Refining Automation System Control with MDE

Pascal André[1] and Mohammed El Amin Tebib[2]
[1]*LS2N CNRS UMR 6004 - University of Nantes, France*
[2] *Davidson, Paris , France*
{firstname.lastname}@ls2n.fr

Keywords: Model Driven Engineering, Cyber-physical systems, Statecharts, Refinement, Model Transformation process

Abstract: Software engineering gets increasing matter in cyber-physical systems and pervasive computing. In the industry of the future, software must not only be of high quality in terms of reliability and performance, but also reinforce the need of continuous evolution to fit to physical system changes. Model engineering aims to shorten the development cycle by focusing on abstractions and partially automating code generation. 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 model covers structural, dynamic and functional aspects of the studied system. The target code is that of a system distributed over several devices. We advocate a model transformation approach in which transformations remain simple, the complexity lies in the process of transformation that is adaptable and configurable. To conduct the experiments, the models are written in UML (or SysML) and programs deployed on Android and Lego EV3. We report the lessons learnt for future work.

1 Introduction

The software plays an increasing role in the industry, at the level of products, which become connected, or at the automated manufacturing of these products and also in the field of services with the assistance of robots and artificial intelligence. Programs such as the European "Industrial Leadership in H2020" make it possible to converge efforts in cybernetics. As a result, the methods and tools of software development impact the entire development of these cyber-physical systems.

Conversely the physical world propagates influences the software development *e.g.* reconfiguring the physical part implies to update software. 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 lifecycle must support continuous evolution. Software maintenance costs, which traditionally accounted for 70% of the total cost of the software, still increase (**?**; **?**). We are convinced that Model Driven Engineering (MDE) is a mandatory approach to develop long-term software systems.

Model engineering aims to collapse the development cycle by focusing on abstractions, property checking, and partial coding automation. Reasoning to verify the system properties can happen at the level of the models (there are methods and tools for this) but more hardly at the code level, because of the complexity inherent to the implementation and distribution details (security, communication...). Code generation exists for many years for operational models, in the compilation of source programming language or the grammar-based generators for dedicated languages (or *Domain Specific Language DSL*) such as the ancestor lex-yacc for C, antrl, parsers XML or JSON, or more recently XTEXT... On the other hand, 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 for a software engineering work. Automation becomes more cost-effective than (manual) development when considering the evolutionary maintenance of functional, non-functional and technical requirements (hardware modification for example).

In this paper, we investigate the step-by-step transition from abstract heterogeneous models to executable source code to reduce the time between software analysis and production. By heterogeneous model we mean a model that covers structural, dynamic and functional aspects of the modelled system. We also assume general and expressive languages to describe these models such as UML or SysML (**?**). 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 calculations. The code is that of a system distributed over several devices. We share the vision given in (**?**) that reduces MDE to two main ideas: raising the level of abstraction and raising the degree of computer automation.

The transition from model to code is still a challenge from the point of view of automation (or assistance). The code generators of many UML/SysML tools typically produce skeletons where the bulk of the development remains to be done. Running MDA solutions are dedicated to one technical platform which inevitably reduce the applicability range. The context chosen here remains humble, it consists in implementing a model-based approach (model engineering) (**?**) to build software (safe) control automata. We recommend a model transformation approach in which transformations remain simple. The complexity is postponed to the transformation process which must be adaptable and configurable. Related *engineering practice* challenges of (**?**) are: managing automation, usability of tools, hybrid migration approaches and external interfaces.

In this paper we focus on methodological issues not on technical ones. The work reported is empirical using a trial and error methods to find suitable solutions. We conduct experiments with students. We start from models written with expressive but heterogeneous semantic languages such as UML or SysML and we target programmable controllers, Lego EV3 in this case, remotely controlled by an app written in Java deployed on Android. We implement and compare three ways to get the source code: manual, code generation and model transformation. The experiments illustrate the complexity of the task. The lessons learnt from these experimental works open tracks for future vision and future work.

The paper is structured as follows. Section **??** introduces the context elements and then presents the illustrating case study, a simple home automation (domotics) system. We overview t There are three approaches to refine models to code: fully manual, fully automatic and stepwise refinement which can be seen as a kind a compromise approach. The two primer are overview in Section **??** while the latter is given more details in Section **??** and experimentations in Section **??**. The different approaches are illustrated on the case study. Lessons learnt are discussed in Section **??**. Finally, Section **??** summarises the contribution and draws the new vision perspectives.

