Complex Systems Informatics and Modeling Quarterly (CSIMQ)
eISSN: 2255-9922
Published online by RTU Press, https://csimq-journals.rtu.lv
Article 209, Issue 38, March/April 2024, Pages 54–99
https://doi.org/10.7250/csimq.2024-38.03

Assistance in Model Driven Development: Toward an Automated Transformation Design Process

Pascal André[1⋆] and Mohammed El Amin Tebib[2]

[1]LS2N CNRS UMR 6004 - University of Nantes, 2 rue de la Houssinière F-44322 Nantes, France
[2]LCIS lab, Grenoble INP, 50 Rue Barthélémy de Laffemas, 26000 Valence, France

pascal.andre@ls2n.fr,
mohammed-el-amin.tebib@univ-grenoble-alpes.fr

Abstract Model driven engineering aims to shorten the development cycle by focusing on abstractions and partially automating code generation. We long lived in the myth of automatic Model Driven Development (MDD) with promising approaches, techniques, and tools. Describing models should be a main concern in software development as well as model verification and model transformation to get running applications from high level models. We revisit the subject of MDD through the prism of experimentation and open mindness. In this article, we explore assistance for the stepwise transition from the model to the code to reduce the time between the analysis model and implementation. The current state of practice requires methods and tools. We provide a general process and detailed transformation specifications where reverse-engineering may play its part. We advocate a model transformation approach in which transformations remain simple, the complexity lies in the process of transformation that is adaptable and configurable. We demonstrate the usefulness, and scalability of our proposed MDD process by conducting experiments. We conduct experiments within a simple case study in software automation systems. It is both representative and scalable. The models are written in UML; the transformations are implemented mainly using ATL, and the programs are deployed on Android and Lego EV3. Last we report the lessons learned from experimentation for future community work.

Keywords: Model Driven Software Engineering, Refinement, Model Transformation Process, Automation Control Systems.

1 Introduction

Since the advent of Model Driven Architectures (MDA), we have long lived the myth of automatic Model Driven Development (MDD) with promising maintenance cost reduction. We are convinced that Model Driven Engineering (MDE) is gainful in developing long-term software systems. We do share the vision given in [1] that reduces MDD to two pillars: raising the level of abstraction

⋆ Corresponding author

Reference: P. André and M. El Amin Tebib, "Assistance in Model Driven Development: Toward an Automated Transformation Design Process," Complex Systems Informatics and Modeling Quarterly, CSIMQ, no. 38, pp. 54-99, 2024. Available: https://doi.org/10.7250/csimq.2024-38.03

Additional information. Author's ORCID iD: P. André – https://orcid.org/0000-0002-1112-2973. PII S225599222400209X. Article received: 16 September 2023. Accepted: 22 February 2024. Available online: 30 April 2024.

and raising the degree of computer automation. MDD shortens the development cycle by focusing on abstractions and partially automating code generation: describe abstract models, verify and transform them to get running applications. MDD should help not only to improve the resulting software quality and reduce the development time but also to make it evolve continuously fit to new needs and constraints. MDD helps to face the challenge of software maintenance costs, which traditionally accounted for 70% of the total cost of the software, still increase [2], [3]. Twenty years later, despite numerous contributions to techniques and tools, MDD has not yet reached the promises [4] for the software development practice and therefore further work is necessary.

This article extends [5] and [6] by providing deeper knowledge on the state of practice, specification details for practitioners, and application principles for automation and assistance. In all these aspects, we take an MDD **practitioner**'s point of view. The practitioner's problem is how to efficiently develop and maintain applications from abstract models (including structural, dynamic, and functional aspects of the modeled system). Code generation from operational models has existed for many years by means of compilers or grammar-based generators such as Antlr, XML, and JSON parsers or recently Xtext. However, the generation of code from higher level models of abstraction, resulting from the analysis or software design, is still a prerogative of the developers. Automation becomes more cost-effective than manual development when considering the evolutionary maintenance of functional, non-functional, and technical requirements (hardware modification, for instance).

Our goal is to assist developers in terms of guidelines and automation to make MDD more effective in terms of model transformations. We focus more on methodological issues than on purely technical ones; it is empirical and use trial and error methods to find suitable solutions. This work is a practical contribution to the *engineering practice* challenges [7], such as managing automation and heterogeneity, and locates in the social and domain challenges [8] such as technical abstraction, and co-engineering. To conduct the study we proceeded in the following three steps:

1. An assessment study to evaluate the maturity of both the tool support and the development methodologies in assisting developers in an MDD process (forward engineering),
2. A proposition to structure the MDD process in terms of four macro model transformations that guide the activities of the software engineers,
3. An approach based on model driven reverse engineering (MDRE) to find abstractions in the technical platform to feed those macro-transformations.

For both the assessment part and the proposal part, we conduct trial and error experiments, i.e., this is not a systematic study but lessons from experience. This results in an assisted MDD process with the following features: (i) an original combination of stepwise forward engineering with model driven reverse engineering of technical frameworks, (ii) a comprehensive 4-step MDD transformation process progressively incorporating design concerns and decisions, (iii) a detailed specification for systematic transformations at each step, and (iv) implemented transformations, written using existing tools, to help automate the process.

To lead our experiments, we define a small but representative case study that can be reused for benchmarking related contributions. We assume the entry models (also called logical models here) are written using general-purpose modeling languages such as Unified Modeling Language (UML) [9] or SysML [10] instead of Domain Specific Languages (DSL). Of course, DSLs bring better results in terms of executable transformation but their scope is very limited. In addition to a broader scope, general-purpose modeling languages are also part of software developers' background and can be used in software engineering teaching. In this case study we target programmable controllers (e.g., Lego EV3) remotely controlled by an Android client. We implement and compare three ways to obtain the source code: manual forward development, automatic code generation, and stepwise model transformation which can be seen as a compromise approach in terms of automation and genericity. The lessons learned from these early experimental works will open tracks for future work.

The rest of the article is structured as follows. Section 2 introduces the context elements of model driven concepts. We assess the current practice of model driven development in Section 3, first, by evaluating the current means of automatic code generation in Section 3.1, and, second, by observing the practice of Model Based Engineering (MBE) during experimentation in Section 3.2. This assessment shows the need for a true MDD process. We then propose a stepwise MDD process in Section 4, defined as a transformation process built on macro-transformations where the complexity yields in the process which must be adaptable and configurable. Implementation details of the macro-transformations are given in Section 5 and illustrated by experimentation on the case study. A stepwise transformation process is not sufficient because we need to feed the transformation with technical information on the target platform. To fill the missing technical model, we explore reverse engineering in Section 6 which should also be raised step by step in abstractions. We discuss the lessons learned during this study with related works in Section 7. Finally, Section 8 summarises the contribution and draws open perspectives that go beyond model transformation topics.

2 Background

Let us clear the MD-acronyms jungle by referring to the discussions of Selic [1] and Cabot [11]. MBE is the historical branch of using models in software development. All the software development methods use models for high level reasoning (e.g., SADT, SART, Merise, OMT, RUP, etc.)[3]. MDD focuses on the generation of implementations from models [12], especially for software development, as well as Model Driven Software Engineering (MDSE) [11]. Model transformation (MT) is a convenient frame to handle stepwise generations and implement MDD [13]. In the context of MDA of the Object Management Group (OMG) [14], forward engineering can be seen as a *transformation process* from a *Computation Independent Model (CIM)* to a *Platform Independent Model (PIM)* and then to (more) concrete *Platform Specific Model (PSM)* injecting elements of the *Platform Description Model (PDM)*. MDE is a broader field than MDD: it includes model evolution, MDRE, and, recently, also Model Language Engineering (MLE). In this article, we mainly focus on MDD and the implementation of PIM to PSM.

The goal of MDD is to set up a software production chain based on expressive models for distributed systems. In this article, we illustrate the discourse by automation systems because they include various concerns on distribution, communication, and control, which cover many paradigms of the UML language, but also on non-functional requirements.

Typically, software development follows a forward engineering process that starts from requirements analysis to execution code deployment. The Rational Unified Process [16] is a reference process for MBE with UML. But in this article, we chose the 2 Track Unified process (2TUP) (also called "Y") [15] because it puts in evidence the role of design as the central activity that combines an analysis model with technical support as depicted in Figure 1. Having reuse in mind, the "Y" process shows that (i) the technical debt can be solved by changing the right part of the "Y" and (ii) the software supplier can apply a technical background to different projects of different customers. Also, the "Y" process makes sense from a MDD point of view when seeing the

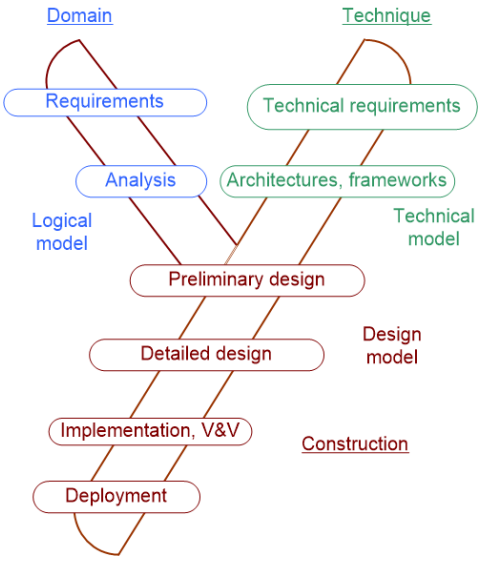

Figure 1. 2TUP Unified Process [15]

[3] A development methodology includes paradigms, notations, processes, and tools. Agile methods such as SCRUM are not mentioned here because they focus on the process only

development as a (complex) model transformation. The development cycle is implicitly iterative and incremental, and the "Y" process is applied for each iteration.

From the software point of view we consider at least two levels:

1. The model level (the blue part in Figure 1), is where the individual and collective behaviors are described and where the constraints are analyzed. Specifications are written in general purpose modeling languages such as UML [9], SysML [17], or AADL [18] using their associated tool support. The models at the analysis stage will be called *logical models* or PIM in the MDA spirit (see above) because the technical details are not yet given. As illustrated by Figure 1, the analysis model is plunged into a technical model to build a design model.
2. The operational level (the red part in Figure 1), is where the data, functions, and controls of the physical devices are implemented by software programs.

To reach the implementation from high-level models, one can use intermediate models [11]. In MDD, it is essential to ensure the model's correctness before starting the process of transformations and code generation; this reduces the high cost of late detection of errors [19], [20]. Whatever the modeling language, the models should be sufficiently detailed to be made executable[4]. This allows to verify properties on the models, using *theorem proving*, *model checking*, or *model testing* techniques [21]. In the next section, we assess the current means to achieve MDD.

3 Assessment of Current Practice of MDD

In this section, we evaluate the means to implement MDD and show their limits. Basically, there are the following three, classified by the growing *degree of automation*, ways to refine models to code:

1. **Manual forward engineering** is the traditional way to develop software. In MBE software engineers start from high level models and design software solutions.
2. **Stepwise refinement by a MDD transformations process.** The process is a sequence of model transformations that ends with software. To the best of our knowledge, there are no proposals of MDD process as a stepwise process of model transformations.
3. **Fully automated code generation.** The model is compiled in an executable source code. There are two main use cases for code generation: (i) generate parts of the software application, and (ii) simulate the model in a model validation process (also called animation). In the latter case, the model is qualified as *operational* as it can be executed.

We will evaluate the code generation in Section 3.1 and experiment manual forward engineering in Section 3.2. The stepwise approach, being a contribution, we will explore in Section 4.

3.1 Code Generation

We report here our investigations on automated code generation, categorized into two groups: UML tools generating source code (I) and those producing code for compilation and execution (II).

UML Case Tools Generating Source Code (I). In the last years, we have noticed that there is prominent progress in modeling tools to support model driven engineering processes: (i) code generation, (ii) reverse engineering, (iii) modeling of real time and event-driven complex systems, and (iv) dealing with the significant features of embedded systems such as object distribution, synchronous and asynchronous communication, become a key tool advantage for the industry. Our study tries to find prototype tools for the following representative categories: free (F), community

[4] Model transformations become relevant if the models contain enough information.

edition (CE), open-source (O), and commercial (C). The tools are mainly selected on their availability and coverage. The list of considered tools is given in Appendix A.

We compared the code generation facilities of some UML related tools according to the selected features. These features correspond to facilities we need in model engineering (interoperability, diagrams, model-code coupling) and technical support (communications and middleware), and code generation (programming languages, state-machines, API mapping).

- **Model Interoperability (MI).** The interchange formats UML-XMI and MOF-XMI are key standards for writing model transformations. The XMI header provides the UML version and may influence the input model compatibility for the transformation.
- **Diagrams.** We mainly focus on UML Class Diagrams (CD) and State-Transition Diagrams (STD). UML editors cover the main UML notation but differ one from another on specific notations, notation semantics, and consistency links between the diagrams. Operation body is rarely defined in logical models despite we can use Activity Diagrams (AD), OCL assertions, or actions from the UML Actions Semantics (AS). However, this would enable accurate transformation rules. Some tools consider the semantic link between classes from CD and STD in their code generation while others consider them separately.
- **Model Synchronisation (MS) or Round-trip (RT).** It keeps a strong link between a model element and a source code. This allows to replay code generation after a model evolution without losing the updates made at the code level. RT is also a way to define the body of an operation, i.e., a concrete semantics that is convenient in practice but prevents accurate verification and transformation at the model level.
- **Communicating State Machine (CSM).** In plain UML, STD are associated with classes as state machine protocols but STD can also be used separately to model system behaviours or a main control program. The code generation for CSM is often postponed to programming but it is a main feature for distributed applications.
- **Message Oriented Middleware (MOM).** UML assumes an implicit middleware for message sent (reliable order preserving medium) but deploying a distributed application requires to implement distant communications.
- **Target Programming Language (Target PL).** This entry indicates the target languages for the *behavioral part*, since we assume the static part to be provided by default.
- **Support for State Machines (SSM).** We consider here the None/Partial/Complete code generation for STD. We will mention the STD elements supported by the partial generator engine in further sections.
- **API Mapping (APIM).** At the low level, model elements are connected to predefined elements in libraries. In the case of a model element that exists, with a different shape, in a framework library, we call that API mapping. This point will be developed in Section 5.4. API mapping is different from round-trip facilities which annotate models with source code references.

Starting from these entries, the goal is to draw the strengths and weaknesses of some representative tools we have tested. The study is summarized in Table 1

Many tools mentioned in Table 1 are not bound to only one language, e.g., Modelio, Papyrus, or Visual Paradigm integrate different OMG standards such as SysML, BPMN, etc. Several tools support the full modeling of structural and dynamic behavior views of such a complex system. Support for model verification exists for static and type checking conformance but advanced support is required for full consistency, completeness, and dynamic correctness. However, code generation is not as developed as modeling tasks, especially for STD where limits are related to the code generation from some pseudo states like *join* and *fork* elements in Umple. The model *interoperability* is insured in most tools through XMI format, except Umple which has a textual representation and Yakindo which uses a specific XML format called SCXML. Some tools have specific or esoteric notations for some model elements.

Table 1. Comparison of some tools on their code generation facilities

	MI	Diagrams	MS	CSM	MOM	Target PL	SSM	APIM
Papyrus (O)	2.5	CD, STD	√	-	-	C++	Full	-
Modelio (O)	2.4.1	CD, STD	√	-	-	C, C++, Java, Python	Partial	-
Umple (O)	-	CD, STD	√	-	-	C++, Java, PHP	Partial	-
StarUML (F,C)	2.0	CD	-	-	-	JAVA, C++, C#	-	-
Visual Paradim (CE, C)	2.0	CD, STD	√	-	-	Java, C++, Python	Full	-
UModel (C)	2.4	CD, STD	√	-	-	Java, C#, VB	Full	-
Rhapsody (C)	2.4	CD, STD	√	-	-	C, C++, Java	Full	-
Yakindo (F,C)	-	STD	-	√	Event	C, C++, JAVA, Python	Full	-
FXU	2.2	CD, STD		-	-	C	Partial	-

Tool limits are related to the target programming languages, e.g., the generated code from STD in Papyrus cannot be deployed or integrated into other frameworks or protocols such as Android. Table 1 illustrates that, to our knowledge, no tool deals clearly with the problems of (heterogeneous) communications (MOM) and mapping to code libraries (APIM). Recently Yakindo 3.0 introduced features through the concept of multi state machines that share events; this can help in MOM.

Executable UML (II). The Executable UML school aims at running a system directly from its UML logical model specification. Executing a model is interesting mainly for simulation or animation purposes (the animation is a visual part of the simulation). It is hardly the final system because it would imply a fixed technical framework for all models, however, it may work for simple information systems where the functions are aligned with the data structure like CRUD (Create, Read, Update, Delete) application generation in relational databases. In any case, code generation is involved, but not necessarily a source code is available.

Executable UML is interesting when evaluating the quality of the UML logical model. Executable UML requires unique and operational semantics for each concept. This means that the UML Semantics weaknesses must be overcome. It is suggested to:

- Remove useless, complex or ambiguous UML paradigms to build a consistent profile. In particular, the semantics of the UML state diagrams must be limited because it is too expressive for a consistent interpretation[5]. Activity diagrams are usually ignored because they have a double interpretation of data flow and control flow semantics, which is hard to implement.
- Add missing paradigms. In particular, a semantics for *actions* in standard UML to define usual functional communications. OCL enables us to define assertions but not really computations [22].

A major result of the Executable UML school is their reflection and deliveries on action semantics. The *Action Semantics* is defined by a meta-language since UML 1.4. No standard concrete syntax was proposed. Early concrete syntaxes were associated with XUML tools, especially for the following real-time systems:

- *Action Specification Language (ASL)* was defined for iUMLLite of Kennedy-Carter (Abstract Solutions) supporting xUML [23].
- *BridgePoint Action Language (AL)* (and the derived SMALL, OAL, TALL) proposed by Balcer & Mellor was implemented in xtUML of Mentor Graphics [24].
- *Kabira Action Semantics (Kabira AS)* proposed by Kabira Technologies (and later TIBCO Business Studio).

[5] It is not surprising to find Stephen Mellor (SA-RT, OOAD, xtUML) among the Executable UML contributors.

