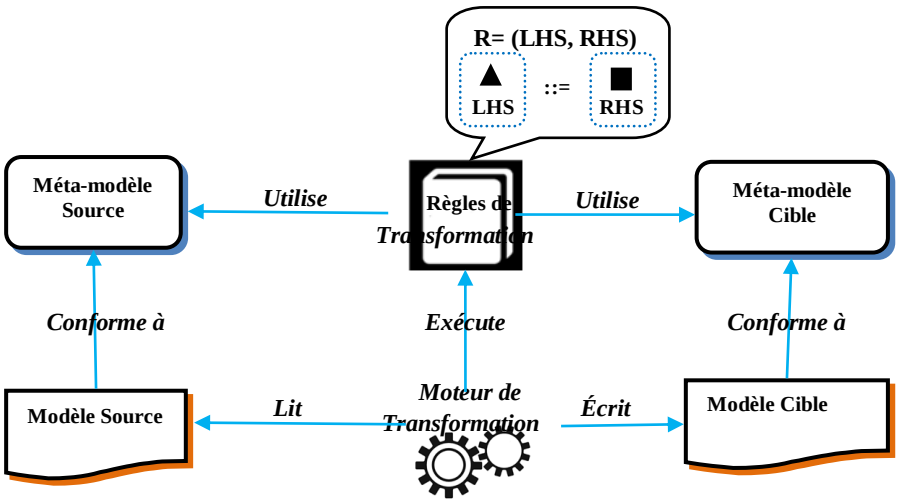


Figure 6. Architecture de transformation de modèle

1.5.2 Outils de transformation de graphes

Actuellement, divers outils de transformation de graphes sont disponibles, chacun a des avantages particuliers ainsi que des limites spécifiques.

Selon (Kahani et al., 2019), un grand nombre de langages et d'outils de transformation de modèles ont été proposés ces dernières années. Ces outils peuvent être utilisés pour développer, transformer, fusionner, échanger, comparer et vérifier des modèles et méta-modèles. Dans (Kahani et al., 2019) les auteurs ont présenté un catalogue complet d'outils de transformation existants. Ils ont organisé les 60 outils identifiés en une

classification générale basée sur l'approche de transformation utilisée. Ensuite ils ont comparé ces outils en utilisant un certain nombre de facettes particulières.

Le Tableau 1 présente des outils de transformation de graphes, La deuxième colonne du tableau fournit une brève description informelle de l'outil, et la troisième colonne indique le langage de programmation dans lequel il est implémenté. Les colonnes intitulées PV et DV indiquent respectivement les dates de la première et la dernière version des outils jusqu'à la date de réalisation de l'étude (2017).

Tableau 1. Outils de transformation de graphes (Kahani et al., 2019)

Outil	Description	Lang	PV	DV
GROOVE (Rensink, 2004)	prend en charge la vérification de modèle des systèmes de transformation de graphes	Java	2003	2014
UMLX (Willink, 2003)	prend en charge une syntaxe graphique concrète pour compléter le langage QVT	Java	2005	2017
AToM³ (Juan de Lara, 2002)	un outil de modélisation multi-paradigmes pour les langages visuels	Python	2004	2008
AToMPM (Syriani et al., 2013)	successeur d'AToM³ qui génère des outils DSM basés sur le Web	Python	2012	2016
AGG (Ermel et al., 1999)	une approche algébrique attribuée à une transformation de grammaire de graphes	Java	1997	2017
BOTL (Braun & Marschall, 2003)	un langage de transformation bidirectionnel avec une base formelle précise	Java	2003	2008
GRoundTram (Hidaka et al., 2011)	Cadre aller-retour basé sur les graphes pour les transformations de modèles bidirectionnels	OCaml	2009	2014
eMotIon (Lauder et al., 2012)	prend en charge la modélisation basée sur les scénarios et les TGGs	Java	2006	2017
MoTE (Giese et al., 2014)	offre la bidirectionnalité, la synchronisation et la cohérence des modèles	Java	2010	2016
GReAT	basé sur la spécification de modèles, la transformation	VC++	2004	2014

(GReAT)	de graphes et les langages de flux de contrôle			
TGGInterpreter (Greenyer & Kindler, 2007)	utilise les règles TGG pour spécifier les transformations	Java	2006	2011
MOMoT (Fleck et al., 2015)	un framework qui combine la modélisation avec des techniques basées sur la recherche	Java	2014	2016
EMorF (Klassen & Wagner, 2012)	basé sur TGG incrémentiel pour prendre en charge la synchronisation du modèle	Java	2012	2012
DSLTrans (Barroca et al., 2011)	un langage et un outil visuel pour les transformations de modèles	Java	2011	2014
MoTMoT (Van Gorp, 2008)	Graphe de réecriture basé sur des diagrammes d'UML(de scénario)	Java	2004	2006

Dans l'étude menée par (Aouat et al., 2012), les auteurs ont présenté une étude comparative générale de quatre outils de transformation de graphes, à savoir AGG, GROOVE, GrGen et AToM³. Par ailleurs, dans (Varro et al., 2005) les auteurs ont proposé une méthode systématique de benchmarking quantitative afin d'évaluer les performances des outils de transformation de graphes. De plus, ils ont mesuré et comparé les performances de plusieurs outils de transformation de graphes sur un problème d'exclusion mutuelle distribuée. Cette étude comprenait les outils suivants : AGG, PROGRES, FUJABA , et l'approche database(DB) (Andries et al., 1999).

Pour la réalisation de notre approche , notre choix s'est porté sur l'outil AToM^3. Ce choix s'est imposé pour plusieurs raisons. Tout d'abord, la simplicité d'utilisation d'AToM^3 facilit le processus de modélisation et de transformation de modèles, ce qui contribue à accélérer le développement et à améliorer l'efficacité du travail. De plus, la prise en charge multi-formalismes, permettant le chargement simultané de plusieurs modèles créés dans des formalismes différents. Cette caractéristique facilite grandement la modélisation des systèmes complexes en modélisant les différentes parties du système en utilisant différents formalismes dans un environnement unifié. Enfin, sa disponibilité en open source et sa capacité à gérer les deux types de transformations de modèles : les

transformations de type modèle vers code et les transformations de type modèle vers modèles.

1.5.3 AToM³

AToM^3 (A Tool for Multi-formalism and Meta-Modelling) est un outil visuel qui combine la méta-modélisation et la modélisation multi-formalismes. Il est le fruit du travail de recherche et développement réalisé au sein du laboratoire MSDL (Modeling, Simulation and Design Lab), qui est une unité de recherche affiliée à l'école d'informatique de l'université de McGill, située au Canada. AToM^3 est un outil libre implémenté en python (Lutz, 2006) et compatible avec différentes plateformes telles que Windows, Linux, etc.
Les deux tâches principales d'AToM3 sont la méta-modélisation et la transformation de modèles. En ce qui concerne la méta-modélisation, AToM^3 offre deux méta-formalismes pour la création de méta-modèles : le formalisme Entité-Association et le diagramme de classe UML. les méta-modèles peuvent être étendus par l'expression de contraintes en utilisant le langage OCL (Object Constraint Language) ou le code Python. Une fois que le méta-modèle est créé en utilisant le méta-formalisme choisi, AToM^3 peut automatiquement générer un outil de modélisation visuel. Avec cet outil, l'utilisateur peut créer et éditer de nouveaux modèles. La transformation de modèles se base sur la réécriture de graphes. Les règles de grammaire de graphes sont utilisées pour définir les transformations entre les formalismes et la génération de code, ainsi que la spécification de simulateurs. L'utilisateur doit définir les règles (LHS, RHS), les priorités et les conditions.
AToM³ facilite la modélisation et l'analyse des systèmes complexes. Ces derniers sont caractérisés par la multiplicité et la diversité de ses composants. L'analyse d'un tel système nécessite l'évaluation de ses propriétés dans l'ensemble du système. AToM³ permet de modéliser les différentes parties du système en utilisant les différents formalismes, et les modèles peuvent être automatiquement convertis entre ces formalismes pour faire l'analyse. Cette fonctionnalité permet de modéliser et d'analyser les systèmes complexes de manière efficace et précise, en utilisant les formalismes les plus appropriés pour chaque partie du système.

AToM^3 dispose d'une interface graphique conviviale pour faciliter la création, la modification et la transformation de modèles. La Figure 7 présente l'interface graphique de l'outil AToM^3.

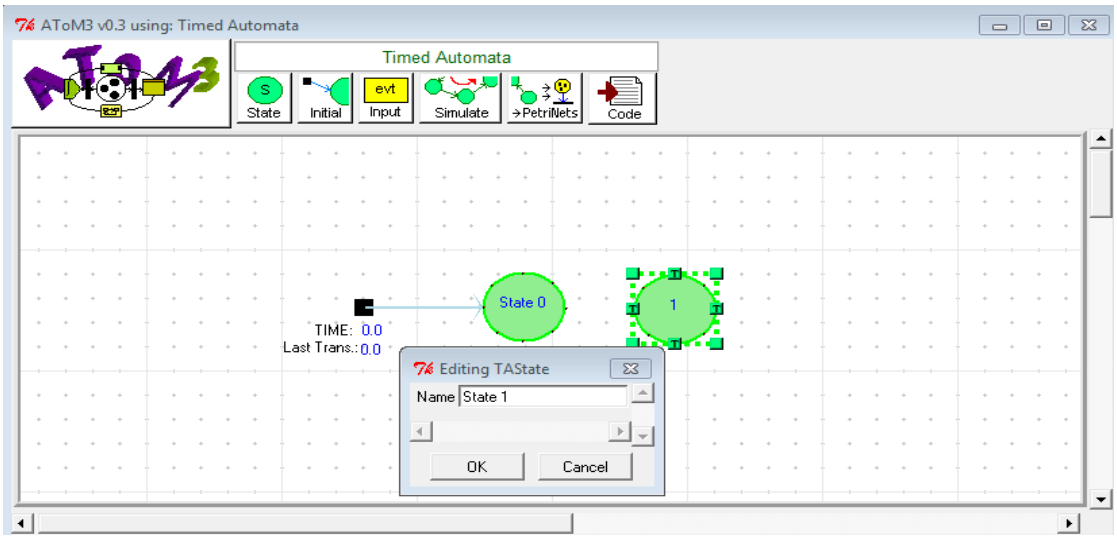


Figure 7. Interface graphique de l'outil AToM^3

1.6 UNIFIED MODELING LANGUAGE

UML est une norme de l'OMG. C'est un langage graphique de modélisation qui unifie les méthodes d'analyse et de conception orientée objet de BOOCH (Grady Booch), OMT (James Rumbaugh) et OOSE (Ivar Jacobson).

Une méthode se compose principalement d'un langage de modélisation et d'un processus. Le langage de modélisation est la notation utilisée par les méthodes pour exprimer les conceptions (élaboration des modèles), tandis que le processus décrit les étapes à suivre pour effectuer une conception. Ainsi, UML est un langage et non une méthode (Fowler, 2003).

1.6.1 Les diagrammes UML

Un diagramme particulier d'UML montre certaines parties du modèle mais pas nécessairement tout. la première chose à comprendre est que votre modèle se trouve derrière votre outil de modélisation et vos diagrammes en tant que collection d'éléments. Chacun de ces éléments peut être un cas d'utilisation, une classe, une activité ou toute autre construction prise en charge par UML. L'ensemble de tous les éléments

qui décrivent votre système, y compris leurs connexions les unes aux autres, constituent votre modèle. les diagrammes sont utilisés simplement comme un canevas sur lequel vous pouvez créer de nouveaux éléments qui sont ensuite ajoutés à votre modèle et organiser les éléments associés, dans un ensemble de vues sur votre modèle. Donc, un diagramme est une vue du contenu du modèle, mais n'est pas le modèle. C'est simplement un moyen utile de présenter une petite partie des informations contenues dans le modèle (Miles & Hamilton, 2006).

Comme le montre La Figure 8. Taxonomie des diagrammes de structure et de comportement d'UML , UML propose deux groupes de diagrammes : les diagrammes de structure et les diagrammes de comportement. Les diagrammes de structure montrent la structure statique des objets dans un système, décrivant les éléments des spécifications d'une manière indépendante du temps. D'autre part, les diagrammes de comportement illustrent le comportement dynamique des objets qui peut être décrit comme une série de changements apportés au système au fil du temps (UML, 2017).

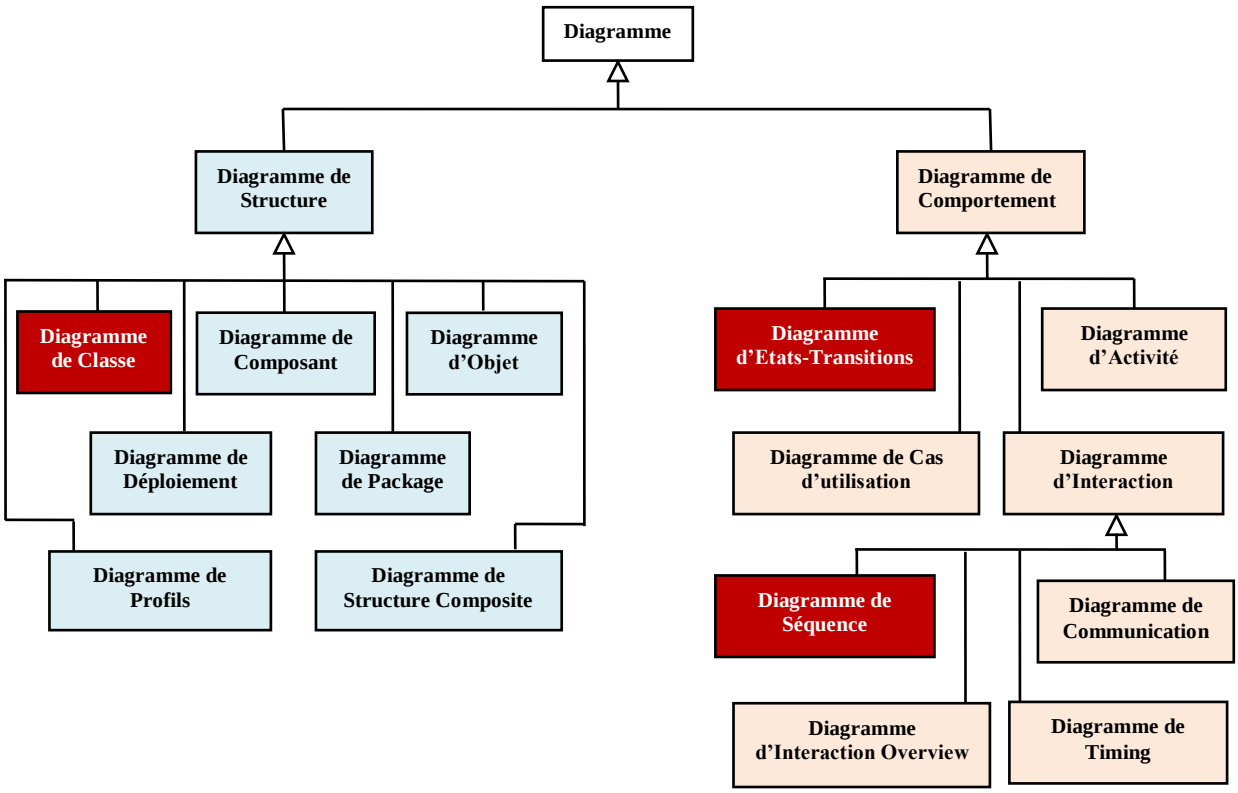


Figure 8. Taxonomie des diagrammes de structure et de comportement d'UML (UML, 2017)

Les éléments modélisés dans chaque diagramme, ainsi que la version UML d'introduction de ce dernier, sont résumés dans le Tableau 2.

Tableau 2. Les diagrammes UML2.0 (Miles & Hamilton, 2006)

Type de diagramme	Nom de diagramme	Eléments modélisés	Diagramme introduit par
Diagramme de structure	Diagramme de Classe	Il modélise les classes, les types, les interfaces et les relations entre eux.	UML 1.x
	Diagramme d'Objet	Il modélise les instances d'objet des classes définies dans les diagrammes de classes.	UML 1.x
	Diagramme de Composant	Ce diagramme met l'accent sur les composants importants dans le système modélisé et les interfaces qu'ils utilisent pour interagir ensemble.	UML 1.x, mais prend un nouveau sens en UML 2.0
	Diagramme de Déploiement	Il décrit la façon dont le système est déployé dans une situation réelle donnée.	UML 1.x
	Diagramme de Package	Ce diagramme décrit l'organisation hiérarchique des groupes de classes et de composants.	UML 2.0
	Diagramme de Structure Composite	Il décrit les éléments internes d'une classe ou d'un composant, et peut également décrire les relations d'une classe dans un contexte donné.	UML 2.0
	Diagramme de Profils	C'est un mécanisme d'extensibilité permettant d'étendre et de personnaliser UML. Il comporte trois types de mécanismes d'extensibilité : stéréotypes, valeurs étiquetées et contraintes.	UML 2.2
Diagramme de comportement	Diagramme d'Etats-Transitions	Ce diagramme décrit l'état d'un objet tout au long de sa durée de vie et les événements qui peuvent modifier cet état.	UML 1.x
	Diagramme d'Activité	Ce diagramme décrit les activités séquentielles et parallèles au sein du système modélisé.	UML 1.x
	Diagramme de Cas d'utilisation	Il décrit les interactions entre le système modélisé et les utilisateurs ou d'autres systèmes externes.	UML 1.x
	Diagram Diagramme de Séquence	Il décrit les interactions entre les objets où l'ordre des interactions est important.	UML 1.x

		Diagramme de Communication	Ce diagramme décrit les façons d'interaction des objets et les connexions nécessaires pour soutenir cette interaction.	Renommé à partir des diagrammes de collaboration d'UML 1.x
		Diagramme d'Interaction Overview	C'est une variante de diagramme d'activité, il est utilisé pour collecter les diagrammes de séquence, de communication et de timing pour capturer une interaction importante qui se produit au sein du système modélisé.	UML 2.0
		Diagramme de Timing	Il décrit les interactions entre objets et il se concentre sur les contraintes temporelles.	UML 2.0

1.6.2 UML 2.0 SD

Les diagrammes d'interaction décrivent comment des groupes d'objets collaborent dans certains comportements. UML 2.0 définit plusieurs formes de diagramme d'interaction, la plus courante étant le diagramme de séquence (Fowler, 2003). En général, un diagramme de séquence capture le comportement d'un seul scénario et permet de représenter les échanges entre les différents objets et acteurs du système en fonction du temps. Si le système à modéliser est complexe, il n'est pas possible de modéliser la dynamique globale du système par un diagramme unique. On fera donc appel à un ensemble de diagrammes de séquence, chacun correspondant à une sous-fonction du système.

Un événement est la plus petite partie d'une interaction, c'est tout point où quelque chose se produit, dont les plus courants sont l'envoi ou la réception d'un message ou d'un signal. (Miles & Hamilton, 2006)

La sémantique d'un Message complet est simplement la trace *<Evénement d'envoi, Evénement de réception>*. Un message perdu est un message sans occurrence d'événement de réception (sa trace est : *< Evénement d'envoi >*), et un message trouvé est un message sans occurrence d'événement d'envoi (sa trace est : *< Evénement de réception >*) (UML, 2017). Si le message est synchrone, l'émetteur est bloqué pendant la réalisation du traitement par le récepteur, contrairement au message asynchrone, où l'émetteur du message continue son exécution pendant le traitement du message par le

récepteur, et il sera occupé à invoquer d'autres messages avant le retour du message d'origine (Miles & Hamilton, 2006).

Une ligne de vie décrit la chronologie d'un processus, où le temps augmente vers le bas de la page. L'ordre des spécifications d'occurrence le long d'une ligne de vie indique l'ordre dans lequel ils se sont produits, et la distance entre deux événements sur la même ligne de temps ne représente aucune mesure du temps. (UML, 2017)

Le fragment combiné est une partie du diagramme de séquence qui définit une combinaison de fragments d'interaction. Ce mécanisme permet à l'utilisateur de décrire un certain nombre de traces de manière compacte et concise. Un fragment combiné contient des opérandes d'interaction et un opérateur qui défini l'ordre d'exécution des messages. La Figure 9 présente un exemple d'un diagramme de séquence.

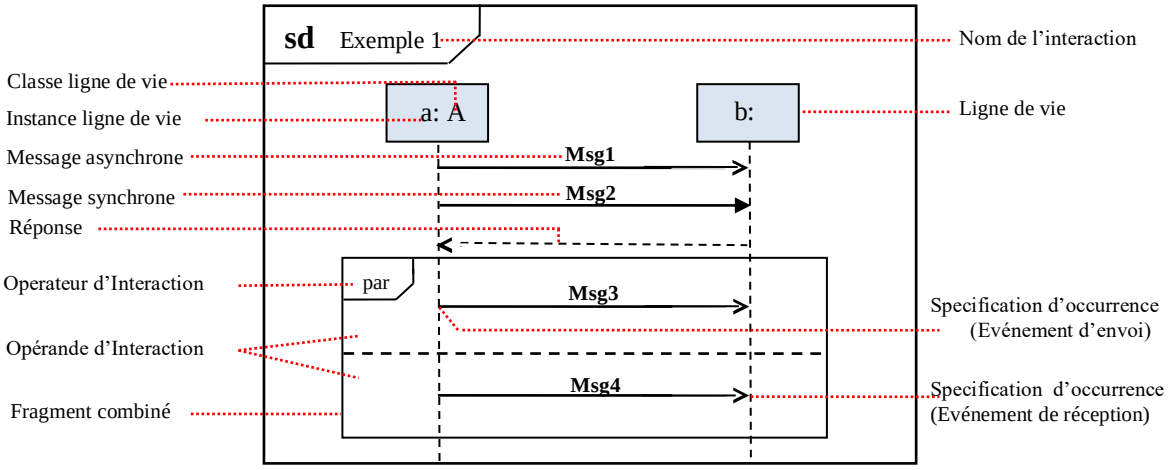


Figure 9. Exemple d'un diagramme de séquence

Le Tableau 3 montre la description des différents types d'opérateurs d'interaction (OI) du fragment combiné.

Tableau 3. Types d'opérateurs d'interaction du fragment combiné

Type d'OI	***Signification***	***Description***
alt	Alternative	Le fragment combiné a de nombreux opérandes qui représentent le choix du comportement. L'opérande choisi est celui dont l'expression de garde est évaluée à vrai (un opérande au maximum sera choisi).

opt	Option	Le fragment combiné n'a qu'un seul opérande qui représente le choix du comportement. Si l'expression de garde est évaluée à vrai, l'opérande se produit, sinon rien ne se passe.
break	Interruption	Le fragment combiné a un seul opérande qui représente le scénario d'interruption. Si l'expression de garde est vraie, l'opérande est choisi et le reste de l'interaction est ignoré. Sinon, l'opérande est ignoré et le reste de l'interaction est choisi.
par	Parallèle	Les comportements des opérandes dans ce fragment combiné peuvent être exécutés en parallèle.
seq	Séquençage Faible	1- Dans chaque opérande, l'ordre des spécifications d'occurrences d'événements est conservé dans le résultat. 2- Les spécifications d'occurrences d'événements dans les différents opérandes sur les différentes lignes de vie peuvent être dans n'importe quel ordre. 3- Sur la même ligne de vie, la spécification d'occurrence d'événement du premier opérande s'exécute avant la spécification d'occurrence d'événement du deuxième opérande.
strict	Séquençage Strict	Cet opérateur d'interaction définit un ordre strict des opérandes.
neg	Négative	Le fragment combiné représente un ensemble de traces non valides.
critical	Région Critique	Le fragment combiné représente une région critique dont ses traces ne peuvent pas être entrelacées par d'autres spécifications d'occurrences d'événements.
ignore /consider	Ignorer/ Considérer	L'opérateur d'interaction « *ignore* » désigne que certains types de messages peuvent être considérés comme non significatifs, ils sont donc ignorés. D'autre part, l'opérateur d'interaction « *consider* » désigne les messages considérés dans le fragment combiné.
assert	Assertion	Dans ce cas, les seules continuations valides sont les séquences

		de l'opérande d'assertion.
loop	Boucle	L'opérande sera itéré un certain nombre de fois et la boucle se terminera si la garde est évaluée à faux.
ref	Référence	Le fragment combiné fait référence à une interaction définie dans un autre diagramme. Le cadre est dessiné pour couvrir les lignes de vie impliquées dans l'interaction.

1.6.3 UML 2.0 STM

Les diagrammes d'activité et les diagrammes d'interaction sont utiles pour décrire le comportement d'un système, mais parfois, l'état d'un objet ou d'un système est un facteur important dans son comportement. Dans de telles situations, il est utile de modéliser les états d'un objet et les événements provoquant des changements d'état.

La vue logique d'UML décrit les descriptions abstraites des parties d'un système, y compris quand et comment ces parties peuvent être dans différents états à l'aide de diagrammes d'états-transitions. Ces derniers présentent une forme spécifique des automates à états finis basée sur une variante orientée objet du formalisme *statechart* de David Harel. Un diagramme d'états-transitions est un ensemble d'états interconnectés par des transitions, chaque transition porte une étiquette qui se décompose en trois parties optionnelles : *Evénement[Condition]/Action,* c'est l'événement qui déclenche un éventuel changement d'état, et si la condition booléenne est vraie l'action sera exécutée pendant la transition (Fowler, 2003). La Figure 10 montre un exemple d'un diagramme d'états-transitions.

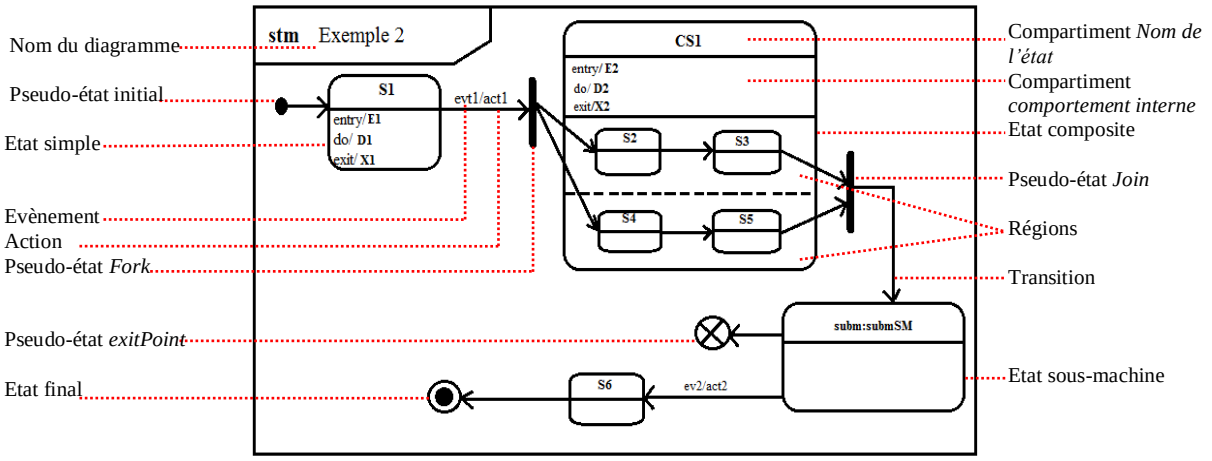
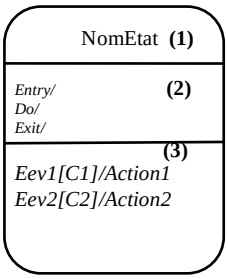
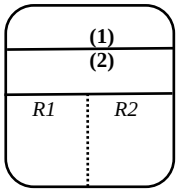
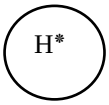
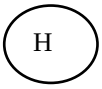
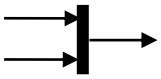
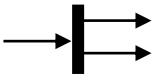
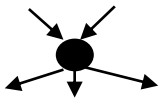
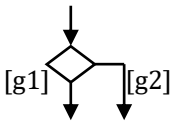
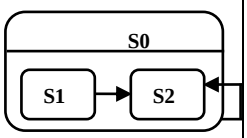


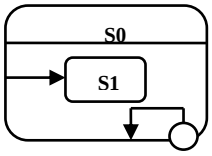
Figure 10. Exemple d'un diagramme d'états-transitions

L'apparence graphique ainsi que la description des éléments de base d'UML 2.0 STM sont résumés dans le Tableau 4.

Tableau 4. Eléments de base d'UML 2.0 STM

Composant		Apparence graphique	Description
Etat	Etat simple		Peut être divisé en trois compartiments : **(1) le nom d'état**, **(2) Le comportement interne :** tout comportement qui se produit pendant que l'objet est dans un état, *entry*: se produit dès que l'état devient actif par une transition externe, *do*: se produit pendant que l'état est actif, *exit* : se produit juste avant que l'état devient inactif. **(3) : les transitions internes :** un état peut réagir à un événement sans transition en mettant l'étiquette *Evénement[Condition]/Action* à l'intérieur de l'état elle-même.
	Etat composite		Contient au moins une région. Il peut être soit un simple état composite avec une seule région, soit un état orthogonal avec plusieurs régions concurrentes *(R1, R2)*. **(1)** et **(2)** sont identiques aux compartiments de l'état simple.
	Sous-machine		L'état sous-machine fait référence à un diagramme d'états-transitions entier, conceptuellement, il est considérée comme «imbriqué» dans l'état. Cet état est décrit comme un état normal, où **(1)** a la syntaxe suivante :*<nom de l'état>':'<nom du diagramme d'états-transitions référencé>*
	Etat final		Cet état marque la fin du diagramme d'états-transitions ou la fin de la région englobante.
	Initial *(Initial)*		Il représente le début d'exécution du comportement d'un diagramme d'états-transitions ou d'une région lorsqu'elle est entrée via l'activation par défaut.
	Historique profond *(deepHistory)*		C'est une sorte de variable qui représente la configuration d'état actif la plus récente de sa région propriétaire. Il implique la restauration de sa région dans cette même configuration d'état.
	Historique plat *(shallowHistory)*		C'est une sorte de variable qui représente le sous-état actif le plus récent de sa région propriétaire, mais pas les sous-états de ce sous-état. Il implique la restauration de la région à ce sous-état.

Pseudo-état	Jointure *(Join)*		C'est un sommet cible commun pour deux ou plusieurs transitions provenant de sommets dans différentes régions orthogonales. Il exécute une fonction de synchronisation, dans laquelle toutes les transitions entrantes doivent se terminer avant de poursuivre l'exécution via une transition sortante.
	Bifurcation *(Fork)*		Il permet de diviser une transition entrante en deux ou plusieurs transitions se terminant sur des sommets dans des régions orthogonales d'un état composite.
	Jonction *(Junction)*		Il permet de connecter une ou plusieurs transitions entrantes avec une ou plusieurs transitions sortantes (avec différentes contraintes de garde) représentant des chemins de continuation partagés. (diviser 1 en n, fusionner n en 1 ou connecter n à m).
	Choix *(Choice)*		Il permet de diviser les transitions composées en plusieurs chemins alternatifs dont les contraintes de garde sont évaluées dynamiquement. Par conséquent, le choix est utilisé pour réaliser une branche conditionnelle dynamique.
	Point d'entrée *(entryPoint)*		Ce pseudo-état représente un point d'entrée d'un diagramme d'états-transitions ou d'un état composite, il assure l'encapsulation de l'intérieur de son élément propriétaire. Si l'état composite est orthogonal, le point d'entrée agit comme un pseudo-état de bifurcation.
	Point de sortie *(exitPoint)*		Ce pseudo-état représente un point de sortie d'un diagramme d'états-transitions ou d'un état composite, il assure l'encapsulation de l'intérieur de son élément propriétaire. Si plusieurs transitions de régions orthogonales au sein de l'état composite se terminent sur ce pseudo-état, il agit comme un pseudo-état de jointure.
	Terminer *(Terminate)*		L'utilisation de ce pseudo-état implique la terminaison immédiate de l'exécution du diagramme d'états-transitions. Ce dernier ne quitte aucun état et n'exécute aucun comportement de sortie.
Transition	Externe		Elle quitte son sommet source, et si ce dernier est un état, l'exécution de cette transition entraînera l'exécution de tout comportement de sortie associé à cet état.

	Locale		Une transition locale ne peut exister qu'au sein d'un état composite et les deux sommets cible et source doivent être différents. Elle ne quitte pas son état propriétaire et par conséquent, le comportement de sortie ne sera pas exécuté.
	Interne		C'est un cas particulier d'une transition locale où les deux sommets source et cible sont identiques et l'état n'est jamais quitté. Par conséquent, aucun comportement de sortie ou d'entrée n'est exécuté.

1.6.3.1 Séquence d'exécution des transitions

La sémantique d'entrée dans un état dépend du type d'état et de la manière dont il est entré. Cependant, le comportement d'entrée n'est exécuté qu'après la terminaison d'exécution de tout événement associé à la transition entrante. De plus, si un comportement *do* est défini pour l'état, ce comportement commence l'exécution immédiatement après l'exécution du comportement d'entrée (*entry*). Ensuite, c'est le comportement *exit* qui s'exécute lors de la sortie d'un état, et si le comportement *do* est toujours en cours d'exécution lorsque l'état est quitté, ce comportement est abandonné avant le lancement du comportement de sortie. Finalement, au cours d'une transition, un certain nombre de comportements d'actions peuvent être exécutés. Cet ordre d'exécution des transitions est illustré par l'exemple présenté dans la Figure 11. Dans ce cas, lorsque l'événement "*sig*" est envoyé et la machine d'état est dans l'état "*S11*", la séquence d'actions suivante sera exécutée: *xS11;t1;xS1;t2;eT1;eT11;t3;eT111.*

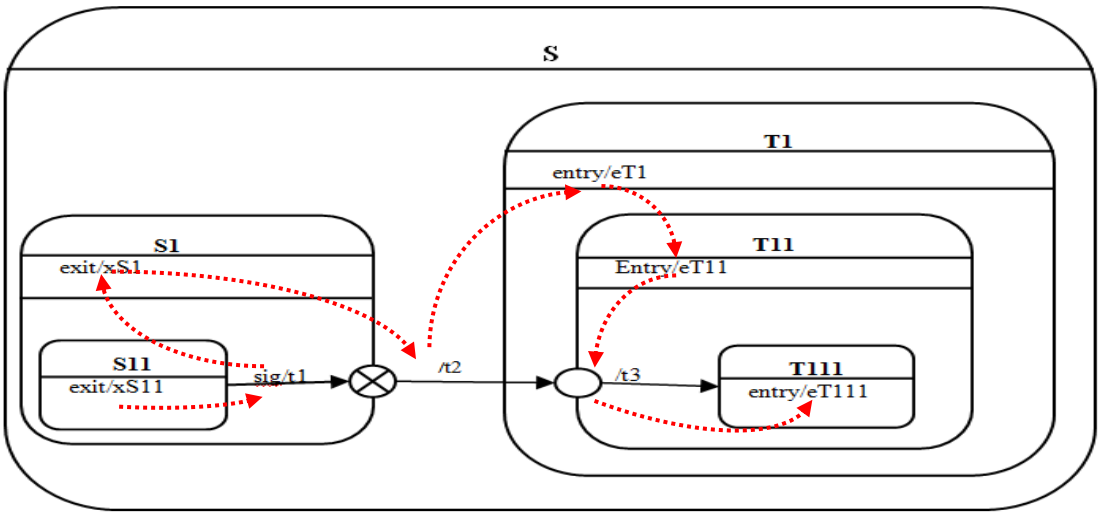


Figure 11. Exemple d'une transition composée (UML, 2017)

1.6.4 UML 2.0 CD

Les cas d'utilisation spécifient les exigences fonctionnelles du système. Les classes décrivent les différents types d'objets du système pour répondre à ces exigences. Un diagramme de classes fait parti de la vue logique d'UML, il décrit les classes et les relations statiques entre eux, il montre également les propriétés et les opérations des classes et les contraintes de connexion des objets. Les classes abstraites permettent de décrire partiellement le comportement commun à plusieurs classes (attributs et méthodes), les interfaces poussent les classes abstraites un peu plus loin en spécifiant uniquement les opérations nécessaires d'une classe mais sans aucune implémentation d'opération (Miles & Hamilton, 2006). La Figure 12 présente un exemple d'un diagramme de classe.

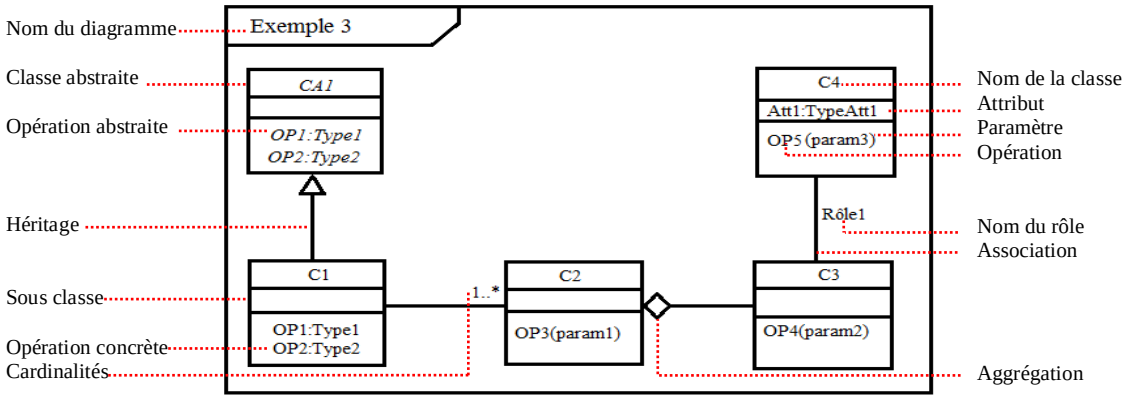


Figure 12. Exemple d'un diagramme de classes

Les principaux éléments structurels d'un diagramme de classes sont présentés dans le Tableau 5.

Tableau 5. Eléments de base d'un diagramme de classes

Composant		Apparence graphique	Description
Classe		NomClasse **(1)** attribut1 :type1 attribut2 :type2 **(2)** Operation1 **(3)** Operation2	Divisée en trois compartiments dont (2) et (3) sont facultatifs. (1) : contient le nom de la classe, (2) : contient les attributs de la classe, (3) : contient les opérations qui représentent le comportement de la classe.
	dépendance	<- - - - -	lorsque des objets d'une classe fonctionnent brièvement avec des objets d'une autre classe.

Relation (Miles & Hamilton, 2006)	Association		lorsque des objets d'une classe fonctionnent avec des objets d'une autre classe pendant une durée prolongée.
	agrégation		lorsqu'une classe possède mais partage une référence à des objets d'une autre classe.
	composition		lorsqu'une classe contient des objets d'une autre classe.
	Héritage		quand une classe est un type d'une autre classe.

1.7 CONCLUSION

Ce chapitre a été consacré aux fondements de l'Ingénierie Dirigée par les Modèles et à la modélisation de systèmes en utilisant le langage UML 2.0. Dans la première partie, nous avons commencé par une description des notions de base de l'IDM et une présentation de l'approche MDA. Nous avons exposé les approches de transformation de modèles, les outils de transformation de graphes existants, ainsi que l'outil de méta-modélisation multi-formalisme AToM³. Concernant la deuxième partie, nous avons clarifié la distinction entre un diagramme UML et un modèle. Ensuite, nous avons introduit les diagrammes du langage UML 2.0, mettant l'accent sur UML 2.0 CD, UML 2.0 SD et UML 2.0 STM, lesquel sont utilisés dans le cadre de cette thèse.

En conclusion, bien que UML soit riche, claire et largement utilisé, il pèche par le manque de rigueur nécessaire pour la vérification formelle des systèmes.

CHAPITRE

2

LE CSP ET LA VERIFICATION FORMELLE

2.1 INTRODUCTION

Au fil des années récentes, les dépenses liées au développement de logiciels et à leur maintenance ont pris une place importante dans l'ensemble du système. La localisation des pannes logicielles est extrêmement difficile et coûteuse. Selon une étude menée par Microsoft, il faut environ 12 heures de programmation pour localiser et corriger un défaut logiciel. À ce rythme, cela peut prendre plus de 24 000 heures pour déboguer un programme de 350 000 LOC (Lines Of Code). De plus, les phases d'analyse et de tests de conception cumulées représentent environ 64 % du temps total de développement (Pham, 2006).
Une étude menée par (Selby & Selby, 2007) a révélé que 72,5 % des défauts logiciels proviennent des phases de spécification et de conception. Les développeurs parviennent à détecter 47 % des défauts déphasés, issus de phases de développement antérieures, et détectables dans des phases ultérieures. Les phases de spécification, de conception préliminaire et de conception détaillée présentent respectivement des taux de détection de défauts déphasés de 58,3 %, 23,6 % et 36,9 %. Cette détection tardive entraîne une augmentation significative des délais et des coûts de développement.

D'après ce qui précède, on peut déduire que la vérification des systèmes doit être effectuée dès les premières phases du processus de développement logiciel.

Dans ce chapitre, nous explorerons le domaine de l'ingénierie système, les langages de spécification, et plus particulièrement les langages de spécification formels. Nous examinerons également les différentes méthodes de vérification, notamment les tests, la simulation, le model-checking et le theorem-proving. Nous présenterons également le langage de spécification formel CSP en nous concentrant sur sa syntaxe, ses sémantiques et ses outils de vérification automatique. Dans cette perspective, nous introduirons FDR4 en tant que model-checker du langage CSP.

2.2 INGENIERIE SYSTEME

Devant la complexité croissante des systèmes, il devient impératif de disposer d'un ensemble d'activités organisées pour la conception et le développement du système de

manière structurée. Cette nécessité a conduit à l'émergence d'un nouveau domaine connu sous le nom d'ingénierie système (Yangui, 2016).

