A Case Study and Model Transformations Experimentations

We provide here additional information on one of the case studies, the door automate: an extract of the logical model and examples of student's implementation models.

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[17].

A.1 Logical Model

In cybernetics, SysML [25] is recommended for PLC design *e.g.* the detailed SysML model of a transmission control for Lego NXT[18] 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. 7 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.

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 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. 8).

The motor, when activated, can push or hire depending on the way we expect to move the door. (Fig. 9).

The state diagram of Fig. 10 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 and refined in the logical view of the analysis activity to be later reused as test cases in model or code verification. As

[17] 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.

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

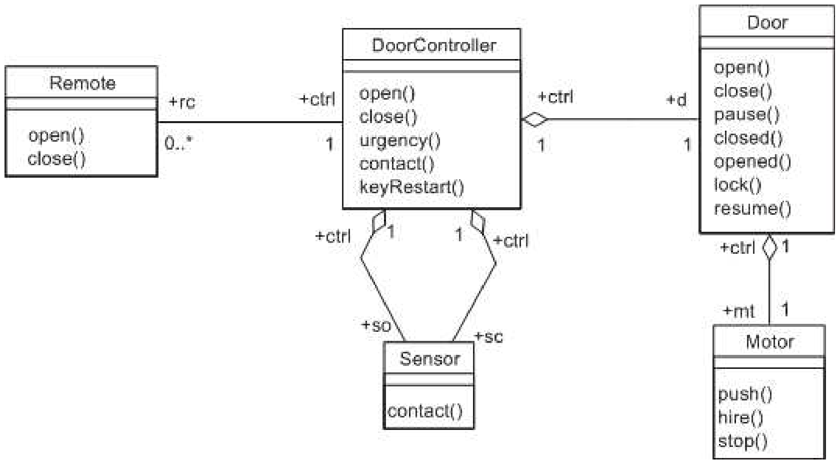

Fig. 7. Analysis Class diagram - garage door

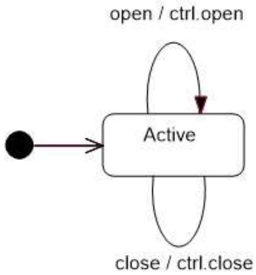

Fig. 8. Remote control State diagram - garage door

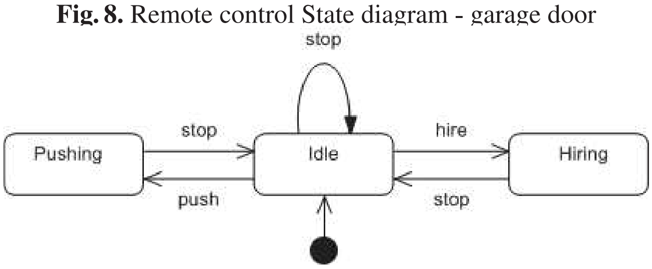

Fig. 9. Motor State diagram - garage door

an example, the sequence diagram of Fig. 11 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 [21] where advanced verification of properties require *model checking* for communications, *theorem proving* for functional contract assertions, and testing for behavioural conformance [20]. 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.

