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 والبحث العلمي
جامعة 20 أوت 1955سكيكدة

Faculté des Sciences

MEMOIRE DE FIN D'ETUDES

Spécialité : Systèmes d'Information Avancés et Applications (SIAA)

Intitulé :

Master en informatique

UNE APPROCHE DE MODÉLISATION ET DE VÉRIFICATION DES DIAGRAMMES UML 2.0 D'ÉTATS-TRANSITIONS EN UTILISANT GROOVE

Présenté par :
Manel Gheribi
Saoussene Bouslama

Encadré par :
Houda HAMROUCHE

2024-2025

Dédicaces

À mes chers parents.

À mon cher mari.

À ma petite fille Célia.

À mes chères sœurs et mon cher frère.

À ma famille et ma belle-famille

À mes chers amis.

Saoussene

Dédicaces

À mes parents, mon premier amour et mon soutien indéfectible
Votre amour, vos sacrifices silencieux et votre confiance sont les fondations de ma réussit

À mes sœurs et à mon frère, votre tendresse qui rassure
et votre présence, toujours réconfortante

À mes amis sincères
Vous êtes les étoiles discrètes qui ont illuminé mes jours d'effort.
Votre soutien, votre écoute et votre bienveillance ont été une force précieuse

À vous tous
Je vous dédie ce travail avec une profonde gratitude et tout mon cœur

Manel

Remerciements

D’abord, nous remercions Dieu le Tout-Puissant, qui nous a accordé la force, la patience et la volonté nécessaires pour mener à bien ce travail.

Nous exprimons notre profonde gratitude à Madame Houda Hamrouche, notre encadrante, pour sa disponibilité, son écoute attentive, ses conseils éclairés et son accompagnement constant tout au long de l’élaboration de ce mémoire.

Nous adressons également nos sincères remerciements aux membres du jury pour avoir accepté d’évaluer ce travail et pour l’intérêt qu’ils y ont porté.

Nos remerciements les plus chaleureux vont à nos parents, pour leur amour inconditionnel, leurs prières, leurs sacrifices et leur soutien sans faille.

À l’ensemble de nos familles, merci pour votre compréhension, vos encouragements et votre présence tout au long de cette aventure.

Ce mémoire est le fruit de tous ces soutiens.
À vous tous, merci du fond du cœur.

Résumé

Ce mémoire s'inscrit dans le cadre de l'Ingénierie Dirigée par les Modèles (IDM), une approche visant à maîtriser la complexité croissante des systèmes logiciels en plaçant les modèles au cœur du processus de développement. Il s'intéresse particulièrement aux diagrammes d'états-transitions UML 2.0 (UML 2.0 STM), utilisés pour modéliser le comportement dynamique des systèmes interactifs. Toutefois, la sémantique informelle de ces diagrammes constitue un frein majeur à leur vérification automatique, en particulier pour les systèmes critiques où la fiabilité est très importante. Communicating Sequential Processes (CSP) est un langage de spécification formel adapté pour décrire les interactions dans les systèmes concurrents.

Dans ce mémoire nous proposons une approche formelle intégrée combinant UML 2.0 STM avec le langage CSP visant à modéliser et à vérifier le comportement des systèmes modélisés. Notre approche repose sur la transformation de graphes et utilise l'outil de modélisation et de vérification GROOVE. Elle propose un méta-modèle des diagrammes UML 2.0 STM (source), un méta-modèle de CSP (cible), un méta-modèle de correspondances établissant les relations entre les modèles source et cible, et une grammaire de graphes. Afin d'illustrer cette approche, nous présentons une étude de cas.

Mots clés : Diagrammes d'états-transitions UML 2.0, CSP, IDM, Grammaires de graphes, GROOVE.

Abstract

This thesis falls within the scope of Model-Driven Engineering (MDE), an approach aimed at managing the growing complexity of software systems by placing models at the core of the development process. It focuses specifically on UML 2.0 state-transition diagrams (UML 2.0 STM), which are used to model the dynamic behavior of interactive systems. However, the informal semantics of these diagrams pose a significant barrier to their automatic verification, especially in critical systems where reliability is paramount. Communicating Sequential Processes (CSP) is a formal specification language well-suited for describing interactions in concurrent systems.

In this thesis, we propose an integrated formal approach that combines UML 2.0 STM with the CSP language to model and verify the behavior of system models. Our approach is based on graph transformation and utilizes the GROOVE modeling and verification tool. It includes a meta-model of UML 2.0 STM diagrams (source), a CSP meta-model (target), a correspondence meta-model establishing the relationships between source and target models, and a graph grammar.

To illustrate this approach, we present a case study.

Keywords : UML 2.0 State diagrams, CSP, MDE, Graph Grammars, Transformation Verification, Groove.

ملـخــص

تندرج هذه الأطروحة ضمن إطار الهندسة الموجهة(MDE)، وهي منهجية تهدف إلى التحكم في التعقيد المتزايد لأنظمة البرمجيات من خلال جعل النماذج محورا رئيسيا في عملية التطوير. وتركز هذه الدراسة بشكل خاص على مخططات انتقال الحالة (UML 2.0 STM)، التي تستخدم لنمذجة السلوك الديناميكي للأنظمة التفاعلية. غير أن الطابع غير الرسمي لدلالات هذه المخططات يشكل عائقا كبيرا أمام التحقق الآلي منها، لا سيما في حالة الأنظمة الحرجة التي تتطلب درجة عالية من الموثوقية. في هذا السياق، تعد العمليات المتسلسلة المتواصلة (CSP) لغة مواصفات رسمية ملائمة لوصف التفاعلات داخل الأنظمة المتزامنة.

في هذه المذكرة نقترح، مقاربة رسمية متكاملة تجمع بين مخططات UML2.0 STMولغة CSPبهدف نمذجة والتحقق من سلوك الأنظمة الممثلة. وتعتمد هذه المقاربة على تحويل الرسوم البيانية كما تستند إلى أداة النمدجة والتحقق GROOVE. وتشمل هذه المقاربة نموذجا ميتا لمخططات UML 2.0 STM(كمصدر)، ونموذجا ميتا للغة CSP(كهدف)، إضافة إلى نموذج ميتا للمراسلات يحدد العلاقات بين النموذجيين المصدر والهدف، فضلا عن قواعد الرسم البياني.

ولتوضيح فعالية هذه المقاربة، نقدم في نهاية العمل دراسة حالة تطبيقية.

الكلمات المفتاحية: CSP, UML 2.0 , قواعد الرسوم البيانية النمدجة ميتا,,مدقق النموذج ,GROOVE

Table des matières

Liste des Figures

Liste des Tableaux

INTRODUCTION GÉNÉRALE

Contexte

L’évolution rapide des systèmes logiciels, notamment dans les domaines critiques et embarqués, impose des exigences de plus en plus élevées en matière de fiabilité, de vérification et de maintenabilité. Face à cette complexité croissante, l’Ingénierie Dirigée par les Modèles (IDM) s’impose comme un paradigme puissant, plaçant les modèles au cœur du processus de développement pour améliorer la conception et la vérification des systèmes logiciels. L'IDM a la capacité d’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.

Problématique et objectif

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.

UML propose un ensemble de diagrammes permettant de représenter à la fois les aspects structurels et comportementaux des systèmes. Les diagrammes d’états-transitions (STM) 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.

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. 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 à l'aide d'un ensemble d'opérateurs.

L'objectif de ce mémoire est de proposer une approche intégrée UML 2.0/CSP pour la modélisation et la vérification des diagrammes STM en utilisant l'outil de modélisation et de transformation de graphes GROOVE ainsi que son modèle checker intégré.

Contribution

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

1. Méta-modélisation
   - Définition d'un méta-modèle des diagrammes STM.
   - Définition d'un méta-modèle pour le langage CSP.
   - Définition d'un méta-modèle de correspondances entre STM et CSP.
2. Transformation des diagrammes STM en définissant une grammaire de graphes pour transformer les diagrammes STM en CSP.
3. Vérification du modèle CSP généré en utilisant le modèle checker intégré de GROOVE.

Organisation du document

Le premier chapitre présente les bases essentielles nécessaires à la compréhension de l'Ingénierie Dirigée par les Modèles. Nous présentant une exploration approfondie de l'IDM, décrivant sa philosophie et ses principes fondamentaux.

Le deuxième chapitre se concentre sur l'introduction des langages de spécification, et plus particulièrement des 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 les langages UML et CSP avec une attention particulière portée à GROOVE et son model-checker intégré.

Le troisième chapitre se consacre à la présentation détaillée de notre approche en exposant une étude de cas portant sur un système de contrôle d'une barrière afin d'illustrer notre approche.

Enfin, nous terminons par une conclusion générale qui récapitule les principaux apports de notre étude.

CHAPITRE1.

L'INGÉNIERIE DIRIGÉE PAR LES MODÈLES

1.1 Introduction

L'ingénierie dirigée par les modèles (IDM), ou *Model Driven Engineering* (MDE), est une approche qui vise à maîtriser la complexité croissante du développement logiciel en plaçant les modèles au cœur du processus de conception. L'une de ses variantes les plus notables est l'architecture dirigée par les modèles (*Model Driven Architecture* – MDA), proposée et soutenue par l'OMG depuis 2000. Cette approche permet d'automatiser une partie du développement grâce à des transformations successives de modèles, facilitant ainsi la génération automatique de code applicatif. En s'appuyant sur des standards bien établis, l'IDM contribue à l'amélioration significative du développement des systèmes complexes en optimisant leur conception et leur mise en œuvre. Parmi les outils dédiés à la transformation de modèles, Groove se distingue particulièrement dans le domaine de la transformation de graphes, mettant en évidence l'importance de ces processus dans l'ingénierie logicielle moderne.

Ce chapitre a pour objectif d'introduire les concepts fondamentaux de la transformation de modèles. Nous y présentons d'abord les différents types de transformations, puis nous nous focalisons sur un cadre spécifique : la transformation de graphes. Enfin, nous explorons les grammaires de graphes et offrons un aperçu de l'outil Groove, utilisé pour ces transformations.

1.2 Concepts de base

Dans le cadre de l'IDM, un système se définit comme un ensemble d'entités en interaction visant à atteindre un objectif. Il peut englober un programme, un système informatique, une combinaison de composants issus de divers systèmes, ainsi que des acteurs humains ou organisationnels. La représentation d'un système repose sur un modèle, qui constitue un élément central de l'IDM. Cette section présente les concepts fondamentaux liés aux modèles, méta-modèles, langages de modélisation et à la méta-modélisation.

1.2.1 Modèle

Un modèle est une représentation abstraite d'un système. Il constitue une description simplifiée du système étudié, mettant en avant certains aspects tout en en délaissant d'autres. Il n'existe pas de définition universellement admise du concept de modèle. La déclaration suivante expose celle formulée par (Bézivin et Gerbé 2001).

« 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)

1.2.2 Méta-modèle

De nombreuses définitions du méta-modèle ont été proposées, mais celle de Da Silva se distingue par sa précision, comme l'illustre l'énoncé suivant :

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

En analysant la Figure 1, nous pouvons approfondir la compréhension du concept de méta-modèle en explorant en détail 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.

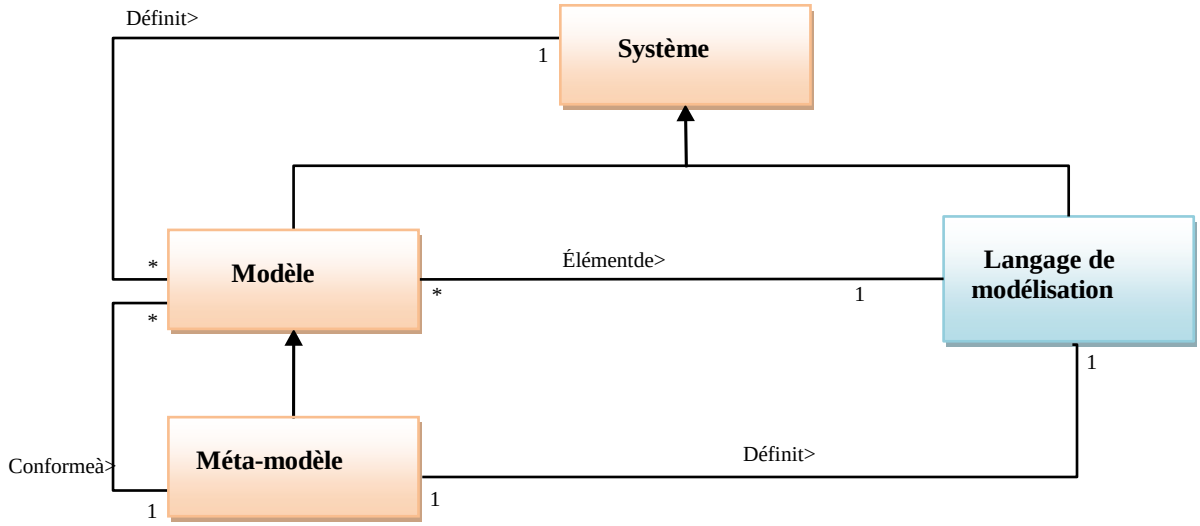


Figure 1: Définition du méta-modèle : relations entre méta-modèle et modèle (Da Silva ,2015)

1.2.3 Méta-méta-modèle

Le concept de méta-modèle joue un rôle fondamental, mais il ne suffit pas à lui seul. La multiplication de méta-modèles variés et incompatibles (data warehouse, workflow, processus logiciel, etc.) a nécessité l'établissement d'un cadre global permettant leur intégration dans les domaines de l'ingénierie logicielle, de l'ingénierie des systèmes et de l'ingénierie des données. La solution a été d'introduire un langage de définition des méta-modèles, où chaque méta-modèle définit un langage spécifique servant à décrire un domaine d'intérêt particulier (Bézivin & Briot, 2004). La Figure 2 illustre les quatre niveaux de modélisation établis par l'OMG.

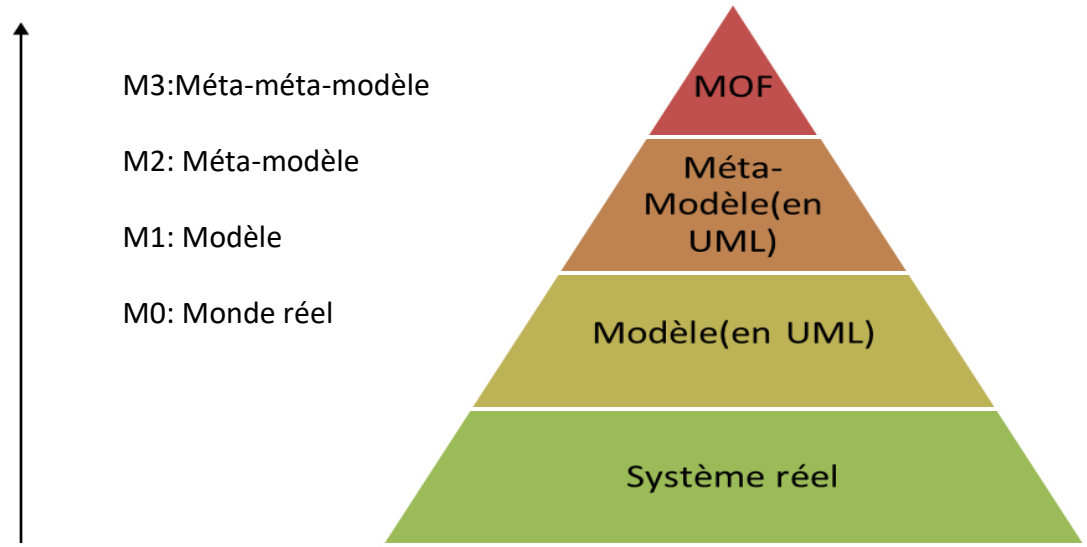


Figure 2: Les quatre niveaux d'abstraction pour le MDA

1.3 Model Driven Architecture

L'Architecture Dirigée par les Modèles(MDA) repose sur la séparation entre la spécification d'un système et les détails de son implémentation. Le processus de développement débute par une description abstraite et indépendante du système, suivie de l'identification des différentes plateformes potentielles pour son déploiement. Une fois la plateforme sélectionnée, la spécification est adaptée en conséquence afin d'assurer une transition fluide vers l'implémentation.

Cette approche poursuit trois objectifs essentiels : la portabilité, qui garantit le fonctionnement du système sur diverses plateformes ; l'interopérabilité, facilitant l'échange d'informations entre systèmes hétérogènes ; et la réutilisabilité, permettant d'exploiter les modèles pour d'autres développements. En rendant les modèles lisibles par machine, MDA apporte une flexibilité accrue tout au long du cycle de développement, facilitant ainsi l'évolution et l'adaptation des systèmes logiciels aux nouvelles exigences technologiques.

