SoSym manuscript No.
(will be inserted by the editor)

Assistance in Model Driven Development

Toward an automated transformation design process

Pascal André · Mohammed El Amin Tebib

Received: date / Accepted: date

Abstract Model engineering aims to shorten the development cycle by focusing on abstractions and partially automating code generation. We longly lived in the myth of automatic Model Driven Development with promising approaches, techniques and tools. Describing models should be the main concern in development also with 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 stepwise transition from the model to the code to reduce the time between analysis model and implementation. The current state of practice requires method 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 conduct experiments within a simple case study in software automation systems. It is both representative and scalable. The models are written in UML (or SysML) and programs deployed on Android and Lego EV3. We report the lessons learnt from experimentations for future community work.

Keywords Model Driven Engineering · Communication Refinement · Statecharts Refinement · Model Transformation process · Automation Control Systems

Pascal André
LS2N CNRS UMR 6004 - University of Nantes
2 rue de la Houssinière F-44322 Nantes Cedex France
E-mail: pascal.andre@ls2n.fr

Mohammed El Amin Tebib
Davidson Consulting Paris
40 Rue Fanfan la Tulipe 92100 Boulogne-Billancourt France
E-mail: mohammed.tebib@outlook.com

1 Introduction

Since the advent of MDA, we longly lived the myth of automatic Model Driven Development (MDD) with promising maintenance cost reduction. 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. Twenty years later, this goal has not been reach despite numerous contributions on techniques and tools. The software must not only be of quality in terms of reliability and performance, but also be able to evolve and fit to new needs and constraints. The development and maintenance life cycle must support continuous evolution but software maintenance costs, which traditionally accounted for 70% of the total cost of the software, still increase [33,18]. We revisit the subject of MDD through the prism of experimentationand open mindness, including Model-Driven Reverse Engineering (MDRE) concerns, leading to the field of Model driven engineering (MDE).

We are convinced that MDE is gainful to develop long-term software systems. Reasoning to verify the system properties can happen at the model level (methods and tools exist for this) but at the code level it is hard, due to the complexity inherent to implementation and deployment details. Code generation from operational models exists for many years by the means of compilers or grammar-based generators such as Antlr, XML and JSON parsers or recently Xtext. But the generation of code from higher level models of abstraction, resulting from the analysis or the software design, remains 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 example).

The general problem is how to efficiently develop and maintain applications from abstract heterogeneous models (including structural, dynamic and functional aspects of the modelled system). Typically, we can illustrate the case of a UML model with a class (or component) diagram for the static part, Statecharts for dynamics and activity diagrams supplemented by an action language for computations. The application is a system distributed over several devices. We do share the vision given in [54] that reduces MDE to two main ideas: raising the level of abstraction and raising the degree of computer automation. In [5] we highlighted the problems by comparing the approaches and drew a vision for a generic MDE transformation process. This article extends [5] by providing deeper sharpness on the state of practice, specification details for practitioners and application principles for automation and assistance.

This work is a practical contribution to the *engineering practice* challenges of [42] and locates in the social and domain challenges of [12]. To the best of our knowledge, there are no proposals of MDD process as a process of model transformations. Our goal is to introduce an assisted MMD process with the following contributions: (i) a frame to reason about MDE in software development in terms of transformations, (ii) detailed specifications for automatable transformations, illustrated by experiments, (iii) a case study for benchmarking collaborative contributions, (iv) an integrated vision of MDE and reverse engineering and (v) a combination of tools to achieve the transformations process. We focus more on methodological issues than on technical ones. We conduct trial and error experiments, some with students, starting from models written with expressive modelling languages and we target programmable controllers (*e.g.* Lego EV3) remotely controlled by an Android client. This is not a systematic study but lessons from experience. We implement and compare three ways to get 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 experiments illustrate the complexity of the task. The lessons learnt from these experimental works will open tracks for future work.

The article is structured as follows. Section 2 introduces the context elements and we present the illustrating case study, a simple home automation (domotics) system. The next sections relate the experimentations and limitations of the manual forward development (Section 3) and the automatic code generation (Section 4). We draw a stepwise refinement process in Section 5 by the means of macro-transformation where the complexity yields in the process which must be adaptable and configurable. Implementation details of the macro-transformations are given in Section 6 and illustrated by experimentations on the case study. To fill the missing technical model, we explore reverse engineering in Section 7. Lessons learnt are discussed with related works in Section 8. Finally, Section 9 summarises the contribution and draws open perspectives that go beyond model transformation topics.

2 Background

Consider forward engineering. The goal 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 that cover many paradigm of the UML notation but also on non-functional requirements. In particular, we are interested in programmable controllers having an effective execution environment that take into account operating, safety and performance constraints [49]. Some properties are general (safety, liveness), others are related to the environment or the system itself (energy, dangerousness, quality of service...). Moreover the focus can be extended to software/hardware issues and properties. Note however that the results presented here are definitely not restricted to such systems.

2.1 Software development

The software development in forward engineering starts from requirements analysis to execution code deployment. We do not consider here the project management part or the production part (DevOps). Among the development process models, we chose the 2 Track Unified process (2TUP) (also called ”Y”) of Fig. 1 coming from [50] because it really puts forward the role of design as the central activity that combines an analysis model with technical support. 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 make sense from a MDE point of view when seeing the development as a (complex) model transformation. The development cycle is implicitly iterative and incremental.

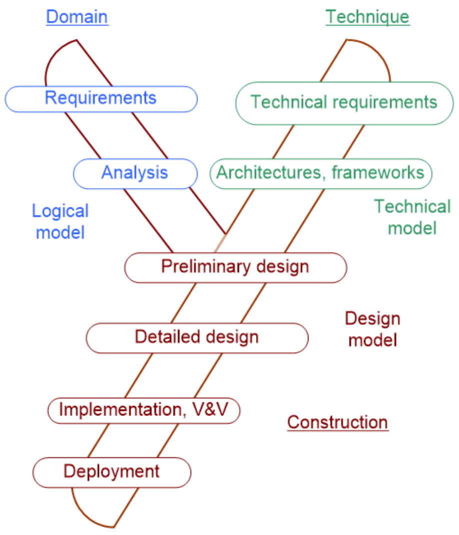


Fig. 1 2TUP Unified Process

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

1. the modelling and simulation level, where the individual and collective behaviours are described and where the constraints are analysed (digital twin). Specifications are written in modelling languages such as UML [26], SysML [22], Kmelia [38], AADL [20]) and associated verification or simulation tools, etc. The models at this level will be called *logical models* in the sense that the technical details are not yet given. As illustrated by Fig. 1, the analysis model is plunged into a technical model to build a design model.
2. the operational level, where the controls of the physical devices are implemented. This is achieved using communication tools based on programmable logic controllers (PLCs), robots, sensors and actuators.

Intermediate steps can be processed to reach the implementation level in the spirit of the model-driven development (MDD) [10] and software product lines (Software Product Lines) [8].

In MDD, it is essential to ensure the model correctness before starting the process of transformations and code generation [38]. This reduces the high cost of late detection of errors [24,55]. Whatever is the modelling language, the models are considered to be sufficiently detailed to be made executable[1]. This allows to verify properties on the models, using *theorem proving*, *model checking* or *model testing* techniques [1].

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

2.2 Case Study

A case study is simple, to be easily understood, and complete to cover a representative set of software development artefacts including object communications that goes beyond the simple procedure call and object protocols ordering the API method invocation. We chose a simple control systems in cybernetics and selected a simplified home automation equipment (domotic): a garage door including hardware devices (remote control, door, PLC, sensor, actuators ...) and the software that drives these devices[2].

In cybernetics, SysML [62] is recommended for PLC design *e.g.* the detailed SysML model of a transmission control for Lego NXT[3] has been simulated by the Cameo tool. However we chose UML because it belongs to the student's program and because the UML modelling ecosystem is rich. We provide a *Software Requirements Specification (SRS)* and a logical model (LM) of the case given in the UML notation *i.e.* the class diagram of Fig. 2 including the operation signature. Note that the SRS is larger than the LM ; it includes for example user management for the remote, additional devices such a warning light, motion detectors, safety and security constraints but also requirement priority list for an agile incremental development.

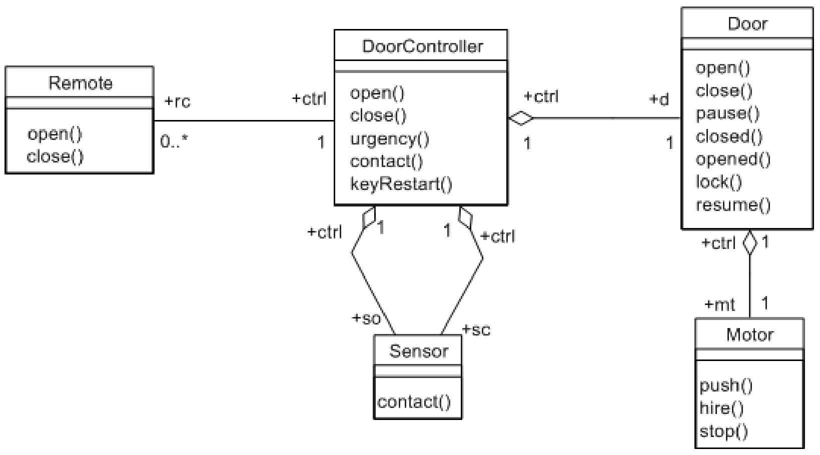


Fig. 2 Analysis Class diagram - garage door

The system operates as follows. 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,

[2] 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 be later be integrated as subsystems in larger applications.

[3] `https://tinyurl.com/wkja25u`

the door opens completely and triggers the open sensor `so`, 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` 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 (Fig. 3).

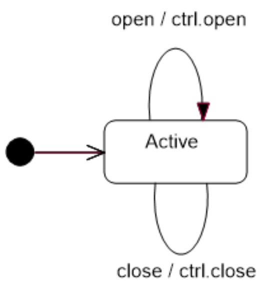


Fig. 3 Remote control State diagram - garage door

The state diagram of Fig. 4 describes the behaviour of the door controller. The actions on the doors are transferred to the engines by the door itself. User stories can be defined in requirement analysis

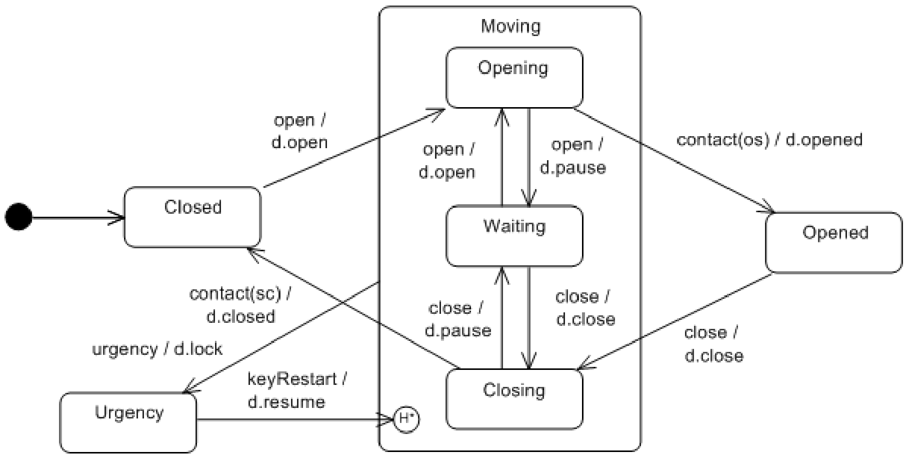


Fig. 4 Door controller State diagram - garage door

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 of Fig. 5 describes the collaboration of the door components when opening the door. Door actions are transferred to the motors by the door itself.