2 Context

The goal is to set up a software production chain based on models for distributed automation systems. In particular, we are interested here in programmable controllers having a "real" execution environment that take into account operating, safety and performance constraints (**?**). Some properties are general (safety, liveness), others are related to the environment or the system itself (energy, dangerousness, quality of service...). From the software point of view we consider at least two levels:

- the modelling and simulation level, where the individual and collective behaviours are described and the constraints are analysed in the form of a digital image (digital twin). We can use one or more modelling languages such as UML (**?**), SysML (**?**), Kmelia (**?**)...) and associated verification or simulation tools, etc. The models at this level will be called logical models (or analysis) in the sense that the technical details are not yet given. The analysis model plunged into a technical environment will be called a design model, as illustrated by the Y development model of the Figure **??** coming from (**?**).
- 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.

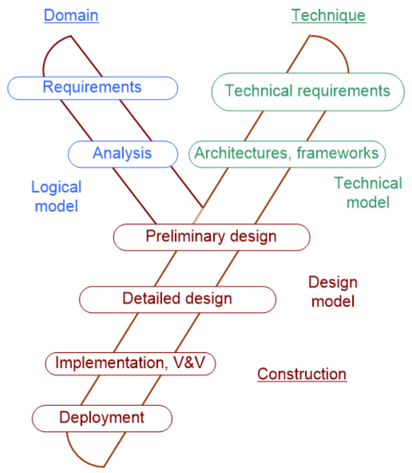


Figure 1: 2TUP Unified Process

Intermediate steps can be processed to reach the implementation level in the spirit of the model-driven development (MDD) (**?**) and software product lines (Software Product Lines) (**?**). In MDD, it is essential to ensure the model correctness before starting the process of transformations and code generation (**?**).

This reduces the high cost of late detection of errors (**?**). Whatever is the modelling language, the models are considered to be sufficiently detailed to be made executable[1]. This allows, in addition to several verification techniques based on proof of theorem or *model checking*, to consider testing these models (**?**).

Case Study We 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. We assume the system behaviour to be simple enough to be understood.We provide a logical model of the case given in the expressive UML notation *i.e.* the class diagram of Figure **??** including the operation signature.

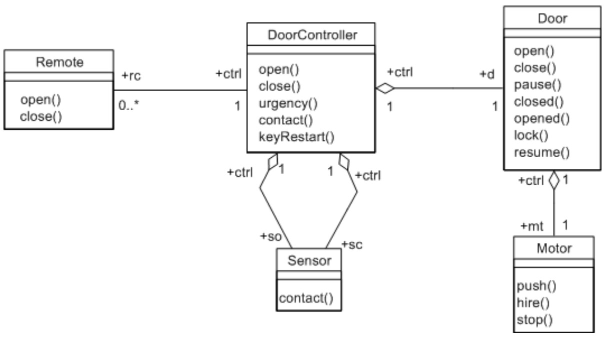


Figure 2: Diagramme de classes - porte de garage

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, the door opens completely and triggers the open sensor `so`, the motor stops. Pressing the `close` button will 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 state diagram of Figure **??** describes the behaviour of the door controller. The actions on the doors are transferred to the engines by the door itself. 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.

The verification of logic models includes static analysis, type checking, *model checking* for communications, theorem proving for functional contract as-

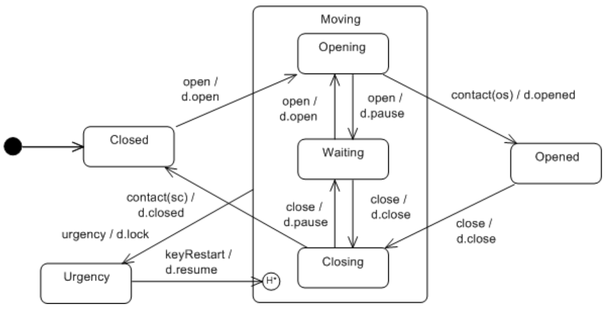


Figure 3: Door controller State diagram

sertions, and testing (**?**), 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.