L'Association Française d'Ingénierie Système (AFIS), rassemblant plusieurs entreprises industrielles impliquées dans le développement de systèmes complexes, propose les deux définitions suivantes concernant l'ingénierie système et les exigences, respectivement :

Ingénierie Système : « *une démarche méthodologique générale qui englobe l'ensemble des activités adéquates pour concevoir, faire évoluer et vérifier un système apportant une solution économique et performante aux besoins d'un client tout en satisfaisant l'ensemble des parties prenantes* » (de travail ingénierie système de l'AFIS, 2009).

Exigence : *« Une exigence prescrit une propriété dont l'obtention est jugée nécessaire. Son énoncé peut être une fonction, une aptitude, une caractéristique ou une limitation à laquelle doit satisfaire un système, un produit ou un processus »* (de travail ingénierie système de l'AFIS, 2009).

Les exigences peuvent être fonctionnelles ou non-fonctionnelles. Ces dernières peuvent être exprimées à travers un ensemble de propriétés définies en termes de contraintes temporelles, de qualité de service, de sûreté de fonctionnement, de sécurité informatique, etc (Yangui, 2016).

2.2.1 ingénierie des exigences

L'ingénierie des exigences représente une discipline essentielle au sein de l'ingénierie système. Elles représentent les exigences qui décrivent la vision du système du point de vue des concepteurs, formalisant ainsi l'expression des besoins qui représentent la vision du système du point de vue des utilisateurs. Le découpage du processus d'ingénierie des exigences peut varier selon les points de vue des différents auteurs. Selon Nuseibeh et Easterbrook (Nuseibeh & Easterbrook, 2000), ce processus peut être découpé en quatre phases, à savoir l'élicitation, la modélisation, l'analyse de spécification et la validation. En revanche, d'autres chercheurs, tels que Kotonya et

Sommerville (Kotonya & Sommerville, 1998), proposent une approche qui intègre la modélisation dans la phase d'analyse (Yangui, 2016).

La phase d'élicitation des exigences implique la collecte et l'identification des besoins auprès des parties prenantes, la modélisation vise à représenter ces exigences de manière structurée et compréhensible, l'analyse de spécification vise à évaluer leur cohérence et leur complétude, et enfin, la validation cherche à confirmer que les exigences répondent effectivement aux attentes des utilisateurs.

L'ingénierie des exigences vise à établir une compréhension commune entre les différents intervenants impliqués dans le système à concevoir. Cette compréhension est généralement formalisée à travers des documents constituant la spécification des exigences. Cette spécification est rédigée dans un langage de spécification, qui se divise en trois catégories principales : les langages informels, les langages semi-formels et les langages formels.

2.3 LANGAGES DE SPECIFICATION

2.3.1 Langages informels

Les langages de spécification informels sont des langages naturels bénéficiant d'une expressivité plus élevée que les langages semi-formels ou formels. Leur avantage réside dans leur compréhensibilité par divers acteurs du projet, une caractéristique hautement appréciée des professionnels industriels, qui les continuent d'utiliser pour spécifier les exigences. Cependant, ces langages présentent l'inconvénient d'être ambigus, et sujets à plusieurs interprétations, les rendant ainsi difficiles à analyser par une machine et potentiellement source d'erreurs. Ces derniers sont fréquemment introduites lors de la phase de conception.

La norme SBVR 1.5 du groupe OMG propose un dictionnaire contrôlé pour la spécification des exigences. À la différence d'un langage purement naturel, ces langages contrôlés offrent la possibilité d'analyser automatiquement les exigences (Kesraoui, 2017).

2.3.2 Langages semi-formels

Les langages semi-formels se caractérisent par une syntaxe formelle bien définie. Néanmoins, leur sémantique peut comporter des ambiguïtés d'interprétation. Comparativement aux langages formels, les langages semi-formels offrent une expressivité accrue et s'appuient souvent sur des notations graphiques ou textuelles, facilitant ainsi leur utilisation, bien qu'ils présentent simultanément des défis lors de l'analyse. L'analyse de ces langages nécessite souvent leur transformation en langages formels. Parmi les langages semi-formels, on peut citer UML et SADT (Structured Analysis and Design Technique).

2.3.3 Langages formels

Les langages formels se distinguent par une syntaxe et une sémantique formelles. Les notations mathématiques, fréquemment employées dans ce type de langages, offrent l'avantage de permettre la vérification automatique des spécifications. Cependant, elles présentent l'inconvénient de demander une expertise particulière, rendant leur utilisation difficile par les non-experts.
Les principaux travaux qui ont proposé des classifications pour catégoriser des langages formels sont : (Wing, 1990), (Barroca & McDermid, 1992), (Lamsweerde, 2000), (Almeida et al., 2011) et (Alagar et al., 2011).

Dans **(Kesraoui, 2017)** l'auteur a proposé une classification des langages formels à partir des classifications existantes. Il combine et adapte les classifications de **(Almeida et al., 2011)** et **(Lamsweerde, 2000)**, tout en gardant la classification en langages basés sur les modèles et langages basés sur les propriétés. La Figure 13 illustre cette classification.

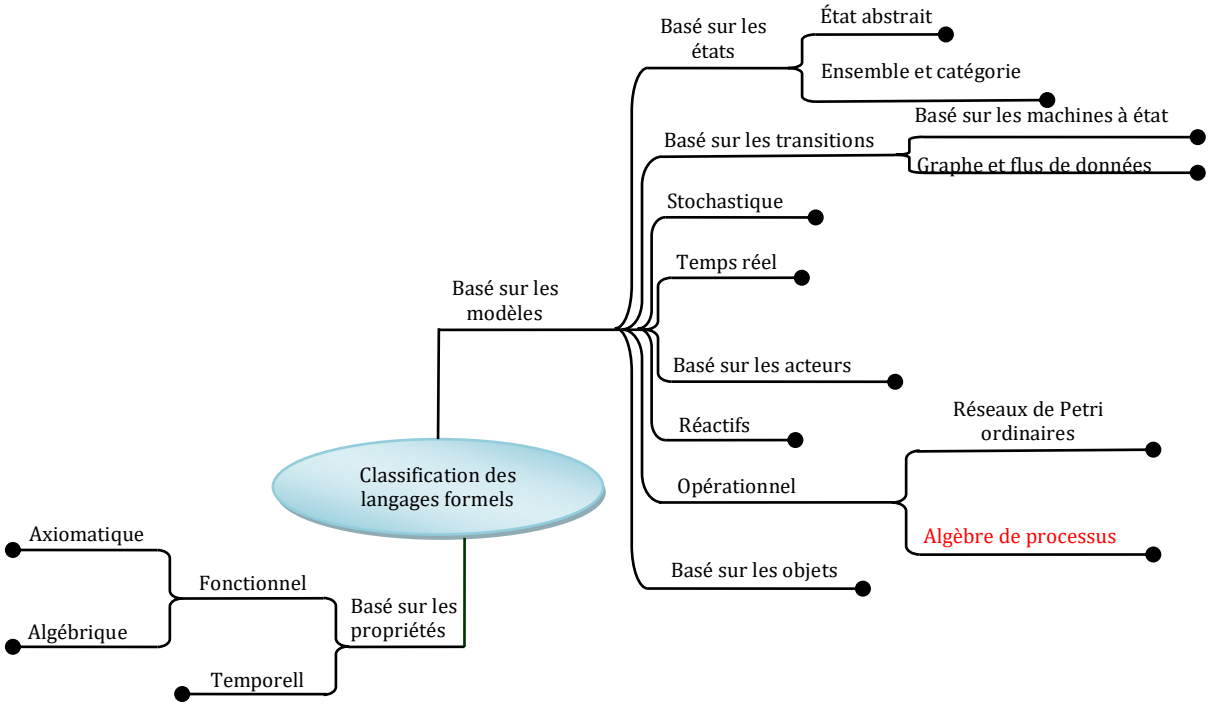


Figure 13. Classification des langages formels (Kesraoui, 2017)

2.3.3.1 Langages basés sur les modèles

Les langages basés sur les modèles permettent la description du système en utilisant ses états internes. Ce type de langage se caractérise par la notion d'état, et il englobe les classes suivantes :

- **Langages basés sur les états :** Cette catégorie englobe deux types de langages : les langages basés sur les états abstraits et ceux basés sur les ensembles. Le premier type, exemplifié par ASM (Abstract State Machines) (Gurevich & others, 1993) et B, permet de décrire le système en se focalisant sur ses états abstraits et les conditions qui régissent la transition entre ces états. En revanche, le deuxième type, représenté par Alloy, VDM (Vienna Development Method) et Z, offre la possibilité de décrire les états du système à travers des structures mathématiques telles que les ensembles, les relations et les fonctions.

- **Langages basés sur les transitions :** Cette classe de langage met l'accent sur les transitions entre les divers états d'un système. La description de ce dernier repose sur un ensemble de fonctions de transition qui associent à chaque état initial et à chaque

événement déclencheur l'état de sortie correspondant. Ces langages peuvent être classifiés en deux catégories distinctes : d'une part, les langages basés sur les machines à états, tels que les automates, les StateCharts et PROMELA (Promela, 2023) ; d'autre part, les langages basés sur les graphes et les flux de données, qui offrent la possibilité de représenter les données au sein du modèle, tels que SCR (Alspaugh et al., 1992), SDL (Rockstrom & Saracco, 1982) et Lustre (Halbwachs & Raymond, 1993).

- **Langages stochastiques :** Ces langages décrivent le système à l'aide d'un modèle probabiliste, où chaque transition est étiquetée par une probabilité. les chaînes de Markov s'inscrivent dans cette classe de langages.

- **Langages pour les systèmes temps réel :** Ces langages doivent être en mesure de spécifier diverses propriétés physiques, telles que le temps, la température, l'altitude, etc. (Almeida et al., 2011). Ils peuvent également être classés selon plusieurs critères, comme souligné par (Wang, 2004). Parmi ces langages, on peut citer les automates temporisés.

- **Langages basés sur les acteurs :** Ces langages permettent de spécifier la structure d'un système en utilisant les relations et les dépendances entre les différents acteurs du système. Un acteur, dans ce contexte, représente une entité active qui exécute des actions pour atteindre ses objectifs (Fuxman et al., 2004). Des exemples de ces langages incluent TROPOS (Fuxman et al., 2001) et le langage i* (Yu, 1997).

- **Langages réactifs :** Dans ce type de langages, Le système est modélisé par des entités qui s'exécutent en parallèle, jouant le rôle de répondre continuellement aux événements externes issus de l'environnement (Harel & Pnueli, 1984). REBECA (Sirjani et al., 2004) et ESTEREL (Berry & Gonthier, 1992) sont des exemples de ces langages.

- **Langages opérationnels :** Ces langages spécifient le système par un ensemble de processus capables de s'exécuter en parallèle. Nous pouvons distinguer deux catégories au sein de ces langages : les réseaux de Petri ordinaires et l'algèbre de processus tels que CCS et CSP. Une exploration détaillée de ce dernier sera abordée dans la section 2.5.

- **Langages basés sur les objets :** Ces langages adoptent la notion d'objet et proposent des formalismes pour modéliser le système à travers ses objets. Parmi ces langages, on

peut citer Object-Z (Smith, 2012), Disco (Järvinen & Kurki-Suonio, 1991) et TROL (Bucci et al., 1994).

2.3.3.2 Langages basés sur les propriétés

Les langages basés sur les propriétés décrivent le système en utilisant les données et les fonctions qui les manipulent. Ces langages peuvent être classés en langages fonctionnels et langages basés sur l'historique.

- **Langages fonctionnels :** Les langages fonctionnels décrivent le système sous la forme de collections mathématiques de fonctions, comme mentionné par (Lamsweerde, 2000). La notion centrale dans ce type de langage est celle de fonction. On distingue principalement deux types de langages : les langages algébriques et les langages axiomatiques. Dans les langages algébriques, les fonctions sont groupées selon les types d'objets de leur domaine, et les propriétés sont exprimées par des équations. Parmi ces langages, on cite LOTOS (Bolognesi & Brinksma, 1987) et CASL (Astesiano et al., 2002).

Les langages axiomatiques décrivent les fonctions du système en utilisant généralement des logiques d'ordre supérieur (Almeida et al., 2011). Les fonctions peuvent avoir des paramètres (Lamsweerde, 2000). Parmi les langages appartenant à cette catégorie, on peut citer HOL (Gordon, 1988), PVS (Owre et al., 1999), et COQ (Bertot & Castéran, 2013).

- **Langages basés sur l'historique :** Ce type de langage permet de décrire les traces acceptables du comportement du système à travers le temps (Lamsweerde, 2000). Les propriétés sont ainsi formulées à travers des assertions logiques portant sur les objets du système. Ces langages se divisent en deux catégories : les langages temporels et les langages stochastiques. Les logiques temporelles, telles que LTL et CTL, permettent de décrire les traces temporelles du système. À l'inverse, les logiques stochastiques, comme PCTL (Probabilistic Computation Tree Logic), permettent de décrire la probabilité des traces temporelles.

2.4 LES METHODES DE VERIFICATION

La vérification vise à garantir le respect des exigences découlant de la spécification tout au long du cycle de développement. Son objectif est de repérer les éventuelles erreurs introduites à chaque étape du processus de développement. Elle répond à la question: "Faisons-nous correctement le système?" Conformément à la norme ISO 9000, la vérification est définie comme :

> Verification : *« Confirmation, par la fourniture de preuves objectives, que les exigences spécifiées ont été satisfaites »* (ISO 9000, 2015)
>
> preuves objectives : *« Données confirmant l'existence ou la vérité de quelque chose »* (ISO 9000, 2015)

Les preuves objectives nécessaires à une vérification peuvent résulter d'une inspection ou d'autres formes de détermination telles que la réalisation de calculs alternatifs ou la revue de documents. Ils peuvent être obtenues par l'observation, la mesure, le teste ou par d'autres moyens (ISO 9000, 2015).

2.4.1 Le test

Le test constitue une méthode de vérification qui n'est pas exhaustive, mais qui permet de détecter les erreurs d'un système en fonction d'un ensemble de scénarios préalablement définis. Généralement pratiqué après la phase d'intégration du système, où la correction des erreurs identifiées s'avère souvent la plus coûteuse, le test demeure néanmoins la méthode la plus largement utilisée dans l'industrie. Cela s'explique par sa capacité à être appliqué sur des systèmes réels de grande taille, contrairement à d'autres approches telles que la simulation ou les méthodes formelles, qui s'appliquent sur une abstraction du système. (Kesraoui, 2017)

2.4.2 La simulation

La simulation représente une méthode de vérification non exhaustive qui ne couvre pas l'intégralité du comportement d'un système. Son principe consiste à reproduire le comportement d'un système afin de vérifier son bon fonctionnement. Cette

reproduction du comportement se matérialise sous la forme de modèles abstraits, qui sont ensuite exécutés sur des plateformes spécialisées. La simulation implique ainsi des expérimentations sur ces modèles abstraits, permettant à l'analyste de tester divers scénarios afin de vérifier les performances du système.

Tout comme le test, la simulation peut être utilisée pour évaluer des systèmes de taille réelle, ce qui explique son utilisation répandue dans l'industrie. Cependant, à la différence du test, la simulation peut être mise en œuvre à des stades antérieurs à l'implémentation et à l'intégration. Cette approche permet de détecter les erreurs avant que le système ne soit effectivement implémenté, contribuant ainsi à réduire les coûts de correction. Néanmoins, la simulation présente plusieurs inconvénients, tels que le degré de réalisme ou la granularité du simulateur (Kesraoui, 2017).

2.4.3 Les méthodes de vérification formelle

Les méthodes de vérification formelle sont conçues pour examiner rigoureusement des spécifications rédigées dans des langages formels, évaluant la satisfaction des propriétés définies dans la phase de spécification des exigences par le système en cours de développement. La nature mathématique de ces méthodes les rend plus précises et exhaustives en comparaison avec les approches basées sur les tests et la simulation. Les étapes de ce processus peuvent varier en fonction de la méthode utilisée, mais en général, ce processus peut être décomposé comme suit :

- **Spécification des propriétés :** La première étape consiste à définir formellement les propriétés que le logiciel doit satisfaire. Cette spécification peut être écrite en utilisant une logique ou un modèles mathématiques décrivant le comportement attendu du système.

- **Modélisation :** Une fois la spécification établie, le logiciel est modélisé sous la forme d'un modèle mathématique. Cette modélisation peut utiliser des langages formels, des automates, ou d'autres structures abstraites, selon la nature du système.

- **Vérification des propriétés spécifiées :** Selon la méthode choisie, la vérification peut être effectuée par le biais de différentes techniques. À la fin de cette étape, on peut déterminer si le système modélisé satisfait les propriétés attendues. En cas de détection

d'erreurs, le processus peut nécessiter des itérations pour corriger les défauts et répéter l'étape de vérification.

Les méthodes formelles se déclinent en deux catégories principales : le Model-Checking et le Theorem-Proving.

2.4.3.1 Theorem proving

Le Theorem-Proving est une méthode formelle qui exige de l'expérience et une compréhension approfondie des concepts formels. Elle repose sur l'utilisation de logiques mathématiques pour spécifier le système et définir ses propriétés attendues. Des règles d'inférence et des axiomes sont appliqués pour déduire et prouver les propriétés à partir de la spécification du système. Le Theorem-Proving peut être soit entièrement automatique, soit requérir une intervention humaine.

Dans le cas d'une logique moins expressive, comme la logique du premier ordre (par exemple, dans le cas du theorem prover ACL2), la construction de la preuve est totalement automatisée. En revanche, si la logique est plus expressive, comme la logique d'ordre supérieur, l'automatisation de la preuve est partielle. En raison de l'absence de procédures de décision capables de prouver certaines propriétés dans cette logique, l'assistant de preuve nécessite l'intervention de l'analyste pour guider le processus de démonstration.

Des outils tels que Isabelle/HOL servent d'assistants de preuve pour la logique d'ordre supérieur, tandis que PVS représente un langage de spécification formelle et un démonstrateur de théorèmes. Why3, quant à lui, constitue une plateforme de vérification de programmes basée sur le langage de spécification WhyML (Kesraoui, 2017)

2.4.3.2 Model-checking

L'idée fondamentale de cette méthode de vérification est d'explorer l'ensemble des états d'un système de transitions afin de s'assurer que la propriété spécifiée est respectée dans l'ensemble du système. Il s'agit donc de mettre en place un algorithme automatisant entièrement la tâche de vérification (Yack-Fa, 2018). Généralement, le

système est modélisé à l'aide de machines à états, tandis que les propriétés à vérifier sont exprimées dans une logique temporelle. En se basant sur les spécifications du système et les propriétés définies, le model-checker (outil informatique supportant le model-checking) explore l'ensemble des états possibles, vérifiant la satisfaction de la propriété désirée à chaque étape. En cas de violation de la propriété, le model-checker génère un contre-exemple illustrant la séquence d'états violant la propriété. Cependant, pour des systèmes de grande taille, cette exploration exhaustive des états peut dépasser les capacités des machines, entraînant un échec de la vérification. Ce problème est appelé l'explosion de l'espace des états (Kesraoui, 2017). La Figure 14 décrit le principe dumodel-checking.

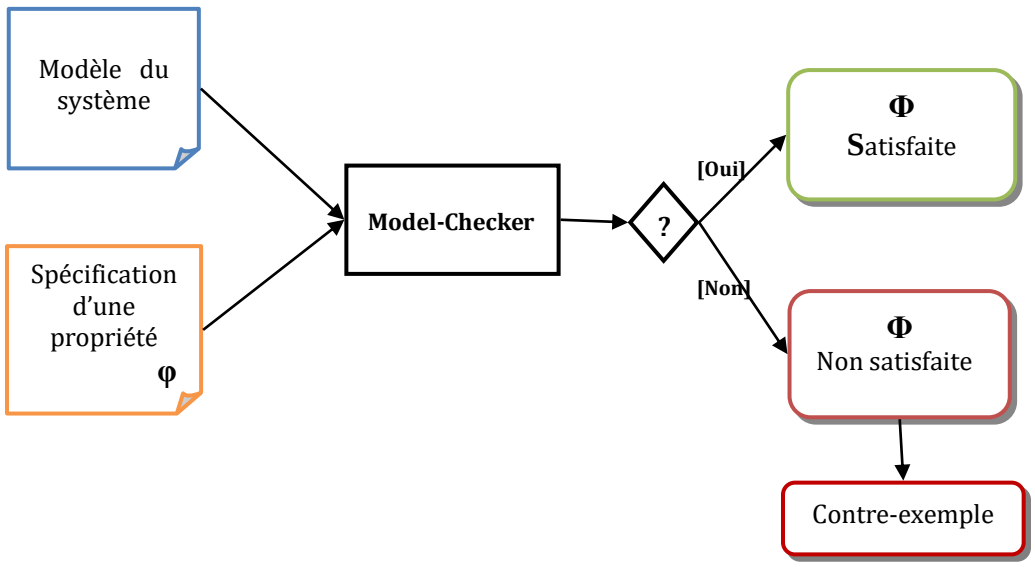


Figure 14. Principe de model-checking

- **Problème de l'explosion combinatoire d'états**

Cette méthode de vérification est fréquemment appliquée à des modèles de grande envergure. La taille d'un système est déterminée par le nombre d'états et de transitions qu'il comporte, cette dernière étant elle-même liée au nombre d'états. Par exemple, dans le cas où l'on envisage un programme comme un système dynamique, le nombre d'états du système augmente de manière exponentielle avec le nombre de variables du programme, entraînant ce que l'on appelle une "explosion combinatoire d'états". La représentation d'un programme par un système de transition d'états peut ainsi conduire à la création d'objets de taille considérable, ce qui s'avère souvent coûteux en termes

d'espace mémoire et de temps de calcul. Il est parfois nécessaire de calculer le résultat de la synchronisation de structures, comme le produit d'automates, ce qui accroît encore davantage l'espace d'états final.

Diverses techniques ont été développées pour tenter de résoudre ce problème. Certaines visent à réduire l'espace d'états en démontrant qu'il n'est pas toujours nécessaire de considérer le système dans sa totalité. D'autres proposent différentes représentations des données afin de comprimer l'espace d'états. (Yack-Fa, 2018)

2.5 CSP

CSP est une notation pour décrire les systèmes concurrents où il y a plus d'un processus à la fois. Il a été introduit pour la première fois par C.A.R. Hoare, dans son livre intitulé " Communicating Sequential Processes" (Hoare, 1978). C'est une collection de modèles mathématiques et de méthodes de raisonnement qui nous aident à comprendre et à utiliser cette notation (Roscoe A. W., 1997). Dans CSP, les processus sont des entités indépendantes qui interagissent les unes avec les autres par le biais de la communication. Un processus peut exécuter des événements qui décrivent son comportement, et l'ensemble des événements que le processus *P* peut effectuer est appelé son alphabet. Les interactions entre les processus ou avec leur environnement sont présentées dans un style algébrique à l'aide d'un ensemble d'opérateurs.

2.5.1 Syntaxe du CSP

Les opérateurs CSP les plus utilisés peuvent être résumé dans l'expression (1) (les notations utilisées sont : *P, Q* : processus ; *X, Y* : ensembles d'événements ; *e* : événement ; et *b* : condition booléenne) :

$$P \triangleq P \mid Stop \mid Skip \mid e \rightarrow P \mid \textit{if } b \text{ then } P \text{ else } Q \mid P \,[]\, Q \mid P \mid\sim\mid Q \mid P \backslash X \mid P;Q \mid P \,[X \| Y]\, Q \mid P \,[|X|]\, Q \mid P /// Q. \qquad (1)$$

La description des opérateurs CSP est résumée dans le Tableau 6.

Tableau 6. Description des opérateurs CSP

Expression	Description
Stop	Le processus CSP du blocage . Désigne le comportement le plus simple, il ne fait rien.
Skip	Processus qui modélise une terminaison réussie.
e → P	*Préfixe* : exécute l'événement *e* puis se comporte comme le processus *P*.
if b then P else Q	*Expression gardée* : se comporte comme *P* si la garde booléenne *b* est vraie, et comme *Q* sinon..
P [] Q	Choix externe (déterministe) : propose à l'environnement de choisir entre les événements initiaux de *P* et *Q*.
P |~| Q	Choix interne (non déterministe) : choisir l'un des processus *P* et *Q*, puis exécutez le processus choisi.
P \ X	*Hide:* se comporte comme P, mais les événements de X sont masqués et transformés en événements internes.
P ; Q	*Composition séquentielle* : se comporte comme *P* jusqu'à sa terminaison réussie, puis se comporte comme *Q*.
P [X||Y] Q	*Parallèle alphabétique* : exécution parallèle synchronisée de *P* et *Q* sur l'ensemble des événements X∩Y.
P*[|*X*|] *Q	*Parallèle généralisé* : exécution parallèle synchronisée de *P* et *Q* sur les événements de X.
P ||| ***Q***	*Entrelacement* : exécution parallèle non synchronisée de *P* et *Q*.

2.5.2 Les approches sémantiques

Roscoe dans (Roscoe A. W., 1997) a mentionné l'existence de trois facons distinctes pour comprendre la signification mathématique d'un programme CSP. Il s'agit de la sémantique opérationnelle, dénotationnelle et algébrique.

2.5.2.1 Opérationnelle

La sémantique opérationnelle est relativement proche des implémentations, elle peut être défini comme une formalisation mathématique d'une stratégie d'implémentation.

Elle interprète les programmes comme des diagrammes de transition, avec des actions visibles et invisibles pour passer d'un état de programme à l'autre.

2.5.2.2 Dénotationnelle

La sémantique dénotationnelle mappe un langage en un modèle abstrait dont la valeur de tout programme composé soit déterminable directement à partir des valeurs de ses parties immédiates. Les différentes sémantiques dénotationnelles de CSP sont basées sur les traces, les échecs et les divergences ; la valeur de tout programme étant juste une combinaison des ensembles de ces choses.

Le modèle de traces associe à chaque processus *P* les séquences de communications entre ce processus et son environnement. En d'autres termes, le modèle de traces permet de modéliser les comportements possibles de processus sous forme de traces. *traces(P)* désignent les traces du processus *P*. En général, une trace peut être soit finie parceque l'observation a pris fin ou parceque le processus et l'environnement atteignent un point où ils ne peuvent s'accorder sur aucun événement, soit infinie lorsque l'observation s'éternise et des nombreux événements sont traités infiniment.

Basé sur ce qui précède, *traces(P)* nous disent ce qu'un processus peut faire. Deux processus différents peuvent avoir les mêmes traces. Afin de les distinguer, nous devons enregistrer non seulement ce qu'ils peuvent faire, mais aussi ce qu'ils peuvent refuser de faire. *refus(P)* est l'ensemble des refus initiaux de *P*, mais nous avons besoin de savoir ce qu'il peut refuser après l'une de ses traces. Un échec stable est un couple *(s,X)*, où $s \in traces(P)$ et $X \in refus(P/s)$. (*P/s* représente le processus *P* après la trace *s*). *failures(P)* représente l'ensemble de tous les échecs de *P*.

La divergence est un type de comportement où un programme exécute une séquence infinie et ininterrompue d'actions internes. Il tourne en boucle infinie sans aucune intéraction avec son environnement. Le modèle échecs/divergences associe à chaque processus *P* l'ensemble de ses échecs et l'ensemble de ses divergences stables noté *divergences(P)*. Ce dernier est l'ensemble des traces *s* telle que le processus *P* entre dans une séquence infinie d'actions internes (*P* est divergent).

2.5.2.3 Algébrique

La sémantique algébrique est définie par un ensemble de lois algébriques. Ces derniers sont les axiomes de base de la sémantique, et l'équivalence de processus est définie en fonction des égalités qui peuvent être prouvées en les utilisant.

2.5.3 Outils CSP

Le langage CSP dispose de nombreuses extensions, chacune étant accompagnée d'outils correspondants. Parmi ces extensions, on peut citer, par exemple FDR (FDR, 2020), CSP++ (Gardner, 2005)et ProB (Leuschel & Butler, 2003) pour CSPм (CSPM, 2023), ainsi que PAT (Sun J. et al., 2009) pour CSP# (Sun J. et al., 2009). L'utilisateur peut choisir l'extension et l'outil de vérification approprié en fonction des caractéristiques du système (Shi et al., 2012).

Dans (Carvalho et al., 2011), les auteurs ont réalisé une comparaison entre FDR et PAT en fonction des données et des aspects comportementaux. Dans (Shi et al., 2012) les auteurs ont prolongé la démarche présentée dans (Carvalho et al., 2011) et ont envisagé un éventail plus large de comparaison de CSPм et CSP#, ainsi qu'entre leurs outils FDR, ProB et PAT. Ils ont exploré également les fonctionnalités de modélisation du point de vue de leur syntaxe et de leur sémantique opérationnelle. Les résultats de ce travail peuvent guider les utilisateurs dans la sélection des langages de modélisation et des outils appropriés pour spécifier et vérifier les systèmes concurrents. Parmi les résultats de ce travail, les auteurs ont préconisé l'utilisation du CSPм pour modéliser les systèmes avec un comportement abstrait ou qui impliquent une synchronisation d'événements en plusieurs parties, et l'utilisation de CSP# pour gérer les systèmes qui nécessitent des variables partagées ou qui implémentent des mécanismes de transmission de messages. Roscoe dans (Roscoe A. W., 2023) a décrit un certain nombre d'outils permettant à ses utilisateurs d'animer, d'analyser et de vérifier des programmes écrits en CSP.

- **FDR** (Failures Divergences Refinement) : est l'outil CSP le plus connu, c'est un vérificateur de raffinement pour les modèles comportementaux de CSP tels que les traces et les échecs. Il ne fonctionne que sur les processus à états finis.

- **ProBE** (Process Behavior Explorer) : est un animateur pour CSP basé sur le même compilateur CSPM que FDR. Il permet à l'utilisateur d'explorer l'espace d'états des processus CSP et de rechercher certains types d'états et d'actions.
- **Checker** : est un prototype de vérificateur de type CSPM. Il n'est pas intégré à ProBE et FDR car son implémentation a rencontré beaucoup de bifurcations dans la recherche d'un type pour un programme CSPM, à cause de sa façon d'interpréter le point utilisé dans la construction des événements et des types de données, Ce qui mène parfois à un éxécution très lent et produire une sortie très répétitive sur certains programmes. C'est un outil pour un utilisateur relativement expérimenté en CSPM plutôt que pour un débutant.
- **ProB :** est principalement un outil d'analyse des spécifications B, mais il permet l'intégration de CSP et B, prenant donc en charge une analyse de CSP, plus précisément du CSPM. Il fournit une interface d'animation comme ProBE ainsi que des capacités de vérification de modèle (LTL, deadlock, raffinement de trace). Il a une bonne intégration de la syntaxe et de la sémantique. Il est plus tolérant aux processus avec des espaces d'états infinis et grands. son exploration d'état sur la plupart des exemples est beaucoup plus lente que FDR.
- **ARC** (Adélaïde Refinement Checker) : a été développé en tant qu'outil de développement et de vérification CSP à l'Université d'Adélaïde. Il avait un animateur, un vérificateur de type et un vérificateur de raffinement pour CSP.
- **PAT** (Process Analysis Toolkit) : est un environnement de vérification de modèle ambitieux, dont l'un des modes vérifie une interprétation plutôt libérale de CSP d'une manière qui ne semble pas proche de la sémantique définie dans (Roscoe A. W., 2010). Son langage inclut des variables partagées dans CSP.
- **CSP Prover :** est une intégration de la théorie des CSP avec le démonstrateur de théorèmes Isabelle. Il tente de prouver des résultats généraux plutôt que d'analyser automatiquement les espaces d'états des systèmes d'états finis.

  Après l'examen des model-checkers pour CSP, notre préférence s'est orientée vers FDR. Les principales caractéristiques qui le distinguent par rapport aux autres outils seront explorées dans la section suivante.

2.6 FDR4

Le modèle checker FDR4 (Failures Divergence Refinement) est un outil puissant de vérification formelle utilisé pour garantir la fiabilité des systèmes concurrents. Il est conçu pour analyser les programmes écrits en CSP_M, la machine lisible de CSP qui combine les opérateurs CSP de Hoare avec un langage de programmation fonctionnel (Gibson-Robinson et al., 2014).

2.6.1 Présentation

Principalement, FDR4 cheche à découvrir si un programme CSP affine un autre, c'est-à-dire s'il réduit le nombre d'observations possibles. Ainsi, le raffinement coïncide essentiellement avec la réduction du non-déterminisme. FDR est un vérificateur de modèles, mais inhabituel car il se concentre sur le raffinement. Il compte une large base d'utilisateurs dans le monde universitaire, gouvernemental et industriel, à la fois comme outil en soi et également comme « back-end » pour d'autres outils. Sa conception a été dirigée par Bill Roscoe depuis sa première rédaction en 1991 (Roscoe A. W., 2023).

FDR4 comprend une interface interactive complète qui permet d'évaluer des expressions, de fournir de nouvelles définitions et de définir de nouvelles assertions. Il intègre également un outil de visualisation de débogage puissant qui facilite la compréhension des contre-exemples. Cette interface est illustrée par la Figure 15.

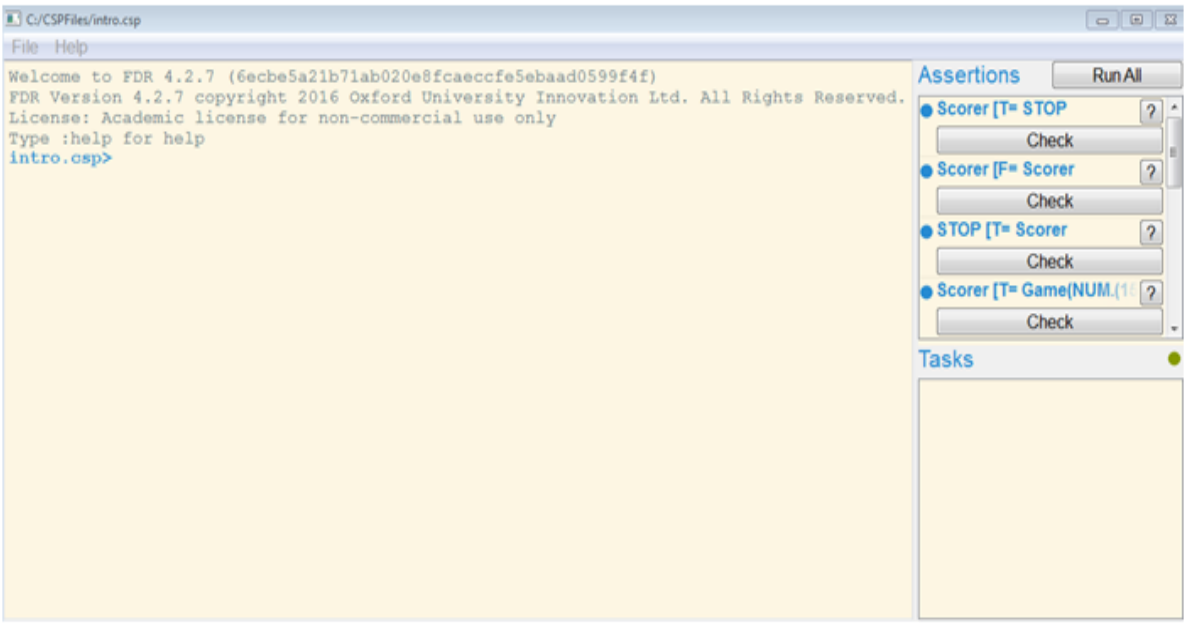


Figure 15. Interface de l'outil FDR4

FDR4 contient un moteur de vérification de raffinement parallèle capable de vérifier des processus avec des milliards d'états et d'utiliser efficacement le stockage sur disque pour compléter la mémoire. FDR4 comprend également une version en cluster qui permet d'obtenir une accélération linéaire à mesure que plus de nœuds de calcul sont ajoutés, permettant ainsi de résoudre efficacement d'énormes problèmes (Gibson-Robinson et al., 2014).

2.6.2 Les assertions

Le premier objectif pour FDR4 n'est pas de décrire des algorithmes sous une forme exécutable mais pour prendre en charge la description de systèmes parallèles sous une forme de manipulation automatique. FDR4 permet diverses formes d'assertions pour vérifier les propriétés de comportement. Si P et Q sont deux processus, et S est l'un des trois modèles sémantiques :

- T: Traces (traces).
- F: Failures (échecs).
- FD : Failures-Divergences (échecs-divergences).

alors FDR peut simplement vérifier si Q affine P *(P ⊑S Q)* en insérant l'assertion suivante dans le code CSP_M : *assert P[s = Q*.

2.6.3 CSP_M

CSP_M combine l'algèbre de processus CSP avec un langage de programmation fonctionnel qui est inspiré des langages comme Miranda/Orwell et Haskell/Gofer. Les langages de programmation sont utilisés pour décrire des algorithmes sous une forme exécutable, mais le premier objectif pour CSP_M est de supporter la description des systèmes parallèles sous une forme pouvant être manipulée automatiquement (Scattergood & Armstrong, 2011).

2.6.3.1 Opérateurs repliqués

FDR propose une version répliquée ou indexée d'un certain nombre d'opérateurs. Ceux-ci fournissent un moyen simple de construire un processus qui englobe un certain nombre de processus composés à l'aide du même opérateur. La forme générale d'un

opérateur répliqué est : *op <déclarations> @ P* où *op* est un opérateur, les *déclarations* sont une liste de déclarations et *P* est la définition du processus. *P* est évalué pour chaque valeur prise par les *déclarations,* ensuite les processus sont composés ensemble à l'aide de l'*op* (Gibson-Robinson et al., 2014). Une brève description de quelques opérateurs repliqués, ainsi que le résultat retourné en cas de la composition vide des processus, sont résumés dans le Tableau 7**.**

Tableau 7. Description des opérateurs repliqués

Expression	Description	Si le résultat de la composition est vide
[ ] <set statements> @ P ***<definir des déclarations>*** ***<énoncés>***	*Choix externe répliqué* : pour chaque valeur des déclarations, *P* est évalué puis les processus résultants sont composés à l'aide de l'opérateur [].	Retourner STOP.
|~| <set statements> @ P(x)	*Choix interne répliqué* : pour chaque valeur des déclarations, *P* est évalué puis les processus résultants sont composés à l'aide de l'opérateur |~|.	Afficher ERREUR
|| <set statements> @ [A] P	*Parallèle alphabétique répliqué* : pour chaque valeur des déclarations, *P* et son alphabet *A* sont évalués, puis les processus résultants sont composés à l'aide de l'opérateur parallèle alphabétique.	Retourner SKIP.
[|X|] <set statements> @ P	*Parallèle généralisé répliqué* : pour chaque valeur des instructions, *P* est évalué, puis les processus résultants sont composés et synchronisés sur *X* à l'aide de l'opérateur parallèle généralisé.	Retourner SKIP.
||| <set statements> @ P	*Entrelacement répliqué* : pour chaque valeur des instructions, *P* est évalué puis les processus résultants sont composés à l'aide de l'opérateur d'entrelacement.	Retourner SKIP.

2.6.3.2 Séquences et ensembles

CSP_M est un langage riche, il définit les processus, les fonctions, les expressions, et inclut un support direct pour les séquences, les ensembles, les booléens, les tuples, les types définis par l'utilisateur, les définitions locales, la correspondance de modèles et les termes lambda.

CSP_M n'applique aucune restriction sur l'utilisation des lettres majuscules ou minuscules dans les identifiants. Il est cependant courant que les utilisateurs adoptent une convention sur l'utilisation des identifiants . Par exemple, Les noms des processus doivent être en majuscules, les types et les constructeurs de type doivent commencer par une majuscule, et les fonctions et canaux doivent être en minuscules (Scattergood & Armstrong, 2011). Le Tableau 8 présente la description de quelques séquences et ensembles dans le langage CSP_M (s,t :séquence, x :expression générale, m,n :nombres, a :ensemble, A : ensemble d'ensembles, b : booléen)

Tableau 8. Séquences et ensembles dans CSP

Syntaxe		Description
Séquences	<>, <1,2,3>	séquences littéraux.
	<m..n>	plage fermée, de l'entier m à n inclus.
	s^t	caténation de séquence.
	#s, length(s)	longueur d'une séquence.
	null(s)	tester si une séquence est vide (s==<>).
	head(s)	le premier élément d'une séquence non vide.
	tail(s)	tous sauf le premier élément d'une séquence non vide.
	concat(s)	réunissent une séquence de séquences.
	elem(x,s)	teste si un élément apparaît dans une séquence.
	<x1,..., xn| x<-s, b>	compréhension.
Ensembles	union(a1,a2)	ensemble d'union
	inter(a1,a2)	ensemble d'intersection
	diff(a1,a2)	Ensemble de différence
	Union(A)	union distribuée
	Inter(A)	intersection distribuée (A doit être non vide)
	member(x,a)	test d'appartenance
	card(a)	cardinalité (nombre d'éléments)
	empty(a)	vérifie l'ensemble vide

	set(s)	convertir une séquence en ensemble
	Set(a)	tous les sous-ensembles de a
	seq(a)	convertit un ensemble en séquence
	Seq(a)	ensemble de séquences sur a
	{x1,..., xn| x<-a, b}	Compréhension
Booléens	true, false	booléens littéraux
	b1 and b2 ≡ if b1 then b2 else false	booléen *et*
	b1 or b2 ≡ if b1 then true else b2	booléen *ou*
	not b ≡ if b then false else true	booléen *non*
	x1==x2, x1!=x2	opérations d'égalité
	x1<x2, x1>x2, x1<=x2, x1>=x2	l'ordre de x1 et x2
	if b then x1 else x2	expression conditionnelle
Termes lambda	\ x1, ...xn @ x	terme lambda est une fonction sans nom. Par exemple, la définition f(x,y,z) = x+y+z est équivalente à f = \ x, y, z @ x+y+z

2.7 CONCLUSION

En conclusion de ce chapitre, nous avons parcouru les principaux éléments de l'ingénierie système, la classification des langages de spécification, ainsi que la classification des langages formels. Nous avons également exploré les différentes méthodes de vérification : les tests, la simulation, le model-checking et le theorem-proving . Le langage CSP ainsi que son model-checker FDR4 ont été présenté en détail.