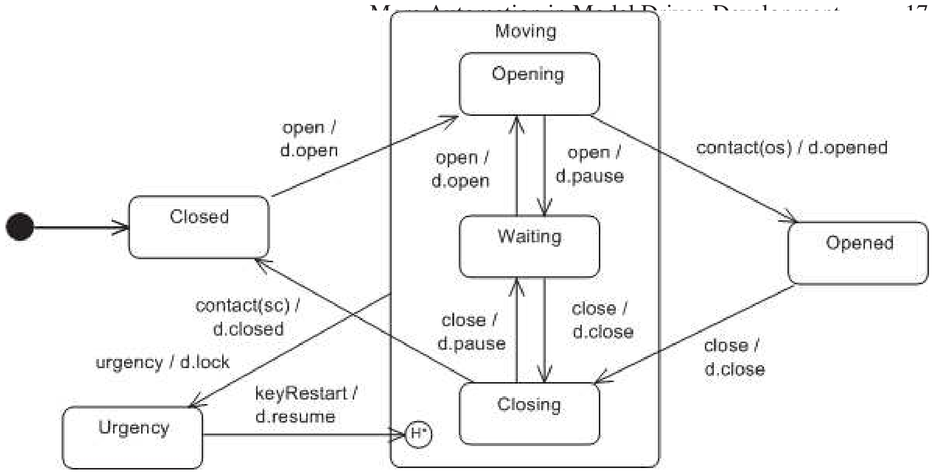

Fig. 10. Door controller State diagram - garage door

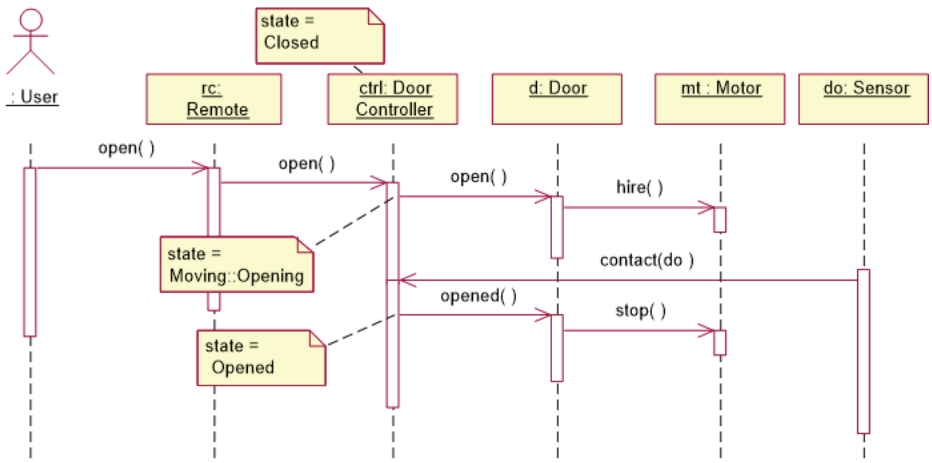

Fig. 11. Opening Sequence diagram - garage door

A.2 Technical Model

We assume in the following the 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. 12. 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.

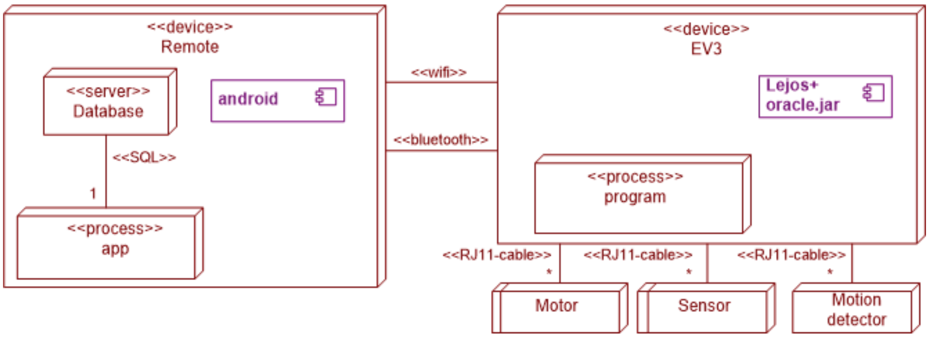

Fig. 12. Technical Architecture - EV3 and app

A.3 Forward Engineering Implementation Model

In the case of the garage door, a basic version called v1[19] was proposed in 2018 and led to the class diagram of Fig. 13. Its implementation with enumeration types for STDs has been proposed with the physical prototype of Fig. 14 that has been extended later until having an Android App to play the remote device with Bluetooth connection. Note that the students implemented the door by a two panels system activated by two motors.