Lors de l'implémentation, MDA permet d'intégrer de nouvelles infrastructures ou d'assurer la compatibilité avec des conceptions existantes. De plus, la disponibilité des modèles sous une forme exploitable par les machines simplifie la maintenance et optimise la gestion du système. Enfin, ces modèles offrent la possibilité de générer du code automatiquement, de le valider par rapport aux exigences, de le tester sur différentes infrastructures et de simuler le comportement du système avant sa mise en œuvre. Cette approche améliore ainsi la fiabilité, réduit les erreurs potentielles et garantit une adaptation optimale aux besoins du projet.

1.3.1 classes de modelés

Le MDA repose sur plusieurs modèles qui, dans un premier temps, permettent de représenter le comportement d'un système avant d'être progressivement transformés pour aboutir à la génération du code. Sur le plan conceptuel, il s'articule autour de trois perspectives essentielles, chacune associée à un modèle distinct.

1.3.1.1 Computation Independent Model (CIM)

Le CIM est un modèle conceptuel de haut niveau qui représente l'application en définissant précisément les besoins du client. Son principal objectif est d'établir et de clarifier les relations entre les fonctionnalités de l'application et les différentes entités avec lesquelles elle interagit. Il ne contient aucune information sur l'implémentation, les processus internes ou le comportement du système (Blanc, 2005).

Dans le cadre d'une spécification MDA, le CIM doit être traçable vers les modèles PIM et PSM, qui assurent sa mise en œuvre. De plus, il peut être considéré comme un élément contractuel, servant de référence pour vérifier que l'application développée est bien conforme aux exigences du client (Blanc & Salvatori, 2011).

1.3.1.2 Platform Independent Model (PIM)

Le modèle PIM a pour objectif de structurer l'application en modules et sous-modules. L'analyse et la conception des exigences peuvent être effectuées à l'aide de différentes

méthodes de modélisation, telles que Merise**,** Coad/Yourdon, ainsi que des approches orientées objet comme Schlaer et Mellor, OMT, OOSE et Booch**.** Aujourd'hui, le langage UML s'est imposé comme la référence pour la modélisation des modèles d'analyse et de conception. Ces modèles jouent un rôle essentiel en faisant le lien entre la définition des besoins et le développement du code de l'application. Par ailleurs, ils doivent rester indépendants de toute plateforme ou technologie de mise en œuvre [Blanc, 2005].

1.3.1.3 Platform Model (PM)

Un modèle de plateforme(PM) décrit l'environnement d'exécution d'un logiciel. Il englobe un ensemble de concepts techniques représentant les composants de la plateforme ainsi que les services qu'elle fournit. De plus, il définit les mécanismes par lesquels une application interagit avec cette plateforme.

1.3.1.4 Platform Specific Model (PSM)

Le modèle spécifique à une plateforme(PSM) constitue l'étape la plus délicate du MDA, car il associe les spécifications du PIM aux particularités de la plateforme cible **(**PM) grâce à la transformation de modèle. Il définit comment les fonctionnalités décrites dans le PIM sont concrètement mises en œuvre sur une plateforme spécifique (Belaunde et al., 2003).

Une fois le PSM établi, l'étape suivante du processus de développement consiste à générer le code correspondant, puis à déployer le système dans son environnement cible.

1.3.2 transformation de modèle dans l'approche MDA

Le MDA est une concrétisation de l'approche IDM qui repose sur la séparation entre les spécifications fonctionnelles d'un système et son implémentation sur une plateforme spécifique. Cette approche suit le principe de la séparation des préoccupations, garantissant une distinction claire entre les différents aspects du développement. Du stade de l**'**ingénierie des exigences (IE) jusqu'à l**'**implémentation, le processus de développement orienté MDA considère chaque étape comme un modèle à part entière (Favre et al., 2006).

Les transformations de modèles sont appliquées de manière séquentielle jusqu'à la génération du code final. Dans le cadre du MDA, le code d'une application se compose d'une suite de lignes textuelles, tandis qu'un modèle de code **est** une représentation structurée intégrant divers éléments tels que les boucles, les événements, les composants, les instructions et les conditions. Ainsi, la génération de code à partir d'un modèle de code devient une tâche relativement simple (Blanc & Salvatori, 2011). La Figure3 présente une vue d'ensemble du processus de transformation de modèles dans l'approche MDA.

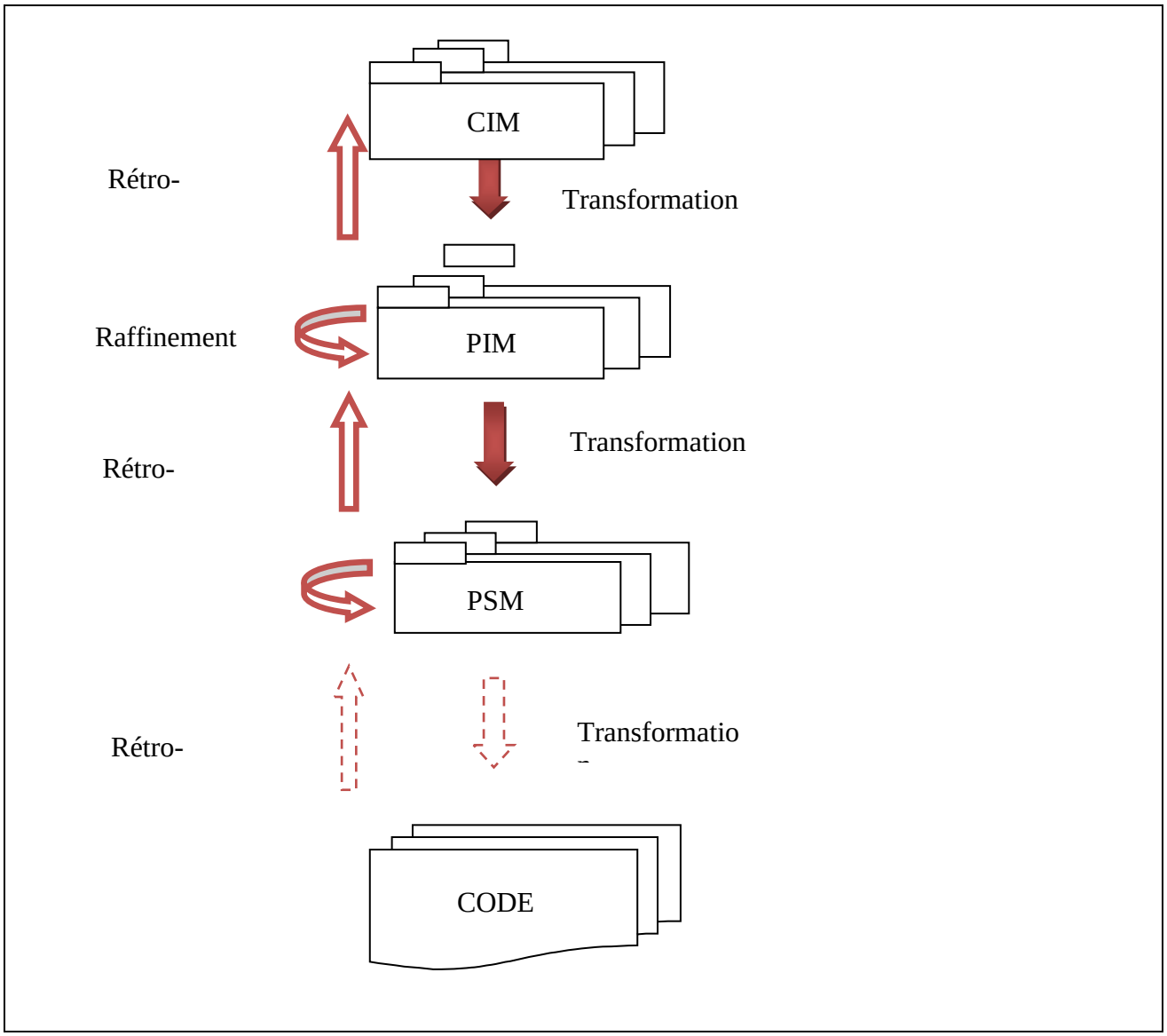


Figure 3: Processus de transformation de modèles de l'approche MDA (OMG03b, 2003)

1.4 Classification des transformations de modèles

Les transformations de modèles se divisent en trois grandes catégories : modèle-vers-modèle(M2M)**,** modèle-vers-texte (M2T)et texte-vers-modèle (T2M), selon la nature des représentations source et cible.

Les transformations M2M permettent de convertir un ou plusieurs modèles sources, conformes à un méta-modèle donné, en un ou plusieurs modèles cibles respectant un autre méta-modèle. Elles se déclinent en quatre principales approches :

- Approche relationnelle/déclarative : Elle définit des relations entre les éléments des modèles source et cible sous forme de prédicats ou de contraintes mathématiques. Parmi les outils utilisant cette approche, on trouve UMLX, BOTL et GRoundTram.
- Approche impérative/opérationnelle : Basée sur une exécution séquentielle d'actions ou de règles définies dans un langage de transformation, elle est implémentée dans des outils comme MoTE, QVTo-Eclipse, Melange et Together.
- Approche basée sur les graphes : Les modèles sont représentés sous forme de graphes typés et attribués, et la transformation s'effectue à travers un ensemble de règles appliquées selon une logique de correspondance structurelle. Chaque règle comprend une partie gauche (LHS - Left-Hand Side), une partie droite (RHS - Right-Hand Side) et éventuellement une condition de non-application (NAC - Negative Application Condition). GROOVE, AToM³, AToMPM, TGG, AGG et DSLTrans sont des outils reposant sur cette méthode.
- Approche hybride : Elle combine plusieurs des stratégies précédentes afin d'optimiser la transformation des modèles. ATL, par exemple, associe les approches relationnelle et impérative, tandis que GrGen.NET et Fujaba intègrent des éléments graphiques et impératifs.

Les transformations M2T, quant à elles, permettent de générer du texte ou du code à partir de modèles. Elles suivent trois stratégies principales **:**

- Approche basée sur le visiteur : Elle parcourt la structure interne du modèle source et génère le texte en fonction des éléments rencontrés. Cette technique exige d'écrire manuellement le code dans les règles de transformation. AToM³ et AToMPM figurent parmi les outils exploitant cette approche.
- Approche basée sur les templates : Elle repose sur la combinaison de parties statiques et dynamiques pour générer du code. Le texte statique reste constant pour tous les modèles, tandis que la partie dynamique adapte le contenu en fonction des éléments du modèle source. Xpand, Acceleo, SPARX* et Henshin* utilisent cette méthode.
- Approche hybride : Fusionnant les concepts des deux précédentes, cette approche tire parti des mécanismes de parcours (visiteur) et de génération basée sur des modèles (templates). Elle est mise en œuvre dans des outils comme Actifsource, MetaEdit+ et Xtend.

Enfin, les transformations T2Mconvertissent du texte en modèles, mais elles sont moins courantes que les deux premières catégories.

1.4.1 Transformation de Graphes

Les graphes et les diagrammes sont des outils puissants et intuitifs pour la modélisation des systèmes complexes. Parmi les exemples les plus courants figurent les diagrammes UML, les réseaux de Pétri et les automates. Grâce à leur capacité de représentation visuelle, les graphes simplifient la compréhension des structures complexes des modèles. Par ailleurs, les techniques de transformation et de réécriture de graphes permettent de formaliser et d'automatiser l'évolution ou la modification de ces modèles.

Une transformation de graphes (Karsai et Agrawal, 2004 ; Andries et al., 1999 ; Rozenberg, 1999) consiste à sélectionner une règle parmi un ensemble défini et à l'appliquer à un graphe. La partie du graphe correspondant à cette règle est alors remplacée par une nouvelle structure. Ce processus se poursuit jusqu'à ce qu'aucune règle ne puisse être appliquée.

Les règles de transformation de graphes constituent une grammaire de graphes, qui généralise la grammaire de Chomsky aux structures graphiques. Chaque règle est composée de deux éléments :

- ✓ LHS (Left-Hand Side) : la partie du graphe initial (host graph) sur laquelle la règle est appliquée.
- ✓ RHS (Right-Hand Side) : la transformation effectuée sur le host graph**.**

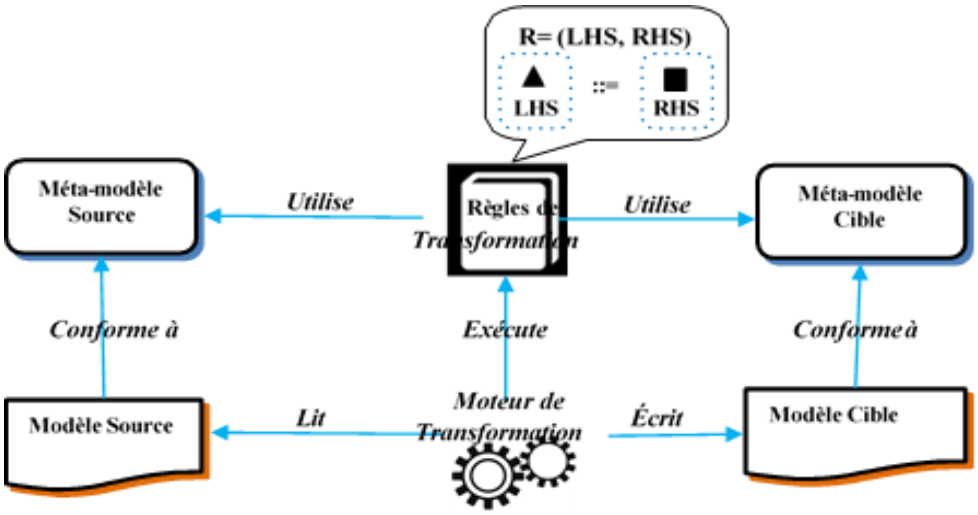


Figure 4: Architecture de transformation de modèle (Hamrouche,2024)

1.4.2 Outils de transformation de Graphes

De nombreux outils de transformation de graphes ont été développés afin de simplifier et d'optimiser les processus de transformation de modèles. (Kahani et al.,2019) ont réalisé une analyse approfondie en répertoriant 60 outils de transformation existants et en les classant selon l'approche de transformation adoptée. Ils ont ensuite comparé ces outils en s'appuyant sur plusieurs critères spécifiques, offrant ainsi une vision structurée et détaillée des solutions disponibles dans ce domaine.

Le Tableau 1 recense divers outils de transformation de graphes en fournissant pour chacun une brève description, le langage de programmation utilisé pour son implémentation, ainsi que les dates de sa première version(PV) et de sa dernière mise à jour(DV) enregistrée jusqu'en 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
AGG (Ermeletal.,1999)	une approche algébrique attribuée à	Java	1997	2017
AToMPM (Syrianietal.,2013)	Successeur d'AToM³ qui génère des outils DSM basés sur le Web	Python	2012	2016
GRoundTram (Hidakaetal.,2011)	Cadre aller-retour basé sur les graphes pour les transformations de modèles bidirectionnels	OCaml	2009	2014
UMLX (Willink,2003)	Prend en charge une syntaxe graphique concrète pour compléter le langage QVT	Java	2005	2017
MoTE (Gieseetal.,2014)	Offre la bidirectionnalité ,la synchronisation et la cohérence des modèles	Java	2010	2016
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
AToM³(Juande Lara 2002)	Un outil de modélisation multi-paradigmes pour les langages visuels	Python	2004	2008

Dans le cadre de cette Mémoire, nous avons opté pour GROOVE afin de développer notre grammaire de graphes. Ce choix s'explique par les nombreux avantages qu'offre cet outil. Tout d'abord, sa simplicité, grâce à une interface intuitive et des fonctionnalités claires,

facilite la conception et l'exécution des règles de transformation. De plus, en tant qu'outil open source, soutenu par une communauté active, GROOVE constitue une solution accessible, pérenne et évolutive.

Sa flexibilité multi-paradigmes permet une méta-modélisation avancée et une adaptation à divers formalismes de graphes, répondant ainsi aux exigences des scénarios complexes. Spécialisé dans les transformations modèle vers modèle, GROOVE excelle dans l'application de règles de réécriture de graphes, garantissant une conversion structurelle fiable tout en préservant la cohérence sémantique.

Enfin, ses fonctionnalités complémentaires, telles que la vérification formelle et la génération d'espaces d'états, renforcent la rigueur des processus de transformation. Grâce à cette combinaison de simplicité, de puissance formelle et d'adaptabilité, GROOVE se révèle être un choix optimal pour répondre aux exigences méthodologiques de cette recherche.

1.4.3 Groove

GROOVE (*Graphs for Object-Oriented Verification*) est un outil puissant dédié à la transformation et à la vérification de graphes. Basé sur une approche formelle et une sémantique dynamique, il utilise des graphes étiquetés simples ainsi que des règles de transformation SPO (Single Push Out). Son principal atout réside dans sa capacité à générer automatiquement un espace d'états sous forme de système de transition étiqueté (STE), où chaque état correspond à un graphe et chaque transition est étiquetée avec le nom de la règle appliquée. Cette approche permet d'analyser et de valider la correction des systèmes modélisés via une vérification basée sur le modèle (*model checking*).