The verification of logic models includes at least static analysis and type checking. These can be designed as a transformation process [3] where advanced verification of properties require *model checking* for communications, *theorem proving* for functional contract assertions, and testing for behavioural conformance [1]. Most of them requires the translation to formal methods. In the following, before refining models to code, we assume model properties to be verified some way.

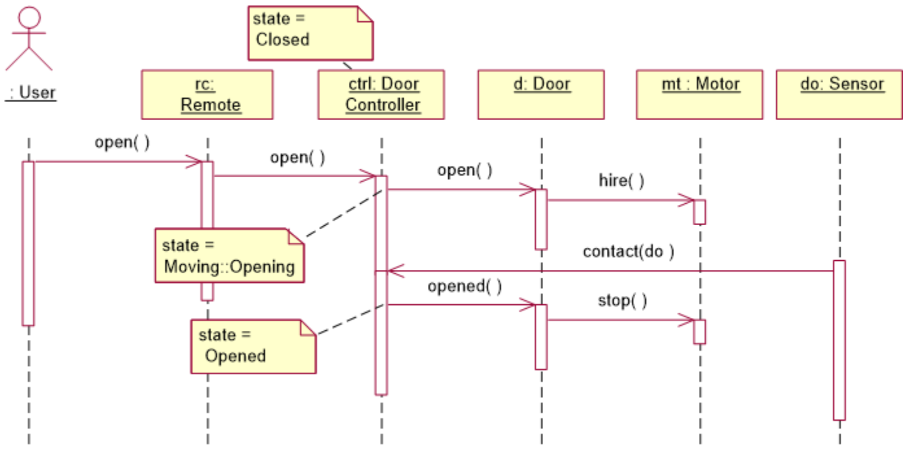


Fig. 5 Opening Sequence diagram - garage door

2.3 From Models to Implementation

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 Fig. 6. Available wireless protocols between EV3 and the remote are WiFi and bluetooth. Next step would be to select a technology in a library and to map model elements.

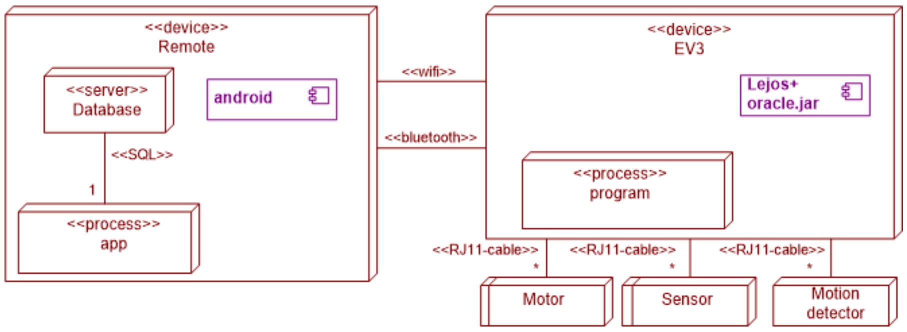


Fig. 6 Technical Architecture - EV3 and app

Software design is the activity that implements requirements inside a technical platform *cf.* Fig. 1. It is an engineering activity where decisions have to be taken that affect the quality of the result. Key design concepts are abstraction, architecture, patterns, separation of concerns, modularity. The result is a design model that cover the complementary aspects such as persistence, concurrency, human interfaces, deployment in an architectural vision that gradually reveals the details. This model should evolve under technical or functional changes. There are three main alternatives to develop an application from a logical model (design, coding, and testing). We classify them by degree of automation: (i) forward engineering, (ii) model transformation process, (iii) automatic code generation. Next we overview solution i) in Section 3, solution iii) which is at the opposite side of the automation spectrum in Section 3. The intermediate solution ii) will be detailed in Section 5.

3 Forward Engineering Experimentations

Developers implement a solution that should conforms to the given models and requirements specification. The case studies of Section 2.2 were given to different groups of students from 2018 to 2020. The starting point was a software requirement specification, a logical model, like the one figured in Section 2.2, on-line documentation on EV3 Lejos, and articles like [28,39,45,52]. All student's work have been prototyped using Lego Mindstorm sets. As an example one student project[4] led to the mock-up of Fig. 7.

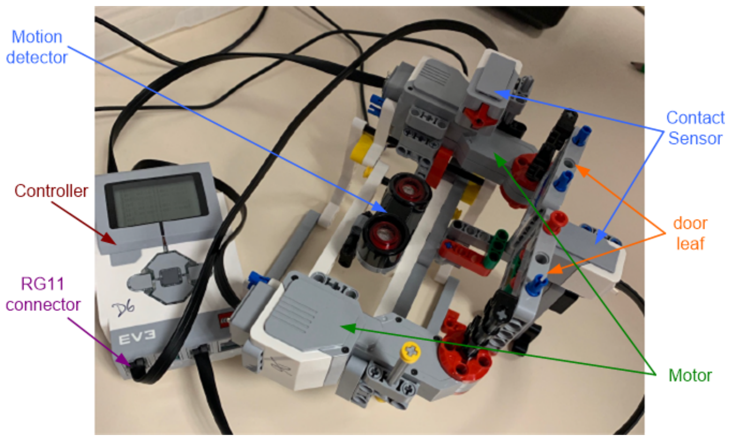


Fig. 7 Lego prototype of the door system

In the case of the garage door, a basic version called v1[5] was proposed in 2018, that has been extended later until having an Android App to play the remote device with Bluetooth connection and led to an implementation with enumeration types for STDs. Another implementation, called v2[6] led to the class diagram of Fig. 8.

The case was given to different groups of students who produced quite different implantations.

[4] `https://www.youtube.com/watch?v=7WBKTgRv7co`
[5] `https://github.com/demeph/TER-2017-2018`
[6] `https://github.com/FrapperColin/2017-2018/tree/master/IngenierieLogicielleDomoDoor`

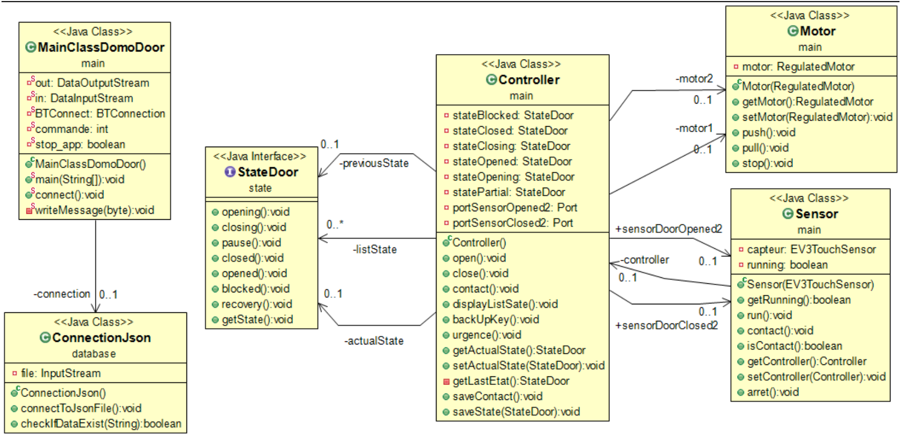


Fig. 8 Class Diagram of the door application (v2)

- 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 2.2) that were not mentioned in the given (simplified) logical model.
- The prototype of Fig. 7 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 example, despite implementation v1 and v2 use 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 pattern can be introduced here to improve the quality of the design [23]. In version `v1` the students used enum types to implement state machines while a *state pattern* has been chosen in version `v2` of Fig. 8.
- The technical background varies between the groups, some use the Java Lejos framework[7], others used ev3dev[8] which enables several programming languages. It depends on the development support (Integrated Development Environment -IDE) and operating system.

Isolating the various design choices is a first step to rationalize development in a refinement process (see Section 5).

[7] `http://www.lejos.org/`
[8] `https://www.ev3dev.org/`

The experimentation continued in 2019[9] by 7 groups of students who started from scratch. The development workflow delivered different products during the project according to the "Y" cycle of Fig. 1: technical analysis (by exploring EV3 frameworks), preliminary design, detailed design, 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 on the project with new student groups[10]. New objectives were fixed by the new groups based on their understanding of the freely chosen project (one of the seven's): 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 repeat similar mistakes due to 'not enough' time reasons, junior experience in projects. Quality goes far beyond functional suitability, but this criterion establishes the least sufficient level to reach at the end of their project.

4 Code generation, animation

Forward engineering makes space for software developers skill and experience leading to unpredictable process and results. Conversely, automatic code generation or animation define the way, even with parameter, to source code. We report in this section investigations on that automated code generation.

4.1 UML Case Tools

Our panorama is definitely not an exhaustive study but tries to find prototype tools for representative categories: free (F), community edition (CE), open-source (O), commercial (C) We compared the code generation facilities of some UML related tools according to the selected features of code generation given in Fig. 9.

- **Model Interoperability (UML/XMI)** In addition to interoperability UML and MOF based XMI are an important input formal for model transformation. The XMI header provide the UML version and may influence on the input models compatibility for the transformation.
- **UML Diagrams** We mainly focus on class diagrams (CD) and state transition diagrams (STD) by the means of code generation. UML editors should cover the main UML notation but differ one another on spe-

[9] Note that the other case studies (footnote 2 of page 5) have also be handled by other groups and lead to the same observations.

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

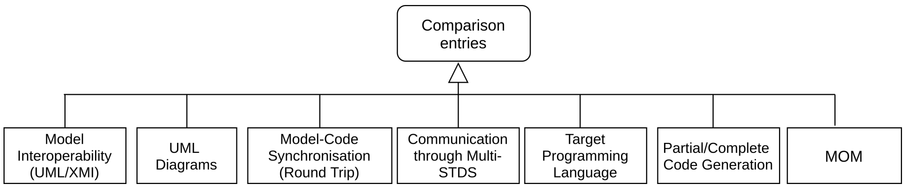

Comparison entries
Model Interoperability (UML/XMI)
UML Diagrams
Model-Code Synchronisation (Round Trip)
Communication through Multi-STDS
Target Programming Language
Partial/Complete Code Generation
MOM

Fig. 9 Comparison entries

cific notations, notation semantics and consistency links between diagrams. (i) Operation body is rarely define by activity diagrams (AD), OCL assertions or actions. However this would enable accurate transformation rules. (ii) Some tools consider the semantic link between classes from CD and STD in their code generation while others consider them separately.
- **Model Synchronization (Round-trip (RT))** keeps a strong link between a model element and a source code. This enable to replay code generation after a model evolution without loosing 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 to 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 communicating state machine (CSM) is often postponed to programming but it is a main feature for distributed applications.
- **Target programming language (TPL)** This entry indicates the target languages for the *behavioural part*, since we assume the static part to be provided by default.
- **STD** We consider here the None/Partial/Complete code generation for state-transitions diagrams. We will mentioned the STD elements supported by partial generator engine.
- **MOM (Message Oriented Middleware)** UML assumes an implicit middleware for message sent (reliable order preserving medium) but deploying a distributed application requires to implement distant communications. We look forward this aspect in code generators.
- **API Mapping** At low level, model elements are connected to predefined elements in libraries. In the case of a model element that exist, with a different shape, in a framework library, we call that API mapping. This point will be developed in Section 6.4.1. 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 strength and the weakness of some representative tools we have tested. The study is summarized in Table 1

Table 1 Comparison of some tools on their code generation facilities

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

– **Papyrus**[11] Code generation in Papyrus includes behaviours to operations with an incremental that overrides the code generation. Starting from the 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 [59]. It brings also some improvements in terms of the generated code *portability*, event processing speed, and optimization. Papyrus generates only *C++* code from STD, which make some difficulties to conduct our experiment when the generated programs will be deployed on Android. The context of state machine elements is defined by the associated class diagram, which provides a complete XML models that can be used to define a specific transformation rules, with the different transformation tools (ex: ATL, Java...).