- The normative telecom SDL [25] has also been used to provide semantics as a UML profile.
- Other proposals are *Platform Independent Action Language (PAL)* of Pathfinder Solutions, or SCRALL [26] which had a visual representation, +CAL [27].

All these efforts led to semantics for a subset of executable UMLs, called `fUML` (*Semantics of a Foundational Subset for Executable UML*) [28], with now a normalized concrete syntax `Alf`. A reference implementation exists[6]. In the tool list of the Modeling Language website[7] there is some confusion between fUML implementations, model execution, debugging, and code generation. In the case of Moka/Papyrus we could not run it or get inside version 2.0. The executable UML tools are not directly convenient to design heterogeneous distributed applications since the technical model is often a fixed option (e.g., the commercial IBM Rational is proposed for Websphere) not a parameter of the transformations.

As mentioned in [12], code generation does not drive MDD. In the next section, we study the practice of software engineers.

3.2 Forward Engineering Experimentations

Forward engineering, and especially software design, makes space for software developers skills and experience leading to unpredictable processes and results. To assess the manual way to refine models to code, we could not compare existing approaches because despite adoption in industry [11], successes [1], and case studies [12] we cannot access the material for confidential reasons. Since our work is not based on existing applications and company developers, we lead our experimentations with students on our case studies. The students are master students becoming future software engineers; they are not MDD experts but their curriculum includes software engineering and model driven engineering. They are more or less aware of methodologies such as the already mentioned RUP or 2TUP and best practices. The case studies were given to different groups of students from 2018 to 2020.

Case Study. The case studies must be simple enough to be handled during student projects and complex enough to be representative of software design concerns such as distribution, networking, concurrency, persistence, graphical user interface (GUI), etc. A simple Java program is not representative enough. For the work we report here, we chose a simple home automation equipment (domotic), a garage door. It is a simple cyber physical system (CPS) including hardware devices (remote control, door, programmable logic controllers (PLC), sensors, actuators, etc.) and the software that drives these devices[8]. In CPS, SysML [10] is recommended for PLC design, e.g., the detailed SysML model of transmission control for Lego NXT[9] has been simulated by the Cameo tool. However, we chose UML because it belongs to the student's program and because the UML modeling ecosystem is rich. Last but not least, despite the case study being a CPS our work is not limited to CPSs. The case study is described in Appendix B.

From Models to Implementation. Developers implement a solution that should conform to the given models and requirements specification. As illustrated in the "Y" process of Figure 1, software design is the key activity (join point of the "Y") that implements requirements (left branch of the "Y") inside a technical platform (right branch of the "Y"). It is an engineering activity where

[6] http://modeldriven.github.io/fUML-Reference-Implementation/

[7] https://modeling-languages.com/list-of-executable-uml-tools/

[8] A variant is given with an outdoor gate to access a home property. A third case is the Riley Rover (http://www.damienkee.com/rileyrover-ev3-classroom-robot-design/) driven by a remote Android application. An additional interest of these cases (https://ev3.univ-nantes.fr/en/) is that they can later be integrated as subsystems in larger applications.

[9] https://tinyurl.com/wkja25u

decisions have to be taken that affect the quality of the result. Key design concepts are abstraction, architecture, patterns, separation of concerns, and modularity. The result is a design model that covers complementary aspects such as persistence, concurrency, human interfaces, and deployment in an architectural vision that gradually reveals the details [29].

We assume a technical architecture made of Lego EV3 (java/Lejos) and a remote computer (smartphone, tablet, laptop) under Android as pictured by the deployment diagram of Figure 2. Available wireless protocols between EV3 and the remote are WiFi and Bluetooth. To establish the technical platform we need to select a technology in a library and to map model elements.

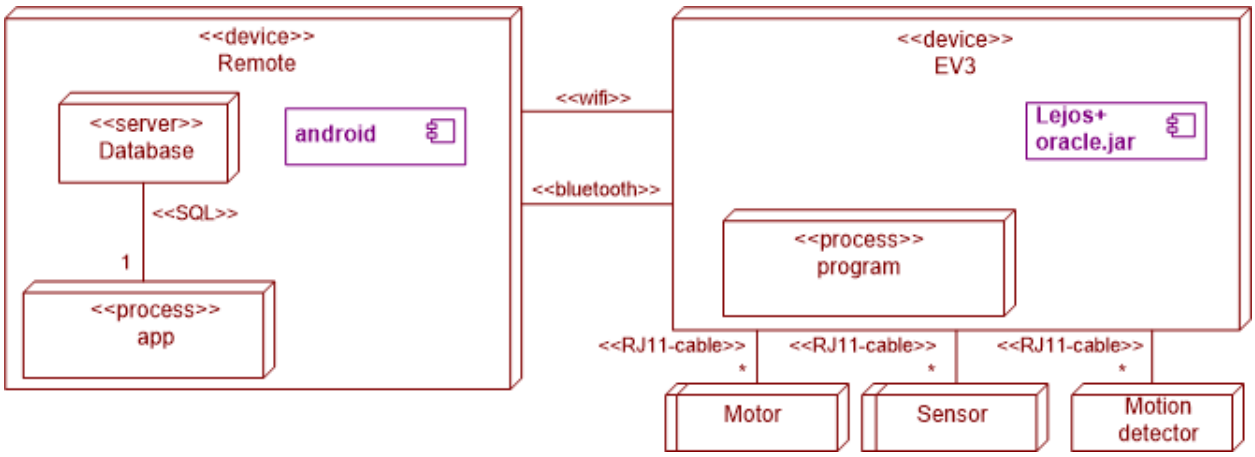

Figure 2. Technical Architecture – EV3 and app

Student Experimentation. The starting point is the case study, online documentation on EV3 Lejos, and articles such as [30], [31], [32], and [33]. All students' works have been implemented using Lego Mindstorm EV3 sets. As an example, one student project[10] led to the mock-up of Figure 3.

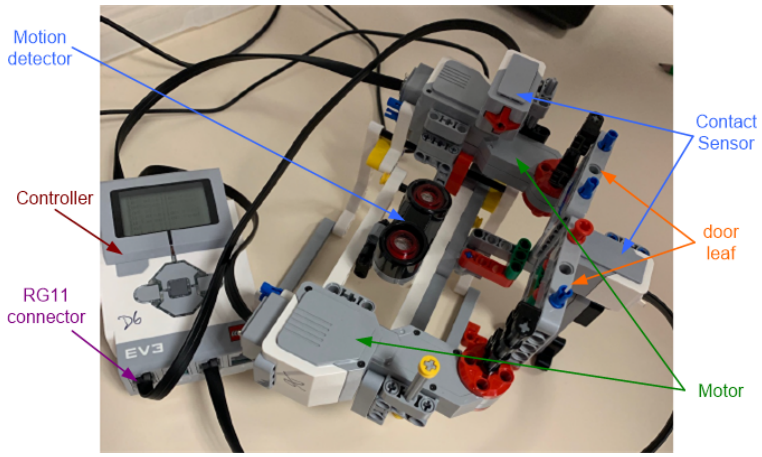

Figure 3. Lego prototype of the door system

The case was given to different groups of students who produced quite different implementations. In the case of the garage door, a basic version called v1[11] was proposed in 2018, that has been extended later until having an Android App to play the remote device with Bluetooth connection

[10] https://www.youtube.com/watch?v=7WBKTgRv7co
[11] https://github.com/demeph/TER-2017-2018

and led to an implementation with enumeration types for STDs. Another implementation, called v2[12] led to the class diagram of Figure 4.

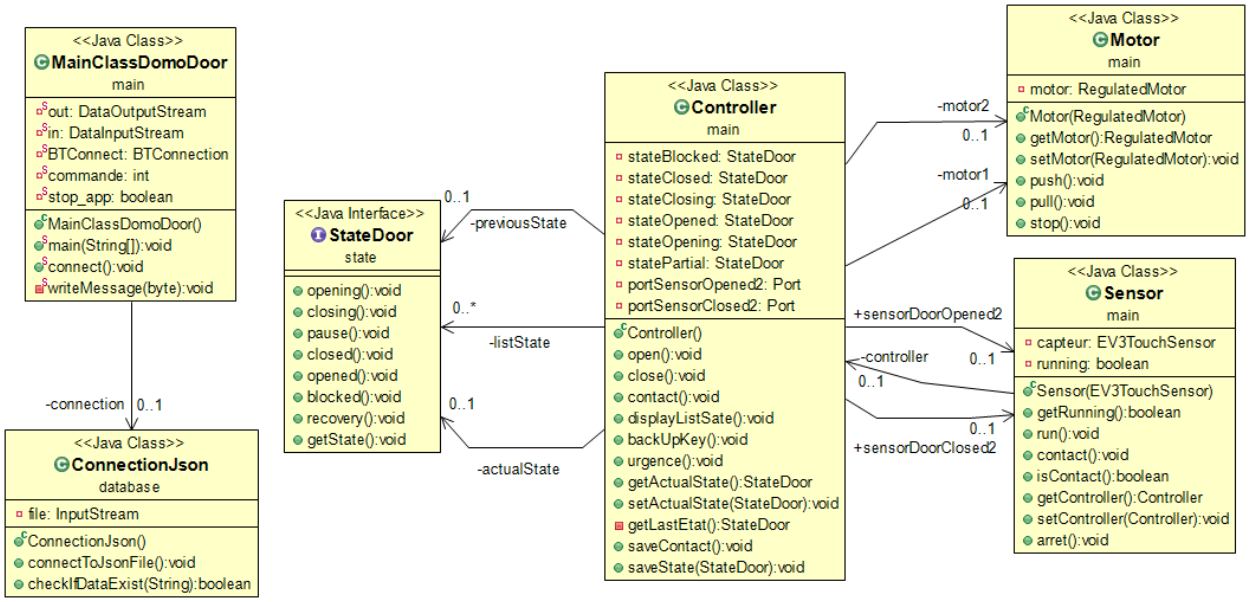

Figure 4. Class Diagram of the door application (v2)

We summarise in the following the lessons resulting from the analysis of the student deliveries.

- The code produced by students does not necessarily conform to the logical UML models. Indeed, the model serves to understand and interpret the case study, it is perceived as a documentation reference rather than an abstract model. The students do not aim at a strict conservation of semantics: they implement rather than refine. Moreover, their program includes functional and non-functional requirements (see Section 3.2) that were not mentioned in the given (simplified) logical model.
- The prototype of Figure 3 uses two motors for two door swings while a single door and motor were specified in the logical model.
- The preliminary design decisions are different. The remote device was also implemented in different ways according to the student experience: from Java Swing GUI with wired TCP-IP communication with EV3 or Android app with Bluetooth or Wifi connection. For instance, despite implementations v1 and v2 use of the Bluetooth protocol, the remote device (Android App) is not connected to the EV3 application in version v2 while it is in version v1. Using architectural patterns can help to improve the preliminary design product quality.
- The detailed design decisions are different. Design patterns can be introduced here to improve the quality of the design [34]. In version `v1` the students used enum types to implement state machines while a *state pattern* has been chosen in version `v2` of Figure 4.
- The technical background varies between the groups, some used the Java Lejos framework[13], others used ev3dev[14] which enables several programming languages. It depends on the development support (Integrated Development Environment – IDE) and operating system.

The experimentation continued in 2019[15] by 7 groups of students who started from scratch. The development workflow delivered different products during the project according to the "Y"

[12] https://github.com/FrapperColin/2017-2018/tree/master/IngenierieLogicielleDomoDoor
[13] http://www.lejos.org/
[14] https://www.ev3dev.org/
[15] Note that the other case studies (footnote 8 of page 60) have also been handled by other groups and led to the same observations.

cycle of Figure 1: technical analysis (by exploring EV3 frameworks), preliminary design, detailed design, and implementation. Each product includes models and documentation. The implementation includes a source code archive, a user manual, and a reference manual including evolution perspectives. Only a part of the requirements has been implemented, the results suffer from errors and weaknesses. These products have been reused as an entry point in 2020 to engage a second iteration of the project with new student groups[16]. New objectives were fixed by the new groups based on their understanding of the freely chosen project (one of the seven): software correction (errors, bugs, low quality), software evolution (user requirements, technical change), and software validation (integration and testing). The students systematically criticized the quality of their input background (quality of the documentation and the code) but usually repeated similar mistakes due to 'not enough' time reasons, and junior experience in projects. While quality, in general, goes far beyond functional suitability, this criterion establishes the minimum sufficient level to be reached at the end of the project.

3.3 General Assessment

There are software development methodologies that provide general guidelines for the development activities but the application is in charge to the developers. There are tools for code generation but they have a very limited scope, except when it is dedicated to a particular framework (e.g., for executable UML). There exist many transformation tools and many research activities on model language engineering but either their application is limited to DSL for specific contexts or left to the developers for general purpose contexts. Isolating the various design choices is the first step to rationalizing the development in a stepwise refinement process. We follow this track in Section 4.

4 Toward a Software MDD Transformation Process

The assessment stage showed that there is no fully satisfying means to apply MDD. In this section, we draw the picture of a MDD process. We first set the principles of software design as a transformation. Then we propose a general transformation process made of four macro-transformations. Finally, we discuss implementation issues to design this transformation process.

4.1 Principles of Software Design as a Transformation

As illustrated in Figure 1, the software design consists of "weaving" the logical model (the *PIM* in MDA) and the technical model (the *platform* in MDA) to obtain *in fine* an executable model (the *PIM* in MDA). We aim to define it as a **transformation process**. We draw the reader's attention to the following principles coming from our practice observations.

P1 Model transformation can be inferred but cannot be created from scratch; we need a **complete and consistent** logical model as an input of the MDD process in order to refine toward source code. Only the technical elements ought to be injected during software design. For instance, *the input models for Executable UML tools are very detailed.*

P2 Due to the **semantic distance** between the logical model and the technical model, a single transformation (e.g., code generation) is impossible, we need several organized transformations; especially if the target is composed of orthogonal related aspects, called *domains* (e.g., persistence, GUI, control, communications, inputs/outputs), on which the logical model must be "woven". *This is the main difference between model execution for simulation purposes and model refinement into a technical target.*

[16] A teaching issue is to make them understand how much are the models and documentation valuable during software maintenance and evolution.

P3 Design, as an engineering activity, is linked to the designers' experience (*cf.* Section 3.2): a process can be automated only if all the activities are precisely known and described in a **systematic** way. The goal of development methodologies is to structure hierarchically the tasks to guide people. Experience is required when the process misses guidelines.

P4 MDE practice shows that transformations are effective when the source and target models are semantically close, e.g., class diagram and relational model for persistence. This suggests working with small transformations that focus on a few elements to transform. For instance, *when transforming a UML class diagram to Java, it is easier to handle separately the transformations for multiple inheritance, associations, class features etc.* Small transformations leave the complexity to the process, not the atomic transformations. Small transformations are probably easy to verify and compose (reusability). To be scalable, a transformation process can be encapsulated in a **composite transformation**, that will be part of another process according to the composite design pattern.

P5 There are different ways to specify the transformations. According to the recent survey of Kahani et al. [35], the tools can be classified into three main categories Model-to-Model (M2M), Model-to-Text (M2T), and Text-to-Model (T2M). The M2M transformation is the most currently used, it distinguishes four categories: relational/declarative (e.g., ATL[17]), imperative/operational (e.g., Kermeta[18]), graph based (e.g., Groove[19]), and hybrid (e.g., the QVT standard family [36]). The M2T transformation is useful, particularly for generating source code with various techniques (visitor based, template-based, hybrid).

P6 Modelling means finding abstractions that are easier to reason about. Refinement is a process of elaboration; you elaborate on the original model, providing more and more details as each successive refinement (elaboration) occurs, until reaching an operational model that can be executed. Abstraction and refinement are complementary concepts [29]. Inspired by formal methods [37], the MDD process can be seen as a stepwise refinement process [38], [36]. However, to implement the refinement steps as model transformations, we need information on the target technical system, ideally on the shape of a technical model. This is the main challenge we discuss in Section 5.4 and Section 6. In the best case, an element of the source model is implemented by an element of the technical model and it is represented in the transformation by a **model mapping** [39].

Guided by these principles, we propose, in Section 4.2, a design process composed of four configurable composite transformations.

4.2 The Proposed MDD Transformation Process

On the principles of Section 4.1, we propose a general MDD process that targets the development of software applications from models. Of course, its applicability is related to the complexity of the software system to implement. We do not target hardware development.

The process is composed of four configurable composite transformations. A *composite transformation* is a process hierarchically composed of other (simpler) transformations, according to the principle of *small step-transformations*. It is depicted in Figure 5. Each macro-transformation addresses either design or programming aspects.

- The deployment transformation **T1** starts by structuring subsystem applications with a mapping on the application architecture by describing the APIs and the communication protocols. If the logical model includes component and deployment diagrams for a preliminary design (Figure 1), the deployment transformation will be simplified.