GROOVE offre une large gamme de fonctionnalités avancées, notamment la détection automatique d'isomorphismes, la manipulation d'attributs, l'application contrôlée des règles, ainsi que des stratégies d'exploration sophistiquées de l'espace d'états. Il prend en charge la vérification de propriétés critiques, telles que la sûreté et la vivacité, exprimées en logiques temporelles CTL et LTL.

L'outil repose sur un mécanisme de transformation structuré autour de règles bien définies. Une règle en GROOVE comprend plusieurs composants visuels :

- Un patron devant être présent dans le graphe source pour permettre l'application de la règle (en noir).
- Un patron interdit, dont la présence empêche l'application de la règle (en rouge, optionnel).
- Les éléments supprimés après l'application de la règle (en bleu).
- Les éléments ajoutés au graphe après l'application de la règle (en vert).

GROOVE est composé de plusieurs modules essentiels : un simulateur avec interface graphique pour la création et l'édition de systèmes de transformation de graphes, un générateur en ligne de commande pour produire l'espace d'états, un visualiseur permettant d'examiner les graphes de manière indépendante, ainsi qu'un générateur d'images pour exporter les graphes sous différents formats (JPG, GIF, SVG, PDF). Grâce à ses capacités avancées, sa flexibilité et son intégration de concepts formels, GROOVE constitue un outil de référence pour l'analyse et la validation des systèmes complexes reposant sur la transformation de graphes. La Figure5 présente l'interface graphique de l'outil

Groove.

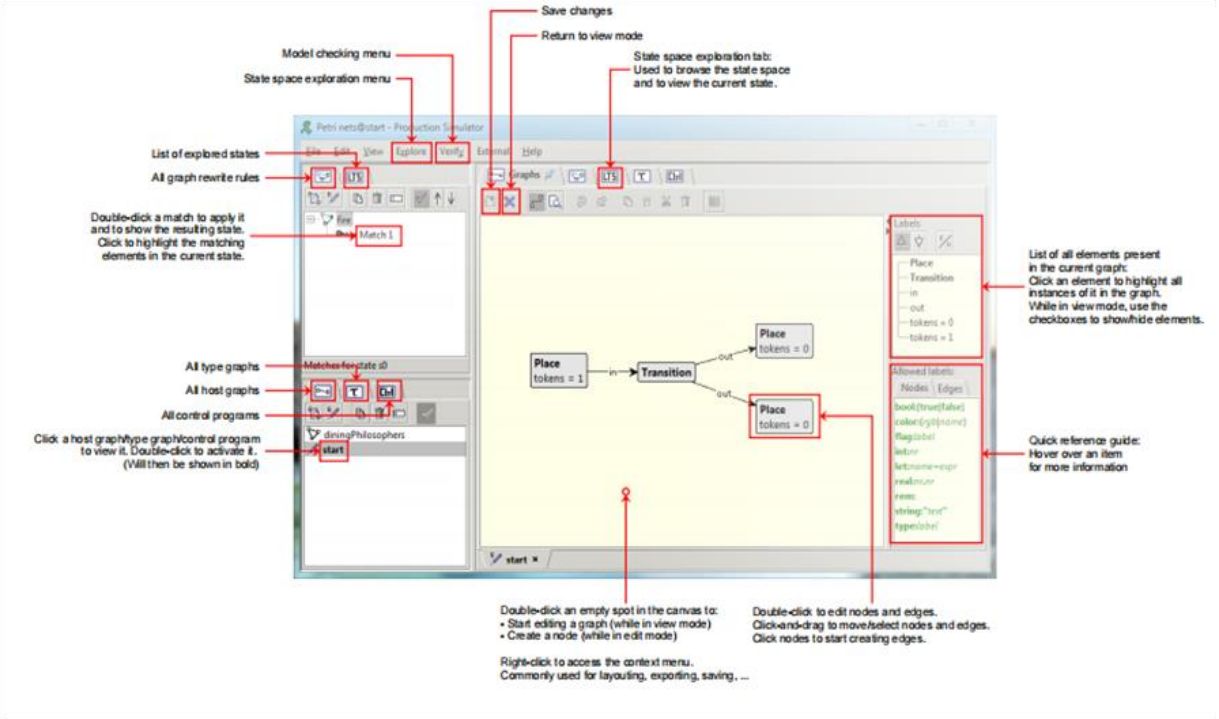


Figure 5: L'interface graphique de l'outil Groove

1.5 Conclusion

Dans ce chapitre, nous avons exploré l'ingénierie dirigée par les modèles en introduisant ses concepts fondamentaux, notamment les notions de modèle, méta-modèle et méta-méta-modèle. Nous avons ensuite abordé l'architecture dirigée par les modèles en détaillant les différentes classes de modèles ainsi que les mécanismes de transformation propres à cette approche. Par la suite, nous avons présenté la classification des transformations de modèles, en mettant un accent particulier sur la transformation de graphes. Dans ce cadre, nous avons examiné plusieurs outils dédiés à cette tâche, avant de nous focaliser sur GROOVE, un outil spécialisé dans la transformation de graphes.

Les concepts présentés dans ce chapitre constituent un background nécessaire pour la compréhension de nos contributions dans le cadre de cette thèse, lesquelles seront exposées dans le chapitre suivant.

.

CHAPITRE 2. MODÉLISATION ET VÉRIFICATION FORMELLE DE SYSTÈMES

Introduction

Ce chapitre présente les différents langages et méthodes utilisés pour la spécification et la vérification des systèmes. Il débute par une classification des langages de spécification, allant des langages informels aux langages formels. Ensuite, il s'intéresse au langage de modélisation unifié UML, en abordant ses différents types de diagrammes et la version UML 2.0 STM. Le langage formel CSP est également étudié, notamment sa syntaxe, ses approches sémantiques et les outils qui lui sont associés. Par la suite, les principales méthodes de vérification sont décrites, incluant le test, la simulation et les techniques formelles. Enfin, le chapitre se conclut par la présentation de l'outil Groove, un model checker intégré.

2.2 Langages de spécification

La modélisation peut être catégorisée en fonction du niveau de formalisme des langages ou des méthodes utilisés dans le processus. Elle peut ainsi être qualifiée de formelle, semi-formelle ou informelle (Celso et al.). La table de la figure (6) ci-dessous présente une définition des catégories de langages ainsi que des exemples de langages ou de méthodes qui les utilisent.

Catégories de Langage			
Langage Informel		**Langage Semi-Formel**	**Langage Formel**
Simple	Standardisé		
Langage qui n'a pas un ensemble complet de règle pour restreindre une construction	Langage avec une structure, un format et des règles pour la composition d'une construction.	Langage qui a une syntaxe définie pour spécifier les conditions sur lesquelles les construction sont permises.	Langage qui possède une syntaxe et une sémantique définies rigoureusement il existe un modèle théorique qui peut être utilisé pour valider une construction.
Exemple de Langage ou Méthode			
Langage Naturel.	Texte Structuré en Langage Naturel	Diagramme Entité-Relation, Diagramme à Objets.	Réseaux de Petri, Machines à états finis, VDM,Z.

Figure 6: Classification et utilisation des langages ou des méthodes (Celso et al.)

2.2.1 Langages informels

La modélisation informelle repose sur l'utilisation d'un langage naturel, ce qui lui confère une expressivité supérieure aux langages semi-formels ou formels. Son principal atout réside dans sa facilité de compréhension, favorisant ainsi un consensus entre les parties prenantes, qu'il s'agisse des concepteurs ou des commanditaires du projet. De plus, son caractère intuitif en fait un moyen de communication privilégié entre les acteurs du développement logiciel. C'est d'ailleurs pour ces raisons que les professionnels de l'industrie continuent à l'employer pour spécifier les exigences. Cependant, cette approche présente des limites notables. L'absence de structure rigoureuse rend difficile toute standardisation et engendre une certaine imprécision. De plus, l'ambiguïté inhérente aux langages informels peut conduire à des interprétations multiples, ce qui complique leur traitement automatique et augmente le risque d'erreurs, notamment lors de la phase de conception. (Hamrouche,2024)

2.2.2 Langage semi- formels

Les langages semi-formels possèdent une structure syntaxique bien définie, mais leur interprétation peut varier en raison d'une sémantique parfois ambiguë. Comparés aux langages formels, ils offrent une expressivité plus grande, notamment grâce à l'utilisation de notations graphiques ou textuelles facilitant leur manipulation. Toutefois, cette expressivité accrue engendre des défis lors de leur analyse, nécessitant parfois une conversion en langages formels. UML et SADT sont des exemples courants de langages semi-formels.

Le processus de modélisation semi-formelle repose sur l'usage de représentations graphiques ou textuelles dotées d'une syntaxe rigoureuse et d'une sémantique souvent partielle. Malgré cela, il permet d'effectuer des vérifications et d'automatiser certaines tâches. La puissance des modèles graphiques joue un rôle clé dans la plupart des approches de modélisation semi-formelles. En particulier, UML est fréquemment utilisé pour produire des modèles clairs et interprétables.

2 .2.3 Langages formels

Un langage formel se caractérise par une syntaxe et une sémantique rigoureusement définies, souvent exprimées à l'aide de notations mathématiques. Cette précision permet d'assurer la vérification automatique des spécifications, garantissant ainsi la cohérence et la complétude des systèmes développés. Grâce aux techniques de modélisation formelle, il est possible de formaliser un système de manière explicite et de mener des vérifications rigoureuses sur ses propriétés. Toutefois, l'utilisation des langages formels requiert une expertise approfondie, ce qui peut constituer un frein pour les non-spécialistes. Malgré cette complexité, ils jouent un rôle clé dans le développement de systèmes fiables, notamment lorsqu'ils sont combinés avec des approches de traduction facilitant leur interprétation par les utilisateurs.

Kesraoui (2017) a proposé une classification des langages formels en s'appuyant sur des classifications préexistantes. Il a notamment combiné et adapté les travaux d'Almeida et al. (2011) et de Lamsweerde (2000), tout en conservant la distinction entre les langages basés sur les modèles et ceux basés sur les propriétés. Cette classification est illustrée à la Figure 7.

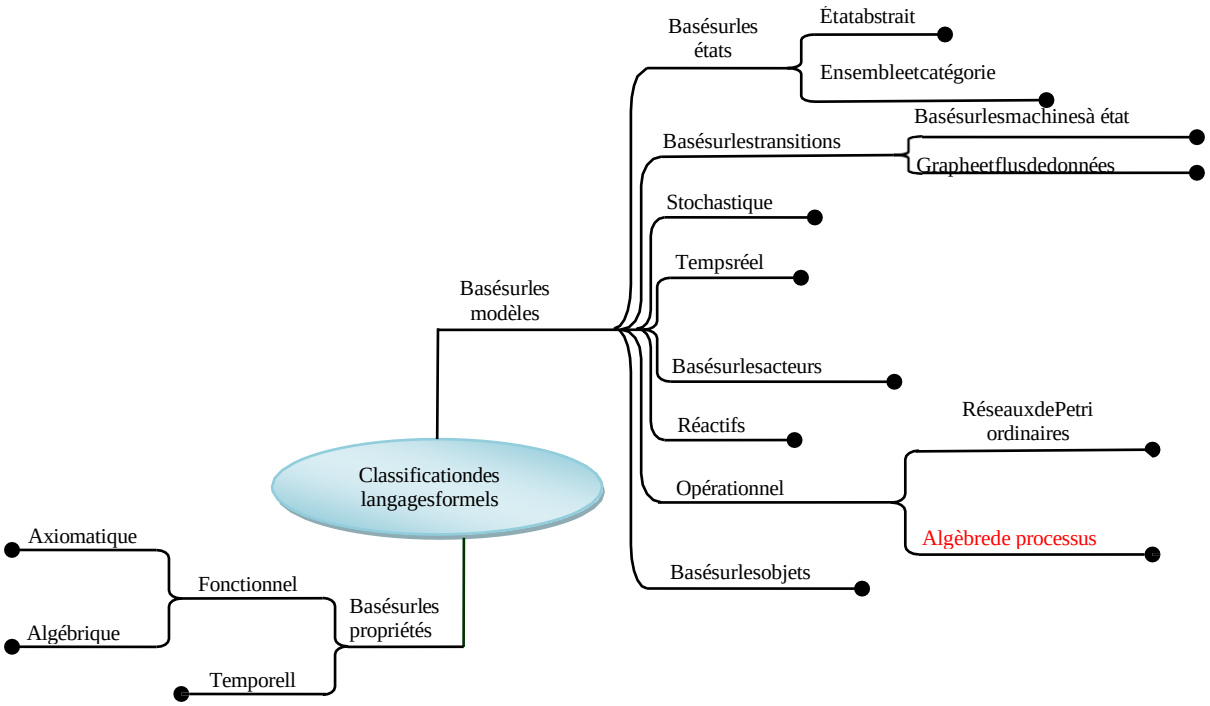


Figure 7 : Classification des langages formels (Kesraoui, 2017) .

2.3 Unified Modeling Language

UML (Unified Modeling Language) est un langage de modélisation standardisé par l'OMG (Object Management Group). Il a été conçu pour fournir aux développeurs et aux architectes de système des outils adaptés à l'analyse, la conception et la mise en œuvre de systèmes logiciels, ainsi qu'à la modélisation de processus métier et d'autres domaines. Il repose sur une approche orientée objet et utilise divers types de diagrammes permettant de représenter aussi bien les aspects statiques que dynamiques d'un système.

Issu de la fusion des principales méthodes de modélisation objet des années 1990, notamment OMT (Object Modeling Technique) de James Rumbaugh, OOSE (Object-Oriented Software Engineering) d'Ivar Jacobson et la méthode de Grady Booch, UML est avant tout un langage et non une méthode. Il permet de spécifier, visualiser, construire et documenter les artefacts des systèmes informatiques et non informatiques. Son adoption par l'OMG en 1997 dans sa version 1.1 a marqué un tournant dans la standardisation des pratiques de modélisation, et son évolution vers UML 2.0 en 2006 a renforcé son rôle en ingénierie logicielle. Aujourd'hui,

UML est un outil essentiel pour la modélisation de systèmes complexes dans divers domaines, allant de l'informatique aux processus métier en passant par la biologie et la mécanique. La Figure 8 illustre les différentes étapes d'évolution d'UML.

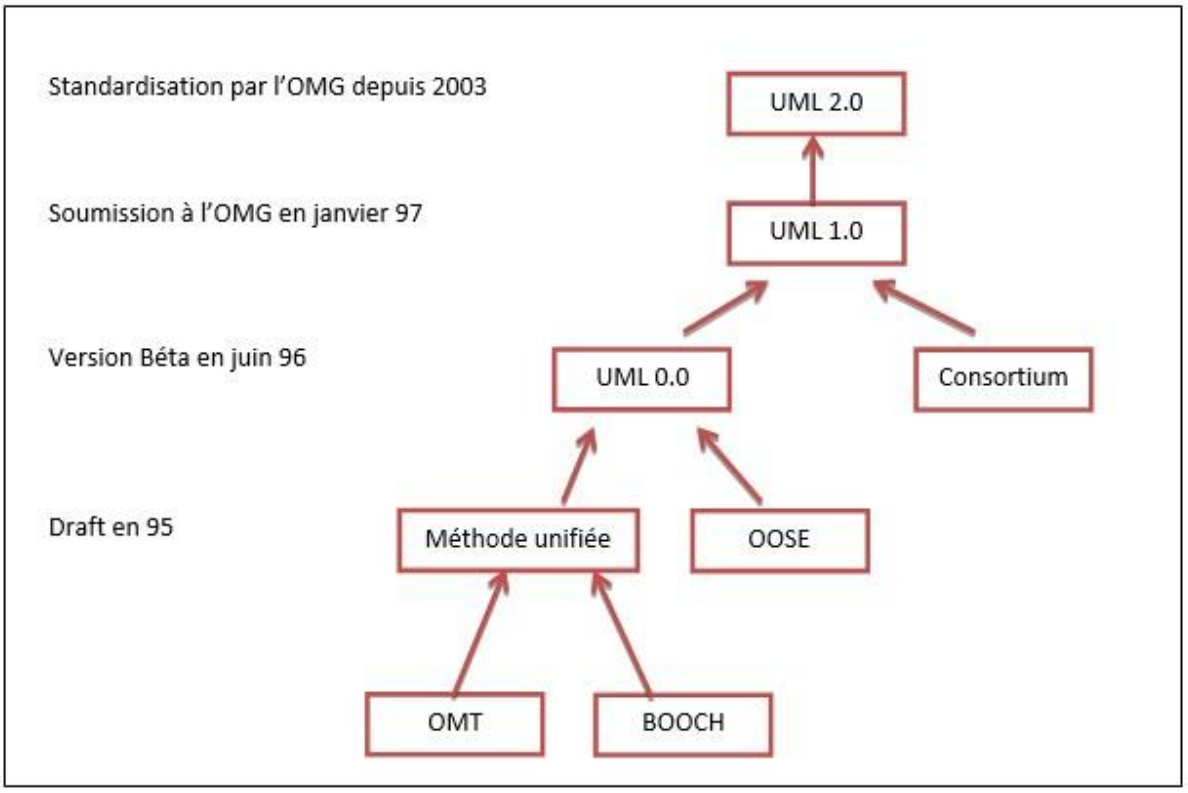


Figure 8: Évolution d'UML

2.3.1 Les diagrammes UML

Le langage UML 2.0 propose treize types de diagrammes permettant de modéliser les différents aspects d'un système. Ces diagrammes se divisent en deux grandes catégories : les diagrammes structurels (ou statiques) et les diagrammes comportementaux (ou dynamiques).

