République Algérienne Démocratique et Populaire
Université Abou Bakr Belkaid– Tlemcen
Faculté des Sciences
Département d'Informatique

Mémoire de fin d'études

pour l'obtention du diplôme de Master en Informatique

Option: *Génie Logiciel (G.L)*

Thème

Développement du prototype de génération du code Arduino

Réalisé par :

- Sihem BENDAOUD
- Leila GUERRAB

Présenté le 06 Juillet 2019 devant le jury composé de :

- *M. Fethallah HADJILA* — *Président*
- *Mme. Yassamine SELADJI* — *Examinatrice*
- *M. Mohamed MESSABIHI* — *Encadrant*

Année universitaire : 2018-2019

13 juillet 2019

Résumé :

La génération automatique de code à partir de la conception des systèmes est un domaine de recherche intéressant dans le développement dirigé par les modèles. Cela a conduit en effet la mise en œuvre de nombreux outils de génération de code. Cependant, la génération automatique de code Arduino à partir de modèles abstrait tels que les diagrammes d'état n'est pas totalement prise en charge par les outils existants car il n'y a pas de concordance directe entre les composants du Arduino et les éléments du diagramme d'états d'une part et les constructions syntaxiques des languages de programmation d'autre part.

L'idée de base de notre travail est de développer un prototype de génération du code Arduino depuis un diagramme de machine à état étendu afin de faciliter le développement des systèmes embarqués en particulier. Notre approche est basée sur l'MDE, le language ThingML et le méta-modèle UML pour les diagrammes de machine à état.

Mots Clés : Génération de code, Arduino, métamodèle, machine à état, MDE, ThingML.

ملخص :

يعد إنشاء الرمز تلقائيًا من تصميم الأنظمة مجالًا مثيرًا للاهتمام للبحث في التطوير القائم على النموذج.مما أدى إلى إنتاج أدوات توليد الشفرة. تطبيق مثل هذا الإنشاء التلقائي لرمز الأردوينو من الرسوم البيانية للدولة غير مدعوم بالكامل من قبل لغات البرمجة الحالية لأنه لا يوجد تطابق فردي بين مكونات البرنامج اردوينو وعناصر الرسم البياني للدولة وبنية البرمجة.

الفكرة الأساسية لعملنا هو تطوير نموذج أولي كود اردوينو من مخطط آلة الدولة الموسعة ، من أجل تسهيل تطوير النظم المضمنة على وجه الخصوص. يعتمد منهجنا على أسلوب MDE ولغة ThingML و Uml metamodel لمخططات آلة الحالة.

الكلمات المفتاحية: إنشاء الرمز، اردوينو ،metamodel، ThingML .

Abstract :

Automatic code generation from system design is an interesting area of research in model-driven development. This has led to the implementation of many code generation tools. However, the automatic generation of Arduino code from abstract models such as state diagrams is not fully supported by existing tools because there is no direct correspondence between Arduino components and state diagram elements on the one hand and syntactic constructions of programming languages on the other.

The basic idea of our work is to develop a prototype for generating Arduino code from an extended state machine diagram in order to facilitate the development of embedded systems in particular. Our approach is based on MDE, ThingML and UML meta-model for state-based machine diagrams.

Keyword : Code generation, Arduino, metamodel, state chart, MDE, ThingML.

Remerciements

Tout d'abord, nous tenons à remercier le bon Dieu le tout puissant de nous avoir donné la force et le courage de mener à bien ce travail, également nous remercions nos parents pour leur aide et leur soutien.

Nous remercions sincèrement notre encadrant M. Mohamed MESSABIHI pour l'attention et l'intérêt qu'il a porté à notre travail et pour son soutien, ses précieux conseils, son encouragement, orientations, et sa disponibilité tout au long de l'élaboration de ce projet.Vous avez su partager votre savoir avec patience, une gentillesse et un enthousiasme remarquable.

Nos plus sincères remerciements vont également aux honorables membres du jury, qui ont accepté d'évaluer notre travail. Merci à M. Fethallah HADJILA d'avoir accepté de présider cette soutenance et à Mme.Yassamine SELADJI d'avoir accepté d'examiner ce manuscrit, et de nous faire l'honneur de juger notre travail avec intérêt.

Enfin, que tous ceux qui, d'une façon ou d'une autre nous ont soutenu, encouragé, conseillé, trouvent ici l'expression de notre profonde gratitude.

Dédicaces

« *À mes parents qui ont toujours été là pour moi, et qui m'ont donné un magnifique modèle de labeur et de persévérance.*
À mes soeurs, mon frère, mon beau-frère et mes nièces qui ont été toujours dans mon esprit et mon coeur.
À tout qui me sont chers et proches.
À tous ceux qui ont semé en moi à tout point de vue.
Puisse ce travail témoigne ma profonde affection et ma sincère estime. »

Sihem

« C'est avec profonde gratitude et sincère mot, que je dédie ce modeste travail.
À mes chers parents qui depuis mon jeune âge ont toujours fait leur maximum et qui ont sacrifié leur vie pour ma réussite.
À mes chères soeurs pour leur contribution au succès de ce travail indirectement, et pour tout le soutien moral.
À ma famille et à tous mes proches grands et petits.
À mes amis fidèles et à tous ceux qui nous sont chers. Que Dieu les protège tous... »

Leila

Table des matières

Table des figures

Liste des abréviations

ATL	**A**TLAS **T**ransformation **L**anguage
IOT	**I**nternet **O**f **T**hings
JMI	**J**ava **M**étadata **I**nterface
M2M	**M**odel to **M**odel
M2T	**M**odel to **T**ext
MDA	**M**odel **D**riven **A**rchitecture
MDE	**M**odel **D**riven **E**ngineering
MOF	**M**eta**O**bject **F**acility
OMG	**O**bject**M**anagment **G**roup
UML	**U**nified **M**odeling **L**anguage
XML	E**X**tensible **M**arkup **L**anguage
XSD	**X**ML **S**chema **D**efinition

Introduction générale

Contexte

L'automatisation de la production est souvent l'envie de l'homme, envie d'efficacité, envie d'exactitude, envie de rapidité, envie d'enlever ce qui l'encombre, *etc.* sans oublier le facteur d'économie. Dans la plupart des domaines comme l'automobile, l'électronique, le textile, l'alimentation *etc.* l'automatisation prend une place considérable. Paradoxalement, en informatique, le résultat reste encore un très modeste.

De nos jours, l'électronique est de plus en plus substituée par l'électronique programmée. On parle aussi de systèmes embarqués ou d'informatique embarquée. Son objectif est de simplifier les schémas électroniques et de se faire diminuer l'utilisation des composants électroniques, réduisant ainsi le coût d'élaboration d'un produit. Il en résulte des systèmes performants mais de plus en plus complexes [4].

Parmi les tendances actuelles pour contrer cette complexité croissante des systèmes on peut citer l'ingénierie dirigée par les modèles qui permet, entre autres, d'assurer la génération automatique du code à partir de modèle de haut niveau en utilisant des modèles intermédiaires. Le modèle ThingML est un bon exemple qui illustre cette démarche.

Problématique

La vulgarisation des systèmes embarqués a, en effet, engendré une complexité accrue dans leur conception et leur validation. À cela s'ajoute des contraintes commerciales qui exigent une production dans des délais de plus en plus courts.

Des processus de génération de codes ont donc été proposés afin de simplifier le développement et la validation des systèmes. Le concepteur munit d'une abstraction du système à partir de laquelle il affirme et approuve les exigences et traduit automatiquement cette abstraction en un code exécutable.

Objectifs

L'objectif principal de notre projet est de faciliter le travail de l'ingénieur en lui évitant de spécifier les détails d'implémentation. Ce processus a un effet considérable sur la capacité de conserver les propriétés initialement exprimées dans la spécification. Plus le modèle est de haut niveau d'abstraction, plus sa spécification est simple, mais plus le travail du générateur de code est complexe. Ce générateur de code est en mesure de produire le code approprié, d'une façon cohérente, rapide et sûre en utilisant les données issues du modèle abstrait (conception) combinées avec les règles fondamentales du développement du logiciel (règles de bonnes pratiques en génie logiciel). Il apporte donc une aide précieuse aux projets de développement, en s'occupant notamment du travail laborieux liés aux aspects techniques. Il apporte également des bénéfices économiques importantes ainsi qu'une amélioration des processus et méthodes de développement. Il contribue considérablement à la qualité de code à sa cohérence et sa facilité à être réutiliser.

Dans le reste, nous détaillons ces idées à travers le plan du présent manuscrit :

— **Le premier chapitre :** présente les notions de base de l'ingénierie systèmes, des systèmes complexes et embarqués, ainsi qu'un aperçu sur l'internet des objets et sa relation avec Arduino.

— **Le deuxième chapitre :** présente l'ingénierie dirigée par les modèles pour le développement des systèmes en présentant ces deux axes principaux qui sont la méta modélisation et les approches de transformation de modèles.

— **Le troisième chapitre :** décrit particulièrement les matériels et méthodes que nous avons utilisés pour le développement du prototype de génération du code Arduino où nous présentons les définitions de base, les différentes approches et technologies utilisées et nous terminons avec une expérimentation à travers une étude de cas concret.

Enfin, nous concluons ce mémoire puis nous donnons quelques perspectives qui permettront de donner une suite à ce travail.

Chapitre 1

L'ingénierie système

1.1 Introduction

Aujourd'hui l'internet ne se limite plus aux ordinateurs, smart-phones et tablettes, il existe à présent une multitude d'objets connectés qui forment un réseau étendu : l'Internet des objets(Ido) ou Intrernet of things (IoT) en anglais. Récemment, L'IoT fera partie intégrante du monde physique et des systèmes complexes ce qui a permis aux équipements de devenir plus intelligents.Une grande partie, voire la majorité des milliards d'appareils intelligents de l'Internet des objets seront des systèmes embarqués équipés d'un système d'exploitation et beaucoup nécessiteront un système d'exploitation temps réel. Ce chapitre est consacré à l'ingénierie système, les systèmes complexes, l'électronique embarquée, de même sur l'internet des objets et sa relation avec Arduino.

1.2 L'ingénierie système

Un système est un ensemble d'éléments identifiables, interdépendants, c'est-à-dire liés entre eux par des relations telles que, si l'une d'elles est modifiée, les autres le sont aussi et par conséquent tout l'ensemble du système est modifié et transformé. C'est également un ensemble borné dont on définit les limites en fonction des objectifs (propriétés, buts, projets, finalités) que l'on souhaite privilégier [52].

L'ingénierie système (IS) est une démarche méthodologique générale qui englobe l'ensemble des activités adéquates pour concevoir, faire évoluer et vérifier un système apportant une solution économique et performante aux besoins d'un client tout en satisfaisant l'ensemble des parties prenantes [36]. Plus précisément, l'ingénierie système peut se définir comme :

— Un processus coopératif et interdisciplinaire de résolution de problèmes.

— S'appuyant sur les connaissances, méthodes et techniques issus de la science et de l'expérience.
— Mis en œuvre pour définir, faire évoluer et vérifier la définition d'un système (ensemble organisé de matériels, logiciels, compétences humaines et processus en interaction).
— Apportant une solution a un besoin opérationnel identifié conformément à des critères d'efficacité mesurables qui satisfasse aux attentes et contraintes de l'ensemble de ses parties prenantes et soit acceptable pour l'environnement.
— En cherchant à équilibrer et optimiser sous tous les aspects l'économie globale de la solution sur l'ensemble du cycle de vie du système.

L'ingénierie des systèmes englobe les activités suivantes : l'analyse de besoins, la spécification et la conception du système, le développement des composants qui composent le système et l'ensemble de ses sous-systèmes, l'intégration du système et la qualification opérationnelle [50].

la figure 1.1 ci-dessous montre les différentes étapes de la production d'une voiture en suivant le cycle de vie de l'ingénierie système (à partir de l'analyse des besoins jusqu'au retrait).

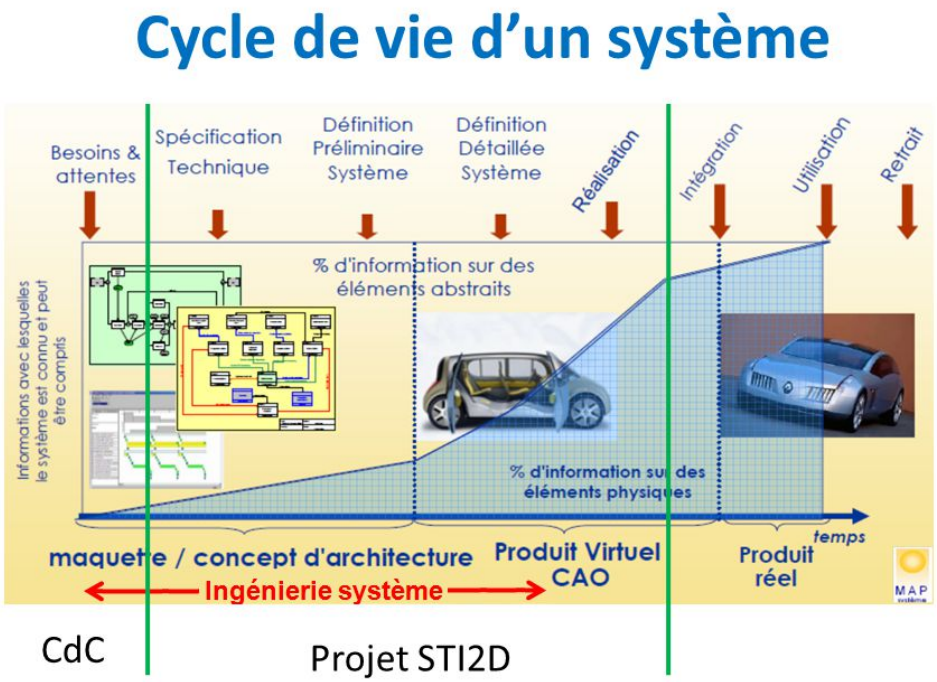

FIGURE 1.1 – Cycle de vie du production d'une voiture

D'autre part **"Ingénierie des systèmes complexes"** selon [47], IS est une approche interdisciplinaire munie avec des moyens permettant la réalisation réussie d'un système complexe. Elle intègre toutes les disciplines et les groupes spécialisés, y compris les moyens humains ou matériels, formant un processus de développement structuré [17]. L'objectif de cette approche est d'évoluer et vérifier un ensemble intégré et équilibré de personnes, de produits et de processus qui sont inclus dans le processus de conception du système. L'ingénierie des systèmes complexes recouvre les mêmes activités que l'ingénierie des systèmes. Par exemple pour la vérification et la construction d'un système complexe la simulation offre un outil adéquat car la construction d'un système complexe est plus complexe que le système lui-même.

1.3 Les systèmes complexes

Les systèmes complexes sont partout ! un système complexe est un ensemble constitué de nombreuses entités dont les interactions produisent un comportement global difficilement prévisible. Chaque entité peut agir sur un groupe et modifier le système [39]. Que ce soit dans notre environnement - mobilité, réseaux sociaux...etc tout est lié et en évolution permanente. Et donc la réalisation d'un grand système technique moderne a besoin la mise en place de processus d'ingénierie complexes allant parfois jusqu'à impliquer plusieurs milliers d'ingénieurs provenant de multiples spécialités différentes.