3 From Models to Implementation

Software design is the activity that implements requirements inside a technical platform *cf.* Figure **??**. It is an engineering activity where decisions have to be taken that affect the quality of the result. Key design concepts are abstraction, architecture, pattern 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. We assume in the following a given technical target. For example, our experimentations are led with Lego EV3 controller (java/Lejos) and android. 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) manual development, (ii) model transformation process, (iii) automatic code generation. Next we overview solutions i) et iii) which are at opposite side of the automation spectrum. The intermediate solution ii) will be detailed in Section **??**.

Manual Development

The case study was given to different groups of students. The starting point was the logical model figured in Section **??**, documentation on EV3 Lejos, tutorial examples and also articles like (**?**; **?**; **?**).

A first version[2] has been proposed with the physical prototype of Figure **??** that has been extended later until having an android app to play the remote device

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

[2]`https://github.com/demeph/TER-2017-2018`

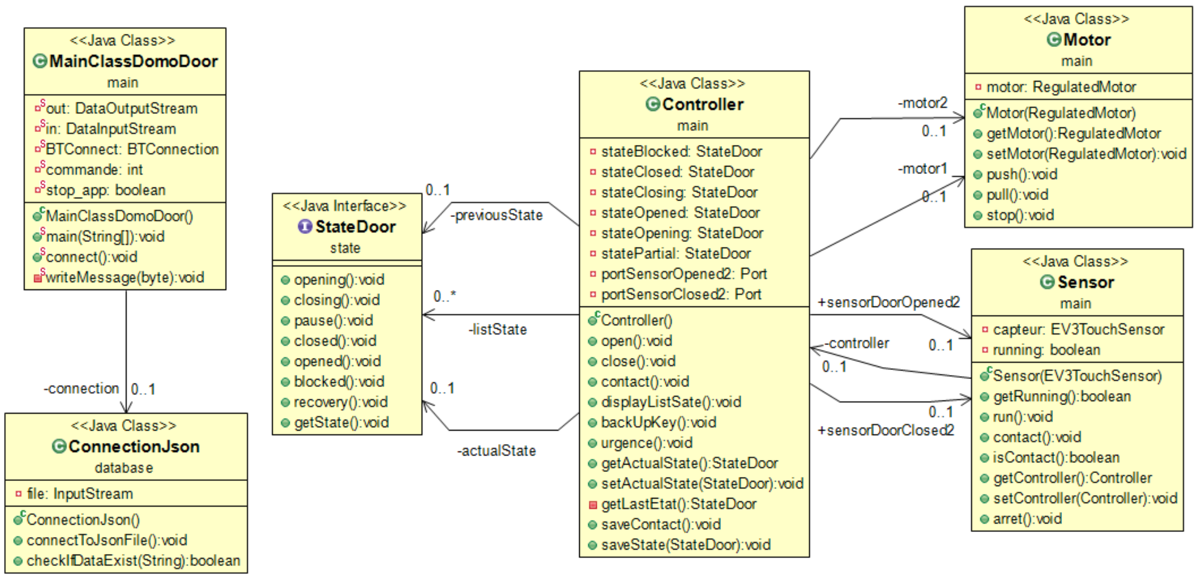


Figure 4: Class Diagram of the door application (v2)

with bluetooth connection. Another version[3] led to the class diagram of Figure **??**.

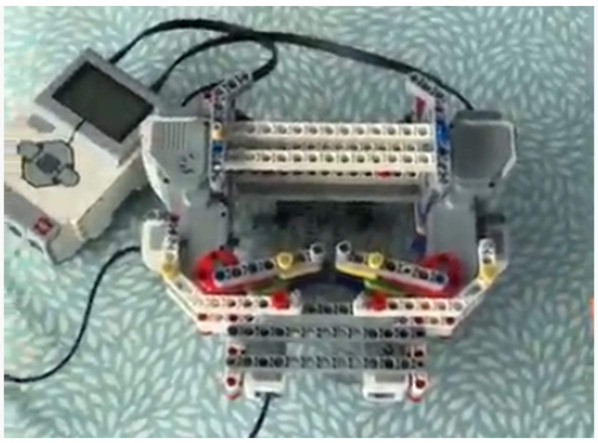


Figure 5: Lego prototype of the door system

The code they produced does not necessarily conform to the logical models which were perceived as a documentation reference rather than an abstract model. The detailed design decisions are different. In version `v1` the students used enum types to implement state machines while a *state pattern* has been chosen in version `v2` of Figure **??**. The remote device was also implemented in different ways according to the student experience and motivation: from java swing GUI with wired TCP-IP communication with EV3 or Android app with bluetooth or wifi connection. Finally, the prototype of Figure **??** uses two motors for two door swings while a single door and motor were specified in the logical model. Isolating the various design choices is a first step to rationalize development in a refinement process (see Section **??**).