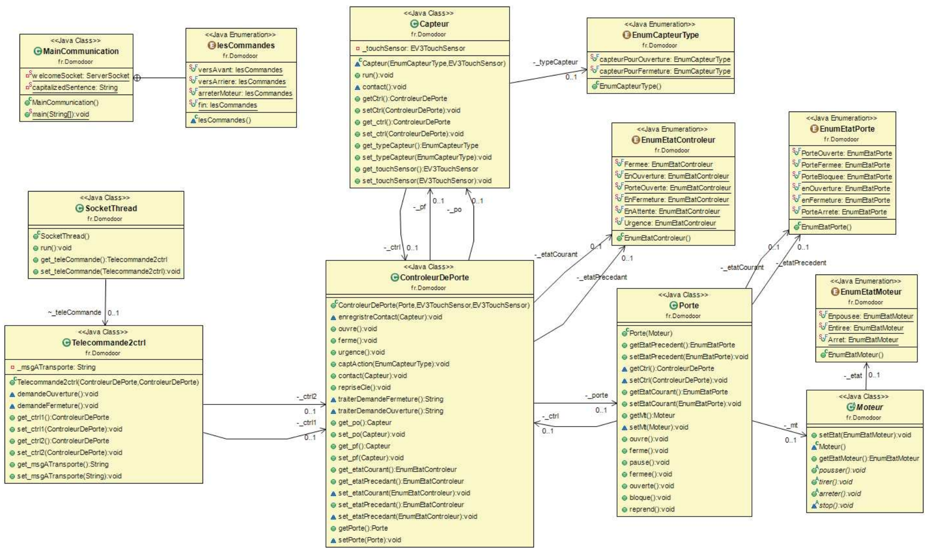

Fig. 13. Class Diagram of the door application (v1)

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

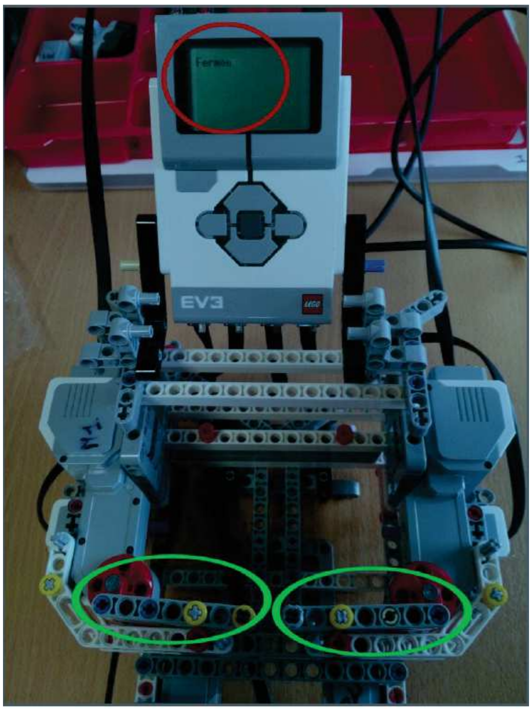

Fig. 14. Lego prototype of the door system

Another implementation, called v2[20] led to the class diagram of Fig. 15.

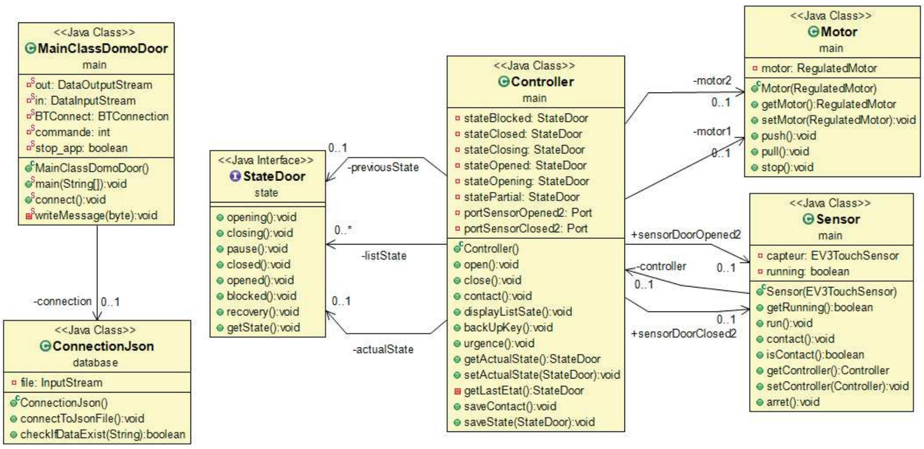

Fig. 15. Class Diagram of the door application (v2)

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

B Macro Transformations Experimentations

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

B.1 Deployment Transformation (T1)

The T1 composite transformation was designed manually by providing a deployment model of Fig. 16 from the analysis models of Section A.1 and the technical architecture of Section A.2. The bluetooth protocol has been selected to connect the EV3 and the remote computer.