La difficulté de concevoir un système est liée notamment à sa complexité[31]. La complexité provient principalement de deux aspects : l'émergence de niveaux d'organisation différents et l'hétérogénéité des composants, celle-ci tient à au moins trois facteurs :

— Le nombre et la nature de ses éléments. Le nombre d'éléments peut être élevé. Il peut même être éventuellement variable au cours du temps et la nature de ces éléments peut également être variable.
— La nature de son organisation interne, qui est liée aux relations qui existent entre ses éléments. Ceux-ci peuvent former des réseaux, des hiérarchies,. . . etc. L'organisation du système peut également varier au cours du temps.
— Le couplage avec l'environnement ; plus le système est en interaction forte avec l'environnement plus il est exposé à l'incertitude et à l'imprévisibilité de ses réactions.

La figure 1.2 ci-dessous représente une infographie qui fournit un aperçu d'un système complexe d'une ville intelligente .

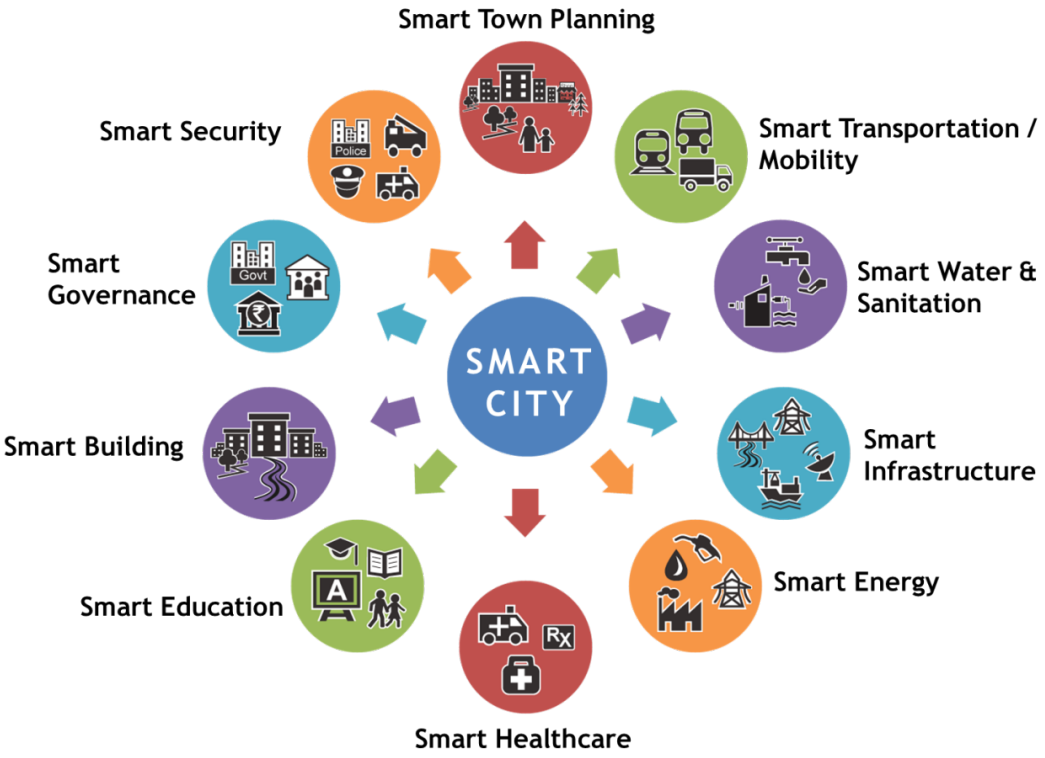

FIGURE 1.2 – Un système complexe de "Smart city"

1.4 Les systèmes embarqués

Un système embarqué est un système complexe qui intègre du logiciel et du matériel conçus ensemble afin de fournir des fonctionnalités données. Un système embarqué est un système électronique et informatique autonome[12], qui ne possède pas d'entrées/sorties standard (clavier, souris,...etc). IL est généralement produit pour résoudre un problème bien précis, pour exécuter une tâche bien définie [28]. Un système embarqué est considéré comme un système mixte, il interagit avec l'environnement lequel il appartient [30].
Par rapport aux autres systèmes informatisés, les systèmes embarqués sont caractérisés par :

- Encombrement mémoire (mémoire limitée, pas de disque en général).
- Consommation d'énergie (batterie : point faible des SE) .
- Poids et volume.

— Autonomie .
— Mobilité.
— Communication (attention : la communication affecte la batterie).
— Contraintes de temps réel.
— Contraintes de sécurité.
— Coût de produits en relation avec le secteur cible.

La structure de base d'un système embarqué :

Le système embarqué se retrouve comme composant d'un système plus large, interagissant avec des processus externes en récupérant de l'information via des senseurs et en agissant dessus via des actuateurs. Il peut également être en liaison directe avec l'utilisateur.

La figure 1.3 montre comment le système embarqué interagit avec son environnement pour lequel il rend des services bien précis (contrôle, surveillance, communication,..).

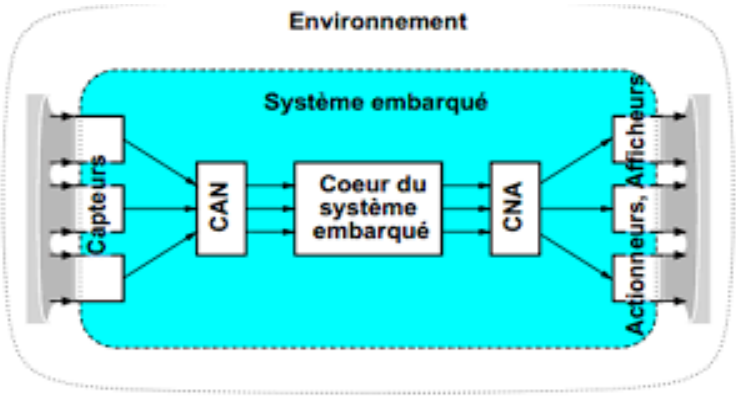

FIGURE 1.3 – Structure de base d'un système embarqué dans son environnement [46]

Une information est captée par l'environnement, une transformation est réalisée sur cette information, avant d'être lisible par le cœur du système embarqué (constitué d'une partie matérielle et une partie logicielle), qui effectue un traitement spécifique à cette information pour rendre à son tour une réponse à son environnement, cette réponse doit être transformée avant d'être envoyée à l'environnement [21]

La croissance globale de marché de l'embarqué est liée à la diffusion généralisée des technologies de l'embarqué dans le contexte de ce que l'on appelle "l'internet des objets". Présents essentiellement dans des secteurs traditionnels comme l'automobile et l'aéronautique, les systèmes et les logiciels embarqués diffusent aujourd'hui dans tous les secteurs innovant, par exemple dans les domaines de la ville numérique, de la mobilité, de la santé... Grâce à

la connectivité et aux réseaux, les systèmes embarqués deviennent de plus en plus communicants. Cette connectivité permet de développer de nombreux produits et services destinés aux professionnels comme au grand public.

1.5 L'internet des objets (IOT)

— Définition de l'IoT :
Le terme d'Internet des objets ne possède pas encore une définition officielle et unifiée, qui s'explique par le fait que l'expression est encore jeune et que le concept est encore en train de se construire[44]. L'Internet des choses (IoT) est un système de dispositifs informatique interconnecté, de machines mécaniques et numériques, d'objets, qui sont pourvues d'identificateurs uniques et de la capacité de transférer des données sur un réseau sans nécessiter des interconnexions homme-à ordinateur[35].

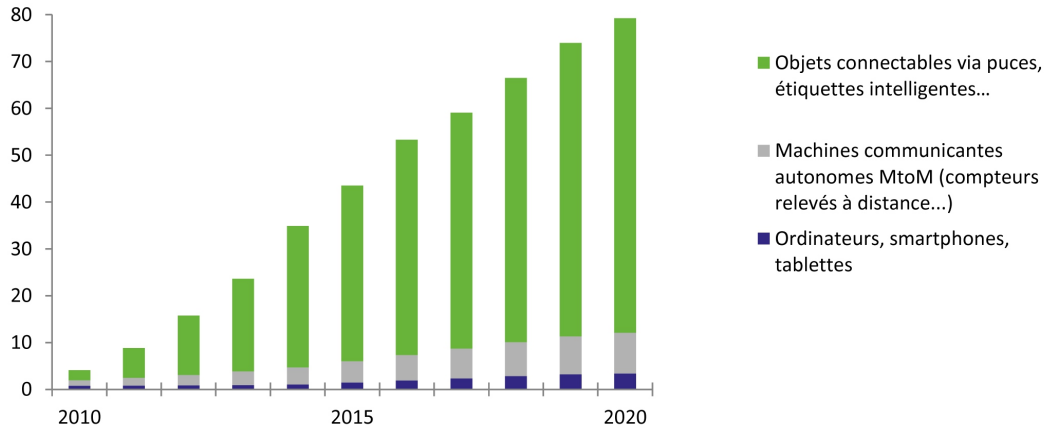

FIGURE 1.4 – Évolution du nombre d'objets connectés par type (milliards)[1]

— Principes des IoT :
L'internet des objets est composé de plusieurs éléments complémentaires ayant chacun ses propres spécificités. Il permet à l'aide des systèmes d'identification électroniques normalisés et des dispositifs mobiles sans fil, d'identifier directement et sans ambiguïté des objets physiques, ainsi que pouvoir récupérer, stocker, transférer et traiter sans discontinuité les données s'y rattachant. L'IoT est une combinaison d'innovations technologiques récentes et de solutions déjà existantes[48].

Chaque objet est muni d'une identification électronique unique capable de lire et transmettre à travers un protocole dans le réseau internet. Il est nécessaire cependant de définir la nature de l'objet, ses fonctionnalités, sa position dans l'espace, l'historique de ses déplacements, etc. Pour effectuer ce lien entre physique et virtuel, le dispositif technique doit donc modéliser des contextes réels et les rendre virtuels.[14]

D'un point de vue conceptuel, l'Internet des objets affecte, à chaque objet une identification unique sous forme d'une étiquette lisible par des dispositifs mobiles sans fil, afin de pouvoir de se communiquer les uns avec les autres. Ce réseau crée une passerelle entre le monde physique et le monde virtuel. D'un point de vue technique, l'IdO consiste l'identification numérique directe et normalisée (adresse IP, protocole http...) d'un objet physique grâce à un système de communication sans fil (puce RFID, Bluetooth ou WiFi).

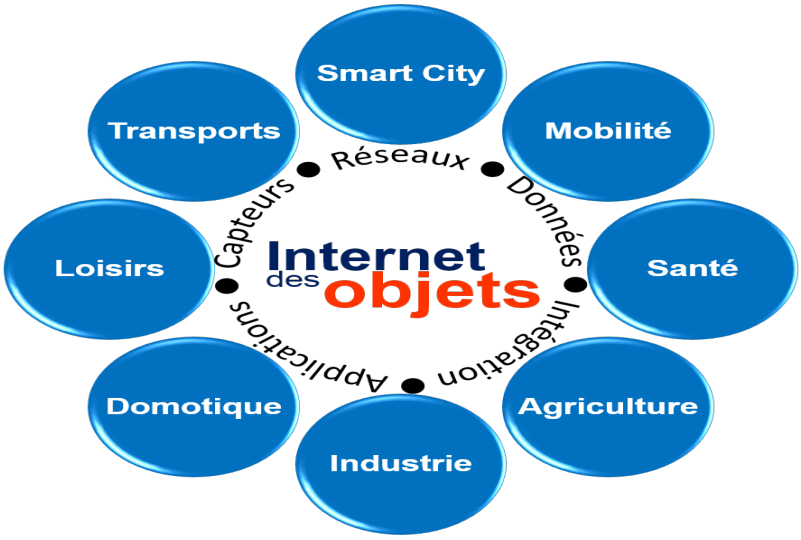

FIGURE 1.5 – Les domaines d'application de l'internet des objets

1.6 Arduino et Internet des objets

Le monde d'internet des objets (IoT) est divisé en deux parties principales [19] :

Hardware : inclut les appareils qui envoient et reçoivent des données sur Internet. Comme Arduino, Raspberry PI.

Software : c'est le logiciel nécessaire dans les deux appareils mentionnés ci-dessus(Arduino et Raspberry PI), en plus des bases de données énormes Big-Data, ainsi la plate-forme spécifiée à ceci IoT Platform.

Arduino est un outil permettant de construire des dispositifs qui peuvent interagir avec l'environnement qui les entoure.

Arduino est une plate-forme open source [7] de prototypage d'objets interactifs à usage créatif constitué d'une carte électronique (figure 1.6) et d'un environnement de programmation. Arduino est un système magique, qui se situe au cœur de toutes ses actions. Il collecte des informations à partir de ses capteurs, évaluant ainsi le monde réel qui l'entoure. Il prend ensuite des décisions basées sur les données recueillies et provoque en retour des actions, sous forme de sons, de lumière, ou encore de mouvements. Arduino est un microcontrôleur [22], autrement dit un ordinateur très simple. Il ne peut pas faire beaucoup de choses en même temps, mais ce qu'on lui dit de faire, il le fait très bien.

FIGURE 1.6 – Carte Arduino Uno

Aujourd'hui, le monde dans lequel nous vivons dépend énormément de la technologie. Cela signifie qu'il est nécessaire de disposer d'une main-d'œuvre plus qualifiée sur le plan technique pour concevoir et gérer les technologies requises. De nombreuses nouvelles technologies sont interactives, ce qui facilite la création d'environnements dans lesquels l'apprentissage peut être réalisé de manière pratique, en recevant des informations en retour, en améliorant la compréhension et en créant de nouvelles connaissances [20].

Pour répondre à la difficulté croissante de concevoir et analyser des systèmes de plus en plus complexes et embarqués et à une palette de plus en

plus large de technologies, l'Object Management Group [1] a introduit l'architecture dirigée par les modèles. Celle-ci place le modèle au centre du processus de production à travers des techniques de manipulation et de transformation. La MDA définit une approche qui facilite la conception en séparant les préoccupations que sont la spécification fonctionnelle (comportement du système modélisé) et l'implémentation (langage de programmation, composants logiciels). De cette manière, la validation fonctionnelle est réalisée indépendamment des choix d'implémentation. Une même spécification fonctionnelle peut ainsi être implantée sur différentes plates-formes d'exécution en étant traduite en différentes spécifications d'implémentations.

1.7 Conclusion

Dans ce chapitre nous avons parlé de l'ingénierie système, des systèmes complexes en donnant au début une définition, puis nous avons expliqué qu'est-ce qu'un système embarqué et enfin nous avons présenté quelques notions d'internet des objets et sa relation avec l'Arduino, dans le prochain chapitre nous allons évoquer les notions de base de l'ingénierie dirigée par les modèles.

1. The Object Management Group (OMG), cf : http ://www.omg.org/.

Chapitre 2

L'ingenierie dirigée par les modèles

2.1 Introduction

Le développement du logiciel fait face à une augmentation de la complexité des systèmes malgré l'utilisation des techniques spécifiques qu'on les appelle modèles de cycles de vie d'un logiciel comme le modèle en cascade, le modèle en V et le modèle en W. À l'instar d'autres sciences, la modélisation est de plus en plus utilisée pour maîtriser cette complexité des systèmes. L'ingénierie dirigée par les modèles (IDM), ou Model Driven Engineering (MDE) en anglais s'inscrit dans cette évolution en prônant l'utilisation systématique de modèles pour automatiser une partie des processus de développement suivis par les ingénieurs. Donc l'IDM propose de modéliser les applications à un haut niveau d'abstraction où elle se place le modèle au cœur du processus de conception puis génère le code de l'application à partir des modèles.

2.2 Concepts de base de l'IDM

L'ingénierie dirigée par les modèles est une approche générale et ouverte qui fait suite à la proposition du standard MDA (Model-Driven Architecture)(figure 2.1 ci-dessous) proposé par l'OMG [1] en 2000. Elle a permis plusieurs améliorations significatives dans le développement des logiciels en permettant de se concentrer sur l'utilisation intensive des modèles et leur transformation [8].