Les diagrammes structurels décrivent la structure statique d'un système à différents niveaux d'abstraction et d'implémentation. Parmi eux, on retrouve :

- Les diagrammes de classes, qui représentent les classes, leurs attributs, opérations et relations.
- Les diagrammes d'objets, qui illustrent des instances spécifiques d'un diagramme de classe à un moment donné.
- Les diagrammes de paquetages, qui organisent et regroupent les éléments du modèle, notamment les classes et cas d'utilisation.
- Les diagrammes de composants, qui montrent la structure logicielle d'un système en mettant en avant ses composants et interfaces réutilisables.
- Les diagrammes de déploiement, qui détaillent la répartition des composants logiciels sur le matériel physique du système.
- Les diagrammes de structures composites, qui permettent de représenter la structure interne d'un classificateur et les relations entre ses composants.

Les diagrammes comportementaux, quant à eux, modélisent les interactions et dynamiques du système, incluant la communication entre objets et acteurs. Parmi eux, on distingue :

- Les diagrammes de cas d'utilisation, qui décrivent les interactions entre le système et ses utilisateurs.
- Les diagrammes d'activités, qui modélisent des processus métiers ou le comportement d'un cas d'utilisation.
- Les diagrammes d'états-transitions, qui détaillent les changements d'état d'un objet en réponse à des événements.
- Les diagrammes de séquence, qui illustrent les échanges entre objets selon un ordre temporel précis.
- Les diagrammes de communication, une variante simplifiée des diagrammes de séquence axée sur les interactions entre objets.
- Les diagrammes de temps, qui représentent l'évolution d'un objet ou d'une donnée au fil du temps.
- Les diagrammes d'interaction globale, qui combinent des éléments de diagrammes d'activités et de séquence pour illustrer le flux d'interactions.

Grâce à cette diversité de diagrammes, UML constitue un outil puissant pour l'analyse, la conception et la documentation des systèmes, qu'ils soient logiciels ou non. Dans cette étude, nous nous intéressons aux diagrammes d'états-transitions et détaillons leur principe de fonctionnement.

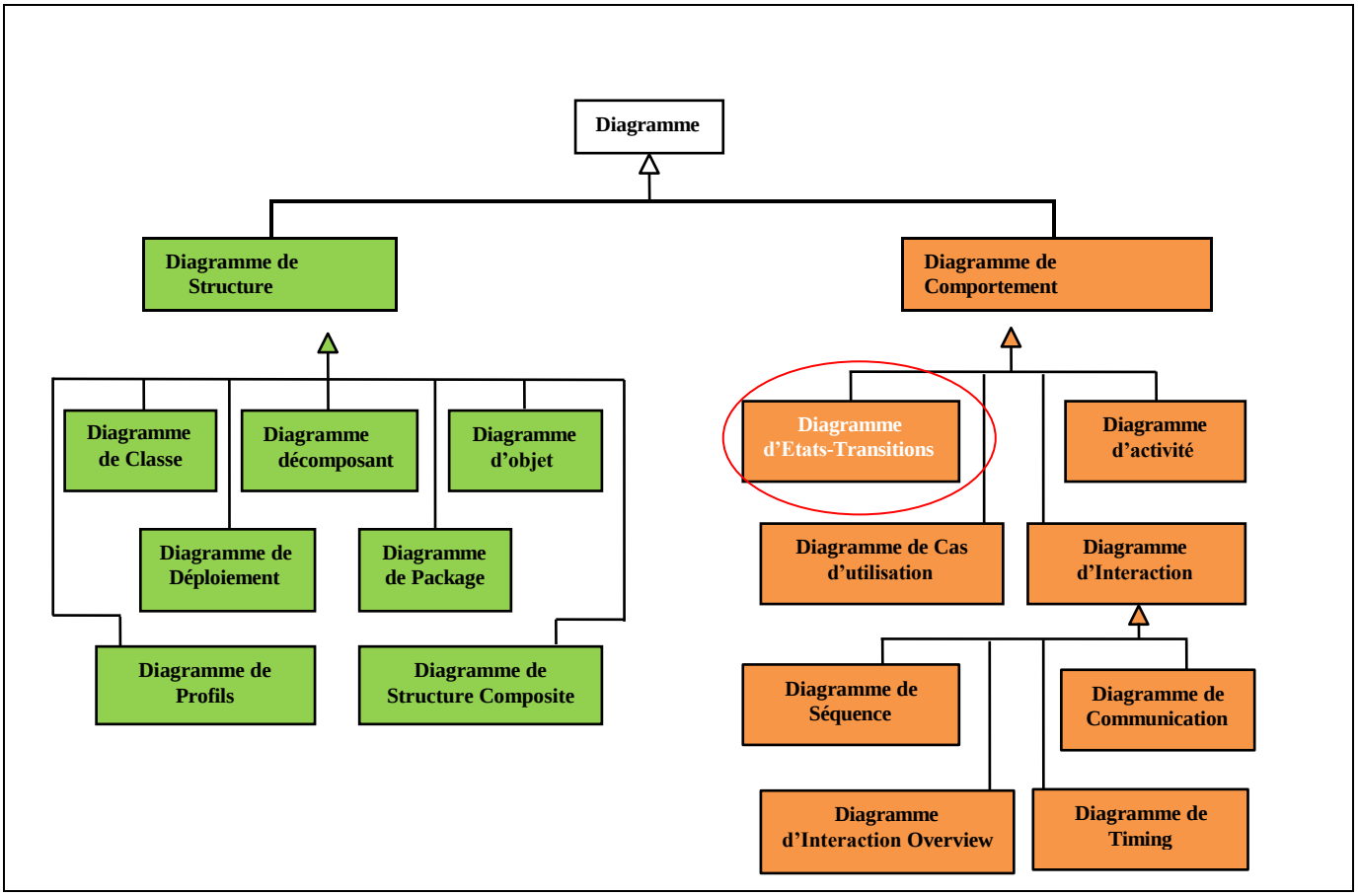


Figure 9: Taxonomie des diagrammes de structure et de comportement d'UML (UML, 2017).

2.3.2 UML 2.0 STM

Les diagrammes d'états-transitions constituent une évolution orientée objet des Statecharts classiques, développés au fil des années depuis l'introduction du formalisme par Harel. Ils prennent la forme d'un graphe permettant de représenter et de modéliser l'évolution de l'état d'un objet ainsi que son comportement en réponse à l'arrivée d'événements. Les Statecharts illustrent une machine à états en mettant en avant les états possibles et les transitions entre eux. Un état se distingue par sa durée et sa stabilité, tandis qu'une transition correspond à un

passage instantané d'un état à un autre. Cette transition peut être déclenchée soit par un événement, soit automatiquement en l'absence d'un déclencheur spécifique. Toutefois, ces diagrammes ne possèdent pas de sémantique explicitement formalisée.

La modélisation à l'aide des diagrammes d'états-transitions présente plusieurs avantages :

- Décrire un processus, notamment dans le cadre d'un workflow.
- Représenter un aspect du modèle dynamique.
- Faciliter la conception des activités et des interfaces homme-machine (IHM).
- Aider à l'élaboration de scénarios de test.
- Déterminer les états permettant de gérer les risques de dysfonctionnement.
- Offrir une visualisation claire du système en réduisant sa complexité.

Le fonctionnement des diagrammes d'états-transitions en UML repose sur le même principe qu'un automate fini. Autrement dit, chaque entité se trouve à tout moment dans un état donné et peut évoluer vers un autre état par le biais d'une transition conditionnelle précisément définie. Dans ce qui suit, nous présentons les éléments essentiels des diagrammes d'états-transitions, suivis d'un exemple illustrant ces concepts dans la figure 9.

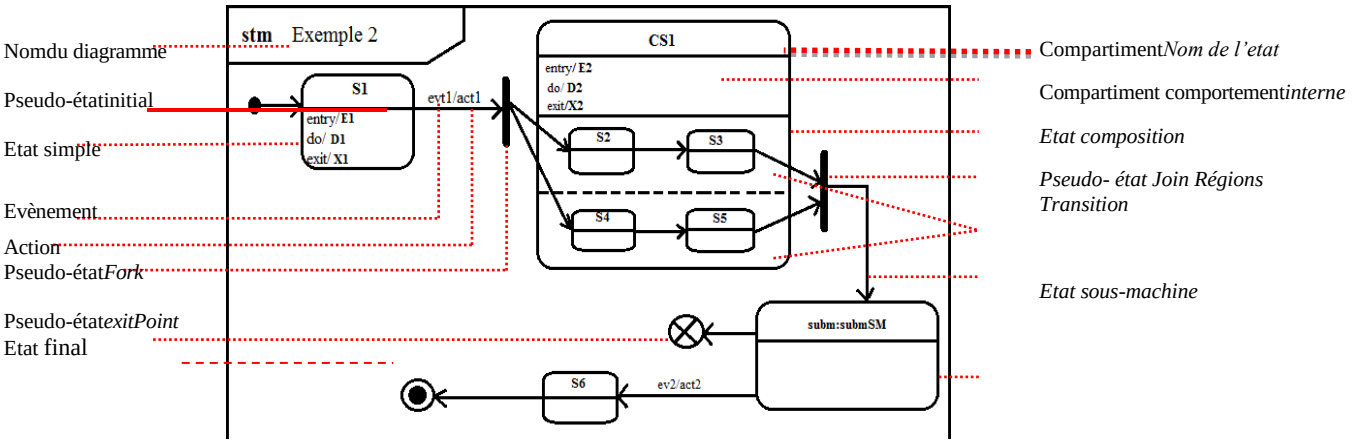


Figure 10: Exemple d'un diagramme d'états-transitions (Hamrouche,

UML définit trois types d'états, qui peuvent contenir une ou plusieurs régions : les états simples, les états composites et les sous-états. Un sous-état correspond à une machine d'états complète pouvant être imbriquée dans un autre état.

- État racine (root state) : C'est un état qui ne dépend d'aucun état composite
- État simple : Il ne contient aucune région, ce qui signifie qu'il n'a ni état interne ni transitions.
- État composite : Il inclut au moins une région et peut être soit simple, soit orthogonal.
    - Un état composite simple possède une seule région, qui peut contenir d'autres états, permettant ainsi une hiérarchisation des diagrammes d'états-transitions.
    - Un état orthogonal : comprend plusieurs régions pouvant elles-mêmes contenir des états, ce qui permet de représenter des exécutions simultanées. Ces régions sont séparées par une ligne pointillée. Tout état composite doit contenir au moins une région et chaque région doit comprendre au moins un état.
- État final : Il représente un état particulier indiquant la fin d'une région une fois atteint.
- État initial (pseudo-état) : Il marque le point de départ d'une région. Chaque diagramme d'états-transitions et chaque état composite (qu'il soit simple ou orthogonal) peut comporter un état initial. Une transition issue de cet état peut avoir un comportement défini, mais elle n'est associée ni à un événement ni à une condition.
- État historique : Il mémorise le dernier sous-état actif d'un état composite. Il est représenté par un cercle contenant la lettre "H". Par exemple, la région supérieure de S1 dans la figure 9 possède un état historique. Un état composite peut contenir au maximum un état historique par région.

Dans ce que suit, nous présentons les notions de transitions externes et internes.

- Transition externe : Elle exprime la réponse d'un objet à un événement. Par exemple, la transition de l'état S1 vers l'état S2, étiquetée avec l'événement "b" dans la figure 9, est une transition externe.
- Transition interne : Elle est spécifiée directement dans le compartiment de l'état auquel elle est associée. Contrairement aux transitions externes, elle ne cible aucun autre état et ne modifie pas l'état actif après son déclenchement. Par exemple, les transitions *entry /bEn1* et *exit/bEx1*dans l'état S1(figure 9) sont des transitions internes.

Un événement correspond à une occurrence spécifique dans le domaine étudié et constitue une information instantanée à traiter immédiatement. Les transitions définissent les relations orientées entre les états, tandis que les événements déterminent le moment où ces transitions doivent se produire. Lorsqu'un événement est reçu, il déclenche une transition, amenant l'objet d'un état source vers un état cible.

Différents types d'événements sont définis dans UML :

- Événement signal : Il est déclenché par une communication asynchrone entre deux objets (un objet émetteur envoie un signal, et un objet récepteur le reçoit).
- Événement d'appel : Il survient lorsqu'un objet invoque une opération. Les paramètres de cette opération sont ceux de l'événement d'appel. Contrairement aux événements signaux, les événements d'appel sont définis comme des méthodes dans le diagramme de classes.
- Événement de changement : Il se produit lorsqu'une expression booléenne devient vraie. Tant que la condition reste fausse, la transition ne se déclenche pas. L'expression est continuellement évaluée jusqu'à ce qu'elle devienne vraie, moment où la transition est activée.
- Événement temporel : Il est basé sur le passage du temps et peut être défini de manière absolue (à une date précise) ou relative (après un certain délai). Par défaut, le temps commence à s'écouler dès que l'objet entre dans l'état source de la transition.

2.4 CSP

Le CSP (Communicating Sequential Processes) est un modèle formel conçu pour représenter des systèmes concurrents dans lesquels plusieurs processus s'exécutent simultanément tout en interagissant. Il repose sur une approche mathématique qui permet d'analyser la synchronisation et la communication entre processus, facilitant ainsi la vérification du comportement du système (Roscoe A. W., 1997).

Dans CSP, les processus sont autonomes, mais ils interagissent en échangeant des événements. L'ensemble des événements qu'un processus P peut générer ou recevoir

constitue son alphabet. Ces interactions sont définies par des opérateurs algébriques, qui structurent la communication et la synchronisation entre les processus.

2.4.1 Syntaxe du CSP

CSP propose une syntaxe basée sur des opérateurs permettant de formaliser les comportements des processus. Ces opérateurs sont décrits comme suit :

$P \triangleq P \ |Stop|\ Skip|\ e \rightarrow P|\ if\ b$ then P else $Q|P[]Q|P|\sim|Q|P \backslash X|P\ ;\ Q|P[X \| Y]Q|[|X|]Q|P|||Q.$ (1)

Avec :

- **P, Q :** représentant des processus,
- **X, Y :** désignant des ensembles d'événements,
- **e :** un événement,
- **b :** une condition booléenne.

Une explication détaillée de chaque opérateur est disponible dans le Tableau 2.

Tableau 2: Description des opérateurs CSP (Hamrouche, 2024)

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	*Préfix e*:exécutel'événement*e* puisse 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*.

2.4.2 Les approches sémantiques

Trois principales approches permettent d'interpréter un programme CSP d'un point de vue mathématique (Roscoe A. W., 1997).

2.4.2.1 Modélisation Opérationnelle

Cette approche met l'accent sur l'exécution concrète d'un programme CSP. Elle repose sur des diagrammes de transition, où chaque état évolue selon des actions visibles ou invisibles. Cela permet d'étudier le comportement des processus concurrents et d'analyser leur évolution en fonction des interactions possibles.

2.4.2.2 Approche Dénotationnelle

La sémantique dénotationnelle attribue à chaque programme CSP une représentation abstraite, facilitant l'analyse de son comportement à partir de ses éléments constitutifs. Trois modèles sont principalement utilisés :

- Le modèle de traces : il capture les séquences d'événements échangés entre un processus et son environnement (traces(P)).
- Le modèle échecs/traces : il enrichit le modèle précédent en intégrant les événements qu'un processus peut refuser après certaines séquences d'exécution.
- Le modèle échecs/divergences : il prend en compte la notion de divergence, où un processus peut entrer dans une boucle infinie d'actions internes sans interagir avec son environnement.

2.4.2.3 Perspective Algébrique

Cette approche repose sur un ensemble de règles et d'axiomes établissant des équivalences entre processus. Grâce à ces lois mathématiques, il devient possible de simplifier et d'analyser le comportement des systèmes CSP de manière rigoureuse et formelle.