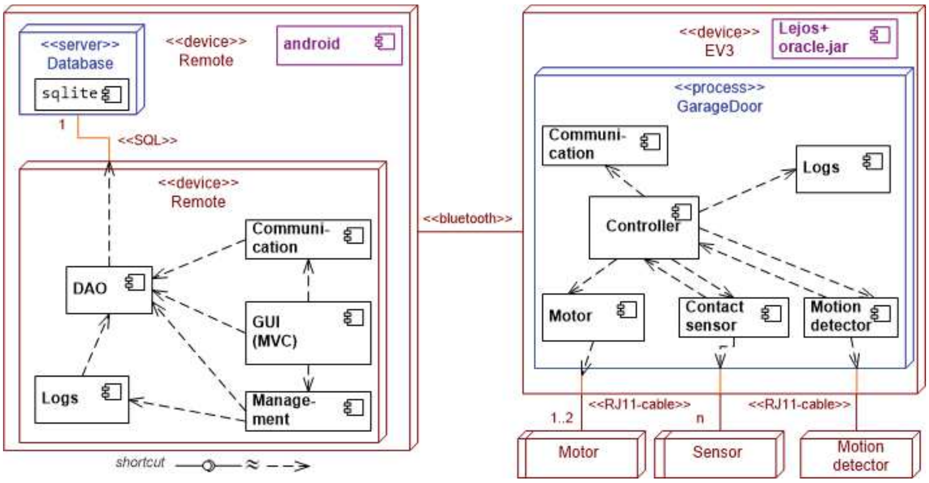

Fig. 16. Deployment diagram - garage door

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.

B.2 MOM Transformation (T2)

The problem is to refine UML communications according to the basic causality principle of UML[21]. *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*

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

is matched to that message. The dispatching method by which a particular behavior is associated with a given message depends on the higher-level formalism used and is not defined in the UML specification (i.e., it is a semantic variation point)".

During an object interaction *e.g.* in a sequence diagram, objects exchange messages (synchronous/ asynchronous, call and reply, 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[22].

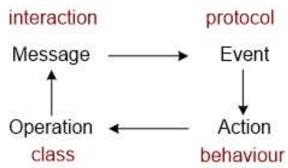

Fig. 17. Basic causality principle

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 connection 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[23] 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[24] 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

[22]Note that this principle binds sequence, state-transition and class diagrams providing a way to check some inter-diagram consistency rules [21] but also a way to organise models.

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

[24]Abstract to raising signals or time events.

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 [26].

B.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. 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 [27] 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[25] is to determine the main inheritance flow either the first in the multiple inheritance order or by a metric

[25] 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.

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 [27] 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.
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)[26] 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.

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. 18 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. 18. 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),

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

Methods(MM!Operation). As described in Fig. 19, 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.

```
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. 19. From UML elements to Java elements

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. 20) 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 [27], 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 vari-

ables are Property elements typed by the enumeration type. To initialize the current state, a child element OpaqueExpression is added, with two parameters: ’language’ which takes the value ’JAVA’ and ’body’. The body parameter is initialized with the concatenation of the enumeration 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.

```
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. 20. From STD to Java elements

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

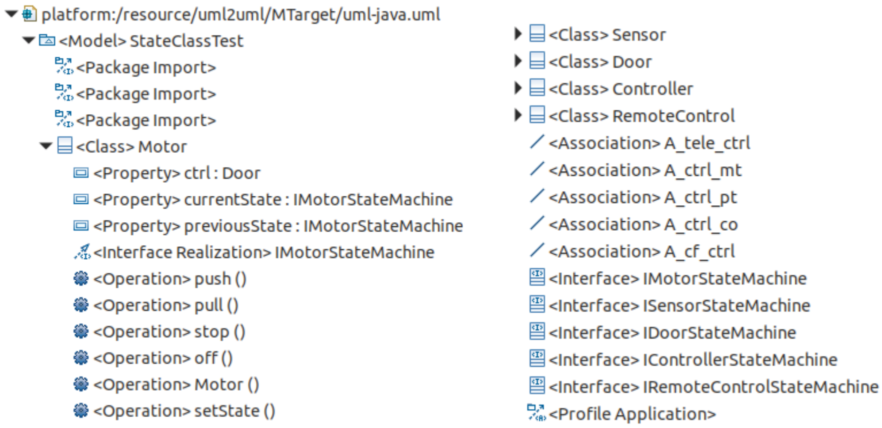

Fig. 21. The Java elements generated for the garage door

B.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. **??**). As mentioned in Section **??**, 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.

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 **??**.
   - 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 [28].
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. **??**. 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.