L'IDM a permis de prendre en charge la croissance de la complexité des systèmes logiciels développés où la modélisation de ces systèmes se base sur l'usage intensif des modèles [18]. De cette manière, le développement est réalisé avec un niveau d'abstraction plus élevé que celui de la programmation

1. The Object Management Group (OMG), cf : http ://www.omg.org/.

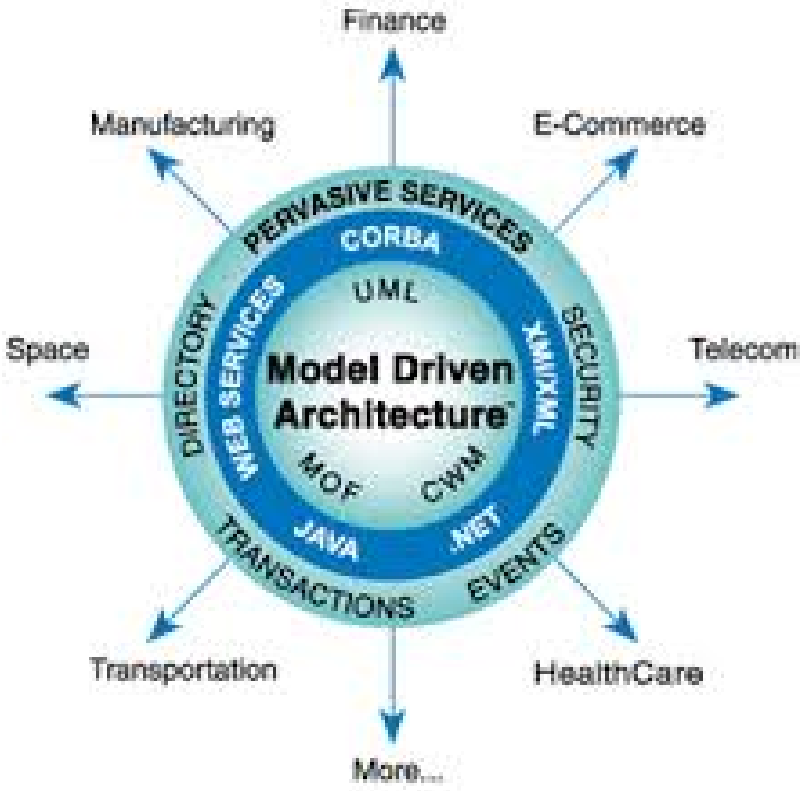

FIGURE 2.1 – L'architecture du MDA [3]

classique. Cette approche permet alors d'automatiser, où au moins de dissocier et de reporter, la part du développement qui est proprement technique et dédiée à une plate-forme d'implémentation. L'ingénierie dirigée par les modèles (IDM) est donc une approche du génie logiciel sur laquelle le modèle est considéré comme une première présentation du système à modéliser, et qui vise à développer, maintenir et faire évoluer le logiciel en réalisant des transformations de ce modèle. Au sens large, le paradigme de l'IDM propose d'unifier tous les aspects du processus du cycle de vie en utilisant les notions de modèle et de transformation [8].

En ingénierie logiciel, le modèle permet de mieux répartir les tâches et d'automatiser certaines d'entre elles [10]. Il contribue de manière importante à la réduction des coûts et des délais. Par exemple, les plate-formes de modélisation savent maintenant exploiter les modèles pour faire de la génération de code.

Le choix du modèle influence fortement sur les solutions obtenues. Les systèmes sont modélisés par un ensemble de modèles indépendants. Selon les modèles employés, la démarche de modélisation n'est pas la même [16].

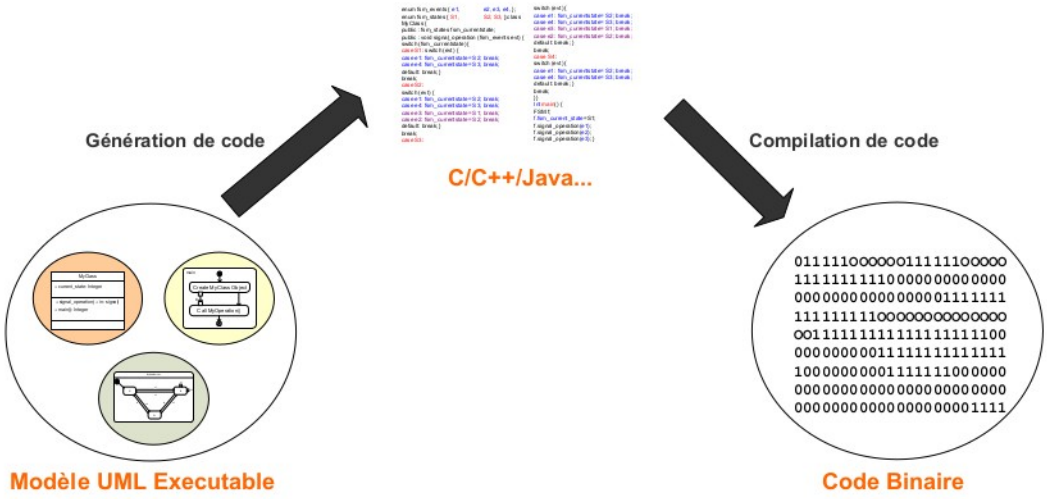

FIGURE 2.2 – Processus de transformation des modèles

Généralement, une approche dirigée par les modèles pour le développement des systèmes et plus particulièrement les systèmes embarqués comporte 3 étapes : la modélisation, la génération du code à partir du modèle et la compilation du code généré (figure 2.2) .

Chacune de ces 3 étapes peut comporter à son tour d'autres étapes selon le besoin et le type du système embarqué considéré. En effet, pour les systèmes embarqués ayant des contraintes de sûreté de fonctionnement, la première étape (la modélisation) s'accompagne généralement d'une étape de validation de comportement du système [33], [38] et [54]. Si le concepteur souhaite modéliser à la fois la partie matérielle et la partie logicielle de son système, la modélisation se décompose dans ce cas en sous étapes : la modélisation de l'application, la modélisation de la plateforme et l'association entre les fonctionnalités de l'application et les éléments de la plateforme [6]. De la même manière, l'étape de la génération de code peut comporter plusieurs transformations de modèles jusqu'à atteindre le code cible choisi (généralement un langage de programmation tel que C/C++/Java,..).

2.3 Modélisation

A partir des travaux de l'OMG , de Bézivin[9] et de Seidewitz[45], Un modèle est une description abstraite d'un système construite dans un but donné. Donc, il doit pouvoir être utilisé pour répondre à des questions sur le système. Un modèle est une abstraction d'un système, modélisé sous la forme d'un ensemble de faits construit dans une intention particulière.

La modélisation d'un système avant sa réalisation permet de mieux comprendre le fonctionnement du système, de maîtriser sa complexité et d'assurer sa cohérence [16].

L'histoire du développement des systèmes est une histoire de montée dans le niveau d'abstraction. Les langages de modélisation sont plus abstraits que les langages de programmation (C/C++/Java) qui eux même sont plus abstraits que les langages machines (assembleur, binaire). Le langage UML [2] (Unified Modeling Language) est le langage de modélisation orienté objet standardisé par l'OMG (Object Managment Group) depuis plus que 15 ans,et un langage visuel dédié à la spécification, la construction, et la documentation des artefact d'un système . Au début, en définissant une syntaxe unifiée pour la modélisation des systèmes, UML intervenait uniquement dans la phase de conception. Il facilitait ainsi la conception des systèmes et la communication entre les concepteurs. L'implémentation du système, bien qu'elle doive être compatible avec les modèles UML conçus, était une étape indépendante. C'est en développant des outils de génération automatique de code à partir des modèles UML que ce langage intervient désormais dans l'étape de l'implémentation des systèmes. La génération automatique de code a ainsi contribué à la réduction du temps d'implémentation des systèmes et à la production d'un code plus consistant avec le modèle [26],[34]. Les modèles à partir desquels une génération automatique de code exécutable est possible sont appelés modèles exécutables.

Les diagrammes UML nous aident à améliorer notre vue d'ensemble et à éclaircir les relations unissant les éléments du logiciel, tout en nous permettant d'ignorer les détails sans intérêt. [32].

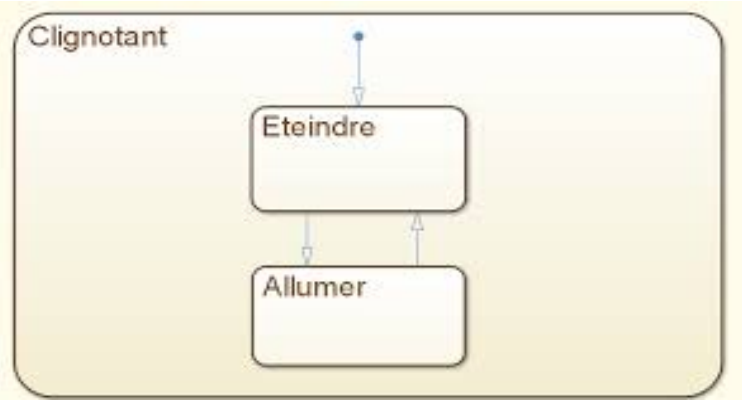

FIGURE 2.3 – Diagramme d'états-transitions du bouton poussoire

2. Uml :site web :https ://www.uml.org/

La figure 2.3 montre un exemple simple de modélisation d'un diagramme d'état transition d'un bouton poussoire

• Cet automate possède deux états (Allumé et Eteint) et deux transitions correspondant au même évènement : la pression sur un bouton d'éclairage domestique.

• Cet automate à états finis illustre en fait le fonctionnement d'un tel rupteur (bouton poussoir) dans une maison. Lorsque l'on appuie sur un bouton d'éclairrage, la réaction de l'éclairage associé dépendra de son état courant (de son historique) : s'il la lumière est allumée, elle s'éteindra, si elle est éteinte, elle s'allumera.

2.4 Métamodélisation

Chaque modèle est exprimé à partir d'un langage de modélisation qui doit être clairement défini, et comme l'IDM prend la définition de tout est modèle donc le langage de modélisation prend la forme d'un modèle, appelé méta-modèle [43]. Un méta-modèle est un modèle qui permet d'exprimer un modèle. Le méta-modèle est l'élément de structuration de modèles. Il permet de définir de façon précise les différents formalismes qui permettent de construire les modèles. Dans [27]. la métamodélisation est une activité qui consiste à définir des métamodèles contemplatifs qui reflètent la structure statique des modèles. La métamodélisation est une activité qui consiste à définir le métamodèle d'un langage de modélisation. Il s'agit non seulement de produire des métamodèles mais aussi de définir la sémantique du langage, de mettre en œuvre des analyseurs, des compilateurs, des générateurs de code et plus généralement, à construire un ensemble d'outils exploitant les métamodèles.

Dans le contexte de l'IDM, les modèles sont traités via l'utilisation de langage. Un langage est présenté soit graphiquement, soit textuellement, soit par les deux (composé de texte et de symboles graphiques). Cependant les symboles graphiques ou les mots d'un langage textuel ne peuvent pas être exploités sauf si on leur donne une interprétation par rapport aux éléments du système modélisé et pour comprendre et manipuler un élément de modèle, il faut qu'il puisse être interprété. Un système peut être décrit par un modèle où les deux peuvent être considérés comme deux ensembles associés par une relation d'interprétation. L'interprétation établit un lien entre un ou plusieurs éléments du modèle et un ou plusieurs éléments du système modélisé. Pour attribuer un sens aux éléments d'un modèle, la définition explicite

d'une sémantique, qu'elle soit formelle ou non, est indispensable pour relier les éléments de modèles aux informations. L'ensemble des informations décrites constitue le domaine sémantique. Là ou les données d'un modèle qui décrivent une information du domaine sémantique sont appelées des méta-informations.[43]

L'OMG a défini une architecture de métamodélisation à quatre niveaux comme le montre la figure 2.4, connue sous l'appellation de l'architecture dirigée par les modèles (Model-Driven Architecture : MDA) :

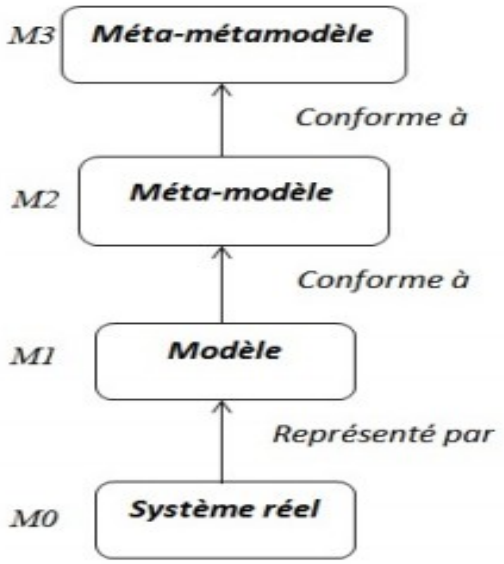

FIGURE 2.4 – Relation entre système, modèle, méta-modèle et méta-métamodèle.

— **Niveau M0** : c'est la couche la plus basse qui présente les données du monde réel (concrète). Les éléments de cette couche sont une instance de la couche supérieure qui est M1.
— **Niveau M1** : c'est le niveau modèle où chaque élément du monde réel est représenté par un modèle. Les modèles de cette couche sont décrits par un métamodèle de la couche M2.
— **Niveau M2** : dans ce niveau sont représentés les métamodèles pour décrire les langages de modélisation.
— **Niveau M3** : c'est le plus haut niveau, c'est le niveau de méta-métamodèle qui permet de définir la structure des métamodèles du niveau M2, et comme le méta-métamodèle est auto-descriptif, il permet de décrire sa propre sémantique.

Il est important de noter que la proposition initiale d'OMG dans l'approche MDA était d'utiliser le langage UML comme unique langage de modélisation. Cependant, les limites d'UML ont rapidement été atteintes et il a fallu rapidement ajouter la possibilité d'étendre le langage UML, par exemple en créant des profils, afin d'exprimer de nouveaux concepts relatifs à des domaines spécifiques. Ces extensions devenant de plus en plus importantes et la communauté MDA a élargi son point de vue en considérant des langages de modélisation spécifiques à un domaine [55].

2.5 Transformation des modèles

Dans l'IDM, la transformation de modèles définit la partie cruciale du processus de développement de logiciels. Généralement, elle exprime la phase de translation d'un modèle en un autre type de modèle à l'aide d'un programme de transformation écrit avec un langage dédié. Un processus de transformation de modèles se base sur les éléments suivants : la définition des métamodèles et leurs modèles et la spécification des règles de transformation. Les métamodèles peuvent être définis avec des langages de métamodélisation comme MOF , Ecore , KM3 (Kernel MetaMeta- Model). Les modèles sont calculés par une instanciation de leurs métamodèles. Pour la transformation de modèles, elle se base sur la spécification des règles de correspondance entre les éléments des métamodèles associés. Dans ce qui suit, nous définissons les notions suivantes : la transformation de modèles, la transformation et les règles de transformation selon [29] .

— **Transformation de modèles :** Une transformation de modèles est une génération automatique d'un modèle cible à partir d'un modèle source selon la définition de la transformation.
— **Définition de transformation :** C'est un ensemble de règles de transformation qui décrivent comment un modèle source peut être transformé en un modèle cible.
— **Règle de transformation :** une règle de transformation est une description de la manière dont une ou plusieurs constructions dans un langage source peuvent être transformées en une ou plusieurs constructions dans un langage cible.