2.4.3 Outils CSP

Plusieurs outils ont été développés pour la conception, l'analyse et la validation des modèles basés sur CSP. Parmi les plus répandus :

- FDR (Failures Divergences Refinement) : un vérificateur de raffinement analysant les traces et échecs des systèmes CSP à états finis.
- ProBE (Process Behavior Explorer) : un animateur permettant d'explorer l'espace d'états d'un processus CSP, basé sur la même technologie que FDR.
- Checker : est un vérificateur indépendant pour CSPM, offrant une analyse avancée des structures et des types de processus. Bien qu'il puisse entraîner des exécutions lentes et générer des sorties redondantes en raison de sa gestion spécifique des événements et des types de données, il demeure un outil puissant. Son approche détaillée en fait un choix privilégié pour les utilisateurs expérimentés cherchant une vérification approfondie des programmes CSPM.
- ProB : un outil d'analyse des spécifications B, intégrant CSPM et permettant d'effectuer des vérifications telles que l'analyse de LTL, la détection de blocages et le raffinement de trace.
- ARC (Adélaïde Refinement Checker) : développé par l'université d'Adélaïde, il propose des fonctionnalités de raffinement, d'animation et de vérification des processus CSP.
- PAT (Process Analysis Toolkit) : une plateforme avancée offrant une flexibilité accrue dans l'interprétation de CSP, notamment avec la gestion des variables partagées.
- CSP Prover : un outil intégrant CSP avec le démonstrateur de théorèmes Isabelle, permettant la preuve formelle de certaines propriétés des modèles CSP.

Après avoir comparé plusieurs model-checkers pour CSP, nous avons opté pour Checker. Cet outil se distingue par sa capacité à analyser avec précision les types CSPM, bien qu'il requière une certaine expertise pour une utilisation optimale. Grâce à son approche rigoureuse, il nous permettra d'examiner en profondeur la conformité des modèles et d'améliorer la fiabilité de notre analyse.

2 .5 Les méthodes de vérification

La vérification a pour objectif de garantir que le développement du système respecte les exigences spécifiées. Elle permet d'identifier d'éventuelles erreurs à chaque étape du processus et d'assurer la conformité du système aux attentes définies. Selon la norme ISO 9000, la vérification repose sur des preuves objectives pouvant être obtenues par différentes méthodes, telles que l'inspection, les calculs alternatifs ou l'analyse de documents. Ces preuves peuvent provenir d'observations, de mesures, de tests ou d'autres moyens d'évaluation (ISO 9000, 2015).

2.5.1 Le test

Le test est une méthode de vérification partielle qui permet d'identifier des erreurs en soumettant le système à des scénarios préétablis. Il est généralement effectué après l'intégration du système, une phase où la correction des erreurs peut être particulièrement coûteuse. Malgré cet inconvénient, il reste l'une des méthodes de vérification les plus répandues dans l'industrie, notamment en raison de sa capacité à être appliqué à des systèmes complexes et de grande envergure. Contrairement à d'autres approches, comme la simulation ou les méthodes formelles, qui reposent sur une abstraction du système, le test permet d'analyser directement son comportement réel (Kesraoui, 2017).

2.5.2 La simulation

La simulation est une méthode de vérification reposant sur la modélisation du comportement du système sous une forme abstraite, exécutée sur des plateformes spécialisées. Elle permet

d'explorer différents scénarios et d'évaluer les performances du système avant son implémentation effective.

Grâce à sa flexibilité, la simulation s'adapte particulièrement à l'analyse des systèmes de grande envergure. Son principal atout réside dans la possibilité de l'appliquer dès les premières phases du développement, bien avant l'intégration et l'implémentation. Cette anticipation permet d'identifier les erreurs en amont et d'en réduire le coût de correction. Toutefois, la simulation présente certaines limites, notamment en ce qui concerne le réalisme des modèles et la précision des résultats obtenus (Kesraoui, 2017).

2.5.3 Les méthodes de vérification formelle

Les méthodes de vérification formelle permettent d'analyser de manière rigoureuse les spécifications exprimées dans des langages formels afin de garantir que le système en développement respecte les exigences définies. Fondées sur des principes mathématiques, elles offrent une précision et une exhaustivité supérieures aux approches basées sur les tests ou la simulation, réduisant ainsi le risque d'erreurs.

Bien que leur mise en œuvre puisse varier selon la méthode employée, le processus de vérification formelle suit généralement les étapes suivantes :

- Définition des propriétés : Il s'agit de formaliser les exigences du système sous forme de propriétés à vérifier. Ces propriétés sont exprimées à l'aide de logiques mathématiques ou de modèles formels qui décrivent le comportement attendu du logiciel.
- Modélisation du système : Une fois les propriétés définies, le système est représenté sous forme d'un modèle mathématique adapté. Cette modélisation peut faire appel à des automates, à des langages formels ou à d'autres structures abstraites en fonction de la complexité et de la nature du logiciel.
- Vérification et correction : Des techniques spécifiques sont appliquées pour analyser la conformité du modèle aux exigences définies. Si des anomalies sont détectées, des ajustements sont effectués, et la vérification est réitérée jusqu'à ce que le système respecte pleinement les critères établis.

Grâce à leur rigueur, les méthodes formelles sont particulièrement adaptées aux systèmes critiques, tels que les logiciels embarqués ou les infrastructures de sécurité, où la fiabilité et l'absence de défauts sont essentielles

2.6 Le model-checker intégré Groove

GROOVE est un outil spécialisé dans la modélisation et la vérification des systèmes basés sur des graphes. Il intègre un model checker, un mécanisme permettant d'analyser automatiquement si un système respecte une propriété spécifiée en explorant son espace d'états de manière exhaustive et systématique.

Dans GROOVE, la vérification repose sur la représentation des transformations de graphes sous forme de systèmes de transition étiquetés, où chaque état correspond à une configuration du graphe et chaque transition représente l'application d'une règle de transformation.

Le model checker de GROOVE suit plusieurs étapes pour évaluer ces transformations, comme illustré dans la Figure 10 :

- Construction du système de transition : À partir d'un état initial, GROOVE génère l'ensemble des états accessibles en appliquant les règles de transformation définies.
- Spécification des propriétés à vérifier : L'utilisateur exprime les exigences sous forme de formules logiques, notamment en LTL (Linear Temporal Logic) ou CTL (Computation Tree Logic), afin de caractériser le comportement attendu du système.
- Exploration et validation : Le model checker parcourt l'espace d'états et détermine si les propriétés spécifiées sont satisfaites.
- Résultats et contre-exemples : Si la propriété est validée, l'outil répond "oui", confirmant la conformité du système. En cas de non-conformité, il retourne "non" et génère un contre-exemple, mettant en évidence un scénario où la propriété est violée.

Grâce à cette approche, GROOVE permet une vérification rigoureuse et automatisée des transformations de graphes, facilitant la détection des erreurs dès la phase de modélisation et réduisant ainsi les coûts et les risques liés à l'implémentation du système.

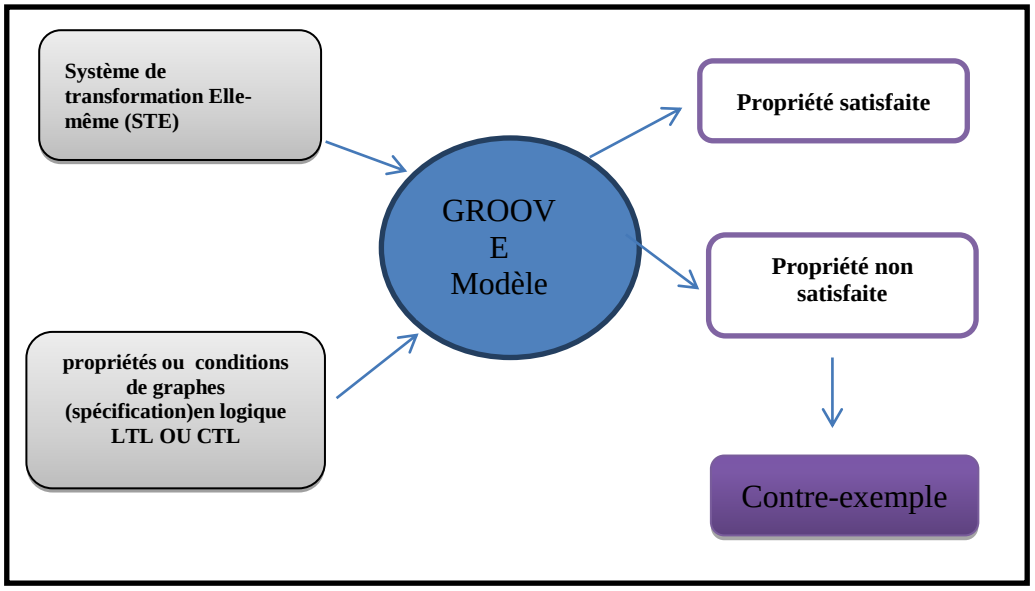


Figure 11: Principe général de l'approche GROOVE model-checking

2.7 Conclusion

Ce chapitre explore les différents langages de spécification, allant des langages informels aux langages formels, en passant par les semi-formels. Il présente ensuite UML et ses diagrammes, ainsi que CSP avec sa syntaxe, ses approches sémantiques et ses outils. Les méthodes de vérification, incluant le test, la simulation et les techniques formelles, sont également abordées. Enfin, l'intégration du model checker Groove est examinée, soulignant son rôle dans la vérification des modèles.

CHAPITRE 3. L'APPROCHE INTÉGRÉE UML 2.0 /CSP PROPOSÉE

3.1 Introduction

Les diagrammes UML sont largement utilisés pour la modélisation des systèmes logiciels, en particulier pour décrire des comportements complexes. Cependant, leur nature semi-formelle limite leur capacité à permettre une vérification rigoureuse. Dans ce contexte, cette étude vise à intégrer UML 2.0 avec un langage de spécification formelles à savoir le CSP, afin de combiner la lisibilité graphique d'UML avec la précision mathématique de CSP.

Nous commençons par une comparaison entre les méthodes formelles et semi-formelles, suivie d'un aperçu des travaux connexes dans ce domaine. Ensuite, nous abordons la formalisation des diagrammes STM avant de présenter l'approche intégrée UML/CSP proposée.
Nous détaillons ensuite les méta-modèles proposés et la grammaire de graphes assurant la modélisation et la transformation des diagrammes STM vers le CSP. Une étude de cas illustre notre approche.

3.2 Méthodes formelles vs Méthodes semi-formelles

Selon Bjorner (2014)**,** une méthode formelle est définie comme une approche dont les techniques et outils peuvent être exprimés de manière mathématique. Par exemple, si elle inclut un langage de spécification, celui-ci doit posséder une syntaxe formelle, une sémantique rigoureuse et un système de preuve bien défini. Les techniques associées aux méthodes formelles permettent de concevoir, d'analyser et/ou de transformer une ou plusieurs spécifications en un programme.

En revanche, les méthodes semi-formelles reposent sur des langages de spécification textuels ou graphiques, caractérisés par une syntaxe bien définie mais une sémantique plus souple. Le Tableau 3 résume les principales différences entre ces deux approches (Idani, 2006).

Tableau 3: Méthodes formelles Vs Méthodes semi-formelles (Idani, 2006).

	Méthodes semi-formelles	Méthodes formelles
Formalisme	Textuel ou graphique (Merise, SADT, UML,…)	Mathématique (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

L'élaboration d'une méthode garantissant le développement de systèmes logiciels automatiquement vérifiables à partir de spécifications claires peut être obtenu en combinant des langages de spécification formels et semi-formels. Cette intégration peut être mise en œuvre selon différentes stratégies.

3.3 Travaux connexes

Dans le cadre des efforts visant à doter les différents diagrammes UML d'une sémantique formelle, les diagrammes d'états-transitions ont particulièrement retenu l'attention en raison de leur rôle central dans la modélisation du comportement dynamique des systèmes. À cet égard, Ng et Butler ont proposé deux contributions majeures, qui se distinguent par leur rigueur méthodologique et leur indépendance vis-à-vis de toute plateforme d'implémentation, en utilisant le langage formel CSP.

Le premier travail, présenté dans Ng et Butler (2002), introduit une formalisation des diagrammes d'états-transitions ainsi que des diagrammes de classes en CSP. Cette approche permet de représenter de manière précise les états, les transitions et les événements déclencheurs, tout en respectant la sémantique définie par UML. Elle offre ainsi un cadre

solide pour la vérification formelle des comportements dynamiques dans les systèmes modélisés.

Le second travail, publié dans Ng et Butler (2003), visent à compléter cette formalisation en intégrant la notion d'états composites, essentielle à la représentation de comportements hiérarchiques complexes. Cette extension renforce la capacité du langage CSP à modéliser fidèlement les spécifications UML, en particulier dans les systèmes embarqués et temps réel.

Outre les contributions de Ng et Butler, plusieurs autres chercheurs ont également proposé des approches pertinentes pour la formalisation des diagrammes d'états-transitions UML. Latella et al. (1999) ont défini une sémantique formelle des machines à états UML basée sur des automates étiquetés, ouvrant la voie à l'application de techniques de model-checking. Hausmann (2002) a proposé une transformation systématique des diagrammes d'états UML en réseaux de Petri, facilitant la vérification de propriétés temporelles. Harel et Politi (1998), quant à eux, sont à l'origine de la formalisation des *Statecharts*, qui ont inspiré les diagrammes d'états UML, et ont établi une base formelle robuste adaptée à la vérification comportementale.

De leur côté, Lilius et Paltor (1999) ont présenté une formalisation opérationnelle des diagrammes d'états UML, accompagnée d'un outil de simulation et de vérification basé sur le model-checker SPIN. Enfin, Sheng et al. (2005) ont utilisé le langage formel B pour représenter les états et les transitions, permettant ainsi la vérification de propriétés de sûreté.

Ces différentes contributions ont considérablement enrichi le domaine de la formalisation des diagrammes d'états-transitions UML, en offrant des approches complémentaires pour la modélisation, la simulation et la validation formelle des systèmes logiciels complexes.

3.4 Formalisation des diagrammes UML 2.0 STM

L'approche de transformation d'UML 2.0 STM en CSP, présentée par Ng et Butler (2003), repose sur la conversion de chaque état en un processus et de chaque événement ou action

UML en un événement CSP. Le Tableau 4 illustre les correspondances entre ces deux langages, source et cible.

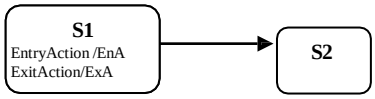
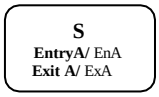
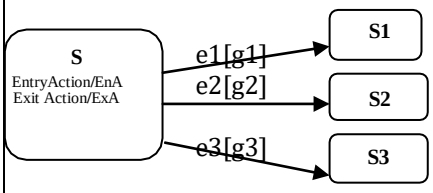
Tableau 4: L'approche de transformation d'UML 2.0 STM en CSP.

État	Processus CSP
Évènement UML	Évènement CSP
Action	Évènement CSP

Le Tableau 5 présente la description des processus CSP générés pour chaque composant des diagrammes STM.

Tableau 5: Formalisation d'UML 2.0 STM en CSP (Ng et Butler ,2003)

UML2.0STM		**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** EntryAction/EnAExit 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)*

3.5 Approche intégrée UML 2.0 /CSP proposée

L'approche proposée repose sur la combinaison de la méta-modélisation et de la transformation de graphes. Pour sa mise en œuvre, nous avons utilisé l'outil GROOVE. Les phases de réalisation de cette approche sont les suivantes :

1. Méta-modélisation
   - Définition d'un méta-modèle des diagrammes STM.
   - Définition d'un méta-modèle pour le langage CSP.
   - Définition d'un méta-modèle de correspondances entre STM et CSP.
2. Transformation des diagrammes STM en définissant une grammaire de graphes pour transformer les diagrammes STM en CSP.
2. Vérification du modèle CSP généré en utilisant le model-checker intégré de GROOVE.

La Figure 12 illustre l'architecture de l'approche proposée.