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. 22 organises the parse-generate process by calling sub-rules.
   – 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. 23).

```
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. 22. ATL transformation rule for classes

```
--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. 23. ATL transformation rules for attributes

```
--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. 24. ATL transformation rules for methods

Fig. 25. The list of the generated classes

– The GenerateMethods() helper generates the method signature: visibility, returned type, name and parameters (see Fig. 24).

Fig. 25 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.

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 [29]. 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 [29] 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.

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 **??**.

In a previous work [30] 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 [28] 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 **??**).
- 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 B.5). 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.

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

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 metamodel. for this fact, we defined a model for the mapping properties as shown in Fig. 26.

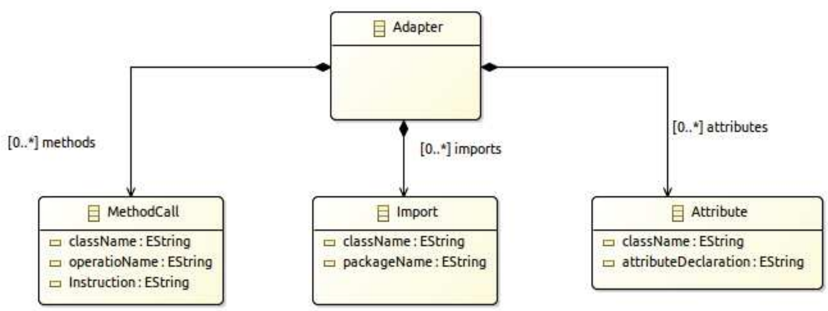

Fig. 26. Adapter Pattern Model

In the example of Fig. 27, the (model) class Motor delegates its method calls to the EV3Large RegulatedMotor.

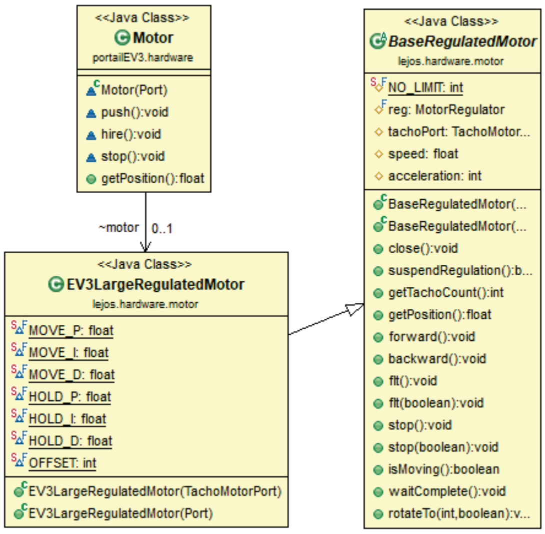

Fig. 27. Class Mapping by Adaptation of the Motor Class

Based on the specification of a simplified *Adapter Pattern* presented in Fig. 26, 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. 28.

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 name of the class and the name of the operation to be mapped. Note that addAdapterAttributes, mappingMethods() and getImports () helpers will run based on the properties file that is defined as an instance of the adapter model. Listing 1.1 presents the content of such a property file in the case of Motor.

Listing 1.1. Instance of Adapter Model

```
<?xml version="1.0" encoding="UTF−8"?>
<Adapter xmi:version="2.0"
    xmlns:xmi="http :// www.omg.org/XMI">
  <methods className="Motor"
```

```
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. 28. ATL helper to generate adapted attributes

```
    operatioName="push"  Instruction ="EV3LargeRegulatedMotor.forward();" />
  <methods className="Motor"
    operatioName="hire"   Instruction ="EV3LargeRegulatedMotor.backward();" />
  <methods className="Motor"
    operatioName="stop"   Instruction ="EV3LargeRegulatedMotor.stop();"  />
  <methods className="ContactSensor"
    operatioName="contact"   Instruction ="EV3TouchSensor.fetchSample();" />
  <methods className="MotionDetector"
    operatioName="contact"   Instruction ="EV3UltrasonicSensor.fetchSample() ;"  />
  < attributes   className="Motor"   attributeDeclaration =" private
      EV3LargeRegulatedMotor ev3LargeRegulatedMotor;" />
  < attributes   className="ContactSensor"
     attributeDeclaration  =" private  EV3TouchSensor ev3TouchSensor;" />
  < attributes   className="MotionDetector"
     attributeDeclaration  =" private  EV3UltrasonicSensor  ev3UltrasonicSensor ;"  />
  < attributes   className="Communication"
     attributeDeclaration  =" private   lejos .remote.nxt  nxt ;"  />
  <imports  className="Motor"
    packageName="lejos.hardware.motor.EV3LargeRegulatedMotor;" />
  <imports  className="ContactSensor"
    packageName="lejos.hardware.sensor . EV3TouchSensor;" />
  <imports  className="MotionDetector"
    packageName="lejos.hardware.sensor . EV3UltrasonicSensor"  />
  <imports  className="Communication"
    packageName="lejos.remote.nxt . BTConnection;" />
</Adapter>
```

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

Listing 1.2. Instance of Adapter Model

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

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

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

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

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

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

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

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.