[17] https://www.eclipse.org/atl/

[18] http://diverse-project.github.io/k3/

[19] https://groove.sourceforge.net/groove-index.html

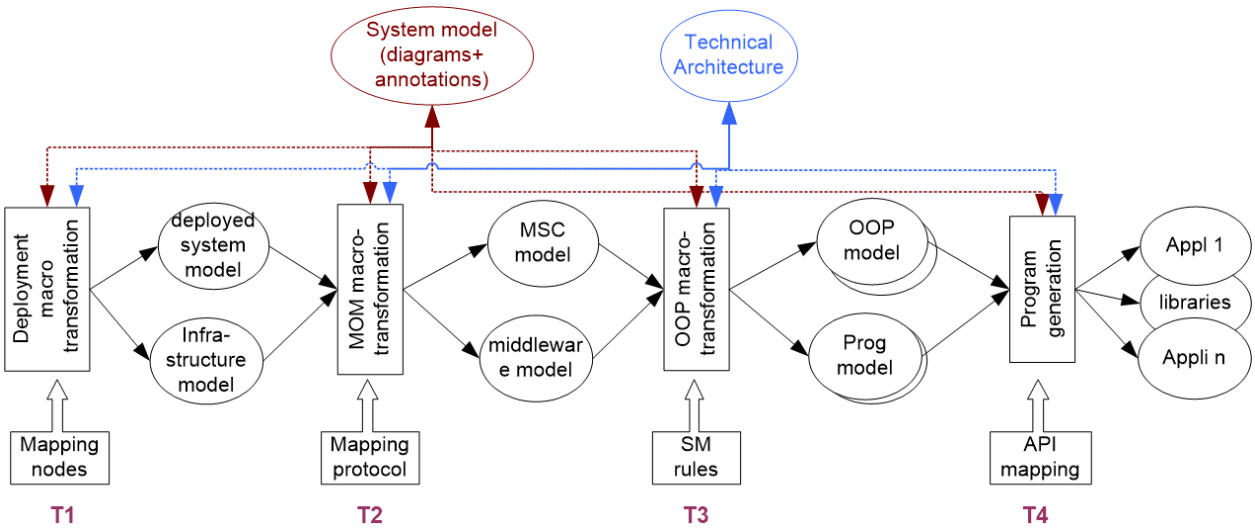

Figure 5. General transformation process

- The MOM transformation **T2** focuses on object communication. For each kind of communication, the UML message sending or signal events are refined according to the protocol under consideration (called MOM in Table 1). In a single node deployment, message sending may be implemented simply by method call in the target Object Oriented Programming (OOP) language (Java, C ++ or C#). But, as soon as the distribution is involved, we need additional communication means such as Remote Invocation Method (RMI) or network protocols (Bluetooth, wifi, wired cables).
- The OOP transformation **T3** refines UML concepts into the target implementation concepts, which are sometimes far from UML (e.g., XML structures for web applications, service or component architecture, and even OOP models). OOP models do not natively include such UML concepts as associations, multiple inheritance, metaclasses, state machines, object flows. At this stage of detailed design, we also look for the quality of software structure or practices such as design patterns or dependency injection. These thorny problems of T3 will be discussed in Section 5.3.
- The program transformation **T4** pre-processes the code generation by matching model elements to predefined libraries of the technical platform. For instance, the class Motor is implemented by the class lejos . robotics .Regulated Motor of the Lejos frameworks. This *API mapping* requires adaptors for sending messages or calling methods. This point will be discussed in Section 5.4.

All configuration parameters and all decisions of transformations must be stored to replay the transformation process in an iterative design process.

4.3 Transformations Implementation: Steps and Issues

The process in Figure 5 is abstract and generic. We discuss some implementation and customization issues.

- *Input Quality*: in addition to the static verification mentioned at the end of Section 3.2, type checking and assertions can be performed using OCL verification tools [40]. This topic is out of the scope of this article but OCL transformations to formal models or code would be of interest. Fine model verification needs adequate verification tools, model transformation is used with the benefit to target these tools having a DSL entry [41].
- *M2M*: every step of the process is an M2M transformation until the last level which is the M2T transformation to generate source code. At this stage, a rational implementation combines model transformation tools and code generation facilities of CASE tools (*cf.* Section 3.1).

- *Parallelism*: for sake of simplicity, Figure 5 hides the multiplicity of sub-models. The more we progress in the process the more we have parallel transformations. At first, T1 works as a one global application. T2 is applied to each subsystem of each node. T3 is applied to each component. T4 applies to each class.
- *Iteration*: In essence, this process is generative. However, the transformations are not fully automated, and manual injections are needed. An effective approach combines this process with a *round trip* approach in which the code injections are attached to access points (*hook*) so as not to be lost in the further (re) generation.
- *Animation*: When the technical platform is fully mastered, the transformation can "plunge" the model into the *framework* to make it executable. Refinement techniques to Java can be found in [42].
- *Tooling*: experimentation showed that no transformation tool was a panacea especially because various kinds of transformation are in play including, for instance, synthesis, extraction, mapping, and refactoring [13]. But again, the overview of model transformation tools and the combination of tools including those of Section 3.1 is beyond the scope of this article. Transformation tool surveys can be found in [43], [35].

In this section, we introduced the general macro-transformation process. In the next sections, we provide details on these macro-transformations.

5 Macro Transformations in Details

Each macro-transformation is specified in a systematic way, in order to select what can be automated or not. We illustrate them using the case study introduced in Section 3.2 and the logical model provided in Appendix B

5.1 Deployment Transformation (T1)

The T1 composite transformation was designed manually by providing a deployment model shown in Figure 6, from the analysis models of Section 3.2, and the technical architecture discussed in Section 3.2.

The Bluetooth protocol has been selected to connect the EV3 and the remote computer. In terms of transformation, the above activity is to group analysis classes into component clusters and to deploy components on deployment nodes (pick and pack). The designer must provide the component model and then interact to select classes and map to technical elements from libraries. However, new classes are necessary that structure the design. The next step would be to select a technology in a library and to map model elements.

Once the distribution support is chosen, the software designer looks at the communication means in transformation T2.

5.2 MOM Transformation (T2)

The problem is to refine UML communications according to the basic causality principle of UML[20]. *The causality model is quite straightforward: Objects respond to messages that are generated by objects executing communication actions. When these messages arrive, the receiving objects eventually respond by executing the behavior that is matched to that message. The dispatching method by which a particular behavior is associated with a given message depends on the higher-level formalism used and is not defined in the UML specification (i.e., it is a semantic variation point)*". During an object interaction , e.g., in a sequence diagram, objects exchange messages (synchronous/asynchronous, call and reply,

[20] UML Superstructure Specification, v2.3 p. 12

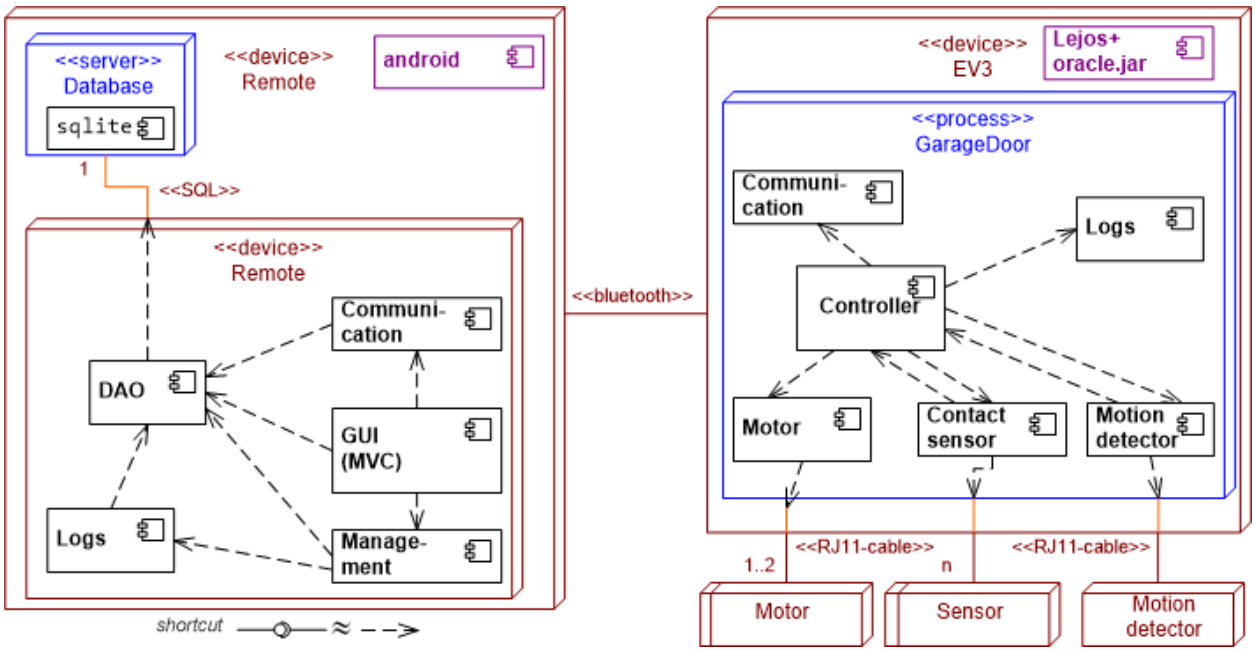

Figure 6. Deployment diagram – garage door

signals). A message receive event is captured by the receiver's protocol state machine that leads to the launch of various actions associated with the transition and the target state (entry, exit, on, and those of the do– activity inside the state). For the sake of uniformity, we consider that an action is a non-uninterruptible processing defined by an operation in the class diagram. An operation can be specified by OCL assertions or *Action Semantics* statements. The statements are useful to send messages so that the causality cycle shown in Figure 7 continues[21].

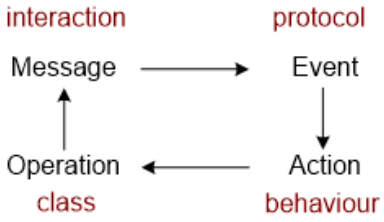

Figure 7. Basic causality principle

In plain OOP, the problem yields in transforming individual messages sent by generating OOP method call. In the general case, the transformation is complex and takes into account

- the communication medium (middleware) which is implicit in UML (reliable, lost),
- the message features (call or signal, synchronous/asynchronous, call-aback, broadcast, unknown senders, time events, etc.),
- the underlying protocols (TCP-IP layers of services),
- the connecting mode (stateless, session).

For instance, in the case study, the remote device and the controller exchange with session-based protocols. It is assumed the devices are physically bound: the EV3 cables are connected to sensors and adapters. A Wifi or Bluetooth connection is required to be done manually and interactions happen during a session (open session, exchanges, close session).

To help the transformation, the software engineer can use *profiling information and stereotypes* in the logical models to provide information on the target communication platforms. Having profiles for the technical model is also helpful.

In our work, we experimented with the abstraction of communications into a *communication primitive* layer in order to map messages sent to those primitives. A project led by a group of master students[22] explains the main issues and illustrates them in the conducted case study. Beyond

[21] Note that this cyclic causality principle establishes a consistency rule to cross-check sequence diagrams, state-transition diagrams, and class diagrams [41]

[22] TER Report - Refinement of communication protocols by models' transformation `https://ev3.univ-nantes.fr/rapport_ter_22-05-2020`

the problem of defining the underlying communication support (service and protocol implementations, configuration, initialization), the main point, considering UML models, is to isolate the message sent from the models before processing the communication statements transformation. For the sake of simplicity, the students chose to extract messages from sequence diagrams, since the messages sent are explicit[23], and processed ATL transformation to introduce lower lever communication messages. Examples of models and concrete Java code have to be implemented. However, sequence diagrams (or communication diagrams) are instance diagrams but not rules. The true sent messages are found in the actions of a state-transition diagram or in the operations defined in the classes. The lessons learned from that experimentation are the following:

- Messages are low level concepts in terms of the UML diagrams except in sequence diagrams. Transforming message communications implies messages to be explicit in state-transition diagrams (actions and activities) and operations (activity diagrams or actions). A full action language is not mandatory, only the part related to messages and events (e.g., as a DSL).
- Some messages are simple procedure calls in the target program. For instance, the communications between EV3 and the sensors/actuators are Java method calls. We call them *primitive messages* in opposition to *protocol messages* which enable distant objects to communicate.
- From the result of transformation T1, we simplify by considering that primitive messages are used for objects deployed on the same node while protocol messages are used for objects deployed on different nodes. Recall that the deployment diagram provides the protocol stereotype on the communication path between nodes. Otherwise, user information is necessary to process the transformation.
- For each communication path, we associate communication services and protocols. This communication infrastructure (middleware) is installed and configured in the main program.
- Each individual protocol message is transformed into a proxy call that will be in charge of transferring the message to the receiver according to the middleware configuration.

When there are different communication media (e.g., wired, wifi, Bluetooth) an alternative is to consider communications as an orthogonal interoperable concern. We proposed a solution to that problem called a Multi-protocol communication tool (MPCT) in [44].

Once the communication support is chosen, the software designer looks at the module's detailed design in transformation T3.

5.3 OOP Transformation (T3)

The goal of macro-transformation T3 is to refine the UML models to UML profiles corresponding to the target programming languages and frameworks. T3 also corresponds to detailed design in software development methodologies and makes use of design patterns, dependency injection, and others to build quality source code structure. T3 covers a very large set of technologies which are not only programming languages, e.g., XML files to describe components, services, or web applications. When the technologies cover orthogonal concerns, the platform can be split into different domains linked by communication bridges [45]. For instance, in [4], the PIM is transformed in a relational domain, an EJB component domain, and a web domain. This makes sense in n-tiers web architectures. As an example, the transformation of UML classes to Relational Database schemas has been extensively discussed in the literature and it is what we qualified as semantically close domains in principle P4 in Section 4.1.

In this section, we will not explore all fields, we illustrate the problem's complexity by transforming plain UML models to UML–java models. We assume UML–java to be a UML profile that accepts only UML concepts that are meaningful in Java. For the sake of conciseness, we sketch the following simplified *sequence* of transformations:

[23] Abstract to raising signals or time events

1. Transforming class diagrams `[T3.1]`,
2. Transform state-transition diagrams `[T3.2]`,
3. Transform activity diagrams associated to operations into OOP structures `[T3.3]`. This problem is a variant of the STD transformation if we do not consider the object flows, otherwise we need a specific flow domain.

T3 is a transformation process implemented with intermediate steps and each rule is implemented by one transformation (or macro-transformation). We could define specific UML profiles for each intermediate step, e.g., UML-SI-OOP, a UML profile dedicated to OOP with single inheritance is an intermediate step to Java. The designer can then select the sub-transformations and organize the macro-transformation T3. Next, we present T3.1 and T3.2.

CD Transformation `T3.1`. Basically, classes exist in OOP. Some features are not available in OOP languages such as association, others exist in some languages only (e.g., Smalltalk, C#, and Java, which consider only single inheritance while Eiffel or C++ accepts multiple inheritance; Smalltalk accepts metaclasses, the other languages do not, etc.). We consider in the following, some such features. The order is important when a transformation results in concepts that will be later transformed again.

1. Transform multiple inheritance to single inheritance[24] is to determine the main inheritance flow either the first in the multiple inheritance order or by a metric that computes the feature reuse rank. If the target model allows the "implement" inheritance variant, e.g., Java or C#, the secondary inheritance flows are defined by interfaces. If it does not, e.g., Smalltalk, features are duplicated.
2. The derived features (attributes, associations) are transformed into operations. If an OCL constraint gives a computation, it can be an assertion of the method associated with this operation.
3. Associations are quite complex features in UML. We suggest to proceed as follows.
   (a) Class-associations are transformed into classes plus associations. The multiplicity is 1 in the new class role side.
   (b) Aggregations and compositions are transformed into simple associations.
   (c) Bidirectional associations ($A \leftrightarrow B$) are transformed into two unidirectional associations ($A \rightarrow B$ and $A \leftarrow B$) with a symmetric constraint ($(a,b) \in A \leftrightarrow B \implies (b,a) \in B \leftrightarrow A$). Keeping only one of both is a (good) design decision that reduces class coupling (*dependency inversion principle* of the SOLID principle). It can be decided automatically if no navigation path exists in the OCL constraints associated with the model.
   (d) Unidirectional associations $A \rightarrow B$ are transformed into attributes (called references in UML to be distinguished with primitive types or utility classes). The attribute name is by order the role name or the association name of the implicit association. The type of the attribute in class A depends on the multiplicity and the constraint:
      - $b : B$ if less or equal to 1. Note that in case of 0..1 it should mention a union of types $B \vee Null$ since it is optional.
      - Otherwise it is a set, an ordered collection if there is an `{ordered}` constraint, a sorted collection (a map if the association is qualified).
4. Dependencies are transformed into <<import>> dependencies. Variants are possible according to given stereotypes.
5. Meta-features (attributes, operations) are handled specifically in C# or C++. For instance, they are implemented by static features in a UML-Java profile. If other meta-facilities are used, e.g., in OCL constraints, using a Factory pattern [34] would be of interest.

[24] Note that the transformation from UML classes to relational databases transforms with no inheritance. Intermediate classes, especially those which are abstract may disappear by aggregating attributes in the root or in the leaf classes. Another transformation replaced inheritance by 1-to-n associations.

6. Operations are transformed into methods. If an OCL assertion (resp. a fUML statement) was associated to the operation, it can be an assertion (resp. a statement) of the method associated to this operation.
7. Stereotypes can be handled depending on the OOP. As an example, a candidate identifier <<key>> (for persistent data) leads to uniqueness constraints in OCL invariants <<abstract>> for abstract class or methods.
8. OCL invariants can be implemented by assertions or operations that are called every time an object is modified. They will be interesting for testing.

STD Transformation `T3.2`. The state diagrams are associated to classes and transformed into OOP structures (this feature is mentioned to be a missing criterion in Table 1). Various strategies are possible to represent states, e.g., (i) a couple of states can be represented by a boolean (light is on or off) and the transitions are subject to an `if` statement, (ii) a few states can be represented by an enumeration and the transitions are subject to a `case` statement, (iii) if the states denote very different behaviors (not the same methods) a State pattern, is welcome [34], (iv) if there are many states, then we need instrumentation machinery (with transition and action matrix).

Example: UML2Java, a STD Transformation with ATL. Due to its expressibility and abstraction, we chose ATLAS Transformation Language (ATL)[25] to conduct these experiments. ATL is a model transformation language based on non-deterministic transformation rules. In a model to model (M2M) transformation ATL reads a source model conforming to the source meta-model and produces a target model conforming to the target meta-model. At this stage, we used the model to text (M2T) transformation type to generate Java source code. The input model is a Papyrus model (XMI format for UML 5) composed of class and state diagrams (CD + STD).

ATL proposes two modes for transformations from and refine. The from mode enables the creation of a model by writing all the parameters, and all the attributes, in the output model. The refine mode is used to copy anything that is not included in the rule into the output template and then apply the rule. A rule can modify, create, or delete properties or attributes in a model. In this mode, the source and target meta-models share the same meta-model. The refine mode is more interesting for our transformations because we are working on partial transformations. Moreover, we want to avoid DSL explosion and limit the number of metamodels or profiles. The ATL transformation information is given in Appendix C.

The experiments highlight the complexity of the problem and some basic aspects to deal with. The results are still far from the final objectives. Once the modules are designed, the software designer looks at their implementation in transformation T4.

5.4 Source Code Transformation (T4)

Transformation T4 aims at unifying model elements and implementation (source code). Usually, it is considered as a Model-To-Text transformation. However, all model elements are not generated from scratch, some already exist, maybe in a different nature, in the technical model (*cf.* Figure 1). As mentioned in Section 3.1, we look forward *API Mapping*, a feature to map model elements to predefined elements in libraries or *frameworks*. In this section, we study the mapping of design classes (and operations) to predefined code source classes and we experiment with source code generation. To simplify the discourse, we will focus on classes as the model elements, but they should be extended to packages, data types, predefined types or operations, and so on.