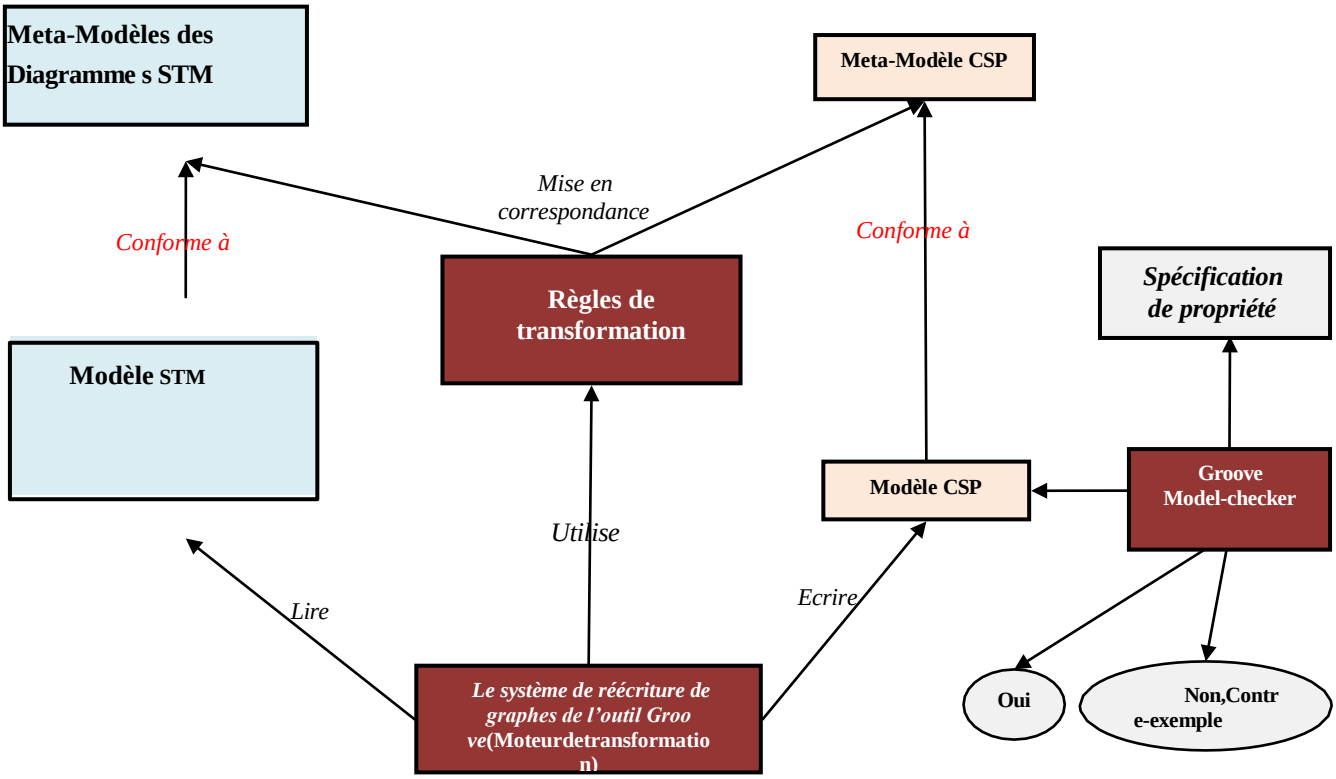


Figure 12: L'architecture de l'approche proposée

3.5.1.1 Méta-modélisation UML

La figure 13 présente un méta-modèle UML pour les diagrammes d'états-transitions, utilisé pour modéliser le comportement dynamique d'un système. Le diagramme débute par un InitialState, représentant l'état de départ, qui mène à un état simple via une transition définie par la relation startsWith. Chaque « SimpleState » peut être enrichi par une EntryAction (action exécutée à l'entrée de l'état) et une ExitAction (action exécutée à la sortie). Un STM se termine éventuellement par un état final « FinalState » à travers la relation endsWith. Les transitions entre les états sont modélisées par l'élément « Transition », qui peut être conditionné par une Guard (condition logique), déclenché par un « EventUML », et associé à une ou plusieurs « Action ». Le méta-modèle permet également de représenter des

« CompositeState », qui héritent des « SimpleState », et permettent d'imbriquer plusieurs sous-états, offrant ainsi une structuration hiérarchique plus complexe du comportement.

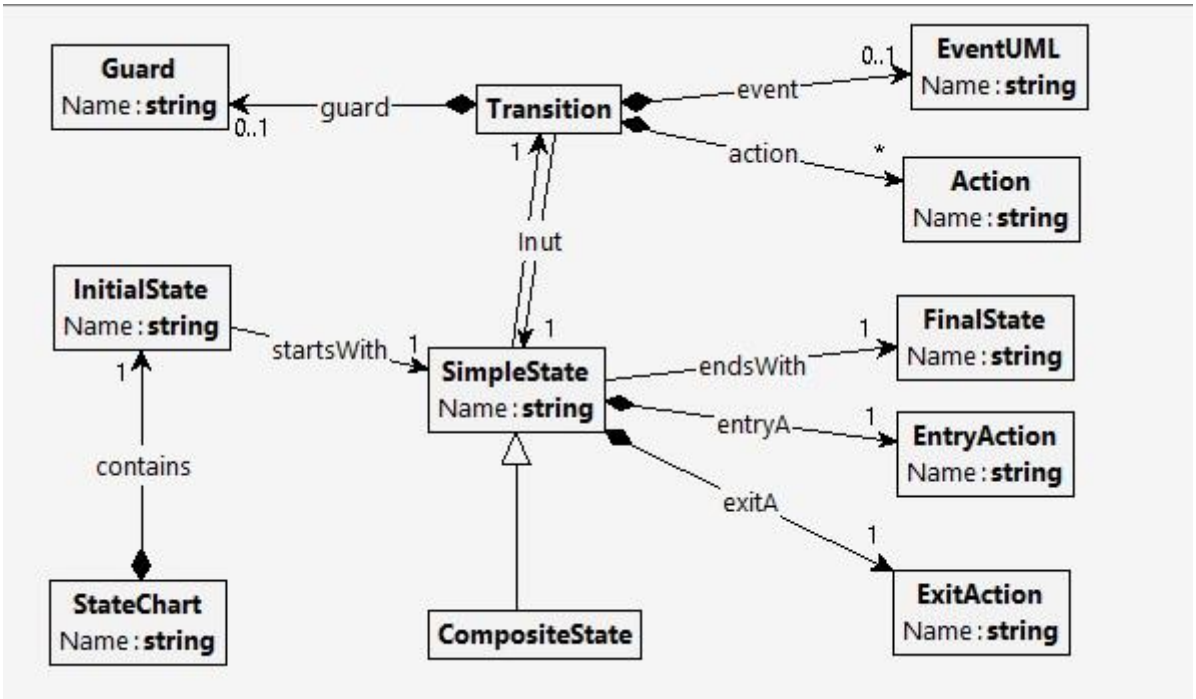


Figure 13: Méta-modèle des diagrammes STM dans Groove

3.5.1.2 Méta-modélisation des processus CSP

Le méta-modèle CSP illustré dans la figure 14 décrit une structure hiérarchique pour la modélisation des processus concurrents, avec « CSPContainer » comme racine contenant des « ProcessAssignment ». Chaque « ProcessAssignment » lie un identifiant de processus à une expression CSP « ProcessExpression », pouvant être un Prefix, une condition, une concurrence, un operateur binaire (choix externe, hiding), un processus ou un « SKIP »**.** Ces expressions permettent de modéliser des comportements complexes comme les séquences, les conditions ou le parallélisme. Les transitions entre états sont déclenchées par des événements ou des conditions.

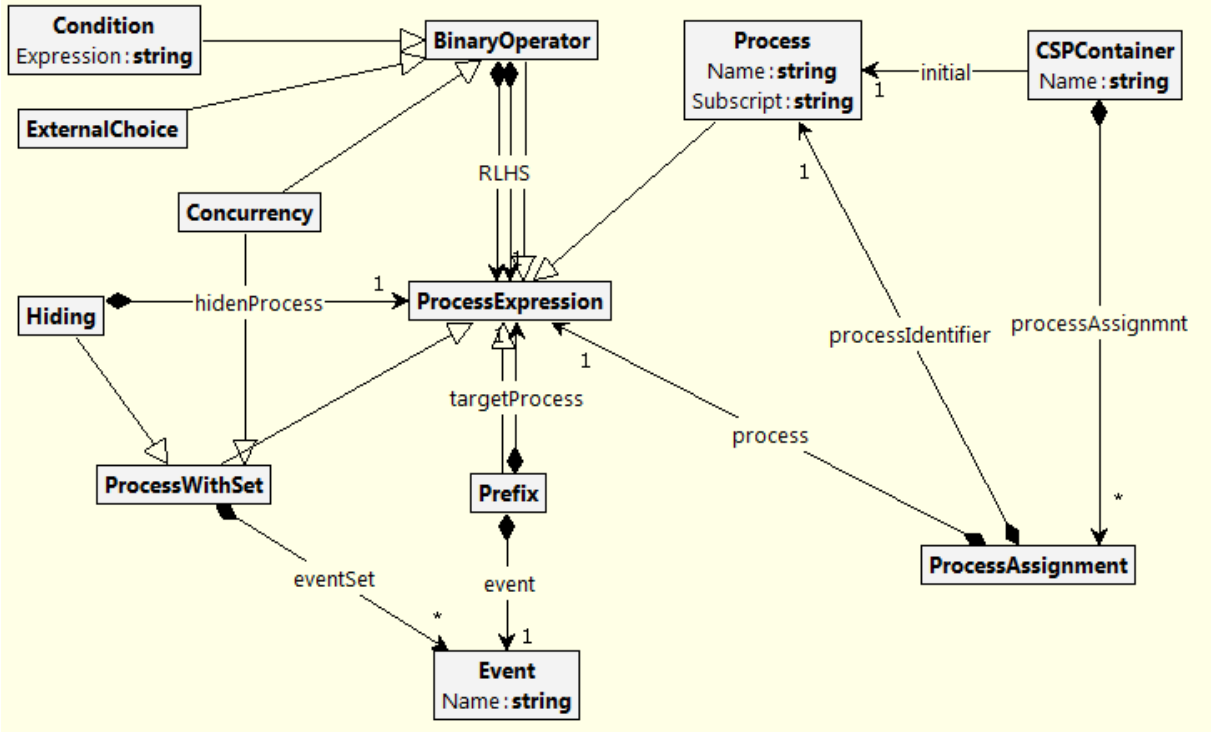


Figure 14: Méta-Modèle du CSP dans Groove

3.5.1.3 Méta-modèle de Correspondance

Le méta-modèle complet de cette transformation est présenté dans la figure 15 ; il définit la relation entre les diagrammes STM et les processus CSP à l'aide d'arêtes supplémentaires, appelées arêtes de correspondance. Ces arêtes servent à établir un lien explicite entre les éléments structurels des deux formalismes. Par exemple, une arête étiquetée « to » relie un nœud de transition dans le diagramme STM à un nœud de processus dans le modèle CSP. Ce mécanisme de correspondance permet de synchroniser la logique de contrôle décrite dans le STM avec les comportements formalisés dans CSP, facilitant ainsi la traduction et la vérification du modèle.

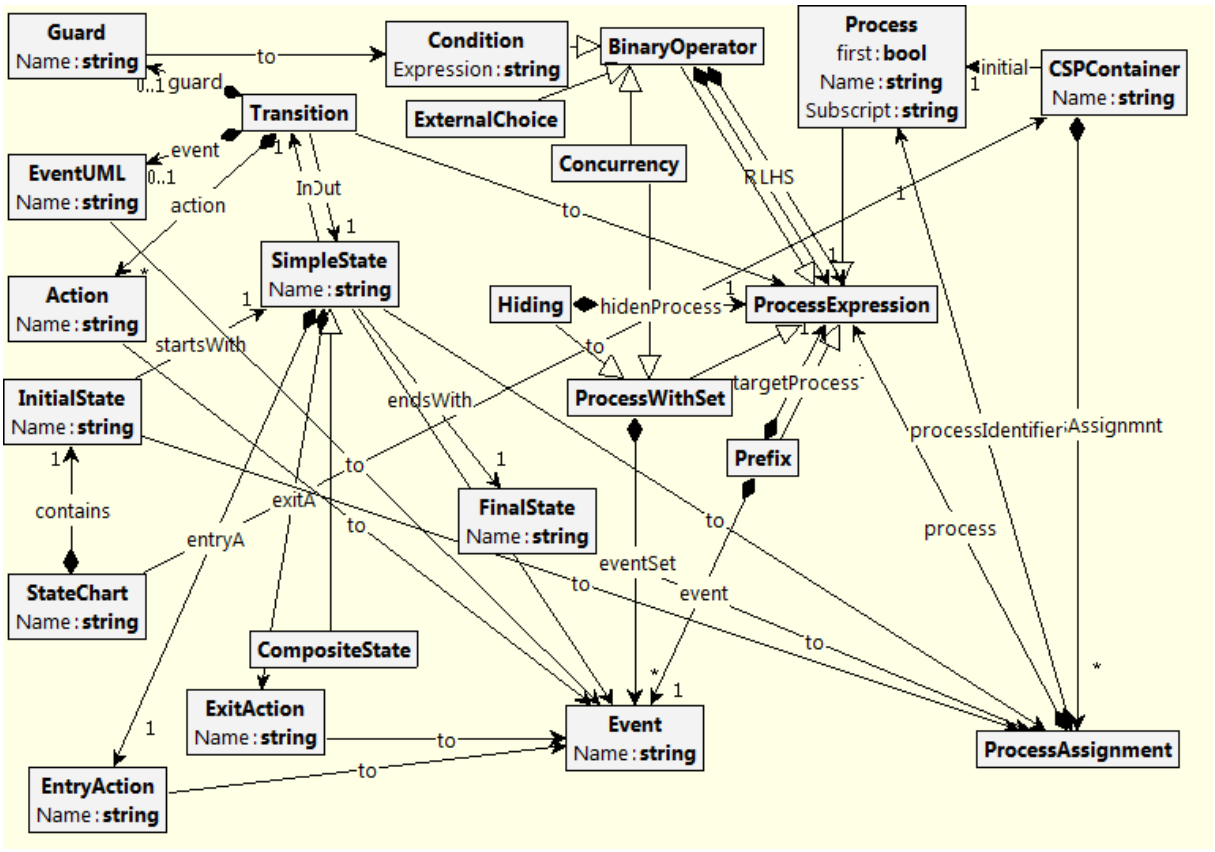


Figure 15: Méta-modèle de correspondances STM/CSP

3.5.2Transformation des diagrammes STM en CSP

Pour transformer les diagrammes STM en processus CSP, nous proposons 11 Règles. Ces règles s'appliquent par ordre décroissant de priorité.

Règle 1 : InitialState (Priorité 11). Cette règle (Figure 16) signifie que l'état initial (*InitialState*) dans un diagramme UML est traduit en une affectation de processus (*ProcessAssignment*) Dans le langage CSP. Le nom de l'état simple (*SimpleState*) associé est utilisé pour déterminer le processus de départ, représentant ainsi le point d'entrée du comportement dans la modélisation CSP.

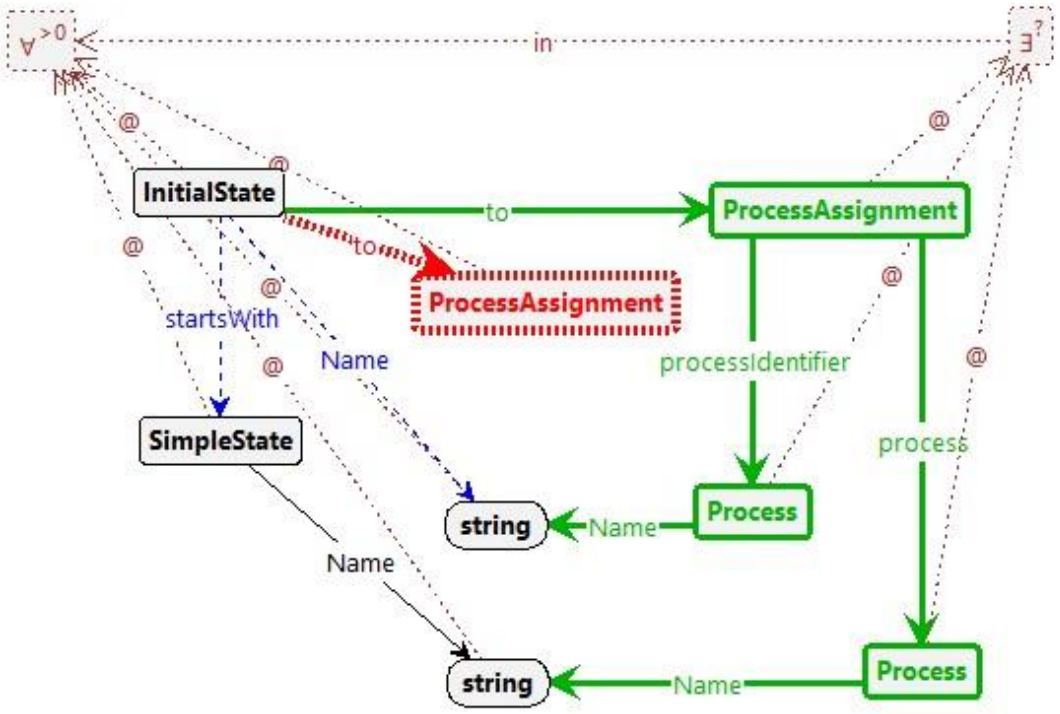


Figure 16:Règle 1(InitialState)

Règle 2 : State2ProcessAssignment (Priorité 10). Cette règle (Figure 17) signifie que chaque état simple d'un diagramme d'état transition est traduite en une affectation de processus (ProcessAssignment) en CSP.