2.6 L'intérêt de la transformation de modèles

La transformation de modèles joue un rôle fondamental dans le développement des logiciels et vise des objectifs divers comme la automatise la génération automatique du code, la translation de modèle, la migration, et l'ingénierie inverse. La figure 2.5 représente les différents activités du transformations des modèles.

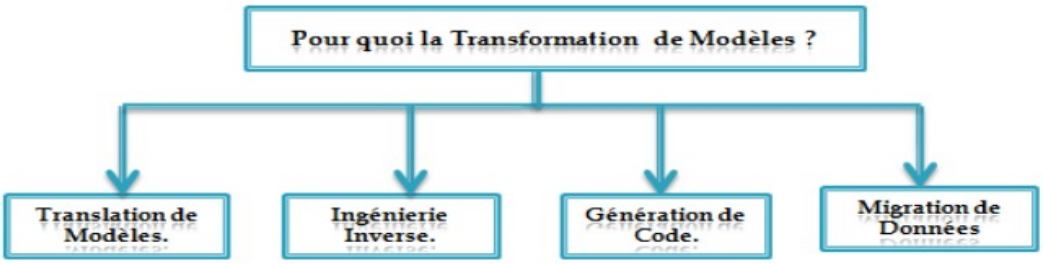

FIGURE 2.5 – Les activités du transformations des modèles

— **Génération automatique du code**
La génération de code est l'une des principales activités dans la technologie de l'IDM qui a été utilisée fréquemment dans la littérature de recherche à des buts différentes. En général, l'objectif est de transformer un modèle vers un un modèle spécifique représentant un code source pouvant être exécuté.

— **Translation de modèle**
La translation de modèle est une transformation d'un modèle à un modèle équivalent mais dans une différente représentation. Une des raisons pour l'utilisation de la translation de modèle est de faciliter l'interopérabilité. Il est souvent souhaitable d'utiliser et d'échanger des modèles entre les différents outils de modélisation, et même entre les différents langages de modélisation.

— **Migration des modèles**
Une autre utilisation de la transformation de modèles est la migration de modèles. La migration peut être définit comme une transformation d'un modèle écrit dans un langage en un autre modèle écrit dans un autre langage, en gardant le même niveau d'abstraction des modèles.

— **Ingénierie inverse**
L'ingénierie inverse est un autre exemple d'une transformation de modèles. Elle est l'inverse de la génération de code, et permet de comprendre la structure du programme. Prenant le code source en entrée, elle permet de construire un modèle mental ou visuel (par exemple un modèle UML) à un niveau supérieur d'abstraction, afin de faciliter la compréhension du code et connaître comment il est structuré. L'ingénierie inverse est utilisée pour l'adaptation des applications vers les nouvelles technologies.

2.7 Les approches de transformation

Il existe dans la littérature, plusieurs classifications des approches de transformation de modèles. Généralement, on distingue deux classes principales[13] : les transformations de "modèles à modèles" et les transformations de "modèles à code source" .

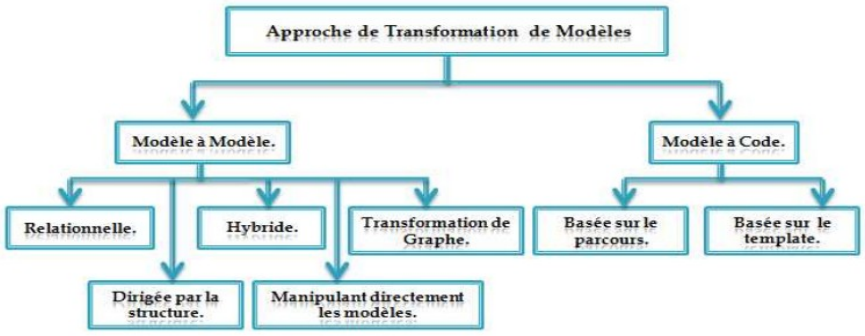

FIGURE 2.6 – Les approches de transformation de modèles

la figure 2.6 ci-dessus résume cette classification d'approches de transformation de modèles, et montre que pour chacune de ces deux catégories, on distingue plusieurs sous-catégories.

2.7.1 Transformation de modèle à modèle (M2M)

Les transformations de type modèle vers modèle consistent à transformer un modèle source en un modèle cible, ces modèles peuvent être des instances de différents méta- modèles. Elles offrent des transformations plus modulaires et faciles à maintenir.

Cette catégorie porte sur la transformation de modèle à modèle. Les méthodes utilisées sont diverses et varies, parmi elles nous allons présenter les

deux qui suivent :

Approche par manipulation directe : cette approche offre une représentation interne d'un modèle plus une API pour le manipuler. ils sont généralement implémentés dans un environnement oriente objet, qui peut également fournir une infrastructure minimale pour organiser les transformations. Mais les utilisateurs doivent implémenter les règles de transformations à partir de zéro en utilisant un langage de programmation comme Java. Des exemples de cette approche comprennent Jamda[2] et JMI (Java Métadata Interface est une API basée sur le MOF). L'approche par manipulation directe est de type impératif, elle est située a un bas niveau d'abstraction.

Approche relationnelle : elle se base sur les transformations de type déclaratif reposant sur les relations mathématiques ou l'idée de base est d'indiquer les types d'éléments de la source et de la cible dans une relation et de les spécifier à l'aide des contraintes. Par définition, une telle transformation n'est pas exécutable telle quelle, mais son exécution est possible par l'adjonction d'une sémantique exécutable, par exemple au moyen de la programmation logique. Il n'y a pas d'effets de bord avec l'approche relationnelle, contrairement a l'approche par manipulation directe. La plupart des transformations basées sur l'approche relationnelle permettent la multidirectionnalité des règles, et fournissent des liens de traçabilité . Elles n'autorisent pas la transformation sur place (source et cible doivent être distinctes). Des exemples sur cette approche sont RFP [37], MOF 2.0[41], QVT[42] [40], ATL[3][5].

FIGURE 2.7 – L'approche basée sur l'IDM pour la transformation des modèles M2M (Exemple ATL)

3. ATL.site web.https ://www.eclipse.org/atl/

2.7.2 Transformations modèle à code (M2T)

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

Génération de code par parcours de modèle : C'est une approche de génération de code la plus basique qui consiste à parcourir la représentation du modèle en entrée et a écrire un code en sortie. L'exemple le plus connu avec cette approche est l'environnement orienté objet Jamda[2] dont les modèles UML sont représentés par des classes où une API manipule les modèles et un outil est utilisé pour la génération de code.

Génération du code par template : C'est l'une des approches la plus utilisée pour la génération de code. Cette approche consiste à définir des canevas de modèles cibles. Ces canevas sont des modèles cibles paramétrés ou modèles templates. L'exécution d'une transformation consiste à prendre un modèle template et à remplacer ses paramètres par les valeurs d'un modèle source. D'une autre façon , elle consiste à utiliser comme RHS Right Hand Side (le côté droit de la règle) un texte fixe dans lequel certaines parties sont variables : elles sont renseignées en fonction des informations récupérées dans le modèle source LHS Left Hand Side (le côté gauche de la règle), et peuvent notamment faire l'objet d'itérations. Parmi les langages les plus utilisées, on a Xpand [4] , Acceleo [5] .

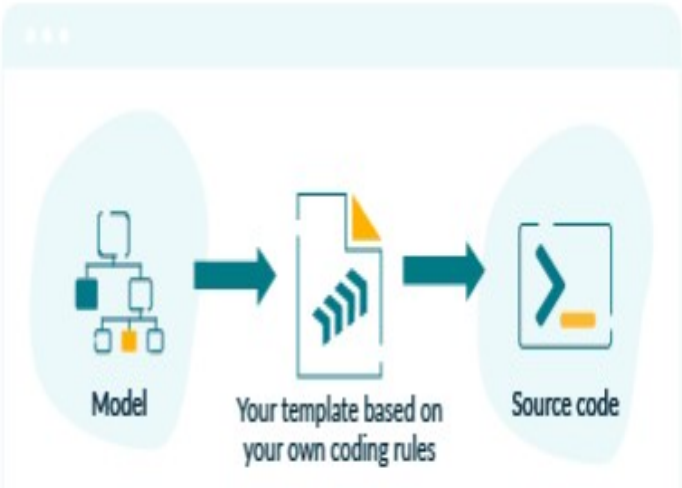

FIGURE 2.8 – L'approche basée sur l'IDM pour la transformation des modèles M2T (Exemple Acceleo)

4. Xpand. site web. http ://wiki.eclipse.org/Xpand
5. acceleo. site web. http ://wiki.eclipse.org/Acceleo.

2.8 Conclusion

Dans ce chapitre, nous avons présenté une vue générale sur l'ingénierie dirigée par les modèles où l'accent a été mis sur la présentation des concepts de base de l'IDM en général et plus spécifiquement la transformation de modèles. Nous avons aussi décrit le rôle principal de la transformation dans le cadre de l'approche IDM et ses activités. L'objectif visé dans ce travail de recherche est de prendre en charge les notions de modélisation et de génération du code dans le processus de transformation de modèles. Ces deux concepts devenus très importants pour assurer une bonne application de l'IDM.

Chapitre 3

Génération de code Arduino à partir des machines à états

3.1 Introduction

La génération automatique du code reste vraiment une technique très utile car elle permet de simplifier les développements et soulager le développeur de certaines lourdeurs de mise en oeuvre.

La génération de code se fait, en générale, par l'utilisation des données (langage source) et le contexte d'utilisation pour générer le langage cible. À la différence du compilateur qui fait référence à la traduction exacte d'un langage source vers un langage cible, le générateur de code est plutôt un interprète, qui ne traduit pas fidèlement un langage vers un autre [51].

Ce chapitre représente la dernière partie de ce rapport, il traite la phase qui a pour objectif l'implémentation de notre prototype de génération de code. Nous débutons par la description de nouvelles notions utilisées dans l'approche de génération du code. Ensuite une présentation sur les technologies utilisée pour mettre en place cette dernière. Finalement nous donnons un aperçu sur le travail réalisé en terme d'expérimentation sur une étude de cas concrète.

3.2 Machines à états UML

Le langage UML offre plusieurs diagrammes pour la spécification du comportement d'un système. Parmi ces diagrammes, les machines à états (appelés aussi diagrammes d'états transitions) [11].

Les diagrammes d'états permettent de spécifier les modèles des machines à états (notions d'état et de transition entre états).

Les machines à états UML sont utilisées pour décrire le cycle de vie du système en définissant les états que peut occuper ce système et les protocoles de passage d'un état à un autre.

Les machines à états UML sont inspirées des machines à états de Harel [24] qui ont elles mêmes été inspirées des FSMs (Finite State Machine) [53]. Les FSMs représentent le premier modèle de comportement orienté état-transition. Il permet la modélisation du comportement d'un système à l'aide d'états simples et de transitions (déclenchées par des événements) entre ces états. Les notions de concurrence (états orthogonaux) et de hiérarchie (état composite) ainsi que la possibilité d'exécuter un comportement en passant d'un état à un autre (effet d'une transition) ne sont pas supportées dans les FSMs.

Harel [24] a étendu ce modèle afin de pouvoir spécifier le comportement des systèmes complexes et concurrents. Son modèle est connu sous le nom de StateCharts. Les StateCharts de Harel définies dans les années 80 ont été améliorées et prises en compte dans l'outil STATEMATE [25] qui supporte d'autres notions utiles pour la modélisation des systèmes telles que la possibilité d'exécuter des actions en entrant et en sortant de chaque état (entry, exit), la définition de transition composée et la priorité entre les transitons.

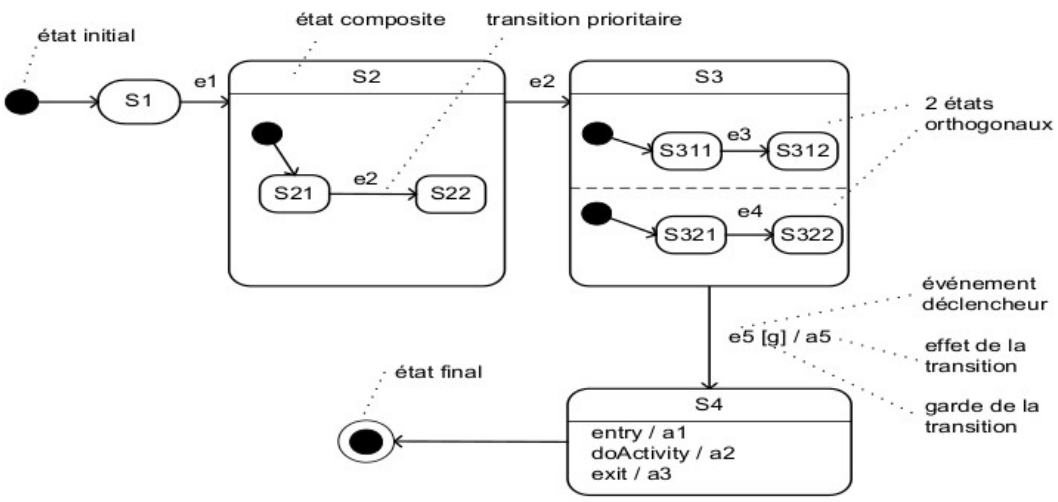

FIGURE 3.1 – Exemple d'une machine à états UML [15]

Concrètement, un diagramme d'états-transitions est un graphe qui représente un automate à états finis, c'est-à-dire une machine dont le comportement des sorties ne dépend pas seulement de la sollicitation de l'entrée, mais aussi d'un historique des sollicitations passées. La machine change d'état

en réponse à des événements extérieurs donnant lieu à des transitions entre états(la figure3.1) ci-dessus. Un automate à états finis est graphiquement représenté par un graphe comportant des états, matérialisés par des rectangles aux coins arrondis, et des transitions, matérialisées par des arcs orientés liant les états entre eux.

3.3 Machines à états étendu

Les machines à états UML ont été introduites dans la norme UML depuis sa création et ont subi plusieurs évolutions. La version actuelle des machines à états UML mettent les StateCharts de Harel dans un contexte orienté objet. Elles ajoutent aussi la possibilité d'exécuter un comportement tant que le système est dans un état donné (do Activities).

Cependant, les machines à états UML gèrent les priorités entre les transitions d'une manière différente que les StateCharts de STATEMATE. Dans STATEMATE, les transitions des états parents sont plus prioritaires que les transitions des états fils. UML définit des priorités inverses (les transitions des états fils sont plus prioritaires).

Afin de mieux modéliser les systèmes embarqués, nous avons ajouté de nouveaux concepts aux diagrammes d'état UML et étendu les notions standards, nous avons également modifié certaines priorités des transitions des machines à états pour prendre en compte certaines spécificités des systèmes embarqués.

Le diagramme d'état transition utilise habituellement des états logiques. Et vu que les états constituent les unités de base des machines d'état, nous avons défini un nouveau concept dans les machines à états pour les composants physiques du système embarqué, on prend comme exemple : boutons, led, buzzer, etc que nous appelons "État physique composite".

Ces états physiques composites contiennent des sous-états qui représentent les différents comportements du composant physique, et un pseudo-état qui prend en valeur le numéro de port du carte Arduino à travers lequel sera connecté le composant.

La figure 3.2 ci-dessous illustre la correspondance entre la connexion réelle du composant "LED" dans la carte Arduino et sa modélisation en tant qu'état physique composite qui contient un numéro de port.