API Mapping. All the classes of the model need not to be implemented, some exist already in the technical framework. In our case study, the sensors and actuators already exist at the code

[25] https://www.eclipse.org/atl/

level in the Lejos library. For the sake of simplicity, we consider that a model element maps to one implementation but an implementation can map to several model elements (1-N relation). When model and implementation elements do not match, developers usually refactor the model to converge. The mapping process includes three activities

1. *Match* to find implementation candidates in libraries with possible matching rates. Different model elements are captured such as class, attribute, operation, etc. We face here the following two issues:
   - Abstraction level. Basically, the model and implementation elements are not comparable and we need a model of the implementation framework. This abstraction issue will be discussed in Section 6.
   - Pattern matching. The model elements are not independent, e.g., operations are in classes which are grouped in packages. The way the model elements are organized influences the matching process.
2. *Select* the adequate implementation of the model element (class, attribute, operation) and bind the model elements. We proposed a non-intrusive solution to this problem [46].
3. *Adapt* to the situation. Once a mapping link is established, it usually implies the need to refactor the design. Adaptation is the core mechanism to bind the two branches of the "Y" process in Figure 1. Different strategies can be chosen:
   - *Encapsulate and delegate.* The model classes are preserved that encapsulate the implementation classes (Adapter pattern). The advantage is to keep traceability and API. The drawback is the multiplication of classes to maintain.
   - *Replace* the model classes with the implementation ones. The transformation replaces the type declarations but also all messages sent. The pros and cons are the inverse of encapsulation.

   During forward engineering, the students used both strategies, depending on their concerns such as traceability, ease of implementation, and code metrics. Replacement is possible when classes have the same structure and same behavior but also for UML/OCL/AS primitive types. In any other cases, the Adapter pattern captures multi-feature adaptations:
   - Attribute: name, type adaptation, default value, visibility, etc.
   - References (role): name, type adaptation, default value, visibility, etc.
   - Operation: name, parameters (order, default), type adaptation, etc.
   - Protocol: STD for the model class but not the implementation class.
   - Composition: a class is implemented by several implementation classes.
   - Communication refinement: MOM communications are distributed.
   - API layering: classify the methods to reduce the dependency.
   - Design principles: improve the quality according to SOLID, IOC, etc.

The high-level frameworks for MOM or STD are not concerned by these issues because they are pluggable components. In the remainder of this section, we describe experimentations on code generation transformations.

Source Code Transformation with ATL. Our transformations chain also includes another type of transformation, specifically the M2T Transformation, which generates Java code from UML models produced by the T3 transformation. To achieve this, specialized tools like Acceleo[26] are particularly suitable for M2T transformations. However, given that Acceleo has not been utilized by the authors before, and due to time constraints, in the initial phase, we opted to prioritize the use of ATL, which also offers capabilities for generating text through its helpers (methods) to parse the XMI model features. This transformation is an M2T transformation that generates source code from the UML models resulting from transformation T3. Details are provided in Appendix D

[26] https://eclipse.dev/acceleo/

These experiments highlight the complexity of the task, especially when different alternatives exist. In the case of STD, again, the design choice for state implementation (enumeration, state pattern, or machinery) impacts the remaining to be done especially for the operation-to-method transformation. For instance, the STD graph can be distributed over the operations or centralized in a unique behavior−protocol. We advise the second way which is easier to maintain. Other issues like threads and synchronization have not been discussed here because they better take place in an STD-framework. Again this reports many implementation problems to API mapping instead of code generation.

Source Code Transformation with Papyrus. Since 2017, Papyrus has provided a complete code generation from StateMachines. The implemented pattern is a part of the Papyrus designer tool. It considers the following Statechart elements during code generation: State, Region, Event(*Call Events, SignalEvents, Time Events, ChangeEvents*), Transitions, Join, fork, choice, junction, shallow history, deep history, entry point, exit point, terminate. A deep presentation of the algorithms designed to translate these elements into code is available on [47]. The code generator engine of papyrus extends IF-Else/Switch construction of programming languages that support state machines hierarchy. It brings many features compared to the existing tools [47] such as:

- All statechart elements are taken into account during code generation,
- Consider sync/asynchronous behaviors through events support,
- The used UML conforms to the OMG standard,
- Efficiency: events processing is fast and the generated code size is small,
- Concurrency and hierarchy support.

The generated code could be only in C++. Accordingly, we have to use ATL transformation as an intermediate to adapt our Papyrus UML models to our Lejos programs based on Java programming language. The transformation pattern we implemented by ATL is based on the State Design Pattern which also supports hierarchical state machines. These solutions suffer from one limit, the explosion of the number of classes that requires much memory allocation. Ongoing work was announced by Papyrus designers to add Java code generation from STDs, but we could not access it. After we conducted our experimentation, we also identified a complementary approach by generating C++ code and enhancing our generated Java code by transforming state chart components. Several automated libraries, such as `c2j`, can be used for converting C++ to Java code.

Source Code Transformation Using Mapping. This transformation finds candidate mappings and establishes the mapping by adaptation.

a) Candidate Mapping For each class of the `Model`, e.g., Motor the goal is to find, if any, candidate `implementation` classes in the framework to map to. A prerequisite is to have at disposal a model of the framework or to establish one if none exists yet. This point will be discussed in Section 6.

In a previous work [48] we faced the problem of identifying components in a plain Java program and one of the issues was to compare a UML component diagram with extracted Java classes. We used string comparison heuristics that were efficient for 1-1 mappings with similar names. When a component was implemented by several classes, even with naming conventions, the problem was inextricable without user expertise. A key best practice is to put traceability annotations, *just like the little thumb places stones*, to find a way back. However, the problem is not really to discover the source code to establish the traceability links but to find potential implementation of some model classes. In [46] we suggested an assistant to present elements in double lists and to *map them by drag and drop*. The mapping is non-intrusive and up to model evolution. This is clearly a convenient solution for small size applications. In order to make it applicable we suggest the general guidelines:

- Model preprocessing: use stereotypes to separate the utility or primitive classes, the STD are not taken into account (State patterns are excluded).
- Implementation preprocessing: get an abstract model of the different implementation libraries and find the entry point libraries (*cf.* Section 6).
- Apply a divide and conquer strategy to avoid mapping link explosion.
  1. Map parts: isolate model subsystems and implementation frameworks
  2. Map concerns: isolate model points of view (design concerns) and implementation libraries
  3. Map packages: isolate model packages and implementation libraries
  4. Map classes: establish links between corresponding classes -if any
  5. Map operations: establish links between corresponding methods -if any.

As an example, we list model classes and candidates in Table 2. The implementation classes come

Table 2. Mapping candidates

Model	Lejos candidates	Choice	Comment
Motor	<<abstract>> Motor		Motor class contains 3 instances of regulated motors.
	EV3Large RegulatedMotor	Installed	Actually, it depends on the installed hardware.
	42		other classes or interfaces with "*motor*.java"
	lejos .hardware.motor package		11 classes or interfaces with "*motor*.java" over 13
ContactSensor	no		
	EV3TouchSensor	Installed	
	49		other classes or interfaces with "*contact*sensor*.java"
MotionDetector	no		0 classes for "*motion*.java"
	EV3UltrasonicSensor	Installed	
Communication	BTConnection		if bluetooth
	lejos .remote.nxt package		
Controller			outside the EV3 libraries scope
Remote			android App
Communication	android . bluetooth package	Installed	BluetoothAdapter, BluetoothDevice, BluetoothSocket

from the Lejos library (see Section 6.2). Recall that the GUI part is considered to be developed separately. For the simple example of class Motor, Table 2 shows that it is not an easy task to detect which candidate class could be appropriate. We definitely do not look for automated mapping but mining facilities to detect candidates based on names (class, attributes, operations), the user is in charge of deciding the class to map.

b) Adaptation To simplify the description of the mapping attributes and their injection in the previous ATL transformation engine. we preferred to represent them as a properties file containing the list of mapping attributes. Following the ATL specification any input file should have an XMI format and respect a description defined by its meta-model. For this fact, we defined a model for the mapping properties as shown in Figure 8. Implementation details and illustrations using ATL are provided in Appendix E. Such transformations are sufficient for direct name-based mappings but additional work is necessary for a more complex transformation, and often developers have to code more complex adaptations.

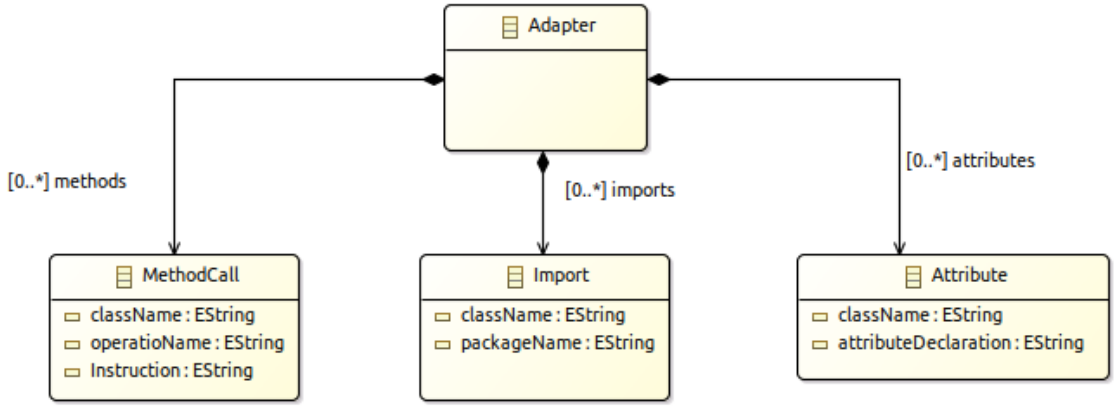

Figure 8. Adapter Pattern Model

5.5 From Systematic Description to Implemented Transformations

In this section, we structure MDD as a transformation of four macro-transformation to organize the development activity. Then we describe, in a systematic way, the tasks to be processed in each macro-transformation. These tasks are to be implemented as a model transformation. We illustrated the implemented transformation in the case study. Writing the transformation is complex work and it requires information from the technical side because one can not implement technical code from scratch. We illustrated this fact with the adaptation transformation. To get this information, we need to infer the technical model (right branch of the "Y" in Figure 1). The next section tackles this issue.

6 Reverse Engineer PDMs to Help Defining the Macro-Transformations

Refinement transformation requires information from the technical model (Principle P6 in Section 4.1). We illustrated this principle by implementing model transformation by model mapping in Section 5.4. Model mapping is made applicable only if a model of the target framework (PDM) exists. One way to get an XMI model is from the framework documentation, which is actually never the case. Another way to fill this hole is to extract the model from the framework source code by reverse engineering as depicted by Figure 9. Note that the mapping persists at the PSM level, e.g., adapters store the API mapping (*cf.* Section 5.4); and our goal is not to reverse engineer PSM programs (Round-trip engineering is the solution for that) but to find abstraction to build the PDM.

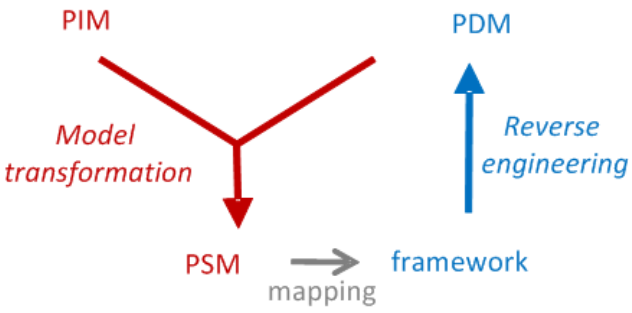

Figure 9. View of a mapping transformation

6.1 Model Driven Reverse Engineering

Reverse engineering is the process of comprehending software and producing a model of it at a high abstraction level, suitable for documentation, maintenance, or re-engineering [49]. It aims at producing high (abstraction) level models from software systems according to various software maintenance objectives including technical upgrading, business process alignment, improving quality, etc.

As far as MDE is perceived as a transformation process, MDRE is also a transformation process [50]. An MDRE step can be represented by a model transformation from a PSM up to a PIM as illustrated in Figure 10. A reverse engineering

process will be a composition of such model reverse transformations where the reverse designer will have to define the meta model of each intermediate model: a PIM model of transformation will be considered as a PSM model of another transformation. In MDE, writing model transformations is not a very simple task, however, the source and target models are usually known. Finding abstraction is an even more difficult problem in MDRE [51], [49]. Abstraction hides implementation details. During the development of software systems, high level abstraction models refer to systems analysis and design while low level ones refer to the implementation and deployment of the solutions. *Abstraction layers* represent the organization of complex architectures; typical examples are the ISO stack of protocols and services for telecommunications or the Service Oriented Architecture (SOA) approach. The relations between *model elements* of different layers are *refinement* or *traceability*. Sometimes inheritance is used to materialize the abstraction between comparable model elements. In our case, the abstraction layers are levels T1 to T4.

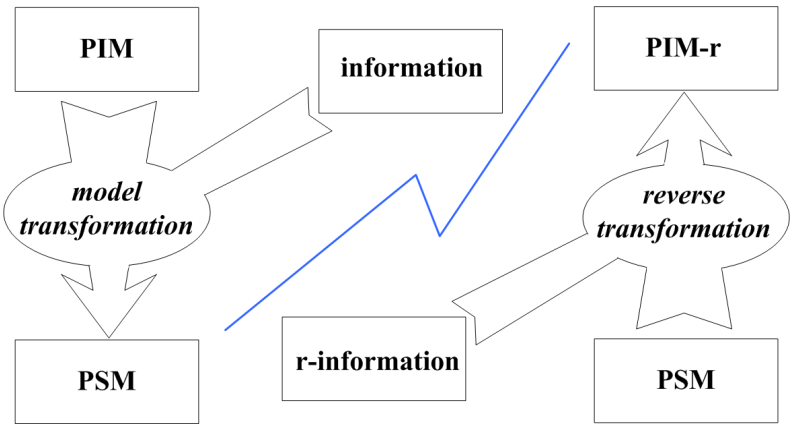

Figure 10. Reverse Model Transformation

MDRE may target different levels of abstraction, from program representation to high level application architectures or business processes. Consequently, different types of models are expected with various notations like de facto MDE standards such as UML, OCL, MOF, EMF, SysML, AADL, BPMN, or customized models defined with DSL. The source information also differs and may include binary code, source code, configuration files, test programs, scenario models, etc. Such diversity makes the RE activity difficult to solve.

The question is not to sort abstract and concrete elements from the source information but to build abstractions. The more we hide in the abstraction, the more difficult it is to find the abstraction. For instance, Knowledge Discovery Metamodel (KDM) is a standard for software system representation [52]. It is useful at a low level of abstraction because it is a model representation and contains very detailed information that misses abstraction.

6.2 Example: Reverse Engineering Lejos Libraries

In our conducting case study, we use a model from the specific Lejos framework[27]. More precisely we consider only the leJOS EV3 library[28] as the PDM but other frameworks exist like EV3dev[29], Python, Android. We extracted the main in its source form `ev3classes-src.zip`. The o3smeasure metrics are given in Figure 11.

For this case study, we experimented with Papyrus, Modisco and AgileJ. Papyrus enabled to reverse engineer[30] individual classes but not packages. In the context of a papyrus project, applying the command `Java>Reverse` on lejosEV3src model elements failed except for classes. Even for a class, the methods were not included. In Modisco [52], UML discovery from Java code is composed of two transformations (Java to KDM / KDM to UML). Unfortunately, the second one

[27] Android/Java libraries are considered as standard for the experimentation purpose.
[28] Lejos is a complete Operating System based on an Oracle JVM.
[29] https://www.juanantonio.info/blog/2017/03/20/why-use-ev3dev-lang-java.html
[30] https://gitlab.eclipse.org/eclipse/papyrus/org.eclipse.papyrus-designer/-/wikis/reverse/java-reverse

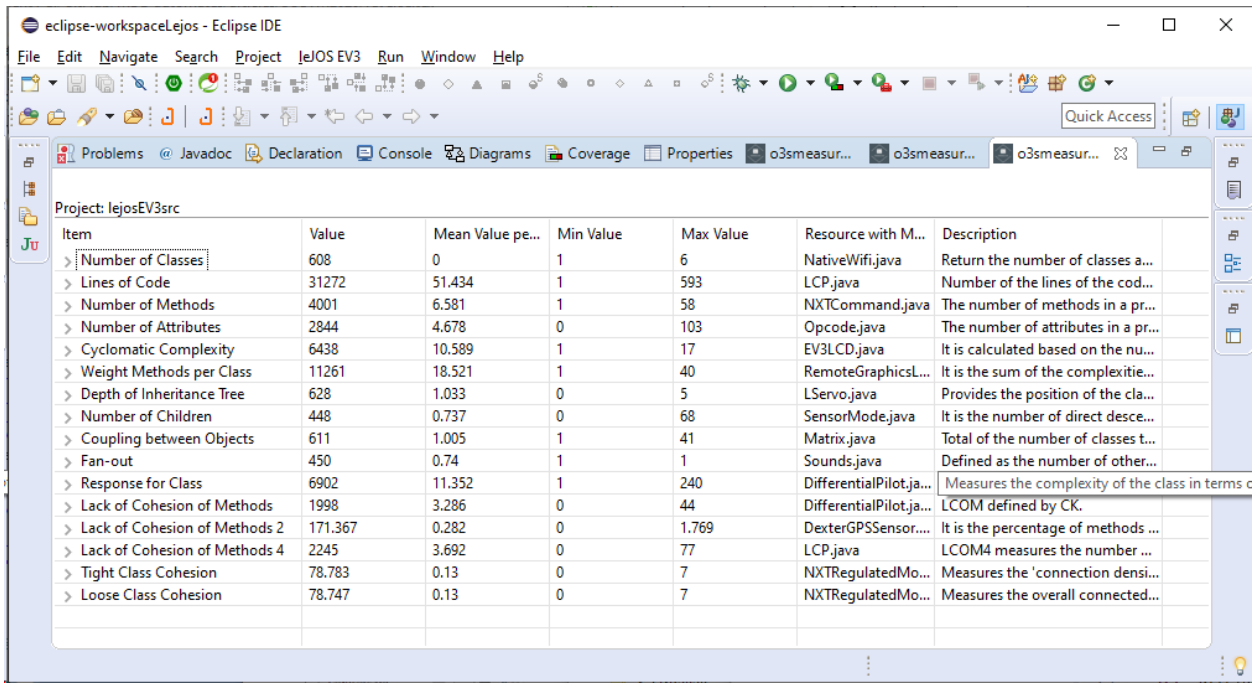

Figure 11. Metrics of the Lejos EV3 classes library

is no more available in the Eclipse Modelling distribution but remains available in the Modisco git repository. Once again, we faced two ATL compatibility problems: lazy rules are not allowed in the refining mode and the distinct ... foreach pattern is also forbidden in that case. Also, the methods were not captured as model elements in KDM. In AgileJ[31], reversing the Java code to UML class diagrams is simple. Figure F1 in Appendix F shows the result of applying reverse engineering on Lejos library using AgileJ/structureviews. The visual representation provides many relationships between classes, compared to other tools like ObjectAid (see Figure 4). In the case when the number of classes is large, the small representations help build and maintain a better overview of the architecture and highlight where the design can be improved and refactored.