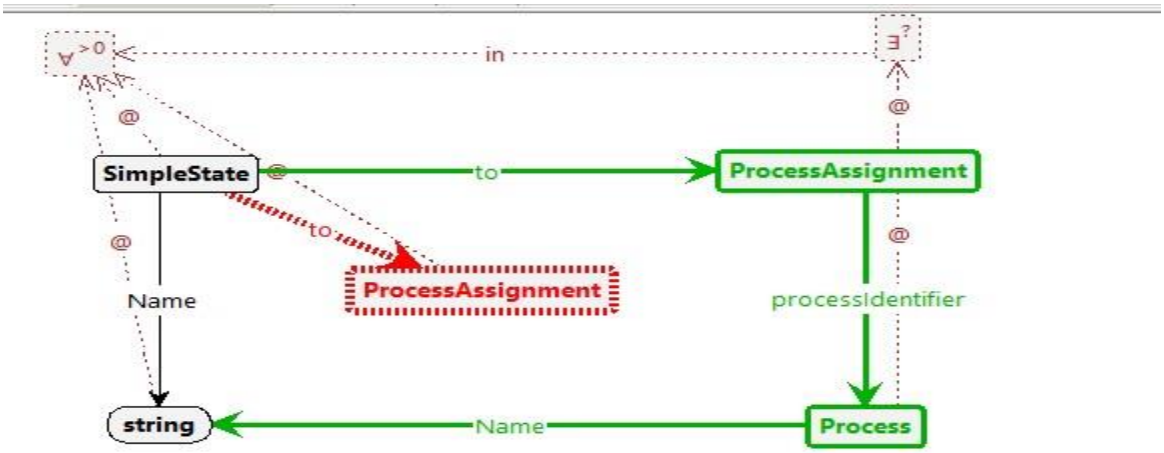


Figure 17:Règle 2 (Stat2ProcessAssignment)

Règle 3 : Transition2ProcessExpression (Priorité 9). Cette règle (Figure 18) transforme une transition UML en une expression représentant un processus dans le langage CSP.

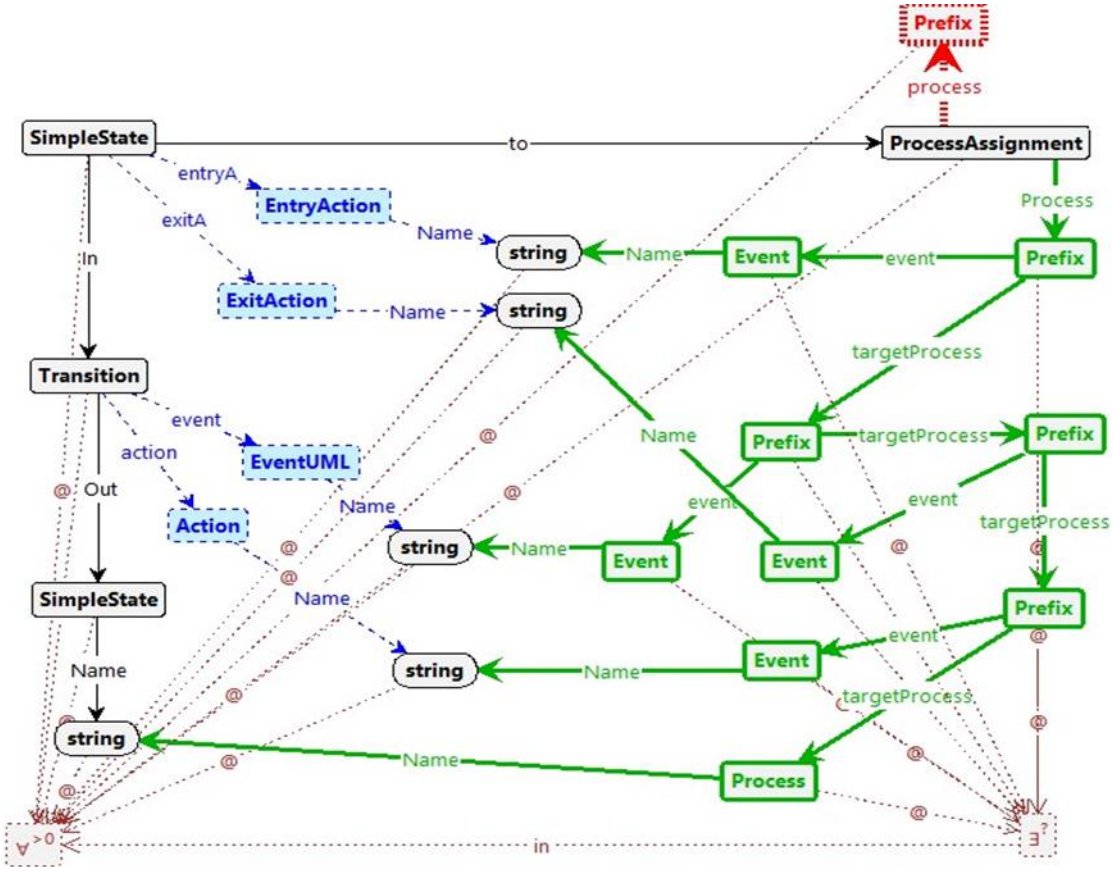


Figure 18 : Règle 3 **(**Transition2ProcessExpression)

Règle 4 : FinalState1 (Priorité 8). Cette règle (Figure 19) signifie qu'un état final (*FinalState*) dans un diagramme UML est converti en une affectation de processus (*ProcessAssignment*) dans CSP, où le processus associé est *SKIP*. Cela indique que l'exécution se termine proprement à cet endroit, sans action supplémentaire. Le nom de l'état final est utilisé pour nommer ce processus.

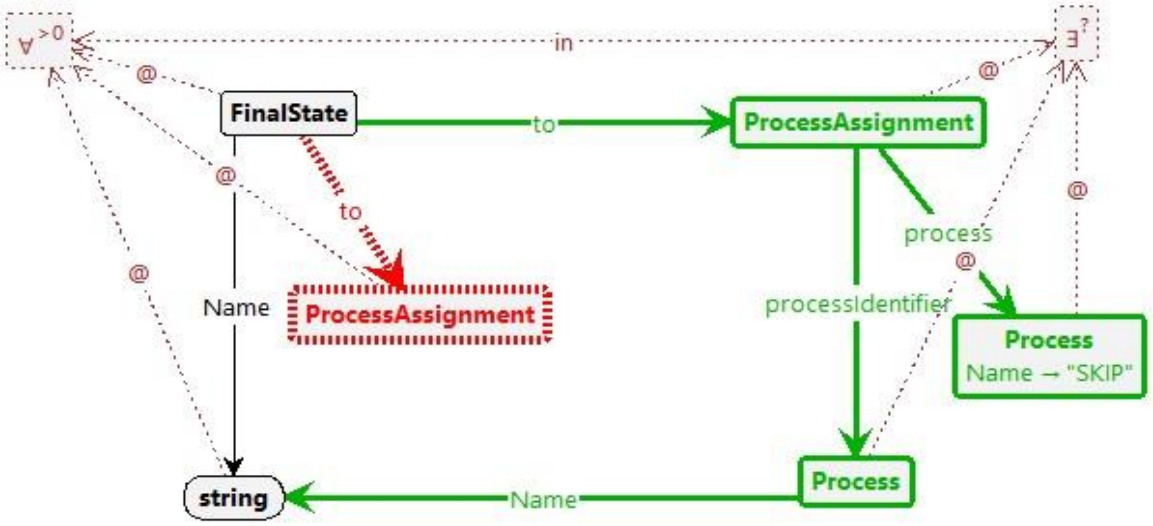


Figure 18:Règle 4 (FinalState1)

Règle 5 : FinalState2 (Priorité 7). Cette règle (Figure 20) signifie que lorsqu'un état final (*FinalState*) est atteint dans un diagramme d'états UML, il est traduit en un processus (*Process*) dans le langage CSP. Le nom de l'état final est extrait et utilisé pour générer l'affectation de processus correspondante, indiquant ainsi la fin de la séquence comportementale.

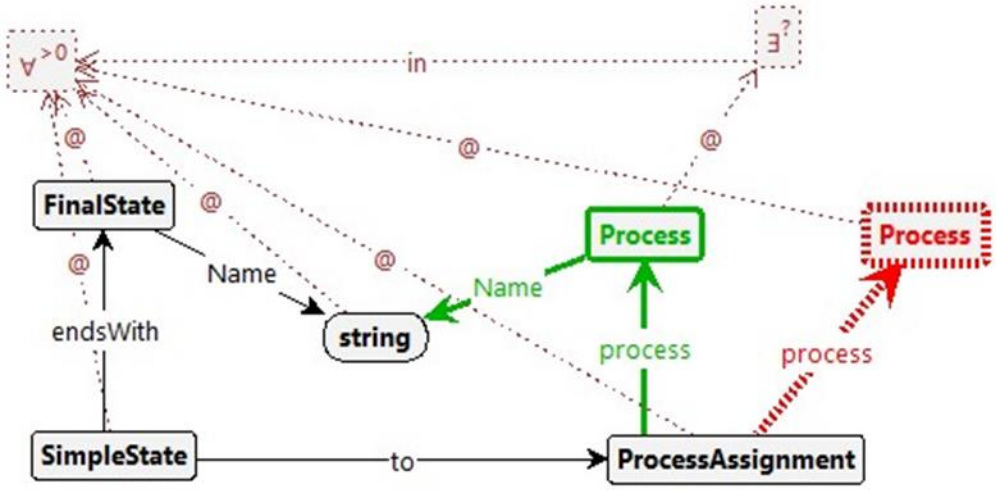


Figure 19:Règle 5 (FinalState 2)

Règle 6: Guard2Condition (Priority 6). Cette règle (Figure 21) transforme une garde UML (par exemple : [x > 0]) en une condition dans CSP (if x > 0 then P).

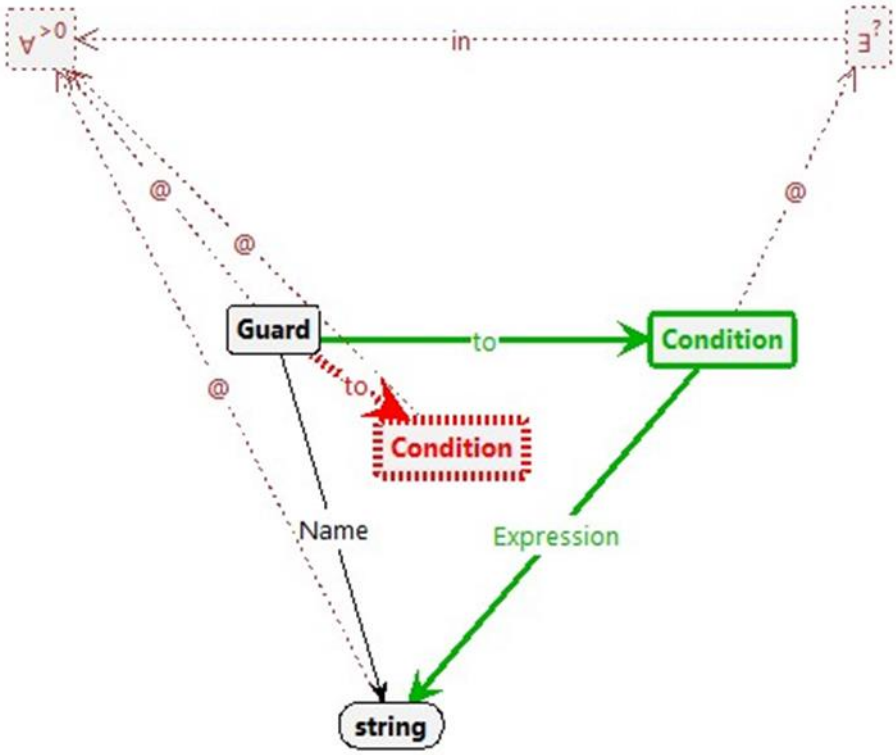


Figure 20:Règle 6 (Guard2Condition)

Règle 7: DelTransition (Priority 5). Cette règle (Figure 22) est appliquée pour supprimer toutes les Transition d'un diagramme d'etat transition.

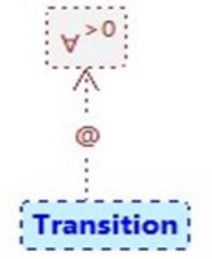


Figure 21:Règle 7 (DelTransition)

Règle 8 : DelSimpleState (Priority 4). Cette règle (Figure 23) est appliquée pour supprimer toutes les etat simple (SimpleS

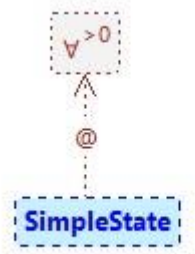


Figure 22:Règle 8 SimpleState

Règle 9: DelInitalState (Priority 3). Cette règle (Figure 24) est appliquée pour supprimer toutes les états initial (InitialState) d'un diagramme d'état transitions.

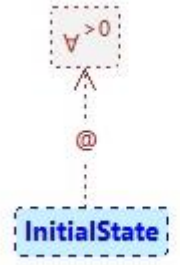


Figure 23:Règle 9 (InitialState)

Règle 10 : DelFinalState (Priority 2). Cette règle (Figure 25) est appliquée pour supprimer toutes les états final (FinalState)d'un diagramme d'etat transitions.

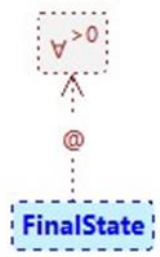


Figure 24:Règle 10(FinalState)

Règle 11 : DelCompositState (Priority 1). Cette règle (Figure 26) est appliquée pour supprimer toutes les états composite (CompositeState) d'un diagramme d'etat transition.

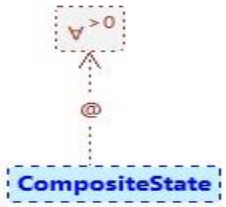


Figure 25:Règle11 (DelCompositeState)

3.6 ÉTUDE DE CAS

Dans cette section, nous présentons, à titre d'illustration de notre approche, la description d'une étude de cas : un système de contrôle d'une barrière permettant aux véhicules de quitter le parking.

La barrière est dotée de trois capteurs :

- ✓ gatePosition : avec trois valeurs « top », « middle », « bottom », représentant la position de la barrière.
- ✓ carAtGate : « Vrai » s'il y a un véhicule devant la barrière pour quitter le parking.
- ✓ CarJustExited : si un véhicule arrive juste de passer par la barrière elle a la valeur « Vrai », puis elle passe à la valeur « Faux » après un seul pas.

La barrière applique les actions : « raise » , « lower », « nop », son fonctionnement est comme suit :

- ✓ Si un véhicule va quitter le parking, la barrière va se lever jusqu'à la position « top ».
- ✓ Une fois la barrière est à la position « top », elle reste dans cette position jusqu'à ce que le véhicule passe par la barrière.
- ✓ Après que le véhicule passe par la barrière, cette dernière doit se baisser jusqu'à atteindre la position « bottom ».

Les diagrammes d'états-transitions présentés dans la Figure 27 modélise le fonctionnement de la barrière.