[3] `https://github.com/FrapperColin/2017-2018/tree/master/IngenierieLogicielleDomoDoor`

Code generation, animation

As opposed to manual development on the design spectrum, we find the automatic generation and animation. We illustrate in this section some research on this subject.

UML Tool Code Generation We compared the code generation facilities of UML (visual) Tools according to selected features of code generation. Our shallow panorama is definitely not an exhaustive study but tries to find prototype tools in some big categories (free, eclipse modelling initiatives, case tools...) and to show tendencies. The model interoperability is ensured by the XMI (XML profiles for UML, SysML or MOF), this standard exists in almost as many version as the modelling languages. Unfortunately, the diagram interchange standard is less interoperable. The tools enable to edit class diagrams (CD) and State transition diagrams (STD) and (re)generate source code. Despite a link exists between method body and activity diagrams we did not seen concretely it in the tools. Usually they provide the operation signature but rarely more except when a *round trip engineering* is allowed that enables to attach target code fragments to model operations in order to keep them when regenerating the code after model evolution.

- StarUML is a representative of free UML editors. StarUML generates the structural and type parts of the class diagrams but not the operations body and STDs.

- Papyrus and Modelio represent the *Eclipse Modeling open source* ecosystem with an active community. The code generation of Papyrus includes behaviours to operations with an incremental that overrides the code generation. Moreover one can add new generators in addition to the native C++ and Java. But we did not found an adequate plugin for statemachines and also the associations have not been generated in the Java code. Modelio includes roundtrip but 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 (not managed) method is under the responsibility of the developer. In this category of tools, one can also mention the Obeo tools, UML Designer and Acceleo, or the new Polarys project including Papyrus and Topcased.
- Yakindu is a tool for STDs proposed by Itemis. The STD code generation provides a detailed implementation but for one STD only, this is not attached to classes and therefore avoids the notion of message communications. Note that a detailed presentation of Java generation from UML state machines can be found in (**?**).
- Visual Paradigm is very rich in standards and features. It supports the generation of class diagrams and state transitions 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.
- IBM Rational Rhapsody, inspired by the authors of OMT, the main contribution to 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. Again we did not have access to the generated code to estimate the manual programming part.

Most of the tools are not bound to one language only, for example Modelio, Papyrys or Visual Paradigm integrate different OMG standards such as SysML, BPMN... Consequently some of the tools can have esoteric notations for some model elements. Also several tools cover a larger perimeter than system modelling, covering parts of enterprise architecture.

Table **??** summarizes some selected features. MOM (Message Oriented Middleware) indicates concrete implementations of the abstract message send or signal raising. We did not retained here the possibility of treating the (real) time as for example with the MARTE profile. The license may be OpenSource, Free, Commercial. We call *API Mapping* a feature that lets you attach model elements to predefined elements in libraries or *frameworks*. This is different from round trip code references for the methods that implement the class operations.

Table **??** illustrates the fact that, to our knowledge, no tool deals clearly yet with the problem of (heterogeneous) communication middleware or to *mapping* with technical features (high level for architectures, low level for framework libraries) except embedding in a given context like Java, .NET, REST... However, we noted that Visual Paradigm can integrate deployment models in the cloud. IBM Engineering Systems Design Rhapsody is rather dedicated to detailed design. Some tools also offer persistence features (*e.g.* mapping object relations or SQL) that we have not be retained here since we focus on automation. Note also that during our experiments, we used Papyrus to generate class diagrams in Java.

Executable UML Generating code from UML for a given technical architecture or even a given *framework* is still reserved to simple cases like the CRUD (Create, Read, Update, Delete) application generation on simple relational databases. The technical *framework* must be generic and complete, but also the models must be simple. We can also animate or execute specifications, which are then qualified as operational. The prerequisite is to have complete models of the system structure (class or component diagrams), of its dynamic behaviour (state-transitions diagrams) and its functional behaviour (activity diagrams).

