Conception assistée de contrôleurs d'automates depuis des modèles UML

Pascal André et Yannis Le Bars

LS2N CNRS UMR 6004 - Université of Nantes, France {firstname.lastname}@ls2n.fr

Résumé

Le logiciel prend une place de plus en plus prépondérante dans les systèmes cyber-physiques, notamment grâce aux performances accrues des réseaux. Dans l'industrie du futur, le logiciel doit non seulement être de qualité en terme en fiabilité et de performance mais il doit aussi pouvoir évoluer rapidement et s'adapter aux nouveaux besoins ou à de nouvelles contraintes. L'ingénierie des modèles vise à raccourcir le cycle de développement en plaçant l'accent sur les abstractions et en automatisant partiellement la génération du code. Dans cet article, nous explorons l'assistance au passage progressif du modèle au code pour réduire le temps entre l'analyse et la production du logiciel. Le modèle couvre des aspects structurels, dynamiques et fonctionnels du système étudié. Le code visé est celui d'un système distribué sur plusieurs dispositifs. Nous préconisons une approche par transformation de modèles dans laquelle les transformations restent simples, la complexité se trouve dans le processus de transformation qui se veut adaptable et configurable. Pour mener les expérimentations, les modèles sont écrits en UML (ou SysML) et les programmes déployés sur Android et Lego EV3.

Mots-clés : Ingénierie des modèles, UML Statecharts, Communication, Raffinement, Génération de code

1 Introduction

Le logiciel prend une part croissante dans l'industrie, que ce soit au niveau des produits, qui deviennent connectés, ou de la production automatisée de ces produits mais aussi dans le domaine des services avec l'assistance de robots et de l'intelligence artificielle. Le lancement de programmes comme "Industrie du futur" ont permis de faire converger les efforts en cybernétique. Il en résulte que les méthodes et outils de développement du logiciel impactent l'ensemble du développement de ces systèmes cyber-physiques. Non seulement le logiciel doit être de qualité en terme en fiabilité et de performance mais il doit aussi être capable d'évoluer et de s'adapter aux nouveaux besoins et aux nouvelles contraintes [1]. Les coûts de maintenance du logiciel, qui représentaient traditionnellement 70% du coût total du logiciel, ne cessent d'augmenter [17].

L'ingénierie des modèles vise à raccourcir le cycle de développement en plaçant l'accent sur les abstractions, la vérification de propriétés et l'automatisation partielle du codage. Raisonner pour démontrer les propriétés du système s'applique bien au niveau des modèles (il existe des méthodes et outils pour cela) mais plus difficilement au niveau du code, du fait de la complexité inhérente aux détails d'implantation et de la distribution (sécurité, communication...). La génération de code existe depuis de nombreuses années pour des modèles opérationnels que ce soit la compilation de langage de programmation source ou les générateurs à base de grammaires pour des langages dédiés [2] tels que l'ancêtre lex-yacc pour C, antrl, les parsers XML ou JSON, ou plus récemment XTEXT... Par contre, la génération de code depuis des modèles de plus haut niveau d'abstraction, issus de l'analyse ou la conception logicielle, reste encore une prérogative des développeurs pour un travail d'ingénierie logicielle. L'automatisation devient plus rentable que le développement (manuel) lorsqu'on prend en compte la maintenance évolutive des besoins fonctionnels, non fonctionnels ou techniques (changement de matériel par exemple).

1. On retrouve la dualité qualité des produits et qualité des processus.
2. ou *Domain Specific Language DSL*

Dans cet article, nous explorons l'assistance au passage progressif de modèles hétérogènes abstraits vers du code source exécutable pour réduire le temps entre l'analyse et la production du logiciel. Par modèle hétérogène nous entendons un modèle qui couvre des aspects structurels, dynamiques et fonctionnels du système modélisé. Nous supposons aussi des langages généralistes et expressifs pour décrire ces modèles tels que UML ou SysML [28]. Typiquement, on peut illustrer le cas d'un modèle UML avec un diagramme de classes (voire de composants) pour la partie statique, des statecharts pour la dynamique et des diagrammes d'activités complétés par un langage d'actions pour les calculs. Le code est celui d'un système distribué sur plusieurs dispositifs.

Le passage du modèle au code reste encore un défi du point de vue de l'automatisation (ou de l'assistance). D'une part, les générateurs de code des éditeurs UML produisent typiquement des squelettes où le gros du développement reste à faire. D'autre par les outils dédiés à une plate-forme technique sont plus riches mais réduisent inévitablement le spectre d'application. Le contexte choisi ici reste modeste, il consiste à mettre en place une approche basée sur les modèles (ingénierie des modèles) [7] pour construire du logiciel (sûr) de contrôle d'automates. Nous préconisons une approche par transformation de modèles dans laquelle les transformations restent simples. La complexité se trouve alors dans le processus de transformation qui se veut adaptable et configurable.

Le travail rapporté dans cet article est empirique au sens d'une démarche par différentes approches pour trouver une solution adaptée. Pour mener les expérimentations avec des étudiants, nous partons de modèles écrits avec des langages à sémantique hétérogène comme UML ou SysML et nous ciblons des automates programmables, EV3 de Lego en l'occurrence, commandés à distance par des programmes écrits en java déployés sur Android. A défaut de génération automatique complète, nous visons une assistance personnalisable, qui aide le concepteur logiciel à rationaliser son développement.

L'article est organisé comme suit. Nous commençons par introduire les éléments de contexte dans la section 2, puis une des études de cas support d'expérimentation (section 2). Nous illustrons les différentes approches et leur limites actuelles sur l'étude de cas dans la section 3. Une proposition d'approche structurée basée sur un processus de transformation de modèles est présentée en section 4 et détaillée en section 5 pour aboutir à une discussion plus générale de la faisabilité en section 6 avant de conclure.

2 Contexte

L'objectif est de mettre en place une chaîne de production logicielle à partir de modèles pour des systèmes distribués. En particulier on s'intéresse ici à des automates programmables ayant un environnement "réel" d'exécution qui prennent en compte des contraintes de fonctionnement, de sûreté de fonctionnement et de performance [26], *e.g.* Certaines propriétés sont générales (absence de blocage, réinitialisabilité...), d'autres sont liées à l'environnement ou au système lui-même (énergie, dangerosité, qualité de service....). Du point de vue logiciel, on considèrera au moins deux niveaux :

— le niveau modélisation et simulation où sont représentés les fonctionnements individuels et collectifs, décrites et analysées les contraintes sous forme d'une image numérique (digital twin). On pourra utiliser un ou plusieurs langages de modélisation (UML [4], SysML [12], Kmelia [3]...), des outils de vérification associés, des outils de simulation, etc. On qualifiera néanmoins les modèles de ce niveau de modèles logiques (ou d'analyse) en ce sens que l'environnement technique de mise en œuvre n'est pas encore déterminé. Le modèle d'analyse plongé dans un environnement technique sera qualifié de modèle de conception, comme l'illustre bien le modèle de développement en Y de la FIGURE 1 inspirée par [27].