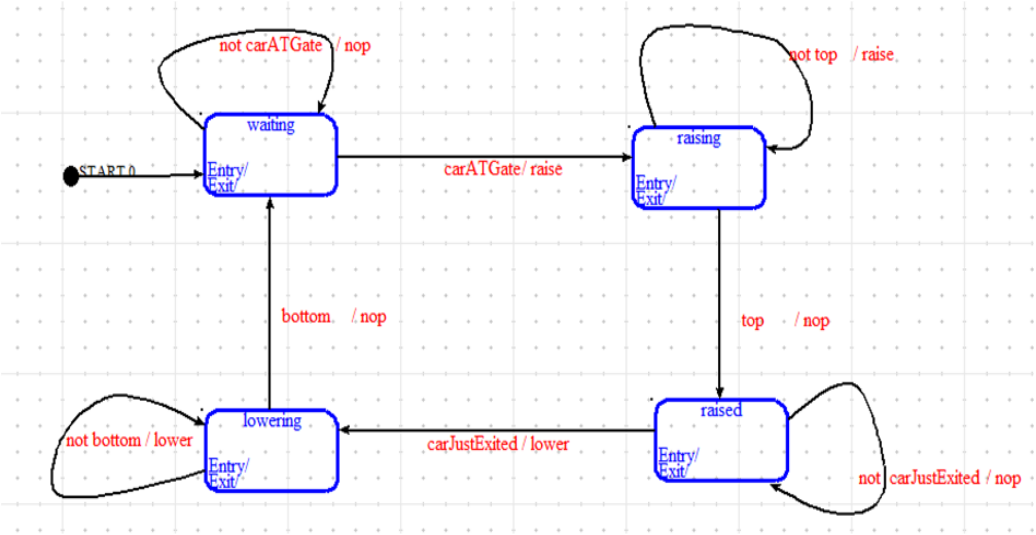


Figure 26:modélise le fonctionnement de la barrière

3.6.1 Modélisation du système dans l'outil GROOVE

La Figure 28 présente le diagramme d'états-transitions qui modélise le système décrit ci-dessus en utilisant GROOVE

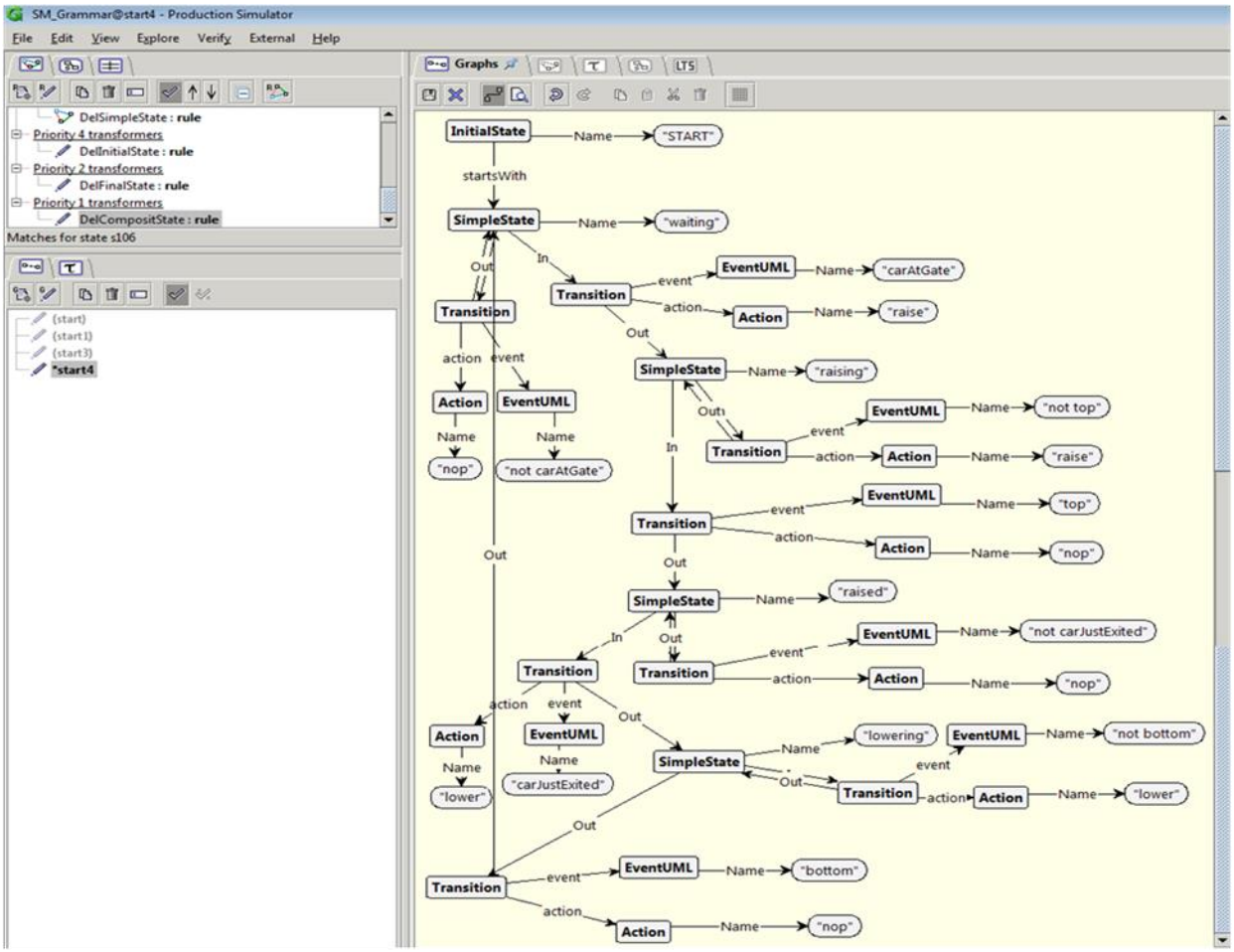


Figure 27:Diagramme d'états-transition qui modélise le système décrit

3.6.2 Transformation automatique des diagrammes UML 2.0 STM en CSP

La Figure 29 présente le système de transition étiqueté (LTS) généré après l'exécution de la grammaire de graphes. Le LTS contient 115 états et 250 transitions.

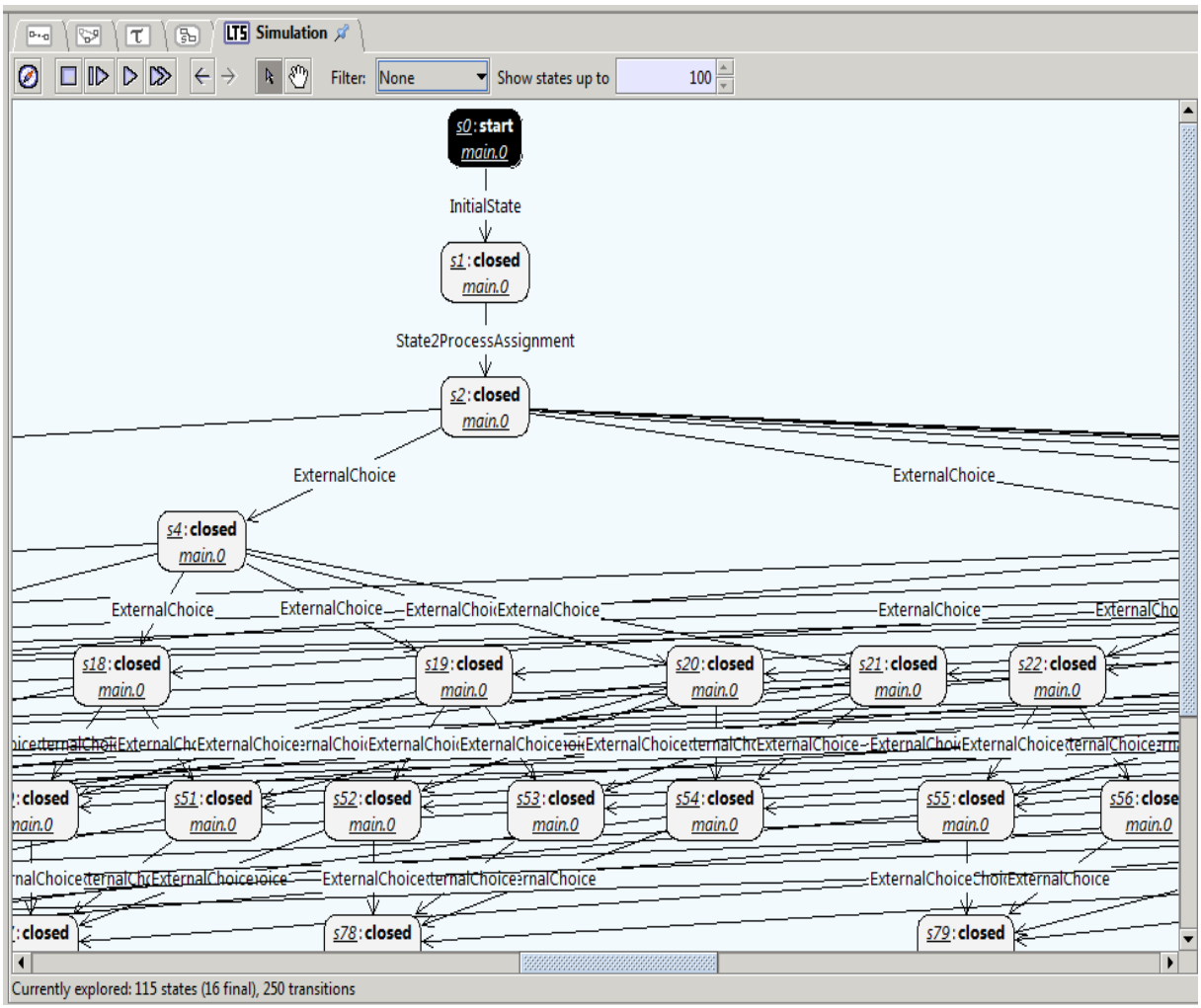


Figure 28:Présente le système de transition étiqueté(LTS) généré après l'exécution de la grammaire de graphe

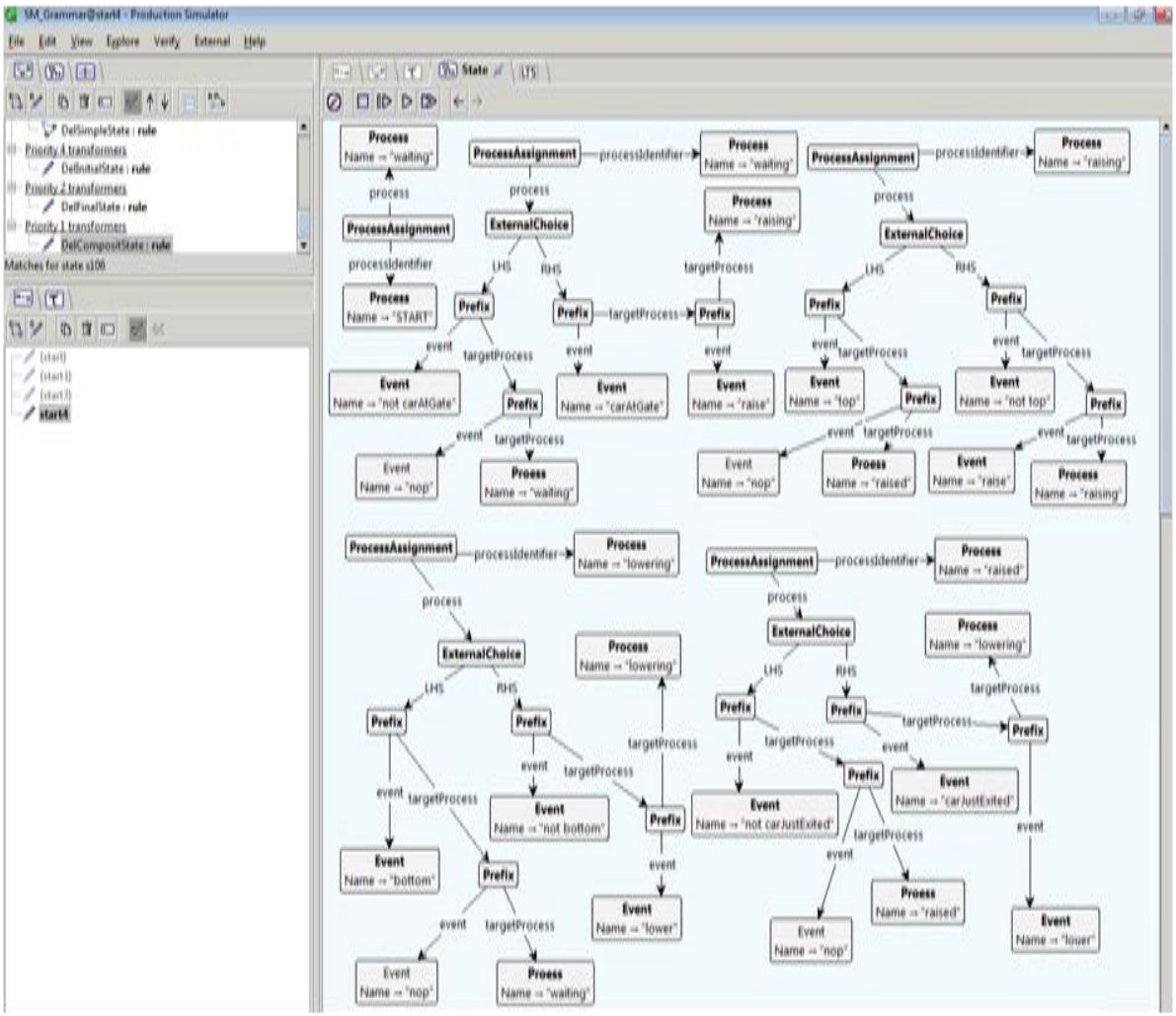


Figure 29 : Le système de transition étiqueté généré

3.7 Conclusion

Ce travail a permis de mettre en œuvre une transformation rigoureuse des diagrammes STM vers des spécifications CSP, en combinant les avantages des méthodes semi-formelles et formelles.

Grace à la méta-modélisation et à l'utilisation d'un modèle de correspondance, nous avons assuré la traçabilité entre les modèles source et cible. La transformation exogène proposée facilite l'analyse des systèmes à l'aide d'outils comme GROOVE.

Deux types de propriétés de correction ont été étudiés : la correspondance structurelle et la préservation des propriétés sémantiques. Cette approche ouvre la voie à une meilleure intégration entre modélisation graphique et vérification formelle, rendant le développement de système critiques plus fiable.

CONCLUSION GÉNÉRALE

À travers ce travail, nous avons proposé une approche intégrée visant à automatiser la transformation et la vérification des diagrammes d'états-transitions UML 2.0**.** Cette approche combine les avantages de la modélisation semi-formelle offerte par UML avec la rigueur du langage formel CSP**,** en s'appuyant sur les capacités avancées de transformation et de vérification de l'outil GROOVE.

L'approche a été concrétisée par la définition de trois méta-modèles **:** celui des diagrammes UML 2.0 STM, celui de CSP, ainsi qu'un méta-modèle de correspondances permettant d'établir des liens entre les modèles source et cible. Une transformation automatique des diagrammes STM vers des spécifications CSP a également été mise en œuvre.

Malgré les efforts déployés, la phase de vérification n'a malheureusement pas pu être finalisée. Nous envisageons donc de poursuivre ce travail dans le cadre de recherches futures**,** ou de le confier à d'autres étudiants souhaitant approfondir cette thématique dans le cadre de leurs travaux académiques.

BIBLIOGRAPHIE

Agrawal, A., Simon, G., & Karsai, G. (2004). Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. *Electronic Notes in Theoretical Computer Science, 109*, 43–56.

Almeida, J. B., Frade, M. J., Pinto, J. S., & De Sousa, S. M. (2011). *Rigorous Software Development: An Introduction to Program Verification* (Vol. 1). Springer.

Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H. J., & Kuske, S. (1999). Graph Transformation for Software Engineering: Science of Computer Programming. *Science of Computer Programming, 34*, 239.

Aouag, M. (2014). *Des diagrammes UML 2.0 vers les diagrammes orientés aspect à l'aide de transformation de graphes.*Ph.D. dissertation, Université de Constantine.

Bagherzadeh, M., Kahani, N., Jahed, K., & Dingel, J. (2020). Execution of partial state machine models. *IEEE Transactions on Software Engineering, 48*, 951–972.

Belaunde, M., Casanave, C., DSouza, D., Duddy, K., El Kaim, W., & Kennedy, A. (2003). MDA Guide Version 1.0.1. *MDA Guide Version 1.0.1*.

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

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

Bjorner, D. (2014). *Software Engineering: Formal Methods.*Springer.

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

Blanc, X., Le Bris, C., & Legoll, F. (2005). Analysis of a prototypical multiscale method coupling atomistic and continuum mechanics. *ESAIM: Mathematical Modelling and Numerical Analysis, 39*, 797–826.

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

Favre, J.-M., Estublier, J., & Le-Floch, M. B. (2006). *L'ingénierie dirigée par les modèles: au-delà du MDA.*Hermès Science.

Hamrouche, N., Djilani, C., Magri, P., Belhocine, Y., Djazi, F., & Kezzar, M. (2024). A novel biosorbent from raw pomegranate peel modified with SnCl2/FeCl2 for the adsorption of crystal violet cationic dye. *Biomass Conversion and Biorefinery,* 1–17.

Harel, D., & Politi, M. (1998). *Modeling reactive systems with statecharts: the STATEMATE approach.* McGraw-Hill.

Hoyle, D. (2017). *ISO 9000 Quality Systems Handbook: Updated for the ISO 9001:2015 Standard.*Routledge.

Kesraoui, S. M. (2017). *Intégration des techniques de vérification formelle dans une approche de conception des systèmes de contrôle-commande.* Ph.D. dissertation, Université de Bretagne Sud.

Latella, D., Majzik, I., & Massink, M. (1999). Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing, 11*, 637–664.

Lilius, J., & Paltor, I. P. (1999). vUML: A tool for verifying UML models. *14th IEEE International Conference on Automated Software Engineering*, (pp. 255–258).

Lowe, G., & Roscoe, B. (1997). Using CSP to detect errors in the TMN protocol. *IEEE Transactions on Software Engineering, 23*, 659–669.

Meghzili, S., Chaoui, A., Strecker, M., & Kerkouche, E. (2019). Verification of model transformations using Isabelle/HOL and Scala. *Information Systems Frontiers*, *21*, 45–65.

Rozenberg, G. (1997). *Handbook of Graph Grammars and Computing by Graph Transformation* (Vol. 1). World Scientific.

Sheng, G.-P., Yu, H.-Q., & Yue, Z.-B. (2005). Production of extracellular polymeric substances from Rhodopseudomonas acidophila in the presence of toxic substances. *Applied Microbiology and Biotechnology, 69*, 216–222.

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

Yoganathan, K., Rossant, C., Ng, S., Huang, Y., Butler, M. S., & Buss, A. D. (2003). 10-Methoxydihydrofuscin, Fuscinarin, and Fuscin, Novel Antagonists of the Human CCR5 Receptor from Oidiodendron griseum. *Journal of Natural Products, 66*, 1116–1117.