In any case, diagrams are usually not enough to specify semantics. One can add constraints written in OCL (**?**) (declarative language for invariant and pre/post-condition assertions) or pseudo-code written in a language conforming to the semantics of UML actions. The concept of action is present in activity and state transitions diagrams.

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: (i) *Action Specification Language (ASL)* was defined for iUMLLite of Kennedy-Carter (Abstract Solutions) supporting xUML (**?**). (ii) *BridgePoint Action Language (AL)* (and the derived SMALL, OAL, TALL) proposed by Balcer & Mellor was implemented in xtUML of Mentor Graphics (**?**). (iii) *Kabira Action Semantics (Kabira AS)* proposed by Kabira Technologies (and later TIBCO Business Studio). (iv) The normative telecom SDL (**?**) has also

Table 1: Comparison of some tools with code generators

	Star UML	Papyrus	Yakindu	Modelio	VisualParadim	IBM rational rhapsody
UML - XMI	2.0	2.5	-	2.4.1	2.0	2.4.1
CD	√	√	-	√	√	√
STD	-	-	one only	√[1]	√	√
Operations	-	incremental	-	RoundTrip	RoundTrip	√
Round-trip	-	override	-	√	√	√
MOM	-	-	-	-	-	-
API Mapping	-	-	-	-	-	-
Licence[d]	F, C	O	F, C	O	C	C

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

be used to provide a semantics as a UML profile. Others are *Platform Independent Action Language (PAL)* of Pathfinder Solutions, or SCRALL (**?**) which had a visual representation, +CAL (**?**). All these efforts led to semantics for a subset of executable UMLs, called `fUML` (*Semantics of a Foundational Subset for Executable UML*) (**?**), with now a normalized concrete syntax `Alf`. A reference implementation exists[4] that we plan to integrate later in the project. We did not experimented the "executable" approach yet since out goal was not to execute or to simulate UML models but to design applications.

4 Toward a design transformation process

In MDE (**?**) a transformation process implements refinements from *Platform Independent Model (PIM)* to (more) concrete *Platform Specific Model (PSM)*. As illustrated by Figure **??**, the software design consists in "weaving" the logic model and to the technical infrastructure (*platform*) to obtain *in fine* an executable model. We draw the reader's attention to the following observations : (i) Only a complete (and consistent) logical model enables to reach an executable source code. Model transformation can infer but not invent. (ii) The generation of code itself is not conceivable as a single transformation step, because of the semantic distance between the logical model and the technical target, especially if it is composed of orthogonal but related aspects, called *domains* (*e.g.* persistence, HMI, control, communications, inputs/outputs) on which the logical model must be "woven". (iii) Design is by nature an engineering activity, linked to the designers' experience (*cf.* page **??**). A process can be automated only if all the activities are known precisely. (iv) 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. (v) The transformations comply an algorithmic style (*e.g.* Kermeta[5]) or a rule-based style (*e.g.* ATL[6]). Working with small transformations enables to make them more verifiable and reusable.

On the basis of these considerations we adopt a principle that we call *small step transformations*. Complexity (or intelligence) should not be in the atomic transformations but in the transformation processes. A complex transformation is hierarchically composed of other transformations, until atomic transformations. Figure **??** sketch the aspects to consider to refine towards implementation. These macro-transformations use configuration information.

- T1 starts by structuring subsystem applications with a mapping on the application architecture by describing the APIs and the communication protocols. Of course, if the logical model includes component and deployment diagrams for a preliminary design in figure **??**, the deployment transformation will be simplified.
- In T2, for each kind of communication, the UML message send are refined according to the protocol under consideration (called MOM in Table **??**). At least, in a sequential implementation, we can have a message sending in the target OOP language (Java, C ++ or C#).
- T3 transforms state machines into a OOP model which in general does not natively include this concept. We can use either `enum` types or *State pattern* depending on the situation. This thorny problem is discussed in Section **??**.
- T4 aims to match 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 message send or method call according two ways: (i) encapsulate

