The Program is the Model: Enabling Transformations@run.time

Jesús Sánchez Cuadrado[1], Esther Guerra[1], and Juan de Lara[1]

Universidad Autónoma de Madrid (Spain)
{Jesus.Sanchez.Cuadrado, Esther.Guerra, Juan.deLara}@uam.es

Abstract. The increasing application of Model Driven Engineering in a wide range of domains, in addition to pure code generation, raises the need to manipulate models at runtime, as part of regular programs. Moreover, certain kind of programming tasks can be seen as model transformation tasks, and thus we could take advantage of model transformation technology in order to facilitate them.
In this paper we report on our works to bridge the gap between regular programming and model transformation by enabling the manipulation of Java APIs as models. Our approach is based on the specification of a mapping between a Java API (e.g., Swing) and a meta-model describing it. A model transformation definition is written against the API meta-model and we have built a compiler that generates the corresponding Java bytecode according to the mapping. We present several application scenarios and discuss the mapping between object-oriented meta-modelling and the Java object system. Our proposal has been validated by a prototype implementation which is also contributed.

Keywords: Model-Driven Engineering, Model Transformations, Transformations at Runtime, APIs, Java Virtual Machine

1 Introduction

Model Driven Engineering (MDE) is becoming a popular software development paradigm to automate development tasks via domain specific languages (DSLs) and code generation. Nowadays, the application of MDE to more advanced scenarios is leading to a trend to use models at runtime as part of running systems [4]. The use of models at runtime requires interacting with functionality written in general purpose languages (GPL), in particular made available in the form of APIs. However, there is a gap between model management frameworks and GPLs that overcomplicate this application scenario. Additionally, we have observed that certain kind of programming tasks can be more naturally expressed using model transformation languages, rather than using GPLs like Java. Hence, we could take advantage of model transformation technology to facilitate them, but the lack of proper integration mechanisms hinders this possibility.

The most common approach to bridge the modeling technical space (also known as modelware) and existing programming APIs is writing ad-hoc pro-

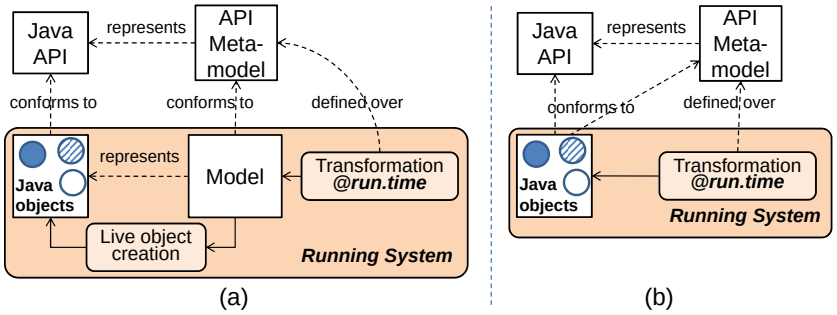


Fig. 1: (a) Transformation at runtime with intermediate model representing the "live" objects. (b) Our approach, where runtime objects are seamlessly seen as models.

grams that map a given model to some API, using the facilities of the underlying meta-modeling framework (e.g., generated classes or reflective interfaces in EMF). Applications of this approach can be found in [5, 17], and some automatic mapping tools have been proposed to facilitate this task [11, 14, 15]. Given a meta-model that represents the API, models are injected from live objects or objects are created from a given model. The main issue with these approaches is that models that imitate the API object structure must be created at runtime as well, together with the machinery to create or read the runtime objects from the models, which is inefficient and adds unnecessary complexity. This approach is depicted in Fig. 1(a).

Instead, in this paper we propose a more direct approach which provides a better integration of model transformations with programs written using GPLs (Java in particular), as depicted in Fig. 1(b). In particular, our approach permits model transformations to use directly Java objects as if they were part of a model, conformant to a meta-model. This takes the benefits of MDE to the program level, realizing some aspects of Bertrand Meyer's *Single Product Principle: "The program is the model. The model is the program"* [13], and enabling a better integration of modeling and programming tasks.

Our approach is based on the specification of a mapping between a Java API (e.g., Swing) and a meta-model describing it. A model transformation definition is written against the API meta-model and then compiled into the corresponding Java bytecode according to the mapping. In this way, the integration of Java programs and model transformation technology is seamless. In this paper, we present several application scenarios and discuss the challenges involved in mapping object-oriented meta-modeling to the Java object system. Our proposal has been validated by a prototype implementation, which is also contributed.

Paper organization. Section 2 gives an overview of our approach. Section 3 presents some background and introduces a running example. Section 4 details our approach to describe APIs by means of meta-models. Section 5 presents more sophisticated mapping constructs to support different API styles. Section 7 eval-

uates the approach on further examples in different scenarios. Section 8 compares with related work and Section 9 ends with the conclusions and future work.

2 Overview

In this section, we identify the requirements and challenges posed by the integration of modelware and object-oriented programming languages, and introduce our approach. We will focus on the integration from the perspective of model manipulations typically performed with model-to-model transformation languages, and using Java as the target GPL. For simplicity, we will consider the manipulation of object-oriented APIs, although the approach can be generalised to any kind of object-oriented program.