— le niveau opérationnel où sont mis en œuvre les contrôles sur les dispositifs physiques. On utilise pour cela des outils de communication basés sur les API des automates (programmable logic controller -PLC), robots, capteurs et actionneurs.

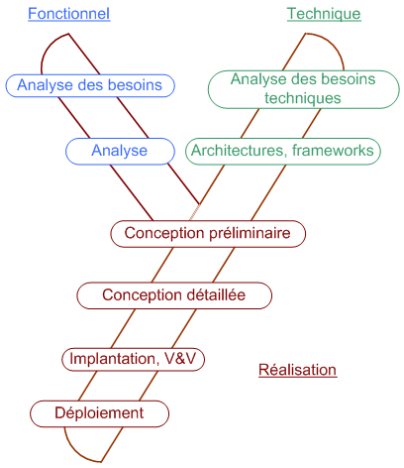

FIGURE 1 – Processus unifié 2TUP

Des niveaux intermédiaires peuvent être mis en œuvre pour faciliter la mise en place de la chaîne de production de code en s'inspirant de l'approche dirigé par les modèles (MDD) [7] et des lignes de production logicielles (Software Product Lines) [5]. Dans le développement MDD, il est essentiel de s'assurer de la correction des modèles avant de commencer le processus de transformations et de génération de code. On diminue ainsi le coût élevé de la détection tardive d'erreurs [13]. Qu'ils soient décrits en UML2, SysML ou dans d'autres langages de description d'architectures, les modèles considérés sont suffisamment détaillés pour être rendus exécutables [3]. Cela permet, en complément de plusieurs techniques de vérification basées sur de la preuve de théorème ou du *model checking*, d'envisager de tester ces modèles.

Etude de cas Le cas d'étude porte sur un équipement domotique simplifié, une porte de garage qui comprend des dispositifs matériels (télécommande, porte, automate, capteur, actionneurs...) et logiciels qui pilotent les différents dispositifs physiques [4]. Dans une version abstraite du diagramme de classes de la FIGURE 2, nous avons uniquement mentionné les opérations.

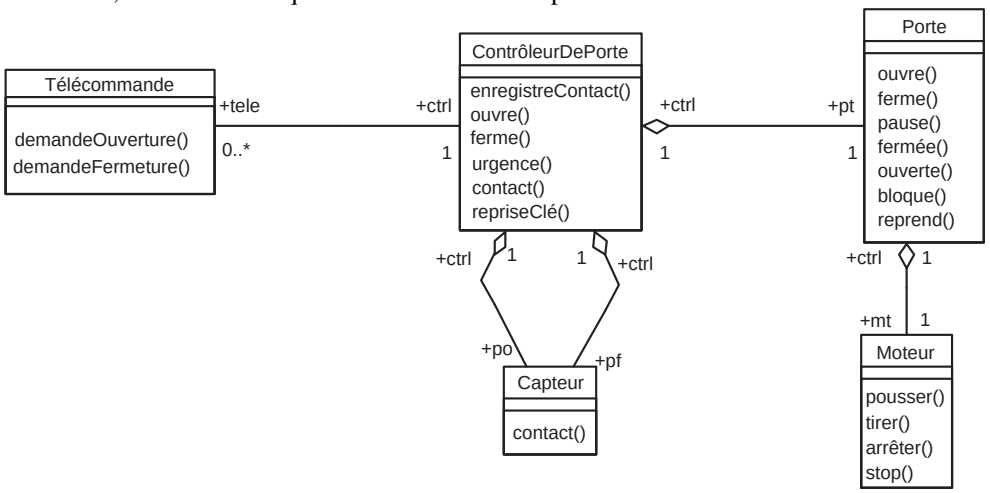

FIGURE 2 – Diagramme de classes - porte de garage

Le principe de fonctionnement du système est le suivant. Supposons la porte fermée. L'usager enclenche l'ouverture de la porte en appuyant sur le bouton `Ouverture` de sa télécommande. Il peut arrêter l'ouverture en réappuyant sur le bouton `Ouverture`, le moteur s'arrête. Dans le cas contraire la porte s'ouvre complètement et déclenche un capteur `po` (porte ouverte) qui entraîne l'arrêt du moteur. Appuyer sur le bouton `Fermeture` entraîne la fermeture de la porte si elle est ouverte (partiellement ou complètement). On peut arrêter la fermeture en réappuyant sur le bouton `Fermeture`, le moteur s'arrête. Dans le cas contraire la porte se ferme complètement et déclenche un capteur `pf` (porte fermée) qui entraîne l'arrêt du moteur. À tout moment, si quelqu'un appuie sur un bouton d'arrêt d'urgence situé

3. Les transformations de modèles deviennent pertinentes si les modèles contiennent suffisamment d'informations.
4. Voir l'énoncé de différents cas sur https://aelos.ls2n.fr/annexe-msr-2019/.

sur le mur alors la porte se bloque. Pour la réactiver (dans le même état) on tourne une clé privée dans une serrure située sur le mur. Le diagramme états-transitions de la FIGURE 3 décrit le comportement du contrôleur de la porte. Les actions sur les portes sont transférées aux moteurs par la porte elle-même.

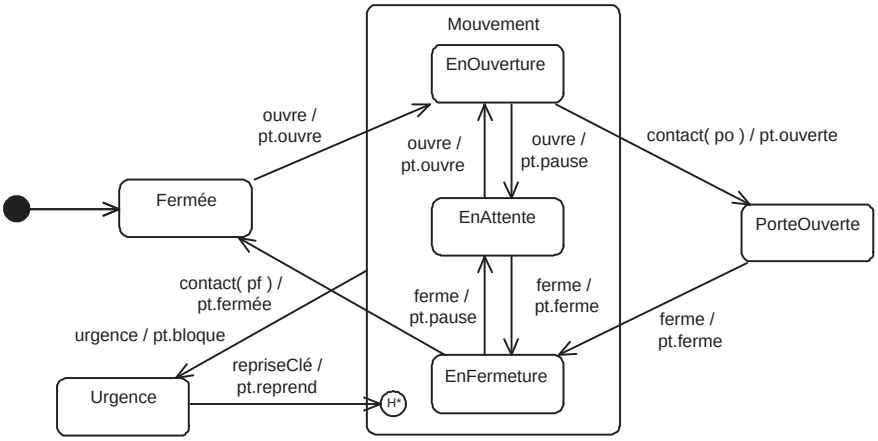

FIGURE 3 – Diagramme Etats-transitions du contrôleur - porte de garage

La télécommande, lorsqu'elle est activée, réagit à deux événements (appui sur le bouton d'ouverture ou appui sur le bouton de fermeture) et, à chaque fois, signale simplement au contrôleur qu'il y a eu pression sur le bouton. Son comportement est modélisé dans la FIGURE 4.