Recall that the initial problem to solve is matching model elements to PDM abstractions. In this experiment, the working unit is the class element. For each model class, e.g., Motor, the goal is to find candidate implementation classes in the framework model. The MDRE process aims at providing foundation classes, those which can be candidates for mapping. In order to reduce the number of classes to compare, we apply the following simple heuristics: (i) focus on Java source files (479 among the KDM elements), (ii) select only interfaces (160) and abstract classes (19), because usually frameworks are structured to evolve, (iii) search according to string matching, or (iv) better on pattern matching (including references, attributes, and operations). These can be implemented by Modisco queries as illustrated by Figure 12. Specific stereotypes or annotations to separate model classes are helpful in the case of iterative processing.

AgileJ provides a filter tool (*cf.* Figure 13) which is powerful enough to remove the noise from the key structural elements. Once the filter is applied, it changes the content of the screen, e.g., shows all interfaces or abstract classes.

In the simple example of class Motor, the string matching provides 11 interfaces and abstract class BasicMotor. This is a reasonable set to find potential API mappings (*pick and adapt*). The above observations are not definitive opinions on the tools. AgileJ provides visual and interactive information while Modisco enables customized querying and transformation. Further details can be found in [50].

[31] https://marketplace.eclipse.org/content/agilej-structureviews

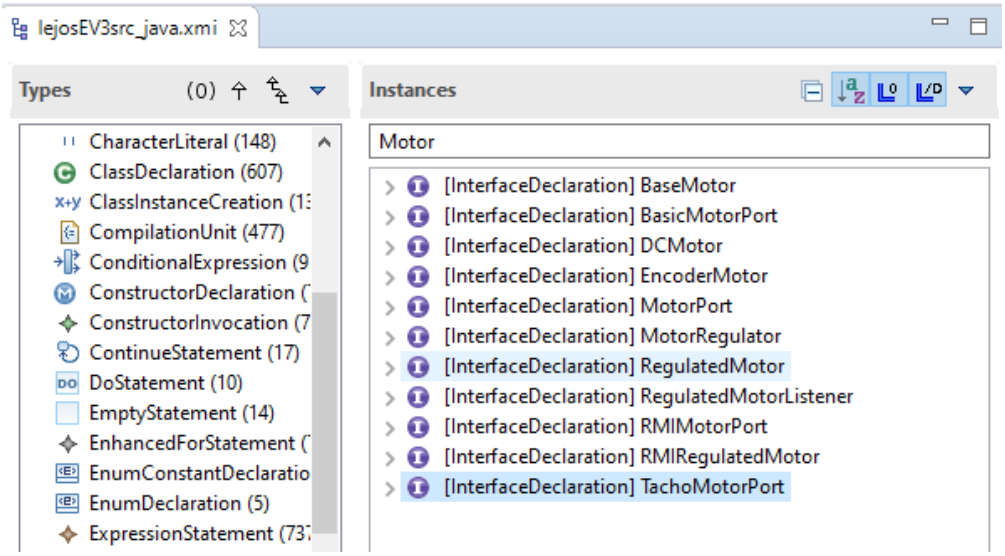

Figure 12. Modisco discovery for interfaces

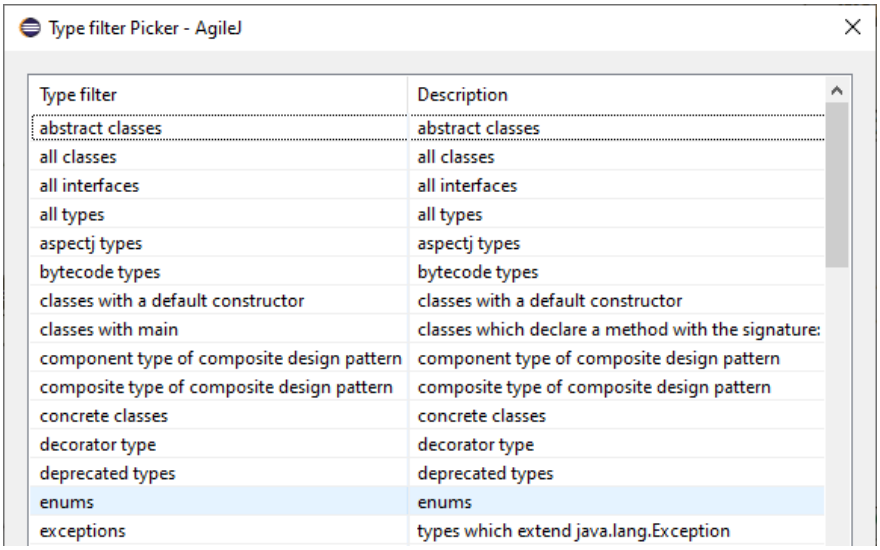

Figure 13. AgileJ filter process

7 Discussion

We discuss here some lessons learned from the above studies and related works. The manual design of the application from a logical model was not greatly difficult for the students, except for learning the target technical environment. However, we found that the code did not meet the requirements or the initial model, which, although detailed, did not guarantee the consistency or completeness of the system specification. In addition, technical constraints are required, such as the fact that the Lego model uses two motors (one per door panel) and not a single engine as in the model. In the same way, the wireless communication between the remote control and the controller remains abstract in the form of sending messages in the model. The manual design shows various orthogonal aspects that were not prioritized by the students. Dependencies remain implicit for them, even if they realize that choices for one aspect will influence other aspects. Last but not least, it is not efficient because (a) forward engineering is time consuming especially when the models evolve and (b) the experience Return On Invest (ROI) is more individual than collective. Our contribution focuses on methodology and guidelines for MDE developers.

Lesson 1: MDD is a complex task and assisted MDD is an open field. To the best of our knowledge, transformations from analysis to code from a practitioner's point of view, has not been addressed as a whole in the literature. The development standards are not immediately applicable here. For instance, in [53], the authors use MDE for process compliance, however, they refer to standard or de facto development processes that are far from practice orientation. They are not concerned with the produced model contents but rather with their meta-information. As mentioned by Aranda et al. in [54], the step from traditional development, even with a model, to MDE is large and disruptive, and *"practitioners and researchers have little information to help guide them on this process."*. It was still the case in 2020, especially for code generation as mentioned by Sebastián et al. in their systematic mapping study [55]: *"There is still a long way to go in the field of MDA, and -in many cases- the automatic generation of code from models is still a software engineers' dream, and the development and subsequent publication of research works that use MDA to generate code is still complicated.* However ad-hoc solutions exist. For instance, Sindico et al. [56] present an MDE process based on the INCOSE framework that conforms to the MIL-STD-498 standard for military real-time embedded systems with SysML Marte, and Simulink. The platforms play the role of integration system for engineers. SysML models are given for the requirements but also for the target platform. The solution works for electronic devices but is interesting to compare with.

Code generators provide incomplete models, which often do not even exploit the information of the model (OCL constraints, operation details). This assertion may be different for some expensive commercial tools that we did not try.

Lesson 2: Even for detailed models, automatic code generation cannot apply in the large. Although many studies have been conducted, the systematic study of Ciccozzi et al. [57] shows that the execution of UML models remains a difficult problem and meets animation, not software development needs. However, the new standards `fUML` and `Alf` contribute to palliate a lack of action semantics. They have been implemented, for instance, in the verification of models [58], [59], execution via C++ [60] or MoKa/Papyrus [61], [58]. It is possible that MDE can cross the threshold in the detailed design (T4) when tool support handles these standards.

In MDD, the abstraction gap between analysis models and the detailed design models is huge. In the development *workflow* in Figure 5, the macro-transformations are ordered according to the impact: architectural choices (deployment, communications), general design choices (programming language, *patterns*), detailed design choices (*library mapping*).

Despite we did not cover all design concerns such as persistence, time, and XML descriptions, the experimentation showed that the transformations are already numerous, complex, and difficult to design in terms of model transformation. Also in our transformation specifications, we did not consider domains and bridges and the orthogonal aspects of PSMs [4]. Software structures such as design patterns can also be abstracted in order to help refinement in the T3 transformation (Sec-

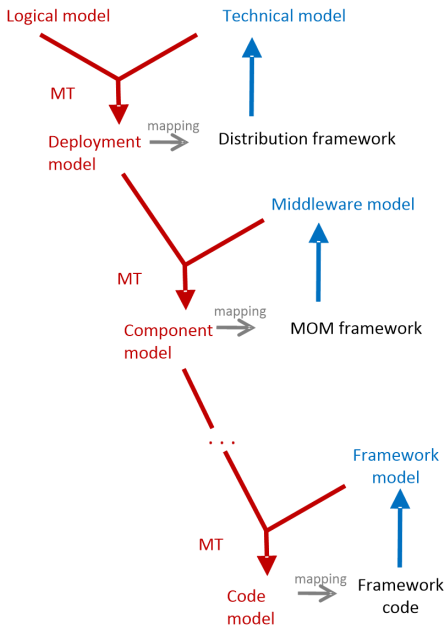

Figure 14. Two-track transformation process

tion 5.3). In any case, we clearly mentioned that technical information on the target platforms is required to define refinement model transformations (the red transformations in Figure 14). In Section 6, we propose to infer it from the frameworks by reverse engineering (the blue transformations in Figure 14).

Lesson 3: Structuring MDD in a hierarchical sequence of transformations is really a challenge; composing a transformation is even harder. Section 4 makes it clear that the transformation specifications are incomplete, due to their complexity. A rigorous approach would be to formalize the practitioner's activity in a larger experience than those of our students. In this formalization, the refinement process should require information from the developer only when design choices depend on design decisions. We need systematic definitions at a low level but the work is not yet finished.

In Section 4, we used a top down presentation of the transformations, however, building the composite transformation uses a bottom-up approach where low-level transformations are composed to build higher-level transformations in sorts of model flow processes. We studied this point in the context of checking UML consistency in [41] and pointed out the problem of structuring such model flows.

Lesson 4: Providing information and decision as parameters to design transformation is really a problem. The technical model is not just an abstract description of the framework(s) code; it should be abstracted to architectural views, styles, and patterns. A design decision is then to select appropriate features and insert them into the transformation. Among the techniques to define transformations [4], [13], we believe that *model mapping* to connect application features to platform features is important. The model mapping is declarative, not intrusive, and can support technical evolution. In an ideal vision, the refinement process introduces required design decisions by mapping them to provided features of the platform models. For the sake of simplicity, Figure 14 shows one couple of models per level but the more is refined to code the more domains are in parallel to be connected by bridges [4]. At each level, a model of the technical frameworks is requested. Reverse engineering tool support exists at a low level and raising in abstraction remains an engineering task [50]. The mapping process requires a matching process. Many techniques exist as pointed by Somogyi and Asztalos [62], based on graphs or text algorithms. However, they are more useful to compare models than to find implementation candidates, as we showed in Section 5.4. A *non-intrusive mapping* technique and a tool are presented in [46] to implement the model mappings.

Among the design decisions, the best practice teaches us to reuse previous experiences and success such as architectural or design patterns. In the forward engineering experience, the students made use of design patterns for detailed design and we also suggest that way in the T3 and T4 transformations. We observed pattern implementation from books or student work and observed that implementations vary making it difficult to process systematically pattern transformations. Such information could be described in the technical platform models in order to comply to the mapping process.

Lesson 5: Reuse MDD transformation remains an emergent field. We did not find MDD transformations: *"MT reuse is a real need, but existing reuse mechanisms have not percolated practice yet."* [63]. Also, there exist works on transformation patterns, e.g., [64], that improve some quality characteristics such as modularity or efficiency of the transformations but their intent seems to be far from ours. Design decisions should be stored to be replayed either later in a new release or in another "similar" transformation. Basically, the idea was to store the transformation

blocks we proceeded in a repository. These transformations may be generalized to be parametric in order to instantiate them with the same or other parameters. However, "transformation engineering" is far easier to implement because, in addition to the transformation statement, a transformation specification includes the metamodel of the input models and the output models.

The MDD process requires technical models. Similarly to the Components On The Shelf (COTS) vision, it would be preferable for the providers of framework platforms delivered not only code libraries but also models (PDM). Those models than would (i) play the role of documentation because current APIs are difficult to overcome and (ii) be inputs for MDD transformations. For instance, a state machine protocol is convenient to express the methods of an API and a component diagram exhibits the architecture of a large library.

Lesson 6: MDRE is required to find abstractions in the technical platforms. As discussed in Lesson 3, platform abstractions are necessary to write mappings from logical elements to technical ones. Section 6 illustrated how a logical motor could be implemented by one of the EV3 platform motors. This is actually the way we proceed in manual engineering; we look for elements in libraries. MDRE can assist by providing candidates in transformation T4. To raise in abstraction level, we need new heuristics and inferences such as finding components in a plain code [48]. We feel that many of the challenges on model evolution given in [65] remain on the way to building PDMs.
An alternative to MDRE is **low code** where somebody already has achieved abstractions. In the Lego context, EV3 Mindstorm is a visual language to program EV3 robots using algorithmic blocks, it is a low code approach, but the API is restricted versus the Lejos ones. Low code is one level of abstraction while we need several levels, corresponding to several concerns in software design. Despite our case study is a cybersystem rather than an IoT application, we share technical issues such as remote communication autonomy and control; and the MDE approaches overviewed in [66] are relevant to investigate.

Our workflow model remains abstract in the sense that the parameters to be provided remain substantial and these transformations are themselves processes of transformation. Nevertheless, it is generic enough to be customized to projects by parameterization and transformation substitutions.

Lesson 7: Collaboration is required to achieve actual transformations. Similarly to the component-based approach, the MDE developer should have models, metamodels, and transformation on the shelf to build its actual transformation process. Standards exist for metamodels and many open source projects propose tools. The challenge is to have community PDM models and transformations and also development components to integrate them. The development strategies should be parameterized in the transformation process. For instance, coding state machines are subject to interpretation and are strongly related to the execution model [32]. Also, we believe that several transformation tools should be combined because the rule-based approach is unsuitable in some operational transformations such as the one mentioned in [32]. In particular, we tried to combine model transformation with code generators.

We observe that the MDE tools have improved but the process does not reach an industrial milestone and MDE does not replace classical software development. Many transformations are in charge of the designer and the tool support offer is usually not mature enough, we always have problems with release compatibility (UML versions, XMI versions, Plugins, libraries, Java, IDE, etc.).

Lesson 8: MDA is an evolving entity that generates growing technical debt. We observed improvements in the standard compliance of tools, sometimes by meta-model co-evolution, but the tool support in general is still not mature. Many tools are not maintained and some important facets, such as incrementality, built-in traceability, verification, and validation are not supported and better tool integration is required [35].

Despite our case study belonging to cyber-physical systems, we did not consider the hardware connection as in [67]. The Ljos Java library is already the hardware platform abstraction. In [67], (low abstraction level) SysML models are mapped to realize the implementation, and model transformation plays a secondary role.

Lesson 3 recalled that refining logical models is not easy to structure in a systematic way. There is no doubt that refinement should be easier in **domain specific languages (DSL)**. However, we did not find examples of systematic MDD transformations in the context of DSL, and proposals like MDE4IoT [68] did not help in that quest. The IoT vision is quite different from ours since it is often data-intensive [69]. In our study we focused on general purpose languages for general software development and our case study includes both heterogeneous aspects (structure, dynamics, and functional) and heterogenous platform features (operating systems, networks, GUI). It does not seem easy to split the problem into several DSLs, simpler to process. In another project, we started work on defining such a language composition with Gemoc. Our DSL trials with Gemoc studio, showed us that the model part was small against the operational semantic part which was coded in Java. Further investigation is required to see whether a similar approach can be applied for MDD detailed transformation.

Threats to Validity. Finally, we discuss some threats to the validity of our experimental work.

- *How far is the method applicable to other cases ?* We use a single context (automated control system with physical devices and mobile app) and presented here only one case study. This minimal context enables the coverage of the main software design concerns (distribution, communication, persistence, concurrency, deployment, etc.) in a reduced complexity set and it is representative of what we can expect to do and the underlying concepts. Other architecture can be used, this is the generic vision of the process that makes it more applicable than dedicated approaches such as the one discussed in Section 3.1. However further experimentation of different types of applications and different PDMs must confirm our assumptions.
- *Is the method bound to UML only?* The answer is "no" because the method is generic but, of course, we need to have transformation implementations available for the modeling languages.
- *Is the method applicable to the whole software development of a system?* At this stage, the answer is "no". Only a part of the application is transformed, e.g., the GUI part is not concerned here but GUI generating facilities exist. Also, the early macro-transformations (T1, T2) appear to be simpler in their principle (mapping logical elements to platform architectural elements) than the late transformation (T3, T4), however, their integration is much more complex because they have organizational consequences on that late transformations. There is a composition issue that we have not handed yet in detail.
- *Can this method be applied in real software development process?* Theoretically, the answer is "yes" but there is a long road to this goal due to the numerous prerequisites for an assisted process: PDM models, transformation engineering (transformation implementation, mapping transformations, operators to combine transformations, verification, etc.). The next step should be a transformation case tool with libraries of (compatible) transformations.