As explained in the introduction, several approaches exist to bridge the modelware technical space and GPLs, most notably Java [1, 3, 9, 11]. However, a practical bridge should fulfil in addition the following requirements:

- *Non-intrusive.* It must not require modifying neither existing meta-models nor existing APIs (e.g., using manually written annotations).
- *Seamless integration.* Once transformations are defined, they should be easily and seamlessly invoked from programs, as black-boxes, using regular constructs of the GPL. For instance, it should be possible to invoke a transformation by creating a new transformation object, setting the involved models and calling a *run* method. This aspect includes integration at the IDE level as well. And the other way round: model transformation developers should be able to deal with runtime objects as if they were model elements, described by a meta-model, using the model transformation language normally.
- *Efficient.* The bridge between Java objects and models should be efficient. In the ideal case, it should not require intermediate data structures, so that the model-based manipulations of Java objects become as efficient as if they were written using Java code. Such an intermediate representation would hinder some uses of transformations at runtime, like streaming transformations.
- *API style coverage.* APIs may provide access to objects in different ways, being the use of *getter* and *setter* methods the simplest one. A practical bridge should consider the most common access mechanism to cover a wider range of APIs.

Although some researchers have proposed solutions to bridge models and Java objects (cf. Section 8), to the best of our knowledge, no existing tool satisfies all the previous requirements.

2.1 Architecture

The elements of our approach are depicted in Figure 2. In general, a transformation definition, written in some transformation language, manipulates models that conform to some meta-models. We extend this pattern to allow a transformation definition to manipulate objects of a given API as if they were model

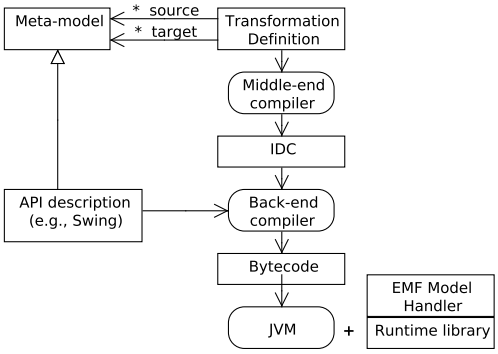


Fig. 2: Elements of our approach.

elements. The underlying idea is to specify an API description model that, from the perspective of the transformation developer, acts as a meta-model for the API. This model establishes a mapping between the API and a set of meta-model elements that are used to write a transformation definition against them.

In our solution, the transformation definition is compiled to an intermediate language (called IDC) that provides primitive instructions for model manipulation (see next section for more details). This compilation step is performed without taking into account whether the transformation will deal with an API or a regular model. Then, the IDC intermediate representation is compiled to the Java Virtual Machine (JVM) bytecode format. At this step, the API description is used by the back-end compiler to generate bytecode to access Java objects directly (e.g., using method calls). Please note that, despite compiling a transformation definition to the JVM, our tool relies on the underlying metamodeling framework when regular (meta-)models are used, with indirect access to model elements via a model handler (e.g., EMF Model Handler in the figure).

3 Background and Running Example

In this section we provide a running example that will be used throughout the paper. However, we first outline the technical context of our work, that is, the Eclectic transformation tool [7], the IDC intermediate language and the Java Virtual Machine (JVM).

Eclectic is a transformation tool based on the idea of a family of model transformation languages. Instead of having a large language with many constructs, we provide several small languages each one of them focussed on a specific kind of transformation task. By now, the family is made of: (i) a rule-based mapping language – in the style of the declarative part of ATL [12] – to specify correspondences between source and target model elements, (ii) a pattern language,

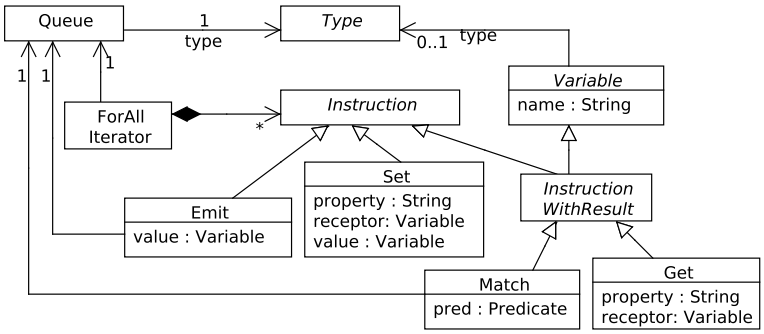


Fig. 3: Excerpt of the IDC meta-model.

(iii) a language for attribute computation, inspired by attribute grammars, (iv) a template-based, target-oriented language and (v) a language to orchestrate the execution of the different transformation tasks. Each language compiles down to the IDC intermediate language, which provides the composition and interoperability mechanisms. The translation from API descriptions to JVM bytecode is performed at the IDC level, so that every language of Eclectic can take advantage of the bridge. For the sake of simplicity, in this paper we will just use one language of Eclectic: the mapping language.

The Intermediate Dependency Code (IDC) is a simple, low-level language composed of a few instructions, some of them specialized for model manipulation. Figure 3 shows an excerpt of its meta-model. Every instruction inherits from the Instruction abstract metaclass. Since most instructions produce a result, they also inherit from Variable (via InstructionWithResult) so that the produced result can be referenced as a variable. Indeed, we use a simplified form of Static Single Assignment (SSA) to represent data dependencies between instructions [8], since every generated value is stored into a uniquely identified variable.

The IDC language provides instructions to create closures, invoke methods, create model elements and set and get properties (Set and Get in Figure 3), among others. In IDC, there is no notion of rule, but the language provides a more general mechanism based on queues. A Queue holds objects of some type, typically source model elements and trace links. The ForAllIterator receives notifications of new elements in a queue, and executes the corresponding instructions. There are two special instructions to deal with queues: Emit puts a new object into a queue, while Match retrieves an element of a queue that satisfies a given predicate. If such an element is not readily available, the execution of this piece of code is suspended into a *continuation* [6] until another part of the transformation provides the required value via an Emit.