Le chapitre suivant s'attache à une intégration spécifique du langage semi-formel UML 2.0 et du langage de spécification formel CSP. Nous explorerons d'une manière profonde les correspondances entre les deux langages source et cible de cette traduction.

Partie 2. Contributions

Chapitre 3. Formalisation des diagrammes UML 2.0 en CSP

Chapitre 4 . L'approche intégrée UML 2.0/ CSP proposée

Chapitre 5 . Étude de cas

CHAPITRE

3

FORMALISATION DES DIAGRAMMES UML 2.0 EN CSP

3.1 INTRODUCTION

L'intégration de langages de spécification semi-formel et formel présente un avantage considérable dans le domaine du développement logiciel. Les équipes de développement peuvent tirer parti de la clarté et de la compréhensibilité des spécifications semi-formelles tout en garantissant la correction formelle grâce aux spécifications formelles. Dans ce chapitre, nous nous intéressons à la formalisation des diagrammes UML 2.0 SD, UML 2.0 STM et UML 2.0 CD en utilisant le langage CSP. Cette formalisation est inspirée des travaux précédents, et notre contribution consiste à l'introduction de la formalisation en CSP de quatre pseudo-états d'UML 2.0 STM, à savoir les régions, la bifurcation (Fork), la jointure (Join) et la jonction (Junction). Pour chaque diagramme UML 2.0, nous présentons les correspondances entre les deux langages source et cible ainsi que le code CSP généré. Avant de présenter les détails de ce travail, nous passons en revue les méthodes formelles, les stratégies d'intégrations des langages formels et semi-formels, ainsi que les travaux connexes dans ce domaine.

3.2 METHODES FORMELLES

Dans [Bjorner2014] une méthode formelle est défini comme une méthode dont les techniques et les outils peuvent être exprimés mathématiquement. Si par exemple la méthode inclut, comme outil, un langage de spécification, alors ce langage a une syntaxe formelle, une sémantique formelle, et un système de preuve formel. Les techniques d'une méthode formelle aident à construire, analyser et/ou transformer une ou plusieurs spécifications en un programme.

Les méthodes semi-formelles utilisent des langages de spécification textuelles ou graphiques, dotés d'une syntaxe précise et une sémantique assez faible . Le Tableau 9 synthétise les principales différences entre les méthodes formelles et les méthodes semi-formelles (Idani, 2006).

Tableau 9. Méthodes formelles Vs Méthodes semi-formelles (Idani, 2006)

	Méthodes semi-formelles	Méthodes formelles
Formalisme	Textuel ou graphique	Mathématique

	(Merise, SADT, UML,…)	(Z, VDM, B,…)
Syntaxe du langage	Précise	Précise
Sémantique du langage	Assez faible	Précise
Validation	Syntaxique + Expertise humaine	Preuve, Démonstration de théorèmes, Model-checking, Animation et test
Outils	Ateliers de Génie Logiciel (AGL)	Prouveurs + Animateurs
Domaine applicatif	Se veulent généralistes	Systèmes sûrs ou critiques
Objectif	Systèmes bien structurés	Systèmes fiables

La disposition d'une méthode assurant le développement de systèmes logiciels automatiquement vérifiables à partir des spécifications claires, peut être assurée par l'intégration des langages de spécification formels et semi-formels. Cette intégration peut être réalisée selon différentes stratégies.

3.3 STRATEGIES D'INTEGRATION

Dans (Idani, 2006), l'auteur a suggéré (la Figure 16)une classification des travaux d'intégration des langages semi-formels et formels selon la stratégie de développement adoptée.

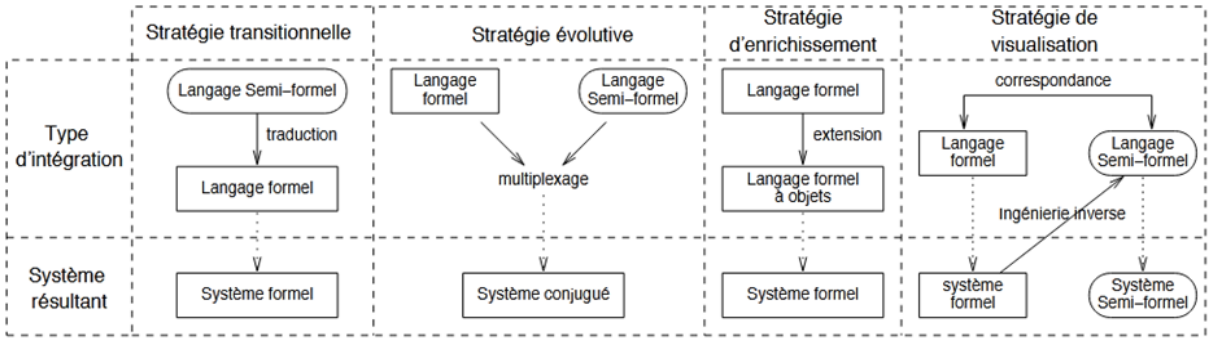


Figure 16. Stratégies d'intégration (Idani, 2006)

Dans le contexte d'un développement principalement fondé sur un langage semi-formel, l'intégration d'un langage formel se fait à travers deux stratégies : la stratégie transitionnelle et la stratégie évolutive.

• **Stratégie transitionnelle :** cette stratégie se base sur une intégration par traduction qui implique une phase de transition de spécifications semi-formelles vers des spécifications formelles équivalentes. Dans ce cadre, on peut enrichir, raffiner, etc., le modèle formel résultant, appliquer des techniques de vérification et de preuves, tout en permettant l'utilisation d'outils conçus autour des méthodes formelles.
Notre travail de thèse s'inscrit dans ce contexte. Nous adoptons une stratégie transitionnelle pour traduire les modèles UML 2.0 semi-formels en spécifications CSP formelles équivalentes.

• **Stratégie évolutive :** Cette stratégie consiste à étendre le langage semi-formel en y introduisant des notations formelles. Elle permet d'exprimer de manière formelle certaines propriétés du modèle semi-formel, faisant ainsi évoluer le système d'une simple description semi-formelle à un système conjugué plus précis. Le langage OCL utilisé pour la description des contraintes des diagrammes UML, peut être cité comme exemple de cette stratégie.

Dans le contexte d'un développement fondé sur un langage formel, l'intégration se réalise aussi à travers deux stratégies : la stratégie d'enrichissement et la stratégie de visualisation.

• **Stratégie d'enrichissement** : Cette stratégie considère le paradigme objet comme un mécanisme de structuration. Son objectif est de définir un langage de spécification formel basé sur le paradigme objet en étendant un langage de spécification formel déjà existant. Cette stratégie présente deux avantages majeurs : elle permet de disposer d'un langage de spécification formel favorisant la composition et la décomposition modulaire, tout en permettant la réutilisation des environnements de spécification et de preuve des méthodes formelles.

• **Stratégie de visualisation :** Cette stratégie vise à élaborer une représentation graphique complémentaire, bien que partielle, sur des spécifications formelles. Pour ce faire, elle requiert une phase de correspondance qui établit les liens structurels et sémantiques entre un sous-ensemble de constructions du langage formel source et

celles du langage semi-formel cible. Les modèles ainsi générés servent principalement de documentation graphique pour le modèle formel. (Idani, 2006)

3.4 TRAVAUX CONNEXES

Dans la littérature, plusieurs initiatives ont été publiées pour formaliser les diagrammes UML 2.0 SD en utilisant le langage CSP. Parmi ces travaux, celle référencée en (Jacobs & Simpson, 2014) mérite une attention particulière. Les auteurs dans cette étude ont contribué en fournissant à UML 2.0 SD une sémantique CSP indépendante de toute implémentation. Ils ont abordé les diagrammes de séquence sous un angle novateur, en les conceptualisant en termes d'observations d'occurrences plutôt qu'en termes de messages. En outre, leur travail a couvert en détail l'ensemble des opérateurs d'interaction, renforçant ainsi la rigueur de la formalisation.

Une autre approche, qui a été proposée dans (Dan & Danning, 2010), consiste en une traduction automatisée d'UML 2.0 SD en CSP en utilisant le langage de programmation XSLT (Kay, 2008). Cependant, il est important de noter que cette approche a présenté certaines restrictions, notamment la formalisation des fragments combinés avec seulement quatre opérateurs d'interaction, à savoir Opt, Break, Loop et Alt.

Pour renforcer l'exactitude des diagrammes de séquence, Kaizu et ses collaborateurs dans leur travail (Kaizu et al., 2013) ont introduit de nouveaux opérateurs CSP dans le cadre du synthèse des diagrammes de séquence. Toutefois, il convient de souligner que cette approche ne traite pas les fragments combinés, un travail qui nécessite encore des développements futurs.

Une autre contribution notable est rapportée dans (Kaizu et al., 2015), où les auteurs ont généré du code CSP à partir de diagrammes de séquence et ont développé un outil de vérification basé sur le model-checking PAT. En cas de détection d'un interblocage, un contre-exemple est affiché. Il est intéressant de noter que ce dernier peut être rétro-traduit en un diagramme de séquence pour faciliter la correction. Cependant, cette approche présente également des limitations, notamment l'omission des fragments combinés.

Il est important de mentionner que la vérification des diagrammes de séquence a fait l'objet d'une attention particulière. Dans diverses travaux, les chercheurs ont développé des méthodes pour traduire automatiquement ces modèles en d'autres formalismes pour effectuer des vérifications automatiques. Nous citons à titre d'exemple, le travail (Bouarioua et al., 2011) dans lequel les diagrammes de séquence ont été traduit en modèles de réseaux de Petri stochastiques généralisés étiquetés (Marsan et al., 1994), ainsi que (Custódio Soares et al., 2018) et (Mozaffari & Harounabadi, 2011) dans lesquels ces diagrammes ont été transformés en réseaux de Petri colorés (Jensen & Kristensen, 2009), et dans (Cunha et al., 2011) en réseaux de Petri.
De plus, dans le travail (Zhao et al., 2006), une transformation des diagrammes de séquence en codes Promela a été présentée pour évaluer la cohérence entre les diagrammes de séquence et les diagrammes d'états-transitions. De manière similaire, les auteurs de (Lima et al., 2009) et (Vidal-Silva et al., 2018) ont introduit une technique de vérification et de validation formelle des diagrammes de séquence, en utilisant le vérificateur de modèle Spin (Holzmann, 1997) pour analyser les propriétés exprimées en logique temporelle linéaire (LTL). Dans le travail (Hlaoui et al., 2017), un outil a été développé pour transformer les diagrammes de séquence en langage Event B (Russo, 2011), ou encore, dans (Messaoudi et al., 2019), la transformation multicouches a été utilisée pour générer des automates de Büchi (Li et al., 2018) à partir des diagrammes de séquence. Dans (Chabbat et al., 2020), une méthode a été établie pour traduire les diagrammes de séquence annotés avec des stéréotypes MARTE en réseaux de Petri temporels avec des spécifications de durée d'action.

Ng & Butler ont proposé deux travaux, que nous jugeons les plus importants, qui formalisent les diagrammes d'états-transitions et les diagrammes de classe en CSP. Les formalisations proposées ont été indépendantes de toute implémentation. Le premier travail (Ng & Butler, 2002) a introduit la formalisation des deux diagrammes cités ci-dessus, alors que le deuxième (Ng & Butler, 2003) a complété la formalisation des états composites des diagrammes d'états-transitions.

Les diagrammes d'activités ont été formalisés en CSP dans le travail (Xu et al., 2008) et (Bisztray et al., 2007). Ce dernier a été concrétisé à travers différentes implémentations,

Nous citons à titre d'exemple (Varró et al., 2007) en utilisant divers outils de transformation de graphes, dans (Elmansouri et al., 2011) en utilisant l'outil AToM^3, dans (Elmansouri et al., 2021) en utilisant AToMPM pour la méta-modélisation et l'outil GROOVE pour la transformation, et la vérification des propriétés. En outre, le travail (Kerkouche et al., 2020) a proposé une approche de transformation de graphes pour générer des spécifications Maude à partir de diagrammes d'activités UML, en utilisant l'outil AToM^3.

Le sujet de la vérification des transformations de modèle a été abordé dans (da Costa Cavalheiro et al., 2017), (Meghzili et al., 2017) et (Cuadrado et al., 2016). Les auteurs dans (Meghzili et al., 2019), ont vérifié la transformation de modèles en utilisant Isabelle/HOL (Nipkow et al., 2002) et Scala (Wenzel, 2012).

Discussion : Les travaux traduisant UML en CSP présentent plusieurs lacunes, parmi lesquelles:

1. L'approche ne prend pas en compte les aspects statique et dynamique du système simultanément.
2. L'approche proposée dépend d'une plateforme d'implémentation spécifique.
3. L'approche se limite à un sous-ensemble restreint des diagrammes UML formalisés.
4. L'approche se concentre uniquement sur l'aspect individuel du comportement des objets du système modélisé, sans prendre en considération la communication entre ces objets.
5. Les processus de transformation sont effectués manuellement.

3.5 FORMALISATION DES DIAGRAMMES UML 2.0 EN CSP

Le langage CSP a trouvé une application concrète dans l'industrie en tant qu'outil de spécification formelle pour l'exécution concurrente de divers systèmes. Le choix du CSP comme langage cible de la transformation des diagrammes UML 2.0 peut être justifié par diverses raisons, notamment parce qu'il représente une collection de modèles

mathématiques et de méthodes de raisonnement. Ainsi, des preuves mathématiques sont employées pour démontrer la validité de ce langage.

La richesse du CSP apparaît à travers l'existence de trois sémantiques - opérationnelle, dénotationnelle et algébrique - qui permettent de comprendre la signification mathématique d'un programme CSP. De plus, le langage CSP présente de nombreuses extensions, chacune accompagnée d'outils correspondants. Ces outils permettent aux utilisateurs d'animer, d'analyser et de vérifier des programmes rédigés en CSP. L'utilisateur peut ainsi choisir l'extension et l'outil de vérification appropriés en fonction des caractéristiques spécifiques du système (Shi et al., 2012).

L'UML et le CSP permettent la spécification de la structure et du comportement d'un système. On peut utiliser le diagramme de classe UML, par exemple, pour modéliser la structure d'un système, ainsi que les concepts de la sémantique algébrique ou dénotationnelle tels que les traces et les échecs en utilisant le CSP.

En UML, le comportement d'un système peut être modélisé en utilisant un ensemble de diagrammes d'états-transitions, où chaque diagramme décrit l'état d'un objet tout au long de sa durée de vie ainsi que les événements susceptibles de modifier cet état. De plus, le diagramme de séquence décrit les interactions entre ces objets, où l'ordre des interactions a une importance particulière. De manière similaire, en CSP, le comportement d'un système est modélisé par les interactions entre un ensemble de processus. Chaque processus spécifie le comportement d'un objet du système à travers son alphabet, qui est un ensemble d'événements exécutables par le processus. De plus, les opérateurs de choix et de parallélisme permettent de décrire l'ordre d'exécution des processus. L'opérateur de préfixe détermine également l'ordre d'exécution des événements dans le système spécifié. Cette capacité à décrire les objets, les interactions, les événements, ainsi que leur ordre d'exécution nous offre la possibilité de réaliser une traduction entre l'UML et le CSP.

3.5.1 Formalisation d' UML 2.0 SD

Dans le cadre de cette étude, nous adoptons de (Jacobs & Simpson, 2014) l'idée de refléter la structure d'un diagramme de séquence. Dans cette formalisation d'UML 2.0 SD en CSP, chaque ligne de vie est transformée à un processus nommé *Lifeline()*, chaque

message est transformé à un processus *Message()*, et chaque occurrence d'bservation est transformé à un événement CSP. L'envoi et la réception de messages sont séparés, et un message ne peut pas être reçu avant d'avoir été envoyé. La Figure 17 résume les correspondances entre UML 2.0 SD et CSP selon (Jacobs & Simpson, 2014).

- *Ligne de vie* → Processus CSP avec la séquence des événements envoyés et reçus dans le même ordre temporel.
- *Occurrence d'observation* → Evénement
- *Message* → Processus CSP avec les événements qui synchronisent les lignes émetteur / récepteur
- Prise en compte des *fragments combinés*.
- Composition séquentiel des *fragments combinés*.

Figure 17. L'approche de transformation de UML 2.0 SD en CSP

L'ordre temporel des occurrences d'observations sur une ligne de vie est respecté à l'aide du processus *PrefixComposition()*. Un séquençage *strict* est défini à l'aide d'un processus supplémentaire *Enforce()*, qui garantit l'ordre des messages sur toutes les lignes de vie participantes. Les processus *Lifelines()* et *Messages()* modélisent la composition parallèle des processus *PrefixComposition()* et *Message()*, respectivement. La figure 18 et la figure 19 montrent la formalisation des processus *Message()* et *PrefixComposition()*, respectivement.

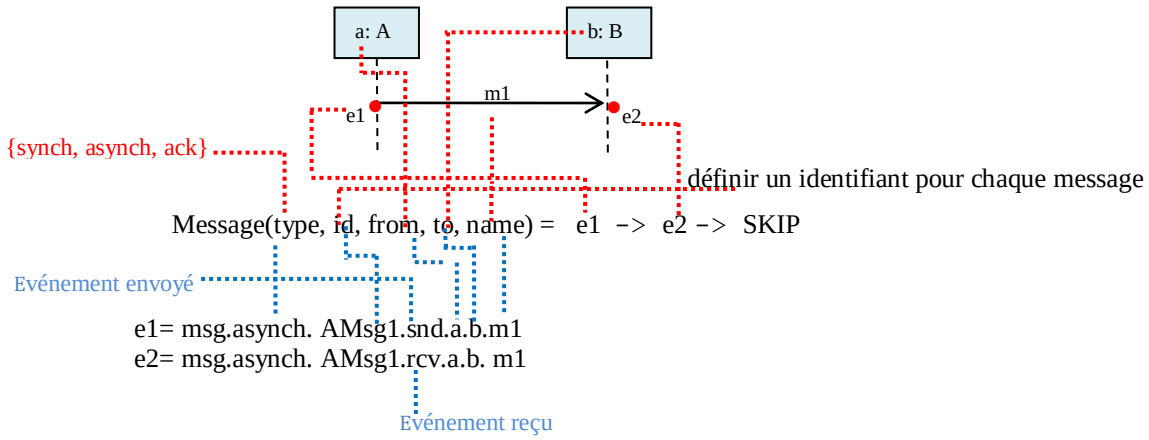
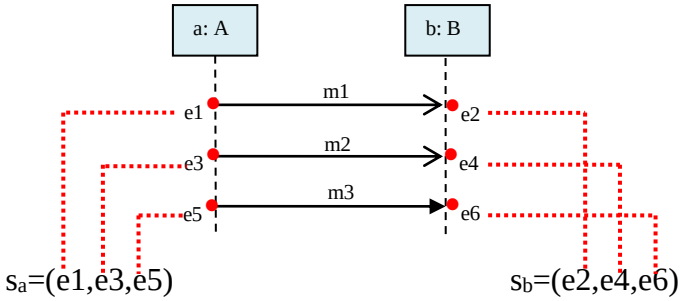


Figure 18. Formalisation du processus *Message()*



$PrefixComposition(s_i) = if\ null(s_i)\ then\ SKIP\ else\ head(s_i)\ \rightarrow\ PrefixComposition\ (tail(s_i))$

Figure 19. Formalisation du processus *PrefixComposition()*

La formalisation des processus proposés est résumée dans le Tableau 10.

Tableau 10. Formalisation des processus proposés pour la transformation d'UML 2.0 SD en CSP

Nom du processus	Code CSP généré
Message	*Message(type,id,from,to,name)=msg.type.id.snd.from.to.name → msg.type.id.rcv.from.to.name → skip alphaMessage(type,id,from,to,name)= {|msg.type.id.snd.from.to.name,msg.type.id.rcv.from.to.name|}*
PrefixComposition	*PrefixComposition(s)=if null(s) then skip else head(s) → PrefixComposition(tail(s))*
Lifelines	*Lifelines(l)=||line :l•[set(line)]PrefixComposition(line)* *alphaLifelines(l)=U{line :l•set(line)}*
Messages	*Messages(m)= ||(t,id,from,to,n) :m•[Message(t,id,from,to,n)]* *alphaMessages(m)=U{(t,id,from,to,n) :m•alphaMessage(t,id,from,to,n)*
Enforce	*Enforce(m)=□(msg,m,i,snd,from,to,n) :Mα•msg.m.i.snd.from.to.n→msg.m.i.rcv.from.to.n→Enforce(m)□skip*

Lors de l'exécution d'une interaction, tous les opérandes utilisent des sémantiques faibles de séquençage « *Seq* » sur leur contenu sauf l'opérateur « *Strict* ». La spécification UML (UML, 2017) définit la valeur par défaut de « *Seq* » comme suit :

• L'ordre des messages dans chaque opérande est maintenu dans le résultat.

• Les messages sur les différentes lignes de vie des différents opérandes peuvent venir dans un ordre quelconque.

• Les messages sur la même ligne de vie de différents opérandes sont ordonnés de telle sorte qu'un message du premier opérande vient avant celui du second opérande.

Le Tableau 11 décrit la formalisation des opérateurs d'interaction, avec la prise en considération de la sémantique décrite ci-dessus.

Tableau 11. Formalisation des opérateurs d'interaction

Type de l'opérateur d'intéraction	Code CSP généré
Seq	*Seq(l,m)= Lifelines(l)[Lα ‖Mα] Messages(m)*
Strict	*Strict(l,m)=(Lifelines(l)[Lα ‖Mα] Messages(m)) [‖Mα‖]Enforce(m)*
Par	*Par(l1,m1,l2,m2)=Seq(l1,m1) ||| Seq(l2,m2)*
Alt	*Alt(l1,m1,g1,l2,m2,g2,l3,m3)=if (g1∧g2)then Seq(l1,m1) ⊓ Seq(l2,m2)* *Else if g1 then Seq(l1,m1)Else if g2 then Seq(l2,m2)Else Seq(l3,m3)*
Opt	*Opt(l,m,g)=if(g) then Seq(l,m) Else skip*
Break	*Break(lpre,mpre,l,m,g,lpost,mpost)=Seq(lpre,mpre);(if g then Seq(l,m)* *Else Seq(lpost,mpost))*
Loop	*Loop(l,m,e)=if(e≥1)then (Seq(l,m) ;Loop(l,m,e-1))Else skip*
Ignore	*Ignore(L,M,ignore)=Seq(l,m) ||| Run(ignore)* *Run(E)=□e :E•e→Run(E)*
	Exemple : StateMachine\ignore ⊑T Seq(lifelines,messages)
Consider	spécifie un ensemble de messages qui doivent être considérés comme faisant partie de ce fragment combiné ; tous les autres messages sont ignorés.

Assert	Déclare que le fragment d'interation modélise les seules continuations valides.toute autre éventualité est considérée comme invalide. La relation de raffinement soit valable dans les deux sens.

3.5.2 Formalisation d'UML 2.0 STM

L'approche de transformation d'UML 2.0 STM en CSP, proposée dans (Ng & Butler, 2003), consiste à transformer chaque état en un processus et chaque événement UML ou action en un événement CSP. La Figure 20 résume les correspondances entre les deux langages cource et cible.

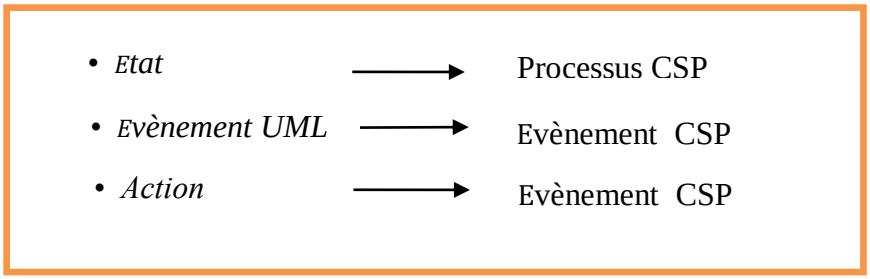
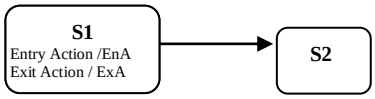
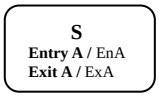
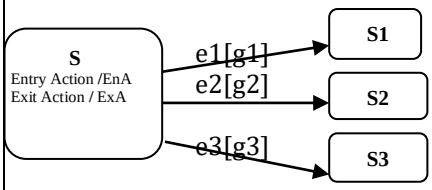
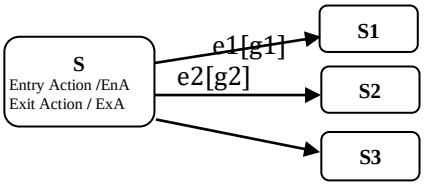
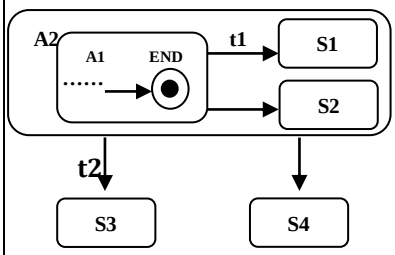
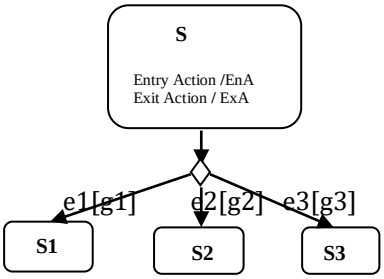


Figure 20. L'approche de transformation d' UML 2.0 STM en CSP

Le Tableau 12 explore la description des processus CSP générés pour chaque composant du du diagramme UML 2.0 STM.

Tableau 12. Formalisation d'UML 2.0 STM en CSP

UML 2.0 STM		CSP
Etat initial	START → S	*START = S*
Etat final	END ***Condition :*** END appartient à un seul état.	*END = SKIP*
Evénement explicite	S1 (**Entry A** / EnA, **Exit A** / ExA) — event[guard] /a1,a2..an → S2 Entry Action / EnA Exit Action / ExA	*S1 = EnA → guard & event → ExA → a1 → a2 →… → an → S2*

Evénement implicite		*S1 = EnA → ExA → S2*
Etat simple	 ***Condition :*** Etat sans transition sortante.	*S = STOP*
Choix externe	 ***Condition :*** Transitions avec événements explicites seulement.	*S = EnA → ((g1 & e1 → ExA → S1) □ (g2 & e2 → ExA → S2)* *□ (g3 & e3 → ExA → S3)*
	 ***Condition :*** Transitions avec événements explicites et implicites.	*S = EnA → ((g1 & e1 → ExA → S1) □ (g2 & e2 → ExA → S2)* *▷ (ExA → S3)*
		*END = (t1 → S1) □ (t2 →▷ S3)→ S2*
Choix interne		*S = EnA → ((g1 & e1 → ExA → S1) ⊓ (g2 & e2 → ExA → S2)* *⊓ (g3 & e3 → ExA → S3)*

La formalisation décrite ci-dessus n'est pas exhaustive, elle ne couvre pas les régions, ni les pseudo-états de jointure (Join), de bifurcation (Fork) et de jonction (Junction). La section suivante présente notre contribution dans ce cadre.

3.5.2.1 Formalisation proposée

- **La jointure :** Ce type de pseudo-état sert de sommet cible commun pour deux ou plusieurs transitions provenant de sommets dans différentes régions orthogonales. Les transitions se terminant sur un pseudo-état de jointure ne peuvent pas avoir de garde ou de déclenchement. Les pseudo-états de jointure remplissent une fonction de synchronisation, grâce à laquelle toutes les transitions entrantes doivent être terminées avant que l'exécution puisse se poursuivre via une transition sortante (UML, 2017).

  Afin de formaliser ce pseudo-état, nous adoptons l'idée de Bisztray et ses collaborateurs (Bisztray et al., 2007) pour la formalisation du nœud de jointure dans un diagramme d'activité en CSP.

  Tout d'abord, Il est nécéssaire de discuter quelques observations avant de commencer la formalisation. Si dans un diagramme d'états-transitions les noms des événements sont uniques, l'intersection des alphabets des processus est vide. Ceci est en partie intentionnel, car les processus ne resteront ainsi pas bloqués en attendant un autre processus aléatoire qui aurait accidentellement des événements avec des noms similaires. En revanche, nous avons besoin d'un point de synchronisation pour mettre en œuvre la jointure des processus. On ajoute donc un événement « *Processjoin »* à l'alphabet de tous les processus participants. Comme les événements qui sont dans les alphabets de tous les processus participants nécessitent une participation simultanée, ce fait est utilisé pour rejoindre des processus concurrents en les bloquant jusqu'à ce qu'ils puissent exécuter l'événement de synchronisation. La Figure 21 illustre l'utilisation du pseudo-état *Join.*

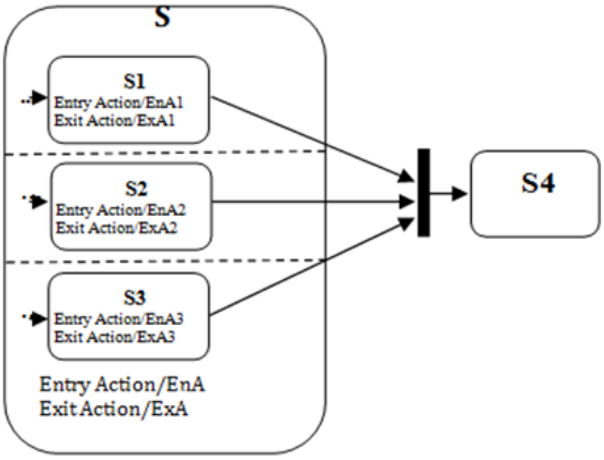


Figure 21. Pseudo-état *Join*

Dans le mappage concret, la première transition rencontrant le pseudo-état « *Join* » est choisie pour effectuer le processus de continuation *(S4)* après l'exécution de l'action de sortie *« ExitAction »* , tandis que les autres transitions se terminent par un SKIP. Le code CSP correspondant à l'exemple présenté ci-dessus est le suivant :

$$S1 = EnA1 \rightarrow ExA1 \rightarrow Processjoin \rightarrow ExA \rightarrow S4$$
$$S2 = EnA2 \rightarrow ExA2 \rightarrow Processjoin \rightarrow SKIP$$
$$S3 = EnA3 \rightarrow ExA3 \rightarrow Processjoin \rightarrow SKIP$$

- **Les régions :** Une région désigne un fragment de comportement qui peut s'exécuter simultanément avec ses régions orthogonales. Elle devient active lorsque son État propriétaire est entré. Chaque région possède un ensemble de sommets et de transitions, qui déterminent le flux comportemental au sein de cette région. Il peut avoir son propre pseudo-état initial ainsi que son propre état final. plusieurs régions orthogonales peuvent être entré en parallèle via des transitions provenant du même pseudo-état fork (UML, 2017).

  Un état composite avec plusieurs régions orthogonales correspond à l'opérateur binaire de la concurrence (||). On peut décrire trois processus qui s'éxécutent

simultanément par : P1 || (P2 || P3) ou encore par (P1 || P2) || P3. Les différentes correspondances sont équivalentes. La Figure 22 présente un état composite avec plusieurs régions orthogonales.

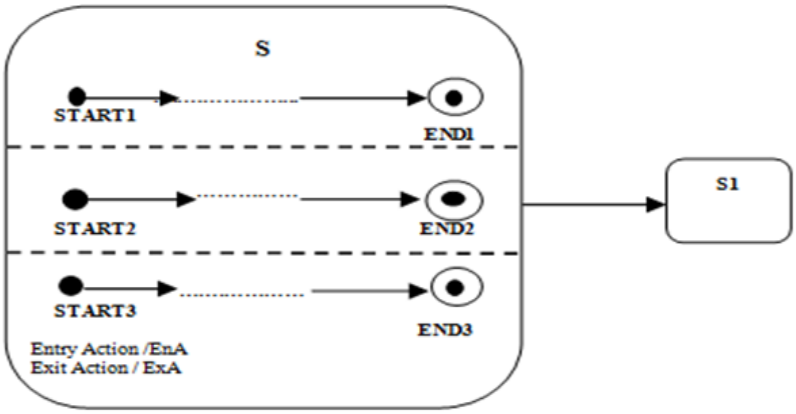


Figure 22. Régions orthogonales

Le code CSP correspondant au cas présenté dans la Figure 22 est le suivant :

S= EnA → (START1 || (START2 || START3))
END1 = Processjoin → ExA → S1
END2 = Processjoin → SKIP
END3 = Processjoin → SKIP

- **La bifurcation (Fork) :** Les pseudo-états fork servent à diviser une transition entrante en deux ou plusieurs transitions se terminant les sommets dans les régions orthogonales d'un état composite. Les transitions sortantes d'un Pseudo-état fork ne peuvent pas avoir une garde ou un déclenchement (UML, 2017). Un pseudo-état *« Fork »* correspond à l'opérateur binaire de la concurrence (||). La Figure 23 illustre l'utilisation du pseudo-état *Fork.*

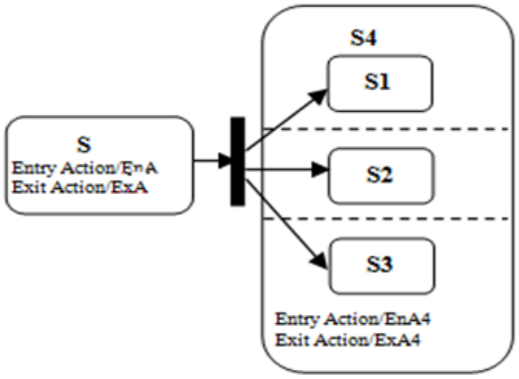


Figure 23. Pseudo-état *Fork*

Le code CSP correspondant à l'exemple présenté dans la Figure 23 est le suivant :

$$S = EnA \rightarrow ExA \rightarrow EnA4 \rightarrow (S1 \,||\, (\, S2 \,||\, S3))$$

- **La jonction (Junction) :** Ce type de pseudo-état est utilisé pour connecter plusieurs transitions en chemins composés entre états. Par exemple, un pseudo-état de jonction peut être utilisé pour fusionner plusieurs transitions entrantes en une seule transition sortante représentant un chemin de continuation partagé. Ou bien, il peut être utilisé pour diviser une transition entrante en plusieurs transitions sortantes avec différentes contraintes de garde (diviser 1 en n, fusionner n en 1 ou connecter n à m). Les contraintes de garde sont évaluées avant qu'une transition composée contenant ce pseudo-état ne soit exécutée, c'est pourquoi on parle de branchement conditionnel statique (UML, 2017). Le pseudo-état de jonction correspond au choix externe entre tous les états sortants pour chaque état entrant du point de jonction. La Figure 24 illustre l'utilisation du pseudo-état *Junction.*

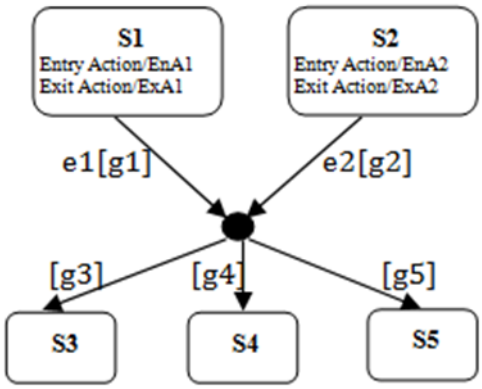


Figure 24. Le pseudo-état Junction

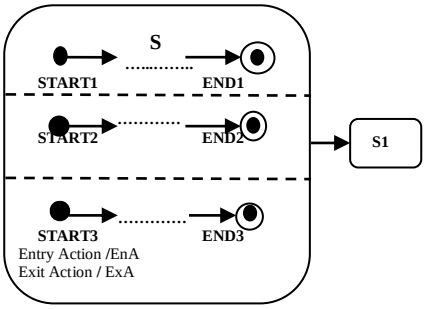
Dans ce cas, le code CSP généré est présenté ci-dessous.

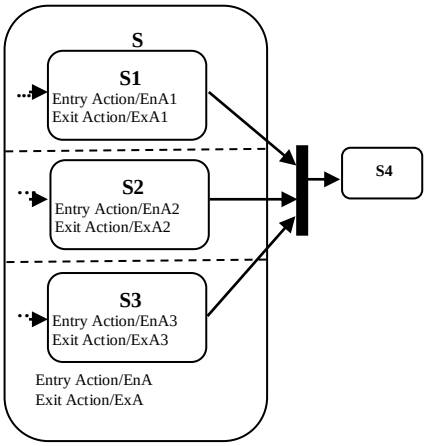
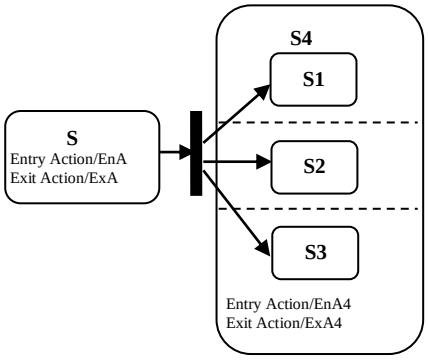
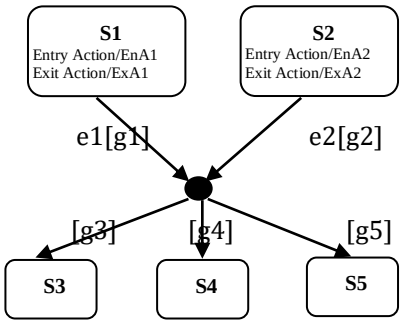
$$S1 = EnA1 \rightarrow g1 \;\&\; e1 \rightarrow (g3 \rightarrow ExA1 \rightarrow S3) \;\square\; (g4 \rightarrow ExA1 \rightarrow S4) \;\square\; (g5 \rightarrow ExA1 \rightarrow S5)$$

$$S2 = EnA2 \rightarrow g2 \;\&\; e2 \rightarrow (g3 \rightarrow ExA2 \rightarrow S3) \;\square\; (g4 \rightarrow ExA2 \rightarrow S4) \;\square\; (g5 \rightarrow ExA2 \rightarrow S5)$$

La formalisation des quatre composants est résumée dans le Tableau 13.

Tableau 13. La formalisations proposée

UML 2.0 STM		CSP
Région		$S = EnA \rightarrow (START1 \;\|\|\; (START2 \;\|\|\; START3))$ $END1 = Processjoin \rightarrow ExA \rightarrow S1$ $END2 = Processjoin \rightarrow SKIP$ $END3 = Processjoin \rightarrow SKIP$

Join		*S1 = EnA1 → ExA1 → Processjoin → ExA → S4* *S2 = EnA2 → ExA2 → Processjoin → SKIP* *S3 = EnA3 → ExA3 → Processjoin → SKIP*
Fork		*S = EnA → ExA →EnA4 → (S1 || (S2 || S3))*
Jonction		*S1 = EnA1 → g1 & e1→ (g3 → ExA1 → S3)* *□ (g4 → ExA1 → S4)* *□ (g5 → ExA1 → S5)* *S2 = EnA2 → g2 & e2→ (g3 → ExA2 → S3)* *□ (g4 → ExA2 → S4)* *□ (g5 → ExA2 → S5)*

3.5.3 Formalisation d'UML 2.0 CD

L'approche de transformation des diagrammes de classe en CSP, proposée dans (Ng & Butler, 2002), est résumée dans la Figure 25.

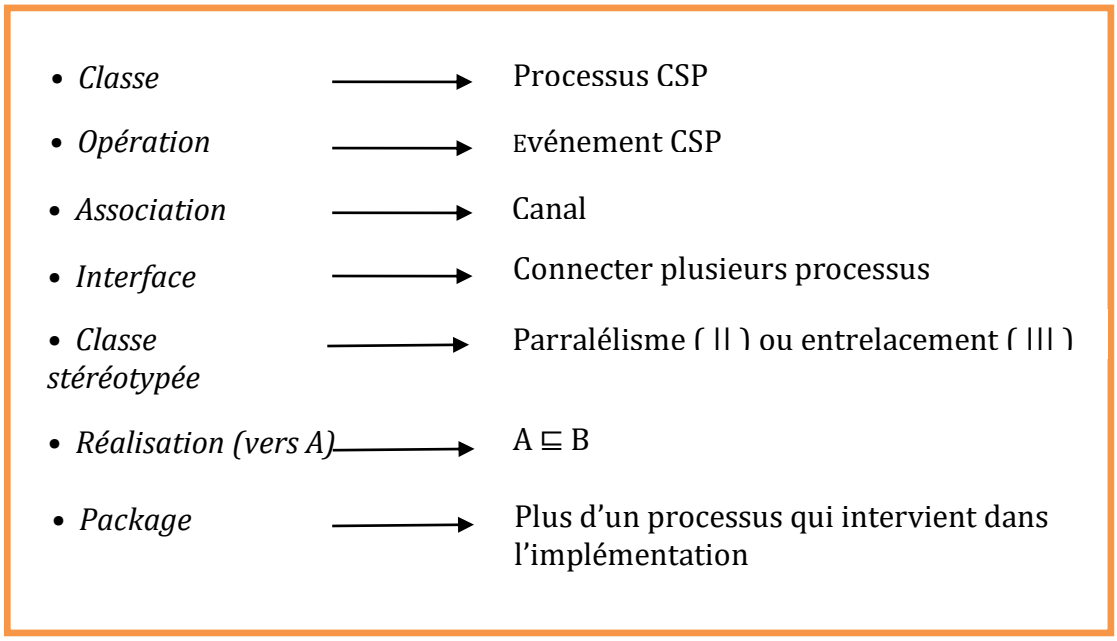
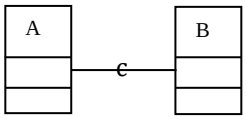
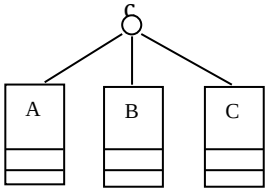
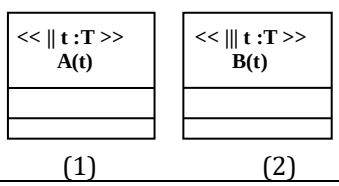
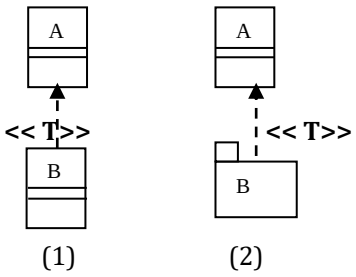


Figure 25. Approche de transformation d'UML 2.0 CD en CSP

La description des processus CSP générés pour chaque composant du diagramme UML 2.0 CD est présentée dans le Tableau 14.