B.5 Example: Reverse engineering Lejos libraries

In our conducting case study, we use the Lejos[27] framework. To abstract a Lejos PDM, we started from the `ev3classes-src.zip` archive of the EV3 library because the other Android/Java libraries are standard. Experimentations were led with Papyrus, Modisco and AgileJ. Papyrus enabled to reverse engineer[28] individual classes but not packages. In the context of a papyrus project, applying the command `Java>Reverse` on lejosEV3src model elements fails except for classes. Even for a class, the methods

[27] Lejos is a complete Operating System based on an Oracle JVM.
[28] `https://wiki.eclipse.org/Java_reverse_engineering`

were not included. With Modisco [18], 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. With AgileJ [29] reverse the Java code to UML class diagrams is simple. From a visual point of view, we note that it provides many relationships between classes compared to other tools like ObjectAid. 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.

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

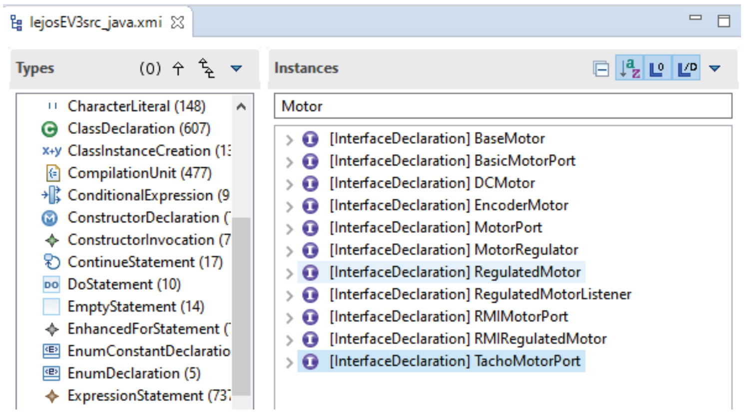

Fig. 29. Modisco discovery for interfaces

AgileJ provides a filter tool (*cf.* Fig. 30) 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.

In the 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*

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

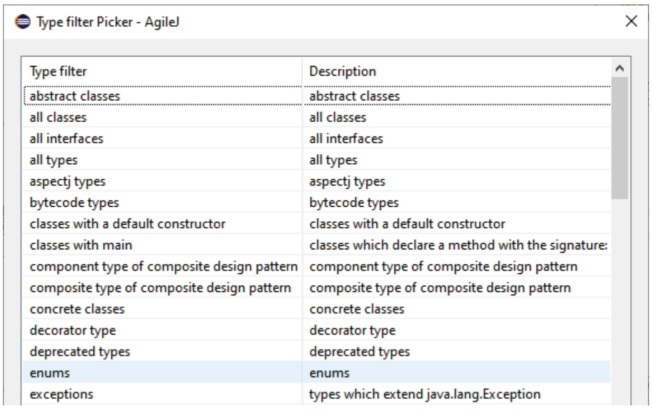

Fig. 30. AgileJ filter process

adapt). Note that AgileJ provides visual and interactive information while Modisco enables customize query and transformation. Further experimentations are required with Papyrus RE which is still on contribution. Other experimentations on MDRE can be found in [17] that show the complexity of the process.