To some extent, IDC can be considered as an event-based approach to model transformation. If the transformation is executed in batch mode (i.e., all source

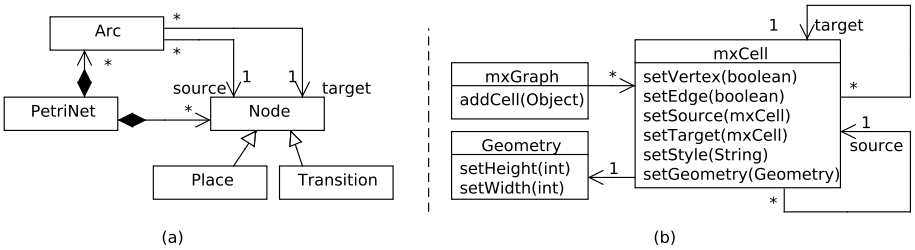


Fig. 4: (a) Petri nets meta-model. (b) A small excerpt of the jGraph API.

model elements are readily available) then the transformation queues are just filled at the beginning and the transformation proceeds normally.

IDC transformations are compiled to the JVM. The JVM is a stack-based virtual machine based on instructions called bytecodes. Interestingly, the JVM instruction set is statically typed, since it requires detailed type information to perform operations over objects. For instance, calling a method requires specifiying the name of the class or interface where the method was defined, the types of the parameters and the return type. We have taken this characteristic into account in the design and implementation of the bridge between meta-models and Java classes.

3.1 Running Example

As a motivating example, let us suppose we are using Petri net models conforming to the meta-model in Figure 4(a) as part of an MDE process, and we are interested in visualizing such models for debugging purposes. A possible solution is to use an API like jGraph[1] which allows visualizing graph-like structures and provides automatic layout capabilities. However, if we address this task using plain Java and e.g., EMF, this would require writing an interpreter that imperatively traverses the model, keeps track of the cycles and instantiate the jGraph classes.

Instead, with our approach, we build a simple transformation that maps Petri net concepts (places, transitions and arcs) to jGraph API concepts. In particular, the excerpt of the jGraph API used in this transformation is shown in Figure 4. The mxCell class represents a visualizable element. Its setVertex and setEdge methods identify whether the cell acts as a vertex or as an edge. If it is an edge, the setSource and setTarget methods can be used to establish connections to other elements. A cell has an associated Geometry that establishes its size. This value is compulsory and can be set in the constructor or with the setGeometry method. For simplicity, we will not consider it until Section 4.3.

[1] http://jgraph.com

```
1  mapping petrinet2jgraph (in) -> (out)
2  in : 'platform:/resource/example/petrinet.ecore'
3  out : 'platform:/resource/example/jgraph.apidesc'
4
5  from p : in!PetriNet to g : out!Graph
6    g.cells <- p.nodes
7    g.cells <- p.arcs
8  end
9
10 from place : in!Place to cell : out!Cell
11   cell.vertex = true
12   cell.value = place.name
13 end
14
15 from t : in!Transition to cell : out!Cell
16   cell.vertex = true
17   cell.value = t.name
18 end
19
20 from arc : in!Arc to cell : out!Cell
21   cell.edge = true
22   cell.source <- arc.source
23   cell.target <- arc.target
24 end
```

Fig. 5: Transformation from Petri nets to jGraph with the Eclectic mapping language.

Figure 5 shows the corresponding transformation, using the Eclectic mapping language. Rules establish a mapping between a source type and a target type. Line 1 declares the transformation name and parameters (input and output models), whose conforming meta-model is given in lines 2 and 3. Please note that for `out` we do not use a regular meta-model, but an API description that acts as a meta-model. Then, four transformation rules are defined translating each element of the Petri net into JGraph. The = operator assigns an attribute value and the ← operator resolves a reference.

This program declaratively specifies the transformation between two data structures: the Petri net model and the jGraph's representation of visualizable graphs. The transformation is defined as if there were models in the source and target domains; however, the transformation actually produces Java objects in the target. This is done by including an API description model in place of the target meta-model. The next section explains how to describe an API.

4 Mapping Meta-models to the Java Object Model

APIs in object-oriented languages are typically formed by a set of classes that represent the elements of some domain (e.g., widgets in a GUI toolkit). We propose to establish a mapping between API classes and a meta-model that describes the structure of the API. Several descriptions may be available for a given API, perhaps focussing on a different aspect. Describing an API as a meta-model permits the use of model transformation tools to manipulate the object graph of the API. An additional advantage is that the meta-model provides a compact description of the API that simplifies the access to its structure, since it does not contain behaviour methods (i.e., in contrast to get/set methods, which are related to the structure).

In this way, a key point of our approach is the mapping between meta-modeling concepts and object-oriented API concepts. To specify this mapping, we have built a meta-model called *API description*. API description models will drive the bytecode generation phase of our compiler.