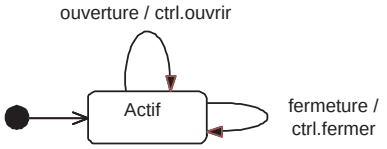

FIGURE 4 – Diagramme Etats-transitions de la télécommande - porte de garage

La vérification de tels modèles peut se faire via de l'analyse statique, du contrôle de type, du *model checking* pour les communications, du *theorem proving* pour les assertions de contrats fonctionnels et du test [2, 1]. Par la suite, nous supposons le modèle vérifié et validé, nous nous focalisons sur le passage du modèle au code, qui peut être un développement, un raffinement ou une génération de code.

3 Du modèle au code

La conception logicielle permet de passer du modèle au code (*cf.* FIGURE 1). C'est l'activité d'ingénierie en ce sens que des décisions doivent être prises qui influent sur la qualité du résultat. Les concepts clés sont l'abstraction, l'architecture, les patrons récurrents (patterns), la séparation des préoccupations, la modularité, l'indépendance fonctionnelle, etc. Le résultat est un modèle qui doit être évolutif (changements techniques ou fonctionnels) et couvrir les aspects complémentaires que sont le modèle des données et la persistance, les traitements et le parallélisme, les interfaces visuelles, le déploiement dans une vision architecturale qui fasse apparaître progressivement les détails [19].

On suppose pour la suite l'environnement technique choisi. La mise en œuvre se fait ici en Java avec un robot Lego EV3 et une application Android pour la télécommande. Trois alternatives principales se présentent pour le développement (conception, codage et test) de l'application. Nous les classons par degré d'automatisation : (i) développement manuel, (ii) processus de transformation de modèles, (iii) génération automatique de code. Nous détaillons par la suite les approches i) et iii) qui sont les cas extrêmes. L'approche intermédiaire ii) sera abordée dans la section 4.

3.1 Développement manuel

Dans les expérimentations menées, différents groupes d'étudiants disposent des modèles et spécifications associées mais aussi de documents tels que [16, 21, 23]. Le code qu'ils ont produit n'est pas nécessairement conforme aux modèles. En effet, le modèle sert à comprendre et interpréter le cas et les étudiants ne visent pas une conservation stricte de la sémantique. Une première version [5] du code du cas "porte de garage", synthétisée dans le diagramme de classe de la FIGURE 6, correspond à la maquette de la FIGURE 5. Plusieurs versions ont été produites sur des sprints différents, jusqu'à inclure une application mobile simplifiée pour la télécommande (communication bluetooth). Une autre version [6], produite par un groupe d'étudiants de Miage de Nantes a conduit à l'implantation de la FIGURE 7. On peut constater que les mises en œuvre sont différentes. Par exemple dans le cas de la FIGURE 6, les états de l'automate sont représentés par des types énumérés alors qu'un *pattern state* est utilisé dans le cas de la FIGURE 7. La manière de réaliser la télécommande et son association avec le contrôleur EV3 a changé parfois du tout au tout : IHM swing avec une communication filaire TCP-IP, application Android et connexion bluetooth, application Android et connexion wifi.

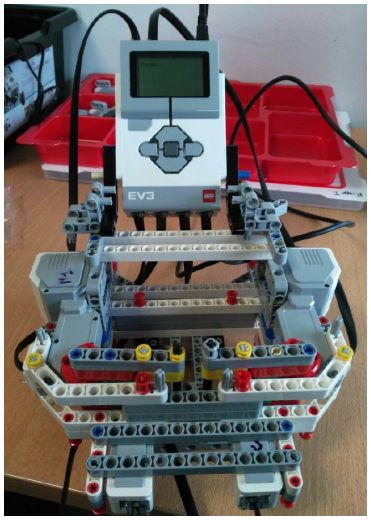

FIGURE 5 – Maquette Lego - porte

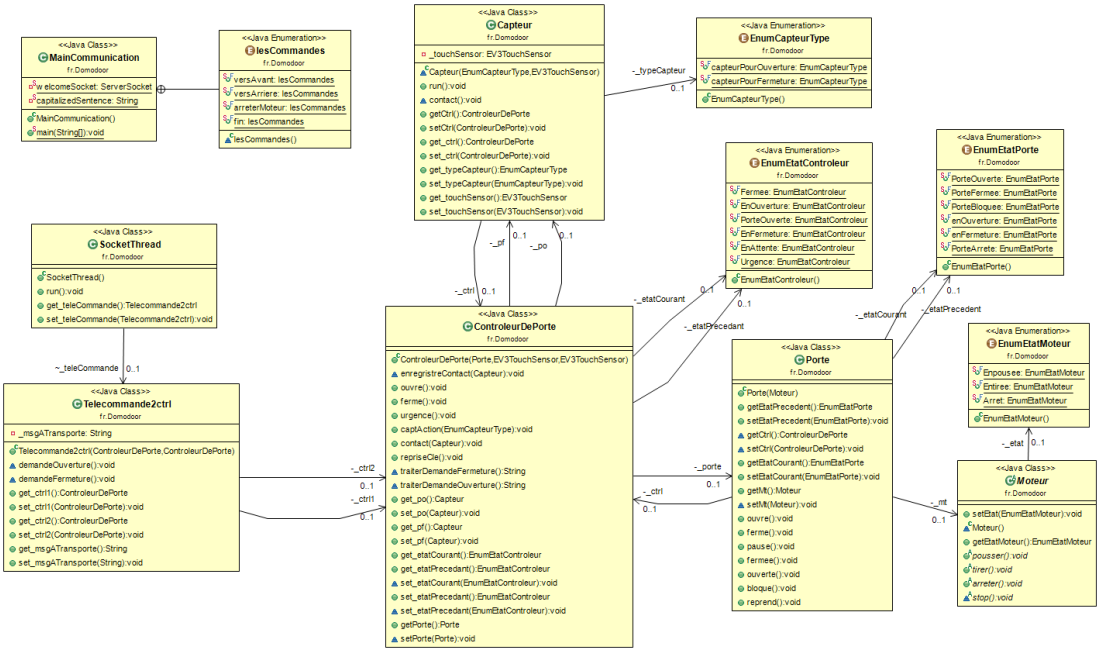

FIGURE 6 – Diagramme de classe de l'application (v1)

5. https://github.com/demeph/TER-2017-2018
6. https://github.com/FrapperColin/2017-2018/tree/master/IngenierieLogicielleDomoDoor