– **Modelio** In addition to the fact that Modelio supports code generation for class and STD, it provides round trip functionalities which ensures synchronisation between code and model. Unlike 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

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

(not managed) method is under the responsibility of the developer. Fork, Join elements are not among the state diagram features in Modelio[12].

- **Umple** supports code generation from state machines to java, php, and c++ codes [41]. In Umple, CD operations are not associated to STDs. Umple has a solution to the problems of round-trip by allowing embedding of arbitrary code directly in the model [21]. Due to the fact that code generation does not support pseudo states (Fork, History, ...), Umple cannot design the behaviour of embedded industrial systems.
- **StarUML** is much more oriented to support modelling at educational and professional institutes. it provides rich modelling features but few 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 CD and STD 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[13]. 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 in synchronised. It provides a full code generation from combined Class and STD diagrams. However, no information confirming that the tool expresses communication between multi STDs. Again, we did not have access to the generated code to estimate the manual programming part.
- **Yakindu**[14] is a statecharts-based modeling and simulation tool proposed by Itemis. It generates a detailed implementation for one STD only which is independent from class diagrams. Despite the fact that Yakindo does not ensure a hight synchronisation level between code and models, it can be considered as a powerful tool to design complex behaviours. In version 4.0, the CSM code generation make it usable for control systems.
- **Framework for eXecutable UML (FXU)**[15] supports execution of concurrent state machines which can specify behaviour of many different objects. Regions of orthogonal states are executed concurrently as well. Transitions across vertices are triggered by events [45]. In FXU, CD and STD are related to each other. Class elements create context for STD diagrams. FXU, is based on IF-THEN-ELSE approach to generate Java code from

[12] `https://www.sinelabore.de/doku.php?id=wiki:landing_pages:modelio`
[13] `https://www.altova.com/umodel`
[14] `https://www.itemis.com/en/yakindu/`
[15] `http://galera.ii.pw.edu.pl/~adr/FXU/`

state machines. We found no way to export the source models with FXU, so we cannot define specific transformation rules using this tools

In the last years, we notice that there is a prominent progress in modelling tools to support model driven engineering process: (i) code generation, (ii) reverse engineering, (iii) modelling of real time and event-driven complex systems, (iv) dealing with the significant features of embedded systems such objects distribution, synchronous and asynchronous communication, become a key tool advantage for the industry.

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 modelling of structural and dynamic behaviour views of such complex system. Support for model verification exist 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 modelling task, 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.

Tool limits are related to the target programming languages *e.g.* the generated code from STD in Papyrus cannot be deployed or integrated to other frameworks or protocols *e.g.* 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.

To conduct our experiment, we used Papyrus models due to their completeness, interoperability and simplicity. Starting from these models (CD + STDs), we defined partial rules to generate step by step Java code.

4.2 Executable UML

Generating code from UML, by targeting a given technical architecture or a given *framework*, is still restricted to simple cases, like the CRUD (Create, Read, Update, Delete) application generation on relational databases. Some modelling environments propose to animate or to execute specifications, which are then qualified as *operational*. In both code generation and animation, the prerequisite is to start with complete logical models of the system structure (class or component diagrams), of its dynamic behaviour (state-transitions diagrams) and its functional behaviour (activity diagrams).

UML diagrams are usually not enough to specify a complete semantics. Constraints, associated to model elements, can help in providing precision. They are written in OCL, a declarative language for invariant and pre/post-condition assertions [13]. But again this is not sufficient. In particular, there usually miss information on *actions*. To be short, an action is a statement describing a computation or an object interaction. This is the basic behavioural structure in activity and state transitions diagrams. It also describes the glue between diagrams which make a foundation concept of UML.

The *Action Semantics* is defined by a meta-language since UML 1.4. No standard concrete syntax was proposed but early concrete syntaxes were associated to XUML tools, especially for real-time systems:

– *Action Specification Language (ASL)* was defined for iUMLLite of Kennedy-Carter (Abstract Solutions) supporting xUML [48].
– *BridgePoint Action Language (AL)* (and the derived SMALL, OAL, TALL) proposed by Balcer & Mellor was implemented in xtUML of Mentor Graphics [37].
– *Kabira Action Semantics (Kabira AS)* proposed by Kabira Technologies (and later TIBCO Business Studio).
– The normative telecom SDL [9] has also be used to provide a semantics as a UML profile.
– Other proposals are *Platform Independent Action Language (PAL)* of Pathfinder Solutions, or SCRALL [15] which had a visual representation, +CAL [44].

All these efforts led to semantics for a subset of executable UMLs, called `fUML` (*Semantics of a Foundational Subset for Executable UML*) [40], with now a normalized concrete syntax `Alf`. A reference implementation exists[16].

The executable UML tools are not directly convenient to design heterogeneous distributed applications since they have fixed targets however the action support is necessary for detailed design and we plan to integrate it in the project once the tool support will be available.

5 Toward a design transformation process

In MDE [10] forward engineering is seen as a *transformation process* from *Platform Independent Model (PIM)* to (more) concrete *Platform Specific Model (PSM)* injecting elements of the *Platform Description Model (PDM).*

5.1 Principles of design as a transformation

As illustrated in Fig. 1, the software design consists in "weaving" the logical model and the technical model (the *platform* in MDE) to obtain *in fine* an executable model. Each transformations comply an algorithmic style (*e.g.* Kermeta[17]) or a rule-based style (*e.g.* ATL[18]). The QVT standard accepts both styles [34]. We draw the reader's attention to the following observations:

[16] `http://modeldriven.github.io/fUML-Reference-Implementation/`
[17] `http://diverse-project.github.io/k3/`
[18] `https://www.eclipse.org/atl/`

– Only a complete (and consistent) logical model enables to reach an executable source code. Model transformation can infer but cannot create from scratch.
– The code generation cannot be viewed as a single transformation step, due to the semantic distance between the logical model and the technical model. Especially if the target is composed of orthogonal related aspects, called *domains* (*e.g.* persistence, HMI, control, communications, inputs/outputs), on which the logical model must be "woven".
– Design, as an engineering activity, is linked to the designers' experience (*cf.* Section 3). A process can be automated only if all the activities are precisely known.
– 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 suggest to work with small transformations. Small transformations leave the complexity to the process, not the atomic transformations. Small transformations are probably easy to verify and to compose (reusable).
– Modelling means finding abstractions. Refinement, the opposite process, introduces news details in models until reaching an executable model. For a long time we though process transformation as stepwise refinement [19, 34] but we showed in [5] that the gap to fill is too large and we had better to think in terms of *mappings*. This is detailed in Section 6.4.1 and Section 7.

5.2 Transformation process

On the above principle we propose a design process composed of four configurable composite transformations. A *composite transformation* is a process hierarchically composed of other (simpler) transformations, according to principle of *small step transformations*. It is depicted in Fig. 10 Each macro-

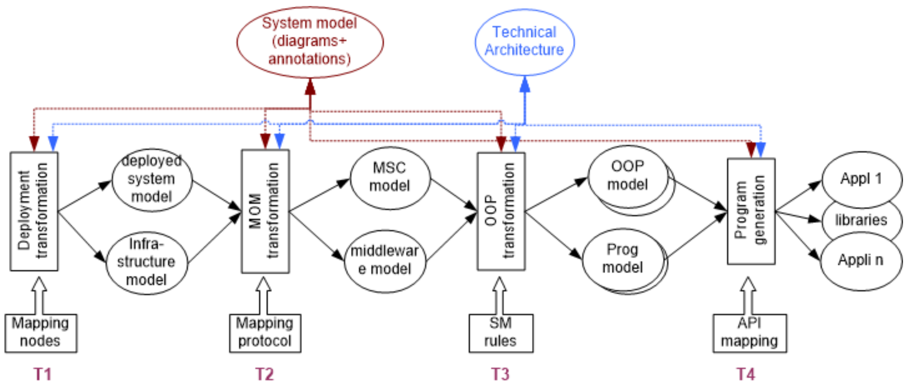


Fig. 10 General transformation process

transformation addresses either a 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 in Fig. 1, the deployment transformation will be simplified.
- The MOM transformation T2 focuses on object communication. For each kind of communication, the UML message sending are refined according to the protocol under consideration (called MOM in Table 1). In a single node deployment, message sending is simply method call in the target OOP language (Java, C ++ or C#).
- The OOP transformation T3 refines UML concepts into a OOP models which in general do not natively include these concepts. This thorny problem is discussed in Section 6.3. In particular T3 refines state-transition diagrams (resp. activity diagrams, multiple inheritance, associations...) in OOP concepts.
- The program transformation T4 pre-processes the code generation by matching model elements to predefined libraries of the technical *frameworks*. For example, the class Motor is implemented by the class lejos.robotics.RegulatedMotor. This *API mapping* requires adaptors for sending messages or calling methods. This point will be discussed in Section 6.4.

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

5.3 Implementation of the design transformation process

The process of Fig. 10 is abstract and generic. We now focus on implementation and customisation issues.

- *Input Quality* In addition to the static verification mentioned at the end of Section 2.2, type checking and assertions can be checked using OCL verification tools [14]. 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 profit to target these tools having a DSL entry [3].
- *M2M* Every step of the process is a Model-to-Model (M2M) transformation until the last level which is Model-to-Text (M2T) transformation to generate source code. At this stage, a rational implementation combines model transformations tools and code generation facilities of CASE tools (*cf.* Section 4).
- *Parallelism* For sake of simplicity, Fig. 10 hides the multiplicity of submodels. The more you progress in the process the more you have parallel transformations. At first T1 works one one global application. T2 is applies to each subsystem of each node. T3 is applies to each component. T4 applies to each class.

– *Iteration* By 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 (re) generation next.
– *Animation* When the technical environment is fully mastered, the transformation can "plunge" the model into the *framework* to make it executable. Refinement techniques to Java can be found in [35].
– *Tooling* Experimentations showed that no transformation tool was a panacea especially because various kind of transformation are in play including for example synthesis, extraction, mapping, refactoring [31]. But again, the overview of model transformation tools and the combination of tools including those of Section 4 is beyond the scope of this article. Transformation tool surveys can be found in [29,30].

In the following we provide details on the macro-transformations.

6 Macro Transformations Experimentations

In this section, we report implementation tracks and the experimentations we led in the context of the case study.

6.1 Deployment Transformation (T1)

The T1 composite transformation was designed manually by providing a deployment model of Fig. 11 from the analysis models of Section 2.2 and the technical architecture of Section 2.3. The bluetooth protocol has been selected to connect the EV3 and the remote computer.

In terms of transformation, the above activity is naively to group analysis classes into component clusters and to deploy components on deployment nodes (pick and pack). The designer must provides 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. Next step would be to select a technology in a library and to map model elements.

6.2 MOM Transformation (T2)

The problem is to refine UML communications according to the basic causality principle of UML[19]. *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*

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

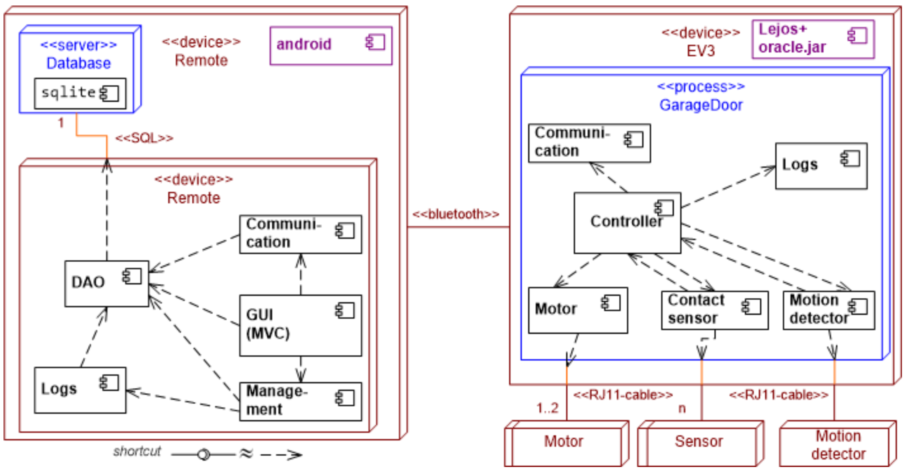


Fig. 11 Deployment diagram - garage door

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, signals). A message receive event is captured by the receiver protocol (state machine) leading to actions (including those of do−activities inside states). An action, described as an operation (for sake of uniformity) described in the class diagram by OCL assertions and *Action Semantics* statements, especially those actions related to message sent to join back the sequence and state-transition diagrams[20].