The rest of the section explains how APIs are described in our approach. First of all, we focus on the basic mappings that correspond to basic object-oriented

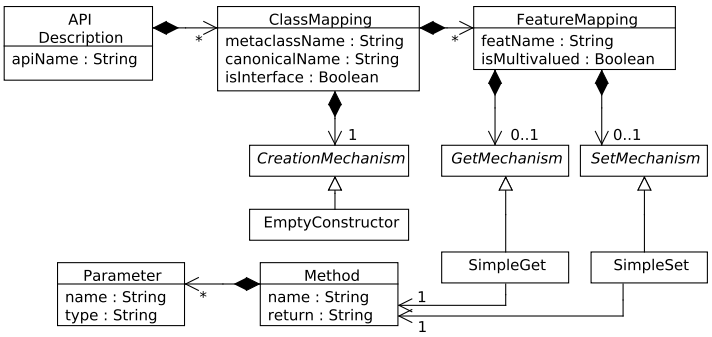


Fig. 6: Core elements of the API description meta-model.

constructs. Next, we introduce a DSL to specify these mappings. Finally, we deal with the mapping of constructors.

4.1 Basic Mapping

We have designed the mapping between a meta-model and an API from the perspective of the primitive operations over model elements that our model transformation engine requires. There are three kinds of operations: create elements, read features and write features (i.e., accessing features).

The simplest mapping is a one-to-one correspondence between meta-classes and API classes, and structural features (attributes for primitive types and references for classes) with getter and setter methods. The mapping for structural features must take into account whether the feature is multivalued or not. Following the Java language conventions, a mapping for a mono-valued feature is a pair of accessor methods FeatureType get⟨⟨featureName⟩⟩() and void set⟨⟨featureName⟩⟩(FeatureType value). In the case of multi-valued features, the pair of accessor methods is Collection<FeatureType> get⟨⟨featureName⟩⟩() and void add⟨⟨featureName⟩⟩(FeatureType value).

However, an API may overlook these conventions, providing different access mechanisms. In [11], a fixed number of simple mappings is proposed. In our case, we do not restrict our API meta-model to some predefined mappings, but it has been designed with extensibility in mind so that new mappings can be added as they are discovered. Figure 6 shows the core meta-model.

An APIDescription is composed of ClassMapping elements, mapping a metaclass to a Java class (canonical name in the meta-model). Each feature of the metaclass must be mapped via a FeatureMapping. The mechanisms to access properties are abstracted by the GetMechanism and SetMechanism classes, which can be specialized with concrete mechanisms. One of such is the one-to-one correspondence explained above (through SimpleGet and SimpleSet).

The meta-model also includes detailed information about parameters and return types. This is needed in our case because the JVM specification requires

the compiler to generate bytecode with this information. In any case, this information can be gathered via reflection to alleviate the user from this burden.

4.2 A DSL to Specify Mappings

We have built a textual DSL to facilitate the task of specifying mappings. The DSL allows specifying the meta-model that describes the API at the same time as the mapping to the API. There are four basic constructs:

1. metaclass maps a metaclass to the corresponding Java class or interface. The metaclass is automatically marked as abstract if it is mapped to an interface or a Java class.
2. ref establishes a mapping for a reference (i.e., a feature whose type is a metaclass). Moreover, an access mechanism has to be specified. The simplest one is to associate a getter and a setter method. The * modifier indicates that the meta-model feature is multivalued. In the Java side, Array<JavaType> and CollectionType<JavaType> indicate that a method takes or returns several elements.
3. attr establishes a mapping for an attribute. It is similar to ref, but it deals with primitive types. We support the Java primitive types, and also java.lang.String as a primitive type. Automatic conversions between the meta-model type and the actual Java type are also supported (e.g., byte is converted to int).
4. constructor indicates how to create a new object. There is a straightforward mapping to the empty constructor if it is available. In the next subsection, we elaborate on how to establish mappings to non-empty constructors.

Figure 7 shows the mapping definition for the running example. As it can be seen, the Graph metaclass is mapped to the mxGraph Java class. The cells multivalued reference is mapped to the addCell method so that the generated transformation code will add cells one-by-one. This method takes a java.lang.Object as parameter, but from the transformation point of view, the object will always be a mxCell object (see next mapping from Cell to mxCell) . Finally, the feature is write-only because jGraph's API does not offer any getter method for cells. In Section 5 we will show an extension to overcome this limitation.

4.3 Mapping Constructors

In meta-modelling, metaclasses do not have constructors to ensure proper initialisation of objects, but the multiplicities assigned to features act as initialisation constraints. On the other hand, Java classes require at least one constructor, which does not necessarily need to be an empty constructor. In a model transformation, a model element (or a set of target elements that form a target pattern) is first created by a rule, and then initialised assigning values to features. Hence, there is a mismatch between model element instantiation and constructor-based instantiation of Java classes when an empty constructor has not been defined.

We have devised two extensions of the API description model to deal with this problem. The first extension permits associating literal values to the parameters

```
1
2  metaclass Graph to com.mxgraph.view.mxGraph {
3    empty constructor
4    ref cells* : Cell
5         set method addCell(java.lang.Object) : java.lang.Object
6         // no get method is defined -> readonly property
7  }
8
9  metaclass Cell to com.mxgraph.model.mxCell {
10   empty constructor
11   attr edge : Boolean
12        get method getEdge() : boolean
13        set method setEdge(boolean) : void
14
15   attr vertex : Boolean
16        get method getVertex() : boolean
17        set method setVertex(boolean) : void
18
19   ref source : Cell
20        set method setSource(com.mxgraph.model.mxCell) : void
21
22   ref target : Cell
23        set method setTarget(com.mxgraph.model.mxCell) : void
24 }
```

Fig. 7: API description for jGraph.

of a constructor. When the class is created, our engine uses the literal values as constructor parameters. This approach only works with constructors whose parameters can be primitive types or *null*.