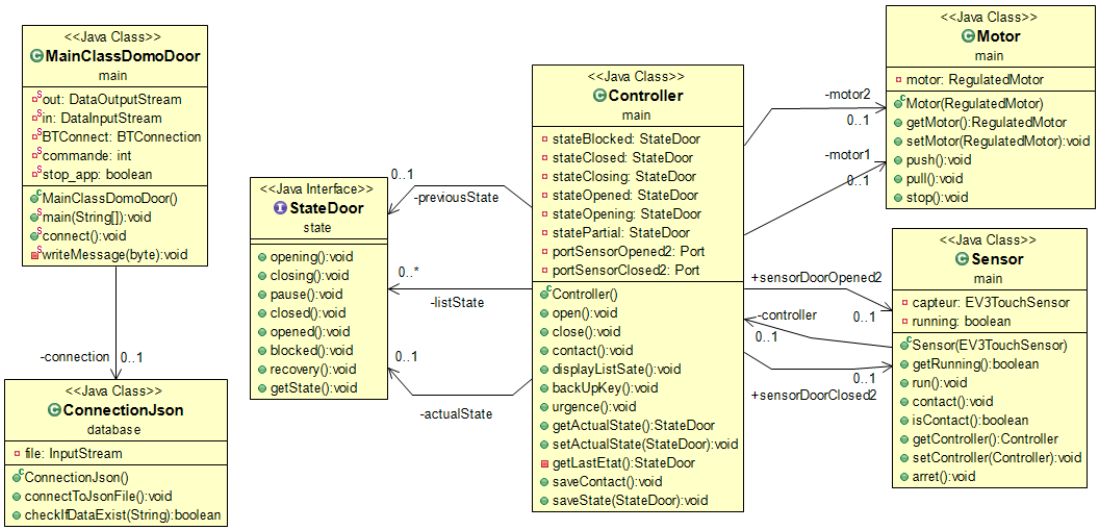

FIGURE 7 – Diagramme de classe de l'application (v2)

Isoler les différents choix de conception est un premier pas pour rationaliser le développement dans un processus de raffinement (cf section 4).

3.2 Génération automatique de code, animation

Sur le spectre de la conception, à l'opposé du développement manuel, nous trouvons la génération automatique et l'animation. Nous illustrons dans cette section nos expérimentations sur ce sujet.

Génération de code depuis les éditeurs UML Nous nous intéressons à des outils UML qui permettent d'éditer notamment des diagrammes états-transitions (DET) et de classes (DC), et (re)générer du code. La possibilité de définir des morceaux de code dans le code généré et de les remonter dans le modèle pour ne pas les perdre lors d'une nouvelle génération de code s'appelle "*round trip engineering*". Ce qui n'est pas la même chose que d'étendre un *framework*, *i.e.* rattacher des éléments spécifiques à une structure générique. Les descriptions sont échangées au format XMI, qui existe en autant de versions que le langage UML lui-même (ou l'un de ses profils). Nous indiquons la version utilisée. Notre sélection ne se veut ni exhaustive ni pleinement représentative que ce soit en terme d'outil que de fonctionnalités ; elle dessine simplement des tendances.

— StarUML est un représentant des éditeurs UML gratuits. Chaque DET est associé à une classe dont il représente le protocole. Star UML génère les parties structurelles des DC (profils des opérations, attributs, relations entre classes) et donc pas le corps des opérations ni les DET.

— Papyrus représente les outils de la tendance "*Modeling*" d'Eclipse. Il présente les avantages suivants : *open source*, extensible, bénéficie des outils EMF et de contributions récentes. La génération des modèles de Papyrus est plus avancée que Star UML et permet d'ajouter des comportements à des opérations/fonctions. Par contre les relations entre classes n'ont pas été générées. Dans cette catégorie d'outils, on peut aussi citer les outils d'Obeo, notamment UML Designer et Acceleo, ou le nouveau projet Polarys qui inclut Papyrus et Topcased.

— Yakindu est un outil dédié aux statecharts (DET) proposé par Itemis. L'inconvénient majeur est de ne pouvoir traiter qu'un DET à la fois (une seule classe), on ne traite donc pas des synchronisations

entre automates, sauf à les simuler sous forme de sous-régions parallèles (état hiérarchique composite). Il est possible de créer deux sous-diagrammes parallèles dans un diagramme, ça permet de simuler le comportement de deux diagrammes mais la génération créera qu'une seule classe pour les deux sous-diagrammes. La génération de code développe par contre toute la machinerie d'exécution de l'automate. On trouvera un exposé détaillé de l'implantation Java des DET dans le mémoire [16].

— Visual Paradigm est très riche en standards et fonctionnalités. Il supporte la génération des diagrammes de classes et états-transitions en code source Java mais aussi en CC++ ou VB.net. Sa fonctionnalité *round-trip engineering* synchronise le code et le modèle.

— Contrairement à Visual Paradigm, Modelio fait la différence entre les méthodes gérées par Modelio et les autres. Une méthode gérée est automatiquement générée à chaque modification. Une méthode simple non gérée par Modelio est à la charge du développeur et conservée pour le *round-trip*.

— IBM Rational Rhapsody est un outil de référence, produit par les concepteurs d'OMT la principale source d'inspiration d'UML. Ce doit être l'outil le plus complet. Le code est mis à jour automatiquement dans une vue en parallèle du modèle. Nous pouvons modifier directement le code, les diagrammes resteront synchronisés.

Noter que ces outils ne sont pas forcément mono-langages. Par exemple Modelio, Papyrys ou Visual Paradigm intègrent différents standards de l'OMG comme SysML, BPMN...

La TABLE 1 résume quelques caractéristiques des générateurs des outils. La ligne MOM (Message Oriented Middleware) indique la présente d'une implantation distribuée des objets qui échangent par envoi de messages et signaux. Nous n'avons pas retenu ici la possibilité de traiter le temps (réel) comme par exemple avec le profil MARTE. La licence d'utilisation peut être de type OpenSource, Free, Commerciale. Nous appelons *API Mapping* une fonctionnalité qui permet de rattacher des éléments de modèle à des éléments prédéfinis dans des bibliothèques, des *frameworks*...

TABLE 1 – Comparaison de quelques outils avec générateurs de code

	Star UML	Papyrus	Yakindu	Modelio	VisualParadim	IBM rational rhapsody
Version UML	2.0	2.5	-	2.4.1	2.0	2.4.1
DC	√	√	-	√	√	√
DET	-	-	√	√[1]	√	√
Operations	-	√	-	RndTrip	RndTrip	√
MOM	-	-	-	-	-	-
API Mapping	-	-	-	-	-	-
Round-trip	-	-	-	√	√	√
Licence[d]	F, C	O	F, C	O	C	C

[1]Extension in http://www.sinelabore.com/doku.php?id=wiki:landing_pages:modelio

La lecture de la TABLE 1 met en évidence qu'aucun outil, à notre connaissance, ne traite directement du problème de distribution et de communication ni du *mapping* explicite avec l'infrastructure technique (hormis le plongement dans un contexte donné Java, .NET, REST...). Cependant, dans Visual Paradigm par exemple, on peut intégrer des modèles de déploiement dans le cloud. IBM Engineering Systems Design Rhapsody est plutôt dédié à la conception détaillée. Certains outils proposent aussi des fonctionnalités de persistance (*e.g.* mapping objets relations ou SQL) que nous n'avons pas retenu ici puisque nous nous focalisons sur les automatismes. A noter que pour les expérimentations, nous avons utilisé Papyrus pour générer notre diagramme de classes en code Java.

UML exécutable Générer du code depuis UML pour une architecture technique donnée ou même un *framework* donné reste encore réservé à des cas simples comme la génération d'applications CRUD (Create, Read, Update, Delete) sur des bases de données relationnelles simples. Plus précisément il faut que le *framework* technique soit générique et complet mais aussi que les modèles soient simples [4]. On peut aussi animer ou exécuter des spécifications, qui sont alors qualifiées d'opérationnelles.