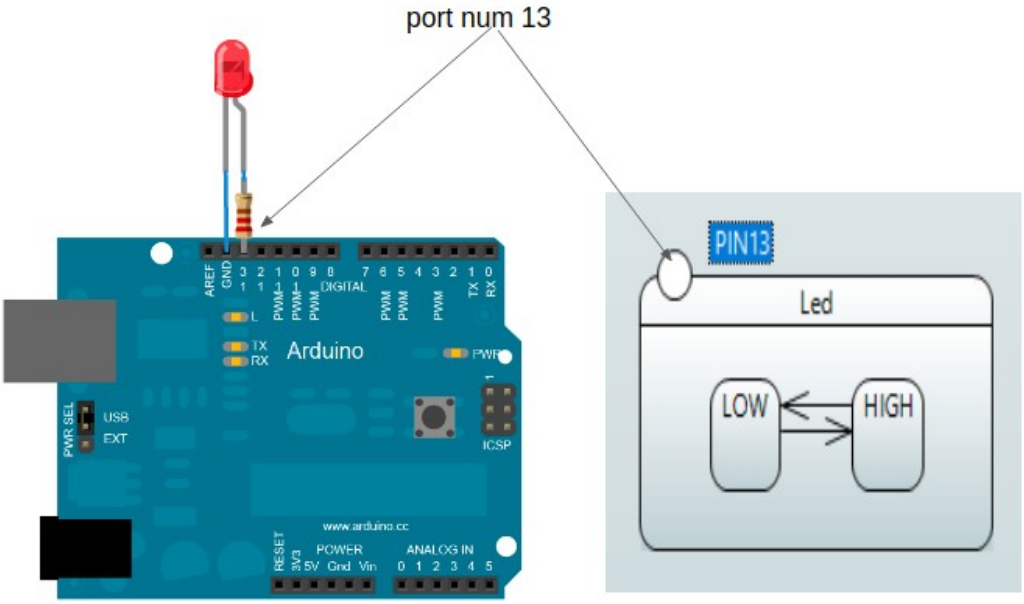

FIGURE 3.2 – Exemple pour faire clignoter une Led avec Arduino et sa modélisation

De plus il y a des états composites logiques qui représentent le système. Ces derniers contienent des sous états qui décrivent les différents étapes de système ainsi qu'un pseudo-état. Mais cette fois-ci le pseudo-état prend la valeur initiale de l'état, c'est-à-dire dans ce cas a le rôle de l'initialisation.

Pour les transitions, on a défini 2 types : les transitions locales représentent le fonctionnement des composants et sont ignorées dans le code, et les autres transitions peuvent prendre des conditions "Gardes" ou des effets.

Pour les états logiques, dans les transitions on garde les mêmes priorités qu'en UML, c'est-à-dire les transitions des états fils sont plus prioritaires que les transitions des états parents. Quant aux états physiques, la priorité est accordée aux transitions qui contiennent un état source et un état "target" avec des états parents différents.

La figure 3.3 ci-dessous représente un extrait d'un diagramme de machine à états d'un assesseur en utilisant les composants Arduino. Ce diagramme montre les différents nouveaux concepts que nous avons mentionnés ci-dessus.

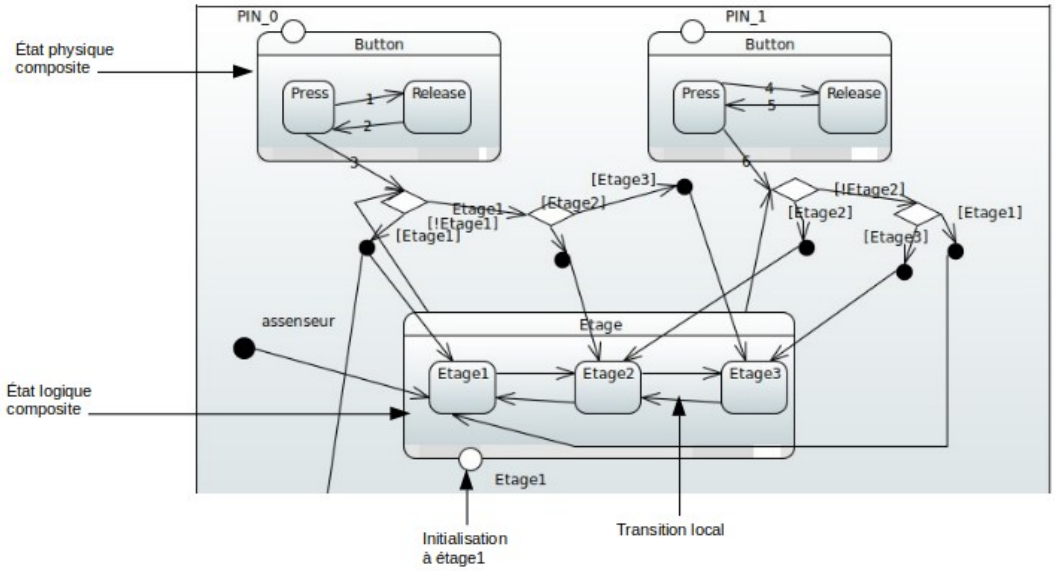

FIGURE 3.3 – Démonstration sur les nouveaux concept du machine à états

3.4 ThingML

ThingML [1] est un langage de modélisation pour les systèmes intégrés et distribués. Le nom ThingML est une référence au terme Internet des objets et signifie langage de modélisation des "Thing" [49].

L'idée de ThingML est de développer une chaîne d'outils pratiques d'ingénierie logicielle dirigée par un modèle, qui cible des systèmes intégrés aux ressources limitées, tels que les capteurs à faible puissance et les dispositifs à microcontrôleur. ThingML est développé en tant que langage de modélisation spécifique incluant des concepts pour décrire des composants logiciels et protocoles de communication.

Le formalisme utilisé est une combinaison de modèles d'architecture, de machines à états et d'un langage d'action impératif. Dans notre travail, l'accent sera mis sur l'ensemble d'outils pour ThingML. L'ensemble d'outils se compose de deux types d'éditeurs permettant à l'utilisateur de créer et éditer le code ThingML, transformer les modèles ThingML en diagrammes et générer le code dans différentes langages, par exemple : C, Java, Arduino , Scala [2]...

L'outil que nous avons utilisés est Eclipse avec le plugin ThingML.

1. ThingML :site web :http ://www.thingml.org/
2. Scala :https ://www.scala-lang.org/

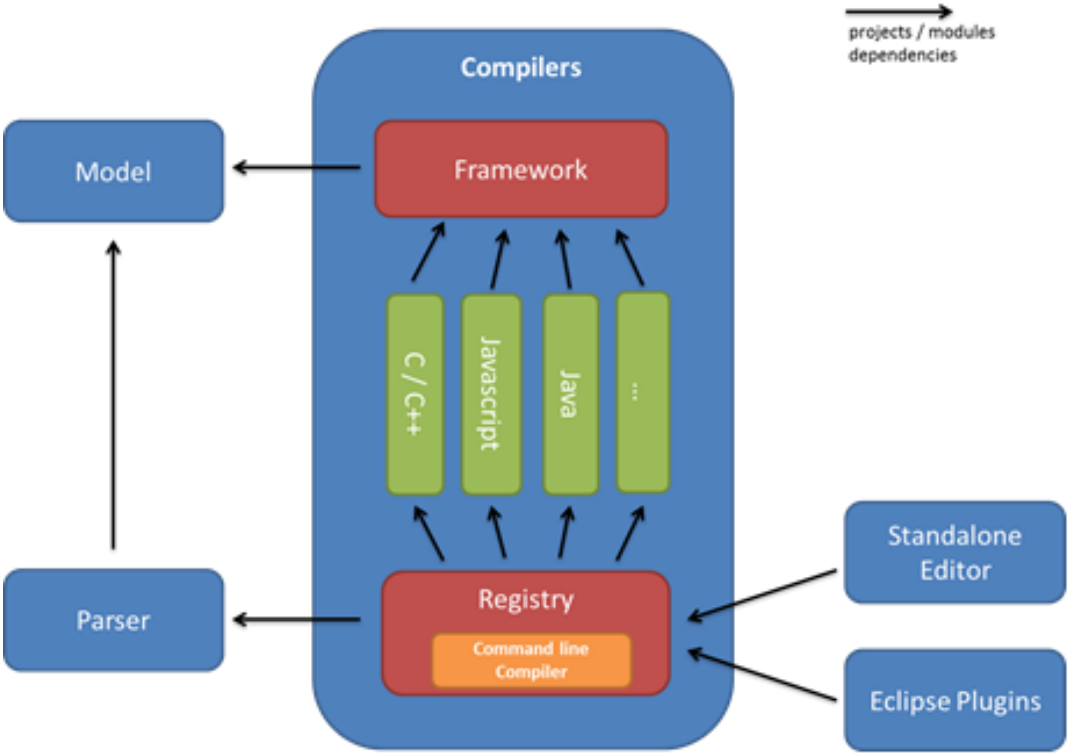

FIGURE 3.4 – les principaux sous-modules du projet "Compilers" et leurs dépendances

Le cadre de génération de code ThingML est structuré en un ensemble de modules. La figure 3.4 présente les principaux sous-modules du projet "Compilers" ainsi que leurs dépendances. L'idée est d'avoir un framework de compilation qui ne dépend que du modèle ThingML. Ce projet-cadre doit englober tout le code et les aides à partager entre les compilateurs. Il définit également les interfaces (en tant que classes abstraites) pour tous les compilateurs ThingML. Ci-dessous, les modules individuels correspondent à la mise en œuvre de différentes familles de compilateurs. L'idée de ces modules est de rassembler des ensembles de compilateurs ayant les mêmes langages cibles. Enfin, le module de registre regroupe tous les compilateurs et fournit un utilitaire simple pour les exécuter à partir de la ligne de commande.

Le projet HEADS comprend des concepts permettant de décrire à la fois les composants logiciels et les protocoles de communication. Il offre aux développeurs la possibilité de déployer la même mise en œuvre sur diverses plates-formes (java, javascript, c / c ++ et Arduino), ainsi que de l'étendre à de nouvelles plates-formes [23]. L'architecture conceptuelle de HEADS ainsi que ses principaux acteurs est décrite dans la figure 3.5.

HEADS IDE comprend deux parties principales : les outils de conception

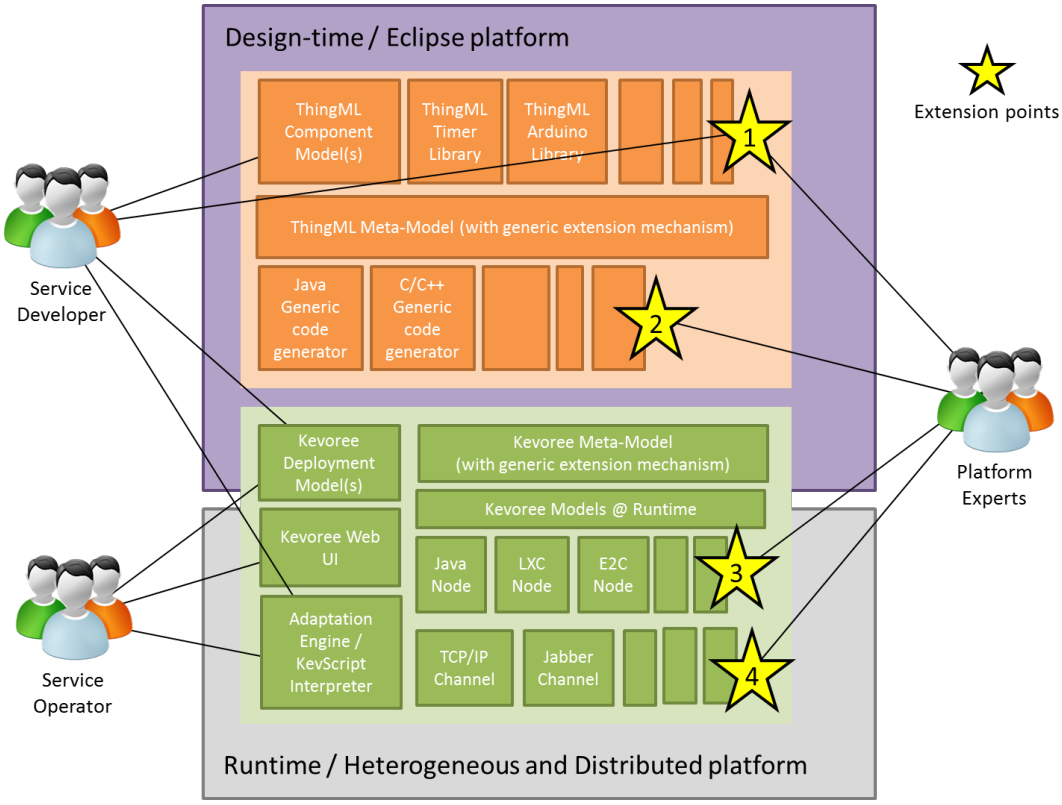

FIGURE 3.5 – L'architecture conceptuelle de HEADS

dans la partie supérieure de la figure et les outils d'exécution dans la partie inférieure de la figure.

HEADS IDE est destiné au développeur HD-Service et à l'opérateur HD-Service afin de créer, déployer, surveiller et faire évoluer les HD-Services. La distinction entre développeur de service et opérateur réside dans le fait que le développeur s'intéresse principalement aux outils de conception HEADS (basés sur ThingML) et que l'opérateur quant à lui s'intéresse plutôt aux outils d'exécution HEADS (basés sur Kevoree).

Par ailleurs, l'IDE HEADS est destiné à être étendu par des experts en plates-formes. HEADS IDE a été conçu comme un framework open-source qui possède un ensemble de points d'extension pour prendre en charge de nouvelles plateformes et canaux de communication et différents aspects d'une plateforme cible particulière.

3.5 Génération du code ThingML à partir des machines à états

Pour générer le code ThingML, nous nous sommes basés sur la notion de transformation modèle à code (M2T) qui se base lui même sur la description de template comme indiqué dans 2.7.2. Nous avons utilisé comme modèle (source) le diagramme de machine à état que nous avons étendu avec de nouvelles notions comme indiqué dans la section 3.3.

3.5.1 Outils et technologies utilisés

Pour développer le prototype de génération de code il y a plusieurs environnements tels que
NetBeans [3], Android Studio [4] et Eclipse [5]. Dans notre travail nous avons choisi Eclipse car il est plus utilisé et plus performants et il intègre les plugins d'une manière plus simple et flexible.

Pour modéliser le système nous avons choisi l'outil "Papyrus". Et pour développer la template de génération du code, on a utilisé le plugin Acceleo. Dans la suite de cette section , nous donnons une brève présentation de chacun de ces outils.

— **Eclipse :**
Eclipse est une plateforme universelle d'intégration d'outils de développement. Il s'agit en effet d'un IDE (Environnement de Développement Intégré) ouvert, facilement extensible et qui n'est pas lié spécifiquement à un langage de programmation. Cette plateforme fournit à la base, un ensemble de services pouvant être éventuellement enrichis par le biais de plugins.
Eclipse Modeling Framework (EMF) est un framework Java de modélisation et un outil de génération de code pour construire des applications basées sur des modèles. Depuis un modèle de spécifications décrit en XMI, EMF fournit des outils et un support de moteur d'exécution pour produire des classes Java. De plus EMF permet de stocker les modèles sous forme de plusieurs fichiers reliées. Les modèles peuvent être spécifiés en utilisant des documents Java, UML, XSD, XML, puis sont importés dans EMF. Par contre EMF ne propose pas d'éditeur graphique pour la modélisation. Le plus important est

3. NetBeans : https ://netbeans.org/
4. Android Studio : https ://developer.android.com/studio
5. Eclipse : https ://www.eclipse.org

qu'EMF fournit les fondements à l'interopérabilité avec d'autres outils ou applications basés sur EMF.