The second extension provides greater flexibility by delaying instantiation until all parameter values are available. The underlying idea is to associate a constructor parameter to a feature of the meta-model, so that the corresponding object is not created until the value of all parameters are given by means of feature assignments (set instructions in IDC). Then, instead of setting the feature, the value is used as part of the constructor.

This process is transparent both to the transformation developer and to the transformation language developer, because the reordering of the instructions is made at the IDC level. As an example, even though jGraph's mxCell has an empty constructor, a better practice is to use another constructor that takes the cell value and a Geometry object as parameters. This ensures that cells are valid by construction. Figure 8 illustrates the rewriting process. The left-hand side shows the rule to transform Places extended to consider that a Geometry object has to be created as well (the linking construct establishes a link between both target elements). Below, the mapping from Cell to mxCell now includes a constructor statement that specifies which properties the constructor depends on.

Figure 8(b) shows the resulting IDC code. First, the target elements are created (instructions 1 and 2), and then, a new trace link is created and emitted to the trace (3-6). Afterwards, the features are set (we use intermediate variables v, lit1, lit2). It is important to note that feature assignments in the original transformation are translated to IDC Set instructions (9, 12, 13), despite the fact that they are actually constructor parameters.

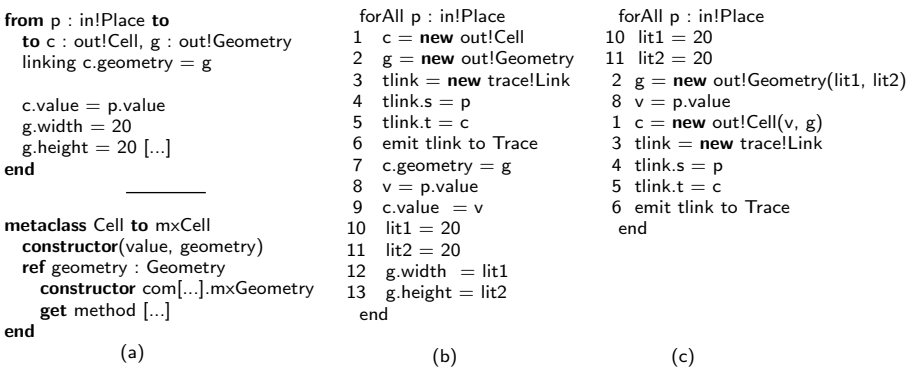


Fig. 8: Translation and rewriting for constructors. (a) Transformation rule and API description. (b) Standard translation to IDC (in pseudocode). (c) Rewritten version that meets constructor dependencies.

Figure 8(c) shows the result of the rewriting. The Set instructions that correspond to constructor parameters are removed, but the assigned values will be used as constructor parameters. To this end, every instruction that is directly or indirectly part of the computation of the value is moved before the creation instruction. If two or more target elements have mutually recursive constructors, it will result in a compiler error. Besides, it is worth noting that, at runtime, computing a value required by a constructor may require another transformation rule to produce it, so the scheduling of the transformation may become affected. In our case, we have the Match instruction to retrieve values from the transformation trace model. This instruction is able to stop the computation of a rule if the value is not available, resuming the rule when another rule provides such a value. This mechanism allows performing this rewriting safely.

An important property of this strategy is that it is non-intrusive, in the sense that we do not need to change the transformation language adding constructors.

5 Extending the Mapping

The presented mapping considers the basic elements needed to manipulate Java APIs with a model transformation language. However, it is limited to APIs that expose their structure via accessor methods. In this section, we present some extensions that we have added in order to cover a wider range of APIs. First, we will present an extension mechanism enabling flexible user-defined mappings. Then, we will present some extensions based on design patterns.

5.1 User-defined Mappings

A simple extension is to consider mappings expressed using Java code. This permits specifying feature mappings that require accessing the API using some mechanism that is not supported by the API description language.

We have extended our meta-model and the DSL to consider get and set mechanisms implemented using Java code. In this extension, method calls are not performed over the original receptor object, but with an additional level of indirection. A "mapper class" implements methods that are mapped to the meta-model features, in charge of getting or setting values for a given API object.

Listing 1 shows a modified version of the API specification for the running example, which enables the retrieval of the cells of a graph. First, the mapper keyword selects the class implementing the methods that will perform the mapping (there is one mapper class per API description). Then, a syntax similar to the one for the getter methods is used, but replacing method by mapper. Listing 2 shows the piece of Java code that implements the mapping for the cells feature in the getCells method. A mapper class must implement IUserDefinedMapping and provide the setContext method. The latter is an initialization method to establish the transformation runtime information in case it is needed (e.g., access the source model to lookup some object).

```
api jgraph described by "http://jgraph/api"

mapper class example.JGraphMapping

metaclass Graph to com.mxgraph.view.mxGraph {
  empty constructor

  ref cells* : Cell
    get mapper getCells(com.mxgraph.view.mxGraph) :
      Array<com.mxgraph.model.mxCell>
    set method addCell(java.lang.Object) :
      java.lang.Object
}
```

Listing 1: User-defined mapping for "cells".

```
public class JGraphMapping implements
      IUserDefinedMapping {
  public mxCell[] getCells(mxGraph receptor) {
    mxCell root= ((mxCell) graph.getDefaultParent());
    mxCell[] cells= new mxCell[root.getChildCount()];
    for(int i = 0; i < root.getChildCount(); i++) {
      cells[i] = (mxCell) root.getChildAt(i);
    }
    return cells;
  }

  public void setContext(Context contex) { ... }
}
```

Listing 2: Mapper class for jGraph.