Dans tous les cas, le pré-requis est d'avoir des modèles complets sur la structure du système (diagrammes de classes et composants), son comportement dynamique (diagrammes états-transitions) et son comportement fonctionnel (diagrammes d'activités). Les diagrammes ne suffisent pas à préciser la sémantique. On doit en plus ajouter des contraintes écrites en OCL [8] (langage déclaratif pour les assertions de type invariant et pre/post-conditions) ou bien du pseudo-code écrit dans un langage conforme à la sémantique des actions en UML. Le concept d'action est présent de manière abstraite dans les diagrammes d'activités ou états-transitions.

La sémantique des actions (*Action Semantics*) est définie par un méta-langage des calculs depuis la version UML 1.4. Il s'agit d'un véritable langage de programmation mais aucune syntaxe concrète n'était proposée en standard (et donc pas de compilateur !). Dès UML 1.4, des syntaxes concrètes étaient proposées dans les outils proposant une version exécutable d'UML, notamment pour le temps-réel : (i) Le langage *Action Specification Language (ASL)* est implanté dans l'outil iUMLLite de Kennedy–Carter (Abstract Solutions) supportant xUML [25]. (ii) Le langage *BridgePoint Action Language (AL)* (et les dérivés aussi SMALL, OAL, TALL) proposé par Balcer & Mellor et implanté dans l'outil xtUML de Mentor Graphics [20]. (iii) Le langage *Kabira Action Semantics (Kabira AS)* proposé par Kabira Technologies (acquis par Tibco, TIBCO Business Studio). (iv) Le sous-ensemble d'actions de la norme de télécommunication SDL [6], qui existe aussi sous la forme d'un profil UML. On trouve aussi le langage *Platform Independent Action Language (PAL)* proposé par Pathfinder Solutions, ou SCRALL [9] qui propose une représentation graphique. Une syntaxe concrète est aussi proposée dans [22]. Les outils associés étant payants ou anciens, nous n'avons pas encore expérimenté cette approche sur nos études de cas. Ces efforts ont conduit à une sémantique pour un sous-ensemble de modèles UML exécutables, appelée `fUML` (*Semantics of a Foundational Subset for Executable UML*) [14], muni cette fois d'une syntaxe concrète normalisée `Alf`. Une implantation de référence existe[7]. Nous prévoyons d'intégrer des expérimentations à ce sujet.

4 Vers un processus de transformation de modèles

Dans la vision MDA [7], un processus de transformation est un enchaînement de transformations (raffinements) permettant de passer d'un *Platform Independent Model (PIM)* à un *Platform Specific Model (PSM)* plus concret. Les modèles logiques en entrée du processus font abstraction de l'environnement technique et des exigences non-fonctionnelles. Comme le montre la FIGURE 1, la conception consiste à "tisser" le modèle logique sur l'infrastructure technique (*platform*) pour aboutir à un modèle exécutable. Nous attirons l'attention du lecteur sur les constats suivants : (i) Il est illusoire de vouloir générer automatiquement du code sans avoir des modèles UML riches et détaillés. (ii) La génération de code elle-même n'est pas envisageable comme unique étape de transformation, du fait de la distance sémantique entre le modèle logique et la cible technique, composées d'aspects orthogonaux mais corrélés, appelés domaines (*e.g.* persistance, IHM, contrôle, communications, entrées/sorties) sur lesquels le modèle initial doit être "tissé". (iii) La conception est par nature une activité d'ingénierie, liée à l'expérience des concepteurs (*cf.* page 5). On ne peut industrialiser un processus que si tous les rouages sont connus avec précision. (iv) La pratique nous a montré que les transformations étaient efficaces lorsque les modèles étaient proches sémantiquement *e.g.* diagramme de classe et modèle relationnel pour la per-

7. http://modeldriven.github.io/fUML-Reference-Implementation/

sistance. (v) Les transformations sont écrites sous forme algorithmique (*e.g.* Kermeta [8] ou sous forme de règles de transformation (*e.g.* ATL [9]. Travailler avec des transformations simples réduit les problèmes de cohérences et de complétude.

Sur la base de ces considérations nous adoptons un principe que nous qualifions de *small step transformations*. La complexité (l'intelligence) n'est pas dans la transformation mais dans le processus de transformation. Une transformation complexe est composée hiérarchiquement d'autres transformations, jusqu'à des transformations élémentaires. La FIGURE 8 esquisse les aspects à considérer pour raffiner vers l'implantation. Ces macro-transformations utilisent des informations de configuration.

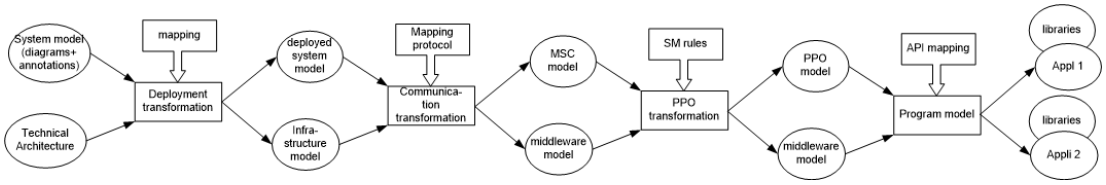

FIGURE 8 – Processus général de transformation

— On commence par structurer en applications en sous-systèmes avec un mapping sur l'architecture technique en décrivant les API et les protocoles de communication. Évidemment, si le modèle de départ inclut un diagramme de composants et un diagramme de déploiement (schématisant une conception préliminaire dans la FIGURE 1), la transformation de déploiement sera simplifiée.

— Ensuite, pour chaque type de communication on doit décliner les envois de messages UML selon le protocole considéré (désigné par MOM dans la TABLE 1. A minima, dans une implantation séquentielle, on peut avoir un envoi de message dans le langage à objets cible (Java, C++ ou C#).

— La troisième transformation concerne la représentation des automates (associés aux classes) dans un modèle du langage de programmation qui en général n'inclut pas nativement ce concept. On pourra utiliser soit des types énumérés soit le *pattern* `State` en fonction des situations. Ce problème épineux est abordé dans la section 5.

— La quatrième transformation consiste à faire correspondre des éléments de modèles à des éléments prédéfinis issus des bibliothèques des *frameworks* supports. Par exemple, la classe `Moteur` est mise en œuvre par la classe `lejos.robotics.RegulatedMotor`. Cet *API mapping* nécessite en général des adaptations portant sur les attributs, méthodes et envois de messages. Deux approches sont envisageables : (i) encapsuler la classe prédéfinie dans la classe du modèle et l'utiliser par délégation (l'avantage est de conserver l'API du modèle), (ii) substituer la classe du modèle par la classe prédéfinie et renommer les appels à l'API du modèle (on perd en traçabilité).

Il est à noter que l'ensemble des paramètres et décisions de la transformation est à conserver pour rejouer le processus de transformation en cas d'évolution du modèle initial.

Le processus de la FIGURE 8 est abstrait mais générique. Des simplifications existent.