Tableau 14. Formalisation des diagrammes UML 2.0 CD en CSP

UML 2.0 CD		CSP
Association		*System = A [|{|c|}|] B*
Interface		*System = A [|{|c|}|] B* *System1 = System [|{|c|}|] C*
Classe stéréotypée		*(1) As = || t :T @ A(t)* *(2) As = ||| t :T @ B(t)*

Réalisation		$A \sqsubseteq_T B \backslash \{événements\ cachés\}$ T : Trace F : Failure FD : Failure Divergence

3.6 CONCLUSION

L' approche de formalisation d'UML 2.0 en CSP offre des avantages considérables pour la vérification formelle des systèmes logiciels. Elle permet de garantir la cohérence et la correction des modèles tout en offrant une clarté et une compréhensibilité des spécifications semi-formelles. Cependant, cette approche nécessite une expertise technique pour être mise en œuvre efficacement. Dans le prochain chapitre, nous allons présenter en détail l'implémentation de l'approche proposée en utilisant l'outil AToM³.

CHAPITRE

4

L'APPROCHE INTEGREE UML 2.0/CSP PROPOSEE

4.1 INTRODUCTION

Dans ce chapitre, nous présentons une approche intégrée UML 2.0/CSP pour la modélisation et la vérification des systèmes distibués. L'approche proposée est une approche automatique basée sur la transformation de graphes en utilisant l'outil AToM³. Les formalisations des diagrammes UML 2.0 en CSP adoptées pour la réalisation de cette approche sont celles présentées dans le chapitre précédent. Les principales étapes de réalisation de notre approche comprennent la méta-modélisation des diagrammes UML 2.0, la transformation automatique de ces derniers en CSP et la vérification des spécifications CSP générées à l'aide du model-checker FDR4.

4.2 APPROCHE INTEGREE UML 2.0 /CSP PROPOSEE

L'approche proposée se base sur l'utilisation combinée de la méta-modélisation et de la transformation de graphes. Pour l'implémenter, nous avons choisi l'outil de modélisation et de méta-modélisation multi-formalismes AToM^3. Nous avons proposé trois méta-modèles pour les diagrammes de séquence, les diagrammes d'états-transitions et les diagrammes de classe. Ensuite, nous avons généré trois outils visuels qui permettent la modélisation des trois diagrammes précédents. Après, nous avons proposé trois grammaires de graphes afin de transformer automatiquement les diagrammes modélisés vers le CSP. Le code généré est en CSP_M, La machine dialecte lisible du CSP, qui est le type d'entrée accepté par l'outil FDR4. Ce dernier vérifie chaque propriété spécifiée si elle est satisfaite ou non. La Figure 26 montre l'architecture de l'approche proposée.

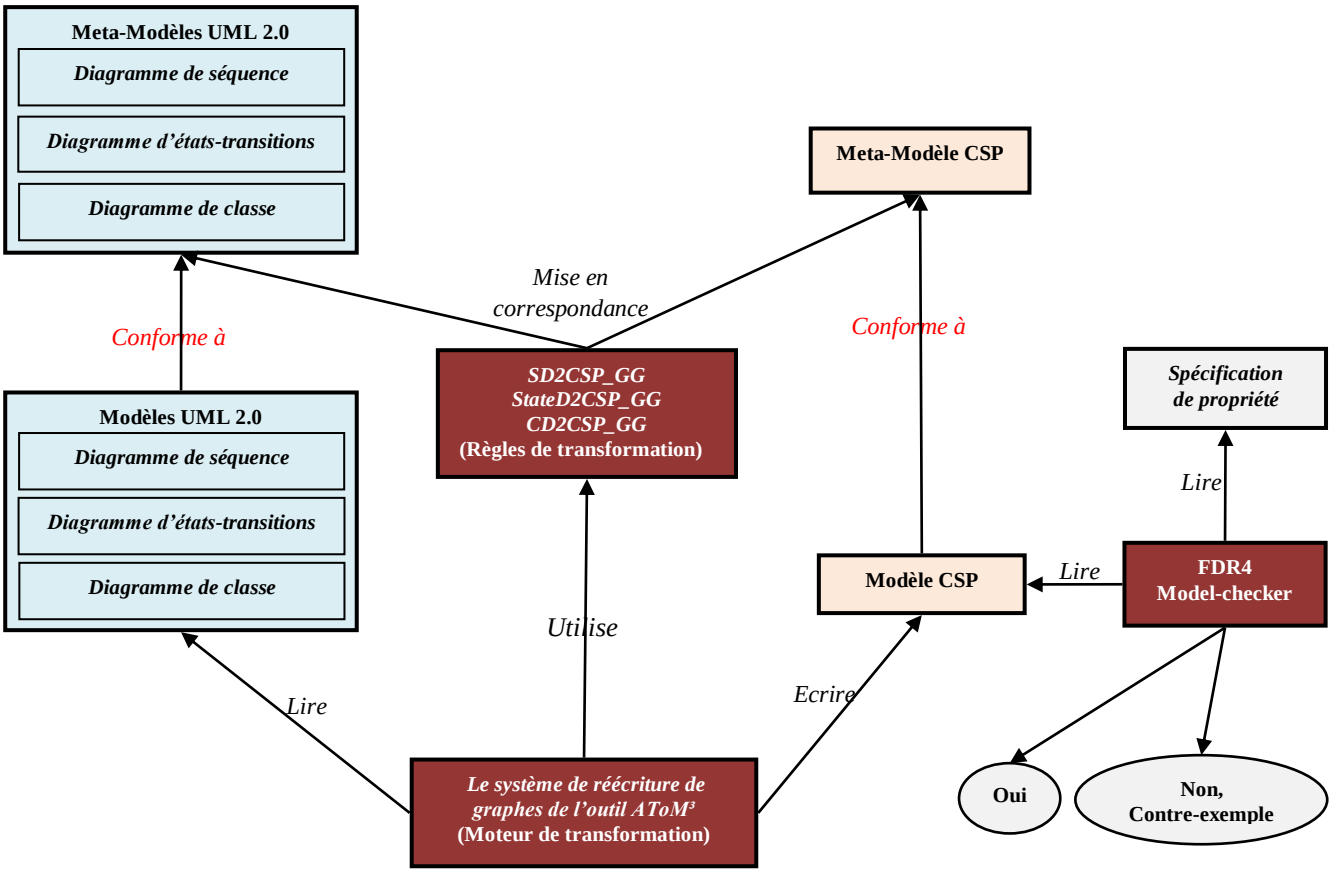


Figure 26. L'architecture de l'approche proposée

Les étapes de réalisation de notre approche pruvent être résumées comme suit :

1. Méta-modélisation des diagrammes UML 2.0

a- Définition de trois méta-modèles :

- SDiagram_META : le méta-modèle d'UML 2.0 SD.
- StateDiagram_META : le méta-modèle d'UML 2.0 STM.
- ClassDiagram_META : le méta-modèle d'UML 2.0 CD.

b- Génération des outils visuels correspondants aux méta-modèles proposés.

2. Transformation des diagrammes UML 2.0 en définissant trois grammaires de graphes pour transformer les diagrammes UML 2.0 en CSPM. Les grammaires développées sont les suivantes :

   - SD2CSP_GG : pour transformer UML 2.0 SD.
   - StateD2CSP_GG : pour transformer UML 2.0 STM.
   - CD2CSP_GG : pour transformer UML 2.0 CD.

3. Vérification des spécifications CSPM générées

a- Définition des assertions CSPM.

b- L'utilisation du model checker FDR4 pour vérifier le code CSPM généré en utilisant les assertions CSPM. Ensuite, la génération du résultat de la vérification (passée si la propriété est satisfaite, échouée avec un contre-exemple sinon).

4.2.1 Méta-modélisation des diagrammes UML 2.0

Dans la méta-modélisation, nous avons définit les structure des diagrammes UML 2.0 considérés: UML 2.0 SD, UML 2.0 STM et UML 2.0 CD. Chaque méta-modèle possède une syntaxe abstraite et une syntaxe concrète. Nous avons utilisé l'outil AToM^3, qui supporte la création des méta-modèles et la génération d'outils visuels, ainsi que son méta-formalisme « *CD-ClassDiagramsV3 »* pour créer les méta-modèles correspondants. Des contraintes écrites en code Python sont utilsées pour ajouter de nouvelles sémantiques aus éléments du méta-modèle.

4.2.1.1 Méta-Modélisation d' UML 2.0 SD

La Figure 27 montre le méta-modèle des diagrammes de séquence UML 2.0 « *SDiagram_META »*, proposé dans (Hamrouche et al., 2022). Ce méta-modèle se compose de quatre classes liées par six associations. Les classes *«Interaction», « LifeLine », « CombinedFragment »* et *« InteractionOperand »* représentent respectivement une interaction, une ligne de vie, un fragment combiné et un opérande d'interaction. Les associations *« AMessage »*, *« SMessage »* et *«Acknowledgement »* représentent les trois types de messages échangés entre les lignes de vie, qui sont : message asynchrone,

message synchrone et message de retour. Les associations *« IContains » et « CFContains »* sont des associations de composition. L'association *« IOContains »* liant les classe « *InteractionOperand* » et « *CombinedFragment* » est une associations de composition (la suppression d'un objet de la classe composite, *« InteractionOperand »,* entraine aotomatiquement la suppression des objets de la classe composant associés), alors que L'association *« IOContains »* liant les classe « *InteractionOperand* » et « *LifeLine*» est une associations d'agrégation (la suppression d'un objet agrégat, n'entraine pas la suppression des objets agrégés). Les contraintes et les actions sont exprimées en code Python. La représentation visuelle de chaque classe et de chaque association est spécifiée dans cette étape.

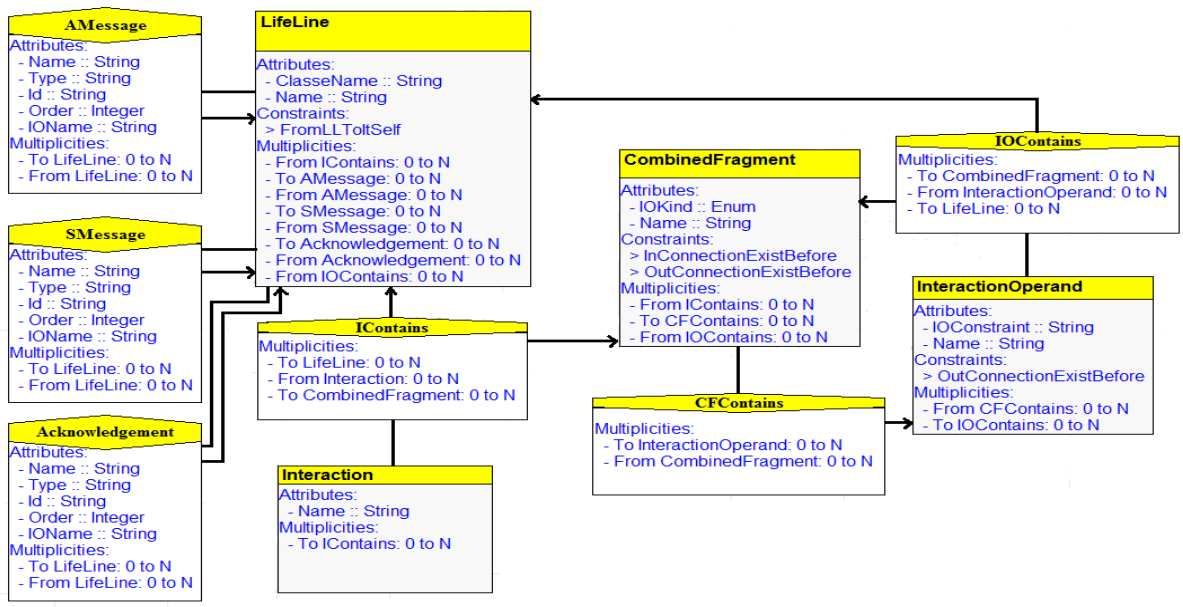


Figure 27. Méta-Modèle d'UML 2.0 SD

La description détaillée des composants du méta-modèle *« SDiagram_META »* est comme suit:

- **La classe *« Interaction » :*** Cette classe représente un diagramme de séquence UML 2.0. Elle est graphiquement visualisée par un rectangle qui dispose d'une étiquette *sd* en haut à gauche, suivi d'un attribut *Name* de type *String* qui signifie le nom du diagramme de séquence.

- **La classe *« LifeLine »* :** Cette classe représente un objet ou un acteur qui participe à une interaction (une ligne de vie). Elle est graphiquement représentée par un trait pointillé à la verticale de l'objet, qui est représenté par un petit rectangle bleu contenant deux attributs de type *String* : *Name* et *ClassName.*
- **La classe *« CombinedFragment »*** : Cette classe représente un fragment d'interaction combiné. Elle est graphiquement représentée par un rectangle associé à une étiquette dans le coin supérieur gauche, qui représente un opérateur d'interaction, permettant de décrire les modes d'exécution des messages dans le fragment combiné. Cette classe possède deux attributs : *Name* de type *String* et *IOKind* de type *Enum,* qui représentent respectivement le nom du fragment combiné et le type de l'opérateur d'interaction. La Figure 28 montre les opérateurs d'interaction modélisés dans notre approche.

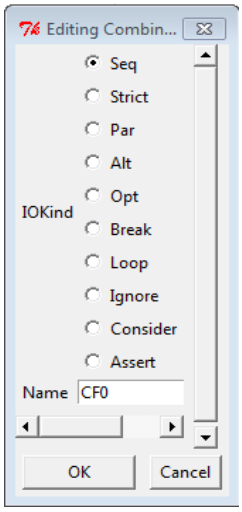


Figure 28. Les opérateurs d'interaction

La classe *« CombinedFragment »* possède deux contraintes : *InConnectionExistBefore* et *OutConnectionExistBefore,* afin d'éviter la création d'une association (invisible) entrante ou sortante déjà créée.

- **La classe *« InteractionOperand »*** : Cette classe représente un opérande d'un opérateur d'interaction. Elle est graphiquement représentée par un rectangle pointillé contenant deux attributs : *Name* et *IOConstraint* de type *String. C*e dernier attribut désigne une condition de choix des opérandes exprimée par une expression booléenne entre crochets. La classe *« InteractionOperand »* possède

une contrainte, *OutConnectionExistBefore,* visant à éviter la création d'une association (invisible) sortante déjà existante.

- **L'association *« AMessage »*** : Cette association modélise un message asynchrone entre deux lignes de vie, pouvant être un signal ou un appel de méthode. Elle est graphiquement représentée par une simple flèche noire allant de ligne de vie de l'objet émetteur à la ligne de vie de l'objet récepteur. Elle contient les attributs *Name, Type, Id* et *IOName* de type *String,* représentant respectivement le nom, le type, l'identifiant et le nom de l'opérande d'interaction. De plus, elle possède aussi un attribut *Order* de type *Integer* qui représente l'ordre du message dans le diagramme de séquence. L'association *« AMessage »* est étiquetée uniquement par les deux attributs *Id* et *Order*. L'association *«AMessage »* permet d'associer à chaque instance de la classe *« LifeLine »* une seule instance d'elle-même.
- **L'association *« SMessage »*** : Cette association modélise un message synchrone entre deux lignes de vie. La réception de ce message provoque le lancement d'une méthode du récepteur. L'émetteur reste bloqué pendant l'exécution de la méthode et attend sa fin pour pouvoir lancer un nouveau message. Graphiquement, elle est représentée par une flèche noire avec un triangle plein à son extrémité. Elle contient les mêmes attributs que l'association *«AMessage »* et étiquetée par les deux attributs *Id* et *Order*. L'association *«SMessage »* permet d'associer à chaque instance de la classe *« LifeLine »* une seule instance d'elle-même.
- **L'association *« Acknowledgement »*** : Cette association modélise un message de retour qui renvoie des valeurs à la fin de l'activation d'une méthode. Graphiquement, elle est représentée par une simple flèche bleue. Elle partage les mêmes attributs que l'association *« AMessage »* et est étiquetée par les deux attributs *Id* et *Order*. L'association *« Acknowledgement »* permet d'associer à chaque instance de la classe *« LifeLine »* une seule instance d'elle-même.
- **L'association *« IContains » :*** C'est une association entre une interaction et une ligne de vie ou un fragment combiné. Elle exprime la notion d'hiérarchie entre les classes reliées (les lignes de vie et les fragments combinés sont à l'intérieur d'une

interaction). Elle n'a pas une apparence graphique d'où la nécessité d'ajouter une contrainte au niveau des classes reliées pour éviter la création d'une association existante. Elle associe à chaque instance de la classe *« Interaction »* une ou plusieurs instances de la classe *« CombinedFragment »* ainsi que deux ou plusieurs instances de la classe *« LifeLine ».*

- **L'association *« IOContains »* :** C'est une association entre un opérande d'interaction et une ligne de vie ou un fragment combiné. Elle exprime la notion d'hiérarchie entre les classes reliées (dans le cas de fragments combinés imbriqués, l'opérande d'interaction contient un ou plusieurs fragments combinés). L'association *«IOContains»* n'a pas une apparence graphique, elle associe à chaque instance de la classe *« InteractionOperand »* une ou plusieurs instances de la classe *« CombinedFragment »* et deux ou plusieurs instances de la classe *« LifeLine ».*
- **L'association *« CFContains »* :** C'est une association entre un fragment combiné et un opérande d'interaction. Elle exprime la notion d'hiérarchie entre les deux classes reliées (les opérandes d'interactions sont à l'intérieur d'un fragment combiné). L'association *«CFContains»* n'a pas une apparence graphique, elle associe à chaque instance de la classe *« CombinedFragment »* une ou plusieurs instances de la classe *« InteractionOperand ».*

Le méta-modèle proposé «*SDiagram_META*» nous a permis de générer l'outil graphique de modélisation des diagrammes de séquence UML 2.0 présenté dans laFigure 29.

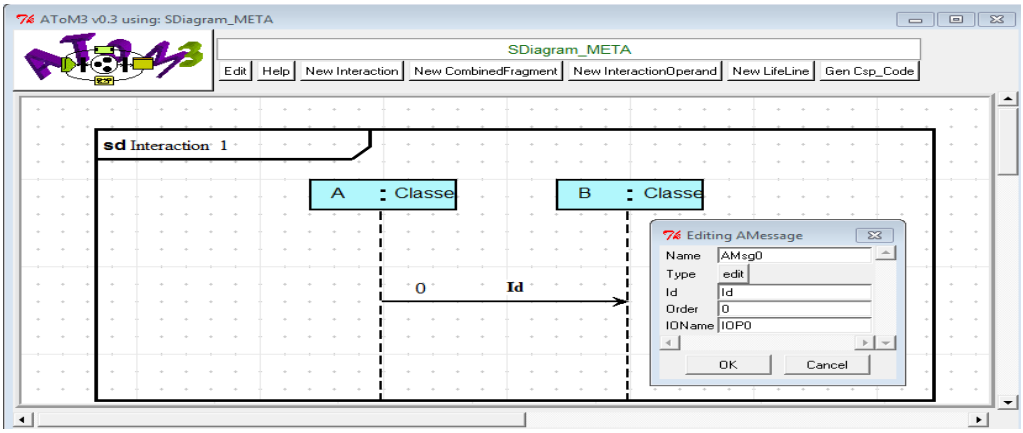


Figure 29. L'outil généré de modélisation d'UML 2.0 SD

4.2.1.2 Méta- Modélisation d'UML 2.0 STM

Pour méta-modéliser les diagrammes d'états-transitions UML 2.0 dans AToM^3, nous avons proposé le méta-modèle *« StateDiagram_META »,* illustré par la Figure 30*.* Celui-ci est composé de dix classes, une association d'héritage, six associations pour spécifier la hiérarchie et douze associations pour modéliser les différentes transitions dans un UML 2.0 STM.

Les classes *« State-Chart »*, *« Simple-State »*, *« Composite-State »*, *« Region »*, *« Initial-State »* et *« Final-State »* représentent respectivement un diagramme d'états-transitions, un état simple, un état composite, une région, un état initial et un état final. Les classes *« PState-Fork »*, *« PState-Join »*, *« PState-Choice »* et *« PState-Junction »* représentent respectivement des pseudo-états de bifurcation, de jointure, de decision et de jonction.

Les associations du méta-modèle sont: *« has_IS »*, *« Rhas_IS », « has_Region » , « has_SS » , « has_CS » , « has_FS » , « IS2SS »*, *« IS2PSFork »*, *« SS2SS »* , *« SS2FS »* , *« SS2PSFork »* , *« PSFork2SS »* , *« SS2PSJoin »* , *« PSJoin2SS »* , *« SS2PSChoice »* , *« PSChoice2SS »* , *« SS2PSJunction »* , *« PSJunction2SS ».*

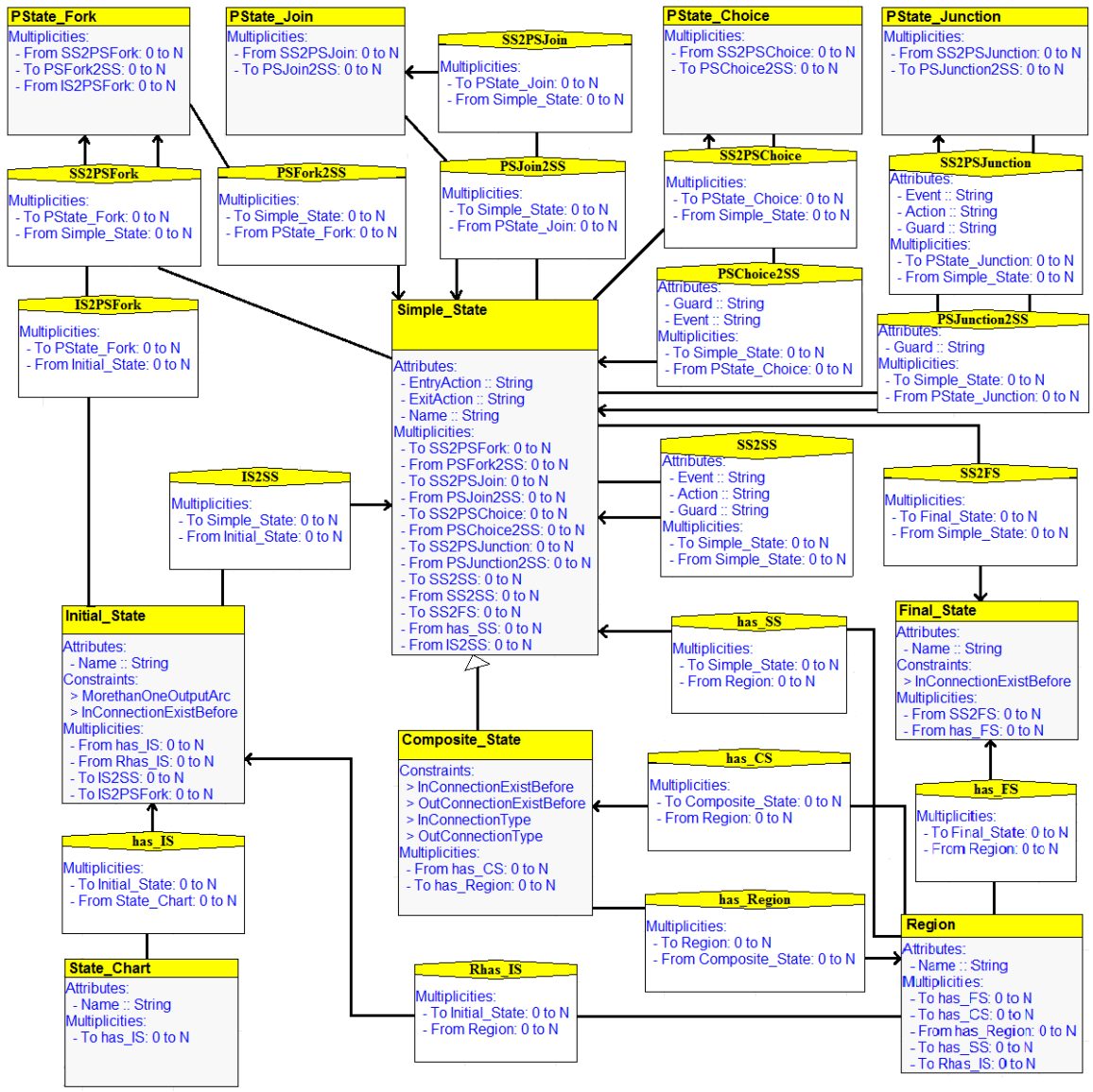


Figure 30. Méta-modèle d'UML 2.0 STM

La description détaillée des composants du méta-modèle *« StateDiagram_META »* est comme suit:

- **La classe *« State-Chart »*** : Cette classe représente un diagramme d'états-transitions UML 2.0. Elle est graphiquement représentée par un rectangle qui dispose d'une étiquette *stm* en haut à gauche suivi d'un attribut *Name* de type *String* qui signifie le nom du diagramme d'états-transitions.
- **La classe *« Simple-State »*** : Cette classe représente un état simple dans un diagramme d'états-transitions. Elle est graphiquement représentée par un

rectangle aux coins arrondis et possède trois attributs de type *String* : *Name*, *EntryAction* et *ExitAction.*

- **La classe *« Composite-State »* :** Cette classe est une sous-classe de la classe *« Simple-State ».* Elle modélise un état composite qui intègre des sous-états simples et / ou composites dans un diagramme d'états-transitions. Graphiquement, elle est représentée par un rectangle aux coins arrondis et possède trois attributs de type *String* : *Name*, *EntryAction* et *ExitAction.*
- **La classe *« Region »*** : Cette classe modélise une région représentant un flot d'exécution dans un diagramme d'états-transitions. Graphiquement, elle est représentée par un rectangle en pointillé et possède un seul attribut *Name* de type *String*.
- **La classe *« Initial-State »*** : Cette classe représente un état initial dans un diagramme d'états-transitions, Graphiquement, elle est représentée par un petit cercle plein. Elle possède un attribut *Name* de type *String* et deux contraintes *MoreThanOneOutputArc* et *InConnectionExistBefore*, limitant respectivement le nombre des arcs sortants de cette classe à un et évitant la création d'une association (invisible) sortante déjà existante.
- **La classe *« Final-State »*** : Cette classe représente un état final dans un diagramme d'états-transitions. Graphiquement, elle est représentée par un cercle vide contenant un petit cercle plein. Elle possède un attribut *Name* de type *String* et une contrainte *InConnectionExistBefore* pour éviter la création d'une association entrante déjà créée.
- **La classe *« PState-Fork »*** : Cette classe modélise un pseudo-état de bifurcation dans un diagramme d'états-transitions. Elle permet de représenter des flots d'exécution parallèles. Graphiquement, elle est représentée par une barre épaisse et elle ne possède aucun attribut.
- **La classe *« PState-Join »*** : Cette classe modélise un pseudo-état de jointure dans un diagramme d'états-transitions. Elle est graphiquement représentée par une barre épaisse et ne possède aucun attribut.

- **La classe *« PState-Choice »*** : Cette classe modélise un pseudo-état de décision dans un diagramme d'états-transitions. elle est graphiquement représentée par un losange et ne possède aucun attribut.
- **La classe *« PState-Junction »*** : Cette classe modélise un pseudo-état de jonction dans un diagramme d'états-transitions. Graphiquement, elle est représentée par un cercle plein, et son utilisation améliore la lisibilité d'un diagramme d'états-transitions. Elle ne possède aucun attribut.
- **L'association *« has_IS »***: Cette association exprime la notion d'hiérarchie entre un diagramme d'états transitions et son état initial. Elle n'a pas une apparence graphique et associe à chaque instance de la classe *« State_Chart »* une seule instance de la classe *«Initial_State ».*
- **L'association *« Rhas_IS »***: Cette association exprime la notion d'hiérarchie entre une région et son état initial. Elle n'a pas une apparence graphique et associe à chaque instance de la classe *« Region »* une seule instance de la classe *«Initial_State ».*
- **L'association *« has_Region »*:** Cette association exprime la notion d'hiérarchie entre un état composite et ses régions. Elle n'a pas une apparence graphique et associe à chaque instance de la classe *« Composite_State »* une ou plusieurs instances de la classe *«Region ».*
- **L'association *« has_SS »***: Cette association exprime la notion d'hiérarchie entre une région et ses états simples. Elle n'a pas une apparence graphique et associe à chaque instance de la classe *« Region »* une ou plusieurs instances de la classe *«Simple_State ».*
- **L'association *« has_CS »*:** Cette association exprime la notion d'hiérarchie entre une région et ses états composites. Elle n'a pas une apparence graphique et associe à chaque instance de la classe *« Region »* une ou plusieurs instances de la classe *«Composite_State ».*
- **L'association *« has_FS »***: Cette association exprime la notion d'hiérarchie entre une région et son état final. Elle n'a pas une apparence graphique et associe à chaque instance de la classe *« Region »* une seule instance de la classe *«Final_State ».*

- **L'association *« IS2SS »*:** Cette association modélise une transition entre un état initial et un état simple. Elle est graphiquement représentée par une simple flèche noire ne portant aucun attribut et associe à chaque instance de la classe *« Initial_State »* une seule instance de la classe *« Simple_State »*.
- **L'association *« IS2PSFork »***: Cette association modélise une transition entre un état initial et un pseudo-état de bifurcation. Elle est graphiquement représentée par une simple flèche noire ne portant aucun attribut. Elle associe à chaque instance de la classe *« Initial_State »* une seule instance de la classe *« PState_Fork »*.
- **L'association *« SS2SS »*:** Cette association modélise une transition entre deux états simples. Graphiquement, elle est représentée par une simple flèche noire. Elle contient les attributs *Event*, *Action* et *Guard* de type *String* qui représentent respectivement un événement, une action et une garde. L'association *« SS2SS »* porte tous les attributs et associe à chaque instance de la classe *« Simple-State »* une ou plusieurs instances d'elle-même.
- **L'association *« SS2FS »***: Cette association modélise une transition entre un état simple et un état final. Elle est graphiquement représentée par une simple flèche noire et ne portant aucun attribut. Cette association relie chaque instance de la classe *« Simple-State »* à une seule instance de la classe *« Final-State »*.
- **L'association *« SS2PSFork »***: Cette association modélise une transition entre un état simple et un pseudo-état de bifurcation. Elle est graphiquement représentée par une simple flèche noire ne portant aucun attribut. Cette association relie chaque instance de la classe *« Simple-State »* à une seule instance de la classe *« PState-Fork »*.
- **L'association *« PSFork2SS »*:** Cette association modélise une transition entre un pseudo-état de bifurcation et un état simple. Elle est graphiquement représentée par une simple flèche noire ne portant aucun attribut. Cette association relie chaque instance de la classe *« PState-Fork »* à deux ou plusieurs instances de la classe *« Simple-State »*.
- **L'association *« SS2PSJoin »***: Cette association modélise une transition entre un état simple et un pseudo-état de jointure. Elle est graphiquement représentée

par une simple flèche noire ne portant aucun attribut. Cette association relie chaque instance de la classe *« Simple-State »* à une seule instance de la classe *« PState-Join »,* et chaque instance de la classe *« PState-Join »* peut être associée à deux ou plusieurs classes *« Simple-State »*.

- **L'association *« PSJoin2SS »***: Cette association modélise une transition entre un pseudo-état de jointure et un état simple. Graphiquement , elle est représentée par une simple flèche noire ne portant aucun attribut. Cette association relie chaque instance de la classe *« PState-Join »* à une seule instance de la classe *« Simple-State »*.
- **L'association *« SS2PSChoice »*:** Cette association modélise une transition entre un état simple et un pseudo-état de décision. Graphiquement, elle est représentée par une simple flèche noire ne portant aucun attribut. Cette association relie chaque instance de la classe *« Simple-State »* à une seule instance de la classe *« PState-Choice »*.
- **L'association *« PSChoice2SS »*:** Cette association modélise une transition entre un pseudo-état de décision et un état simple. Graphiquement , elle est représentée par une simple flèche noire. Elle contient les attributs *Event* et *Guard* de type *String*. L'association *«PSChoice2SS »* porte ces deux attributs et associe à chaque instance de la classe *« PState-Choice »* deux ou plusieurs instances de la classe *« Simple-State »*.
- **L'association *« SS2PSJunction »***: Cette association modélise une transition entre un état simple et un pseudo-état de jonction. Graphiquement, elle est représentée par une simple flèche noire. Elle contient les attributs *Event*, *Action* et *Guard* de type *String.* L'association *« SS2PSJunction »* porte tous les attributs et associe à chaque instance de la classe *« Simple-State »* une seule instance de la classe *« PState-Junction »*, tandis que chaque instance de la classe *« PState-Junction »* peut être associée à une ou plusieurs classes *« Simple-State »*.
- **L'association *« PSJunction2SS »***: Cette association modélise une transition entre un pseudo-état de jonction et un état simple. Graphiquement, elle est représentée par une simple flèche noire portant l'attribut *Guard* de type *String.* Elle associe à

chaque instance de la classe *« PState-Junction »* une ou plusieurs instances de la classe *« Simple-State »*.

Le méta-modèle proposé, «*StateDiagramt_META»,* nous a permis de générer l'outil graphique de modélisation des diagrammes d'états-transitions UML 2.0, tel que présenté dans la Figure 31.

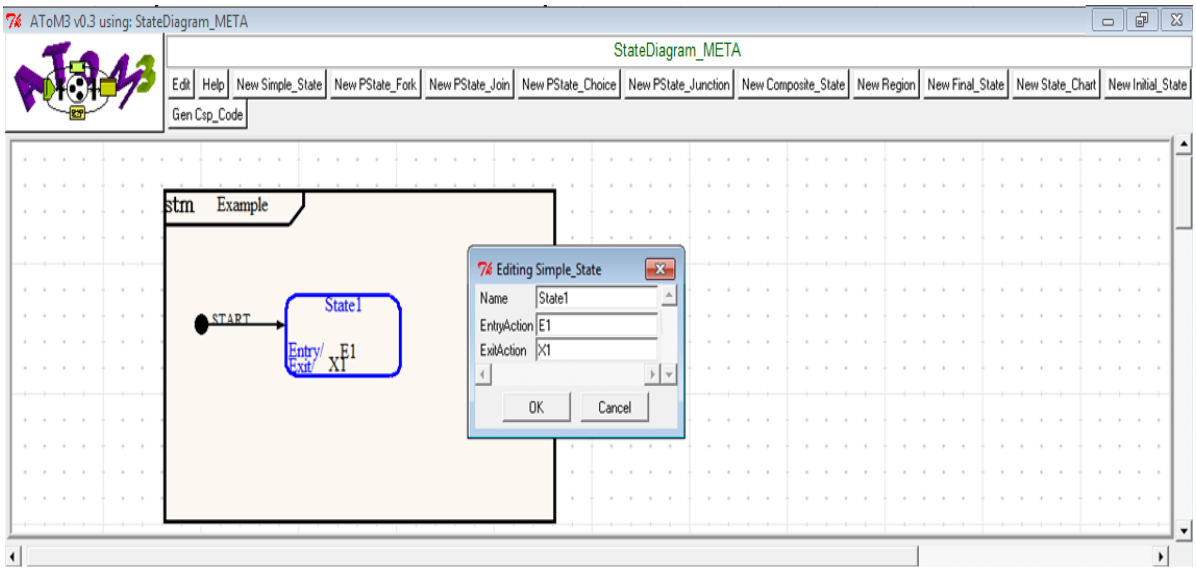


Figure 31. Outil de modélisation d'UML 2.0 STM

4.2.1.3 Méta- Modélisation d'UML 2.0 CD

Pour méta-modéliser les diagrammes de classes UML 2.0 dans AToM^3, nous avons proposé le méta-modèle *« ClassDiagram_META »,* présenté dans la Figure 32. Il se compose de cinq classes et cinq associations.

Les classes *« Class_Diagram »*, *« Class »*, *« Package »*, *« Ternary_Association »* et *« Interface »* représentent respectivement un diagramme de classe, une classe, un paquetage, une association n-aire et une interface. Les associations du méta-modèles sont: *« ClassDiagram_Start »*, *« Simple_Association »*, *«Realize_1 »*, *«Realize_2 »* et *« T_Association ».*

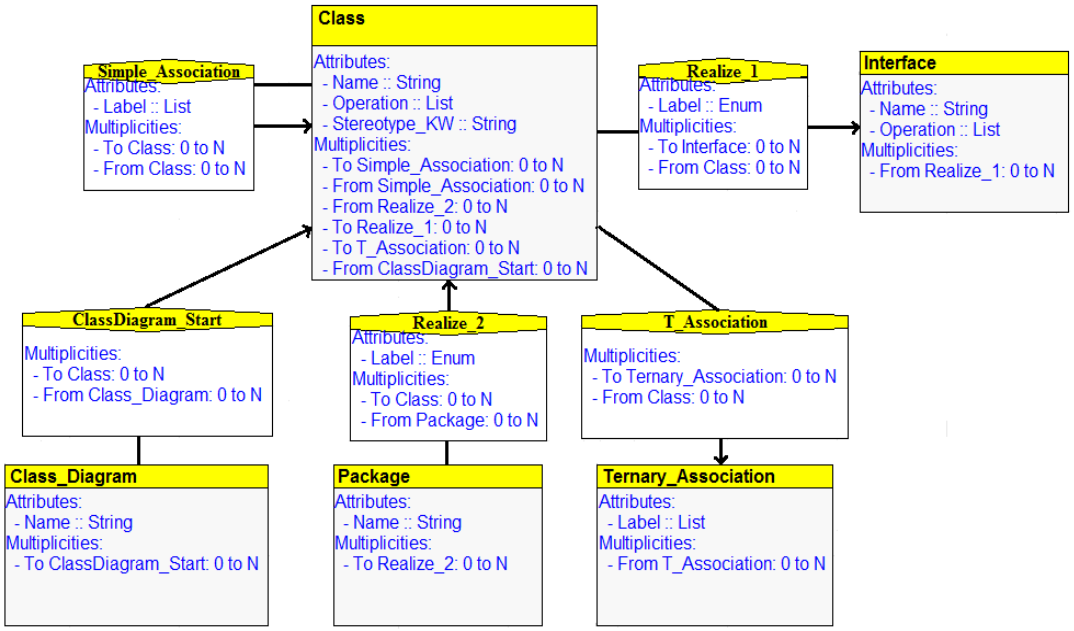


Figure 32. Méta-modèle d'UML 2.0 CD

La description détaillée des composants du méta-modèle *« ClassDiagram_META »* est comme suit:

- **La classe *« Class_Diagram »***: Cette classe représente un diagramme de classe UML 2.0. Elle est graphiquement représentée par un rectangle qui dispose, en haut à gauche, d'un attribut *Name* de type *String,* signifiant le nom du diagramme de classe.
- **La classe *« Class »*** : Cette classe représente une classe dans un diagramme de classe. Elle est graphiquement représentée par un rectangle divisé en trois compartiments et possède deux attributs : *Name* et *Stereotype_KW* de type *String,* représentés dans le premier compartiment, ainsi qu'un attribut *Operation* de type *List,* représenté dans le troisième compartiment.
- **La classe *« Package »*** : Cette classe modélise un paquetage dans un diagramme de classe. Graphiquement, elle est représentée par un rectangle qui porte à son éxtrimité haute gauche un petit carré (un dossier) et possède un attribut *Name* de type *String*.
- **La classe *« Ternary_Association »*** : Cette classe modélise une association n-aire qui lie plus de deux classes. Graphiquement , elle est représentée par un grand

losange et possède un attribut *Label* de type *List* représentant la liste d'opérations.

- **La classe *« Interface »*** : Cette classe modélise une interface. Elle est graphiquement représentée par un rectangle divisé en trois compartiments. Elle possède un attribut *Name* de type *String,* représenté avec le stéreotype *Interface* dans le premier compartiment, ainsi qu'un attribut *Operation* de type *List,* représenté dans le troisième compartiment.
- **L'association *« ClassDiagram_Start »*** : Cette association relie un diagramme de classe à l'une de ses classes. Elle n'a pas d'apparence graphique et associe à chaque instance de la classe *« Class_Diagram »* une seule instance de la classe *« Class ».*
- **L'association *« Simple_Association »*** : Cette association modélise une association simple entre deux classes. Elle est graphiquement représentée par un simple trait noire. Elle contient un attribut *Label* de type *List*. L'association *« Simple_Association »* porte l'attribut *Label* et associe à chaque instance de la classe *« Class »* une seule instance d'elle-même.
- **L'association *«Realize_1 »*** : Cette association modélise une association de réalisation entre une classe et une interface. Elle est graphiquement représentée par une simple flèche bleue. Elle contient un attribut *Label* de type *Enum (« T », « F », « FD »)*. L'association *«Realize_1 »* porte l'attribut *Label* et associe à chaque instance de la classe *« Class »* une seule instance de la classe *« Interface »*.
- **L'association *«Realize_2 »*** : Cette association modélise une association de réalisation entre un paquetage et une classe. Elle est graphiquement représentée par une simple flèche bleue. Elle contient un attribut *Label* de type *Enum (« T », « F », « FD »)*. L'association *«Realize_2 »* porte l'attribut *Label* et associe à chaque instance de la classe *« Package »* une seule instance de la classe *« Class »*.
- **L'association *« T_Association »*** : Cette association relie une classe à une association n-aire. Elle est graphiquement représentée par un simple trait noire et associe à chaque instance de la classe *« Class »* une ou plusieurs instances de la classe *« Ternary_Association »*.