5.2 Mappings Related to Design Patterns

In practice, object-oriented programs use a variety of techniques to provide greater flexibility to some aspects regarding construction of objects, structure or behaviour. Some of these techniques have been documented as design patterns [10]. We have studied which design patterns are relevant for our mapping, so that support for them can be provided extending the core meta-model.

We have identified six relevant patterns for our case: Abstract Factory, Singleton, Composite, Facade, Observer and Iterator. Our approach is to provide a description of how the pattern is instantiated in a given API, so that our compiler is able to generate the access code according to the description. Figure 9 shows the extension of the core mapping meta-model to describe the *Iterator* and *Observer* patterns. Currently, we have just given support to these two patterns, thus in the rest of the section we will focus on them. In any case, we expect that supporting the other four patterns will involve similar elements.

Iterator. An aggregated object provides access to its contents via another object that holds the iteration state (which indeed is often kept using the Memento

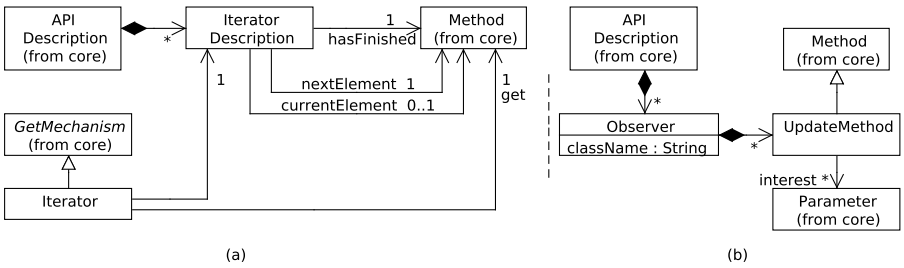


Fig. 9: Extensions to support design patterns. (a) Iterator. (b) Observer.

pattern). In our case, a reference may need to be filled from the elements of an iterator. Figure 9(a) shows how this pattern is represented in our meta-model. One or more IteratorDescription elements are declared, specifying the methods used to perform the iteration. If no currentElement is given, a Java-like iterator style is assumed (i.e., the nextElement method increments the iterator and returns the next element). Given this specification, the access to a multivalued feature can be mapped to an iterator. Our compiler generates the code to perform the iteration and retrieve the corresponding elements.

Observer. This pattern allows one or more subscribers to receive events of some type from a publisher. A transformation can be a subscriber (observer) so that a rule is triggered if its source pattern matches an incoming event. A transformation can behave as a publisher (observable) if it emits an event of some type each time a rule creates a new target element.

If a transformation uses an API that requires an observer, the transformation class automatically implements the required interfaces. The user of the transformation just needs to register the transformation in the corresponding observable. Figure 9(b) shows the extension to deal with observers. One or more Observer classes can be associated to an API. Each one of them contains a set of UpdateMethod elements that receive event objects as parameters. The event objects have to be described with the API description language as well, so that transformation rules can use them. The interest reference indicates which parameters of the update method must be passed to the transformation engine, discarding the rest. As explained in Section 3, at the IDC level, we use queues to feed the transformation engine. Thus, it is straightforward to fill IDC queues with the event objects received in the update methods. Effectively, by using this pattern we enable a form of streaming transformations.

6 Implementation and Integration

This section outlines some details of our implementation and the tool support.

6.1 Integration with Java Code

One distinctive aspect of our approach is that it seamlessly integrates with existing Java code. When a transformation definition is compiled, a class that acts as a front-end to configure and invoke the transformation is created. In this way, from the developer perspective, a transformation definition is just a Java class that performs a complex computation, taking some data structure and returning another one. For example, executing the running example will only involve writing a piece of code similar to the one shown in Listing 3. In constrast, other model transformation tools (e.g., ATL or ETL) provide dedicated launching mechanisms that are mainly intended to deal with e.g., EMF models.

```
1  public class Test {
2    public static void main(String[] args) {
3      petrinet2jgraph t = new petrinet2jgraph();
4      EMFLoader loader = new EMFLoader();
5      t.setInModel(loader.load("PetriNet.ecore", "model.xmi"));
6      t.execute();
7
8      mxGraph g = t.getOutModel().getRoot(mxGraph.class);
9      // some code to launch the visualization
10   }
11 }
```

Listing 3: Invoking a transformation compiled with Eclectic.

6.2 Implementation and Tool Support

We have implemented the architecture shown in Figure 2. Transformation definitions are written with Eclectic, whose concrete syntax has been implemented using XText. The Eclectic compiler has been bootstrapped, so that internally it has the same architecture. Notably, we have used a subset of Eclectic to implement the middle-end compiler of Eclectic. The back-end compiler is written in Java, using the BCEL library to generate bytecode. This step of the compilation deals with the translation of the IDC instructions, using the API description to generate the bytecode that access directly the Java objects.

Regarding tool support, we have built an Eclipse plugin that includes editors for the Eclectic languages and integrates the Eclectic compiler so that transformations are automatically re-compiled after a change. The *.class* files generated by the compilation process are added to the project classpath, so that they can readily be used in the Eclipse workspace. Finally, the binaries and the source code of our tool have been publicly released[2].

7 Case Studies and Assessment

We have evaluated our approach by implementing some case studies that exercise different kinds of APIs. Additionally, we have identified three main application scenarios for our approach, and we organise the case studies according to them.