— Itération A priori, nous ne sommes pas ici dans une approche dite *round trip* dans laquelle les injections de code sont rattachées à des points d'accès (*hook*) pour ne pas être perdues dans la (re)génération suivante. Mais on peut s'en inspirer pour optimiser l'historique.

— Animation, exécution Lorsque l'environnement technique est maîtrisé, la transformation va "plonger" le modèle dans le *framework* pour le rendre exécutable. On trouvera dans [18] des techniques de raffinement vers Java. Nous illustrerons cette situation dans la section 5.

8. http://diverse-project.github.io/k3/
9. https://www.eclipse.org/atl/

5 Expérimentations

Dans cette section, nous rapportons des éléments d'expérimentation avec des étudiants sur des processus de transformations visant Java avec des *middleware* plus ou moins complexes et hétérogènes.

Transformation de modèles pour la vérification et l'animation Nous avons traduit manuellement une partie des modèles UML vers le langage Kmelia [3]. Cette approche de type raffinement est plus intéressante que UML 'exécutable' car la spécification formelle permet de vérifier des propriétés du modèle, de le tester via des harnais de test et d'exécuter les spécifications via un générateur de code java. L'objet de l'article n'est pas directement la vérification et validation mais le travail a été mené et des éléments sont fournis à ce sujet dans un rapport de projet [10]. Nous suggérons au lecteur de consulter l'article [3] pour connaître l'étendue des vérifications possibles.

Transformation de modèles vers le programme Java avec ATL Pour mener ce type d'expérimentation, nous avons utilisé ATL, un langage de transformation de modèle à base de règles de transformation non déterministes. ATL lit un modèle source conforme au méta-modèle source et produit un modèle cible conforme au méta-modèle cible. Dans notre cas le modèle source sera le modèle UML de notre diagramme de classes et nos diagrammes états-transitions. Nous récupérons le modèle UML sous la forme d'un fichier `.uml` après l'avoir édité à l'aide de Papyrus (*cf.* FIGURE 9).

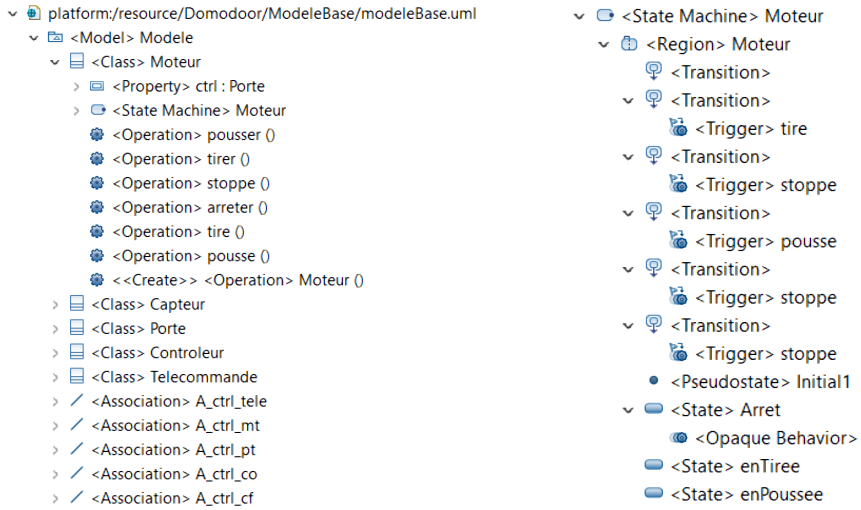

FIGURE 9 – Représentation UML des classes et d'un diagramme états-transitions

Dans ATL, deux modes de transformations peuvent être utilisés. Le mode `from` permet de créer un modèle en écrivant tous les paramètres, tous les attributs dans le modèle de sortie. Le mode `refine` permet de copier dans le modèle de sortie tout ce qui n'est pas inclus dans la règle puis applique la règle. Une règle peut modifier, créer ou supprimer des propriétés ou des attributs dans un modèle. Dans ce mode, les méta-modèles source et cible partagent le même méta-modèle. Le mode `refine` est plus intéressant pour nos transformations car travaillons sur des transformations partielles.

Considérons la troisième (macro-)transformation de la FIGURE 8, celle des DETs, en supposant ici des automates simples (pas de composites, pas de temps, pas d'historique). Des conventions d'écriture

10. https://costo.univ-nantes.fr/wp-content/uploads/sites/8/2019/06/RapportTERCoutandLeBerre.pdf

ont été déterminées (par exemple les éléments `Region` et `StateMachine` ont le nom de leur classe) qui facilitent par la suite l'écriture des règles de transformations. Nous conservons le nom par défaut pour chaque état initial soit '`Initial1`'. Quatre transformations sont décrites ci-après.

La première transformation crée un élément `Enumeration` pour toutes les classes qui ont un diagramme états-transitions. La règle parcourt le modèle et pour chaque classe qui a comme élément fils un diagramme états-transitions, nous créons un élément `Enumeration` avec comme paramètre de visibilité : `private`. Nous concaténons '`State`' et le nom de la classe pour le nom de l'énumération. Le résultat est donné en partie droite de la FIGURE 10.

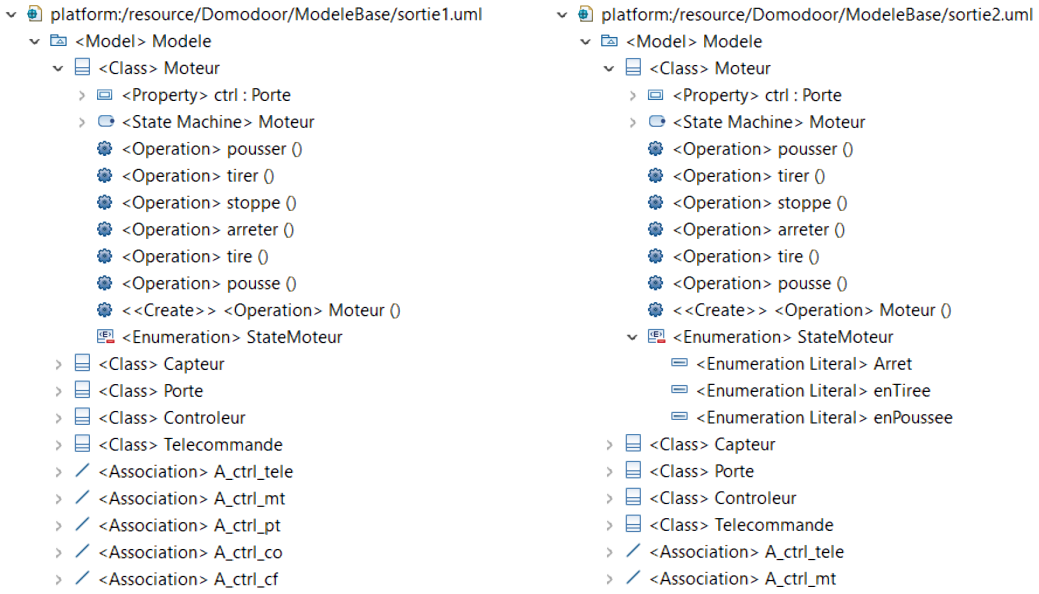

FIGURE 10 – Résultats des deux premières transformations

La seconde transformation remplit l'énumération précédente avec des éléments `EnumerationLiteral` pour chaque état dans le DET de la classe. Nous récupérons tous les noms des éléments `State` dans le diagramme états-transitions. Les super-états sont ôtés, qui ont des sous-états, pour ne conserver que les états simples. Les super-transitions (issues des super-états) sont alors dupliquées. Nous créons un élément `Enumeration Literal` dont les valeurs sont les noms des états retenus. Le résultat est donné en partie gauche de la FIGURE 10.