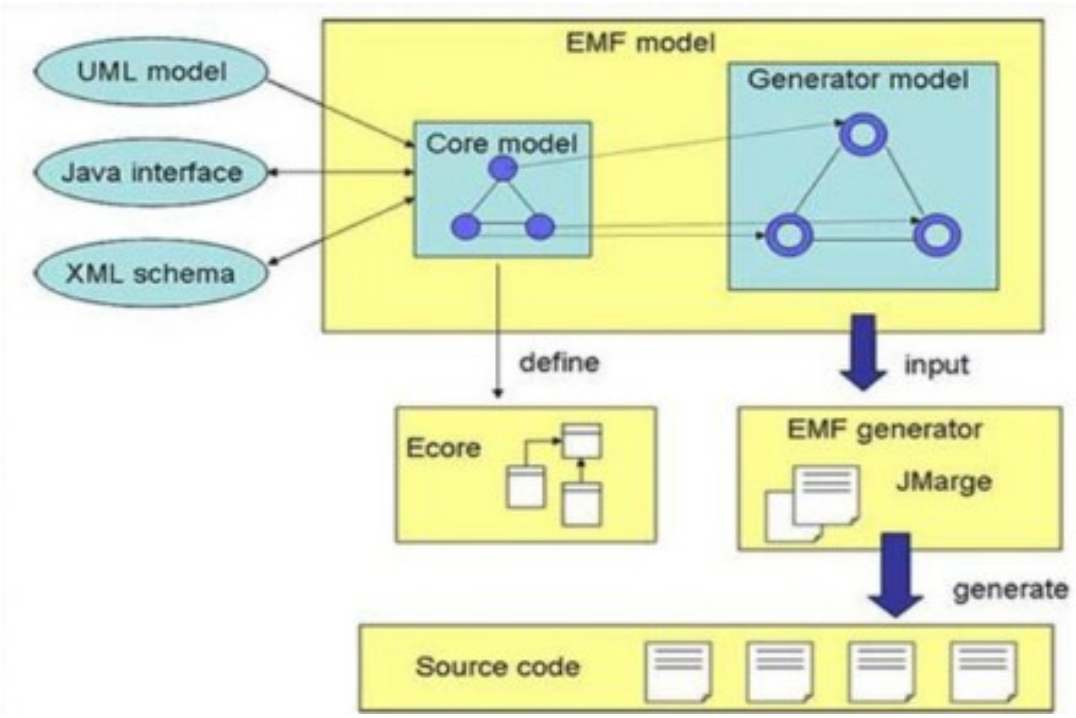

FIGURE 3.6 – EMF framework de modélisation et génération de code

— **Papyrus :**
Outil de modélisation gratuit sous Eclipse, il présente l'avantage d'exportation de modèle, éditer les diagrammes et de créer ses propres diagrammes en dehors de ceux de base. Etant un Plugin sous Eclipse, il permet d'exporter et importer des projets réalisés sous Enterprise Architect [6] ou Magicdraw [7].
Le plugin papyrus UML permet la génération de codes en C++, et sa rétro conception. La génération de codes est un peu particulière avec ce plugin, il ne génère aucune opération ni fonction automatiquement. Pour que papyrus UML les génère,s il faut les créer soi-même et les appliquer à la classe concernée. Une fois cette étape est réalisée il pourra les générer. La démarche est tout de même complexe et consommatrice en terme de temps, mais elle laisse à l'utilisateur le choix de générer ce dont il a besoin.

6. Enterprise Architect : https ://sparxsystems.com/
7. Magicdraw :https ://magicdraw-uml.informer.com/

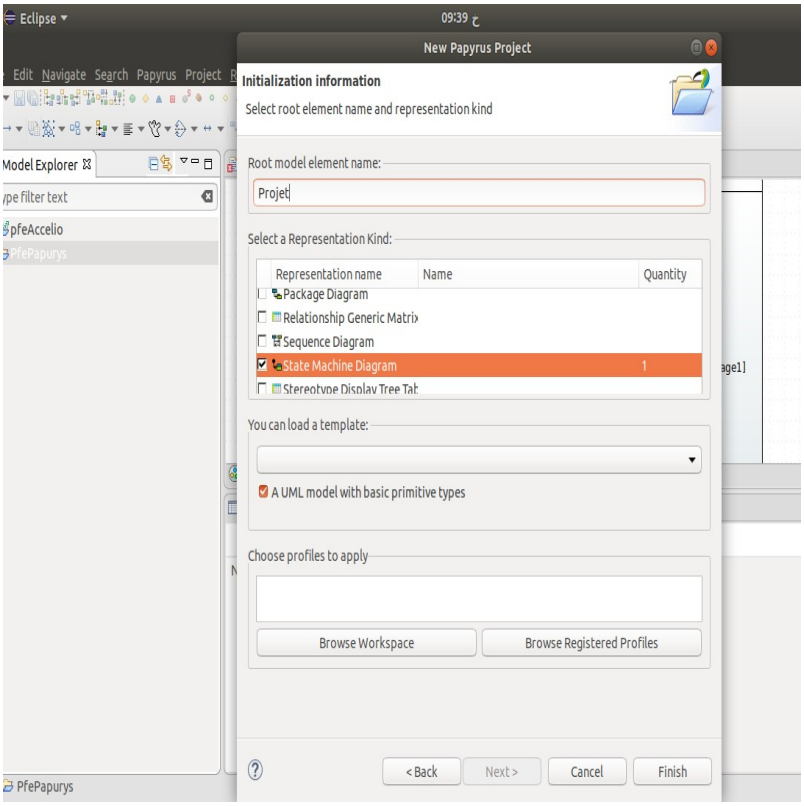

FIGURE 3.7 – Création du diagramme de machine à états sous Papyrus

La figure 3.7 représente un exemple sur la création d'un diagramme de machine à états sous l'outil Papyrus.

— **Acceleo :**

Accelo est un outil de génération de code open source sous Eclipse. Il permet de concevoir des modules de génération de code dans un langage choisi par le développeur, à partir d'un ou plusieurs modèles, et fournit aussi des modules de génération de code prêts à être utilisés (UML vers Java, UML vers C, UML vers Thingml, etc.).

Acceleo est profondément intégré à l'IDE Eclipse avec un éditeur à part entière avec mise en évidence de la syntaxe, détection des erreurs en temps réel, solutions rapides, refactoring et bien plus encore. Il est également livré avec des vues dédiées pour aider à naviguer dans le générateur de code et à accéder rapidement aux modèles de conception de génération de code.

La figure 3.8 représente le principe de fonctionnement de l'outil Acceleo.

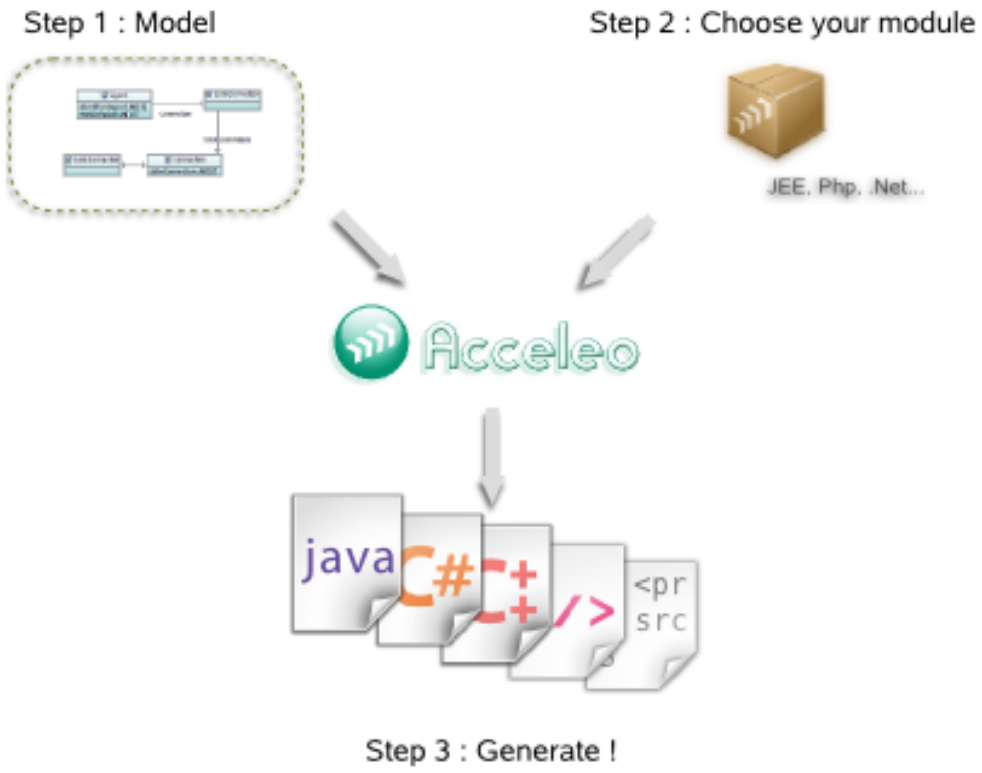

FIGURE 3.8 – Génération du code par Acceleo

3.5.2 Notre approche de génération de code

La génération du code Thingml dans ce projet est faite pour faciliter notre travail afin de générer le code Arduino qui est déjà supporté par le language ThingML, et de même dans le but d'utiliser des technologies qui existent déjà et qui sont intégrées dans le language ThingML, sachant que ce dernier est récent et il était évalué à travers plusieurs études de cas et utilisé dans le développement.

L'idée de base de notre approche est de générer le code ThingML depuis un diagramme d'état transition en utilisant une template Acceleo. Cette template utilise un métamodèle UML pour interpréter le modèle source et extraire les données nécessaires et les tisser avec les données du code ThingML (le rôle d'Acceleo).

Dans ce qui suit nous allons monter un extrait de code source de la template Acceleo qui génère le fichier ThingML.

```
1  [module main('http://www.eclipse.org/uml2/5.0.0/UML', 'http://www
     .eclipse.org/papyrus/sysml/1.4/SysML', 'http://www.eclipse.org/
     papyrus/sysml/1.4/SysML/Blocks')/]
2  [template public genClass(model : Model)]
```

```
3     [file (model.name.concat('.thingml'), false, 'UTF-8')]
4       [for (stateMachine : StateMachine |model.eContents(
      StateMachine))]
5         [for (region: Region| stateMachine.eContents(Region))]
6 [if region.eContents(State).name->select(s: String | s.toLower().
      startsWith('button')) <> 0]
7 import "composants/BUTTON.thingml"
8 [/if]
9 ...
```

La ligne 1 : les urls du métamodèle utilisés.
La ligne 2 : la récupération du model.
La ligne 3 : la création du fichier thingml qui porte le nom du model avec l'extension ".thingml".
La ligne 4 : la création d'un objet qui porte le nom du state machine.
La ligne 5 :la récupération des régions pour créer le thingml.
La ligne 6 : la récupération des états pour importer les autres fichiers dont il a besoin.

Après la compilation de cette template, elle va générer un fichier ThingML avec l'extension ".thingml".

Le diagramme d'état transition utilise des états logiques. Dans ce projet on a défini un nouveau concept dans les états qui s'appelle "État physique" comme indiqué dans la section 3.3 pour bien modéliser le comportement logique et physique du composant.

À ce niveau nous avons défini un domaine de définition pour chaque composant dans la template Acceleo, c'est-à-dire nous avons spécifié à chaque composant qui va prendre un état physique un mot-clé spécial.

Ce domaine de définition va nous aider à la recherche d'un mot clé dans la template pour importer et intégrer les donnes nécessaires, et orienter la template juste aux composants qu'il faudrait utiliser pour éviter les problèmes de la complexité temporelle.

Le code source ci-dessous montre comment récupérer les fichiers thingml déjà développés des différents composants selon le nom de l'état physique.

```
1 ...
2 [if region.eContents(State).name->select(s: String | s.toLower().
      startsWith('button')) <> 0]
3  import "composants/BUTTON.thingml"  [/if]
4  ...
```

Le domaine de définition est défini comme suite :

— Pour les noms des composants physiques, il est obligé de préciser comme préfixe à ce nom le nom réel du composant comme "button", "led", "buzzer", etc. On ignore la casse et peu importe la suite de ce mot, par exemple buttonRouge, BUTTONVert, Ledbleu...etc.
Cette précision du mot clé va permettre de préciser à la template qu'il s'agit bien d'un composant physique et le chercher selon son préfixe.

```
1 {
2       [for (state:State | region.eContents(State))]
3
4      [if state.name.toLower().startsWith('button')]
5
6      required port [state.name.toLower()/] {receives press,
       release}
7   ...
```

La linge 4 montre comment récupérer l'état selon son préfixe.

— Pour les états du système, on considère le nom de l'état composite logique comme une variable et ses sous-états comme des valeur pour cette variable. Et pour l'initialisation, il prendra la valeur du pseudo-state qui lui est attribué, comme montre la ligne 3 dans le code source ci-dessous.

```
1 {...
2  [elseif state.isComposite]
3 [for(initState:Pseudostate | state.eContents(Pseudostate))]
4     [if initState.kind.toString() <> 'initial']
5 property [state.name.toUpper()/] : String = "[initState.name
     /]"   [/if] [/for]   [/if]
```

Concernant le métamodèle nous avons utilisé le métamodèle UML intégré dans l'outil Papyrus, et qui est généré automatiquement lors de modélisation du diagramme de machine à état. Et dans la dernière étape, ce fichier ThingML va générer lui-même le code Arduino grâce aux fonctionnalités intégrées dans Thingml (HEADS).

La figure 3.9 ci-dessous résume notre approche :

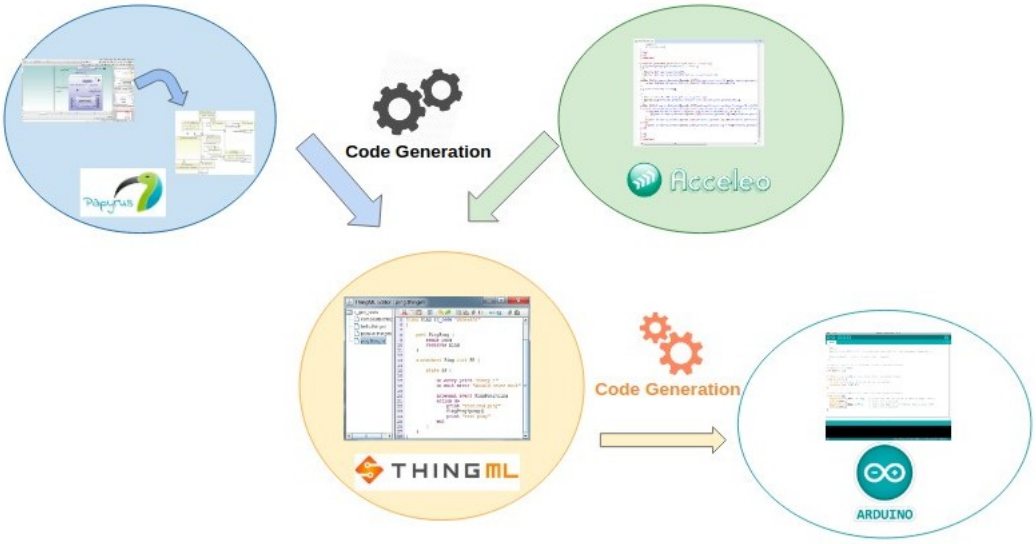

FIGURE 3.9 – Le cycle de vie de notre approche

3.6 Étude de cas

Dans les sections précédentes, nous avons présenté et décrit notre approche pour la génération du code Arduino. Afin de valider cette approche, nous avons utilisé le cas d'étude de feu tricolore. Les feux de signalisation routière qui se trouvent aux grandes intersections et qui permettent de réguler la circulation.