[2] Eclectic web site: `http://sanchezcuadrado.es/projects/eclectic`

7.1 Application Scenarios

Model to Java. The first scenario consists of transforming a regular model, conforming to some meta-model, to Java objects that will be integrated with a running system. The running example of Section 3 belongs to this scenario. We have also implemented a transformation from UML class diagrams to Swing graphical user interfaces. This enable the dynamic generation of basic dialogs for a desktop application, in the style of Wazaabi [17].

Java to model. The second scenario is the reverse situation. A set of Java live objects are processed by the transformation engine to yield a given model. We have identified two situations where this scenario could be particularly useful. The first one is reverse engineering at runtime, where the structure of a system at a given moment is analysed by means of a model transformation in order to discover certain information. This has the advantage that the source code of the application is not needed. We have implemented a case study where a transformation is used to reverse engineer a Swing GUI at runtime. A UML class model that represents the underlying design of the GUI is obtained, so that it can be compared against Swing best practices.

The second one is the possibility of taking advantage of existing APIs that perform some complex processing and generate an in-memory data structure that has to be further manipulated. Some examples of APIs of this kind are Selenium[3], which parses HTML pages and generates a DOM tree and Apache POI[4], which provides a Java bridge to read and write Microsoft Office documents. An important advantage of our approach is that there is no need to create a dedicated injector (i.e., a program that reads an artifact in the source technology and generates a model that can be manipulated using MDE technology). Instead, an existing API is directly used.

We have implemented two case studies of this kind. First, we have built a simplified version of the *Jar2Uml* case study[5]. In this application, the BCEL (Byte Code Engineering Library) API[6] is used to read the structure of a Jar file, which is then transformed into an UML class diagram.

The second case study, involves using the *Twitter4J* API[7] to read a stream of *tweets* from Twitter. A model-to-model transformation is in charge of discovering a graph of relationships between users emiting these tweets (and mentioned in them) and hashtags (i.e., keywords). Figure 10 shows (a) the Twitter4J API and (b) a simple meta-model to represent some relationships that arise in Twitter. The description of this API includes the Observer pattern to notify about events such as new tweets produced by users. Figure 10(c) shows the declaration of the TweetListener and the update method onStatus. The [0] modifier indicates that we are interested in the first parameter (as in general the *update* method

[3] http://seleniumhq.org/
[4] http://poi.apache.org/
[5] http://ssel.vub.ac.be/ssel/research/mdd/jar2uml
[6] http://jakarta.apache.org/bcel/
[7] http://twitter4j.org

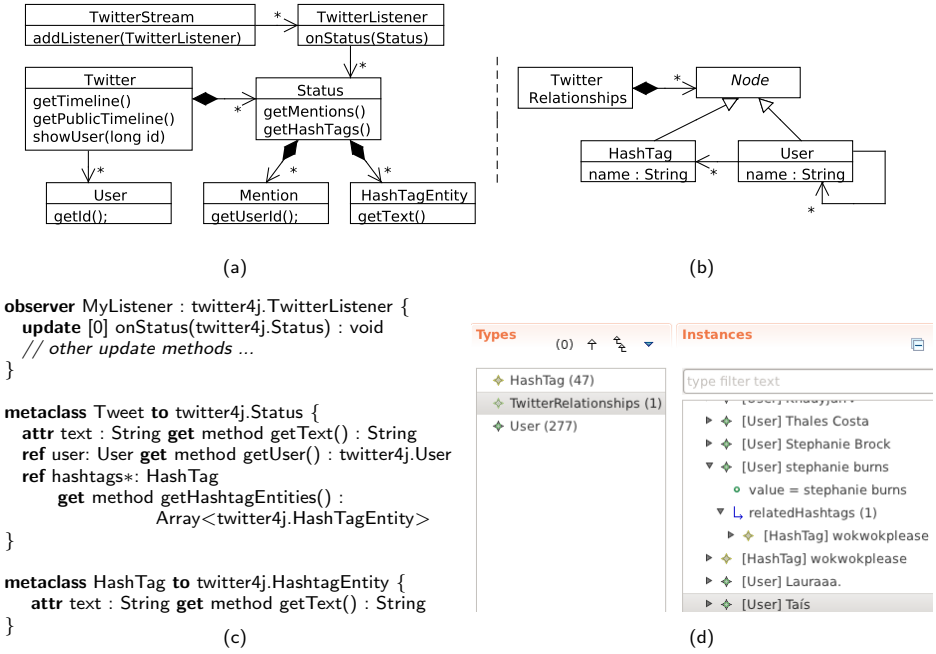


Fig. 10: (a) Twitter4J API. (b) Meta-model for relationships. (c) Excerpt of the JTwitter4J API description. (d) Resulting model.

could include more than one parameter). Our compiler automatically implements the update method and connects the received event to the rules "waiting" for values of this type. It is worth noting that this is a *streaming transformation*, which produces a target model, which gets continously updated as new events arrive. This is possible because IDC features a scheduling mechanism based on continuations that allows us to stop a transformation rule until the required data is available, in this case associated to another event (a new tweet).

Pure Java transformations. In our experience there are several programming tasks that could be seen as a model transformation task. In many cases one needs to establish a mapping between semantically equivalent data structures. For instance, the Transfer Object J2EE pattern advocates creating POJOs (Plain Old Java Object) to transfer information between application tiers. The mappings between Business Objects and POJOs can be seen as a model transformation.