La troisième transformation crée une variable `_etatCourant` et une variable `_etatPrecedant` (*cf.* FIGURE 11). La première variable est créée pour toute classe qui à un diagramme états-transitions, la deuxième variable est créée que si le diagramme a un élément `Pseudostate` de type `deepHistory`. Les deux variables sont des éléments `Property` et sont du type de l'énumération. Pour initialiser l'état courant, nous devons lui rajouter un élément fils `OpaqueExpression`, il a deux

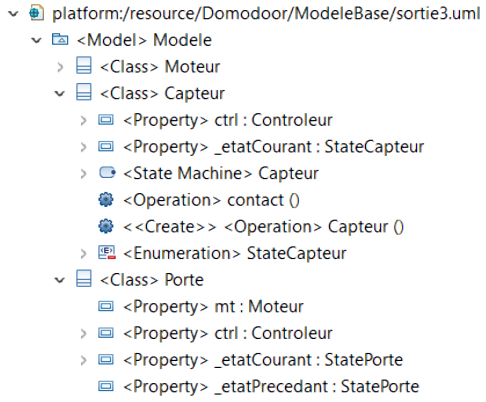

FIGURE 11 – Résultat de la 3e transformation

paramètres : '`language`' qui prend pour valeur '`JAVA`' et '`body`'. Le paramètre `body` est initialisé avec la concaténation du nom de l'énumération et l'état initial. L'état initial est trouvé en récupérant l'état cible de la transition qui a l'état initial comme état source de la transition.

La quatrième transformation détermine le comportement des fonctions. Pour chaque fonction utilisée comme déclencheur dans un diagramme états-transitions, nous allons créer une condition `switch`. Pour remplir la condition, nous récupérons l'état source et l'état cible de toutes les transitions qui ont comme déclencheur la fonction. Les états sources correspondent aux cas possibles pour le changement d'état et les états cibles correspondent à la nouvelle valeur de l'état courant. Nous ajoutons dans chaque cas du `switch` l'action de sortie de l'état de départ et l'action d'entrée de l'état d'arrivée s'il y en a.

Ces expérimentations mettent en évidence la complexité du problème et quelques aspects fondamentaux à traiter. Les résultats restent encore assez loin des objectifs finaux.

6 Discussion

La génération automatique de code, fournit un modèle (très) incomplet, qui souvent n'exploite même pas les informations du modèle (contraintes OCL, détail des opérations). Bien que de nombreux travaux aient été menés, l'étude systématique de Ciccozzi et al. [11] montre que l'exécution de modèles UML reste un problème difficile et s'adapte plus à la simulation qu'au développement logiciel. On notera toutefois l'apport essentiel des nouveaux standards `fUML` et `Alf` qui pallient un manque dans la sémantique des actions. Ils ont été mis en œuvre, par exemple, dans la vérification de modèles [24], l'exécution via C++ [10] ou MoKa/Papyrus [15, 24]. En cybernétique, SysML [28] est plus préconisé qu'UML pour la conception d'automates. Ainsi dans un exemple de contrôle de transmission pour lego NXT [11], le modèle SysML est très détaillé et peut ainsi être simulé par l'outil Cameo. La modélisation est plus adaptée mais cela ne change pas fondamentalement notre problématique.

La conception manuelle de l'application à partir d'un modèle initial n'a pas posé de grandes difficultés aux étudiants, hormis l'apprentissage de l'environnement technique cible. Par contre, nous avons constaté que le code ne respectait ni les exigences ni le modèle initial, qui bien que détaillé, ne garantissait ni la cohérence ni la complétude de la spécification du système. De plus, des contraintes techniques s'imposent, comme par exemple le fait que le modèle Lego utilise deux moteurs (un par roue) et non un moteur unique comme dans le modèle. De même la communication sans fil entre la télécommande et le contrôleur reste abstraite sous forme d'envoi de messages dans le modèle. La conception manuelle fait apparaître divers aspects orthogonaux à traiter qui ne sont pas a priori hiérarchisés par les étudiants. Les dépendances restent implicites pour eux, même s'ils se rendent compte que des choix pour un aspect ont des répercussions ou des contraintes sur d'autres aspects.

En ce qui concerne la modélisation du processus de développement sous forme d'un "*workflow*" d'activités, nous avons classé dans la FIGURE 8 les transformations par ordre d'impact : choix architecturaux (déploiement, communications), choix de conception générale (langage de programmation), choix de conception détaillée (*patterns*, *mapping* des bibliothèques). Le modèle reste abstrait en ce sens que les paramètres fournis restent conséquents et ces transformations sont elle-mêmes des processus de transformation. Il nous paraît néanmoins indispensable en ce sens qu'il pourra s'adapter aux besoin de chaque projet via le paramétrage ou en remplaçant des transformations par d'autres transformations. A titre d'exemple, le codage des machines à états est très complexe, soumis à interprétation et fortement corrélé au modèle global d'exécution [23]. Divers stratégies (énumérations, *pattern* `State`, moteur d'exécution) sont applicables pour un même cas d'étude en fonction de la nature des automates. Par exemple, les énumérations sont inévitables pour un automate de moins de 3 états. Le *pattern* `State` est utile si les opérations associées ont un comportement différent d'un état à l'autre et que le nombre

11. https://blog.nomagic.com/building-executable-sysml-model-automatic-transmission-system-part-1/

d'états reste limité. Au delà de 10 états, une instrumentation via un *framework* dédié est nécessaire, on doit alors brancher sur l'API du *framework* (instantiation, héritage, appel, mapping...). Nous pensons aussi que plusieurs outils de transformation devraient être utilisés car l'approche à base de règle est inadaptée pour certaines transformations plus opérationnelle comme celle mentionnée dans [23]. L'ancrage dans le support d'exécution peut être paramétré par un *mapping* de types, classes et opérations. De même la vérification et le test de modèles peuvent s'insérer à tout moment dans le processus [1].

L'intervention humaine dans les transformations reste prépondérante lorsqu'il y a des alternatives, comme celle ci-dessus du choix de représentation des automates ou celle des protocoles de communication pour les envois de messages. Cette intervention se fait sous forme de paramètres mais aussi décisions interactives. Ce point reste prématuré dans l'état de nos expérimentations.