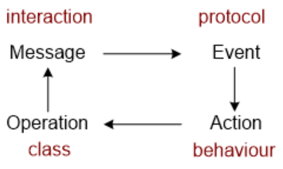


Fig. 12 Basic causality principle

In plain OOP, the problem yields in transforming individual message 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...),
- the underlying protocols (TCP-IP layers of services),
- the connecting mode (stateless, session).

For example, 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 sensors and adapters. A Wifi or Bluetooth con-

[20] Note that this principle binds sequence, state-transition and class diagrams providing a way to check some inter-diagram consistency rules [3] but also a way to organise models.

nection is required to be done manually and interactions happen during a session (open session - exchanges - close session).

A project led by a group of master students[21] explains the main issues and illustrates them on the conducted case study. Beyond the problem of defining the underlying communication support (service and protocol implementations, configuration, initialisation), the main point, considering UML models, is to isolate the message sending from the models before processing the communication instantiation transformation. For a sake of simplicity, the students chose to extract messages from sequence diagrams since the message sent are explicit[22] 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 learnt from that experimentation are:

- Messages are low level concepts in terms 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 is the part related to message and events (*e.g.* as a DSL).
- Some messages are simple procedure call in the target program. For example, the communications between EV3 and the sensors/actuators are Java method calls. We call them *primitive messages* in opposition to *protocol message* 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 message are used for objects deployed on different nodes. Recall that the deployment diagram provides the protocol stereotype on communication path between nodes. Otherwise, a 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 in a proxy call that will be in charge of transferring the message to the receiver according to the middleware configuration.

When there are variable communication media, an alternative is to consider communications as an orthogonal interoperable concern. We proposed a solution to that alternative called Multi-protocol communication tool (MPCT) in [4].

[21] `https://ev3.univ-nantes.fr/rapport_ter_22-05-2020/`

[22] Abstract to raising signals or time events.

6.3 OOP Transformation (T3)

Suppose UML−java, a UML profile that accepts only UML concepts which are meaningful in Java. The macro-transformation T3 transforms UML models to UML−java models. For a sake of conciseness, we sketch the following simplified *sequence* of transformations:

1. Transform STD `[T3.1]` associated to classes into OOP structures (a missing criterion in Table 1). Various strategies (enumerations, *State* pattern, execution engine) are possible for the same case study according to the nature of the automata. For example, binary states (light is on or off) or enumerations are simple solutions for an automaton with few states, while the State pattern [23] is useful if the associated operations have different behaviour from one state to another and the number of states remains limited (see the illustrating example below). Beyond 10 states, an instrumentation machinery (a framework) is necessary, connecting to a *framework* API for instantiation, inheritance, call, mapping...
2. Transform Activity Diagrams (AD) associated to operations into OOP structures. This problem is a variant of the STD transformation.
3. Transform multiple inheritance to single inheritance[23] 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.
4. Class-associations are transformed into classes plus associations. The multiplicity is 1 in the new class role side.
5. Aggregations and compositions are transformed into simple associations.
6. Dependencies are transformed into <<import>> dependencies. Variants are possible according to given stereotypes.
7. 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 to the model.
8. Process the meta-features (attributes, operations) is not required in Smalltalk but it is for Java, C# or C++. 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 [23] would be of interest.
9. The derived features (attributes, associations) are transformed by operations. If an OCL constraint gives a computation, it can be an assertion of the method associated to this operation.

[23] 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 replace inheritance by 1-to-n associations.

10. 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 name. 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 be mention a union of types $B \vee Null$ since it is optional.
    - Otherwise it is a set, an ordered collection, a sorted collection, a map if the association was qualified).
11. Operations are transformed into methods. If an OCL assertion was associated to the operations, it can be an assertion of the method associated to this operation.
12. Stereotypes can be handle. As an example, a candidate identifier <<key>> (for persistent data) lead to uniqueness constraints in OCL invariants.
13. OCL invariants are implemented by test assertions (*e.g.* jUnit) or operations that are called every time an object is modified.

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 organise the macro-transformation T3.

Now we describe the experimentations on the STD sub-transformation `[T3.1]` in the above sequence.

Example: UML2Java, a STD Transformation with ATL

Due to its expressibility and abstraction, we chose ATLAS Transformation Language (ATL)[24] 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 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).

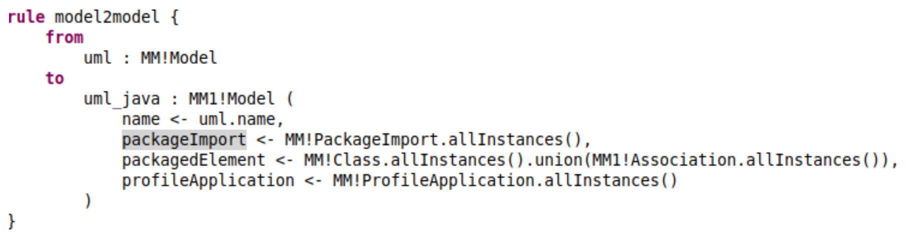
ATL proposes two modes for transformations from and refine. The from mode enables to create a model by writing all the parameters, all the attributes in the output model. The refine mode is used to copy anything that is not included in the rule into the output template and then apply the rule. A rule 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. Morever we want to avoid DSL explosion, we limit the number of metamodels or profiles by keeping UML as far as we can.

[24] `https://www.eclipse.org/atl/`

STDs are assumed to be simple automata: no composite state, no time, no history. Also a 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 (for example, the elements Region and StateMachine have the name of their class) that make it easier to write the transformation rules.

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

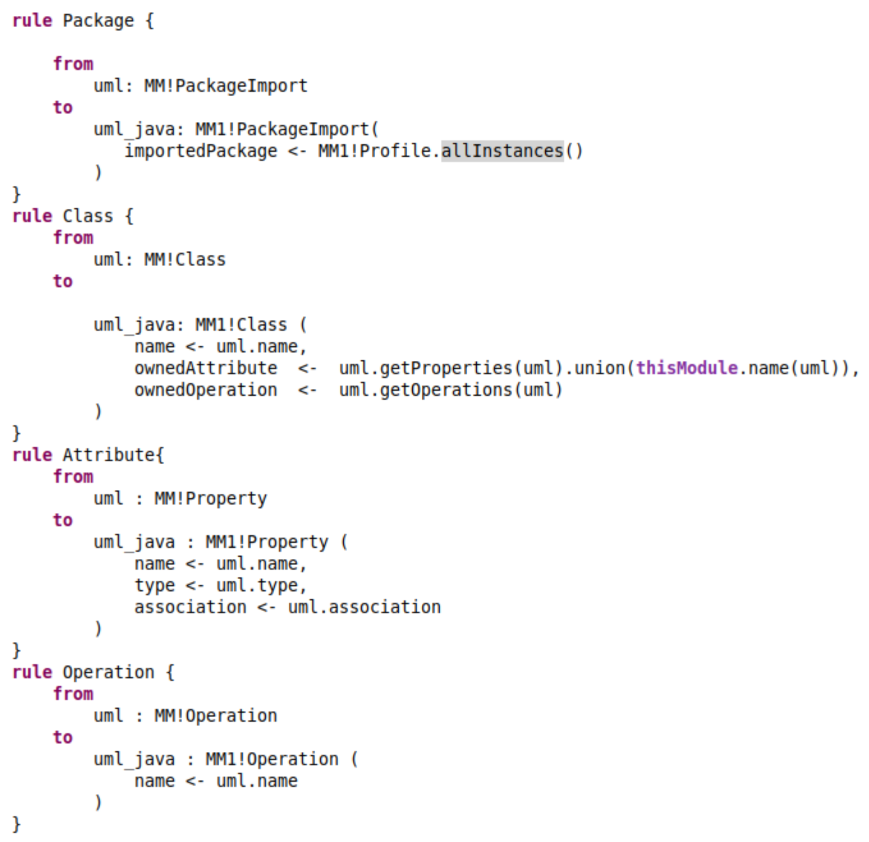
1. Generate a Java model that have exactly the same UML-Papyrus models structure. In this line, Fig. 13 describes the ATL rule building a target XMI model with respect to Papyrus specification. The model2model rule builds the main structure of the generated XMI model. This model, called uml_java (MM1!Model), 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()
        )
}
```

Fig. 13 Model to model transformation -basic rule

2. Once the main XMI structure of the Java target model is built. The second step copy 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), Methods(MM!Operation). As described in Fig. 14, 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 (Fig. 15) to transform this behaviour into UML-Java. Among the alternatives given in transformation `[T3.1]`, we chose the *State* pattern because it is straight forward. According to the pattern definition [23], the corresponding Java elements will be generated:
   (a) A Java *Interface* representing the STD of each object,
   (b) The Java class should *implements* the generated stateMachine *interface*,
   (c) For each context class (1) a private attribute references the STD and (2) a public method setState() defines of the current object state.
   (d) We generated 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



```
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
        )
}
```

Fig. 14 From UML elements to Java elements

two parameters: 'language' which takes the value 'JAVA' and 'body'. The body parameter is initialized with the concatenation of the enumeration name and the initial state. The initial state is found by retrieving the target state of the transition having the initial state as source state.

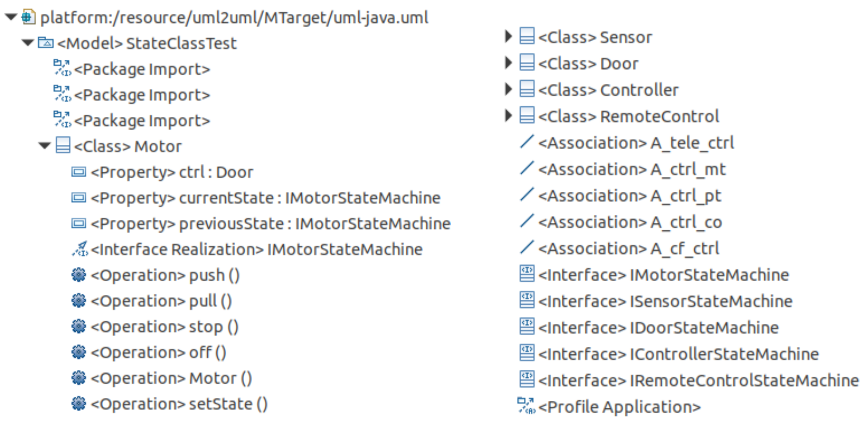
(e) To determine the behaviour of the operations. For each operation used as a trigger in a state machine, we will create a condition switch in the method implementing the operation. To fulfil the condition, we retrieve the source state and target state of all transitions that trigger the function. The source states correspond to the possible cases for the change of 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.

The experiments highlight the complexity of the problem and some basic aspects to deal with. The results are still far from the final objectives.