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

[5]`http://diverse-project.github.io/k3/`
[6]`https://www.eclipse.org/atl/`

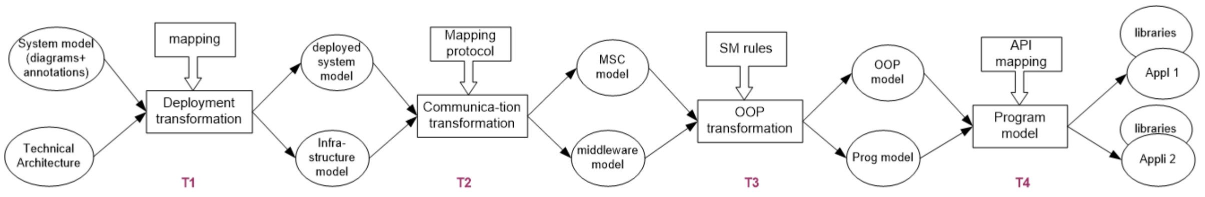


Figure 6: General transformation process

the predefined class in the model class and use it by delegation, the advantage is to preserve the model API. (ii) substitute the class of the model by the predefined class and rename the calls to the API of the model (we lose traceability).

It should be noted that all the parameters and decisions of the transformation must be stored to replay the transformation process in case of evolution of the initial model.

The process of Figure **??** is abstract but generic. Simplifications exist.

- Iteration We are not here in a so-called *round trip* approach but we can use it to optimize the history.
- Animation, execution When the technical environment is mastered, the transformation will "plunge" the model into the *framework* to make it executable. Refinement techniques to Java can be found in (**?**). We will illustrate this situation in the Section **??**.

Transformation T1 and T2 have not been implemented. In the following we provide details on the T3 transformation.

5 Experimentations

In this section, we report elements of students' experimentation for model to Java transformations. The T1 and T3 issues are implemented manually here.

Model Transformation for verification and animation Kmelia is a formal service based component model supporting multi-aspect descriptions. It enables the verification of various kind of properties using internal or external test and proof tools (**?**). We manually translated parts of the UML models into Kmelia. This refinement approach is more interesting than UML 'executable' because the formal specification allows to check model properties, test it through test harnesses, and run the specifications via a java code generator. The purpose of the current article is not directly verification and validation but the work has been carried out and elements are provided about it in (**?**).

Model Transformation to Java programs with ATL Due to its expressivity and abstraction, we chosed ATL[7] to conduct these experiments. ATL is a model transformation language based on non-deterministic transformation rules. In a model to model (M2M) 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).

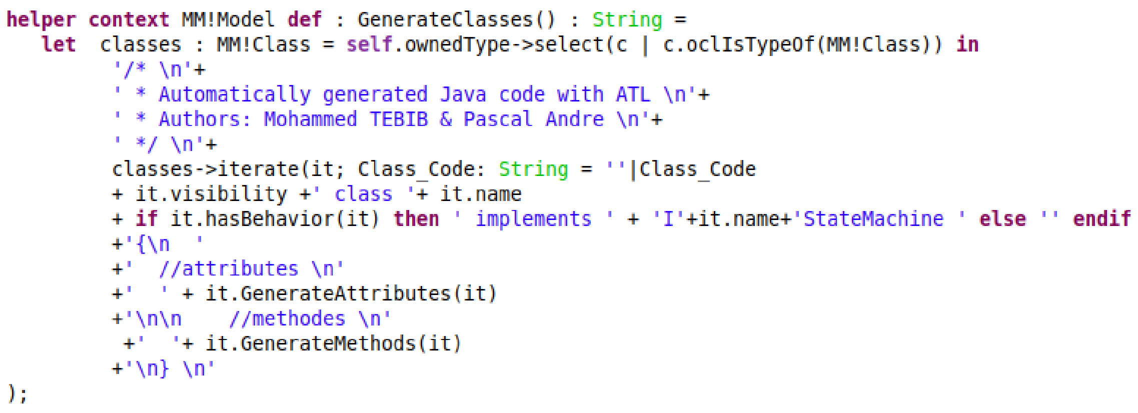
ATL proposes two modes of transformations from and refine. The from mode enables to create a model by writing all the parameters, all the attributes in the output model. The refine mode is used to copy anything that is not included in the rule into the output template and then apply the rule. A rule 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 meta-models or profiles by keeping UML as far as we can.