7 Conclusion

Le logiciel prend une place croissante dans les systèmes cybernétiques. La maintenance de ces systèmes met en évidence le besoin d'outils d'industrialisation qui vont au delà des plate-formes intégrées de développement et de déploiement. Pour éviter la dette technique, on doit faire abstraction des infrastructure et raisonner au niveau des modèles tout en facilitant le raffinement de ces modèles dans des versions exécutables. Les expérimentations menées ici travaillent en ce sens. Enrichir les modèles, formaliser les processus de raffinement, rendre modulaire et personnaliser le développement pour s'appuyer sur des outils de transformation sont les pistes suivies. Il reste encore beaucoup à faire mais les défis sont moteurs. D'un point de vue théorique, les processus de transformation restent peu explorés. Une perspective est de concevoir une algèbre de transformations pour les combiner [2]. D'un point de vue pratique, il faut rationaliser le processus d'ingénierie logicielle sous forme d'une combinaison de décisions et expérimenter une typologie de transformations. D'un point de vue outillage, il faut pouvoir combiner des transformations écrites avec différents langages et qui soient interactives pour que le concepteur influe sur les choix de conception.

Références

[1] Pascal André, Christian Attiogbé, and Jean-Marie Mottu. Combining techniques to verify service-based components. In Proceedings of the International Workshop on domAin specific Model-based AppRoaches to vErificaTion and validaTiOn, AMARETTO@MODELSWARD 2017, Porto, Portugal, February 2017.

[2] Pascal André and Gilles Ardourel. Domain Based Verification for UML Models. In Ludwik Kuzniarz, Gianna Reggio, Jean-Louis Sourrouille, and Miroslaw Staron, editors, Workshop on Consistency in Model Driven Engineering C@Mode'05, pages 47–62, November 2005.

[3] Pascal André, Gilles Ardourel, and J. Christian Attiogbé. Kmelia, un modèle abstrait et formel pour la description et la composition de composants et de services. TSI, 30(6) :627–658, 2011.

[4] Pascal André and Alain Vailly. Développement de logiciel avec UML2 et OCL ; cours et exercices corrigés, volume 6 of Collection Technosup. Editions Ellipses, 2013. ISBN 9782729883539.

[5] C. Atkinson. Component-based Product Line Engineering with UML. Addison-Wesley object technology series. Addison-Wesley, 2002.

[6] Ferenc Belina, Dieter Hogrefe, and Amardeo Sarma. SDL with Applications from Protocol Specification. The BCS Practitioner. Prentice Hall, 1991. ISBN 0-13-785890-6.

[7] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-Driven Software Engineering in Practice : Second Edition. Morgan & Claypool Publishers, 2nd edition, 2017.

[8] Jordi Cabot and Martin Gogolla. Object constraint language (ocl) : A definitive guide. In Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio, editors, Formal Methods for Model-Driven Engineering, volume 7320 of Lecture Notes in Computer Science, pages 58–90. Springer Berlin Heidelberg, 2012.

[9] Anis Charfi, Artur Schmidt, and Axel Spriestersbach. A hybrid graphical and textual notation and editor for uml actions. In Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA '09, pages 237–252, Berlin, Heidelberg, 2009. Springer-Verlag.

[10] Federico Ciccozzi. On the automated translational execution of the action language for foundational uml. Software & Systems Modeling, 17(4) :1311–1337, Oct 2018.

[11] Federico Ciccozzi, Ivano Malavolta, and Bran Selic. Execution of uml models : a systematic review of research and practice. Software & Systems Modeling, 18(3) :2313–2360, Jun 2019.

[12] Sanford Friedenthal, Alan Moore, and Rick Steiner. A Practical Guide to SysML : Systems Modeling Language. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[13] Martin Gogolla, Jarn Bohling, and Mark Richters. Validating uml and ocl models in use by automatic snapshot generation. Software and Systems Modeling, 4(4) :386–398, 2005.

[14] Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML), version 1.4. Technical report, OMG, available at https://www.omg.org/spec/FUML/1.4/, December 2018.

[15] Sahar Guermazi, Jérémie Tatibouet, Arnaud Cuccuru, Ed Seidewitz, Saadia Dhouib, and Sébastien Gérard. Executable modeling with fuml and alf in papyrus : Tooling and experiments. In Proc. of the 1st International Workshop on Executable Modeling in (MODELS 2015)., pages 3–8, Ottawa, Canada, September 2015.

[16] Morten Olav Hansen. Exploration of UML State Machine implementations in Java. Master's thesis, University of Oslo, Norway, February 2011.

[17] Jussi Koskinen. Software Maintenance Costs. Technical report, School of Computing, University of Eastern Finland, Joensuu, Finland, April 2015.

[18] Kevin Lano. Advanced Systems Design with Java, UML and MDA. Computer Science. Elsevier, 1 edition, 2005. ISBN 0-7506-6496-7.

[19] Jacques Lonchamp. Conception d'applications en Java/JEE, Principes, patterns et architectures. Sciences sup - Informatique. Dunod, 1 edition, 2014. ISBN 2-10-071686-7.

[20] Stephen J. Mellor and Marc J. Balcer. Executable UML : A Foundation for Model-Driven Architecture. Object Technology Series. Addison-Wesley, 1 edition, 2002. ISBN 0-201-74804-5.

[21] Iftikhar Azim Niaz, Jiro Tanaka, and Key Words. Mapping uml statecharts to java code. In in Proc. IASTED International Conf. on Software Engineering (SE 2004, pages 111–116, 2004.

[22] Isabelle Perseil and Laurent Pautet. A concrete syntax for uml 2.1 action semantics using +cal. In Proceedings of the 13th IEEE International Conference on on Engineering of Complex Computer Systems, ICECCS '08, pages 217–221, Washington, DC, USA, 2008. IEEE Computer Society.

[23] Romuald Pilitowski and Anna Dereziñska. Code generation and execution framework for uml 2.0 classes and state machines. In Tarek Sobh, editor, Innovations and Advanced Techniques in Computer and Information Sciences and Engineering, pages 421–427, Dordrecht, 2007. Springer Netherlands.

[24] Elena Planas, Jordi Cabot, and Cristina Gómez. Lightweight and static verification of uml executable models. Comput. Lang. Syst. Struct., 46(C) :66–90, November 2016.

[25] Chris Raistrick, Paul Francis, Ian Wilkie, John Wright, and Colin B. Carter. Model Driven Architecture with Executable UML. Cambridge University Press, 2004. ISBN 0-521-53771-1.

[26] L. Rierson. Developing Safety-Critical Software : A Practical Guide for Aviation Software and DO-178C Compliance. Taylor & Francis, 2013.

[27] P. Roques and F. Vallée. UML 2 en action : De l'analyse des besoins à la conception. Architecte logiciel. Eyrolles, 2011.

[28] Tim Weilkiens. Systems Engineering with SysML/UML : Modeling, Analysis, Design. The MK/OMG Press. Elsevier Science, 2008.

Les auteurs remercient Colin Frapper, Clément Jéhanno, Loïc Mahier, Demetre Phalavandishvili, Guillaume Fortin, Romain Brohan, Oussama El Kourri, Antoine Godet et Ronan Gueguen pour leur contribution aux expérimentations de cette étude.