8 Conclusion

Model Driven Development did not reach the promises given in the MDA vision [4] of software development made of successive transformations from a PIM model to a code to deploy and back the models for the next release. This is still difficult for software engineers to bypass model-based engineering, where the model is lost after the first design because the developers focus on making the source code evolve. In this article, we took a practitioner's point of view to revisit the idea, maybe mythical, of MDD as a transformation process for general purpose usage. We first assessed that despite many contributions to model transformation engineering and tools, we felt deprived of facing forward with a logical UML model. This does not mean there is a lack of approaches and tools, but that they are usually either platform dependent, especially for the commercial tools, or dedicated to a subset of logical model languages, especially domain specific languages which were out of the scope of our study. To contribute to our goal of MDD as transformations, we proposed a process of four macro-transformations, each focusing on software design concerns, and the need to abstract models from the platform to feed the transformations. We detailed the transformation in a systematic way and experimented with parts of it to show their feasibility. We illustrated both the assessment and the proposal with a simple but representative case study, a domotic application implemented in Lego EV3 mindstorms with a remote smartphone application. This case study could be used for benchmarking MDD proposals.

Our work is a first contribution to a very ambitious goal and we hope to be followed by other contributors. We try to rationalize the MDD activity, which is helpful for both software engineering practitioners and students. We do not pretend to replace developers but to assist them in building integration chains that make maintenance and evolution less expensive. Human intervention in transformations remains predominant when there are alternative choices, such as state machines or detailed design of message sequences. The process has to be more rationalized to be automated or even assisted by the means of interactive design decisions. This point remains premature in the state of our experiments.

Many tracks remain to be explored. From a theoretical point of view, the transformation processes remain yet little explored. One perspective is to design an algebra of transformations to combine them by assertion conditions. From a practical point of view, we still need to rationalize the software engineering process as a combination of decisions and experiments with a typology of transformations. From a tooling point of view, we need a transformation process factory to build specific MDD processes based on the generic design transformation process where one could pick and combine transformations and macro-transformations, a composite level of transformation patterns [64]. An interesting study is to select the best transformation tools for each kind of transformation and to combine them to achieve macro-transformations. Also, it is necessary to be able to reverse engineer the design frameworks as platform models and to combine transformations written in different languages and be interactive so that the designer influences the design choices. This last perspective goes beyond the context of MDD as a means to document platforms.

Acknowledgements

We thank the Master Students in Computer Science at Nantes University for their contribution in part of the experiments. We sincerely thank the anonymous CSIMQ reviewers for their careful reading of our manuscript and the many insightful comments and suggestions they made.

References

[1] B. Selic, "Personal reflections on automation, programming culture, and model-based software engineering," *Automated Software Engineering*, vol. 15, pp. 379–391, 2008. Available: https://doi.org/10.1007/s10515-008-0035-7

[2] J. Koskinen, “Software Maintenance Costs,” School of Computing, University of Eastern Finland, Joensuu, Finland, Tech. Rep., Apr. 2015. Available: https://wiki.uef.fi/download/attachments/38669960/SMCOSTS.pdf

[3] S. M. H. Dehaghani and N. Hajrahimi, “Which factors affect software projects maintenance cost more?” *Acta Informatica Medica*, vol. 21, no. 1, pp. 63–66, 2013. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3610582/

[4] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[5] P. André and M. E. A. Tebib, “Refining automation system control with MDE,” in *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD*, 2020, pp. 425–432. Available: https://doi.org/10.5220/0009147804250432

[6] P. André and M. E. A. Tebib, “More automation in model driven development,” in *Model and Data Engineering - 10th International Conference, MEDI 2021, Tallinn, Estonia, June 21-23, 2021, Proceedings*, ser. Lecture Notes in Computer Science, vol. 12732. Springer, 2021, pp. 75–83. Available: https://doi.org/10.1007/978-3-030-78428-7_7

[7] R. F. Paige, N. Matragkas, and L. M. Rose, “Evolving models in model-driven engineering: State-of-the-art and future challenges,” *Journal of Systems and Software*, vol. 111, pp. 272–280, 2016. Available: https://doi.org/10.1016/j.jss.2015.08.047

[8] A. Bucchiarone, J. Cabot, R. F. Paige, and A. Pierantonio, “Grand challenges in model-driven engineering: an analysis of the state of the research,” *Software and Systems Modeling*, vol. 19, pp. 5–13, 2020. Available: https://doi.org/10.1007/s10270-019-00773-6

[9] OMG, “The omg unified modeling language specification, version 2.4.1,” Object Management Group, Tech. Rep., 2011. Available: https://www.omg.org/spec/UML/2.4.1

[10] T. Weilkiens, *Systems Engineering with SysML/UML: Modeling, Analysis, Design*, ser. The MK/OMG Press. Elsevier Science, 2008. Available: https://doi.org/10.1016/B978-0-12-374274-2.X0001-6

[11] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice: Second Edition*, 2nd ed. Morgan & Claypool Publishers, 2017.

[12] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, “Empirical assessment of mde in industry,” in *2011 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 471–480. Available: https://doi.org/10.1145/1985793.1985858

[13] T. Mens and P. V. Gorp, “A taxonomy of model transformation,” *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125–142, 2006. Available: https://doi.org/10.1016/j.entcs.2005.10.021

[14] W. A. Brown, “Model driven architecture: Principles and practice,” *Software and Systems Modeling*, vol. 3, pp. 314–327, 2004. Available: https://doi.org/10.1007/s10270-004-0061-2

[15] P. Roques and F. Vallée, *UML 2 en action: De l’analyse des besoins à la conception*, ser. Architecte logiciel. Eyrolles, 2011, (in French).

[16] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Software Development Process*, ser. Object-Oriented Series. Addison-Wesley, 1999, iSBN 0-201-57169-2.

[17] S. Friedenthal, A. Moore, and R. Steiner, *A Practical Guide to SysML: Systems Modeling Language*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[18] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, 1st ed. Addison-Wesley Professional, 2012.

[19] M. Gogolla, J. Bohling, and M. Richters, "Validating uml and ocl models in use by automatic snapshot generation," *Software and Systems Modeling*, vol. 4, no. 4, pp. 386–398, 2005. Available: https://doi.org/10.1007/s10270-005-0089-y

[20] G. Shanks, E. Tansley, and R. Weber, "Using ontology to validate conceptual models," *Commun. ACM*, vol. 46, no. 10, pp. 85–89, 2003. Available: https://doi.org/10.1145/944217.944244

[21] P. André, C. Attiogbé, and J.-M. Mottu, "Combining techniques to verify service-based components," in *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development - MODELSWARD*, 2017, pp. 645–656. Available: https://doi.org/10.5220/0006212106450656

[22] J. Cabot and M. Gogolla, "Object constraint language (ocl): A definitive guide," in *Formal Methods for Model-Driven Engineering*, ser. Lecture Notes in Computer Science, M. Bernardo, V. Cortellessa, and A. Pierantonio, Eds. Springer, 2012, vol. 7320, pp. 58–90. Available: https://doi.org/10.1007/978-3-642-30982-3_3

[23] C. Raistrick, P. Francis, I. Wilkie, J. Wright, and C. B. Carter, *Model Driven Architecture with Executable UML*. Cambridge University Press, 2004, iSBN 0-521-53771-1.

[24] S. J. Mellor and M. J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*, 1st ed., ser. Object Technology Series. Addison-Wesley, 2002, iSBN 0-201-74804-5.

[25] F. Belina, D. Hogrefe, and A. Sarma, *SDL with Applications from Protocol Specification*, ser. The BCS Practitioner. Prentice Hall, 1991, iSBN 0-13-785890-6.

[26] A. Charfi, A. Schmidt, and A. 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*, ser. ECMDA-FA '09. Springer, 2009, pp. 237–252. Available: https://doi.org/10.1007/978-3-642-02674-4_17

[27] I. Perseil and L. 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*, ser. ICECCS '08. IEEE Computer Society, 2008, pp. 217–221. Available: https://doi.org/10.1109/ICECCS.2008.34

[28] OMG, "Semantics of a Foundational Subset for Executable UML Models (fUML), version 1.4," Object Management Group, Tech. Rep., Dec. 2018. Available: https://www.omg.org/spec/FUML/1.4/

[29] R. Pressman, *Software Engineering: A Practitioner's Approach*, 7th ed. McGraw-Hill, 2010.

[30] M. O. Hansen, "Exploration of UML State Machine implementations in Java," Master's thesis, University of Oslo, Norway, 2011.

[31] I. A. Niaz, J. Tanaka, and K. Words, "Mapping uml statecharts to java code," in *Proceedings of the IASTED International Conf. on Software Engineering (SE 2004*, 2004, pp. 111–116.

[32] R. Pilitowski and A. Dereziñska, "Code generation and execution framework for uml 2.0 classes and state machines," in *Innovations and Advanced Techniques in Computer and Information Sciences and Engineering*. Springer, 2007, pp. 421–427. Available: https://doi.org/10.1007/978-1-4020-6268-1_75

[33] M. Schader and A. Korthaus, "Modeling Java threads in UML," in *The Unified Modeling Language – Technical Aspects and Applications*. Springer, 1998, pp. 122–143. Available: https://doi.org/10.1007/978-3-642-48673-9_9

[34] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing, 1995.

[35] N. Kahani, M. Bagherzadeh, J. R. Cordy, J. Dingel, and D. Varró, "Survey and classification of model transformation tools," *Software and Systems Modeling*, vol. 18, pp. 2361–2397, 2019. Available: https://doi.org/10.1007/s10270-018-0665-6

[36] I. Kurtev, "State of the art of qvt: A model transformation language standard," in *Applications of Graph Transformations with Industrial Relevance*, A. Schürr, M. Nagl, and A. Zündorf, Eds. Springer, 2008, pp. 377–393. Available: https://doi.org/10.1007/978-3-540-89020-1_26

[37] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Computing Surveys*, vol. 41, no. 4, pp. 1–36, 2009. Available: https://doi.org/10.1145/1592434.1592436

[38] D. Di Ruscio, R. Eramo, and A. Pierantonio, "Model transformations," in *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012*, M. Bernardo, V. Cortellessa, and A. Pierantonio, Eds. Springer, 2012, pp. 91–136. Available: https://doi.org/10.1007/978-3-642-30982-3_4

[39] D. Lopes, S. Hammoudi, J. Bézivin, and F. Jouault, "Mapping specification in mda: From theory to practice," in *Interoperability of Enterprise Software and Applications*, D. Konstantas, J.-P. Bourrières, M. Léonard, and N. Boudjlida, Eds. Springer, 2006, pp. 253–264. Available: https://doi.org/10.1007/1-84628-152-0_23

[40] J. Cabot and E. Teniente, "Constraint support in MDA tools: A survey," in *Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006*, ser. Lecture Notes in Computer Science, A. Rensink and J. Warmer, Eds., vol. 4066. Springer, 2006, pp. 256–267. Available: https://doi.org/10.1007/11787044_20

[41] P. André and G. Ardourel, "Domain Based Verification for UML Models," in *Workshop on Consistency in Model Driven Engineering C@Mode'05*, 2005, pp. 47–62.

[42] K. Lano, *Advanced Systems Design with Java, UML and MDA*, 1st ed., ser. Computer Science. Elsevier, 2005, iSBN 0-7506-6496-7. Available: https://doi.org/10.1016/B978-0-7506-6496-7.X5000-7

[43] E. Jakumeit, S. Buchwald, D. Wagelaar, L. Dan, Ábel Hegedüs, M. Herrmannsdörfer, T. Horn, E. Kalnina, C. Krause, K. Lano, M. Lepper, A. Rensink, L. Rose, S. Wätzoldt, and S. Mazanek, "A survey and comparison of transformation tools based on the transformation tool contest," *Science of Computer Programming*, vol. 85, pp. 41–99, 2014. Available: https://doi.org/10.1016/j.scico.2013.10.009

[44] P. André, F. Azzi, and O. Cardin, "Heterogeneous communication middleware for digital twin based cyber manufacturing systems," in *Proceedings of SOHOMA*, ser. Studies in Computational Intelligence, T. Borangiu, D. Trentesaux, P. Leitão, A. G. Boggino, and V. J. Botti, Eds., vol. 853. Springer, 2019, pp. 146–157. Available: https://doi.org/10.1007/978-3-030-27477-1_11

[45] M. Brambilla, P. Fraternali, and M. Tisi, "A Transformation Framework to Bridge Domain Specific Languages to MDA," in *Models in Software Engineering*, ser. Lecture Notes in Computer Science, M. R. V. Chaudron, Ed. Springer, 2009, pp. 167–180. Available: https://doi.org/10.1007/978-3-642-01648-6_18

[46] J. Pepin, P. André, J. C. Attiogbé, and E. Breton, "Definition and visualization of virtual meta-model extensions with a facet framework," in *6th Int. Conf. MODELSWARD 2018, Revised Selected Papers*, ser. Communications in Computer and Information Science, S. Hammoudi, L. F. Pires, and B. Selic, Eds., vol. 991. Springer, 2018, pp. 106–133. Available: https://doi.org/10.1007/978-3-030-11030-7_6

[47] A. R. Van Cam Pham, S. Gérard, and S. Li, "Complete code generation from uml state machine," in *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development*, vol. 1, 2017, pp. 208–219.

[48] N. Anquetil, J. Royer, P. Andre, G. Ardourel, P. Hnetynka, T. Poch, D. Petrascu, and V. Petrascu, "JavaCompExt: Extracting architectural elements from java source code," in *2009 16th Working Conference on Reverse Engineering*, 2009, pp. 317–318. Available: https://doi.org/10.1109/WCRE.2009.53

[49] S. Rugaber and K. Stirewalt, "Model-driven reverse engineering," *IEEE Software*, vol. 21, no. 4, pp. 45–53, 2004. Available: https://doi.org/10.1109/MS.2004.23

[50] P. André, "Case studies in model-driven reverse engineering," in *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, MODELSWARD*, 2019, pp. 256–263.

[51] C. Raibulet, F. A. Fontana, and M. Zanoni, "Model-driven reverse engineering approaches: A systematic literature review," *IEEE Access*, vol. 5, pp. 14 516–14 542, 2017. Available: https://doi.org/10.1109/ACCESS.2017.2733518

[52] H. Brunelière, J. Cabot, G. Dupé, and F. Madiot, "MoDisco: A model driven reverse engineering framework," *Information and Software Technology*, vol. 56, no. 8, pp. 1012 – 1032, 2014. Available: https://doi.org/10.1016/j.infsof.2014.04.007

[53] F. R. Golra, F. Dagnat, R. Bendraou, and A. Beugnard, "Continuous process compliance using model driven engineering," in *Model and Data Engineering - 7th International Conference, MEDI 2017*, ser. Lecture Notes in Computer Science, Y. Ouhammou, M. Ivanovic, A. Abelló, and L. Bellatreche, Eds., vol. 10563. Springer, 2017, pp. 42–56. Available: https://doi.org/10.1007/978-3-319-66854-3_4

[54] J. Aranda, D. E. Damian, and A. Borici, "Transition to model-driven engineering," in *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012*, ser. Lecture Notes in Computer Science, R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, Eds., vol. 7590. Springer, 2012, pp. 692–708. Available: https://doi.org/10.1007/978-3-642-33666-9_44

[55] G. Sebastián, J. A. Gallud, and R. Tesoriero, "Code generation using model driven architecture: A systematic mapping study," *Journal of Computer Languages*, vol. 56, p. 100935, 2020. Available: https://doi.org/10.1016/j.cola.2019.100935

[56] A. Sindico, M. D. Natale, and A. L. Sangiovanni-Vincentelli, "An industrial system engineering process integrating model driven architecture and model based design," in *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012*, ser. Lecture Notes in Computer Science, R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, Eds., vol. 7590. Springer, 2012, pp. 810–826. Available: https://doi.org/10.1007/978-3-642-33666-9_51

[57] F. Ciccozzi, I. Malavolta, and B. Selic, "Execution of uml models: a systematic review of research and practice," *Software & Systems Modeling*, vol. 18, no. 3, pp. 2313–2360, 2019. Available: https://doi.org/10.1007/s10270-018-0675-4

[58] E. Planas, J. Cabot, and C. Gómez, "Lightweight and static verification of uml executable models," *Comput. Lang. Syst. Struct.*, vol. 46, pp. 66–90, 2016. Available: https://doi.org/10.1016/j.cl.2016.07.002

[59] M. Tisi, F. Jouault, Z. Saidi, and J. Delatour, "Enabling ocl and fuml integration byźtransformation," in *Proceedings of the 12th European Conference on Modelling Foundations and Applications*, vol. 9764. Springer, 2016, pp. 156–172. Available: https://doi.org/10.1007/978-3-319-42061-5_10

[60] F. Ciccozzi, "On the automated translational execution of the action language for foundational uml," *Software & Systems Modeling*, vol. 17, no. 4, pp. 1311–1337, 2018. Available: https://doi.org/10.1007/s10270-016-0556-7

[61] S. Guermazi, J. Tatibouet, A. Cuccuru, E. Seidewitz, S. Dhouib, and S. Gérard, "Executable modeling with fuml and alf in papyrus: Tooling and experiments," in *Proceedings of the 1st International*

Workshop on Executable Modeling in (MODELS 2015), vol. 1560, 2015, pp. 3–8. Available: http://ceur-ws.org/Vol-1560/paper1.pdf

[62] F. A. Somogyi and M. Asztalos, "Systematic review of matching techniques used in model-driven methodologies," *Software and Systems Modeling*, vol. 19, no. 3, pp. 693–720, 2020. Available: https://doi.org/10.1007/s10270-019-00760-x

[63] J. Bruel, B. Combemale, E. Guerra, J. Jézéquel, J. Kienzle, J. de Lara, G. Mussbacher, E. Syriani, and H. Vangheluwe, "Comparing and classifying model transformation reuse approaches across metamodels," *Softw. Syst. Model.*, vol. 19, no. 2, pp. 441–465, 2020. Available: https://doi.org/10.1007/s10270-019-00762-9

[64] K. Lano, S. Kolahdouz-Rahimi, S. Yassipour-Tehrani, and M. Sharbaf, "A survey of model transformation design patterns in practice," *Journal of Systems and Software*, vol. 140, pp. 48–73, 2018. Available: https://doi.org/10.1016/j.jss.2018.03.001

[65] A. Van Deursen, E. Visser, and J. Warmer, "Model-driven software evolution: A research agenda," in *Proceedings Int. Ws on Model-Driven Software Evolution held with the ECSMR'07*, 2007.