References

1. Selic, B.: Personal reflections on automation, programming culture, and model-based software engineering. Automated Software Engineering **15**(3) (Dec 2008) 379–391
2. Pressman, R.: Software Engineering: A Practitioner's Approach. 7 edn. McGraw-Hill, Inc., New York, NY, USA (2010)
3. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Software Development Process. Object-Oriented Series. Addison-Wesley (1999) ISBN 0-201-57169-2.
4. Selic, B.: From model-driven development to model-driven engineering. In: Proceedings of the 19th Euromicro Conference on Real-Time Systems. ECRTS '07, USA, IEEE Computer Society (2007) 3
5. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice: Second Edition. 2nd edn. Morgan & Claypool Publishers (2017)
6. Mussbacher, G., al.: The relevance of model-driven engineering thirty years from now. In: Model-Driven Engineering Languages and Systems: 17th International Conference, MODELS 2014, Valencia, Spain, September 28 – October 3, 2014. Proceedings, Cham, Springer International Publishing (2014) 183–200
7. 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) (2020) 5–13
8. Di Ruscio, D., Eramo, R., Pierantonio, A.: Model transformations. In Bernardo, M., Cortellessa, V., Pierantonio, A., 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, Berlin, Heidelberg, Springer Berlin Heidelberg (2012) 91–136
9. 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)

10. André, P., Tebib, M.E.A.: Refining automation system control with MDE. In Hammoudi, S., Pires, L.F., Selic, B., eds.: Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2020, Valletta, Malta, February 25-27, 2020, SCITEPRESS (2020) 425–432
11. Essebaa, I., Chantit, S.: Model driven architecture and agile methodologies: Reflexion and discussion of their combination. In: 2018 Federated Conference on Computer Science and Information Systems (FedCSIS). (2018) 939–948
12. Ciccozzi, F., Malavolta, I., Selic, B.: Execution of uml models: a systematic review of research and practice. Software & Systems Modeling **18**(3) (Jun 2019) 2313–2360
13. Pilitowski, R., Dereziñska, A.: Code generation and execution framework for uml 2.0 classes and state machines. In Sobh, T., ed.: Innovations and Advanced Techniques in Computer and Information Sciences and Engineering, Dordrecht, Springer Netherlands (2007) 421–427
14. Mehmood, A., Jawawi, D.N.A.: Aspect-oriented model-driven code generation: A systematic mapping study. Inf. Softw. Technol. **55**(2) (2013) 395–411
15. Clavreul, M.: Model and Metamodel Composition: Separation of Mapping and Interpretation for Unifying Existing Model Composition Techniques. PhD thesis, Université Rennes 1 (December 2011)
16. Raibulet, C., Fontana, F.A., Zanoni, M.: Model-driven reverse engineering approaches: A systematic literature review. IEEE Access **5** (2017) 14516–14542
17. 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. (2019) 256–263
18. Brunelière, H., Cabot, J., Dupé, G., Madiot, F.: Modisco: A model driven reverse engineering framework. Information and Software Technology **56**(8) (2014) 1012 – 1032
19. Pepin, J., André, P., Attiogbé, C., Breton, E.: An improved model facet method to support EA alignment. CSIMQ **9** (2016) 1–27
20. 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 (February 2017)
21. André, P., Ardourel, G.: Domain Based Verification for UML Models. In Kuzniarz, L., Reggio, G., Sourrouille, J.L., Staron, M., eds.: Workshop on Consistency in Model Driven Engineering C@Mode'05. (November 2005) 47–62
22. Karsai, G., Taentzer, G., Mens, T., Gorp, P.V.: A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science **152** (2006) 125 – 142 Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005).
23. 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** (2014) 41 – 99
24. Kahani, N., Bagherzadeh, M., Cordy, J.R., Dingel, J., Varró, D.: Survey and classification of model transformation tools. Software and Systems Modeling **18**(4) (2019) 2361–2397
25. Weilkiens, T.: Systems Engineering with SysML/UML: Modeling, Analysis, Design. The MK/OMG Press. Elsevier Science (2008)
26. André, P., Azzi, F., Cardin, O.: Heterogeneous communication middleware for digital twin based cyber manufacturing systems. In Borangiu, T., Trentesaux, D., Leitão, P., Boggino, A.G., Botti, V.J., eds.: Proceedings of SOHOMA. Volume 853 of Studies in Computational Intelligence., Springer (2019) 146–157
27. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., USA (1995)

28. Pepin, J., André, P., Attiogbé, J.C., Breton, E.: Definition and visualization of virtual meta-model extensions with a facet framework. In Hammoudi, S., Pires, L.F., Selic, B., eds.: 6th Int. Conf. MODELSWARD 2018, Revised Selected Papers. Volume 991 of Communications in Computer and Information Science., Springer (2018) 106–133
29. 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. Volume 1. (2017) 208–219
30. 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. (Oct 2009) 317–318