Le méta-modèle proposé, *«ClassDiagram_META»,* nous a permis de générer l'outil visuel de modélisation des diagrammes de classes UML 2.0, présenté dans la Figure 33.

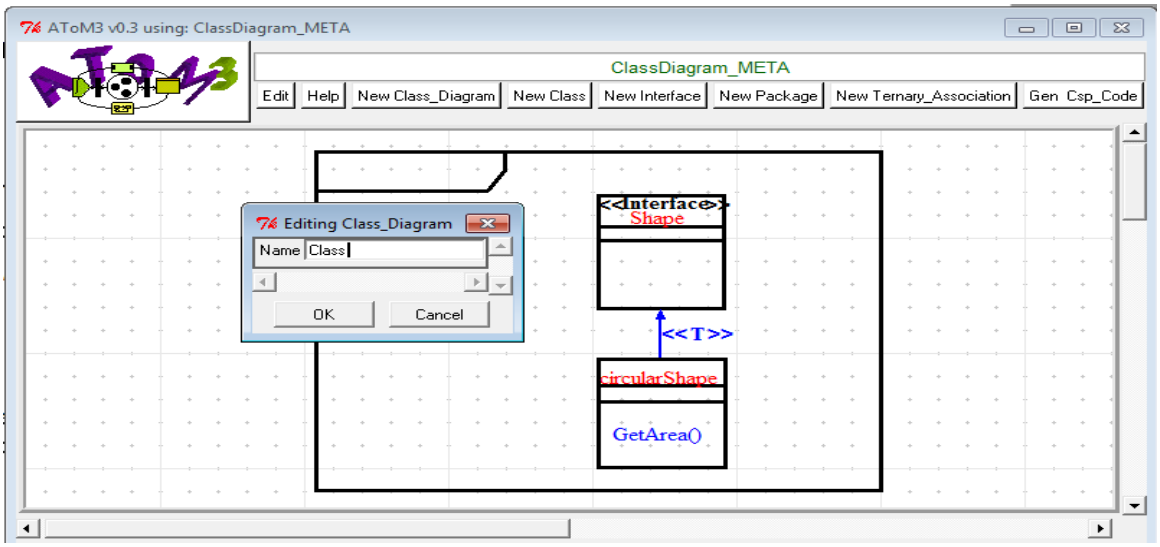


Figure 33. Outil de modélisation des diagrammes de classes UML 2.0

4.2.2 Transformation des diagrammes UML 2.0

Dans cette étape, nous avons proposé trois grammaires de graphes pour générer automatiquement les spécifications CSP à partir des diagrammes UML 2.0. La définition des règles des grammaires repose sur les formalisations d'UML 2.0 en CSP présentées dans le chapitre précédent. Tous les composants formalisés ont été implémentés, et pour chaque grammaire, les actions des règles codent automatiquement dans le langage CSP_M les différents composants modélisés à l'aide de l'outil visuel généré.

4.2.2.1 La grammaire SD2CSP_GG : Transformation d'UML 2.0 SD en CSP

Afin de générer le code CSP à partir des diagrammes de séquence UML 2.0, nous avons proposé la grammaire de graphes *SD2CSP_GG* en utilisant l'outil de transformation de graphes $AToM^3$ (Hamrouche et al., 2022). La grammaire développée contient onze règles, une action initiale et une action finale. Chaque règle se compose d'une condition d'application, une action et une représentation graphique contenant deux cotés identiques: *LHS* et *RHS*. Si la condition est vérifiée, l'action écrite en code Python sera exécutée et les expressions CSP correspondantes seront automatiquement

générées. Le résultat de la transformation est en CSP_M, La machine dialecte lisible du CSP, il est enregistré dans un fichier texte « *mycspfilesd.csp* », qui est le type d'entrée accepté par l'outil FDR4. La Figure 34 montre la grammaire de graphes *SD2CSP_GG*, alors que la description des règles est la suivante:

Action initiale: Le rôle de l'action initiale est d'ouvrir le fichier « *mycspfilesd.csp* » , où le code CSP sera généré, et de créer et initialiser les variables temporaires.

Règle 1: rule_AMessage *(Priorité 1):* Cette règle est appliquée pour localiser un message *asynchrone* non encore traité et générer l'événement envoyé, l'événement reçu, le processus *« message* », ainsi que son alphabet.

Règle 2: rule_SMessage *(priorité 2):* Cette règle est appliquée pour localiser un message *synchrone* non encore traité et générer l'événement envoyé, l'événement reçu, le processus *« message »,* ainsi que son alphabet.

Règle 3: rule_Acknowledgement *(priorité 3):* Cette règle est appliquée pour localiser un message de retour non encore traité et générer l'événement envoyé, l'événement reçu, le processus *« message* », ainsi que son alphabet.

Règle 4: rule_IOcontainsLLam *(priorité 4):* Cette règle est utilisée pour générer le processus *« PrefixComposition »* pour la ligne de vie de l'objet émetteur et la ligne de vie de l'objet récepteur d'un message asynchrone.

Règle 5: rule_IOcontainsLLsm *(priorité 5):* Cette règle est utilisée pour générer le processus « PrefixComposition » pour la ligne de vie de l'objet émetteur et la ligne de vie de l'objet récepteur d'un message synchrone.

Règle 6: rule_IOcontainsLLack *(priorité 6):* Cette règle est utilisée pour générer le processus « PrefixComposition » pour la ligne de vie de l'objet émetteur et la ligne de vie de l'objet récepteur d'un message de retour.

Règle 7: rule_IOcontainsCF *(priorité 7):* Cette règle est appliquée pour localiser un fragment combiné fils d'un opérande d'interaction.

Règle 8: rule_CFcontainsIO *(priorité 8):* Cette règle est appliquée pour localiser un opérande d'interaction sans fragment combiné imbriqué (haschild==0) et générer le processus *« Messages »* et son alphabet, le processus « *Lifelines »* et son alphabet, le processus « *PrefixComposition »* et le processus « *seq ».*

Règle 9: rule_NestedCF *(priorité 9):* Cette règle est appliquée pour localiser une interaction complexe contenant deux fragments combinés imbriqués et générer le processus « *Messages »* et son alphabet, le processus « *Lifelines »* et son alphabet, le processus « PrefixComposition » et le processus « *seq ».*

Règle 10: rule_IcontainsCF *(priorité 10):* Cette règle est utilisée pour générer le processus adéquat au type de l'opérateur d'interaction « *IOKind* ».

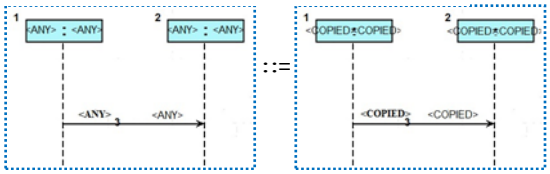
Règle 11: rule_Interaction *(priorité 11):* Cette règle est utilisée pour générer le processus principal « *P»* du diagramme de séquence et enregistrer le code généré dans le fichier « *mycspfilesd.txt »*.

Action Finale: Le rôle de l'action finale est d'effacer les attributs temporaires et de fermer le fichier de sortie.

1- rule_AMessage (priorité 1)

Condition : Node(3).visited == 0

LHS ***RHS***

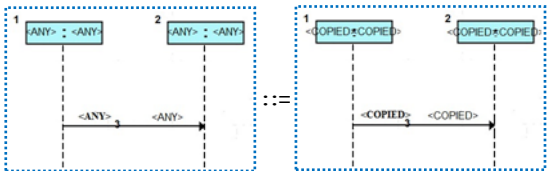


Action: - Générer Event Sent, Event Received, le processus Message, AlphaMessage
- Node(3).visited = 1

2- rule_SMessage (priorité 2)

Condition : Node(3).visited == 0

LHS ***RHS***

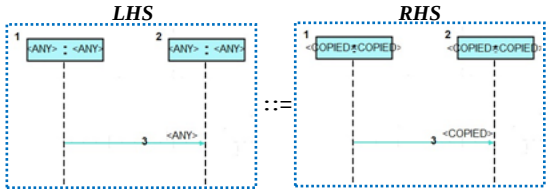


Action: - Générer Event Sent, Event Received, le processus Message, AlphaMessage
- Node(3).visited = 1

3- rule_Acknowledgement (priorité 3)

Condition : Node(3).visited == 0

LHS ***RHS***

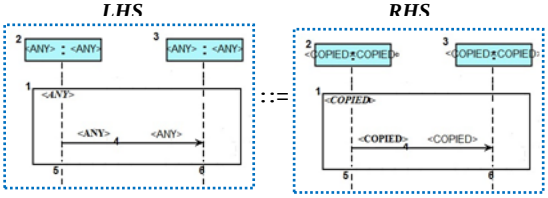


Action: - Générer Event Sent, Event Received, le processus Message, AlphaMessage
- Node(3).visited = 1

4- rule_IOcontainsLLam (priorité 4)

Condition : Node(4).visited == 1

LHS ***RHS***

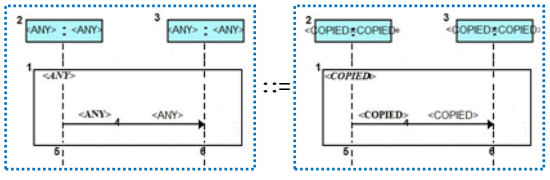


Action: - Générer le processus PrefixComposition
- Node(4).visited = 2

5- rule_IOcontainsLLsm (priorité 5)

Condition : Node(4).visited == 1

LHS ***RHS***

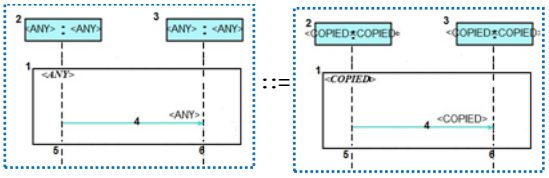


Action: - Générer le processus PrefixComposition
- Node(4).visited = 2

6- rule_IOcontainsLLack (priorité 6)

Condition : Node(4).visited == 1

LHS ***RHS***

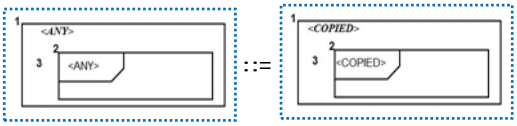


Action: - Générer le processus PrefixComposition
- Node(4).visited = 2

7- rule_ IOcontainsCF (priorité 7)

Condition : Node(3).visited == 0

LHS ***RHS***

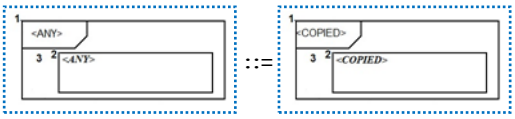


Action: - Node(1).haschild=1
- Node(3).visited = 1

8- rule_CFcontainsIO (priorité 8)

Condition: (Node(2).haschild == 0) and (Node(3).visited == 0)

LHS ***RHS***

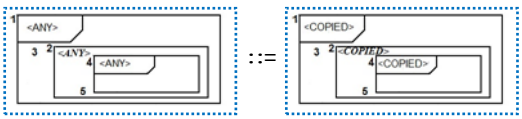


Action: - Générer le processus Messages, AlphaMessages, le processus Lifelines, AlphaLifelines, le processus Seq.
- Node(3).visited = 1

9- rule_NestedCF (priorité 9)

Condition : (Node(2).haschild == 1) and (Node(3).visited == 0)

LHS ***RHS***

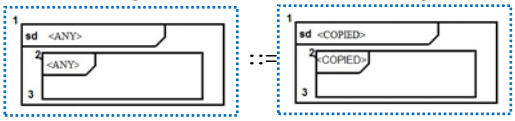


Action: - Générer le processus Messages, AlphaMessages, le processus Lifelines, AlphaLifelines, le processus Seq.
- Node(3).visited = 1

10- rule_IcontainsCF (priorité 10)

Condition : Node(3).visited == 0

LHS ***RHS***

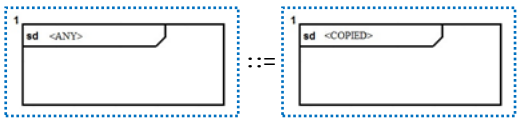


Action: - Générer le code CSP correspondant à IOKind
- Node(3).visited = 1

11- rule_Interaction (priorité 11)

Condition : Node(1).visited == 0

LHS ***RHS***



Action: - Générer le processus principal P
- Enregistrer code généré dans *mycspfilesd.txt*
- Node(1).visited = 1

Figure 34. Grammaire de graphes pour transformer UML 2.0 SD vers CSP

la

Figure 35 montre en détail la première règle de la grammaire de graphes (*rule_AMessage*).

Condition : localiser un message asynchrone qui n'a pas été traité auparavant

```
node=self.getMatched(graphID, self.LHS.nodeWithLabel(3))
return node.visited == 0
```

Action*:* Générer événement envoyé, événement reçu et le processus *Message()* pour un message

```
import string
node1=self.getMatched(graphID,self.LHS.nodeWithLabel(1))
node2=self.getMatched(graphID, self.LHS.nodeWithLabel(2))
node3=self.getMatched(graphID, self.LHS.nodeWithLabel(3))
sender = node1.Name.getValue()
receiver = node2.Name.getValue()
message = node3.Name.getValue()
id = node3.Id.getValue()
O = node3.Order.getValue()
Iop = node3.IOName.getValue()
es=msg.asynch.'+id+'.snd.'+sender+'.'+receiver+'.'+message
ess='msg.asynch.'+id+'.snd.'+sender+'.'+receiver+'.'+ message +':' + iop
er= 'msg.asynch.'+id+'.rcv.'+sender+'.'+receiver+'.'+message
err='msg.asynch.'+id+'.rcv.'+sender+'.'+receiver+'.'+message+':'+iop
node1.listS= es+':'+er
node3.m=(asynchronous,'+id+','+sender+','+receiver+','+ message+')'
node3.PMsg= 'Message(asynchronous,'+id+','+sender+','+receiver+','+mes sage+')='+es+' -> '+ er+' ->
skip'
node3.AMsg='AlphaMessage(asynchronous,'+id+','+sender+','+receiver+','+message+')={|'+es+','+er+'|}'
pmsg = node3.PMsg
amsg = node3.AMsg
node1.listS.insert(O,ess)
node2.listS.insert(O,err)
node3.visited  = 1
```

Figure 35. Code python pour la transformation d'un message asynchrone

4.2.2.2 La grammaire StateD2CSP_GG : Transformation d'UML 2.0 STM en CSP

Pour la production du code CSP à partir des diagrammes d'états-transitions UML 2.0, nous avons proposé la grammaire de graphes *StateD2CSP_GG*, comprenant vingt-cinq règles, une action initiale et une action finale. Les Figures 36, 37 et 38 exposent la grammaire proposée, tandis que la description des règles est la suivante :

Règle1 : rule_stm2IS *(priorité 1)* : Cette règle est appliquée pour localiser l'état initial du diagramme d'états-transitions (*Node.begin=1*).

Règle2 :rule_IS2SS *(priorité 2)* : Cette règle est utilisée pour transformer un état initial lié à un état simple en code CSP correspondant. Elle gère le cas d'un état initial d'un diagramme d'états-transitions et celui d'une région à l'intérieur d'un état composite.

Règle3 :rule_IS2Fork *(priorité 3)* : Cette règle est utilisée pour générer les processus correspondant au cas d'un état initial lié à un pseudo-état de bifurcation, lui-même lié à un état simple.

Règle4 :rule_SS2SS *(priorité 4)* : Cette règle transforme deux états simples liés par une transition vers le code CSP correspondant. Elle traite les différents cas de transitions étiquetées par événement, garde ou action, ainsi que les transitions non étiquetées.

Règle5 :rule_SS2CS *(priorité 5)* : Cette règle transforme un état simple lié à un état composite par une transition vers le code CSP correspondant. Elle traite les différents cas de transitions étiquetées par événement, garde ou action, ainsi que les transitions non étiquetées.

Règle6 :rule_CS2SS *(priorité 6)* : Cette règle transforme un état composite lié à un état simple par une transition vers le code CSP correspondant. Elle traite les différents cas de transitions étiquetées par événement, garde ou action, ainsi que les transitions non étiquetées.

Règle7 :rule_CS2CS *(priorité 7)* : Cette règle transforme deux états composites liés par une transition vers le code CSP correspondant. Elle traite les différents cas de transitions étiquetées par événement, garde ou action, ainsi que les transitions non étiquetées.

Règle8 :rule_CS_hasRegion *(priorité 8)* : Cette règle transforme un état composite contenant une région avec état initial vers le code CSP correspondant.

Règle9 :rule_Region_hasCS *(priorité 9)* : Cette règle est appliquée pour transformer deux états composites, où l'un des deux est un super-état qui englobe l'autre, vers le code CSP

correspondant (d'après le méta-modèle proposé, le super-état contient une région qui englobe le deuxième état composite).

Règle10 :rule_Region_hasFS *(priorité 10)* : Cette règle est appliquée pour transformer un état composite contenant une région qui englobe un état final en code CSP correspondant.

Règle11 :rule_SS2FS *(priorité 11)* : Cette règle est utilisée pour transformer un état simple lié à un état final en code CSP correspondant.

Règle12 :rule_CS2FS *(priorité 12)* : Cette règle est utilisée pour transformer un état composite lié à un état final en code CSP correspondant.

Règle13 :rule_PseudoState_Choice *(priority 13)* : Cette règle transforme un pseudo-état de décision avec deux états simples, l'un en entrée et l'autre en sortie du pseudo-état, en code CSP correspondant. Elle traite les différents cas de transitions étiquetées par événement, garde ou action, ainsi que les transitions non étiquetées.

Règle14 :rule_ PseudoState_Fork *(priorité 14)* : Cette règle transforme un pseudo-état de bifurcation lié à deux états simples, l'un en entrée et l'autre en sortie du pseudo-état, en code CSP correspondant.

Règle15 :rule_ State2Junction *(priorité 15)* : Cette règle transforme un état simple lié à un pseudo-état de jonction en code CSP correspondant. Elle traite les différents cas de transitions étiquetées par événement, garde ou action, ainsi que les transitions non étiquetées.

Règle16 :rule_ Junction2State *(priorité 16)* : Cette règle génère le processus correspondant au cas d'un pseudo-état de jonction lié à un état simple et enregistre la garde de la transition.

Règle17 :rule_ PseudoState_Join *(priorité 17)* : Cette règle transforme un pseudo-état de jointure lié à deux états simples, l'un en entrée et l'autre en sortie du pseudo-état, en code CSP correspondant.

Règle18 :rule_ CS_Gen *(priorité 18)* : Cette règle génère le code CSP correspondant à un état composite, puis l'enregistre dans le fichier « *mycspfilestm.txt ».*

Règle19 :rule_ IS_Gen *(priorité 19)* : Cette règle transforme un état initial qui appartient à une région en code CSP correspondant, ensuite l'enregistre dans le fichier « *mycspfilestm.txt ».*

Règle20 :rule_ Junction_Gen *(priorité 20)* : Cette règle génère le code CSP correspondant à un pseudo-état de jonction, puis l'enregistre dans le fichier « *mycspfilestm.txt ».*

Règle21 :rule_ Fork_Gen *(priorité 21)* : Cette règle génère le code CSP correspondant à un pseudo-état de bifurcation, puis l'enregistre dans le fichier « *mycspfilestm.txt ».*

Règle22 :rule_ Choice_Gen *(priorité 22)* : Cette règle génère le code CSP correspondant à un pseudo-état de décision, ensuite l'enregistre dans le fichier « *mycspfilestm.txt ».*

Règle23 :rule_ Join_Gen *(priorité 23)* : Cette règle génère le code CSP correspondant à un pseudo-état de jointure, ensuite l'enregistre dans le fichier « *mycspfilestm.txt ».*

Règle24 :rule_ State_Gen *(priorité 24)* : Cette règle génère le code CSP correspondant à un état simple, puis l'enregistre dans le fichier « *mycspfilestm.txt ».*

Règle25 :rule_FS_Gen *(priority 25)* : Cette règle génère le code CSP correspondant à un état final, puis l'enregistre dans le fichier « *mycspfilestm.txt ».*

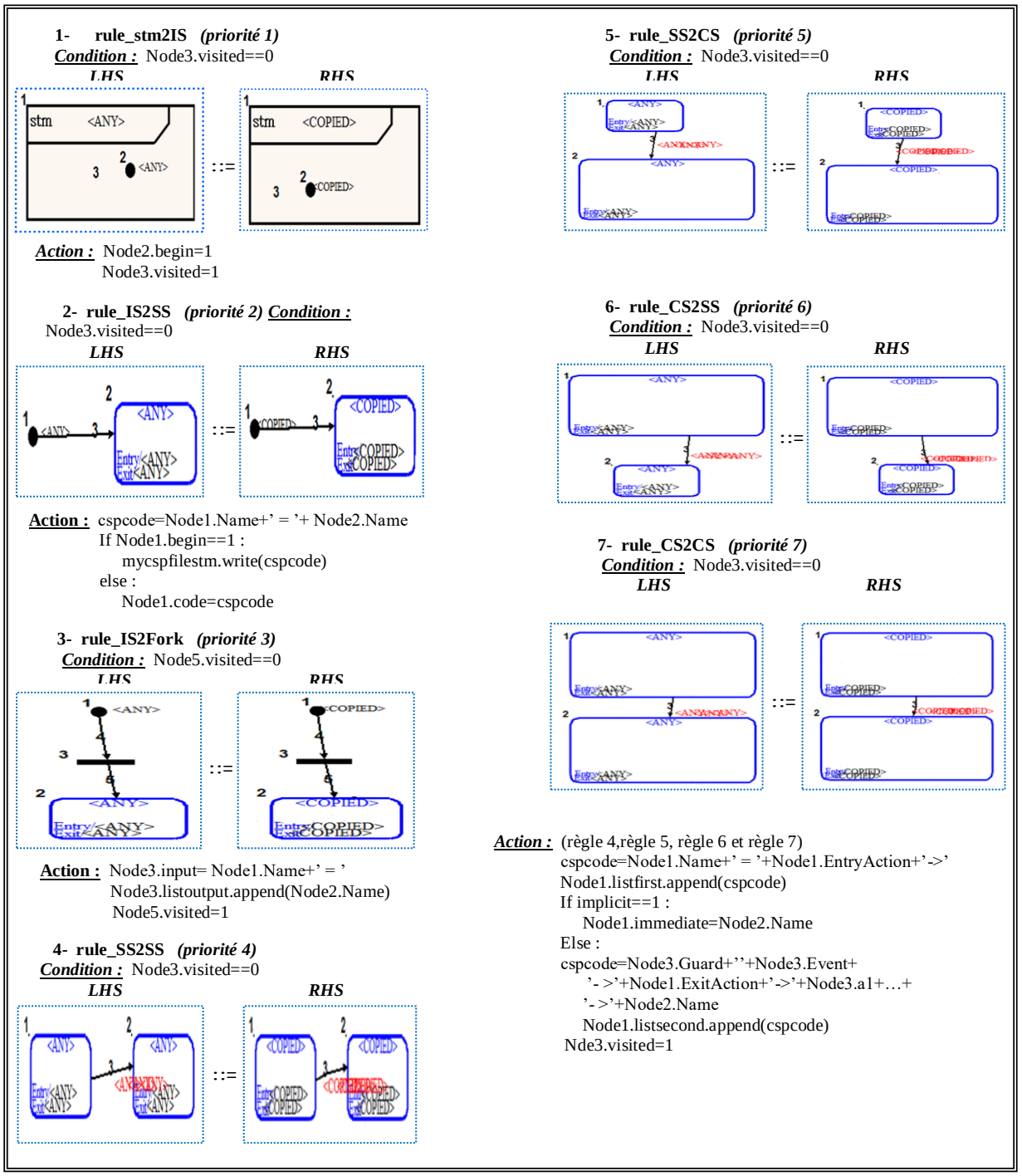
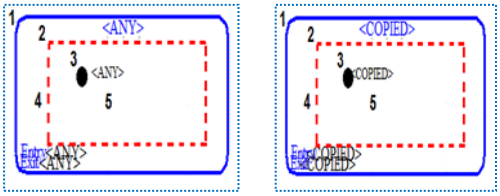


Figure 36. Grammaire de graphes pour la transformation d'UML 2.0 STM en CSP (partie 1)

8- rule_CS_hasRegion ***(priorité 8)***

Condition : Node5.visited==0

LHS ***RHS***

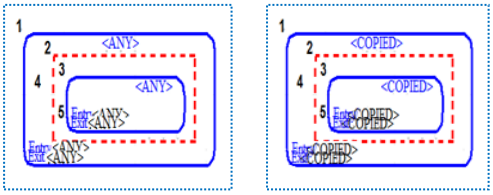


Action : Node1.listfirst.append(Node1.Name+' = '+ Node1.EntryAction+'->')
Node1.listinitiaux.append(Node3.Name)
Node5.visited=1

9- rule_Region_hasCS ***(priorité 9)***

Condition : Node5.visited==0

LHS ***RHS***

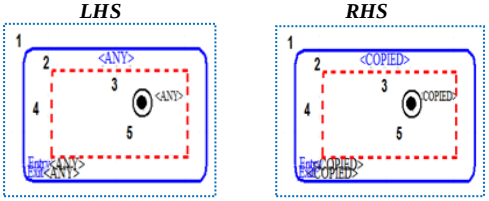


Action : for cspcode in Node1.listsecond :
Node3.listsecond.append(cspcode)
Node5.visited=1

10- rule_Region_hasFS ***(priorité 10)***

Condition : Node5.visited==0

LHS ***RHS***

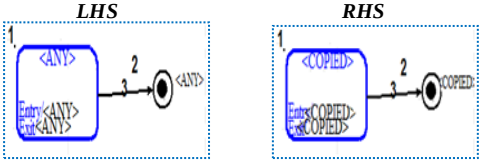


Action : Node3.listsecond= Node1.listsecond
Node3.immediate= Node1.immediate
Node5.visited=1

11- rule_SS2FS ***(priorité 11)***

Condition : Node3.visited==0

LHS ***RHS***

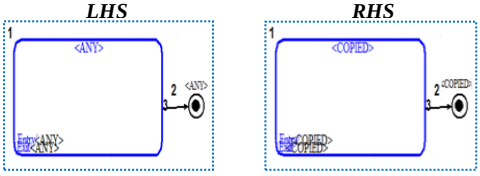


Action : Node1.listfirst.append(Node1.Name+' = '+ Node1.EntryAction+' ->')
Node1.listsecond.append(Node1.ExitAction+ '->'+Node2.Name)
Node3.visited=1

12- rule_CS2FS ***(priorité 12)***

Condition : Node3.visited==0

LHS ***RHS***

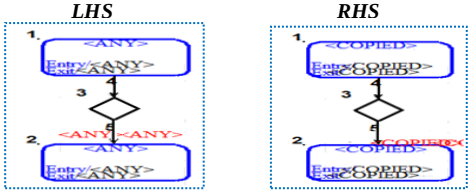


Action : Node1.listfirst.append(Node1.Name+' = '+ Node1.EntryAction+' ->')
Node1.listsecond.append(Node1.ExitAction+ '->'+Node2.Name)
Node3.visited=1

13- rule_PseudoState_Choice ***(priority 13)***

Condition : Node5.visited==0

LHS ***RHS***



Action : Node3.input=Node1.Name+' = '+Node1.EntryAction +' -> ('
Node3.listoutput.append('('+Node5.Guard+ ' &'+Node5.Event+'->'+Node1.ExitAction+'->'+Node2. Name+ ')')
Node5.visited=1

14- rule_ PseudoState_Fork ***(priorité 14)***

Condition : Node5.visited==0

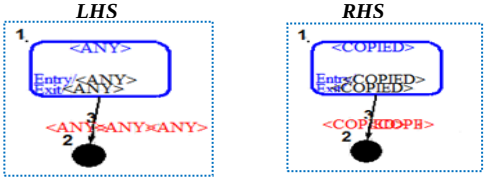
LHS ***RHS***

Action : Node3.input=Node1.Name+' = '+Node1.EntryAction +' -> '+Node1.ExitAction+'->'
Node3.listoutput.append(Node2.Name)
Node5.visited=1

15- rule_ State2Junction ***(priorité 15)***

Condition : Node3.visited==0

LHS ***RHS***

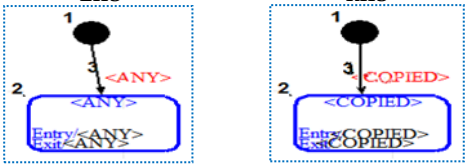


Action : Node2.listfirst.append(Node1.Name+' = '+Node1.EntryAction+' ->'+Node3.Guard+' &'+ Node3.Event+'->')
Node2.listsecond.append(Node1.ExitAction+'->'+Node3.a1+..+'->')
Node3.visited=1

16- rule_ Junction2State ***(priorité 16)***

Condition : Node3.visited==0

LHS ***RHS***

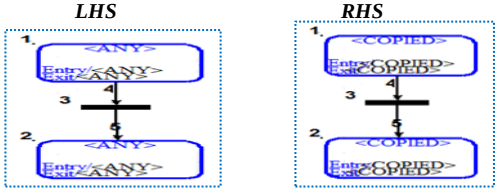


Action : Node1.listtarget.append(Node2.Name)
Node1.listguard.append(Node3.Guard)
Node3.visited=1

17- rule_ PseudoState_Join ***(priorité 17)***

Condition : Node4.visited==0

LHS ***RHS***



Action : Node3.output= Node2.Name
Node3.listinput.append(Node1.Name+' = '+ Node1.EntryAction+'->'+
Node1.ExitAction+'-> ProcessJoin ->')
Node4.visited=1

Figure 37. Grammaire de graphes pour la transformation d'UML 2.0 STM en CSP (partie2)

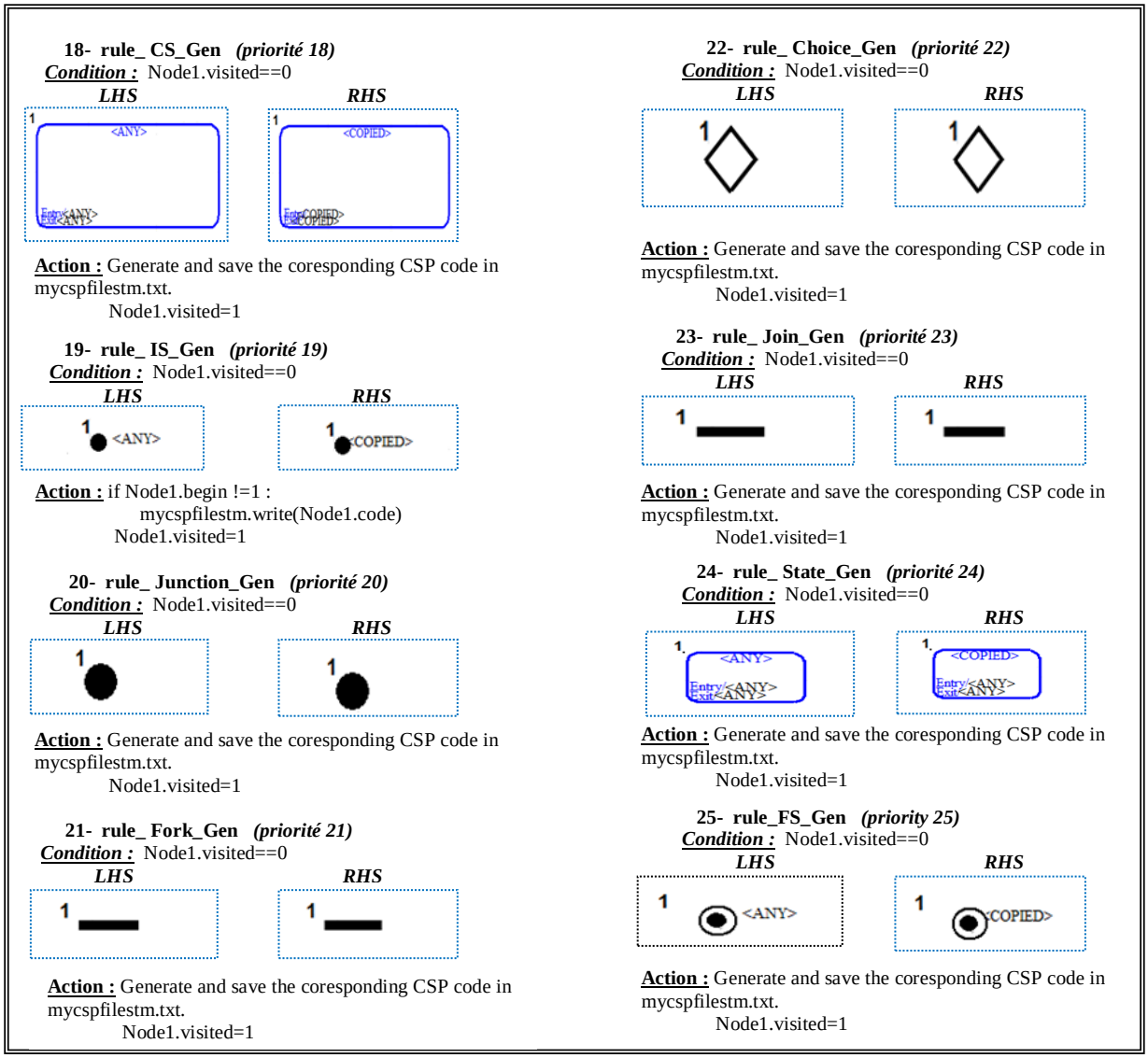


Figure 38. Grammaire de graphes pour la transformation d'UML 2.0 STM en CSP (partie 3)

La Figure 39 présente en détail la règle 24 (rule_ State_Gen).

Condition : Localiser un état simple qui n'a pas été traité auparavant.

```
node=self.getMatched(graphID, self.LHS.nodeWithLabel(1))
return node.visited == 0
```

Action*:* Générer et enregistrer le code CSP correspondant à un état simple dans « mycspfilestm.txt ».

```
import string
node = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
v='0'
for connection in node.out_connections_:
   for nodex in connection.out_connections_:
      if  v=='0' :
        v='1'
if v=='0':
   source=node.Name.getValue()
   codecsp=source+' = STOP'
elif v=='1':
   lfirst=node.listfirst
   lsecond=node.listsecond
   immed=node.imm
   n=len(lsecond)
   codecsp=lfirst[0]
   if n==0:
     codecsp=codecsp+immed
   elif n==1:
     codecsp=codecsp+lsecond[0]
     if immed!='':
        codecsp=codecsp+' timeout '+'('+immed+')'
   elif n>1:
     codecsp=codecsp+'(('+lsecond[0]+')'
     i=1
     while i<n:
        codecsp=codecsp+' [] '+'('+lsecond[i]+')'
        i=i+1
     codecsp=codecsp+')'
     if immed!='':
        codecsp=codecsp+' timeout '+'('+immed+')'

obFichier=open('mycspfilestm.txt','a')
obFichier.write(codecsp+' \n')
node.visited=1
```

Figure 39. Génération du code CSP correspondant à un état simple

4.2.2.3 La grammaire CD2CSP_GG : Transformation d'UML 2.0 CD en csp

Nous avons proposé la grammaire de graphes *CD2CSP_GG* pour générer le code CSP à partir des diagrammes de classes UML 2.0. La grammaire développée contient six règles, une action initiale et une action finale. La Figure 40 montre la grammaire proposée, alors que la description des règles est la suivante:

Règle1 : rule_Class2Class *(priorité 1) :* Cette règle transforme deux classes liées par une association simple en code CSP correspondant, puis l'enregistre dans le fichier « *mycspfilecd.txt ».*

Règle2 : rule_Class *(priorité 2) :* Cette règle génère le code CSP correspondant à une classe, ensuite l'enregistre dans le fichier « *mycspfilecd.txt ».*

Règle3 : rule_Realize_Interface *(priorité 3) :* Cette règle transforme une classe liée à une interface par une association de réalisation en code CSP correspondant, ensuite l'enregistre dans le fichier « *mycspfilecd.txt ».*

Règle4 : rule_Realize_Package *(priorité 4) :* Cette règle est appliquée pour transformer un paquetage lié à une classe par une association de réalisation en code CSP correspondant, puis l'enregistre dans le fichier « *mycspfilecd.txt ».*

Règle5 : rule_TAssociation *(priorité 5) :* Cette règle transforme une classe participante à une association n-aire en code CSP correspondant.

Règle6 : rule_TAssociation_Gen *(priorité 6) :* Cette règle génère le code CSP correspondant à une association n-aire, puis l'enregistre dans le fichier « *mycspfilecd.txt ».*

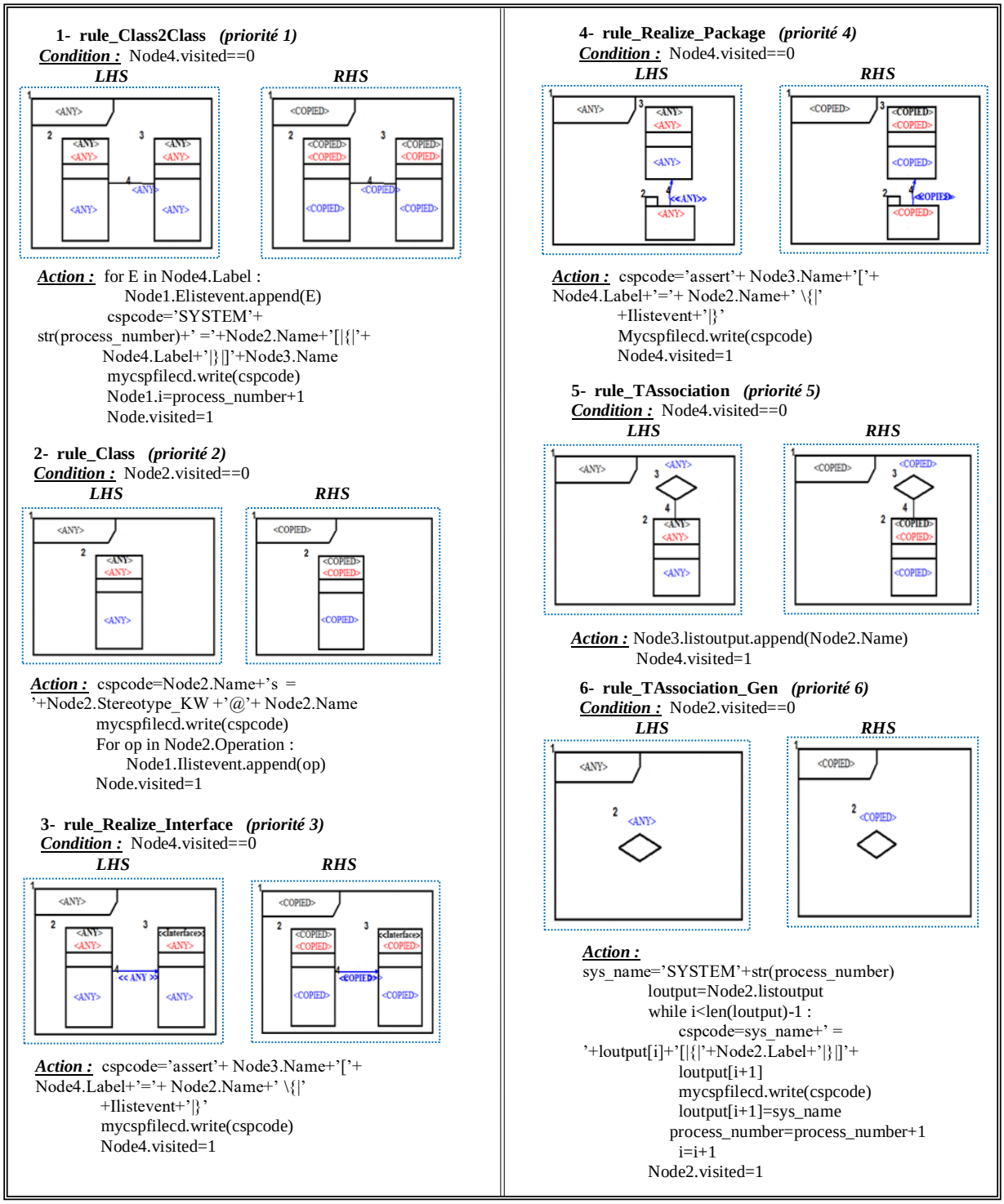


Figure 40. Grammaire de graphes pour la transformation d'UML 2.0 CD vers CSP

La

Figure 41 présente en détail la règle 6 (rule_TAssociation_Gen).