[66] F. Ihirwe, D. D. Ruscio, S. Mazzini, P. Pierini, and A. Pierantonio, "Low-code engineering for internet of things: A state of research," *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2020. Available: https://doi.org/10.1145/3417990.3420208

[67] B. Vogel-Heuser, D. Schütz, T. Frank, and C. Legat, "Model-driven engineering of manufacturing automation software projects – a sysml-based approach," *Mechatronics*, vol. 24, no. 7, pp. 883–897, 2014, 1. Model-Based Mechatronic System Design 2. Model Based Engineering.

[68] F. Ciccozzi and R. Spalazzese, "MDE4IoT: Supporting the internet of things with model-driven engineering," in *Intelligent Distributed Computing X – Proceedings of the 10th International Symposium on Intelligent Distributed Computing – IDC 2016*, ser. Studies in Computational Intelligence, C. Badica, A. E. F. Seghrouchni, A. Beynier, D. Camacho, C. Herpson, K. V. Hindriks, and P. Novais, Eds., vol. 678, 2016, pp. 67–76. Available: https://doi.org/10.1007/978-3-319-48829-5_7

[69] F. Corradini, A. Fedeli, F. Fornari, A. Polini, B. Re, and L. Ruschioni, "X-IoT: a model-driven approach to support iot application portability across iot platforms," *Computing*, vol. 105, pp. 1981–2005, 2023. Available: https://doi.org/10.1007/s00607-023-01155-z

[70] M. H. Orabi, A. H. Orabi, and T. C. Lethbridge, "Concurrent programming using umple." in *MODELSWARD*, 2018, pp. 575–585.

[71] A. Forward, O. Badreddin, T. C. Lethbridge, and J. Solano, "Model-driven rapid prototyping with Umple," *Software: Practice and Experience*, vol. 42, no. 7, pp. 781–797, 2012. Available: https://doi.org/10.1002/spe.1155

Appendices A–F

A Selected Tools for Comparison

This appendix provides information on the tools selected for the comparison in Section 3.1.

- **Papyrus**[32] Code generation in Papyrus includes behaviors to operations with an incremental that overrides the code generation. Starting from version 3.0, the code generator engine of Papyrus extends the IF-Then-Else constructions of programming languages with multi thread-based *concurrency* and *state machines hierarchy* support [47]. It brings also some improvements in terms of the generated code *portability*, event processing speed, and optimization. Papyrus generates only *C++* code from STDs, which makes it difficult to conduct our experiment when the generated programs are deployed on Android. The context of state machine elements is defined by the associated class diagram, which provides a complete XML model that can be used to define specific transformation rules, with the different transformation tools (ex: ATL, Java, ect.).
- **Modelio** In addition to the fact that Modelio supports code generation for CDs and STDs, it provides round trip functionalities which ensures synchronization between code and model. Unlike, e.g., Visual Paradigm; it makes the difference between the methods managed by Modelio and the others. A managed method is automatically generated for each release. A simple (not managed) method is under the responsibility of the developer. Fork and Join elements are not among the state diagram features in Modelio[33].
- **Umple** supports code generation from state machines to Java, PHP, and C++ codes [70]. In Umple, CD operations are not associated to STDs. Umple has a solution to the problems of round-trip by allowing the embedding of arbitrary code directly into the model [71]. Due to the fact that code generation does not support pseudo states (Fork, History, etc.), Umple cannot design the behavior of embedded industrial systems.
- **StarUML** is much more oriented to support modeling at educational and professional institutes. it provides rich modeling features but scarce support for code generation (only CD). Reverse engineering exists for programming languages including Java, C#, and C++ via open source extensions.
- **Visual Paradigm** is rich in standards and features. It supports the generation of CDs and STDs in Java source code but also in C++ or VB.net. Its *round-trip engineering* feature synchronizes the code and the model. We did not have access to the generated code to estimate the programming effort to add communication between state machines.
- **UModel** belongs to the commercial tools. We have tested its trial version which generates a full Java, C#, and Visual Basic executable code from complete STD[34]. Umodel provides a model-code synchronization through round trip engineering. Model interoperability is ensured in this tool through supporting XMI. The CSM is not available in UModel.
- **IBM Rational Rhapsody** inspired by the authors of UML, appears to be the most complete tool. The code is updated automatically in a parallel view of the model. One can edit the code directly, the diagrams will stay synchronized. It provides a full code generation from combined Class and STD diagrams. However, no information confirms that the tool expresses communication between multi STDs. Again, we did not have access to the generated code to estimate the manual programming part.

[32] **Papyrus** and **Modelio** belong to the *Eclipse Modeling* ecosystem with an active community. In this category, also the Obeo tools, UML Designer and Acceleo, or the Polarys project including Papyrus and Topcased can be mentioned

[33] `https://www.sinelabore.de/doku.php?id=wiki:landing_pages:modelio`

[34] https://www.altova.com/umodel

- **Yakindu**[35] is a statecharts-based modeling and simulation tool proposed by Itemis. It generates a detailed implementation for one STD only which is independent from CDs. Despite the fact that Yakindo does not ensure a high synchronization level between code and models, it can be considered as a powerful tool to design complex behaviors. In version 4.0, the CSM code generation makes it usable for control systems.
- **Framework for eXecutable UML (FXU)**[36] supports execution of concurrent state machines which can specify the behavior of many different objects. Regions of orthogonal states are executed concurrently as well. Transitions across vertices are triggered by events [32]. In FXU, CD and STD are related to each other. Class elements create a context for STDs. FXU, is based on IF-THEN-ELSE approach to generate Java code from state machines. We found no way to export the source models with FXU, so we cannot define specific transformation rules using these tools

B Case Study Description

To lead the experimentation, we provided a *Software Requirements Specification (SRS)* and a Logical Model (LM) to the students. The SRS describes the user requirements, the constraints, the technical targets and methodological guidelines. Requirements are prioritized in an agile incremental process. Not all the system functionalities are described here, for instance, the following requirements are out of the scope of the current experimentation: user management for the remote control and additional devices such as a warning light, motion detectors, and safety and security constraints. Only an excerpt of the logical model is provided here, as a collection of UML diagrams. The class diagram of Figure B1 describes the object types and operations.

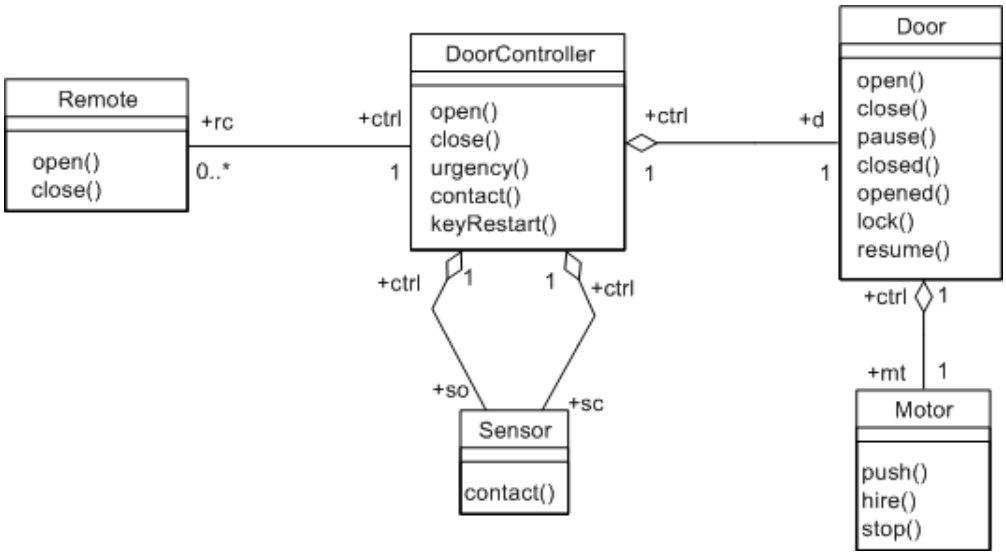

Figure B1. Analysis Class diagram - garage door

The system operates as follows (Figure B2). Suppose the door is closed. The user starts opening the door by pressing the `open` button on his remote control. It can stop the opening by pressing the `open` button again, the motor stops. Otherwise, the door opens completely and triggers the open sensor `so`, and the motor stops. Pressing the `close` button close the door if it is (partially or completely) open. Closing can be interrupted by pressing the `close`

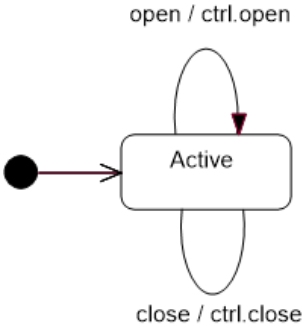

Figure B2. Remote STD

[35] https://www.itemis.com/en/yakindu/
[36] http://galera.ii.pw.edu.pl/ adr/FXU/

button again, the motor stops. Otherwise, the door closes completely and triggers a closed sensor `sc`, the motor stops. At any time, if someone triggers an emergency stop button located on the wall, the door will lock. To resume we turn a private key in a lock on the wall. The remote control, when activated, reacts to two events (pressing the open button or pressing the close button) and then simply informs the controller which button has been pressed (Figure B2).

The STD in Figure B3 describes the behavior of the door controller. The actions on the doors are transferred to the engines by the door itself. User stories can be defined through requirement

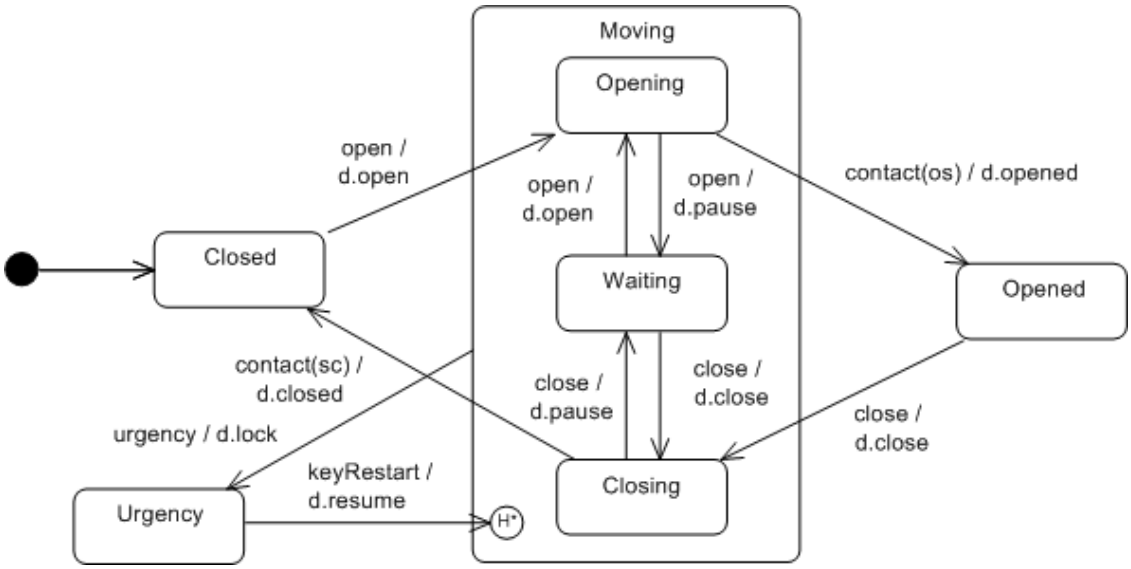

Figure B3. Door controller State diagram – garage door

analysis and refined in the logical view of the analysis activity to be later reused as test cases in model or code verification. As an example, the sequence diagram in Figure B4 describes the collaboration of the door components when opening the door. Door actions are transferred to the motors by the door itself.

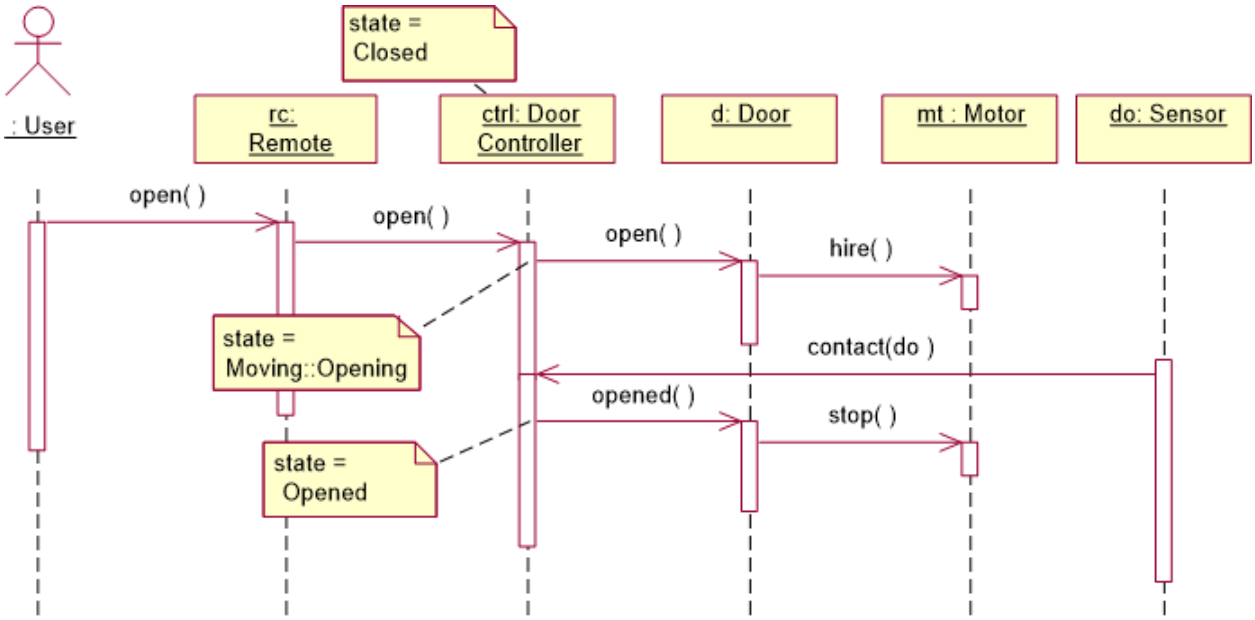

Figure B4. Opening Sequence diagram – garage door

The verification of logic models includes at least static analysis and type checking. These can be designed as a transformation process [41] where advanced verification of properties requires *model*

checking for communications, *theorem proving* for functional contract assertions, and testing for behavioral conformance [21]. Most of them require translation to formal methods. In the following, before refining models to code, we assume model properties to be verified.

C T3 STD Transformation with ATL

STDs are assumed to be simple automata: no composite state, no time, no history. Also, the main restriction is that state machine inheritance through class inheritance is not allowed here because the UML rules have different interpretations and vary from one tool to another. Most of them do not consider STD inheritance. Code style conventions have been determined (e.g., the elements Region and StateMachine have the name of their class) which makes it easier to write the transformation rules.

The *UML2Java* transformation is structured in three main steps:

1. Generate a Java model that has exactly the same UML-Papyrus model structure. In this line, Figure C1 describes the ATL rule building a target XMI model with respect to Papyrus specification. The model2model rule builds the main structure of the generated XMI model. This model, called uml_java (MM1!Model), has the same name as the source model and contains all the instances of the UML source model that conform to Java.

```
rule model2model {
    from
        uml : MM!Model
    to
        uml_java : MM1!Model (
            name <- uml.name,
            packageImport <- MM!PackageImport.allInstances(),
            packagedElement <- MM!Class.allInstances().union(MM1!Association.allInstances()),
            profileApplication <- MM!ProfileApplication.allInstances()
        )
}
```

Figure C1. Model to model transformation -basic rule

2. Once the main XMI structure of the Java target model is built, the second step copies all the existing elements from the source model that refer to UML-Java Profile such as Packages (MM!PackageImport), Classes (MM!Class), Attributes (MM!Property), and Methods(MM!Operation). As described in Figure C2, after a deep analysis of the XMI file, four main elements could be copied directly to the Java target model: Package, Class, Property, and Operation. For each element, an ATL matched rule is defined.
3. For each UML class (MM!Class) containing a subsection (MM!StateMachine) or possibly MM!Activity), we carried out a set of ATL rules (Figure C3) to transform this behavior into UML-Java. Among the alternatives given in transformation `[T3.1]`, we chose the *State* pattern because it is straightforward. According to the pattern definition [34], the corresponding Java elements will be generated:
   (a) A Java *Interface* representing the STD of each object,
   (b) The Java class should *implement* the generated stateMachine *interface*,
   (c) For each context class (1) a private attribute references the STD and (2) a public method setState () defines the current object state.
   (d) We generate a path variable _currentState and a memory variable _previousState if the state diagram holds a Pseudostate element of type deepHistory. Both variables are Property elements typed by the enumeration type. To initialize the current state, a child element OpaqueExpression is added, with two parameters: 'language' which takes the value 'JAVA' and 'body'. The body parameter is initialized with the concatenation of the enumeration

```
rule Package {

    from
        uml: MM!PackageImport
    to
        uml_java: MM1!PackageImport(
           importedPackage <- MM1!Profile.allInstances()
        )
}
rule Class {
    from
        uml: MM!Class
    to

        uml_java: MM1!Class (
            name <- uml.name,
            ownedAttribute  <-  uml.getProperties(uml).union(thisModule.name(uml)),
            ownedOperation  <-  uml.getOperations(uml)
        )
}
rule Attribute{
    from
        uml : MM!Property
    to
        uml_java : MM1!Property (
            name <- uml.name,
            type <- uml.type,
            association <- uml.association
        )
}
rule Operation {
    from
        uml : MM!Operation
    to
        uml_java : MM1!Operation (
            name <- uml.name
        )
}
```

Figure C2. From UML elements to Java elements

name and the initial state. The initial state is found by retrieving the target state of the transition having the initial state as the source state.