```
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'
     ),
}
```

Fig. 15 From STD to Java elements



```
platform:/resource/uml2uml/MTarget/uml-java.uml
  <Model> StateClassTest
    <Package Import>
    <Package Import>
    <Package Import>
    <Class> Motor
      <Property> ctrl : Door
      <Property> currentState : IMotorStateMachine
      <Property> previousState : IMotorStateMachine
      <Interface Realization> IMotorStateMachine
      <Operation> push ()
      <Operation> pull ()
      <Operation> stop ()
      <Operation> off ()
      <Operation> Motor ()
      <Operation> setState ()
    <Class> Sensor
    <Class> Door
    <Class> Controller
    <Class> RemoteControl
    <Association> A_tele_ctrl
    <Association> A_ctrl_mt
    <Association> A_ctrl_pt
    <Association> A_ctrl_co
    <Association> A_cf_ctrl
    <Interface> IMotorStateMachine
    <Interface> ISensorStateMachine
    <Interface> IDoorStateMachine
    <Interface> IControllerStateMachine
    <Interface> IRemoteControlStateMachine
    <Profile Application>
```

Fig. 16 The Java elements generated for the garage door

6.4 Source Code Transformation (T4)

Transformation T4 aims at unifying model elements and implementation (source code). All model elements are not generated from scratch, some already exist, maybe in a different nature, in the technical model (*cf.* Fig. 1). As mentioned in Section 4, 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 source code generation. To simplify the discourse will focus on classes as to be the model elements, but it should be extended to packages, data types, predefined types or operations and so on.

6.4.1 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 level in the Lejos library. For 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 if possible matching rates. Different model elements are taken into account such as class, attribute, operation... We face here 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 7.
   – 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 organised influence the matching process.
2. *Select* the adequate implementation of model element (class, attribute, operation) and bind the model elements. We proposed a non intrusive solution of this problem in [43].
3. *Adapt* to the situation. Once a mapping link is established, it usually implies to refactor the design. Adaptation is the core mechanism to bind the two branches of the "Y" process of Fig. 1. Several strategies can be chosen
   – *Encapsulate and delegate*. The model classes are preserved that encapsulate the implementation classes (Adapter pattern). The advantage are to keep traceability and API. The drawback is the multiplication of classes to maintain.
   – *Replace* the model classes by the implementation classes. The transformation must replace the type declarations but also all messages sent. The pro and cons are the inverse of encapsulation.

   During forward engineering, the students used both strategies, depending on their concerns with traceability, easy of implement, code metrics...
   Replacement is possible when classes have same structure and same behaviour but also for UML/OCL/AS primitive types. In any other cases the Adapter pattern captures multi-feature adaptations:

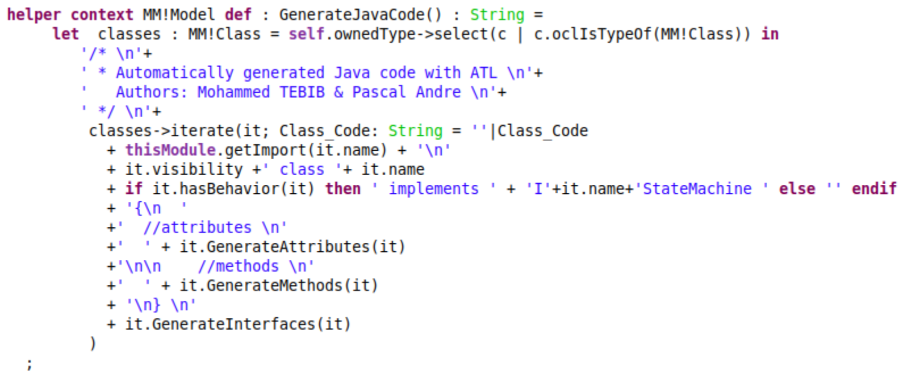
- Attribute: name, type adaptation, default value, visibility...
- References (role): name, type adaptation, default value, visibility...
- Operation: name, parameters (order, default), type adaptation...
- Protocol: STD for the model class but not the implementation class.
- Composition: a class is implemented by several implementation classes.
- Communication refinement: MOM communictaions are distributed.
- API layering: classify the methods to reduce the dependency.
- Design principles: improve the quality according to SOLID, IOC...

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

6.4.2 Source Code Transformation with ATL

This transformation is a Model-To-Text (M2T) transformation that generates source code from the UML models resulting from transformation T3. To parse the XMI model and generate the Java code, we defined an ATL transformation engine composed from a set of sub-transformation rules.

1. *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 of Fig. 17 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)
         )
  ;
```

Fig. 17 ATL transformation rule for classes

- The GenerateClasses() helper parses every class presents in XMI model (UML-Java) and generate 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 modelled interfaces if there exist.

– The GenerateAttributes() helper parses all classes and generates all information related to the attributes: visibility, name and type (see Fig. 18).

```
--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  '
);
```

Fig. 18 ATL transformation rules for attributes

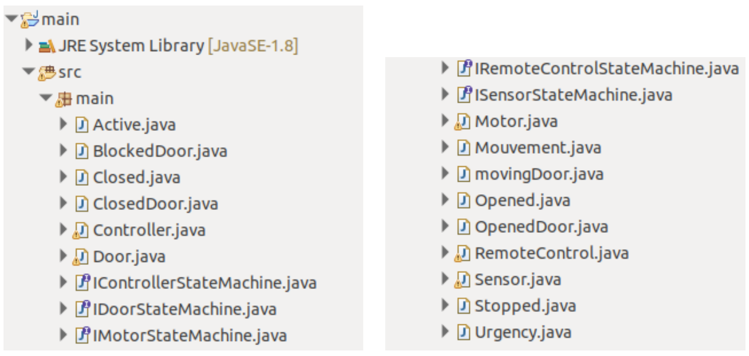
– The GenerateMethods() helper generates the method signature: visibility, returned type, name and parameters (see Fig. 19).

```
--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   '
    )
;
```

Fig. 19 ATL transformation rules for methods

Fig. 20 shows the list of Java files generated for the Motor model class.

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



```
main
  JRE System Library [JavaSE-1.8]
  src
    main
      Active.java
      BlockedDoor.java
      Closed.java
      ClosedDoor.java
      Controller.java
      Door.java
      IControllerStateMachine.java
      IDoorStateMachine.java
      IMotorStateMachine.java
      IRemoteControlStateMachine.java
      ISensorStateMachine.java
      Motor.java
      Mouvement.java
      movingDoor.java
      Opened.java
      OpenedDoor.java
      RemoteControl.java
      Sensor.java
      Stopped.java
      Urgency.java
```

Fig. 20 The list of the generated classes

6.4.3 Source Code Transformation with Papyrus

Since 2017, Papyrus provides 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 [59]. As explained in section, the code generator engine of papyrus extends IF-Else/Switch construction of programming languages that supports state machines hierarchy. It brings many features compared to the existing tools [59] such as:

- All statechart elements are taken into account during code generation,
- Consider sync/asynchronous behaviours through events support,
- The used UML is conformed to the OMG standard,
- Much more improvements in terms of efficiency: events processing is fast and the generated code size is small,
- Concurrency and hierarchy support.

The generated code could be only on 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 State Design Pattern which is an oriented object approach that could also support hierarchical state machines. These solution suffer from one limit that is related to the explosion of the number of classes that requires much memory allocation. Note that there is an ongoing work by Papyrus designers to add Java code generation from STDs.

6.4.4 Source Code Transformation using Mapping

This transformation find candidate mappings and establish 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 have at disposal a model of the framework or to establish one if none exist yet. This point will be discussed in Section 7.

In a previous work [6] we faced the problem of identifying components in a plain Java program and one of the issue 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 another work [43] 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 7).
- 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 here model classes and candidates. The implementation classes come from the Lejos library (see Section 7.2). Recall that the GUI part is considered to be developed separately. For the simple example of class Motor, Table 2 show it is not an easy task to detect which candidate class could be the good one. 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 Fig. 21.

In the example of Fig. 22, the (model) class Motor delegates its method calls to the EV3Large RegulatedMotor.

Based on the specification of a simplified *Adapter Pattern* presented in Fig. 21,

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

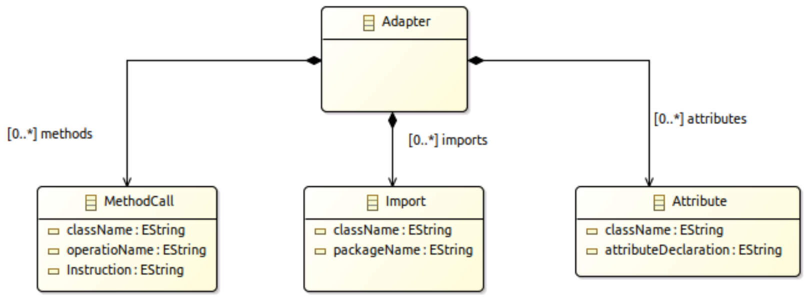


Fig. 21 Adapter Pattern Model

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 perform on the defined *operationName* existing in the class specified by the *className* attribute, (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 Fig. 23.

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 the one of the framework. For API calls, the helper mappingMethods takes as an input a couple of parameters representing the

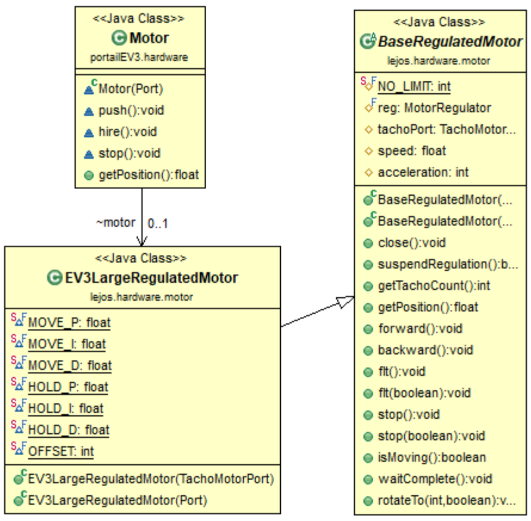


Fig. 22 Class Mapping by Adaptation of the Motor Class

```
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)
;
```

Fig. 23 ATL helper to generate adapted attributes

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

Listing 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 2. It implements direct mapping for class, imports and method call.