Condition : Localiser une association n-aire qui n'a pas été traité auparavant.

```
node=self.getMatched(graphID, self.LHS.nodeWithLabel(2))
return node.visited == 0
```

Action: Générer et enregistrer le code CSP correspondant à une association n-aire dans

```
import string
node1 = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
node2 = self.getMatched(graphID, self.LHS.nodeWithLabel(2))
events=node2.Label.getValue()
loutput=node2.listoutput
process_num=node1.i
code=[]
l=len(loutput)-1
i=0
sysname='SYSTEM'+str(process_num)
while i<l:
   codecsp=sysname+' = '+loutput[i]+' [|{|'+events+'|}|] '+loutput[i+1]
   code.append(codecsp)
   loutput[i+1]=sysname
   process_num=process_num+1
   sysname='SYSTEM'+str(process_num)
   node1.i=process_num
   i=i+1

for cod in code:
   obFichier=open('mycspfilecd.txt','a')
   obFichier.write(cod+' \n')
node2.visited=1
```

Figure 41. Génération du code CSP correspondant à une association n-aire

4.2.3 Vérification

Nous pouvons distinguer trois façons pour obtenir la compréhension mathématique de la signification d'un programme CSP. Ces façons sont : la sémantique algébrique, opérationnelle et dénotationnelle. Les différentes sémantiques dénotationnelles pour le CSP sont basées sur les traces, les échecs et les divergences (Roscoe A. W., 1997). Le modèle de traces associe à chaque processus P, les séquences finies d'événements permises par ce processus. Ainsi, ce modèle permet de représenter les comportements possibles des processus sous forme de traces. Les traces(P) désignent les traces du processus P. Le modèle de défaillances stables du processus P, noté Failures(P), associe à chaque processus P des couples de la forme (s,X), où s est une trace finie admise par P, et X est l'ensemble des événements qui ne peuvent être exécuté q'après l'exécution des événements de s. Enfin, le modèle Failures-divergences associe à chaque processus P

l'ensemble de ses défaillances et divergences stables. Le processus P est divergent s'il est dans un état où les seuls événements possibles sont les événements internes. Cet état est dit divergent. L'ensemble des divergences(P) est l'ensemble des traces s ; en conséquence, le processus se trouve dans un état divergent après l'exécution de s. Le concept de raffinement consiste à calculer et comparer les modèles sémantiques de deux processus. FDR4 est un vérificateur de raffinement pour CSP conçu pour analyser des modèles écrits en CSP_M. La machine dialecte lisible de CSP, elle combine les opérateurs CSP de Hoare avec un langage de programmation fonctionnel. L'objectif principal n'est pas de décrire des algorithmes sous une forme exécutable mais de soutenir la description de systèmes parallèles d'une manière automatique de manipulation. FDR4 permet diverses formes d'assertions pour vérifier les propriétés d'interblocage, de livelock, de déterminisme et spécifié la réduction d'ordre partiel et les assertions de de raffinement. Si P et Q sont deux processus, et S est l'un des trois modèles sémantiques (T : Traces, F : Failures, FD : Failures-Divergences), alors FDR peut simplement vérifier si Q affine P ($P \sqsubseteq_S Q$) en insérant le assertion suivante dans le code CSP_M : assert P[s = Q. Nous pouvons utiliser le modèle de raffinement de trace pour vérifier la sécurité et le modèle de défaillances stables pour vérifier la propriété de vivacité.

4.3 CONCLUSION

Dans ce chapitre, nous avons proposé une approche basée sur la méta-modélisation et la transformation de graphes, permettant la modélisation de la structure et du comportement des systèmes en utilisant UML 2.0 SD, UML 2.0 STM et UML 2.0. De plus, nous avons introduit la transformation des modèles spécifiés en code CSP vérifiable.. L'automatisation de cette approche repose sur la définition de trois méta-modèles et trois grammaires de graphes, en utilisant AToM3. Dans le prochain chapitre, nous utiliserons les outils visuels générés pour modéliser et transformer automatiquement les modèles UML 2.0 crées en CSP dans une étude de cas.

CHAPITRE

5

ETUDE DE CAS

5.1 INTRODUCTION

Dans ce chapitre, nous illustrons notre approche et les outils visuels générés à travers une étude de cas portant sur un ATM (Automated Teller Machine), autrement dit, un distributeur automatique de billets. Cette démonstration commence par la modélisation de la structure du système au moyen du diagramme de classes, et la modélisation de son comportement grâce à l'utilisation des diagrammes de séquence et des diagrammes d'états-transitions. Par la suite, nous appliquons les trois grammaires de graphes pour générer les spécifications CSP correspondantes. Le code ainsi généré sera soumis à une vérification à l'aide du model-checker FDR4. Enfin, nous vérifions quelques propriétés de notre approche de transformation telles que la terminaison, le déterminisme et l'exactitude syntaxique.

5.2 MODELISATION EN UML 2.0

Dans ce chapitre, nous présentons à titre d'illustration de notre approche une version modifiée d'une étude de cas traditionnelle, proposée dans le travail (Lima et al., 2009) : le système ATM (Automated Teller Machine). Celui-ci est composé de trois entités en interaction : l'ATM, l'utilisateur (User) et la banque (Bank). Après l'insertion de la carte par l'utilisateur, l'ATM doit vérifier la carte et le numéro d'identification personnel (PIN). Si la vérification échoue, la carte doit être éjectée. Sinon, l'utilisateur a le choix d'effectuer certaines opérations et la carte est conservée dans la machine jusqu'à ce que l'utilisateur termine les transactions.

La Figure 42 présente le diagramme de séquence, le diagramme d'états-transitions et le diagramme de classe qui modélisent le système ATM en utilisant les outils proposés.

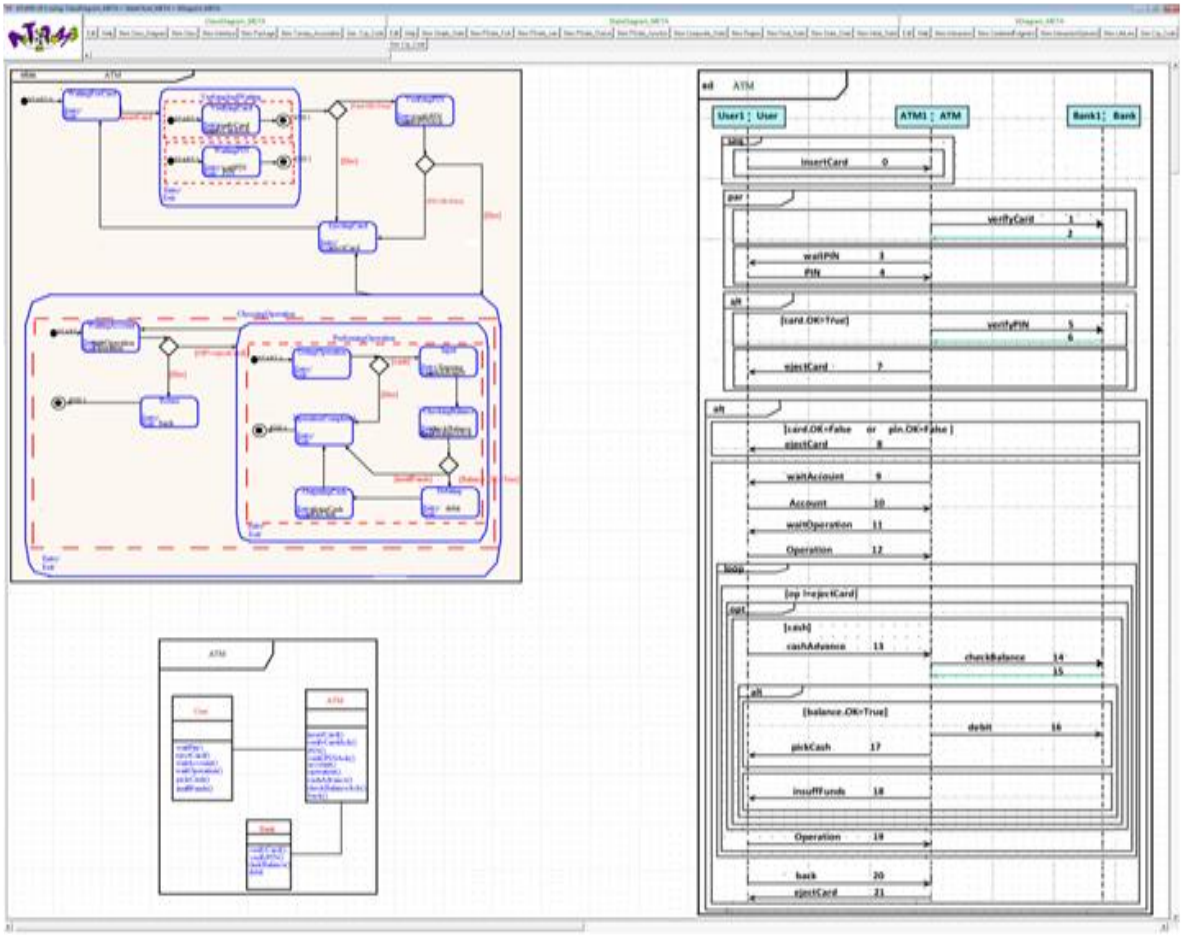


Figure 42. Modélisation de la machine ATM en utilisant UML 2.0 CD, UML 2.0 STM et UML 2.0 SD

5.2.1 Modélisation en UML 2.0 SD

La Figure 43 montre le comportement du système en utilisant UML 2.0 SD. Un utilisateur insère d'abord sa carte dans l'ATM (message *InsertCard()*); puis ce dernier va en parallèle (fragment combiné *par*): 1. communiquer avec la banque pour valider le statut de la carte ; 2. demander et recevoir le code PIN introduit par l'utilisateur. Deux résultats possibles concernant le statut de la carte (fragment combiné *alt*): 1. si l'état de la carte est valide (*Card.ok = true*), l'ATM demande et reçoit la validation du PIN ; 2. si la carte n'est pas valide (*Card.ok = false*), alors la carte est éjectée. Deux résultats possibles concernant la validation du PIN (fragment combiné *alt*) : 1. si le code PIN était non valide, la carte est éjectée ; 2. si le code PIN était valide, l'utilisateur peut procéder à l'opération avec son compte bancaire en utilisant l'ATM.

Un fragment combiné *alt* existe pour procéder avec une transaction de banque: 1. si la carte ou le code PIN n'était pas valide, la carte est éjectée ; 2. si la carte et le code PIN étaient valides, alors l'Utilisateur renseigne son compte et l'opération bancaire souhaitée pour que l'ATM procéder. En supposant que seulement deux opérations existent dans l'ATM : *CashAdvance* et *ejectCard*, le dernier fragment combiné *alt* contient une boucle (fragment combiné *loop*) concernant la sélection d'opération qui itère tant que l'opération choisie n'est pas *ejectCard*. Concernant *CashAdvance*, l'Utilisateur indique la quantité d'espèces requise, l'ATM vérifie le solde de la carte, puis délivre son statut. Ensuite, il existe un fragment combiné *alt* qui précise si (*Balance.Ok = true*) ou (*Balance.Ok = false*), indiquant ainsi soit de ramasser l'argent ou que le solde est insuffisant. Suivant ce fragment combiné *alt*, à l'intérieur de la boucle, l'utilisateur peut choisir une nouvelle opération dans l'ATM. Enfin, après avoir terminé la boucle, la carte est éjectée.

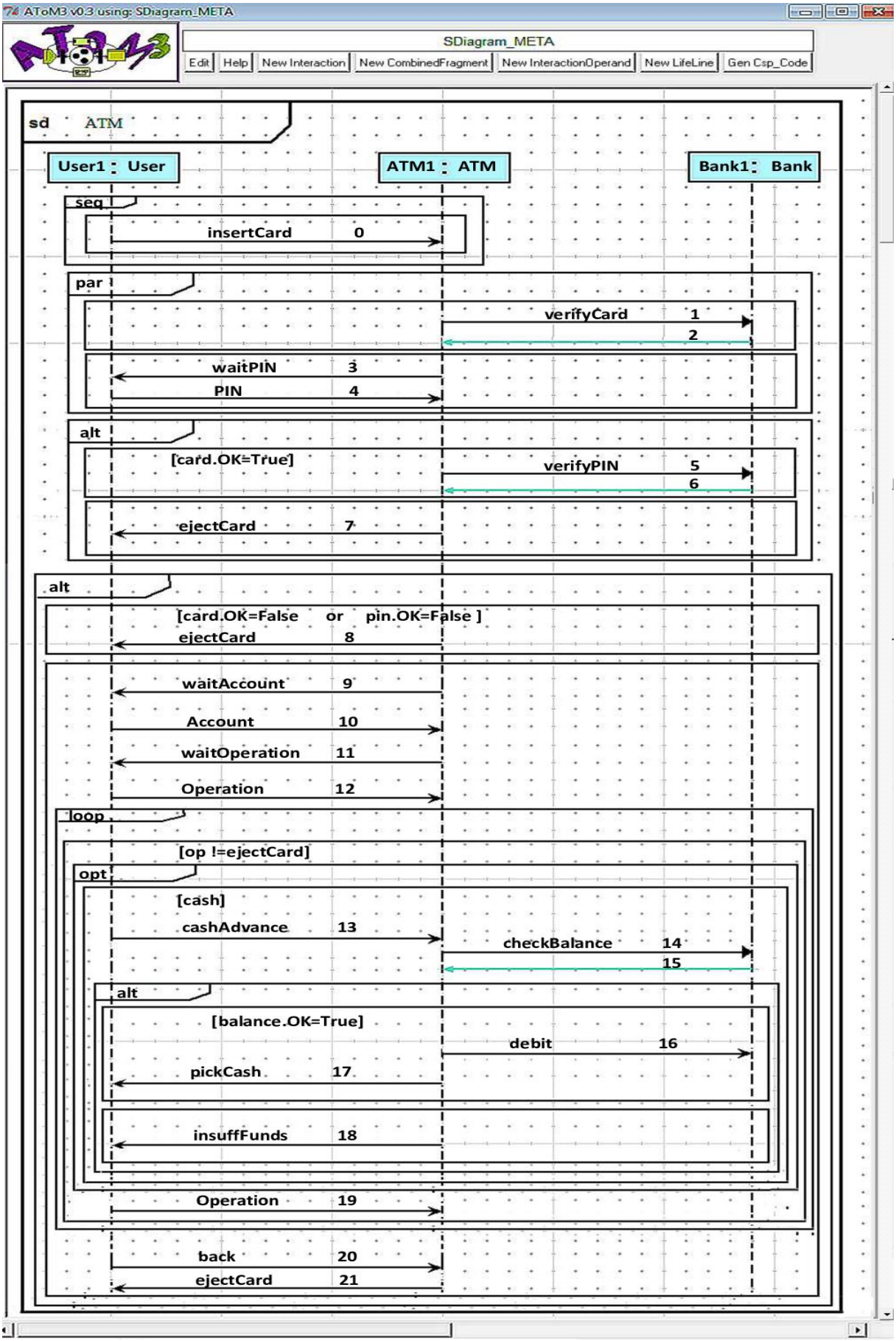


Figure 43. UML 2.0 SD d'ATM

5.2.2 Modélisation en UML 2.0 STM

La Figure 44 illustre le comportement du système modélisé en utilisant UML 2.0 STM. Tout d'abord, l'ATM est dans l'état *WaitingForCard* en attente d'interaction avec l'utilisateur. Lorsqu'une carte est insérée, l'événement *insertCard* fait passer l'ATM vers un état composé *VerifyingAndWaiting* pour vérifier la validité de la carte et attendre le code PIN (exécution parallèle de deux régions). Un choix interne est appliqué en sortant de cet état pour éjecter la carte si elle n'était pas valide (état *EjectingCard*) ou vérifier le code PIN dans le cas contraire (état *VerifyingPIN)*. Après la réception de l'action de sortie *verifyPINAck*, un deuxième choix interne est appliqué pour éjecter la carte si (*PIN.OK=False*) ou changer l'état vers l'état composé *ChoosingOperation* dans le cas contraire. Ce dernier état permet l'exécution de plusieurs opérations avant l'éjection de la carte. Le système procède comme suit : 1. le système attendre une opération *WaitingAccount*, 2. L'introduction d'une opération par l'utilisateur (action *Operation*) permet de faire un choix interne pour la traitée (si (*Operation != ejectCard*) alors passer à l'état composé *PerformingOperation*, sinon éjecter la carte. 3. Exécution de l'opération choisie. 4. Revenir à l'état *WaitingAccount* pour exécuter une nouvelle opération. 5. Ejecter la carte et revenir à l'état *WaitingForCard*.

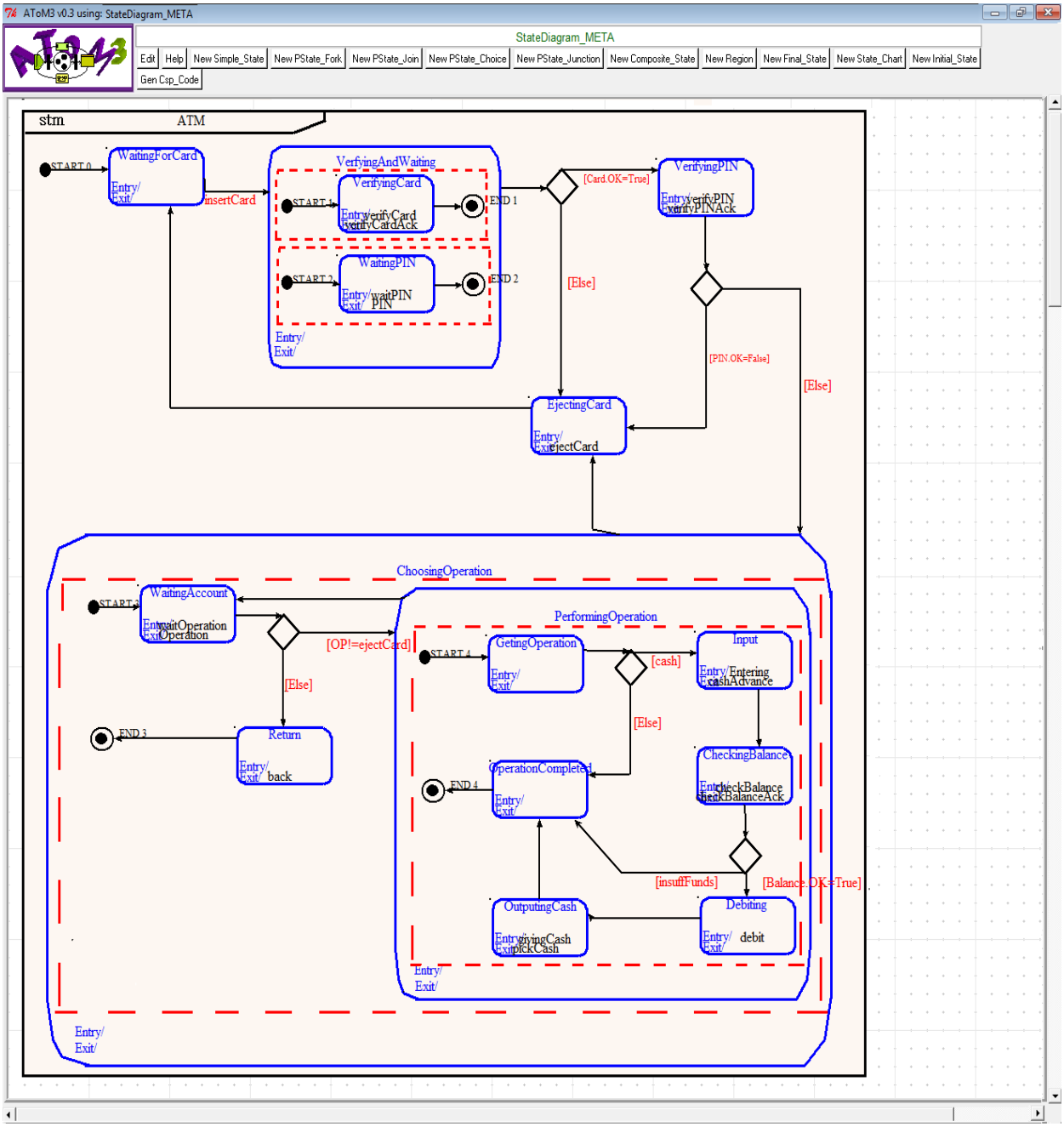


Figure 44. UML 2.0 STM d'ATM

5.2.3 Modélisation en UML 2.0 CD

La Figure 45 montre la structure statique du système ATM en utilisant UML 2.0 CD. Chaque objet dans le diagramme de séquence devient une classe dans le diagramme de classe, et chaque message entre deux lignes de vie devient une opération dans la classe correspondante à la ligne de vie destination.

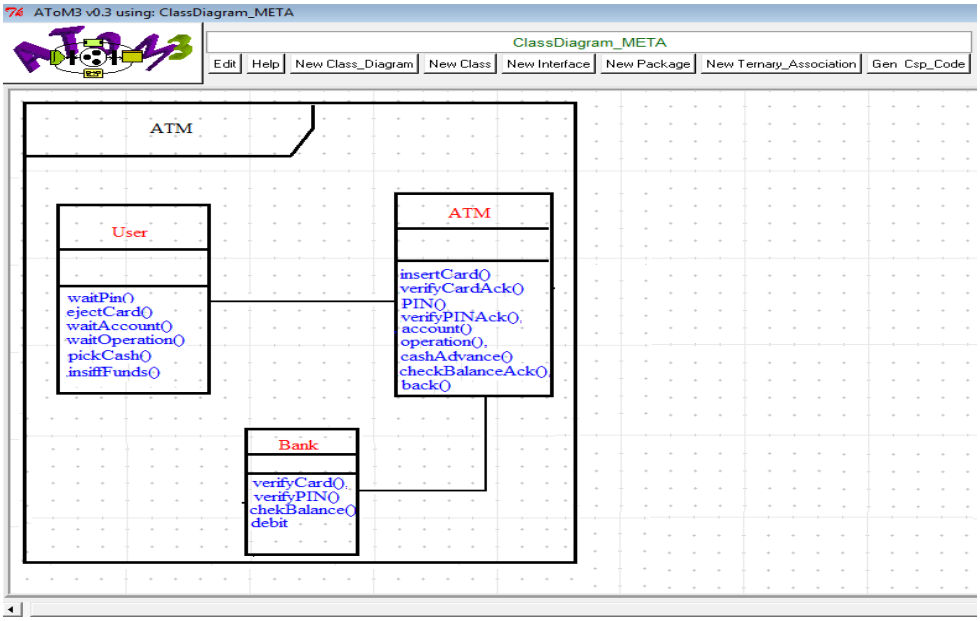


Figure 45. UML 2.0 CD d'ATM

5.3 TRANSFORMATION AUTOMATIQUE DES DIAGRAMMES UML 2.0 EN CSP

Dans cette étape, nous devons transformer les diagrammes UML modélisant la structure et le comportement de l'ATM en spécifications CSP équivalentes, afin de les vérifier dans l'étape suivante. Les étapes d'exécution des trois grammaires de graphes sont présentées dans la Figure 46. Les Figures 47, 48, 49, 50, 51, 52, 53 et 54 présentent la spécification CSP générée à partir d'UML 2.0 SD pour le système ATM. Il est important de noter que les assertions spécifient les propriétés à vérifier, et elles sont ajoutées manuellement après la génération du code CSP.

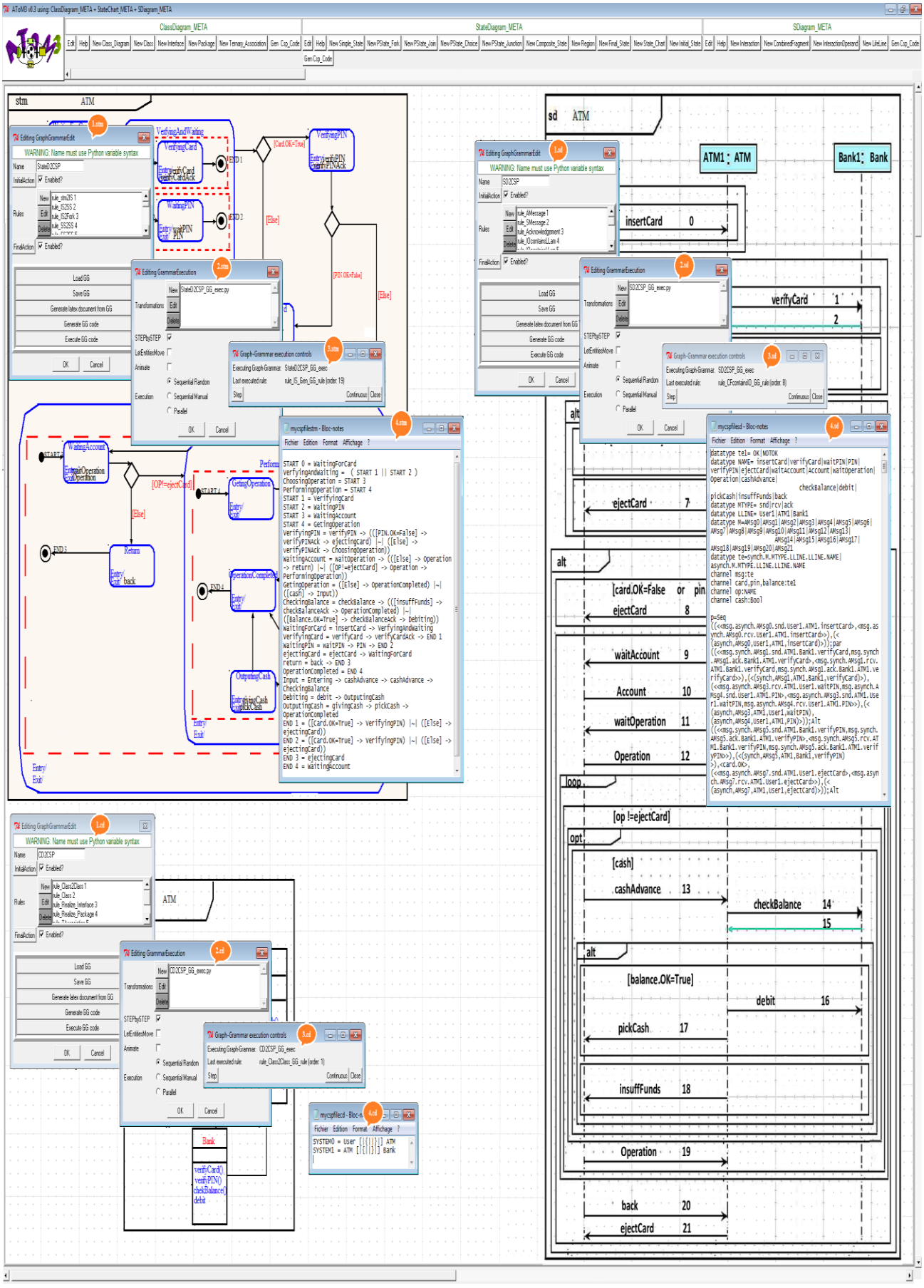


Figure 46. Étapes d'exécution des trois grammaires de graphes

```
datatype te1= OK|NOTOK
datatype NAME=
insertCard|verifyCard|waitPIN|PIN|verifyPIN|ejectCard|waitAccount|Account|waitOperation|Operation|cashAdvance|
            checkBalance|debit|pickCash|insuffFunds|back
datatype MTYPE= snd|rcv|ack
datatype LLINE= User1|ATM1|Bank1
datatype M=AMsg0|AMsg1|AMsg2|AMsg3|AMsg4|AMsg5|AMsg6|AMsg7|AMsg8|AMsg9|AMsg10|AMsg11|AMsg12|AMsg13|
        AMsg14|AMsg15|AMsg16|AMsg17|AMsg18|AMsg19|AMsg20|AMsg21
datatype te=synch.M.MTYPE.LLINE.LLINE.NAME| asynch.M.MTYPE.LLINE.LLINE.NAME
channel msg:te
channel card,pin,balance:te1
channel op:NAME
channel cash:Bool

p=Seq((<<msg.asynch.AMsg0.snd.User1.ATM1.insertCard>,<msg.asynch.AMsg0.rcv.User1.ATM1.insertCard>>),(<(asynch,AM
sg0,User1,ATM1,insertCard)>));par((<<msg.synch.AMsg1.snd.ATM1.Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.ver
ifyCard>,<msg.synch.AMsg1.rcv.ATM1.Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard>>),(<(synch,AMsg1,A
TM1,Bank1,verifyCard)>),(<<msg.asynch.AMsg3.rcv.ATM1.User1.waitPIN,msg.asynch.AMsg4.snd.User1.ATM1.PIN>,<msg.asy
nch.AMsg3.snd.ATM1.User1.waitPIN,msg.asynch.AMsg4.rcv.User1.ATM1.PIN>>),(<(asynch,AMsg3,ATM1,User1,waitPIN),(asy
nch,AMsg4,User1,ATM1,PIN)>));Alt((<<msg.synch.AMsg5.snd.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.veri
fyPIN>,<msg.synch.AMsg5.rcv.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN>>),(<(synch,AMsg5,ATM
1,Bank1,verifyPIN)>),<card.OK>,(<<msg.asynch.AMsg7.snd.ATM1.User1.ejectCard>,<msg.asynch.AMsg7.rcv.ATM1.User1.eject
Card>>),(<(asynch,AMsg7,ATM1,User1,ejectCard)>));Alt((<<msg.asynch.AMsg8.snd.ATM1.User1.ejectCard>,<msg.asynch.A
Msg8.rcv.ATM1.User1.ejectCard>>),(<(asynch,AMsg8,ATM1,User1,ejectCard)>),<card.NOTOK,pin.NOTOK>,(<<msg.asynch.A
Msg9.rcv.ATM1.User1.waitAccount,msg.asynch.AMsg10.snd.User1.ATM1.Account,msg.asynch.AMsg11.rcv.ATM1.User1.waitO
peration,msg.asynch.AMsg12.snd.User1.ATM1.Operation>,<msg.asynch.AMsg9.snd.ATM1.User1.waitAccount,msg.asynch.AM
sg10.rcv.User1.ATM1.Account,msg.asynch.AMsg11.snd.ATM1.User1.waitOperatio n,msg.asynch.AMsg12.rcv.User1.ATM1.Oper
ation>>),(<(asynch,AMsg9,ATM1,User1,waitAccount),(asynch,AMsg10,User1,ATM1,Account),(asynch,AMsg11,ATM1,User1,w
aitOperation),(asynch,AMsg12,User1,ATM1,Operation)>))

par((<<msg.synch.AMsg1.snd.ATM1.Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard>,<msg.synch.AMsg1.rcv.
ATM1.Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard>>),(<(synch,AMsg1,ATM1,Bank1,verifyCard)>),(<<ms
g.asynch.AMsg3.rcv.ATM1.User1.waitPIN,msg.asynch.AMsg4.snd.User1.ATM1.PIN>,<msg.asynch.AMsg3.snd.ATM1.User1.wait
PIN,msg.asynch.AMsg4.rcv.User1.ATM1.PIN>>),(<(asynch,AMsg3,ATM1,User1,waitPIN),(asynch,AMsg4,User1,ATM1,PIN)>))=
Seq((<<msg.synch.AMsg1.snd.ATM1.Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard>,<msg.synch.AMsg1.rcv
.ATM1.Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard>>),(<(synch,AMsg1,ATM1,Bank1,verifyCard)>)) |||
Seq((<<msg.asynch.AMsg3.rcv.ATM1.User1.waitPIN,msg.asynch.AMsg4.snd.User1.ATM1.PIN>,<msg.asynch.AMsg3.snd.ATM1.
User1.waitPIN,msg.asynch.AMsg4.rcv.User1.ATM1.PIN>>),(<(asynch,AMsg3,ATM1,User1,waitPIN),(asynch,AMsg4,User1,ATM
1,PIN)>))

Alt((<<msg.synch.AMsg5.snd.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN>,<msg.synch.AMsg5.rcv.A
TM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN>>),(<(synch,AMsg5,ATM1,Bank1,verifyPIN)>),<card.OK>,
(<<msg.asynch.AMsg7.snd.ATM1.User1.ejectCard>,<msg.asynch.AMsg7.rcv.ATM1.User1.ejectCard>>),(<(asynch,AMsg7,ATM
1,User1,ejectCard)>))= if f(card.OK) then
Seq((<<msg.synch.AMsg5.snd.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN>,<msg.synch.AMsg5.rcv.A
TM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN>>),(<(synch,AMsg5,ATM1,Bank1,verifyPIN)>)) else
Seq((<<msg.asynch.AMsg7.snd.ATM1.User1.ejectCard>,<msg.asynch.AMsg7.rcv.ATM1.User1.ejectCard>>),(<(asynch,AMsg7,A
TM1,User1,ejectCard)>))

Alt((<<msg.asynch.AMsg8.snd.ATM1.User1.ejectCard>,<msg.asynch.AMsg8.rcv.ATM1.User1.ejectCard>>),(<(asynch,AMsg8,A
TM1,User1,ejectCard)>),<card.NOTOK,pin.NOTOK>,(<<msg.asynch.AMsg9.rcv.ATM1.User1.waitAccount,msg.asynch.AMsg10.
snd.User1.ATM1.Account,msg.asynch.AMsg11.rcv.ATM1.User1.waitOperation,msg.asynch.AMsg12.snd.User1.ATM1.Operatio
n>,<msg.asynch.AMsg9.snd.ATM1.User1.waitAccount,msg.asynch.AMsg10.rcv.User1.ATM1.Account,msg.asynch.AMsg11.snd.
ATM1.User1.waitOperation,msg.asynch.AMsg12.rcv.User1.ATM1.Operation>>),(<(asynch,AMsg9,ATM1,User1,waitAccount),(
asynch,AMsg10,User1,ATM1,Account),(asynch,AMsg11,ATM1,User1,waitOperation),(asynch,AMsg12,User1,ATM1,Operation)
>))= if f(card.NOTOK) or f(pin.NOTOK) then
Seq((<<msg.asynch.AMsg8.snd.ATM1.User1.ejectCard>,<msg.asynch.AMsg8.rcv.ATM1.User1.ejectCard>>),(<(asynch,AMsg8,A
TM1,User1,ejectCard)>)) else
Seq((<<msg.asynch.AMsg9.rcv.ATM1.User1.waitAccount,msg.asynch.AMsg10.snd.User1.ATM1.Account,msg.asynch.AMsg11.r
cv.ATM1.User1.waitOperation,msg.asynch.AMsg12.snd.User1.ATM1.Operation>,<msg.asynch.AMsg9.snd.ATM1.User1.waitAc
count,msg.asynch.AMsg10.rcv.User1.ATM1.Account,msg.asynch.AMsg11.snd.ATM1.User1.waitOperation,msg.asynch.AMsg12.
rcv.User1.ATM1.Operation>>),(<(asynch,AMsg9,ATM1,User1,waitAccount),(asynch,AMsg10,User1,ATM1,Account),(asynch,A
Msg11,ATM1,User1,waitOperation),(asynch,AMsg12,User1,ATM1,Operation)>));Loop((<<msg.asynch.AMsg13.snd.User1.AT
M1.cashAdvance>,<msg.asynch.AMsg13.rcv.User1.ATM1.cashAdvance,msg.synch.AMsg14.snd.ATM1.Bank1.checkBalance,ms
g.synch.AMsg14.ack.Bank1.ATM1.checkBalance>,<msg.synch.AMsg14.rcv.ATM1.Bank1.checkBalance,msg.synch.AMsg14.ack.
Bank1.ATM1.checkBalance>>),(<(asynch,AMsg13,User1,ATM1,cashAdvance),(synch,AMsg14,ATM1,Bank1,checkBalance)>),o
p);Seq((<<msg.asynch.AMsg20.snd.User1.ATM1.back,msg.asynch.AMsg21.rcv.ATM1.User1.ejectCard>,<msg.asynch.AMsg20.r
cv.User1.ATM1.back,msg.asynch.AMsg21.snd.ATM1.User1.ejectCard>>),(<(asynch,AMsg20,User1,ATM1,back),(asynch,AMsg2
1,ATM1,User1,ejectCard)>))
```

Figure 47. Code CSP_M générée à partir d'UML 2.0 SD pour le système ATM (partie 1)

```
Alt((<<msg.asynch.AMsg17.rcv.ATM1.User1.pickCash>,<msg.asynch.AMsg16.snd.ATM1.Bank1.debit,msg.asynch.AMsg17.snd.
ATM1.User1.pickCash>,<msg.asynch.AMsg16.rcv.ATM1.Bank1.debit>>),(<(asynch,AMsg16,ATM1,Bank1,debit),(asynch,AMsg
17,ATM1,User1,pickCash)>),<balance.OK>,(<<msg.asynch.AMsg18.snd.ATM1.User1.insuffFunds>,<msg.asynch.AMsg18.rcv.A
TM1.User1.insuffFunds>>),(<(asynch,AMsg18,ATM1,User1,insuffFunds)>))
= if f(balance.OK) then
Seq((<<msg.asynch.AMsg17.rcv.ATM1.User1.pickCash>,<msg.asynch.AMsg16.snd.ATM1.Bank1.debit,msg.asynch.AMsg17.snd.
ATM1.User1.pickCash>,<msg.asynch.AMsg16.rcv.ATM1.Bank1.debit>>),(<(asynch,AMsg16,ATM1,Bank1,debit),(asynch,AMsg
17,ATM1,User1,pickCash)>)) else
Seq((<<msg.asynch.AMsg18.snd.ATM1.User1.insuffFunds>,<msg.asynch.AMsg18.rcv.ATM1.User1.insuffFunds>>),(<(asynch,A
Msg18,ATM1,User1,insuffFunds)>))

f(card.OK)=True
f(card.NOTOK)=False
f(pin.OK)=True
f(pin.NOTOK)=False
f(op.ejectCard)=False
f(cash.true)=True
f(cash.false)=False
f(balance.OK)= True
f(balance.NOTOK)= False

Loop((<<msg.asynch.AMsg13.snd.User1.ATM1.cashAdvance>,<msg.asynch.AMsg13.rcv.User1.ATM1.cashAdvance,msg.synch.
AMsg14.snd.ATM1.Bank1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>,<msg.synch.AMsg14.rcv.ATM1.Ba
nk1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>>),(<(asynch,AMsg13,User1,ATM1,cashAdvance),(synch
,AMsg14,ATM1,Bank1,checkBalance)>),op)= if f(op.ejectCard) then
Opt((<<msg.asynch.AMsg13.snd.User1.ATM1.cashAdvance>,<msg.asynch.AMsg13.rcv.User1.ATM1.cashAdvance,msg.synch.A
Msg14.snd.ATM1.Bank1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>,<msg.synch.AMsg14.rcv.ATM1.Ban
k1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>>),(<(asynch,AMsg13,User1,ATM1,cashAdvance),(synch,
AMsg14,ATM1,Bank1,checkBalance)>),cash.true);Seq((<<msg.asynch.AMsg19.snd.User1.ATM1.Operation>,<msg.asynch.AMs
g19.rcv.User1.ATM1.Operation>>),(<(asynch,AMsg19,User1,ATM1,Operation)>));
Loop((<<msg.asynch.AMsg13.snd.User1.ATM1.cashAdvance>,<msg.asynch.AMsg13.rcv.User1.ATM1.cashAdvance,msg.synch.
AMsg14.snd.ATM1.Bank1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>,<msg.synch.AMsg14.rcv.ATM1.Ba
nk1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>>),(<(asynch,AMsg13,User1,ATM1,cashAdvance),(synch
,AMsg14,ATM1,Bank1,checkBalance)>),op)else SKIP

Opt((<<msg.asynch.AMsg13.snd.User1.ATM1.cashAdvance>,<msg.asynch.AMsg13.rcv.User1.ATM1.cashAdvance,msg.synch.A
Msg14.snd.ATM1.Bank1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>,<msg.synch.AMsg14.rcv.ATM1.Ban
k1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>>),(<(asynch,AMsg13,User1,ATM1,cashAdvance),(synch,
AMsg14,ATM1,Bank1,checkBalance)>),cash.true)= if f(cash.true) then
Seq((<<msg.asynch.AMsg13.snd.User1.ATM1.cashAdvance>,<msg.asynch.AMsg13.rcv.User1.ATM1.cashAdvance,msg.synch.A
Msg14.snd.ATM1.Bank1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>,<msg.synch.AMsg14.rcv.ATM1.Ban
k1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>>),(<(asynch,AMsg13,User1,ATM1,cashAdvance),(synch,
AMsg14,ATM1,Bank1,checkBalance)>));Alt((<<msg.asynch.AMsg17.rcv.ATM1.User1.pickCash>,<msg.asynch.AMsg16.snd.AT
M1.Bank1.debit,msg.asynch.AMsg17.snd.ATM1.User1.pickCash>,<msg.asynch.AMsg16.rcv.ATM1.Bank1.debit>>),(<(asynch,A
Msg16,ATM1,Bank1,debit),(asynch,AMsg17,ATM1,User1,pickCash)>),<balance.OK>,(<<msg.asynch.AMsg18.snd.ATM1.User1.i
nsuffFunds>,<msg.asynch.AMsg18.rcv.ATM1.User1.insuffFunds>>),(<(asynch,AMsg18,ATM1,User1,insuffFunds)>))
else SKIP
Seq((<<msg.asynch.AMsg0.snd.User1.ATM1.insertCard>,<msg.asynch.AMsg0.rcv.User1.ATM1.insertCard>>),(<(asynch,AMsg0
,User1,ATM1,insertCard)>))=Lifelines(<<msg.asynch.AMsg0.snd.User1.ATM1.insertCard>,<msg.asynch.AMsg0.rcv.User1.ATM
1.insertCard>>)[alphaLifelines(<<msg.asynch.AMsg0.snd.User1.ATM1.insertCard>,<msg.asynch.AMsg0.rcv.User1.ATM1.inser
tCard>>)||alphaMessages(<(asynch,AMsg0,User1,ATM1,insertCard)>)]Messages(<(asynch,AMsg0,User1,ATM1,insertCard)>)

Seq((<<msg.synch.AMsg1.snd.ATM1.Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard>,<msg.synch.AMsg1.rcv.
ATM1.Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard>>),(<(synch,AMsg1,ATM1,Bank1,verifyCard)>))=
Lifelines(<<msg.synch.AMsg1.snd.ATM1.Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard>,<msg.synch.AMsg1
.rcv.ATM1.Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard>>)[alphaLifelines(<<msg.synch.AMsg1.snd.ATM1.
Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard>,<msg.synch.AMsg1.rcv.ATM1.Bank1.verifyCard,msg.synch.A
Msg1.ack.Bank1.ATM1.verifyCard>>)||alphaMessages(<(synch,AMsg1,ATM1,Bank1,verifyCard)>)]Messages(<(synch,AMsg1,A
TM1,Bank1,verifyCard)>)
Seq((<<msg.asynch.AMsg3.rcv.ATM1.User1.waitPIN,msg.asynch.AMsg4.snd.User1.ATM1.PIN>,<msg.asynch.AMsg3.snd.ATM1.
User1.waitPIN,msg.asynch.AMsg4.rcv.User1.ATM1.PIN>>),(<(asynch,AMsg3,ATM1,User1,waitPIN),(asynch,AMsg4,User1,ATM
1,PIN)>))
=Lifelines(<<msg.asynch.AMsg3.rcv.ATM1.User1.waitPIN,msg.asynch.AMsg4.snd.User1.ATM1.PIN>,<msg.asynch.AMsg3.snd.A
TM1.User1.waitPIN,msg.asynch.AMsg4.rcv.User1.ATM1.PIN>>)[alphaLifelines(<<msg.asynch.AMsg3.rcv.ATM1.User1.waitPIN,
msg.asynch.AMsg4.snd.User1.ATM1.PIN>,<msg.asynch.AMsg3.snd.ATM1.User1.waitPIN,msg.asynch.AMsg4.rcv.User1.ATM1.P
IN>>)||alphaMessages(<(asynch,AMsg3,ATM1,User1,waitPIN),(asynch,AMsg4,User1,ATM1,PIN)>)]Messages(<(asynch,AMsg3
,ATM1,User1,waitPIN),(asynch,AMsg4,User1,ATM1,PIN)>)
```