Consider the T3 (macro-)transformation of Figure **??**. A main issue is the state machine transformation. It is assumed there are only simple automata: no composites, no time, no history. Also a main restriction is that state machine inheritance through class inheritance is not allowed because the UML rules for it are fuzzy. 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.

We defined two ATL transformations to parse the XMI model and generate the Java code.

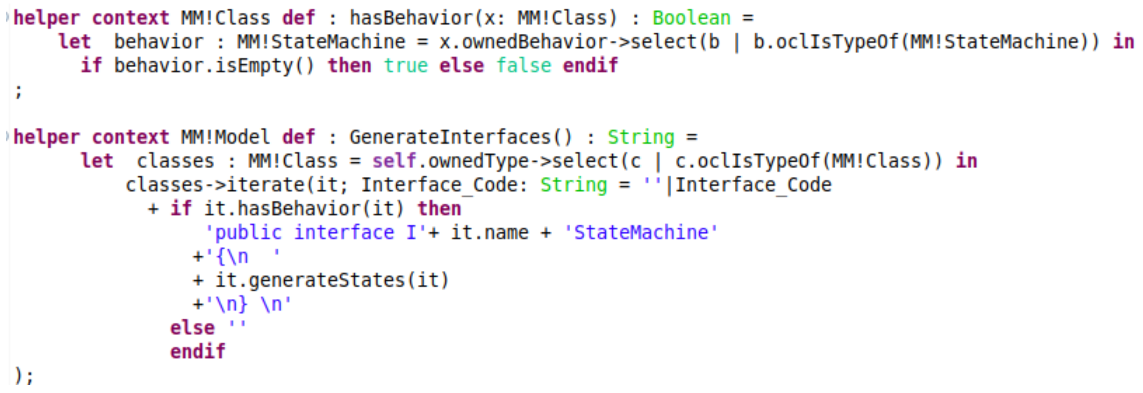
1. The rules of the first transformation generate the static structure based on parsing class diagrams, an overview of these rules is described in Figure **??**. Four ATL helpers (methods) are defined to parse the XMI model, each helper generates a piece of code that conforms to the Java grammar (syntax). Figure **??** shows the implemented ATL

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



```
helper context MM!Model def : GenerateClasses() : 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
         + 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    //methodes \n'
          +'   '+ it.GenerateMethods(it)
         +'\n} \n'
);
```

Figure 7: ATL transformation rule for classes



```
helper context MM!Class def : hasBehavior(x: MM!Class) : Boolean =
    let  behavior : MM!StateMachine = x.ownedBehavior->select(b | b.oclIsTypeOf(MM!StateMachine)) in
      if behavior.isEmpty() then true else false endif
;

helper context MM!Model def : GenerateInterfaces() : String =
      let  classes : MM!Class = self.ownedType->select(c | c.oclIsTypeOf(MM!Class)) in
          classes->iterate(it; Interface_Code: String = ''|Interface_Code
            + if it.hasBehavior(it) then
                 'public interface I'+ it.name + 'StateMachine'
                +'{\n  '
                + it.generateStates(it)
                +'\n} \n'
              else ''
              endif
);
```

Figure 8: ATL transformation rule for state diagrams

helper `GenerateClasses()` used to generate the Java class' code structure, this helper calls other ones (`GenerateAttibutes()`, `GenerateMethods()`) to obtain the complete code.

2. The second transformation is still in the first stage, the goal is to provide a complete generation engine from statecharts. The existing rules are described Figure **??**. Following a `State` pattern, we use helpers to generate the states machine related to each object, this object should implements the interface that declares the fundamental methods to initialize, enter, and exit a state machine. Figure **??** is a screenshot of the the generated class and Figure **??** is the interface of the door states.

A third transformation creates a path variable `_etatCourant` for each class holding a state machine and and a memory variable `_etatPrecedant` is 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, we need to add a child element `OpaqueExpression` with two parameters: '`language`' which takes the value '`JAVA`' and '`body`'. The `body` parameter is initialized with the concatenation of the