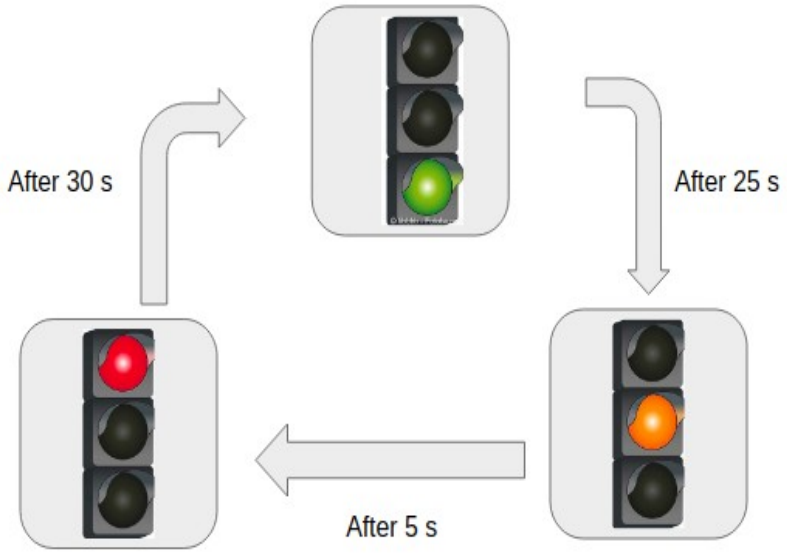

FIGURE 3.10 – Les états successives du feu tricolore

Les feux tricolores sont comme leur nom l'indique une animation lumineuse qui passe successivement de l'état vert, à l'état orange puis à l'état rouge, avant de repasser à l'état vert, et ainsi de suite durant toute sa période de fonctionnement comme montre la figure 3.10 ci-dessus.

IL est à noter que les feux sont pour la plupart, des circuits électroniques programmés pour allumer les lampes pour des durées bien définies. Pour cela, les concepteurs ont créé des programmes qui permettent aux lampes de s'allumer un temps bien déterminé puis de s'éteindre. Dans notre cas, afin d'avoir un exemple plus riche, nous avons compléter le mécanisme du feu tricolore en ajoutant le cas où nous voulons changer la couleur du feu manuellement avec des boutons respectivement, c'est-à-dire nous donnons la main à l'utilisateur de changer la couleur du feu qu'il souhaite allumer.

Le feu tricolore dans notre cas est défini comme suit :

— Une LED verte : allumée pendant 3 secondes.
— Puis une LED orange : allumée pendant 1 seconde .
— Enfin, une LED rouge : allumée pendant 3 secondes.
— Et des boutons qui permettent de changer la couleur des différent LEDs.

Dans ce qui suit, nous allons entamer la partie expérimentation de cette étude de cas de feu tricolore, en utilisant un kit Arduino et notre approche développée pour la génération du code Arduino.

3.7 Expérimentation

Le but est de réaliser une maquette feu tricolore avec trois 3 LEDs (rouge, orange, vert) et 3 boutons poussoirs.

Plus concrètement, nous allons d'abord établir graphiquement le modèle de comportement de feu tricolore et réaliser son diagramme de machine à état en utilisant l'outil Papyrus comme illustre la figure 3.11 ci-dessous .

Ce diagramme d'état transition montre la conception globale de ce système. Il représente les composants d'Arduino comme les boutons et les LEDs comme des états composites physiques. Il indique le numéro de port qu'il à travers lequel un composant sera connecté avec la carte Arduino.

Ce diagramme illustre également le fonctionnement du système et ses différents états. Dans un premier temps, le système est initialisé par la couleur Rouge. La figure montre aussi les différentes transitions locales entre les états composites physiques et les différentes transitions extérieures.

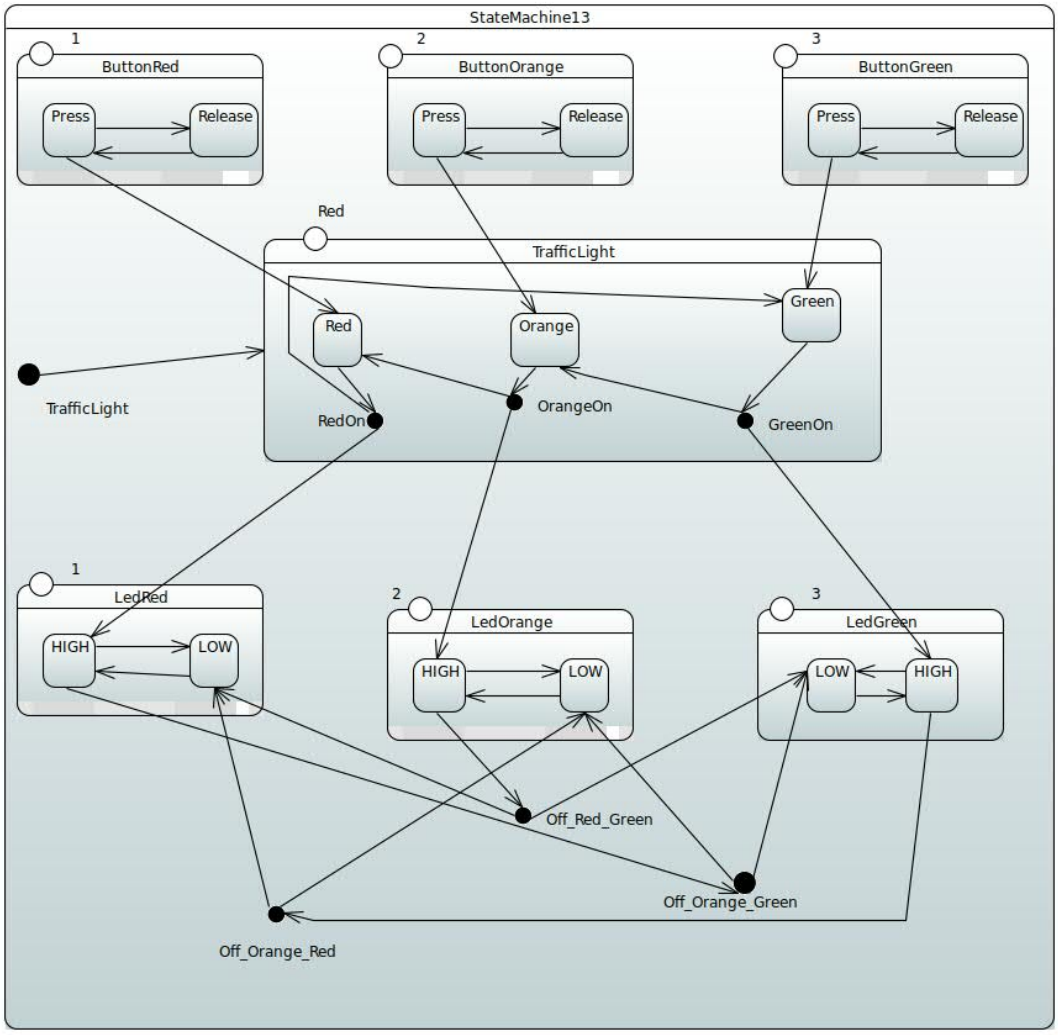

FIGURE 3.11 – Diagramme d'état transition du feu tricolore

Á titre d'exemple, lorsque l'on appuie sur le "ButtonGreen", ce bouton va passer de l'état (Release) à l'état (Presss), et il va donc interrompre le cycle automatique du feu tricolore, ce qui permet à ce dernier de prendre comme état courant l'état "Green". À ce niveau, le point de jonction à deux sorties : la première transition va envoyer un événement à la "LedGreen", qu'elle a à son tour passé de l'état (LOW) à l'état (HIGH) et envoyé un évènement par une transition pour que la led "LedRed" va s'éteindre et passer de l'état (HIGH) à l'état (LOW). la seconde transition change l'état courant de feu tricolore après 3s automatiquement par l'état "Orange". Ce scénario va se répéter par les autres composants en suivant les états appropriés.

Au moment de la création du diagramme d'état transition avec l'outil Papyrus, le métamodèle de ce diagramme est définit automatiquement à partir du modèle.

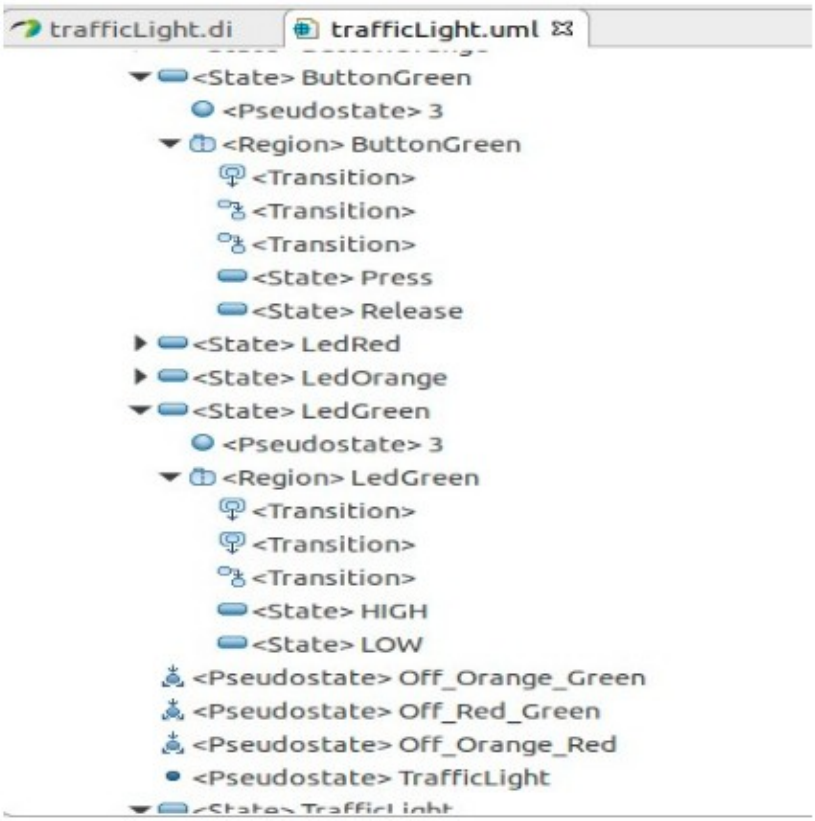

FIGURE 3.12 – Extrais du métamodèle généré du feu tricolore

La figure 3.12 ci-dessus représente un extrait du métamodèle qui se base sur les éléments déjà cités dans le scénario précédent.

À ce stade, toutes les données nécessaires pour générer le code Arduino son disponibles, en passant d'abord par la génération du code Thingml qui est dans notre cas le lien entre le modèle UML et le code Arduino.

Le code source ci-dessous représente un extrait du fichier ThingMl du feu tricolore généré par la template Acceleo .

```
1  import "composants/BUTTON.thingml"
2  import "composants/LED.thingml"
3
4  thing TrafficLight includes ButtonMsgs, TimerMsgs ,LEDMsgs
5  {   required port bUTTONGREEN {receives press,release}
6      required port lEDGREEN  {sends led_ON, led_OFF}
7       ....
8      statechart TrafficLightImpl init Running {
9      ...
10     state OFF_RED_GREEN
11          { transition -> LEDGREENLOW
12          transition -> LEDREDLOW  }
```

```
13      ...
14      state GREENON
15           { transition -> TRAFFICLIGHTORANGE
16             transition -> LEDGREENHIGH }
17      ...
18      state BUTTONGREEN
19           {  transition -> TRAFFICLIGHTGREEN
20              internal event bUTTONGREEN?press}
21       ...
22       state LEDREDLOW {
23       on entry do lEDRED!led_OFF()  end
24       }
25       ...
26       state LEDGREENHIGH
27           {on entry do lEDGREEN!led_ON() end
28            transition -> OFF_ORANGE_RED
29            transition -> LEDGREENLOW }
30      ...
31      state TRAFFICLIGHTGREENON
32           { transition -> TRAFFICLIGHTORANGE
33          transition -> LEDGREENHIGH}
34            }}
```

Les lignes 1 et 2 : l'importation des fichiers nécessaires pour les composants utilisés.

La ligne 4 : la création de notre thing " thing TrafficLight" et aussi il inclura les things des objets existants dans le diagramme.

Les lignes 5 et 6 : la déclaration des composants existants avec leurs valeurs, la linge 5 précise que le composant bouton reçoit 2 valeurs soit "press" ou "release", et la ligne 6 montre que le composants led envoie 2 valeurs "off" et "on" et qui sont à leur tour les différents états du composant.

La ligne 10 : la définition du passage de l'état vert à d'autres états.

La ligne 18 : montre qu'une fois le button reçoit l'évènement "presse" passe à travers la transition "TRAFFICLIGHTGREEN".

La ligne 26 : la définition de l'action une fois on rentre dabs l'etat (on entry) la ledGreen allume (lEDGREEN!led$_{O}N()$).

Le fichier Thingml généré contient aussi des configurations comme montre le code source ci-dessous.

```
1  ...
2  configuration TRAFFICLIGHT
3  {    ...
```

```
4       // Button device configuration.
5       instance button_BUTTONGREEN : Button
6           set button_BUTTONGREEN.PIN = 3
7       connector button_BUTTONGREEN.clock over Timer
8       ...
9       // Led device configuration.
10      instance led_LEDGREEN : LED
11          set led_LEDGREEN.PIN = 6
12      ...
13    // Application configuration.
14       instance app_TRAFFICLIGHT : TrafficLight
15       ...
16       connector app_TRAFFICLIGHT.bUTTONGREEN => button_BUTTONGREEN.
       evt1
17       ...
18       connector app_TRAFFICLIGHT.lEDGREEN => led_LEDGREEN.ctrl1
19       ...
20       connector app_TRAFFICLIGHT.timer over Timer
21
22  }
```

Les lignes 5 jusqu'au 11 représentent les différentes configurations pour chaque composant.

La ligne 14 représente l'instanciation du thing TrafficLight.

Par conséquent, grâce au fichier Thingml produit, on peut maintenant généré le code Arduino du feu tricolore on utilisant les fonctionnalités intégrées dans Thingml comme montre le code source suivant.

```
1  /**************************
2   * Headers for type : Button
3   ****************************/
4
5  // Definition of the instance struct:
6  struct Button_Instance {
7
8  // Instances of different sessions
9  bool active;
10 // Variables for the ID of the ports of the instance
11 uint16_t id_clock;
12 uint16_t id_evt1;
13 // Variables for the current instance state
14 int Button_Button_State;
15 // Variables for the properties of the instance
16 uint8_t Button_PIN_var;};
```

```
17 // Definition of the states:
18 #define BUTTON_BUTTON_STATE 0
19 #define BUTTON_BUTTON_RELEASED_STATE 1
20 #define BUTTON_BUTTON_PRESSED_STATE 2
21 ...
22
23 /*********************************
24  * Headers for type : TrafficLight
25  *********************************/
26
27 // Definition of the instance struct:
28 struct TrafficLight_Instance {
29
30 // Instances of different sessions
31 bool active;
32 // Variables for the ID of the ports of the instance
33 uint16_t id_bUTTONRED;
34 uint16_t id_timer;
35 ...
36 // Variables for the current instance state
37 int TrafficLight_TrafficLightImpl_State;
38 // Variables for the properties of the instance
39 char * TrafficLight_TRAFFICLIGHT_var;};
40 ...
41 /*********************************
42  * Implementation for type : TrafficLight
43  *********************************/
44 ...
45 // On Entry Actions:
46 void TrafficLight_TrafficLightImpl_OnEntry(int state, struct
       TrafficLight_Instance *_instance) {
47 switch(state) {
48 case TRAFFICLIGHT_TRAFFICLIGHTIMPL_STATE:{
49 _instance->TrafficLight_TrafficLightImpl_State =
       TRAFFICLIGHT_TRAFFICLIGHTIMPL_RUNNING_STATE;
50 TrafficLight_TrafficLightImpl_OnEntry(_instance->
       TrafficLight_TrafficLightImpl_State, _instance);
51 break;
52 }
53 ...
54 void setup() {
55 initialize_configuration_TRAFFICLIGHT();
56
57 }
58
59 void loop() {
```

```
60
61  // Network Listener
62  timer2_read();
63  // End Network Listener
64
65  TrafficLight_handle_empty_event(&app_TRAFFICLIGHT_var);
66
67      processMessageQueue();
68  }
```

Pour réaliser le montage de feu tricolore sur la carte Arduino nous avons besoin de matériel suivant :

Le matériel nécessaire

— Une carte Arduino et son câble usb.
— Des fils et une breadboard (plaque de prototypage).
— 3 Led (1 rouge, 1 orange et 1 verte)
— 3 boutons ainsi que 6 résistances de 220 ohms .