Figure 48. Code CSP_M générée à partir d'UML 2.0 SD pour le système ATM (partie 2)

```
Seq((<<msg.synch.AMsg5.snd.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN>,<msg.synch.AMsg5.rcv.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN>>),(<(synch,AMsg5,ATM1,Bank1,verifyPIN)>))=
Lifelines(<<msg.synch.AMsg5.snd.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN>,<msg.synch.AMsg5.rcv.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN>>)[alphaLifelines(<<msg.synch.AMsg5.snd.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN>,<msg.synch.AMsg5.rcv.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN>>)||alphaMessages(<(synch,AMsg5,ATM1,Bank1,verifyPIN)>)]Messages(<(synch,AMsg5,ATM1,Bank1,verifyPIN)>)

Seq((<<msg.asynch.AMsg7.snd.ATM1.User1.ejectCard>,<msg.asynch.AMsg7.rcv.ATM1.User1.ejectCard>>),(<(asynch,AMsg7,ATM1,User1,ejectCard)>))
=
Lifelines(<<msg.asynch.AMsg7.snd.ATM1.User1.ejectCard>,<msg.asynch.AMsg7.rcv.ATM1.User1.ejectCard>>)[alphaLifelines(<<msg.asynch.AMsg7.snd.ATM1.User1.ejectCard>,<msg.asynch.AMsg7.rcv.ATM1.User1.ejectCard>>)||alphaMessages(<(asynch,AMsg7,ATM1,User1,ejectCard)>)]Messages(<(asynch,AMsg7,ATM1,User1,ejectCard)>)

Seq((<<msg.asynch.AMsg8.snd.ATM1.User1.ejectCard>,<msg.asynch.AMsg8.rcv.ATM1.User1.ejectCard>>),(<(asynch,AMsg8,ATM1,User1,ejectCard)>))=
Lifelines(<<msg.asynch.AMsg8.snd.ATM1.User1.ejectCard>,<msg.asynch.AMsg8.rcv.ATM1.User1.ejectCard>>)[alphaLifelines(<<msg.asynch.AMsg8.snd.ATM1.User1.ejectCard>,<msg.asynch.AMsg8.rcv.ATM1.User1.ejectCard>>)||alphaMessages(<(asynch,AMsg8,ATM1,User1,ejectCard)>)]Messages(<(asynch,AMsg8,ATM1,User1,ejectCard)>)

Seq((<<msg.asynch.AMsg9.rcv.ATM1.User1.waitAccount,msg.asynch.AMsg10.snd.User1.ATM1.Account,msg.asynch.AMsg11.rcv.ATM1.User1.waitOperation,msg.asynch.AMsg12.snd.User1.ATM1.Operation>,<msg.asynch.AMsg9.snd.ATM1.User1.waitAccount,msg.asynch.AMsg10.rcv.User1.ATM1.Account,msg.asynch.AMsg11.snd.ATM1.User1.waitOperation,msg.asynch.AMsg12.rcv.User1.ATM1.Operation>>),(<(asynch,AMsg9,ATM1,User1,waitAccount),(asynch,AMsg10,User1,ATM1,Account),(asynch,AMsg11,ATM1,User1,waitOperation),(asynch,AMsg12,User1,ATM1,Operation)>))=
Lifelines(<<msg.asynch.AMsg9.rcv.ATM1.User1.waitAccount,msg.asynch.AMsg10.snd.User1.ATM1.Account,msg.asynch.AMsg11.rcv.ATM1.User1.waitOperation,msg.asynch.AMsg12.snd.User1.ATM1.Operation>,<msg.asynch.AMsg9.snd.ATM1.User1.waitAccount,msg.asynch.AMsg10.rcv.User1.ATM1.Account,msg.asynch.AMsg11.snd.ATM1.User1.waitOperation,msg.asynch.AMsg12.rcv.User1.ATM1.Operation>>)[alphaLifelines(<<msg.asynch.AMsg9.rcv.ATM1.User1.waitAccount,msg.asynch.AMsg10.snd.User1.ATM1.Account,msg.asynch.AMsg11.rcv.ATM1.User1.waitOperation,msg.asynch.AMsg12.snd.User1.ATM1.Operation>,<msg.asynch.AMsg9.snd.ATM1.User1.waitAccount,msg.asynch.AMsg10.rcv.User1.ATM1.Account,msg.asynch.AMsg11.snd.ATM1.User1.waitOperation,msg.asynch.AMsg12.rcv.User1.ATM1.Operation>>)||alphaMessages(<(asynch,AMsg9,ATM1,User1,waitAccount),(asynch,AMsg10,User1,ATM1,Account),(asynch,AMsg11,ATM1,User1,waitOperation),(asynch,AMsg12,User1,ATM1,Operation)>)]Messages(<(asynch,AMsg9,ATM1,User1,waitAccount),(asynch,AMsg10,User1,ATM1,Account),(asynch,AMsg11,ATM1,User1,waitOperation),(asynch,AMsg12,User1,ATM1,Operation)>)
Seq((<<msg.asynch.AMsg13.snd.User1.ATM1.cashAdvance>,<msg.asynch.AMsg13.rcv.User1.ATM1.cashAdvance,msg.synch.AMsg14.snd.ATM1.Bank1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>,<msg.synch.AMsg14.rcv.ATM1.Bank1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>>),(<(asynch,AMsg13,User1,ATM1,cashAdvance),(synch,AMsg14,ATM1,Bank1,checkBalance)>))=Lifelines(<<msg.asynch.AMsg13.snd.User1.ATM1.cashAdvance>,<msg.asynch.AMsg13.rcv.User1.ATM1.cashAdvance,msg.synch.AMsg14.snd.ATM1.Bank1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>,<msg.synch.AMsg14.rcv.ATM1.Bank1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>>)[alphaLifelines(<<msg.asynch.AMsg13.snd.User1.ATM1.cashAdvance>,<msg.asynch.AMsg13.rcv.User1.ATM1.cashAdvance,msg.synch.AMsg14.snd.ATM1.Bank1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>,<msg.synch.AMsg14.rcv.ATM1.Bank1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>>)||alphaMessages(<(asynch,AMsg13,User1,ATM1,cashAdvance),(synch,AMsg14,ATM1,Bank1,checkBalance)>)]Messages(<(asynch,AMsg13,User1,ATM1,cashAdvance),(synch,AMsg14,ATM1,Bank1,checkBalance)>)

Seq((<<msg.asynch.AMsg17.rcv.ATM1.User1.pickCash>,<msg.asynch.AMsg16.snd.ATM1.Bank1.debit,msg.asynch.AMsg17.snd.ATM1.User1.pickCash>,<msg.asynch.AMsg16.rcv.ATM1.Bank1.debit>>),(<(asynch,AMsg16,ATM1,Bank1,debit),(asynch,AMsg17,ATM1,User1,pickCash)>))=
Lifelines(<<msg.asynch.AMsg17.rcv.ATM1.User1.pickCash>,<msg.asynch.AMsg16.snd.ATM1.Bank1.debit,msg.asynch.AMsg17.snd.ATM1.User1.pickCash>,<msg.asynch.AMsg16.rcv.ATM1.Bank1.debit>>)[alphaLifelines(<<msg.asynch.AMsg17.rcv.ATM1.User1.pickCash>,<msg.asynch.AMsg16.snd.ATM1.Bank1.debit,msg.asynch.AMsg17.snd.ATM1.User1.pickCash>,<msg.asynch.AMsg16.rcv.ATM1.Bank1.debit>>)||alphaMessages(<(asynch,AMsg16,ATM1,Bank1,debit),(asynch,AMsg17,ATM1,User1,pickCash)>)]Messages(<(asynch,AMsg16,ATM1,Bank1,debit),(asynch,AMsg17,ATM1,User1,pickCash)>)

Seq((<<msg.asynch.AMsg18.snd.ATM1.User1.insuffFunds>,<msg.asynch.AMsg18.rcv.ATM1.User1.insuffFunds>>),(<(asynch,AMsg18,ATM1,User1,insuffFunds)>))=Lifelines(<<msg.asynch.AMsg18.snd.ATM1.User1.insuffFunds>,<msg.asynch.AMsg18.rcv.ATM1.User1.insuffFunds>>)[alphaLifelines(<<msg.asynch.AMsg18.snd.ATM1.User1.insuffFunds>,<msg.asynch.AMsg18.rcv.ATM1.User1.insuffFunds>>)||alphaMessages(<(asynch,AMsg18,ATM1,User1,insuffFunds)>)]Messages(<(asynch,AMsg18,ATM1,User1,insuffFunds)>)

Seq((<<msg.asynch.AMsg19.snd.User1.ATM1.Operation>,<msg.asynch.AMsg19.rcv.User1.ATM1.Operation>>),(<(asynch,AMsg19,User1,ATM1,Operation)>))=Lifelines(<<msg.asynch.AMsg19.snd.User1.ATM1.Operation>,<msg.asynch.AMsg19.rcv.User1.ATM1.Operation>>)[alphaLifelines(<<msg.asynch.AMsg19.snd.User1.ATM1.Operation>,<msg.asynch.AMsg19.rcv.User1.ATM1.Operation>>)||alphaMessages(<(asynch,AMsg19,User1,ATM1,Operation)>)]Messages(<(asynch,AMsg19,User1,ATM1,Operation)>)
```

Figure 49. Code CSPм générée à partir d'UML 2.0 SD pour le système ATM (partie 3)

Seq((<<msg.asynch.AMsg20.snd.User1.ATM1.back,msg.asynch.AMsg21.rcv.ATM1.User1.ejectCard>,<msg.asynch.AMsg20.rcv.User1.ATM1.back,msg.asynch.AMsg21.snd.ATM1.User1.ejectCard>>),(<(asynch,AMsg20,User1,ATM1,back),(asynch,AMsg21,ATM1,User1,ejectCard)>))=Lifelines(<<msg.asynch.AMsg20.snd.User1.ATM1.back,msg.asynch.AMsg21.rcv.ATM1.User1.ejectCard>,<msg.asynch.AMsg20.rcv.User1.ATM1.back,msg.asynch.AMsg21.snd.ATM1.User1.ejectCard>>)[alphaLifelines(<<msg.asynch.AMsg20.snd.User1.ATM1.back,msg.asynch.AMsg21.rcv.ATM1.User1.ejectCard>,<msg.asynch.AMsg20.rcv.User1.ATM1.back,msg.asynch.AMsg21.snd.ATM1.User1.ejectCard>>)||alphaMessages(<(asynch,AMsg20,User1,ATM1,back),(asynch,AMsg21,ATM1,User1,ejectCard)>)]Messages(<(asynch,AMsg20,User1,ATM1,back),(asynch,AMsg21,ATM1,User1,ejectCard)>)

Lifelines(<<msg.asynch.AMsg0.snd.User1.ATM1.insertCard>,<msg.asynch.AMsg0.rcv.User1.ATM1.insertCard>>)=||line:{<msg.asynch.AMsg0.snd.User1.ATM1.insertCard>,<msg.asynch.AMsg0.rcv.User1.ATM1.insertCard>}@[set(line)]PrefixComposition(line)

Lifelines(<<msg.synch.AMsg1.snd.ATM1.Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard>,<msg.synch.AMsg1.rcv.ATM1.Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard>>)=||line:{<msg.synch.AMsg1.snd.ATM1.Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard>,<msg.synch.AMsg1.rcv.ATM1.Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard>}@[set(line)]PrefixComposition(line)

Lifelines(<<msg.asynch.AMsg3.rcv.ATM1.User1.waitPIN,msg.asynch.AMsg4.snd.User1.ATM1.PIN>,<msg.asynch.AMsg3.snd.ATM1.User1.waitPIN,msg.asynch.AMsg4.rcv.User1.ATM1.PIN>>)=||line:{<msg.asynch.AMsg3.rcv.ATM1.User1.waitPIN,msg.asynch.AMsg4.snd.User1.ATM1.PIN>,<msg.asynch.AMsg3.snd.ATM1.User1.waitPIN,msg.asynch.AMsg4.rcv.User1.ATM1.PIN>}@[set(line)]PrefixComposition(line)

Lifelines(<<msg.synch.AMsg5.snd.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN>,<msg.synch.AMsg5.rcv.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN>>)=||line:{<msg.synch.AMsg5.snd.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN>,<msg.synch.AMsg5.rcv.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN>}@[set(line)]PrefixComposition(line)

Lifelines(<<msg.asynch.AMsg7.snd.ATM1.User1.ejectCard>,<msg.asynch.AMsg7.rcv.ATM1.User1.ejectCard>>)=||line:{<msg.asynch.AMsg7.snd.ATM1.User1.ejectCard>,<msg.asynch.AMsg7.rcv.ATM1.User1.ejectCard>}@[set(line)]PrefixComposition(line)

Lifelines(<<msg.asynch.AMsg8.snd.ATM1.User1.ejectCard>,<msg.asynch.AMsg8.rcv.ATM1.User1.ejectCard>>)=||line:{<msg.asynch.AMsg8.snd.ATM1.User1.ejectCard>,<msg.asynch.AMsg8.rcv.ATM1.User1.ejectCard>}@[set(line)]PrefixComposition(line)
Lifelines(<<msg.asynch.AMsg9.rcv.ATM1.User1.waitAccount,msg.asynch.AMsg10.snd.User1.ATM1.Account,msg.asynch.AMsg11.rcv.ATM1.User1.waitOperation,msg.asynch.AMsg12.snd.User1.ATM1.Operation>,<msg.asynch.AMsg9.snd.ATM1.User1.waitAccount,msg.asynch.AMsg10.rcv.User1.ATM1.Account,msg.asynch.AMsg11.snd.ATM1.User1.waitOperation,msg.asynch.AMsg12.rcv.User1.ATM1.Operation>>)=||line:{<msg.asynch.AMsg9.rcv.ATM1.User1.waitAccount,msg.asynch.AMsg10.snd.User1.ATM1.Account,msg.asynch.AMsg11.rcv.ATM1.User1.waitOperation,msg.asynch.AMsg12.snd.User1.ATM1.Operation>,<msg.asynch.AMsg9.snd.ATM1.User1.waitAccount,msg.asynch.AMsg10.rcv.User1.ATM1.Account,msg.asynch.AMsg11.snd.ATM1.User1.waitOperation,msg.asynch.AMsg12.rcv.User1.ATM1.Operation>}@[set(line)]PrefixComposition(line)

Lifelines(<<msg.asynch.AMsg13.snd.User1.ATM1.cashAdvance>,<msg.asynch.AMsg13.rcv.User1.ATM1.cashAdvance,msg.synch.AMsg14.snd.ATM1.Bank1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>,<msg.synch.AMsg14.rcv.ATM1.Bank1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>>)=||line:{<msg.asynch.AMsg13.snd.User1.ATM1.cashAdvance>,<msg.asynch.AMsg13.rcv.User1.ATM1.cashAdvance,msg.synch.AMsg14.snd.ATM1.Bank1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>,<msg.synch.AMsg14.rcv.ATM1.Bank1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>}@[set(line)]PrefixComposition(line)

Lifelines(<<msg.asynch.AMsg17.rcv.ATM1.User1.pickCash>,<msg.asynch.AMsg16.snd.ATM1.Bank1.debit,msg.asynch.AMsg17.snd.ATM1.User1.pickCash>,<msg.asynch.AMsg16.rcv.ATM1.Bank1.debit>>)=||line:{<msg.asynch.AMsg17.rcv.ATM1.User1.pickCash>,<msg.asynch.AMsg16.snd.ATM1.Bank1.debit,msg.asynch.AMsg17.snd.ATM1.User1.pickCash>,<msg.asynch.AMsg16.rcv.ATM1.Bank1.debit>}@[set(line)]PrefixComposition(line)

Lifelines(<<msg.asynch.AMsg18.snd.ATM1.User1.insuffFunds>,<msg.asynch.AMsg18.rcv.ATM1.User1.insuffFunds>>)=||line:{<msg.asynch.AMsg18.snd.ATM1.User1.insuffFunds>,<msg.asynch.AMsg18.rcv.ATM1.User1.insuffFunds>}@[set(line)]PrefixComposition(line)

Lifelines(<<msg.asynch.AMsg19.snd.User1.ATM1.Operation>,<msg.asynch.AMsg19.rcv.User1.ATM1.Operation>>)=||line:{<msg.asynch.AMsg19.snd.User1.ATM1.Operation>,<msg.asynch.AMsg19.rcv.User1.ATM1.Operation>}@[set(line)]PrefixComposition(line)

Lifelines(<<msg.asynch.AMsg20.snd.User1.ATM1.back,msg.asynch.AMsg21.rcv.ATM1.User1.ejectCard>,<msg.asynch.AMsg20.rcv.User1.ATM1.back,msg.asynch.AMsg21.snd.ATM1.User1.ejectCard>>)=||line:{<msg.asynch.AMsg20.snd.User1.ATM1.back,msg.asynch.AMsg21.rcv.ATM1.User1.ejectCard>,<msg.asynch.AMsg20.rcv.User1.ATM1.back,msg.asynch.AMsg21.snd.ATM1.User1.ejectCard>}@[set(line)]PrefixComposition(line)

alphaLifelines(<<msg.asynch.AMsg0.snd.User1.ATM1.insertCard>,<msg.asynch.AMsg0.rcv.User1.ATM1.insertCard>>)={|msg.asynch.AMsg0.snd.User1.ATM1.insertCard,msg.asynch.AMsg0.rcv.User1.ATM1.insertCard|}

Figure 50. Code CSPM générée à partir d'UML 2.0 SD pour le système ATM (partie 4)

```
alphaLifelines(<<msg.synch.AMsg1.snd.ATM1.Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard>,<msg.synch.AMsg1.rcv.ATM1.Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard>>)={|msg.synch.AMsg1.snd.ATM1.Bank1.verifyCard,msg.synch.AMsg1.rcv.ATM1.Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard|}

alphaLifelines(<<msg.asynch.AMsg3.rcv.ATM1.User1.waitPIN,msg.asynch.AMsg4.snd.User1.ATM1.PIN>,<msg.asynch.AMsg3.snd.ATM1.User1.waitPIN,msg.asynch.AMsg4.rcv.User1.ATM1.PIN>>)={|msg.asynch.AMsg3.rcv.ATM1.User1.waitPIN,msg.asynch.AMsg4.snd.User1.ATM1.PIN,msg.asynch.AMsg3.snd.ATM1.User1.waitPIN,msg.asynch.AMsg4.rcv.User1.ATM1.PIN|}

alphaLifelines(<<msg.synch.AMsg5.snd.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN>,<msg.synch.AMsg5.rcv.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN>>)={|msg.synch.AMsg5.snd.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.rcv.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN|}

alphaLifelines(<<msg.asynch.AMsg7.snd.ATM1.User1.ejectCard>,<msg.asynch.AMsg7.rcv.ATM1.User1.ejectCard>>)={|msg.asynch.AMsg7.snd.ATM1.User1.ejectCard,msg.asynch.AMsg7.rcv.ATM1.User1.ejectCard|}

alphaLifelines(<<msg.asynch.AMsg8.snd.ATM1.User1.ejectCard>,<msg.asynch.AMsg8.rcv.ATM1.User1.ejectCard>>)={|msg.asynch.AMsg8.snd.ATM1.User1.ejectCard,msg.asynch.AMsg8.rcv.ATM1.User1.ejectCard|}

alphaLifelines(<<msg.asynch.AMsg9.rcv.ATM1.User1.waitAccount,msg.asynch.AMsg10.snd.User1.ATM1.Account,msg.asynch.AMsg11.rcv.ATM1.User1.waitOperation,msg.asynch.AMsg12.snd.User1.ATM1.Operation>,<msg.asynch.AMsg9.snd.ATM1.User1.waitAccount,msg.asynch.AMsg10.rcv.User1.ATM1.Account,msg.asynch.AMsg11.snd.ATM1.User1.waitOperation,msg.asynch.AMsg12.rcv.User1.ATM1.Operation>>)={|msg.asynch.AMsg9.rcv.ATM1.User1.waitAccount,msg.asynch.AMsg10.snd.User1.ATM1.Account,msg.asynch.AMsg11.rcv.ATM1.User1.waitOperation,msg.asynch.AMsg12.snd.User1.ATM1.Operation,msg.asynch.AMsg9.snd.ATM1.User1.waitAccount,msg.asynch.AMsg10.rcv.User1.ATM1.Account,msg.asynch.AMsg11.snd.ATM1.User1.waitOperation,msg.asynch.AMsg12.rcv.User1.ATM1.Operation|}
alphaLifelines(<<msg.asynch.AMsg13.snd.User1.ATM1.cashAdvance>,<msg.asynch.AMsg13.rcv.User1.ATM1.cashAdvance,msg.synch.AMsg14.snd.ATM1.Bank1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>,<msg.synch.AMsg14.rcv.ATM1.Bank1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance>>)={|msg.asynch.AMsg13.snd.User1.ATM1.cashAdvance,msg.asynch.AMsg13.rcv.User1.ATM1.cashAdvance,msg.synch.AMsg14.snd.ATM1.Bank1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance,msg.synch.AMsg14.rcv.ATM1.Bank1.checkBalance|}

alphaLifelines(<<msg.asynch.AMsg17.rcv.ATM1.User1.pickCash>,<msg.asynch.AMsg16.snd.ATM1.Bank1.debit,msg.asynch.AMsg17.snd.ATM1.User1.pickCash>,<msg.asynch.AMsg16.rcv.ATM1.Bank1.debit>>)={|msg.asynch.AMsg17.rcv.ATM1.User1.pickCash,msg.asynch.AMsg16.snd.ATM1.Bank1.debit,msg.asynch.AMsg17.snd.ATM1.User1.pickCash,msg.asynch.AMsg16.rcv.ATM1.Bank1.debit|}
alphaLifelines(<<msg.asynch.AMsg18.snd.ATM1.User1.insuffFunds>,<msg.asynch.AMsg18.rcv.ATM1.User1.insuffFunds>>)={|msg.asynch.AMsg18.snd.ATM1.User1.insuffFunds,msg.asynch.AMsg18.rcv.ATM1.User1.insuffFunds|}

alphaLifelines(<<msg.asynch.AMsg19.snd.User1.ATM1.Operation>,<msg.asynch.AMsg19.rcv.User1.ATM1.Operation>>)={|msg.asynch.AMsg19.snd.User1.ATM1.Operation,msg.asynch.AMsg19.rcv.User1.ATM1.Operation|}

alphaLifelines(<<msg.asynch.AMsg20.snd.User1.ATM1.back,msg.asynch.AMsg21.rcv.ATM1.User1.ejectCard>,<msg.asynch.AMsg20.rcv.User1.ATM1.back,msg.asynch.AMsg21.snd.ATM1.User1.ejectCard>>)={|msg.asynch.AMsg20.snd.User1.ATM1.back,msg.asynch.AMsg21.rcv.ATM1.User1.ejectCard,msg.asynch.AMsg20.rcv.User1.ATM1.back,msg.asynch.AMsg21.snd.ATM1.User1.ejectCard|}

PrefixComposition(s)= if null(s) then SKIP else head(s) -> PrefixComposition(tail(s))

Messages(<(asynch,AMsg0,User1,ATM1,insertCard)>)= ||(t,id,from,to,n): {(asynch,AMsg0,User1,ATM1,insertCard)}@
[alphaMessage(t,id,from,to,n)]Message(t,id,from,to,n)

Messages(<(synch,AMsg1,ATM1,Bank1,verifyCard)>)= ||(t,id,from,to,n):{(synch,AMsg1,ATM1,Bank1,verifyCard)}@
[alphaMessage(t,id,from,to,n)]Message(t,id,from,to,n)

Messages(<(asynch,AMsg3,ATM1,User1,waitPIN),(asynch,AMsg4,User1,ATM1,PIN)>)
= ||(t,id,from,to,n): {(asynch,AMsg3,ATM1,User1,waitPIN),(asynch,AMsg4,User1,ATM1,PIN)}@
[alphaMessage(t,id,from,to,n)]Message(t,id,from,to,n)

Messages(<(synch,AMsg5,ATM1,Bank1,verifyPIN)>)= ||(t,id,from,to,n): {(synch,AMsg5,ATM1,Bank1,verifyPIN)}@
[alphaMessage(t,id,from,to,n)]Message(t,id,from,to,n)

Messages(<(asynch,AMsg7,ATM1,User1,ejectCard)>)= ||(t,id,from,to,n): {(asynch,AMsg7,ATM1,User1,ejectCard)}@
[alphaMessage(t,id,from,to,n)]Message(t,id,from,to,n)

Messages(<(asynch,AMsg8,ATM1,User1,ejectCard)>)= ||(t,id,from,to,n): {(asynch,AMsg8,ATM1,User1,ejectCard)}@
[alphaMessage(t,id,from,to,n)]Message(t,id,from,to,n)
```

Figure 51. Code CSPм générée à partir d'UML 2.0 SD pour le système ATM (partie 5)

```
Messages(<(asynch,AMsg9,ATM1,User1,waitAccount),(asynch,AMsg10,User1,ATM1,Account),(asynch,AMsg11,ATM1,User1,wait
Operation),(asynch,AMsg12,User1,ATM1,Operation)>)
= ||(t,id,from,to,n):
{(asynch,AMsg9,ATM1,User1,waitAccount),(asynch,AMsg10,User1,ATM1,Account),(asynch,AMsg11,ATM1,User1,waitOperation)
,(asynch,AMsg12,User1,ATM1,Operation)}@ [alphaMessage(t,id,from,to,n)]Message(t,id,from,to,n)

Messages(<(asynch,AMsg13,User1,ATM1,cashAdvance),(synch,AMsg14,ATM1,Bank1,checkBalance)>)
= ||(t,id,from,to,n): {(asynch,AMsg13,User1,ATM1,cashAdvance),(synch,AMsg14,ATM1,Bank1,checkBalance)}@
[alphaMessage(t,id,from,to,n)]Message(t,id,from,to,n)

Messages(<(asynch,AMsg16,ATM1,Bank1,debit),(asynch,AMsg17,ATM1,User1,pickCash)>)= ||(t,id,from,to,n):
{(asynch,AMsg16,ATM1,Bank1,debit),(asynch,AMsg17,ATM1,User1,pickCash)}@
[alphaMessage(t,id,from,to,n)]Message(t,id,from,to,n)

Messages(<(asynch,AMsg18,ATM1,User1,insuffFunds)>)= ||(t,id,from,to,n): {(asynch,AMsg18,ATM1,User1,insuffFunds)}@
[alphaMessage(t,id,from,to,n)]Message(t,id,from,to,n)

Messages(<(asynch,AMsg19,User1,ATM1,Operation)>)= ||(t,id,from,to,n): {(asynch,AMsg19,User1,ATM1,Operation)}@
[alphaMessage(t,id,from,to,n)]Message(t,id,from,to,n)

Messages(<(asynch,AMsg20,User1,ATM1,back),(asynch,AMsg21,ATM1,User1,ejectCard)>)= ||(t,id,from,to,n):
{(asynch,AMsg20,User1,ATM1,back),(asynch,AMsg21,ATM1,User1,ejectCard)}@
[alphaMessage(t,id,from,to,n)]Message(t,id,from,to,n)

alphaMessages(<(asynch,AMsg0,User1,ATM1,insertCard)>)= alphaMessage(asynch,AMsg0,User1,ATM1,insertCard)

alphaMessages(<(synch,AMsg1,ATM1,Bank1,verifyCard)>)=alphaMessage(synch,AMsg1,ATM1,Bank1,verifyCard)

alphaMessages(<(asynch,AMsg3,ATM1,User1,waitPIN),(asynch,AMsg4,User1,ATM1,PIN)>)=union(alphaMessage(asynch,AMsg3
,ATM1,User1,waitPIN),alphaMessage(asynch,AMsg4,User1,ATM1,PIN))

alphaMessages(<(synch,AMsg5,ATM1,Bank1,verifyPIN)>)= alphaMessage(synch,AMsg5,ATM1,Bank1,verifyPIN)

alphaMessages(<(asynch,AMsg7,ATM1,User1,ejectCard)>)=alphaMessage(asynch,AMsg7,ATM1,User1,ejectCard)

alphaMessages(<(asynch,AMsg8,ATM1,User1,ejectCard)>)=alphaMessage(asynch,AMsg8,ATM1,User1,ejectCard)

alphaMessages(<(asynch,AMsg9,ATM1,User1,waitAccount),(asynch,AMsg10,User1,ATM1,Account),(asynch,AMsg11,ATM1,User
1,waitOperation),(asynch,AMsg12,User1,ATM1,Operation)>)
=union(union(alphaMessage(asynch,AMsg9,ATM1,User1,waitAccount),alphaMessage(asynch,AMsg10,User1,ATM1,Account)),un
ion(alphaMessage(asynch,AMsg11,ATM1,User1,waitOperation),alphaMessage(asynch,AMsg12,User1,ATM1,Operation)))

alphaMessages(<(asynch,AMsg13,User1,ATM1,cashAdvance),(synch,AMsg14,ATM1,Bank1,checkBalance)>)=union(alphaMessa
ge(asynch,AMsg13,User1,ATM1,cashAdvance),alphaMessage(synch,AMsg14,ATM1,Bank1,checkBalance))

alphaMessages(<(asynch,AMsg16,ATM1,Bank1,debit),(asynch,AMsg17,ATM1,User1,pickCash)>)=union(alphaMessage(asynch,A
Msg16,ATM1,Bank1,debit),alphaMessage(asynch,AMsg17,ATM1,User1,pickCash))
alphaMessages(<(asynch,AMsg18,ATM1,User1,insuffFunds)>)=alphaMessage(asynch,AMsg18,ATM1,User1,insuffFunds)

alphaMessages(<(asynch,AMsg19,User1,ATM1,Operation)>)= alphaMessage(asynch,AMsg19,User1,ATM1,Operation)

alphaMessages(<(asynch,AMsg20,User1,ATM1,back),(asynch,AMsg21,ATM1,User1,ejectCard)>)=union(alphaMessage(asynch,A
Msg20,User1,ATM1,back),alphaMessage(asynch,AMsg21,ATM1,User1,ejectCard))

Message(asynch,AMsg0,User1,ATM1,insertCard)= msg.asynch.AMsg0.snd.User1.ATM1.insertCard ->
msg.asynch.AMsg0.rcv.User1.ATM1.insertCard -> SKIP

Message(synch,AMsg1,ATM1,Bank1,verifyCard)= msg.synch.AMsg1.snd.ATM1.Bank1.verifyCard ->
msg.synch.AMsg1.rcv.ATM1.Bank1.verifyCard ->  msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard -> SKIP

Message(asynch,AMsg3,ATM1,User1,waitPIN)= msg.asynch.AMsg3.snd.ATM1.User1.waitPIN ->
msg.asynch.AMsg3.rcv.ATM1.User1.waitPIN -> SKIP

Message(asynch,AMsg4,User1,ATM1,PIN)= msg.asynch.AMsg4.snd.User1.ATM1.PIN -> msg.asynch.AMsg4.rcv.User1.ATM1.PIN -
> SKIP

Message(synch,AMsg5,ATM1,Bank1,verifyPIN)= msg.synch.AMsg5.snd.ATM1.Bank1.verifyPIN ->
msg.synch.AMsg5.rcv.ATM1.Bank1.verifyPIN ->  msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN -> SKIP
```

Figure 52. Code CSPM générée à partir d'UML 2.0 SD pour le système ATM (partie 6)

```
Message(asynch,AMsg7,ATM1,User1,ejectCard)= msg.asynch.AMsg7.snd.ATM1.User1.ejectCard ->
msg.asynch.AMsg7.rcv.ATM1.User1.ejectCard -> SKIP

Message(asynch,AMsg8,ATM1,User1,ejectCard)= msg.asynch.AMsg8.snd.ATM1.User1.ejectCard ->
msg.asynch.AMsg8.rcv.ATM1.User1.ejectCard -> SKIP

Message(asynch,AMsg9,ATM1,User1,waitAccount)= msg.asynch.AMsg9.snd.ATM1.User1.waitAccount ->
msg.asynch.AMsg9.rcv.ATM1.User1.waitAccount -> SKIP

Message(asynch,AMsg10,User1,ATM1,Account)= msg.asynch.AMsg10.snd.User1.ATM1.Account ->
msg.asynch.AMsg10.rcv.User1.ATM1.Account -> SKIP

Message(asynch,AMsg11,ATM1,User1,waitOperation)= msg.asynch.AMsg11.snd.ATM1.User1.waitOperation ->
msg.asynch.AMsg11.rcv.ATM1.User1.waitOperation -> SKIP

Message(asynch,AMsg12,User1,ATM1,Operation)= msg.asynch.AMsg12.snd.User1.ATM1.Operation ->
msg.asynch.AMsg12.rcv.User1.ATM1.Operation -> SKIP

Message(asynch,AMsg13,User1,ATM1,cashAdvance)= msg.asynch.AMsg13.snd.User1.ATM1.cashAdvance ->
msg.asynch.AMsg13.rcv.User1.ATM1.cashAdvance -> SKIP

Message(synch,AMsg14,ATM1,Bank1,checkBalance)= msg.synch.AMsg14.snd.ATM1.Bank1.checkBalance ->
msg.synch.AMsg14.rcv.ATM1.Bank1.checkBalance ->  msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance -> SKIP

Message(asynch,AMsg16,ATM1,Bank1,debit)= msg.asynch.AMsg16.snd.ATM1.Bank1.debit ->
msg.asynch.AMsg16.rcv.ATM1.Bank1.debit -> SKIP

Message(asynch,AMsg17,ATM1,User1,pickCash)= msg.asynch.AMsg17.snd.ATM1.User1.pickCash ->
msg.asynch.AMsg17.rcv.ATM1.User1.pickCash -> SKIP

Message(asynch,AMsg18,ATM1,User1,insuffFunds)= msg.asynch.AMsg18.snd.ATM1.User1.insuffFunds ->
msg.asynch.AMsg18.rcv.ATM1.User1.insuffFunds -> SKIP

Message(asynch,AMsg19,User1,ATM1,Operation)= msg.asynch.AMsg19.snd.User1.ATM1.Operation ->
msg.asynch.AMsg19.rcv.User1.ATM1.Operation -> SKIP

Message(asynch,AMsg20,User1,ATM1,back)= msg.asynch.AMsg20.snd.User1.ATM1.back ->
msg.asynch.AMsg20.rcv.User1.ATM1.back -> SKIP

Message(asynch,AMsg21,ATM1,User1,ejectCard)= msg.asynch.AMsg21.snd.ATM1.User1.ejectCard ->
msg.asynch.AMsg21.rcv.ATM1.User1.ejectCard -> SKIP

alphaMessage(asynch,AMsg0,User1,ATM1,insertCard)={|msg.asynch.AMsg0.snd.User1.ATM1.insertCard,msg.asynch.AMsg0.rcv.User1.ATM1.insertCard|}

alphaMessage(synch,AMsg1,ATM1,Bank1,verifyCard)={|msg.synch.AMsg1.snd.ATM1.Bank1.verifyCard,msg.synch.AMsg1.rcv.ATM1.Bank1.verifyCard,msg.synch.AMsg1.ack.Bank1.ATM1.verifyCard|}

alphaMessage(asynch,AMsg3,ATM1,User1,waitPIN)={|msg.asynch.AMsg3.snd.ATM1.User1.waitPIN,msg.asynch.AMsg3.rcv.ATM1.User1.waitPIN|}

alphaMessage(asynch,AMsg4,User1,ATM1,PIN)={|msg.asynch.AMsg4.snd.User1.ATM1.PIN,msg.asynch.AMsg4.rcv.User1.ATM1.PIN|}

alphaMessage(synch,AMsg5,ATM1,Bank1,verifyPIN)={|msg.synch.AMsg5.snd.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.rcv.ATM1.Bank1.verifyPIN,msg.synch.AMsg5.ack.Bank1.ATM1.verifyPIN|}

alphaMessage(asynch,AMsg7,ATM1,User1,ejectCard)={|msg.asynch.AMsg7.snd.ATM1.User1.ejectCard,msg.asynch.AMsg7.rcv.ATM1.User1.ejectCard|}

alphaMessage(asynch,AMsg8,ATM1,User1,ejectCard)={|msg.asynch.AMsg8.snd.ATM1.User1.ejectCard,msg.asynch.AMsg8.rcv.ATM1.User1.ejectCard|}

alphaMessage(asynch,AMsg9,ATM1,User1,waitAccount)={|msg.asynch.AMsg9.snd.ATM1.User1.waitAccount,msg.asynch.AMsg9.rcv.ATM1.User1.waitAccount|}

alphaMessage(asynch,AMsg10,User1,ATM1,Account)={|msg.asynch.AMsg10.snd.User1.ATM1.Account,msg.asynch.AMsg10.rcv.User1.ATM1.Account|}
```

Figure 53. Code CSPM générée à partir d'UML 2.0 SD pour le système ATM (partie 7)

```
alphaMessage(asynch,AMsg11,ATM1,User1,waitOperation)={|msg.asynch.AMsg11.snd.ATM1.User1.waitOperation,msg.asynch.AMsg11.rcv.ATM1.User1.waitOperation|}

alphaMessage(asynch,AMsg12,User1,ATM1,Operation)={|msg.asynch.AMsg12.snd.User1.ATM1.Operation,msg.asynch.AMsg12.rcv.User1.ATM1.Operation|}

alphaMessage(asynch,AMsg13,User1,ATM1,cashAdvance)={|msg.asynch.AMsg13.snd.User1.ATM1.cashAdvance,msg.asynch.AMsg13.rcv.User1.ATM1.cashAdvance|}

alphaMessage(synch,AMsg14,ATM1,Bank1,checkBalance)={|msg.synch.AMsg14.snd.ATM1.Bank1.checkBalance,msg.synch.AMsg14.rcv.ATM1.Bank1.checkBalance,msg.synch.AMsg14.ack.Bank1.ATM1.checkBalance|}

alphaMessage(asynch,AMsg16,ATM1,Bank1,debit)={|msg.asynch.AMsg16.snd.ATM1.Bank1.debit,msg.asynch.AMsg16.rcv.ATM1.Bank1.debit|}

alphaMessage(asynch,AMsg17,ATM1,User1,pickCash)={|msg.asynch.AMsg17.snd.ATM1.User1.pickCash,msg.asynch.AMsg17.rcv.ATM1.User1.pickCash|}

alphaMessage(asynch,AMsg18,ATM1,User1,insuffFunds)={|msg.asynch.AMsg18.snd.ATM1.User1.insuffFunds,msg.asynch.AMsg18.rcv.ATM1.User1.insuffFunds|}

alphaMessage(asynch,AMsg19,User1,ATM1,Operation)={|msg.asynch.AMsg19.snd.User1.ATM1.Operation,msg.asynch.AMsg19.rcv.User1.ATM1.Operation|}

alphaMessage(asynch,AMsg20,User1,ATM1,back)={|msg.asynch.AMsg20.snd.User1.ATM1.back,msg.asynch.AMsg20.rcv.User1.ATM1.back|}

alphaMessage(asynch,AMsg21,ATM1,User1,ejectCard)={|msg.asynch.AMsg21.snd.ATM1.User1.ejectCard,msg.asynch.AMsg21.rcv.ATM1.User1.ejectCard|}

-- assertions
assert p:[deadlock free[FD]]
assert p:[deadlock free[F]]
assert p:[divergence free [FD]]
assert p:[deterministic[FD]]
assert p:[deterministic[F]]
```

Figure 54. Code CSPM générée à partir d'UML 2.0 SD pour le système ATM (partie 8)

Les Figures 55 et 56 présentent respectivement le code CSPM générée à partir d'UML 2.0 STM et UML 2.0 CD pour le système ATM.

```
datatype  te1= OK|NOTOK
datatype  NAME= ejectCrd

channel   verifyPIN,verifyPINAck,waitOperation,Operation,checkBalance,insuffFunds,checkBalanceAck,insertCard,verifyCard,
verifyCardAck,waitPIN,PIN,ejectCard,back,Entering,cashAdvance,debit,givingCash,pickCash,Processjoin
channel   op:NAME
channel   cash:Bool
channel   card,pin,balance:te1

START0 = WaitingForCard

f(card.OK)=True
f(card.NOTOK)=False
f(pin.OK)=True
f(pin.NOTOK)=False
f(op.ejectCrd)=False
f(cash.true)=True
f(cash.false)=False
f(balance.OK)= True
f(balance.NOTOK)= False

WaitingForCard = insertCard -> VerfyingAndWaiting
VerfyingAndWaiting =  START1 [|{}|] START2
START1 = VerifyingCard
START2 = WaitingPIN
VerifyingCard = verifyCard -> verifyCardAck -> END1
WaitingPIN = waitPIN -> PIN -> END2

END1 = Processjoin -> if f(card.OK) then VerifyingPIN else ejectingCard
END2 = Processjoin -> SKIP

VerifyingPIN = verifyPIN -> if f(pin.NOTOK) then verifyPINAck -> ejectingCard else verifyPINAck -> ChoosingOperation
ejectingCard = ejectCard -> SKIP
ChoosingOperation = START3
START3 = WaitingAccount
WaitingAccount = waitOperation -> if f(op.ejectCrd) then Operation -> PerformingOperation else Operation -> return

PerformingOperation = START4
return = back -> END3
START4 = GetingOperation
END3 = ejectingCard
GetingOperation = if f(cash.true) then Input else OperationCompleted
Input = Entering -> cashAdvance -> CheckingBalance
OperationCompleted = END4
CheckingBalance = checkBalance -> if f(balance.OK) then checkBalanceAck -> Debiting else insuffFunds -> checkBalanceAck -
>  OperationCompleted
END4 = WaitingAccount
Debiting = debit -> OutputingCash
OutputingCash = givingCash -> pickCash -> OperationCompleted

assert START0:[deadlock free[FD]]
assert START0:[deadlock free[F]]
assert START0:[divergence free [FD]]
assert START0:[deterministic[FD]]
assert START0:[deterministic[F]]
```

Figure 55. Code CSPM générée à partir d'UML 2.0 STM pour le système ATM

```
SYSTEM0 = User [|{||}|] ATM
SYSTEM1 = ATM [|{||}|] Bank
```

Figure 56. Code CSPM générée à partir d'UML 2.0 CD pour le système ATM

5.4 ANALYSE ET VERIFICATION

CSPM permet de définir diverses assertions dans les fichiers CSPM. Celles-ci sont ensuite ajoutées à la liste des assertions dans FDR afin de permettre une exécution pratique des assertions. Les assertions les plus simples dans CSPM sont les assertions de raffinement, qui sont des lignes de la forme : assert P [T= Q, permettant à FDR de vérifier si $P \sqsubseteq_T Q$. Le modèle sémantique peut être l'un des suivants : Le modèle de traces ([T=), le modèle des échecs ([F=) ou le modèle des échecs-divergences ([FD=).