Listing 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;
     }
}
```

The above transformations work for direct name-based mappings. Additional work is necessary for more complex transformation, and currently developers have to code more complex adaptations.

7 Reverse engineering PDMs

As mentioned in Section 5.1, stepwise refinement is implemented by model mapping when too many details have be brought in the transformation; low-level model mapping has been illustrated in Section 6.4.1. Model mapping is made applicable only if a model of the target framework (PDM) exists. One way to get an XMI model 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 Model Driven Reverse Engineering (MDRE) as illustrated by Fig. 24. Note that the mapping persists at the PSM level *e.g.* adapters store the API mapping (*cf.* Section 6.4.4).

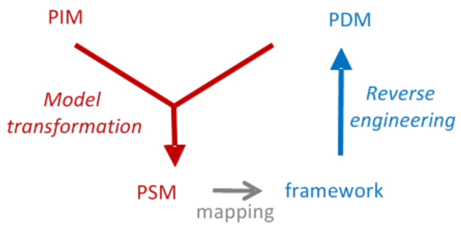


Fig. 24 View of a mapping transformation

7.1 Model Driven Reverse Engineering (MDRE)

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 [51]. 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 perceive as a transformation process, MDRE is itself a transformation process [2]. A MDRE step can be represented by a model transformation from a PSM up to a PIM as illustrated by Fig. 25. 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 very simple but the source and target models are usually known. Finding abstraction is an even more difficult problem in MDRE [47,51]. Abstraction hides implementation details. During the development of software systems, high level abstraction models refer to system analysis and design while low level ones refer to implementation and deployment of the solutions. *Abstraction layers* represent the organisation of complex architectures; typical examples are the ISO stack of protocols and services for telecommunications or the service architecture approach (SOA). 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.

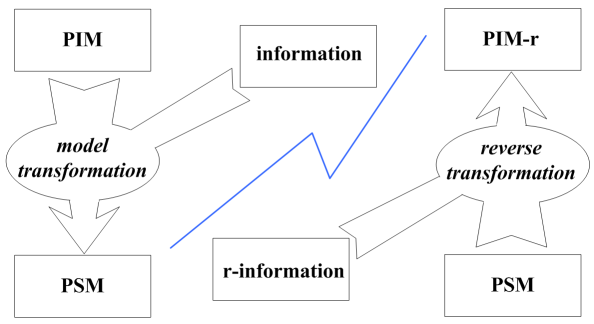


Fig. 25 Reverse Model Transformation

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 customised models defined with domain specific languages (DSL). The source information also differs and may include binary code, source code, configuration files, tests programs or scenarios models... Such a diversity make the RE activity difficult to solve.

The question is not to sort abstract and concrete elements from the source information but to built abstractions. The more you hide in the abstraction, the more it is difficult to find abstraction. For example, Knowledge Discovery Metamodel (KDM) is a standard for software system representation [11]. It is useful at low level because it is a model representation but it keeps a very detailed information which miss abstraction.

7.2 Example: Reverse engineering Lejos libraries

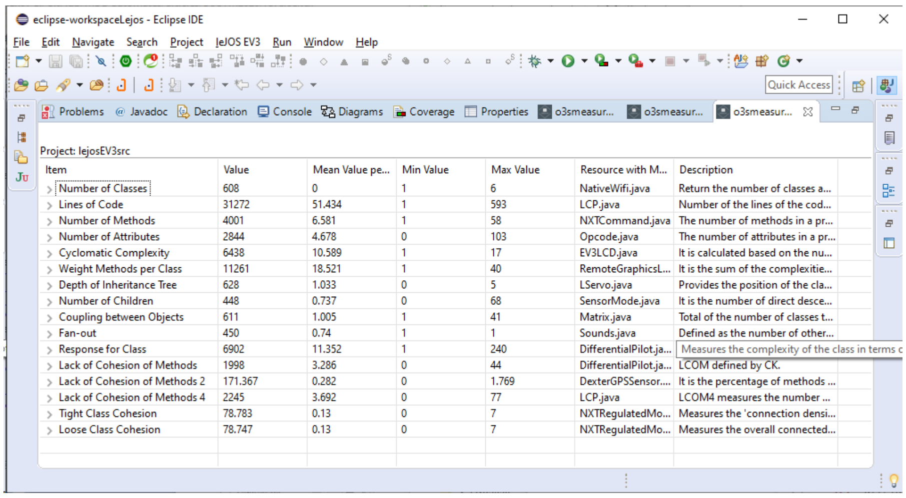
In our conducting case study, we expect a model from the specific Lejos framework[25]. More precisely we consider only the EV3 library[26] as the PDM. We extracted the main in its source form `ev3classes-src.zip`. The o3smeasure metrics are given in Fig. 26.

For this case study, we experimented with Papyrus, Modisco and AgileJ. Papyrus enabled to reverse engineer[27] individual classes but not packages. In the context of a papyrus project, applying the command `Java>Reverse` on

[25] Android/Java libraries are considered as standard for the experimentation purpose.

[26] Lejos is a complete Operating System based on an Oracle JVM.

[27] `https://wiki.eclipse.org/Java_reverse_engineering`


eclipse-workspaceLejos - Eclipse IDE
File Edit Navigate Search Project leJOS EV3 Run Window Help
Quick Access
Problems @ Javadoc Declaration Console Diagrams Coverage Properties o3smeasur... o3smeasur... o3smeasur...

Project: lejosEV3src

Item	Value	Mean Value pe...	Min Value	Max Value	Resource with M...	Description
Number of Classes	608	0	1	6	NativeWifi.java	Return the number of classes a...
Lines of Code	31272	51.434	1	593	LCP.java	Number of the lines of the cod...
Number of Methods	4001	6.581	1	58	NXTCommand.java	The number of methods in a pr...
Number of Attributes	2844	4.678	0	103	Opcode.java	The number of attributes in a pr...
Cyclomatic Complexity	6438	10.589	1	17	EV3LCD.java	It is calculated based on the nu...
Weight Methods per Class	11261	18.521	1	40	RemoteGraphicsL...	It is the sum of the complexitie...
Depth of Inheritance Tree	628	1.033	0	5	LServo.java	Provides the position of the cla...
Number of Children	448	0.737	0	68	SensorMode.java	It is the number of direct desce...
Coupling between Objects	611	1.005	1	41	Matrix.java	Total of the number of classes t...
Fan-out	450	0.74	1	1	Sounds.java	Defined as the number of other...
Response for Class	6902	11.352	1	240	DifferentialPilot.ja...	Measures the complexity of the class in terms c
Lack of Cohesion of Methods	1998	3.286	0	44	DifferentialPilot.ja...	LCOM defined by CK.
Lack of Cohesion of Methods 2	171.367	0.282	0	1.769	DexterGPSSensor....	It is the percentage of methods ...
Lack of Cohesion of Methods 4	2245	3.692	0	77	LCP.java	LCOM4 measures the number ...
Tight Class Cohesion	78.783	0.13	0	7	NXTRegulatedMo...	Measures the 'connection densi...
Loose Class Cohesion	78.747	0.13	0	7	NXTRegulatedMo...	Measures the overall connected...

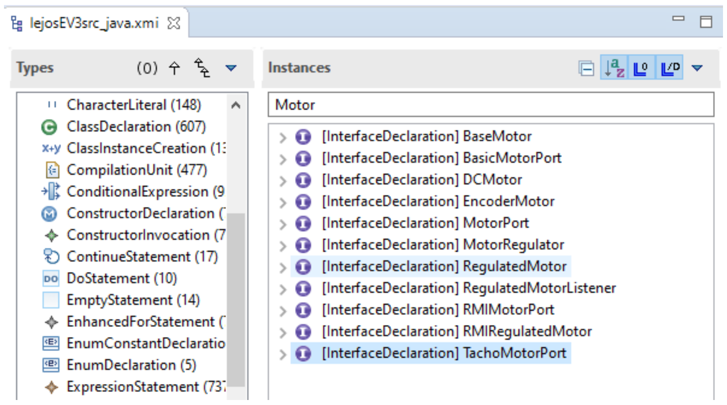
Fig. 26 Metrics of the Lejos EV3 classes library

lejosEV3src model elements fails except for classes. Even for a class, the methods were not included. In Modisco [11], UML discovery from Java code is composed of two transformations (Java to KDM / KDM to UML). Unfortunately, the second one 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 [28] reverse the java code to UML class diagrams is simple. Fig. 30 shows the result of applying reverse engineering on lejos library using AgileJ/structureviews. From a visual point of view, we note that it provides many relationships between classes (see Fig. 30), compared to other tools like ObjectAid (see Fig. 8). Especially in the case when the number of classes is too big, and that by (1) building and maintaining a better overview of the architecture and (2) highlighting where the design can be improved and refactored.

Recall that the initial problem to solve is match model elements to PDM abstractions. In this experimentation, the working unit is the class element. For each model class, *e.g.* Motor to goal is to find candidate implementation classes in the framework model. The MDRE process aims at providing foundations 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 framework are structured to evolve. (iii) search according to string matching or (iv) or better on pattern matching (including references, attributes and operations). These can

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

be implemented by Modisco queries. Specific stereotypes or annotations to separate model classes are helpful in the case of iterative processing.


lejosEV3src_java.xmi

Types (0)

CharacterLiteral (148)
ClassDeclaration (607)
ClassInstanceCreation (1
CompilationUnit (477)
ConditionalExpression (9
ConstructorDeclaration (
ConstructorInvocation (7
ContinueStatement (17)
DoStatement (10)
EmptyStatement (14)
EnhancedForStatement (
EnumConstantDeclaratio
EnumDeclaration (5)
ExpressionStatement (73

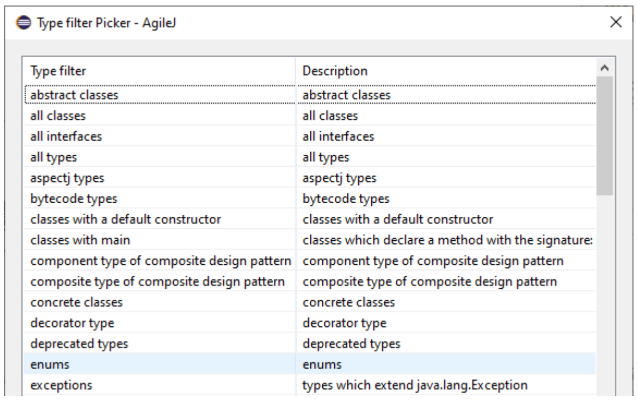
Instances

Motor

[InterfaceDeclaration] BaseMotor
[InterfaceDeclaration] BasicMotorPort
[InterfaceDeclaration] DCMotor
[InterfaceDeclaration] EncoderMotor
[InterfaceDeclaration] MotorPort
[InterfaceDeclaration] MotorRegulator
[InterfaceDeclaration] RegulatedMotor
[InterfaceDeclaration] RegulatedMotorListener
[InterfaceDeclaration] RMIMotorPort
[InterfaceDeclaration] RMIRegulatedMotor
[InterfaceDeclaration] TachoMotorPort

Fig. 27 Modisco discovery for interfaces

AgileJ provides a filter tool (*cf.* Fig. 28) which 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.* show all interfaces or show abstract classes.


Type filter Picker - AgileJ

Type filter	Description
abstract classes	abstract classes
all classes	all classes
all interfaces	all interfaces
all types	all types
aspectj types	aspectj types
bytecode types	bytecode types
classes with a default constructor	classes with a default constructor
classes with main	classes which declare a method with the signature:
component type of composite design pattern	component type of composite design pattern
composite type of composite design pattern	composite type of composite design pattern
concrete classes	concrete classes
decorator type	decorator type
deprecated types	deprecated types
enums	enums
exceptions	types which extend java.lang.Exception

Fig. 28 AgileJ filter process

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 customize query and transformation. Papyrus is still on contribution. Further reading can be found in [2].

8 Discussion

We discuss here some lessons learnt from the above studies and related works. The manual design of the application from a logical model was not greatly difficult to the students, except 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 are not a priori 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 focus on methodology and guideline 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 example, in [25], the authors use MDE for process compliance, however the refer to standard or de facto development process which are far from our practice orientation. They are not concerned with the produced models contents but rather with their meta-information. As mentioned by Aranda et al. in 2012, the step from traditional development, even with model, to MDE is large and disruptive and *"practitioners and researchers have little information to help guide them on this process."* [7]. It seems to be still the case in 2020, especially for code generation as mentioned by Sebastián et al. *"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.* [53]. However ad-hoc solutions exist. For example, Sindico et al. [56] present a MDE process based on the INCOSE framework that conforms to the MIL-STD-498 standard for military real-time embedded systems with SysML Marte, Simulink... The platforms play the role of integration system for engineers. SysML models are given for the requirements but also the target platform. The solution works at low level of electronic devices but is interesting to compare with.

Code generators provide incomplete models, which often does not even exploit the information of the model (OCL constraints, operation details).

Lesson 2: Even for very detailed models, automatic code generation can't apply in the large. Although many studies have been conducted, the systematic study of Ciccozzi et al. [17] shows that the execution of UML models remains a difficult problem and answers to animation needs not to software development. However, the new standards `fUML` and `Alf` contribute to palliate a lack of action semantics. They have been implemented, for example, in the verification of models [46,58], execution via C++ [16] or MoKa/Papyrus [27,46]. We are convinced that MDE can cross the threshold since tool support handle these standards.