(e) To determine the behavior of the operations, for each operation used as a trigger in a state machine, we create a condition **switch** in the method implementing the operation. To fulfill the condition, we retrieve the source state and target state of all transitions that trigger the function. The source states correspond to the possible cases for the change of the state and the target states correspond to the new value of the current state. We add in each case the **switch**, the exit action of the start state, and the input action of the arrival state if any.

```
lazy rule stateMachine2Java{

   from
     uml: MM!Class (uml.hasBehavior(uml))
   to
     class: MM!Class(
        name <- uml.name,
        ownedAttribute  <-  uml.getProperties(uml)
                               .union(privateAttribute, currentState, previousState),
        ownedOperation  <-  uml.getOperations(uml).uml(method)
     ),
     state_interface: MM!Interface(
        name <- 'I'+uml.name+'StateMachine'
     ),
     implements: MM!InterfaceRealization(
        client <- class,
        supplier <- state_interface
     ),
     privateAttribute: MM!Property(
         name <- uml.name,
         type <- 'I'+stateInterface.name+'StateMachine',
         visibility <- 'private'
     ),
     currentState: MM!Property(
         name <- 'currentState',
         type <- 'I'+stateInterface.name+'StateMachine',
         visibility <- 'private'
     ),
     previousState: MM!Property(
        name <- 'currentState',
        type <- 'I'+stateInterface.name+'StateMachine',
        visibility <- 'private'
     ),
     methods: MM!Operation(
        visiblity <- 'public',
        name <- 'setState'
     ),
}
```

Figure C3. From STD to Java elements

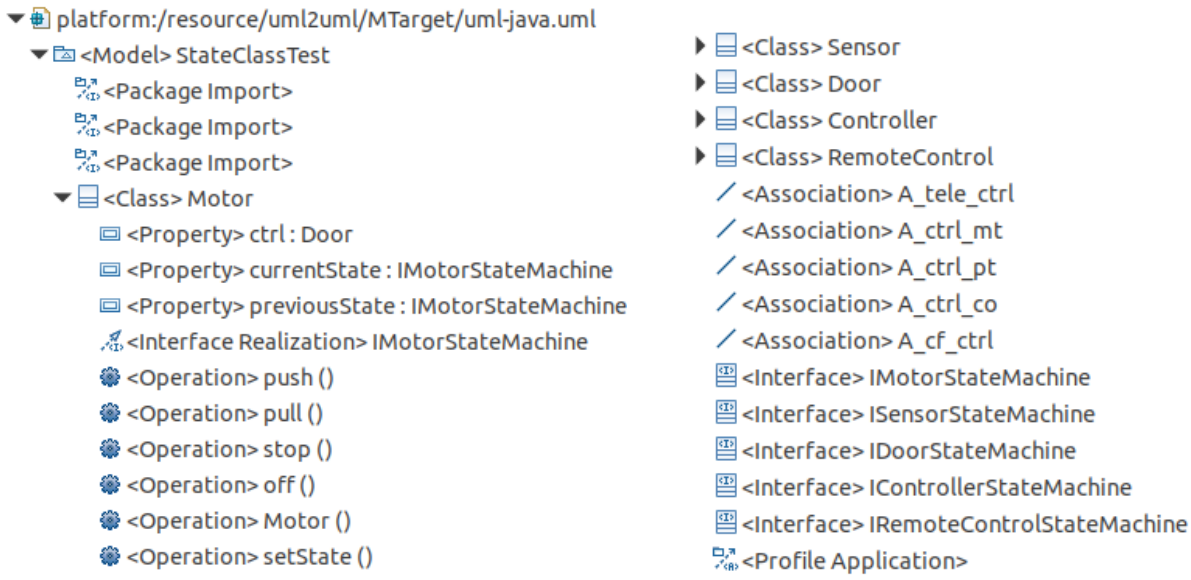

Figure C4. The Java elements generated for the garage door

D T4 Source Code Transformation with ATL

We designed an ATL transformation engine, comprised of a set of sub-transformation rules, to parse the XMI model and generate the corresponding Java code. This process is explained in the subsequent steps, starting with the examination of StateMachine elements and the overarching structure of the resulting Java element (see Figure C3), and progressing to the intricacies of each generated segment of code, including classes (see Figure D1), properties (see Figure D2), and methods (see Figure D3).

1. *To generate the source code structure* in M2T transformations, ATL provides the concept of helpers (methods) to parse the XMI model. Each helper generates a piece of code that conforms to the Java grammar (syntax). The ATL helper in Figure D1 organises the parse-generate process by calling sub-rules.

```
helper context MM!Model def : GenerateJavaCode() : String =
     let  classes : MM!Class = self.ownedType->select(c | c.oclIsTypeOf(MM!Class)) in
        '/* \n'+
        ' * Automatically generated Java code with ATL \n'+
        '   Authors: Mohammed TEBIB & Pascal Andre \n'+
        ' */ \n'+
         classes->iterate(it; Class_Code: String = ''|Class_Code
           + thisModule.getImport(it.name) + '\n'
           + it.visibility +' class '+ it.name
           + if it.hasBehavior(it) then ' implements ' + 'I'+it.name+'StateMachine ' else '' endif
           + '{\n  '
           +'  //attributes \n'
           +'  ' + it.GenerateAttributes(it)
           +'\n\n    //methods \n'
           +'  ' + it.GenerateMethods(it)
           + '\n} \n'
           + it.GenerateInterfaces(it)
         )
  ;
```

Figure D1. ATL transformation rule for classes

- The GenerateJavacode() helper presented in Figure D1 parses every class, presents it in XMI model (UML-Java), and generates the Java class code structure. It is completed by calling other helpers: (i) GenerateAttributes () to generate the attributes corresponding to each class, (ii) GenerateMethods() to generate only the signature of each method, this helper could be extended in the future to generate the method body from the associated activity diagram, and (iii) GenerateInterfaces () to generate the modeled interfaces if such there exist.
- The GenerateAttributes () helper parses all classes and generates all information related to the attributes: visibility, name, and type (see Figure D2).

```
--A method to generate the attributes of a given class
helper context MM!Class def : GenerateAttributes(x:MM!Class) : String =
   let  attributes : MM!Property = x.ownedAttribute->
             select(a | a.oclIsTypeOf(MM!Property)) in
                attributes->iterate(it; att: String = ''| att + '  '
             + thisModule.addAdapterAttributes(x.name)  + '\n'
             + it.visibility + ' '
             + if it.isStatic.toString()='false' then ' ' else 'static  ' endif
             + it.type + ' '
             + it.name + ';'
             + '\n  '
);
```

Figure D2. ATL transformation rules for attributes

- The GenerateMethods() helper generates the method signature: visibility, returned type, name, and parameters (see Figure D3).

```
--A method to generate the methods of a given class
helper context MM!Class def : GenerateMethods(x:MM!Class) : String =
 let  methods : MM!Operation = x.ownedOperation->
    select(a | a.oclIsTypeOf(MM!Operation)) in
      methods->iterate(it; att: String = ' '|  att + '  '
       + thisModule.mappingmethods(x.name, it.name) + '\n'
       + it.visibility
       + if(it.isStatic.toString()='true') then 'static '
         else ' ' endif
       + if(it.isAbstract.toString()='true') then 'abstract '
         else ' ' endif
       + if(it.type.toString()<>'OclUndefined') then
          if(it.type.toString().substring(1, 3)='<un') then
           it.type.toString().substring(11, it.type.toString().size())
          else if (it.type.toString().substring(1, 3)='IN!') then
                it.type.toString().substring(4, it.type.toString().size())
               else '' endif
          endif
         else 'void' endif
       + '  '+it.name + '('
       + '){\n'
       + '     }\n   '
    )
;
```

Figure D3. ATL transformation rules for methods

Figure D4 shows the list of Java files generated for the Motor model class.

Figure D4. The list of the generated classes

E Adaptation Pattern Implementation

We describe here an API mapping by adaptation and illustrate the approach in the case of Motor.

In the example in Figure E1, the (model) class Motor delegates its method calls to the EV3Large RegulatedMotor.

Based on the specification of a simplified *Adapter Pattern* presented in Figure 8 in Section 5 of the article, we delegate to Adapter instances every model class that maps to one existing framework class taking into account the following parameters: (i) *Import*: the packages of each class depending on the *className* and *packageName* values, (ii) *MethodCall*: represents the API calls to

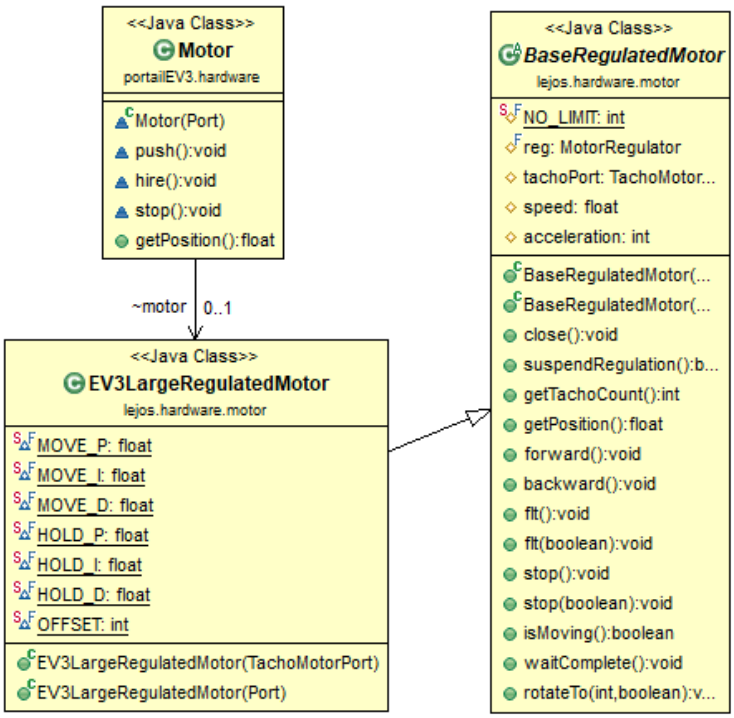

Figure E1. Class Mapping by Adaptation of the Motor Class

perform on the defined *operationName* existing in the class specified by the *className* attribute, and (iii) *Attribute* defines API references declaration. Based on these parameters, our ATL transformation engine will generate the appropriate Java code mapped to the lejos PI using three ATL helpers presented in Figure E2.

```
helper def : addAdapterAttributes(class:String) : String =
 let attributes: Sequence(MM1!Attribute)= MM1!Attribute.allInstances() in
       attributes->iterate(it; attr : String  = '' |
        if(it.className = class) then
          attr + it.attributeDeclaration
        else '' endif)
;
helper def : getImports(s: String) : String =
 let imports: Sequence(MM1!Import)= MM1!Import.allInstances() in
       imports->iterate(it; import : String  = '' |
         if(it.className=s) then
           import + it.packageName
         else '' endif
       )
;
helper def: mappingmethods(class: String, operation: String) : String =
 let instructions: Sequence(MM1!MethodCall)= MM1!MethodCall.allInstances() in
      instructions->iterate(it; cmd : String  = '' |
        if(it.className=class and it.operationName = operation) then
           cmd + it.instruction
        else '' endif)
;
```

Figure E2. ATL helper to generate adapted attributes

The addAdapterAttributes helper adds for each class the specific attributes referencing objects in the corresponding *Lejos* framework. The getImports ATL helper maps each class to one of the frameworks. For API calls, the helper mappingMethods takes as input a couple of parameters representing the name of the class and the name of the operation to be mapped. Note that

addAdapterAttributes , mappingMethods() and getImports () helpers will run based on the properties file that is defined as an instance of the adapter model.

Listing 1.1 presents the content of such a property file in the case of Motor.

Listing 1.1. Instance of Adapter Model

```
<?xml version="1.0" encoding="UTF-8"?>
<Adapter xmi:version="2.0"
    xmlns:xmi="http :// www.omg.org/XMI">
  <methods  className="Motor"
    operatioName="push"  Instruction ="EV3LargeRegulatedMotor.forward();" />
  <methods className="Motor"
    operatioName="hire"   Instruction ="EV3LargeRegulatedMotor.backward();" />
  <methods className="Motor"
    operatioName="stop"   Instruction ="EV3LargeRegulatedMotor.stop();"  />
  <methods className="ContactSensor"
    operatioName="contact"   Instruction ="EV3TouchSensor.fetchSample();" />
  <methods className="MotionDetector"
    operatioName="contact"   Instruction ="EV3UltrasonicSensor.fetchSample() ;"  />
  < attributes   className="Motor"   attributeDeclaration =" private
      EV3LargeRegulatedMotor ev3LargeRegulatedMotor;" />
  < attributes   className="ContactSensor"
     attributeDeclaration  =" private   EV3TouchSensor ev3TouchSensor;" />
  < attributes   className="MotionDetector"
     attributeDeclaration  =" private   EV3UltrasonicSensor  ev3UltrasonicSensor ;"  />
  < attributes   className="Communication"
     attributeDeclaration  =" private   lejos .remote.nxt  nxt;"  />
  <imports  className="Motor"
    packageName="lejos.hardware.motor.EV3LargeRegulatedMotor;" />
  <imports  className="ContactSensor"
    packageName="lejos.hardware.sensor.EV3TouchSensor;" />
  <imports  className="MotionDetector"
    packageName="lejos.hardware.sensor.EV3UltrasonicSensor"  />
  <imports  className="Communication"
    packageName="lejos.remote.nxt.BTConnection;" />
</Adapter>
```

The result of the above adapter transformation in the simple case of class Motor is given in Listing 1.2. It implements direct mapping for class, imports, and method calls.

Listing 1.2. Instance of Adapter Model

```
/*
 * Automatically generated Java code with ATL
   @author Mohammed TEBIB & Pascal Andre
 */
import lejos.hardware.motor.EV3LargeRegulatedMotor;

public class Motor implements IMotorStateMachine {
    //attributes
    private EV3LargeRegulatedMotor ev3LargeRegulatedMotor;
    public Door ctrl;
    private IMotorStateMachine motorState;

    //methods
    public void push(){//delegates to EV3LargeRegulatedMotor
        EV3LargeRegulatedMotor.forward();
    }

    public void hire(){//delegates to EV3LargeRegulatedMotor
        EV3LargeRegulatedMotor.backward();
    }

    public void stop(){//delegates to EV3LargeRegulatedMotor
        EV3LargeRegulatedMotor.stop();
    }

    public void Motor(){//to be completed
    }

    public void setState(IMotorStateMachine motorState){
        this.motorState=motorState;
     }
}
```

F AgileJ Diagram

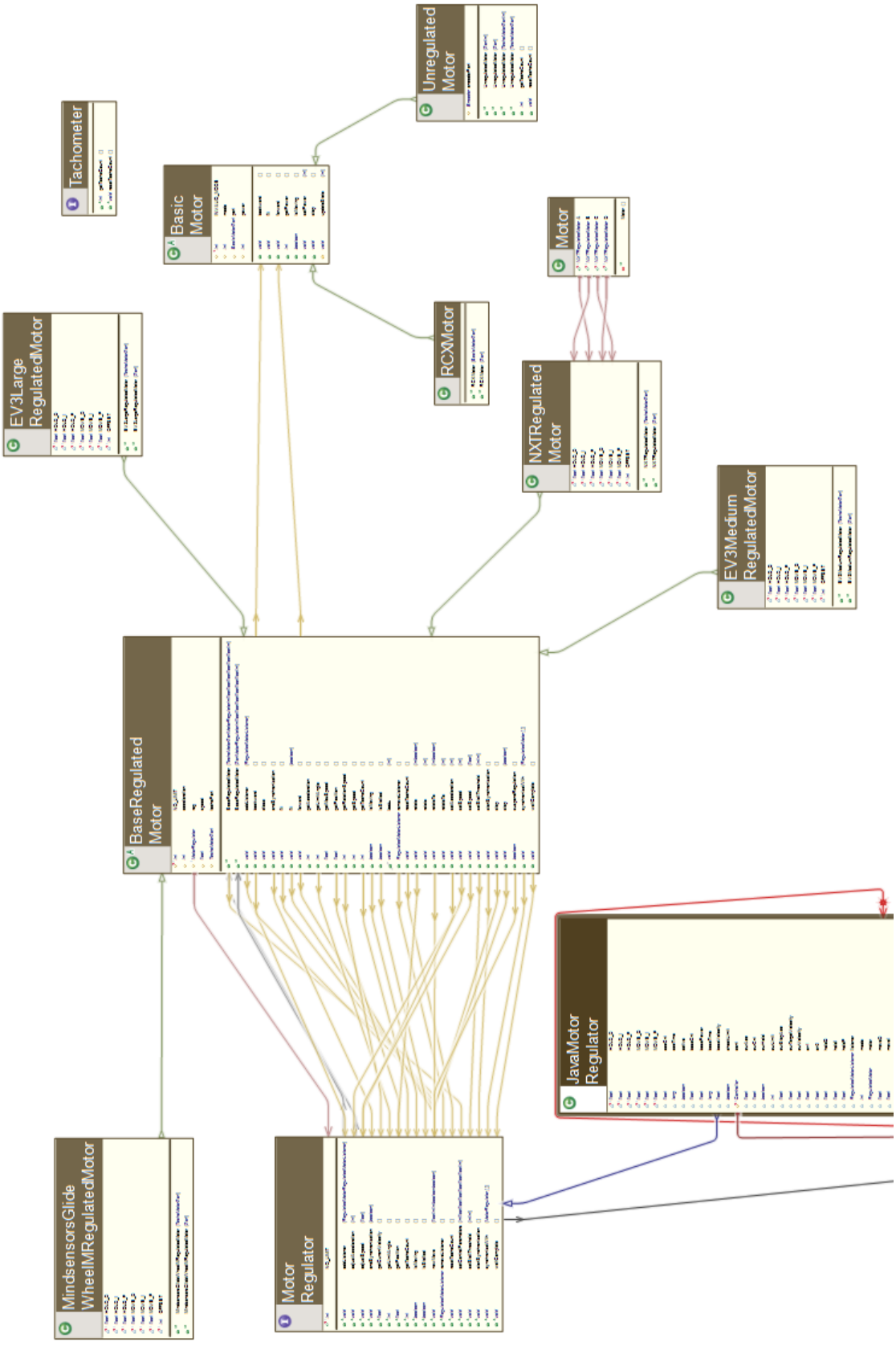

Figure F1. Applying AgileJ RE process on *Motor* package of Lejos Library