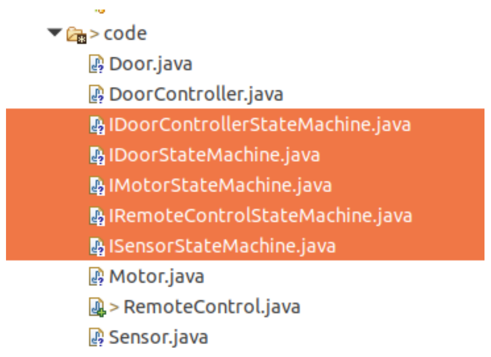
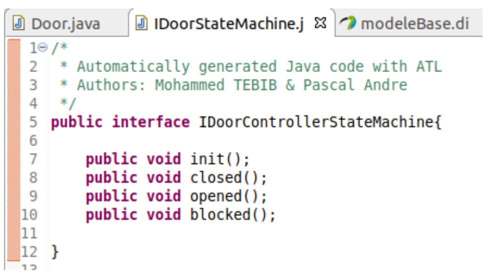


Figure 9: Some generated classes and interfaces



```
/*
 * Automatically generated Java code with ATL
 * Authors: Mohammed TEBIB & Pascal Andre
 */
public interface IDoorControllerStateMachine{

    public void init();
    public void closed();
    public void opened();
    public void blocked();

}
```

Figure 10: ATL transformation rule for state interface

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

A fourth transformation determines 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.

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

6 Discussions

We discuss here some lessons learnt from the above studies.

In cybernetics, SysML (**?**) is recommended for PLC design. We found an example of transmission control for Lego NXT[8]. Its SysML model is very detailed and can then be simulated by the Cameo tool. Modelling with SysML is suitable but it does not fundamentally change the problem. We used UML because it belongs to the student program.

The automatic generation of code, provides an incomplete model, which often does not even exploit the information of the model (OCL constraints, operation details). Although many studies have been conducted, the systematic study of Ciccozzi et al. (**?**) 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 (**?**),execution via C++ (**?**) or MoKa/Papyrus (**?**; **?**)..

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.

In the modelling of the development process in the form of a *workflow* of activities in Figure **??**, we ordered the transformations according to the impact: architectural choices (deployment, communications), general design choices (programming language), detailed design choices (*patterns*, *library mapping*). The 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. For example, coding state machines is subject to interpretation and strongly related to the execution model (**?**). Various strategies (enumerations, State pattern, execution engine) are possible for the same case study according to the nature of the automata. For example, enumerations are mandatory for an automaton with few states, while the State pattern is useful if the associated operations have different behaviour from one state to another and the number of states remains limited. Beyond 10 states, an instrumentation is necessary, connecting to a *framework* API for instantiation, inheritance, call, mapping... These strategies should be parameterisable in the transformation process. We also believe that several transformation tools should be used because the rule-based approach is unsuitable in some operational transformations like the one mentioned in (**?**). The anchoring in the execution support can be parametrised by a *mapping* to types, classes and operations. Accordingly the verification and the test of models can be inserted at any time in the process (**?**).

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.

The experimentations changed our vision. Design is more than refining, which assumes that the concrete models add details to abstract models. We perceive design as a set of parallel mapping transformations from a PIM to PDM (Platform Description Model) leading to a PSM, as illustrated for one transformation

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

in Figure **??**.

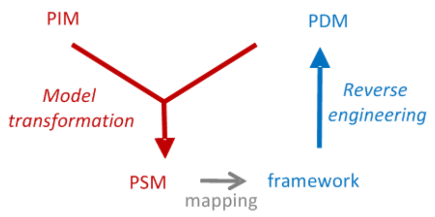


Figure 11: View of a mapping transformation

- Everything is model until the last level which is code generation.
- There are at least four frameworks to be compliant with the design process of Figure **??**.
- The more you derive to code the more you have transformations in parallel: one for T1, then one by deployment node in T2, then one by middleware in T3, etc.
- Transformations should be hierarchically composable in transformation processes (composite) and parametrisable.
- The missing link is usually the PDM which is now mandatory and can be obtain by model driven reverse engineering (**?**).

7 Conclusion

Software is becoming increasingly important in cybernetic systems. The maintenance of these systems implies a high reactivity of the development teams and highlights the need for industrialisation tools that go beyond integrated development and deployment platforms. To reduce a technical debt, we must 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. Enriching models, formalizing processes of refinement, making modular and personalizing development to rely on transformation tools are the tracks we follow.

Much work remains to be done, 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, it is necessary to be able to reverse engineering the design frameworks as PDM and to combine transformations written in different languages and that are interactive so that the designer influences the design choices.