Les résultats de la vérification de la spécification CSP générée à partir de la transformation d'UML 2.0 SD pour le système ATM, en utilisant FDR4, sont illustrés par la Figure 57. La vérification de la propriété de deadlock dans les deux modèles sémantiques, échecs et échecs-divergences, a échoué, bien que les vérifications de livelock et de déterminisme ont réussi.

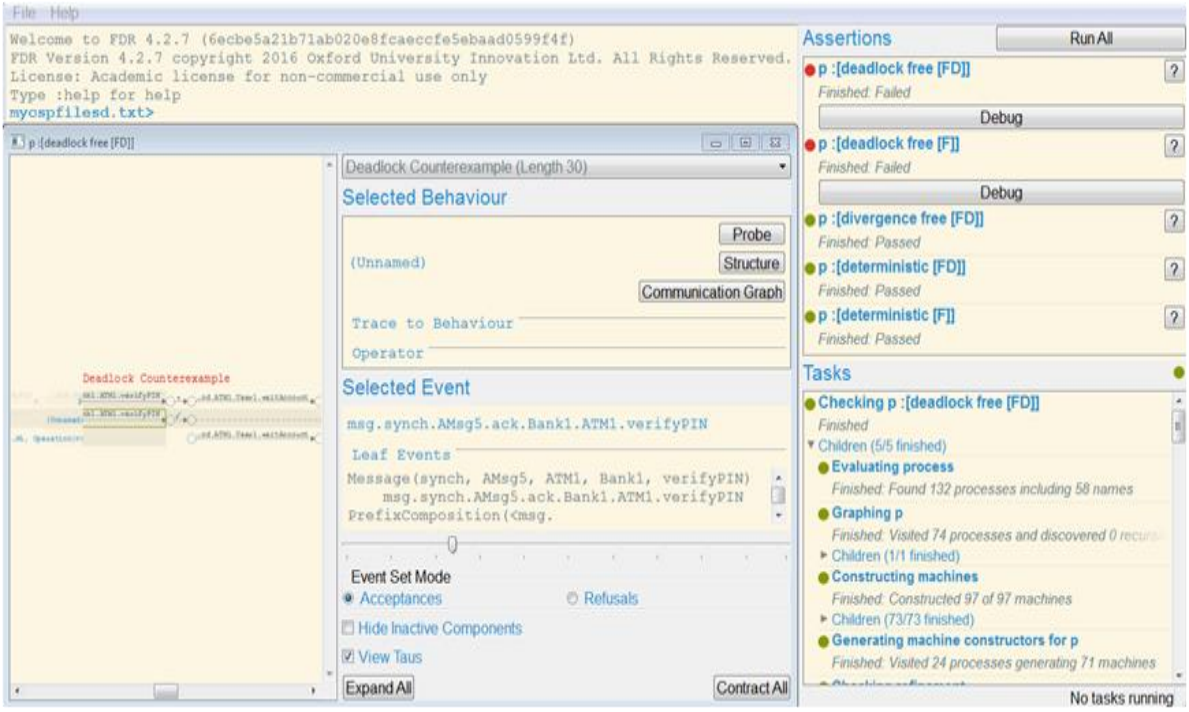


Figure 57. Résultat de l'application du modèle checker FDR4 sur le fichier "*mycspfilesd.csp* "

La Figure 58 présente les résultats de la vérification de la spécification CSP générée à partir de la transformation d'UML 2.0 STM pour le système ATM. La propriété de

livelock est satisfaite, cependant, les vérifications de deadlock et de déterminisme dans les deux modèles sémantiques, échecs et échecs-divergences, ont échoué.

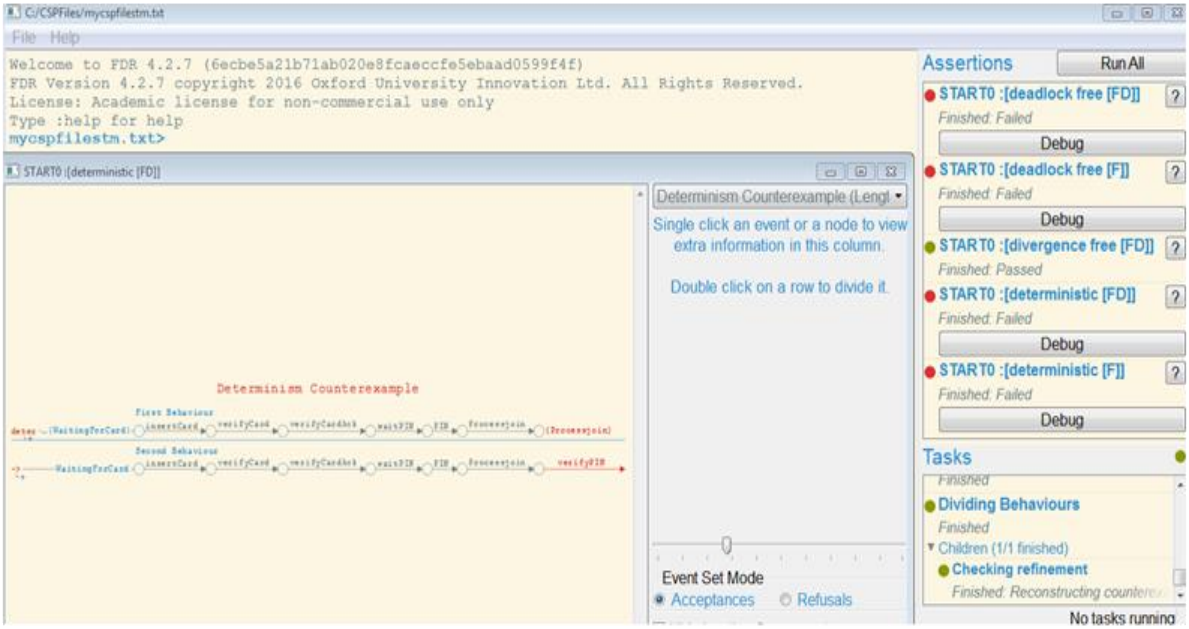


Figure 58. Résultat de l'application du modèle checker FDR4 sur le fichier "*mycspfilestm.csp* "

Dans ce qui suit, nous présentons en détail la vérification des propriétés de Deadlock, livelock et de determinism en utilisant FDR4.

5.4.1 Deadlock

Il est possible d'affirmer qu'un processus est sans deadlock dans le modèle échecs ou le modèle échecs-divergences. Dans les deux cas, FDR convertit cela en interne en une assertion de raffinement de la forme DF(A) ⊑ P, où A est l'alphabet des événements que P effectue et DF(A) est le processus le plus non déterministe sur A, c'est à dire DF(A) = $\sqcap_{x \in A}\ x \rightarrow DF(A)$. Intuitivement, dans le modèle des échecs, cela signifie que le processus ne peut jamais se retrouver dans un état dans lequel aucun événement n'est proposé. Cette affirmation peut s'écrire : assert P : [deadlock free], qui vérifie par défaut l'assertion dans le modèle échecs-divergences, ou un modèle sémantique particulier peut être spécifié en utilisant : assert P :[deadlock free [F]], qui insiste sur le fait que le modèle des échecs sera utilisé pour vérifier l'assertion (FDR, 2020).

Le viseur de débogage, comme indiqué dans la figure 58, permet d'afficher un contre-exemple à une assertion de raffinement. En particulier, il tente d'expliquer comment

l'implémentation a évolué vers un état où elle pourrait adopter un comportement interdit par la spécification. Conceptuellement, le viseur de débogage est un tableau dans lequel les lignes représentent des comportements de machines particulières tandis que les colonnes représentent les événements synchronisés.

5.4.2 livelock

Dans le modèle échecs-divergences, il est également possible de vérifier si un processus peut diverger, c'est-à-dire effectuer une quantité infinie de travail interne (c'est également connu sous le nom de contrôle de liberté de livelock). FDR convertit toutes ces vérifications en une assertion de raffinement de la forme CHAOS(A) ⊑ P, où CHAOS(A) offre un choix non déterministe sur tous les événements de A, mais peut également se bloquer à tout moment. A est l'alphabet du processus P. Cela dit essentiellement que le processus peut avoir un comportement arbitraire, mais ne peut jamais diverger. Cela peut s'écrire comme suit : assert P :[divergence free] qui vérifie par défaut le modèle échecs-divergences, ou utiliser l'assertion : assert P :[divergence free [FD]], qui insiste sur le fait que le modèle échecs-divergences sera utilisé pour vérifier l'assertion (FDR, 2020).

5.4.3 Déterminisme

FDR peut vérifier si un processus donné est déterministe en utilisant l'assertion : assert P :[deterministic]. Comme pour les autres propriétés, un modèle sémantique peut être spécifié comme suit : assert P :[deterministic [FD]].

FDR considère un processus P être déterministe à condition qu'aucun témoin du non-déterminisme n'existe, où un témoin consiste en une trace tr et un événement ev tel que P peut à la fois accepter et refuser a après tr. Formellement, dans le modèle d'échecs, P est non déterministe ssi : $\exists tr,a \cdot tr^\langle a\rangle \in traces(P) \wedge (tr,\{a\}) \in failures(P)$.

En interne, pour le modèle échecs-divergences, FDR construit en fait une version déterministe du processus P, en utilisant la fonction deter, puis vérifie si deter(P) $[_{FD}=$ P. Pour le modèle échecs, FDR utilise la vérification du déterminisme de Lazic qui exécute deux copies du processus en parallèle (FDR, 2020).

5.5 VERIFICATION ET TEST DE L'APPROCHE PROPOSEE

Prouver les approches de transformation peut s'effectuer à travers diverses méthodes, telles qu'une étude de cas théorique, une mise en pratique concrète, ou encore l'utilisation d'un outil automatique garantissant la transformation du modèle (Rahmoune & Chaoui, 2022). Dans notre approche, pour chacune des trois transformations proposées, les règles de la grammaire ont été prouvées par l'outil développé, qui a été testé sur plusieurs études de cas (ATM étant l'un d'entre eux).

Les transformations de modèles présentent diverses propriétés nécessaires pour garantir leur exactitude, telles que la terminaison, le déterminisme, l'exactitude syntaxique, et autres propriétés (Meghzili et al., 2017). Les propriétés de terminaison et de déterminisme sont des exigences importantes pour les applications pratiques des transformations de modèles, assurant que celles-ci se terminent toujours et produisent un résultat unique (Küster, 2006).

Dans notre approche, afin de vérifier les propriétés de terminaison et de déterminisme, nous nous sommes inspirés du travail de vérification présenté par Rahmoune et Chaoui (Rahmoune & Chaoui, 2022). Cette vérification a été réalisée dans le cadre d'une approche de transformation automatique basée sur la transformation de graphes et utilisant l'outil AToM^3.

5.5.1 La terminaison

La propriété de terminaison fait référence à la garantie de l'existence de modèle(s) cible(s) ou à la garantie qu'une transformation de modèle s'exécute sans entrer dans une situation d'infinie. Dans notre approche de transformation, pour chaque grammaire de graphes et pour chaque règle de la grammaire, nous avons utilisé une variable globale « *Visited* » initialisée à 0 dans l'action initiale. Si la condition de la règle est vraie (*Visited == 0*), alors l'action de cette règle sera exécutée et « *Visited* » changera sa valeur à 1 (*Visited = 1*) pour éviter l'exécution répétée de la règle sur le même élément, c'est-à-dire pour éviter les boucles infinies lors de la transformation. Les figures 36,40 et 42 illustrent l'utilisation de cette variable. Les règles de la grammaire de graphes assurent la transformation de tous les éléments du modèle source en code CSP correspondant. A la

fin de la transformation, aucune règle ne peut être exécutée. Par conséquent, nous en déduisons que la propriété de terminaison est vérifiée.

5.5.2 Le Déterminisme

La propriété de déterminisme fait référence à la garantit qu'une transformation produit toujours le même résultat (modèle cible). Dans notre approche, et pour chaque grammaire de graphes proposée, Chaque règle a une priorité invariante jouant un rôle très important dans le processus de transformation. Les règles seront appliquées par ordre croissant jusqu'à aucune règle n'est appliquée. Donc, le choix de la règle à appliquer est guidé par le classement des règles selon leur priorités. Pour toute itération, toutes les règles sont testées par ordre croissant de leurs priorités. Chaque fois que nous exécutons la transformation du modèle source, la transformation suit le même ordre de règles prédéfini, cela garantit la répétition des mêmes étapes et la génération du même modèle cible. Par conséquent, l'utilisation de priorités garantit la propriété de déterminisme.

5.5.3 L'exactitude syntaxique

Dans notre approche, les règles proposées pour les trois grammaires de graphes ont les deux parties LHS et RHS identiques, contenant un ou plusieurs éléments du modèle source. Le code CSP est généré par l'exécution des actions des règles. L'exactitude syntaxique du processus de transformation peut être garantie par le compilateur du langage CSP intégré au niveau du mode-checker FDR4. Durant la phase de vérification du code CSP généré, si ce dernier contient une expression syntaxiquement incorrectes, le compilateur affiche directement l'erreur détectée. Nous avons testé toutes les règles proposées dans les trois grammaires de graphes, et le code généré est syntaxiquement correct. Donc, la propriété d'exactitude syntaxique est vérifiée.

5.6 CONCLUSION

En conclusion de ce chapitre, nous avons illustré notre approche par une étude de cas concernant un distributeur automatique de billets (ATM). Au fil de cette illustration, nous avons modélisé la structure du système à l'aide du diagramme de classe, tout en décrivant son comportement à travers les diagrammes de séquence et d'états-

transitions. Cette approche illustre l'efficacité de notre approche dans la modélisation de systèmes complexes. De plus, nous avons appliqué les trois grammaires de graphes que nous avons développées pour générer les spécifications CSP correspondantes à partir de nos modèles UML 2.0. Cette transformation vers des spécifications formelles nous a permis de vérifier les propriétés des systèmes modélisés en utilisant le model-checker FDR4.

CONCLUSION GENERALE

Le travail de recherche présenté dans cette thèse s'inscrit dans le cadre général de l'IDM. L'objectif était de proposer une approches intégrée UML 2.0/CSP permettant la modélisation et l'analyse des systèmes distribués. Après avoir exposé en détail les connaissances dans le domaine de l'IDM de manière générale, et plus particulièrement dans celui de l'ingénierie système, nous avons pu constater que l'intégration des langages de spécification semi-formels et formels peut améliorer la fiabilité des systèmes développés ainsi que l'efficacité des processus de développement. Les équipes de développement peuvent ainsi bénéficier de la clarté et de la compréhensibilité des spécifications semi-formelles, tout en assurant une correction formelle grâce aux spécifications formelles, permettant de minimiser les erreurs de conception dès les premières phases du développement.

Motivés par diverses considérations, nous avons proposé notre approche intégrée UML 2.0 / CSP, qui repose sur l'utilisation des diagrammes de classe pour représenter la structure du système modélisé, et des diagrammes de séquence et d'états-transitions pour modéliser son comportement. Nous avons introduit une formalisation en CSP de certains composants du diagramme d'états-transitions. L'implémentation de cette approche a été basée sur l'utilisation combinée de la méta-modélisation et de la transformation de graphes à travers l'outil AToM^3. Trois méta-modèles et trois grammaires de graphes ont été définis. Le résultat final de ce travail est une interface graphique permettant à l'utilisateur de modéliser la structure et le comportement du système modélisé, puis de transformer automatiquement les diagrammes élaborés en spécifications CSP correspondantes. Ces dernières sont ensuite vérifiées à l'aide de l'outil FDR4. Enfin, notre approche a été appliquée au système ATM.

Par rapport aux travaux existants, notre approche se distingue par son mode d'application automatique, contrairement aux approches manuelles qui sont souvent difficiles à appliquer, surtout avec l'augmentation de la complexité des systèmes modélisés. Notre approche prend en considération les aspects statique et dynamique du système modélisé. En ce qui concerne l'aspect dynamique, elle modélise le comportement individuel des objets du système à l'aide

des diagrammes d'états-transitions, tout en modélisant le comportement global à travers les diagrammes de séquence.

La transformation des diagrammes de séquence couvre tous les types de messages : synchrones, asynchrones et messages de retour, ainsi que dix opérateurs d'interaction des fragments combinés tels que Seq, Strict, Par, Alt, Opt, Break, Loop, Ignore, Consider et Assert. La transformation des diagrammes d'états-transitions prend en considération les états simples et composites, les transitions étiquetées par des gardes, des événements et des actions. Elle inclut les événements implicites et explicites, ainsi que le choix statique et dynamique. Pour chaque état, les actions d'entrée (entry Action) et de sortie (Exit Action) sont prises en compte. Notre approche englobe également les pseudo-états les plus utilisés, à savoir les pseudo-états initial, final, de choix, ainsi que le Fork, le Join, la jonction et les régions.

Dans le cadre de travaux futurs, nous avons l'intention de transformer d'autres diagrammes UML 2.0 en CSP, notamment les diagrammes d'interaction overview et les diagrammes de cas d'utilisation, ainsi que l'amélioration de la formalisation des diagrammes de classe en CSP. Nous prévoyons également d'utiliser d'autres vérificateurs de modèles CSP tels que PAT et ProB, pour faire face aux limites rencontrées lors des tests de raffinement avec FDR4. Enfin, nous envisageons d'aborder le problème de l'exactitude formelle de la transformation elle-même en utilisant des démonstrateurs de théorèmes tels qu'Isabelle et COQ, et proposer une approche de vérification hybride qui combine les deux techniques de vérification formelles : le model-checking et le theorem-proving.

BIBLIOGRAPHIE

Alagar, V. S., Periyasamy, K., & Periyasamy, K. (2011). *Specification of software systems.* Springer.

Almeida, J. B., Frade, M. J., Pinto, J. S., Melo de Sousa, S., Almeida, J. B., Frade, M. J., . . . Melo de Sousa, S. (2011). An overview of formal methods tools and techniques. *Rigorous Software Development: An Introduction to Program Verification*, 15–44.

Alspaugh, T. A., Faulk, S. R., Britton, K. H., Parker, R. A., Parnas, D. L., & Shore, J. (1992). *Software requirements for the A-7E aircraft.* Tech. rep., Technical Report NRL-9194, Naval Research Lab., Wash., DC.

Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H.-J., Kuske, S., . . . Taentzer, G. (1999). Graph transformation for specification and programming. *Science of Computer programming, 34*, 1–54.

Aouat, A., Bendella, F., & others. (2012). Tools of model transformation by graph transformation. *2012 IEEE International Conference on Computer Science and Automation Engineering*, (pp. 425–428).

Astesiano, E., Bidoit, M., Kirchner, H., Krieg-Brückner, B., Mosses, P. D., Sannella, D., & Tarlecki, A. (2002). CASL: the common algebraic specification language. *Theoretical Computer Science, 286*, 153–196.

Barroca, B., Lúcio, L., Amaral, V., Félix, R., & Sousa, V. (2011). Dsltrans: A turing incomplete transformation language. *Software Language Engineering: Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers 3*, (pp. 296–305).

Barroca, L. M., & McDermid, J. A. (1992). Formal methods: Use and relevance for the development of safety-critical systems. *The Computer Journal, 35*, 579–599.

Belaunde, M., Casanave, C., DSouza, D., Duddy, K., El Kaim, W., Kennedy, A., . . . others. (2003). MDA Guide Version 1.0. 1. *MDA Guide Version 1.0. 1*.

Berry, G., & Gonthier, G. (1992). The Esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming, 19*, 87–152.

Bertot, Y., & Castéran, P. (2013). *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Springer Science & Business Media.

Bézivin, J., & Briot, J.-P. (2004). Sur les principes de base de l'ingénierie des modèles. *Obj. Logiciel Base données Réseaux, 10*, 145–157.

Bézivin, J., & Gerbé, O. (2001). Towards a precise definition of the OMG/MDA framework. *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, (pp. 273–280).

Bisztray, D., Ehrig, K., & Heckel, R. (2007). Case study: UML to CSP transformation. *Applications of Graph Transformation with Industrial Relevance*.

Blanc, X., & Salvatori, O. (2011). *MDA en action: Ingénierie logicielle guidée par les modèles.* Editions Eyrolles.

Bolognesi, T., & Brinksma, E. (1987). Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems, 14*, 25-59. https://doi.org/https://doi.org/10.1016/0169-7552(87)90085-7

Bouarioua, M., Chaoui, A., & Elmansouri, R. (2011). From UML Sequence Diagrams to Labeled Generalized Stochastic Petri Net Models Using Graph Transformation. Dans

J. J. Yonazi, E. Sedoyeka, E. Ariwa, & E. "El-Qawasmeh (Éd.). (pp. 318–328). Berlin: Springer Berlin Heidelberg.

Braun, P., & Marschall, F. (2003). Transforming object oriented models with BOTL. *Electronic Notes in Theoretical Computer Science, 72*, 103–117.

Bucci, G., Campanai, M., Nesi, P., & Traversi, M. (1994). An object-oriented dual language for specifying reactive systems. *Proceedings of IEEE International Conference on Requirements Engineering*, (pp. 6–15).

Carvalho, G. H., Dias, T., Mota, A., & Sampaio, A. (2011). Analytical comparison of refinement checkers. *14th Brazilian Symposium on Formal Methods: Short Papers.*

Chabbat, N., Saidouni, D. E., Boukharrou, R., & Ghanemi, S. (2020). Formal Verification of UML MARTE Specifications Based on a True Concurrency Real Time Model. *Computing and Informatics, 39*, 1022–1060.

CSPM. (2023). "CSPM, https://cocotec.io/fdr/manual/cspm.html, Accessed:2023-12-05".

Cuadrado, J. S., Guerra, E., & de Lara, J. (2016). Static analysis of model transformations. *IEEE Transactions on Software Engineering, 43*, 868–897.

Cunha, E., Custódio, M., Rocha, H., & Barreto, R. S. (2011). Formal Verification of UML Sequence Diagrams in the Embedded Systems Context. Dans A. A. Fröhlich, & L. B. Becker (Éd.), *Brazilian Symposium on Computing System Engineering, SBESC 2011, Florianopolis, Brazil, November 7-11, 2011* (pp. 39–45). IEEE Computer Society. https://doi.org/10.1109/SBESC.2011.18

Custódio Soares, J. A., Lima, B., & Pascoal Faria, J. (2018). Automatic Model Transformation from UML Sequence Diagrams to Coloured Petri Nets. *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development* (pp. 668–679). Setubal: SCITEPRESS - Science and Technology Publications, Lda. https://doi.org/10.5220/0006731806680679

Czarnecki, K., Helsen, S., & others. (2003). Classification of model transformation approaches. *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, *45*, pp. 1–17.

da Costa Cavalheiro, S. A., Foss, L., & Ribeiro, L. (2017). Theorem proving graph grammars with attributes and negative application conditions. *Theoretical computer science, 686*, 25–77.

Da Silva, A. R. (2015). Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures, 43*, 139–155.

Dan, L., & Danning, L. (2010). An Approach to Formalize UML Sequence Diagrams in CSP. *3rd International Conference on Computer and Electrical Engineering (ICCEE 2010).*

de travail ingénierie système de l'AFIS, G. (2009). *Découvrir Et Comprendre L'is V3 22 02 09.* Tech. rep.

Ehrig, H., Engels, G., Kreowski, H. J., & Rozenberg, G. (1999). *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools.* World Scientific Publishing Co., Inc.

Elmansouri, R., Hamrouche, H., & Chaoui, A. (2011). From UML Activity Diagrams to CSP Expressions: A Graph Transformation Approach using Atomˆsup 3ˆTool. *International Journal of Computer Science Issues (IJCSI), 8*, 368.

Elmansouri, R., Meghzili, S., & Chaoui, A. (2021). A UML 2.0 Activity Diagrams/CSP Integrated Approach for Modeling and Verification of Software Systems. *Computer Science, 22*.

Ermel, C., Rudolf, M., & Taentzer, G. (1999). The AGG approach: Language and environment. Dans *Handbook Of Graph Grammars And Computing By Graph*

Transformation: Volume 2: Applications, Languages and Tools (pp. 551–603). World Scientific.

FDR. (2020). *FDR Manual, Release 4.2.7*.

Fleck, M., Troya, J., & Wimmer, M. (2015). Marrying search-based optimization and model transformation technology. *Proc. of NasBASE*, 1–16.

Fowler, M. (2003). *UML Distilled.* Addison-Wesley Professional.

Fuxman, A., Liu, L., Mylopoulos, J., Pistore, M., Roveri, M., & Traverso, P. (2004). Specifying and analyzing early requirements in Tropos. *Requirements Engineering, 9*, 132–150.

Fuxman, A., Pistore, M., Mylopoulos, J., & Traverso, P. (2001). Model checking early requirements specifications in Tropos. *Proceedings Fifth IEEE International Symposium on Requirements Engineering*, (pp. 174–181).

Gardner, W. B. (2005). CSP++: How Faithful to CSPm. *In Communicating Process Architectures 2005 (WoTUG-27), Concurrent Systems Engineering Series* (pp. 129–146). IOS Press.

Gibson-Robinson, T., Armstrong, P., Boulgakov, A., & Roscoe, A. W. (2014). FDR3—a modern refinement checker for CSP. *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*, (pp. 187–201).

Giese, H., Hildebrandt, S., & Lambers, L. (2014). Bridging the gap between formal semantics and implementation of triple graph grammars: Ensuring conformance of relational model transformation specifications and implementations. *Software & Systems Modeling, 13*, 273–299.

Gordon, M. J. (1988). HOL: A proof generating system for higher-order logic. Dans *VLSI specification, verification and synthesis* (pp. 73–128). Springer.

GReAT. (s.d.). Graph Rewriting and Transformation, https://archive.isis.vanderbilt.edu/tools/GReAT , accessed: 2023-12-05. *Graph Rewriting and Transformation, https://archive.isis.vanderbilt.edu/tools/GReAT , accessed: 2023-12-05*.

Greenyer, J., & Kindler, E. (2007). Reconciling tggs with qvt. *Model Driven Engineering Languages and Systems: 10th International Conference, MoDELS 2007, Nashville, USA, September 30-October 5, 2007. Proceedings 10*, (pp. 16–30).

Gurevich, Y., & others. (1993). Evolving algebras 1993: Lipari guide. *Evolving algebras 1993: Lipari guide.* Citeseer.

Halbwachs, N., & Raymond, P. (1993). A tutorial of Lustre. *IMAG, Grenoble*.

Hamrouche, H., Chaoui, A., & Mazouzi, S. (2022). A Graph Transformation Approach for Modeling and Verification of UML 2.0 Sequence Diagrams. *Computing and Informatics, 41*, 1284–1309.

Harel, D., & Pnueli, A. (1984). On the development of reactive systems. Dans *Logics and models of concurrent systems* (pp. 477–498). Springer.

Hidaka, S., Hu, Z., Inaba, K., Kato, H., & Nakano, K. (2011). GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations. *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, (pp. 480–483).

Hlaoui, Y. B., Younes, A. B., Ayed, L. J., & Fathalli, M. (2017). From Sequence Diagrams to Event B: A Specification and Verification Approach of Flexible Workflow Applications of Cloud Services Based on Meta-model Transformation. Dans S.

Reisman, S. I. Ahamed, C. Demartini, T. M. Conte, L. Liu, W. R. Claycomb, . . . K. Hasan (Éd.), *41st IEEE Annual Computer Software and Applications Conference, COMPSAC 2017, Turin, Italy, July 4-8, 2017. Volume 2* (pp. 187–192). IEEE Computer Society. https://doi.org/10.1109/COMPSAC.2017.135

Hoare, C. A. (1978, August). Communicating Sequential Processes. *Commun. ACM, 21*, 666–677. https://doi.org/10.1145/359576.359585

Holzmann, G. J. (1997, May). The Model Checker SPIN. *IEEE Trans. Softw. Eng., 23*, 279–295. https://doi.org/10.1109/32.588521

Idani, A. (2006). *B/UML : Mise en relation de spécifications B et de descriptions UML pour l'aide à la validation externe de développements formels en B.* Université de Grenoble 1.

ISO 9000. (2015). *Quality Management Systems-Fundamentals and Vocabulary (ISO 9000: 2015).* ISO Copyright office.

Jacobs, J., & Simpson, A. C. (2014). On a Process Algebraic Representation of Sequence Diagrams. Dans C. Canal, & A. Idani (Éd.), *Software Engineering and Formal Methods - SEFM 2014 Collocated Workshops: HOFM, SAFOME, OpenCert, MoKMaSD, WS-FMDS, Grenoble, France, September 1-2, 2014, Revised Selected Papers. 8938*, pp. 71–85. Springer. https://doi.org/10.1007/978-3-319-15201-1_5

Järvinen, H.-M., & Kurki-Suonio, R. (1991). DisCo specification language: marriage of actions and objects. *ICDCS*, (pp. 142–151).

Jensen, K., & Kristensen, L. M. (2009). *Coloured Petri Nets: Modelling and Validation of Concurrent Systems* (éd. 1st). Springer Publishing Company, Incorporated.

Juan de Lara, H. V. (2002). AToM 3 : A Tool for Multi-formalism and Meta-modelling. Dans H. Springer (Éd.), *Fundamental Approaches to Software Engineering.* https://doi.org/10.1007/3-540-45923-5_12Book

Kahani, N., Bagherzadeh, M., Cordy, J. R., Dingel, J., & Varró, D. (2019, August). Survey and Classification of Model Transformation Tools. *Softw. Syst. Model., 18*, 2361–2397. https://doi.org/10.1007/s10270-018-0665-6

Kaizu, T., Isobe, Y., & Suzuki, M. (2013, February). Refinement and Verification of Sequence Diagrams Using the Process Algebra CSP. *IEICE TRANSACTIONS ON FUNDAMENTALS OF ELECTRONICS COMMUNICATIONS AND COMPUTER SCIENCES, E96A*, 495–504. https://doi.org/10.1587/transfun.E96.A.495

Kaizu, T., Suzuki, M., & Isobe, Y. (2015). SDVerifier: A tool for verification of sequence diagrams using the process algebra CSP. *Computer Software, 32*, 234–252.

Kay, M. (2008). *XSLT 2.0 and XPath 2.0 Programmer's Reference (Programmer to Programmer)* (éd. 4). GBR: Wrox Press Ltd.

Kerkouche, E., Khalfaoui, K., & Chaoui, A. (2020). A rewriting logic-based semantics and analysis of UML activity diagrams: a graph transformation approach. *International Journal of Computer Aided Engineering and Technology, 12*, 237–262.

Kesraoui, S. M. (2017). *Intégration des techniques de vérification formelle dans une approche de conception des systèmes de contrôle-commande : application aux architectures SCADA.* Université de Bretagne Sud.

Klassen, L., & Wagner, R. (2012). EMorF-A tool for model transformations. *Electronic Communications of the EASST, 54*.

Kleppe, A. G., Warmer, J., & Bast, W. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise.* USA: Addison-Wesley Longman Publishing Co., Inc.

Kotonya, G., & Sommerville, I. (1998). *Requirements engineering: processes and techniques.* Wiley Publishing.

Lamsweerde, A. v. (2000). Formal specification: a roadmap. *Proceedings of the Conference on the Future of Software Engineering*, (pp. 147–159).

Lauder, M., Anjorin, A., Varró, G., & Schürr, A. (2012). Bidirectional model transformation with precedence triple graph grammars. *Modelling Foundations and Applications: 8th European Conference, ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings 8*, (pp. 287–302).

Leuschel, M., & Butler, M. (2003). ProB: A model checker for B. *International symposium of formal methods europe*, (pp. 855–874).

Li, Y., Turrini, A., Chen, Y.-F., & Zhang, L. (2018). Learning Büchi Automata and Its Applications. *International Summer School on Engineering Trustworthy Software Systems*, (pp. 38–98).

Lima, V., Talhi, C., Mouheb, D., Debbabi, M., Wang, L., & Pourzandi, M. (2009). Formal Verification and Validation of UML 2.0 Sequence Diagrams using Source and Destination of Messages. *Electron. Notes Theor. Comput. Sci., 254*, 143-160. Récupéré sur http://dblp.uni-trier.de/db/journals/entcs/entcs254.html#LimaTMDWP09

Lutz, M. (2006). *Programming Python.* O'Reilly Media, Inc.

Marsan, M. A., Balbo, G., Conte, G., Donatelli, S., & Franceschinis, G. (1994). *Modelling with Generalized Stochastic Petri Nets* (éd. 1st). USA: John Wiley & Sons, Inc.

Meghzili, S., Chaoui, A., Strecker, M., & Kerkouche, E. (2017). On the verification of UML state machine diagrams to colored petri nets transformation using Isabelle/HOL. *2017 IEEE International Conference on Information Reuse and Integration (IRI)*, (pp. 419–426).

Meghzili, S., Chaoui, A., Strecker, M., & Kerkouche, E. (2019, February). Verification of Model Transformations Using Isabelle/HOL and Scala. *Information Systems Frontiers, 21*. https://doi.org/10.1007/s10796-018-9860-9

Mens, T., & Van Gorp, P. (2006). A taxonomy of model transformation. *Electronic notes in theoretical computer science, 152*, 125–142.

Messaoudi, N., Chaoui, A., & Bettaz, M. (2019). A technique to validate automatic generation of Büchi automata from UML 2 sequence diagrams based on multi layer transformations. *International Journal of Computational Vision and Robotics, 9*, 172–191.

Miles, R., & Hamilton, K. (2006). *Learning UML 2.0.* O'Reilly.

Mozaffari, M., & Harounabadi, A. (2011). Verification and validation of UML 2.0 sequence diagrams using colored Petri nets. *2011 IEEE 3rd International Conference on Communication Software and Networks*, 117-121.

Ng, M. Y., & Butler, M. (2002). Tool Support for Visualizing CSP in UML. Dans *Formal Methods and Software Engineering* (pp. 287–298). Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-36103-0_31

Ng, M. Y., & Butler, M. (2003). Towards formalizing UML State Diagrams in CSP. Dans A. Cerone, & P. Lindsay (Éd.), *1st IEEE International Conference on Software Engineering and Formal Methods (26/09/03)*, (pp. 138–147). Récupéré sur https://eprints.soton.ac.uk/258324/

Nipkow, T., Wenzel, M., & Paulson, L. C. (2002). *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Berlin, Heidelberg: Springer-Verlag.

Nuseibeh, B., & Easterbrook, S. (2000). Requirements engineering: a roadmap. *Proceedings of the Conference on the Future of Software Engineering*, (pp. 35–46).

OMG. (2023, 11 10). OBJECT MANAGEMENT GROUP, https://www.omg.org,. *OBJECT MANAGEMENT GROUP, https://www.omg.org,*.

Owre, S., Shankar, N., Rushby, J. M., & Stringer-Calvert, D. W. (1999). PVS language reference. *Computer Science Laboratory, SRI International, Menlo Park, CA, 1*, 21.

Pham, H. (2006). *System Software Reliability.* Springer.

Promela. (2023). Promela Manual, http://spinroot.com/spin/Man/promela.html, Accessed: 2023-12-05. *Promela Manual, http://spinroot.com/spin/Man/promela.html, Accessed: 2023-12-05*.

Rahmoune, Y., & Chaoui, A. (2022). Automatic bridge between BPMN models and UML activity diagrams based on graph transformation. *Computer Science, 23*.

Rensink, A. (2004). The GROOVE simulator: A tool for state space generation. *Applications of Graph Transformations with Industrial Relevance: Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27-October 1, 2003, Revised Selected and Invited Papers 2*, (pp. 479–485).

Rockstrom, A., & Saracco, R. (1982). SDL-CCITT specification and description language. *IEEE Transactions on Communications, 30*, 1310–1318.

Roscoe, A. W. (1997). *The Theory and Practice of Concurrency.* USA: Prentice Hall PTR.

Roscoe, A. W. (2010). *Understanding concurrent systems.* Springer Science & Business Media.

Roscoe, A. W. (2023). Récupéré sur University of OXFORD: https://www.cs.ox.ac.uk/ucs/

Ruhela, V. (2012). Z formal specification language–an overview. *International Journal of Engineering Research & Technology (IJERT), 1*, 14–27.

Russo, A. G. (2011). Modeling in event-b - system and software engineering by Jean-Raymond Abrial. *ACM SIGSOFT Softw. Eng. Notes, 36*, 38-39. Récupéré sur http://dblp.uni-trier.de/db/journals/sigsoft/sigsoft36.html#Russo11

Scattergood, B., & Armstrong, P. (2011). CSPm: A Reference Manual. *Tech. Rep.*

Selby, P. C., & Selby, R. W. (2007). 4.4. 2 Measurement-Driven Systems Engineering Using Six Sigma Techniques to Improve Software Defect Detection. *INCOSE International Symposium, 17*, pp. 640–651.

Shi, L., Liu, Y., Sun, J., Dong, J. S., & Carvalho, G. (2012). An analytical and experimental comparison of CSP extensions and tools. *International Conference on Formal Engineering Methods*, (pp. 381–397).

Sirjani, M., Movaghar, A., Shali, A., & De Boer, F. S. (2004). Modeling and verification of reactive systems using Rebeca. *Fundamenta Informaticae, 63*, 385–410.

Smith, G. (2012). *The Object-Z specification language* (Vol. 1). Springer Science & Business Media.

Sun, J., Liu, Y., Dong, J. S., & Chen, C. (2009). Integrating Specification and Programs for System Modeling and Verification. *2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering*, (pp. 127-135). https://doi.org/10.1109/TASE.2009.32

Sun, J., Liu, Y., Dong, J. S., & Pang, J. (2009). PAT: Towards Flexible Verification under Fairness. *5643*, pp. 709-714. Springer.

Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., & Ergin, H. (2013). AToMPM: A web-based modeling environment. *Joint proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013): September 29-October 4, 2013, Miami, USA*, (pp. 21–25).

UML. (2017). Unified Modeling Language, https://www.omg.org/spec/UML/2.0/, Accessed: 2023-12-05. *Unified Modeling Language, https://www.omg.org/spec/UML/2.0/, Accessed: 2023-12-05*.

Van Gorp, P. (2008). Model-driven development of model transformations. *International Conference on Graph Transformation*, (pp. 517–519).

Varró, D., Asztalos, M., Bisztray, D., Boronat, A., Dang, D.-H., Geiß, R., . . . others. (2007). Transformation of UML models to CSP: A case study for graph transformation tools. *International Symposium on Applications of Graph Transformations with Industrial Relevance*, (pp. 540–565).

Varro, G., Schurr, A., & Varro, D. (2005). Benchmarking for graph transformation. *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, (pp. 79-88). https://doi.org/10.1109/VLHCC.2005.23

Vidal-Silva, C. L., Villarroel, R., Rubio, J., Johnson, F., Madariaga, E., Campos, C., & Carter, L. (2018). An Spin / Promela Application for Model checking UML Sequence Diagrams. *International Journal of Advanced Computer Science and Applications, 9*. https://doi.org/10.14569/IJACSA.2018.091071

Wang, F. (2004). Formal verification of timed systems: A survey and perspective. *Proceedings of the IEEE, 92*, 1283–1305.

Wenzel, M. (2012, September). Asynchronous Proof Processing with Isabelle/Scala and Isabelle/jEdit. *Electronic Notes in Theoretical Computer Science, 285*, 101–114. https://doi.org/10.1016/j.entcs.2012.06.009

Willink, E. D. (2003). UMLX: A graphical transformation language for MDA. *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture.*

Wing, J. M. (1990). A specifier's introduction to formal methods. *Computer, 23*, 8–22.

Xu, D., Philbert, N., Liu, Z., & Liu, W. (2008). Towards Formalizing UML Activity Diagrams in CSP. *2008 International Symposium on Computer Science and Computational Technology, 2*, 450-453.

Yack-Fa, R. C. (2018). *Vérification formelle de systèmes d'information.* Université de Sherbrooke.

Yangui, R. (2016). *Modélisation UML/B pour la validation des exigences de sécurité des règles d'exploitation ferroviaires Rahma Yangui.* Ecole Centrale de Lille.

Yu, E. S. (1997). Towards modelling and reasoning support for early-phase requirements engineering. *Proceedings of ISRE'97: 3rd IEEE International Symposium on Requirements Engineering*, (pp. 226–235).

Zhao, X., Long, Q., & Qiu, Z. (2006). Model checking dynamic UML consistency. *International Conference on Formal Engineering Methods*, (pp. 440–459).