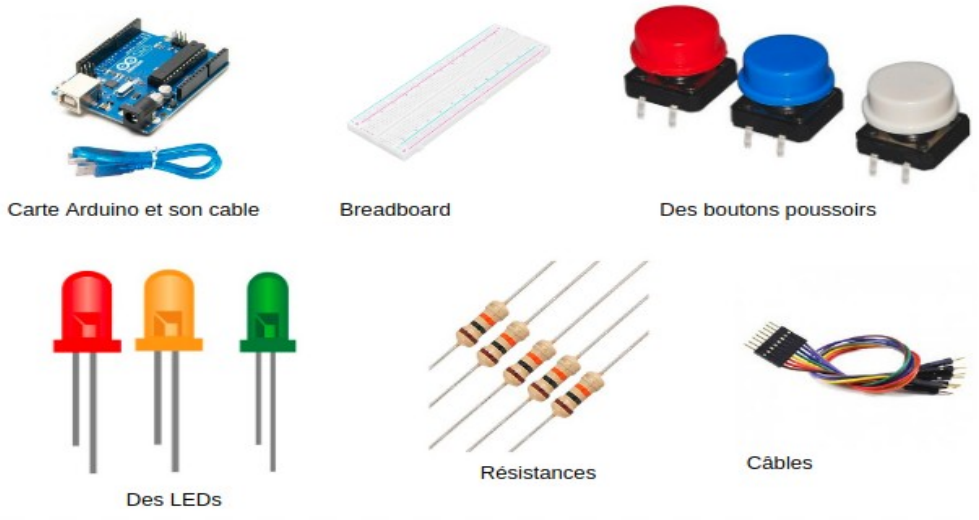

FIGURE 3.13 – Le matériel nécessaire pour réaliser le feu tricolore avec Arduino

On lance le code Arduino généré dans l'IDE Arduino, puis lorsque le programme est téléversé on aura le résultat dans la carte Arduino comme montre la figure 3.14 ci-dessous.

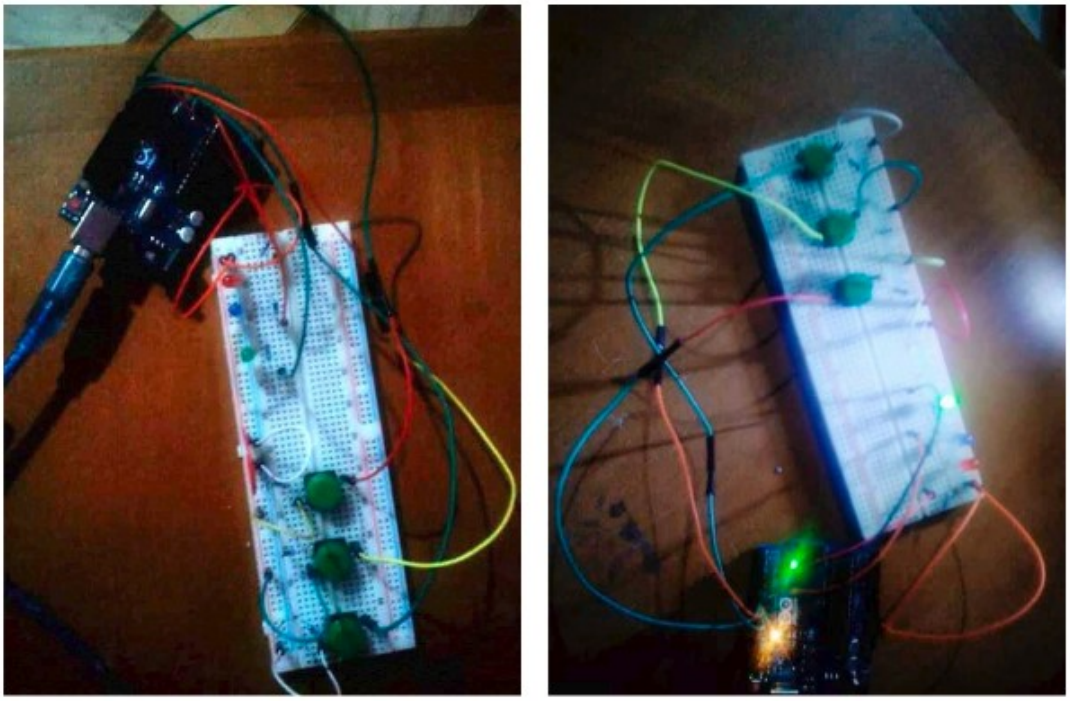

FIGURE 3.14 – Le résultat de programme du feu tricolore sur la carte Arduino

3.8 Conclusion

Dans ce chapitre nous avons présenté les notions de base de notre approche de réalisation du prototype de génération du code. Nous avons précisé les nouveaux concepts définis dans notre diagramme de machine à états, et nous avons validé le travail par une expérimentation sur une étude de cas de "Feu tricolore".

Conclusion générale

Ce mémoire de fin d'études présente la réalisation d'un générateur de code Arduino, outil de développement de logiciel, à partir d'un diagramme d'états-transitions en partant de de sa conception jusqu'à sa mise en oeuvre.

La modélisation objet est l'une des bases du développement logiciel. Le diagramme de machine à états est essentiel à la description du comportement d'un système qui peut donc être raffiné puis traduit par la suite en un code exécutable. C'est pourquoi nous avons opté pour l'utilisation du modèle d'états dans le générateur de code cible.

L'utilisation du ThingML dans la proposition de notre génération de code comme une modèle pivot nous a permis de simplifier notre approche en utilisant de nombreux outils qui lui sont associés.

Il est, toutefois, intéressant de noter que approche ne vise pas à remplacer la programmation ou à masquer le code source, mais plutôt à aider les développeurs à produire rapidement un code source et de qualité meilleure.

Nous avons également veiller à ce que certaines bonnes pratiques dans le développement des logiciels soient respectées. Le générateur de code n'est donc pas uniquement un soulagement face à des tâches extrêmement laborieuses de développement, mais surtout une solution flexible et évolutive qui pourrait s'adapter à de nouveaux besoins en terme d'organisation et techniques de développement.

Perspectives

Les perspectives de ce travail de recherche sont multiples. Nous en citons quelques une ci-dessous :

— Enrichir le langage source et couvrir plus de composants Arduino.
— Générer le code à partir d'autres diagrammes UML combinés (diagrammes d'activité, diagrammes de séquences *etc.*).
— Améliorer l'implémentation du générateur et générer le code pour Raspberry Pi également.
— Étudier le compromis entre réduction temps d'exécution et la réduction de la taille du code généré.

Bibliographie

[1] Source:IDATE2013:https://en.idate.org/.

[2] Jamda:siteweb:http://jamda.sourceforge.net..

[3] Projet ACCORD. « Assemblage de composants par contrats en environnement ouvert et réparti ». In : (2002).

[4] BOUCHERIT AHMED et MERAIMI MOULOUD. « Etude et réalisation d'une carte de contrôle par Arduino via le système Android ». In : (2018).

[5] Jean-Christophe BACH. « Une approche hybride GPL-DSL pour transformer des modeles ». In : (2013).

[6] Felice BALARIN et al. *Hardware-software co-design of embedded systems : the POLIS approach*. 1997.

[7] Massimo BANZI. « Getting Started with arduino : Make ». In : (2009).

[8] Jean BÉZIVIN. « On the unification power of models ». In : *Software & Systems Modeling* (2005).

[9] Jean BÉZIVIN et Olivier GERBÉ. « Towards a precise definition of the OMG/MDA framework ». In : *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. 2001.

[10] Michael BLAHA et James RUMBAUGH. *Modélisation et conception orientées objet avec UML 2*. Pearson Education, 2005.

[11] Conrad BOCK. « Three kinds of behavior models ». In : *Journal of Object-Oriented Programming* (1999).

[12] Meriem BOUMASSATA. « Vérification de code pour plates-formes embarqués ». Thèse de doct. Université de Batna 2, 2012.

[13] Daniel CALEGARI et Nora SZASZ. « Verification of model transformations : A survey of the state-of-the-art ». In : *Electronic notes in theoretical computer science* (2013).

[14] Instituts CARNOT. « Le livre blanc-objets communicants et internet des objets ». In : *Rapport technique, June* (2011).

[15] Asma CHARFI SMAOUI. « Compilation optimisée des modèles UML ». Thèse de doct. Paris 11, 2011.

[16] Benoit CHARROUX, Aomar OSMANI et Yann THIERRY-MIEG. *UML 2, Synthèses de cours et exercices corrigés*. 2005.

[17] Pierre DAVID. « Contribution à l'analyse de sûreté de fonctionnement des systèmes complexes en phase de conception : application à l'évaluation des missions d'un réseau de capteurs de présence humaine ». Thèse de doct. 2009.

[18] Anne ETIEN, Cedric DUMOULIN et Emmanuel RENAUX. « Towards a unified notation to represent model transformation ». Thèse de doct. INRIA, 2006.

[19] Patrice FLICHY. *L'imaginaire d'Internet*. 2012.

[20] Richard C FORCIER et Don E DESCY. *The computer as an educational tool : Productivity and problem solving*. Prentice-Hall, Inc., 2007.

[21] Lovic GAUTHIER. « Génération de système d'exploitation pour le ciblage de logiciel multitâche sur des architectures multiprocesseurs hétérogènes dans le cadre des systèmes embarqués spécifiques. » Thèse de doct. Institut National Polytechnique de Grenoble-INPG, 2001.

[22] Alicia M GIBB. « New media art, design, and the Arduino microcontroller : A malleable tool ». In : *Pratt Institute* (2010).

[23] Jens GRABOWSKI et Steffen HERBOLD. *System Analysis and Modeling. Technology-Specific Aspects of Models : 9th International Conference, SAM 2016, Saint-Melo, France, October 3-4, 2016. Proceedings*. 2016.

[24] David HAREL. « Statecharts : A visual formalism for complex systems ». In : *Science of computer programming* (1987).

[25] David HAREL et Amnon NAAMAD. « The STATEMATE semantics of statecharts ». In : *ACM Transactions on Software Engineering and Methodology (TOSEM)* (1996).

[26] Jack HERRINGTON. *Code generation in action*. Manning Publications Co., 2003.

[27] Manfred A JEUSFELD, Matthias JARKE et John MYLOPOULOS. *Metamodeling for method engineering*. 2009.

[28] P KADIONIK. « Cours de l'option Systèmes Embarqués ». In : *Ecole Normale Supérieure d'Electronique, Informatique et Radiocommunications de Bordeaux–ENSEIRB* (2004).

[29] Anneke G KLEPPE et al. *MDA explained : the model driven architecture : practice and promise*. Addison-Wesley Professional, 2003.

[30] Mohamed KOUDIL. « Cours intitulé-Méthode de conception conjointe des systèmes embarqués ». In : *Institut national de formation et informatique* (2004).

[31] Daniel KROB. « Eléments d'architecture des systèmes complexes ». In : *Gestion de la complexité et de l'information dans les grands systèmes critiques* (2009).

[32] Craig LARMAN. *UML et les Design Patterns*. 2005.

[33] David LUGATO et al. « Validation and automatic test generation on UML models : the AGATHA approach ». In : *International Journal on Software Tools for Technology Transfer* (2004).

[34] Anthony MACDONALD, Danny RUSSELL et Brenton ATCHISON. « Model-driven development within a legacy system : an industry experience report ». In : 2005.

[35] Rafael MARQUES. « Système de produits et services basés sur l'internet des objets : conception et implantation pilote dans une station-service ». Thèse de doct. 2018.

[36] Jean-Pierre MEINADIER. *Ingénierie et intégration des systèmes*. 1998.

[37] OMG MOF. *2.0 query/views/transformations rfp*. 2002.

[38] Chokri MRAIDHA, Sara TUCCI-PIERGIOVANNI et Sebastien GERARD. « Optimum : a marte-based methodology for schedulability analysis at early design stages ». In : *ACM SIGSOFT Software Engineering Notes* (2011).

[39] Alex MUCCHIELLI. « L'émergence du sens des situations à travers les systèmes humains d'interactions ». In : *Revue internationale de psychosociologie* (2007).

[40] MECHERI NACERA. « UNE APPROCHE HYBRIDE POUR TRANSFORMER LES MODELES ». In : ().

[41] OBJECT MANAGEMENT GROUP (OMG). *Meta-Object Facility (MOF) Specification, Version 2.0*. OMG Document Number formal/2006-01-01 (http://www.omg.org/spec/MOF/2.0). 2006.

[42] QVT OMG. « Meta object facility (mof) 2.0 query/view/transformation specification ». In : *Final Adopted Specification (November 2005)* (2008).

[43] Alain PLANTEC. *Terminologie documentée de l'Ingénierie Dirigée par le Modèles*. Rapp. tech. Technical Report, Université de Bretagne Occidentale, 2013.

[44] Abderrahmen RAFAI et Sofia KOUAH. « Développement d'un système d'IoT (Internet of Things) pour le smart lighting sous la plateforme IBM ». In : (2018).

[45] Mark RICHTERS et Martin GOGOLLA. « Validating UML models and OCL constraints ». In : 2000.

[46] Frédéric ROUSSEAU. « Conception des systèmes logiciel/matériel : du partitionnement logiciel/matériel au prototypage sur plateformes reconfigurables ». Thèse de doct. Université Joseph-Fourier-Grenoble I, 2005.

[47] Hillary SILLITTO et al. « Defining "system" : A comprehensive approach ». In : 2017.

[48] Omar TALEB et Abdelkrim MANKOURI. « Programmation de la sécurité Internet des Objet, Etude de cas module WIFI Electric imp ». Thèse de doct.

[49] « Thingml : un framework de génération de langage et de code pour cibles hétérogènes ». In : *Actes de la 19e conférence internationale ACM / IEEE sur les langages et systèmes d'ingénierie dirigée par les modèles*.

[50] Skander TURKI. « Ingénierie système guidée par les modèles : Application du standard IEEE 15288, de l'architecture MDA et du langage SysML à la conception des systèmes mécatroniques ». Thèse de doct. 2008.

[51] BUI VAN HON et Hugues BERSINI. « Génération de code à partir d'un diagramme d'états-transitions ». Thèse de doct.

[52] Ludwig VON BERTALANFFY et al. *Théorie générale des systèmes*. Dunod Paris, 1973.

[53] Ferdinand WAGNER et al. *Modeling software with finite state machines : a practical approach*. 2006.

[54] Murray WOODSIDE et al. « Performance analysis of security aspects by weaving scenarios extracted from UML models ». In : *Journal of Systems and Software* (2009).

[55] Tewfik ZIADI. « Manipulation de Lignes de Produits en UML ». Thèse de doct. Rennes 1, 2004.