In MDD, the abstraction gap between analysis models and the detailed design models is huge. In the development *workflow* of Fig. 10, the macro-transformations are ordered according to the impact: architectural choices (deployment, communications), general design choices (programming language), detailed design choices (*patterns*, *library mapping*). The experimentations showed that transformations are already numerous and difficult to design even if we did not take into account the bridges between domains and the orthogonal aspects of PSMs [32].

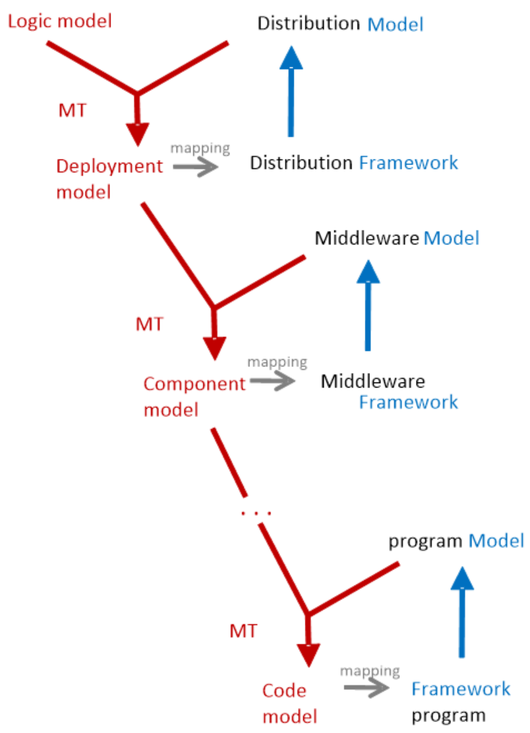


Fig. 29 Mapping transformation process

Lesson 3: providing information and decision as parameters to design transformation is really a problem. Among the techniques to define transformations [32,31] we suggest to use *mappings* to fill the abstraction gap. This refinement process has been transformed into a mapping process in Fig. 29, according to the principles of Section 5.1. For sake of simplicity, Fig. 29 shows one couple of models per level but the more you progess to code the more you have domains in parallel. At each level, a model of the technical frameworks is requested. Reverse engineering tool support exist at low level but raising in abstraction remains an engineering task [2]. The mapping process, requires a matching process. Many techniques exist as pointed by Somogyi and Asztalos [57], based on graphs or text algorithms. However they are more useful to compare models than to find implementation candidate, as we showed in Section 6.4.4. A *non-intrusive mapping* technique and a tool are presented in [43].

We made use of design patterns for detailed design. We observed pattern implementation from books or student work and observed that implementations varies making it difficult to process systematically in pattern transformations.

Lesson 4: Standard design best practices may lead to pattern transformation. Having a pattern description of the PDM models would help to define the mapping transformations. We did not found pattern (based) transformations but there exist works on transformation patterns *e.g.* [36], that improve some quality characteristics such as modularity or efficiency of the transformations.

Similarly to the Components On The Shelf (COTS), our MDE vision requires that technical framework providers delivers not only libraries but also models. First models play the rôle of documentation because current API are difficult to overcome and abstraction is really missing. Components, class and state machine protocols are convenient means to explain and to understand the framework design. Second, models are a mandatory input for our transformation mappings.

Lesson 5: MDE works with MDRE. Assuming a framework model, the question is how to make it consistent with the code libraries, especially after evolution and releases. This is the point of round trip and MDRE facilities. The first implies a MDE development of frameworks, according to the principles introduced in this article. The second requires new abstraction heuristics to get high-level abstractions from low level abstractions [6] as illustrated in Section 7.2. We feel that many of the challenges on model evolution given in [60] remain on the way.

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

Lesson 6: Collaboration is required 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 but also development components to integrate them. The development strategies should be parametrisable in the transformation process. For example, coding state machines is subject to interpretation and strongly related to the execution model [45]. We also believe that several transformation tools should be combined because the rule-based approach is unsuitable in some operational transformations like the one mentioned in [45]. In particular, we tried to combine model transformation with code generators.

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

Lesson 7: MDA is an evolving entity that generates growing technical debt. We observed improvements on 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 [30].

Despite our case study belongs to cyber-physical systems, we did not consider the low-level hardware connection as in [61]. The lejos java library is already the hardware platform abstraction. In [61], (low-level) SysML models are mapped to realise the implementation and model transformation plays a secondary role.

Finally, we discuss some threats to validity.

– *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 contexts enables to cover the main software design concerns (distribution, communication, persistency, concurrency, deployment...) in a reduced complexity set and we think 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 make it more applicable than dedicated approaches such as the one of Section 4.2. But other experimentations of different types of applications and different PDM 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 modelling 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 trans-

formed, *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), their integration is much more complex because they have organisational consequences on those late transformation. There is a composition issue we have not tackle yet in detail.
– *Can this method be applied in real software development process ?* Theoretically the answer is yes but there is a long load to this goal due to the numerous prerequisite for an assisted process: PDM models, transformation engineering (transformation implementation, mapping transformations, operators to combine transformations, verification...). The next step should be a high level transformation case tool with libraries of (compatible) transformations.

9 Conclusion

Model-Driven Engineering did not reach yet the promises given in [32] for the software development practice. Integration modelling process still require methodological standards and tool support despite progress is not mature enough. Domain specific applications coupled with specific platforms can lead to good automation rate but add them up do not provide general-purpose equipment. The maintenance of software systems implies a high reactivity of the development teams and highlights the need for industrialisation tools that go beyond integrated development and deployment platforms.

This article takes the development problem from a practitioner's point of view, we share experiments and vision, and we exhibit solutions where assistance and automation can take place. The gap between logic models and implementation stay large, the "no code" vision is utopian for general purpose modelling languages such as UML or SysML. Starting from low level design model stay difficult if the frameworks have not been integrated in the models.

The problem has been studied by reporting forward engineering experience and comparing modelling code generation tool support to elaborate a stepwise development workflow based on both engineering and reverse-engineering activities. To reduce a technical debt, we propose to abstract the infrastructure and reason at the model level while facilitating the refinement of these models in executable versions. The experiments carried out here stand to that direction and induce the feasibility of the different model transformations which (partially) automate the process.

Our contribution does not pretend to replace developers but to assist them to build integration chains that make maintenance and evolution less expensive. The human intervention in transformations remains predominant when there are alternative choices, such as state machines or message send detailed design. The process has to be more rationalised to be automated or even assisted by the means of interactive design decisions. This point remains premature in the state of our experiments.

This works establishes a starting point from which further works are requested through collaborative community contribution. Enriching models, formalizing processes of refinement, making modular and personalizing development to rely on transformation tools are the tracks we follow. Many tracks remains to be explored, that are challenging. From a theoretical point of view, the transformation processes remain 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 rationalise the software engineering process as a combination of decisions and experiment 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 [36]. Also it is necessary to be able to reverse engineering the design frameworks as platforms models and to combine transformations written in different languages and that are interactive so that the designer influences the design choices.

References

1. André, P., Attiogbé, C., Mottu, J.M.: Combining techniques to verify service-based components. In: Proceedings of the International Workshop on domAin specific Model-based AppRoaches to vErificaTion and validaTiOn, AMARETTO@MODELSWARD 2017. Porto, Portugal (2017)
2. André, P.: Case studies in model-driven reverse engineering. In: Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2019, Prague, Czech Republic, February 20-22, 2019, pp. 256–263 (2019). DOI 10.5220/0007312502560263
3. André, P., Ardourel, G.: Domain Based Verification for UML Models. In: L. Kuzniarz, G. Reggio, J.L. Sourrouille, M. Staron (eds.) Workshop on Consistency in Model Driven Engineering C@Mode'05, pp. 47–62 (2005)
4. André, P., Azzi, F., Cardin, O.: Heterogeneous communication middleware for digital twin based cyber manufacturing systems. In: T. Borangiu, D. Trentesaux, P. Leitão, A.G. Boggino, V.J. Botti (eds.) Proceedings of SOHOMA, Studies in Computational Intelligence, vol. 853, pp. 146–157. Springer (2019)
5. André, P., Tebib, M.E.A.: Refining automation system control with MDE. In: S. Hammoudi, L.F. Pires, B. Selic (eds.) Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2020, Valletta, Malta, February 25-27, 2020, pp. 425–432. SCITEPRESS (2020)
6. Anquetil, N., Royer, J., Andre, P., Ardourel, G., Hnetynka, P., Poch, T., Petrascu, D., Petrascu, V.: Javacompext: Extracting architectural elements from java source code. In: 2009 16th Working Conference on Reverse Engineering, pp. 317–318 (2009). DOI 10.1109/WCRE.2009.53
7. Aranda, J., Damian, D.E., Borici, A.: Transition to model-driven engineering - what is revolutionary, what remains the same? In: R.B. France, J. Kazmeier, R. Breu, C. Atkinson (eds.) Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings, Lecture Notes in Computer Science, vol. 7590, pp. 692–708. Springer (2012)
8. Atkinson, C.: Component-based Product Line Engineering with UML. Addison-Wesley object technology series. Addison-Wesley (2002)

9. Belina, F., Hogrefe, D., Sarma, A.: SDL with Applications from Protocol Specification. The BCS Practitioner. Prentice Hall (1991). ISBN 0-13-785890-6
10. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice: Second Edition, 2nd edn. Morgan & Claypool Publishers (2017)
11. Brunelière, H., Cabot, J., Dupé, G., Madiot, F.: Modisco: A model driven reverse engineering framework. Information and Software Technology **56**(8), 1012 – 1032 (2014). DOI https://doi.org/10.1016/j.infsof.2014.04.007. URL http://www.sciencedirect.com/science/article/pii/S0950584914000883
12. Bucchiarone, A., Cabot, J., Paige, R.F., Pierantonio, A.: Grand challenges in model-driven engineering: an analysis of the state of the research. Software and Systems Modeling **19**(1), 5–13 (2020)
13. Cabot, J., Gogolla, M.: Object constraint language (ocl): A definitive guide. In: M. Bernardo, V. Cortellessa, A. Pierantonio (eds.) Formal Methods for Model-Driven Engineering, Lecture Notes in Computer Science, vol. 7320, pp. 58–90. Springer Berlin Heidelberg (2012)
14. Cabot, J., Teniente, E.: Constraint support in MDA tools: A survey. In: A. Rensink, J. Warmer (eds.) Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings, Lecture Notes in Computer Science, vol. 4066, pp. 256–267. Springer (2006)
15. Charfi, A., Schmidt, A., Spriestersbach, A.: A hybrid graphical and textual notation and editor for uml actions. In: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA '09, pp. 237–252. Springer-Verlag, Berlin, Heidelberg (2009)
16. Ciccozzi, F.: On the automated translational execution of the action language for foundational uml. Software & Systems Modeling **17**(4), 1311–1337 (2018)
17. Ciccozzi, F., Malavolta, I., Selic, B.: Execution of uml models: a systematic review of research and practice. Software & Systems Modeling **18**(3), 2313–2360 (2019). DOI 10.1007/s10270-018-0675-4
18. Dehaghani, S.M.H., Hajrahimi, N.: Which factors affect software projects maintenance cost more? Acta Informatica Medica **21**(1), 63–66 (2013). DOI 10.5455/AIM.2012.21.63-66. URL https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3610582/
19. Di Ruscio, D., Eramo, R., Pierantonio, A.: Model transformations. In: M. Bernardo, V. Cortellessa, A. Pierantonio (eds.) Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures, pp. 91–136. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
20. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language, 1st edn. Addison-Wesley Professional (2012)
21. Forward, A., Badreddin, O., Lethbridge, T.C., Solano, J.: Model-driven rapid prototyping with umple. Software: Practice and Experience **42**(7), 781–797 (2012)
22. Friedenthal, S., Moore, A., Steiner, R.: A Practical Guide to SysML: Systems Modeling Language. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2008)
23. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., USA (1995)
24. Gogolla, M., Bohling, J., Richters, M.: Validating uml and ocl models in use by automatic snapshot generation. Software and Systems Modeling **4**(4), 386–398 (2005)
25. Golra, F.R., Dagnat, F., Bendraou, R., Beugnard, A.: Continuous process compliance using model driven engineering. In: Y. Ouhammou, M. Ivanovic, A. Abelló, L. Bellatreche (eds.) Model and Data Engineering - 7th International Conference, MEDI 2017, Barcelona, Spain, October 4-6, 2017, Proceedings, Lecture Notes in Computer Science, vol. 10563, pp. 42–56. Springer (2017)
26. Group, O.M.: The OMG Unified Modeling Language Specification, version 2.4.1. Tech. rep., Object Management Group, UML 2.4 Superstructure Specification available at http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF (2011)
27. Guermazi, S., Tatibouet, J., Cuccuru, A., Seidewitz, E., Dhouib, S., Gérard, S.: Executable modeling with fuml and alf in papyrus: Tooling and experiments. In: Proc. of the 1st International Workshop on Executable Modeling in (MODELS 2015)., pp. 3–8. Ottawa, Canada (2015). URL http://ceur-ws.org/Vol-1560/paper1.pdf

28. Hansen, M.O.: Exploration of UML State Machine implementations in Java. Master's thesis, University of Oslo, Norway (2011)
29. Jakumeit, E., Buchwald, S., Wagelaar, D., Dan, L., Ábel Hegedüs, Herrmannsdörfer, M., Horn, T., Kalnina, E., Krause, C., Lano, K., Lepper, M., Rensink, A., Rose, L., Wätzoldt, S., Mazanek, S.: A survey and comparison of transformation tools based on the transformation tool contest. Science of Computer Programming **85**, 41 – 99 (2014)
30. Kahani, N., Bagherzadeh, M., Cordy, J.R., Dingel, J., Varró, D.: Survey and classification of model transformation tools. Software and Systems Modeling **18**(4), 2361–2397 (2019)
31. Karsai, G., Taentzer, G., Mens, T., Gorp, P.V.: A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science **152**, 125 – 142 (2006). Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)
32. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise, 1 edn. Object Technology Series. Addison-Wesley (2003). ISBN 0-321-19442-X
33. Koskinen, J.: Software Maintenance Costs. Tech. rep., School of Computing, University of Eastern Finland, Joensuu, Finland (2015). URL `https://wiki.uef.fi/download/attachments/38669960/SMCOSTS.pdf`
34. Kurtev, I.: State of the art of qvt: A model transformation language standard. In: A. Schürr, M. Nagl, A. Zündorf (eds.) Applications of Graph Transformations with Industrial Relevance, pp. 377–393. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
35. Lano, K.: Advanced Systems Design with Java, UML and MDA, 1 edn. Computer Science. Elsevier (2005). ISBN 0-7506-6496-7
36. Lano, K., Kolahdouz-Rahimi, S., Yassipour-Tehrani, S., Sharbaf, M.: A survey of model transformation design patterns in practice. Journal of Systems and Software **140**, 48 – 73 (2018). DOI https://doi.org/10.1016/j.jss.2018.03.001. URL `http://www.sciencedirect.com/science/article/pii/S0164121218300438`
37. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model-Driven Architecture, 1 edn. Object Technology Series. Addison-Wesley (2002). ISBN 0-201-74804-5
38. Mottu, J., André, P., Coutant, M., Berre, T.L.: Shall we test service-based models or generated code? In: L.B. et al. (ed.) 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019, pp. 493–502. IEEE (2019). DOI 10.1109/MODELS-C.2019.00078
39. Niaz, I.A., Tanaka, J., Words, K.: Mapping uml statecharts to java code. In: in Proc. IASTED International Conf. on Software Engineering (SE 2004, pp. 111–116 (2004)
40. OMG: Semantics of a Foundational Subset for Executable UML Models (fUML), version 1.4. Tech. rep., Object Management Group (2018). URL `https://www.omg.org/spec/FUML/1.4/`
41. Orabi, M.H., Orabi, A.H., Lethbridge, T.C.: Concurrent programming using umple. In: MODELSWARD, pp. 575–585 (2018)
42. Paige, R.F., Matragkas, N., Rose, L.M.: Evolving models in model-driven engineering: State-of-the-art and future challenges. Journal of Systems and Software (2016). URL `//www.sciencedirect.com/science/article/pii/S0164121215001909`
43. Pepin, J., André, P., Attiogbé, J.C., Breton, E.: Definition and visualization of virtual meta-model extensions with a facet framework. In: S. Hammoudi, L.F. Pires, B. Selic (eds.) 6th Int. Conf. MODELSWARD 2018, Revised Selected Papers, Communications in Computer and Information Science, vol. 991, pp. 106–133. Springer (2018)
44. Perseil, I., Pautet, L.: A concrete syntax for uml 2.1 action semantics using +cal. In: Proceedings of the 13th IEEE International Conference on on Engineering of Complex Computer Systems, ICECCS '08, pp. 217–221. IEEE Computer Society (2008)
45. Pilitowski, R., Dereziñska, A.: Code generation and execution framework for uml 2.0 classes and state machines. In: T. Sobh (ed.) Innovations and Advanced Techniques in Computer and Information Sciences and Engineering, pp. 421–427. Springer Netherlands, Dordrecht (2007)
46. Planas, E., Cabot, J., Gómez, C.: Lightweight and static verification of uml executable models. Comput. Lang. Syst. Struct. **46**(C), 66–90 (2016)

47. Raibulet, C., Fontana, F.A., Zanoni, M.: Model-driven reverse engineering approaches: A systematic literature review. IEEE Access **5**, 14,516–14,542 (2017)
48. Raistrick, C., Francis, P., Wilkie, I., Wright, J., Carter, C.B.: Model Driven Architecture with Executable UML. Cambridge University Press (2004). ISBN 0-521-53771-1
49. Rierson, L.: Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance. Taylor & Francis (2013)
50. Roques, P., Vallée, F.: UML 2 en action: De l'analyse des besoins à la conception. Architecte logiciel. Eyrolles (2011). (in french)
51. Rugaber, S., Stirewalt, K.: Model-driven reverse engineering. IEEE Software **21**(4), 45–53 (2004). DOI 10.1109/MS.2004.23
52. Schader, M., Korthaus, A.: Modeling Java threads in UML. In: M. Schader, A. Korthaus (eds.) The Unified Modeling Language – Technical Aspects and Applications, pp. 122–143. Physica-Verlag, Heidelberg (1998)
53. Sebastián, G., Gallud, J.A., Tesoriero, R.: Code generation using model driven architecture: A systematic mapping study. Journal of Computer Languages **56**, 100,935 (2020)
54. Selic, B.: Personal reflections on automation, programming culture, and model-based software engineering. Automated Software Engineering **15**(3), 379–391 (2008). DOI 10.1007/s10515-008-0035-7
55. Shanks, G., Tansley, E., Weber, R.: Using ontology to validate conceptual models. Commun. ACM **46**(10), 85–89 (2003). DOI 10.1145/944217.944244
56. Sindico, A., Natale, M.D., Sangiovanni-Vincentelli, A.L.: An industrial system engineering process integrating model driven architecture and model based design. In: R.B. France, J. Kazmeier, R. Breu, C. Atkinson (eds.) Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings, Lecture Notes in Computer Science, vol. 7590, pp. 810–826. Springer (2012)
57. Somogyi, F.A., Asztalos, M.: Systematic review of matching techniques used in model-driven methodologies. Software and Systems Modeling **19**(3), 693–720 (2020)
58. Tisi, M., Jouault, F., Saidi, Z., Delatour, J.: Enabling ocl and fuml integration byźtransformation. In: Proceedings of the 12th European Conference on Modelling Foundations and Applications - Volume 9764, pp. 156–172. Springer-Verlag, Berlin, Heidelberg (2016)
59. Van Cam Pham, A.R., Gérard, S., Li, S.: Complete code generation from uml state machine. In: Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, vol. 1, pp. 208–219 (2017)
60. Van Deursen, A., Visser, E., Warmer, J.: Model-driven software evolution: A research agenda. In: In Proc. Int. Ws on Model-Driven Software Evolution held with the ECSMR'07 (2007)
61. Vogel-Heuser, B., Schütz, D., Frank, T., Legat, C.: Model-driven engineering of manufacturing automation software projects – a sysml-based approach. Mechatronics **24**(7), 883 – 897 (2014). 1. Model-Based Mechatronic System Design 2. Model Based Engineering
62. Weilkiens, T.: Systems Engineering with SysML/UML: Modeling, Analysis, Design. The MK/OMG Press. Elsevier Science (2008)

A Appendix A

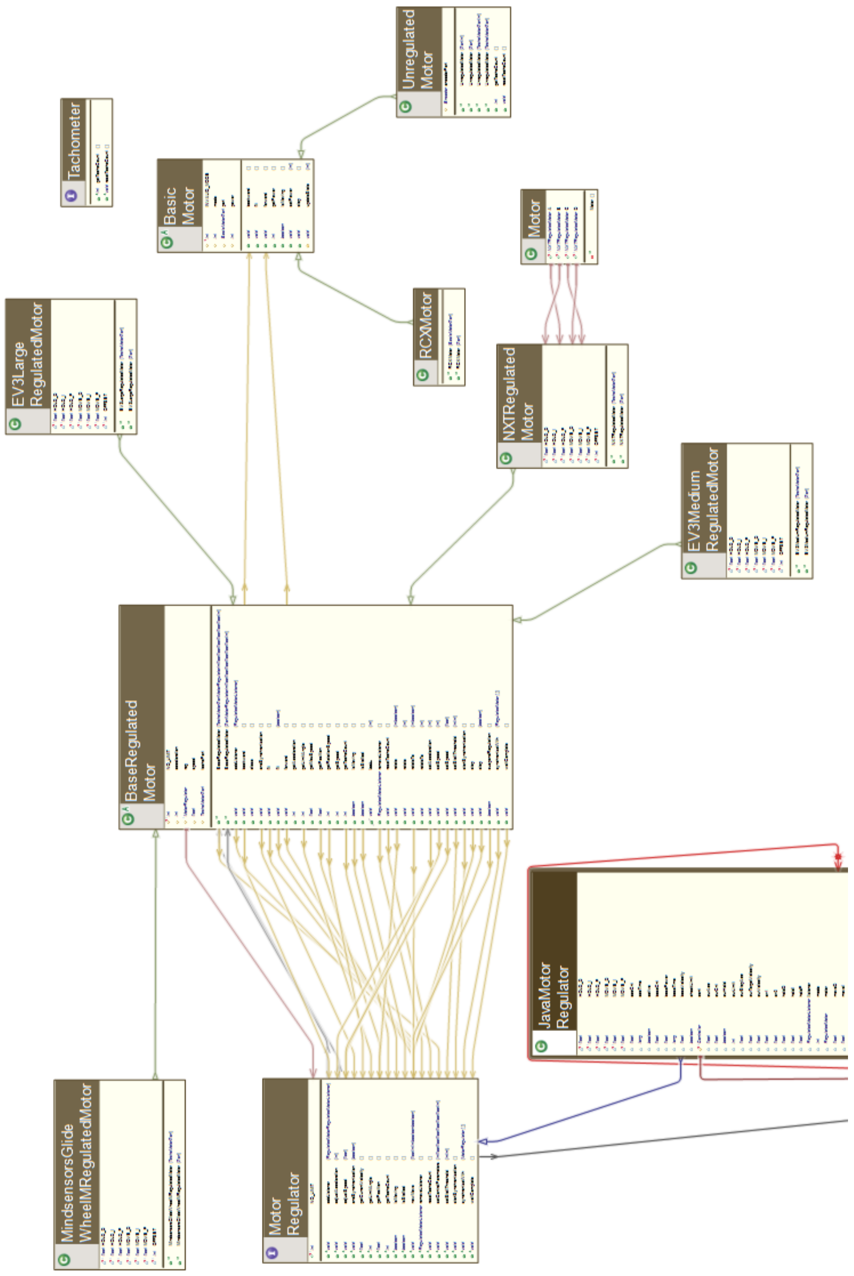


Fig. 30 Applying AgileJ RE process on *Motor* package of Lejos Library

Contents