We have implemented two simple case studies of this scenario as a proof of concept. In the first one, the Java reflective API is used to access the information of classes of a given API, and we generate a PDF file (using the iText API[8]) that summarizes the methods and properties of each class. We plan to use a similar

[8] `http://itext.com`

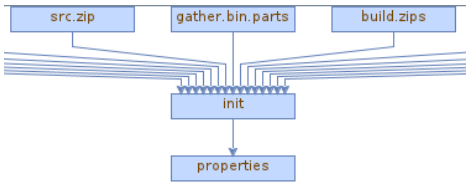
transformation to automatically generate the initial skeleton of the mappings model, in order to facilitate writing API descriptions.

In the second case study, we have implemented a transformation from ANT files to JGraph in order to visualize dependencies among build targets. Figure 11(a) shows an excerpt of the API description. The Iterator pattern is used in the ANT API to give access to the dependencies of a given target. We have described the usage of this pattern in the API as explained in Section 5, by describing the iterator class and establishing which methods are used to perform the iteration. In this case, the API uses a java.util.Enumeration, and hence the model does not include a method to retrieve the current element, but only the method to retrieve the next one. Figure 11(b) shows the visualization obtained after executing the transformation for an ANT build file generated by Eclipse.

```
iterator Enumeration : java.util.Enumeration {
  finished hasMoreElements() : boolean
  next nextElement() : java.lang.Object
}

metaclass Target to org.apache.tools.ant.Target {
 attr name : String get method getName() : String
  ref dependencies* : Target
    get iterator Enumeration method
        getDependencies() : java.util.Enumeration
}
```

(a)



(b)

Fig. 11: (a) ANT API description (excerpt). (b) Dependencies in an Eclipse build file.

7.2 Assessment

The implementation of the case studies has shown that our approach provides a practical mechanism to integrate modelware and object-oriented programs.

The first scenario we tackle (*model to Java*) has been traditionally addressed by creating an ad-hoc processor that traverses a source model programatically an instantiates the API objects. Our approach simplifies this task in cases where there is a mapping between the source model and the API, since we can benefit from model-to-model transformation technology that is specialized for this task.

In the second scenario (*Java to model*), our approach permits leveraging existing APIs to facilitate obtaining an in-memory representation of complex artefacts, while model transformation technology is used to perform the analysis of such artefacts. Using the Observer pattern, we have shown that it is possible to apply this approach to construct *streaming transformations*, which listen to a stream of events and continuously update the target model.

The case studies for the third scenario (*pure Java transformations*) show that it is possible to apply model transformation technology to certain kinds of programming tasks that have a transformation nature, which otherwise would

typically require writing a certain amount of boilerplate code. However, a model transformation language is a specialized language and therefore the programmer only need to focus on the transformation task at hand, abstracting from accidental complexity. Hence, the development is facilitated and readability is improved. On the other hand, applying this approach may require from developers to learn a new language, and the cost-benefit of this trade-off has not been assessed yet.

Finally, we found our mappings DSL expresive enough in most cases. We added the user-defined mappings as a fallback, but we needed to use it in the case studies only three times. Our aim is to extend the DSL as we gain more insight about which idioms are used most frequently in APIs, in particular supporting the four design patterns mentioned in Section 5.

8 Related Work

Even though model-based development is increasingly used in software projects, there is scarce integration between model-based technologies and object-oriented programming. Next we review the few proposals we are aware of.

Api2MoL [11] is a tool to automate the process of bridging models and APIs. Like our approach, it provides a DSL to describe the mapping between a meta-model and an API. However, this DSL is limited to a small fixed number of basic mappings. It is implemented as an interpreter that acts as injector (to create a model from API objects) or extractor (to recreate the API objects from the model). Our API description model is more expresive than that of Api2MoL. In fact, it would be straightforward to implement Api2MoL with our tooling.

The Sm@rt project [14] aims at model-based runtime system management. A meta-model mirroring a system management API is created, and each meta-model element is associated a template defining the Java code that is in charge of performing the mapping. Then, a synchronization engine is automatically generated from this template-based specification. The Sm@rt project is specially tailored for management APIs, although other kinds of APIs can be supported by providing the Java code to perform the mapping.

CHART [9] is a graph transformation language that manipulates Java objects. It requires manual annotation of Java classes and methods to give them a graph-like structure. Besides, it enforces a particular style of the object-oriented programs since graph edges have to be represented with a Java interface.

Tom [3] is a term-rewriting language that has been piggybacked into Java. It uses algebraic terms as the underlying data structure, but is able to transform any data structure as long as a *formal anchor* is provided. Our API description model is indeed a formal anchor. Tom allows mapping algebraic terms to Java classes generated by EMF, but this requires providing some boilerplate code in the rewriting specification to take into account that we are dealing with a graph.

Table 1 summarizes the requirements that a practical bridge between models and objects should fulfil (cf. Section 2), and how they are handled (or not) by the aforementioned approaches. First, the approaches differ in the domain of application (synchronization engines, graph transformation, term rewriting or model

transformation). Only Api2MoL, Sm@rt and Eclectic are non-intrusive. An important requirement is a seamless integration of the used transformation and programming languages, which is only fully achieved by Eclectic (CHART and Tom require generating the transformation as textual Java classes). At runtime, Api2MoL and Sm@rt use an intermediate model, which may affect the efficiency in certain scenarios. On the contrary, the other approaches use Java objects directly, with the constraint that Tom only supports tree-like structures. Finally, Tom and Eclectic provide flexible mechanisms to access objects, including also getter and setter methods.

	Domain	**Non-intrusive**	**IDE Integration**	**Runtime**	**API style**
Api2MoL	SE	Yes	No	Intermediate model	Getter/Setter
Sm@rt	SE	Yes	No	Intermediate model	Management
CHART	GT	No	Java text	Java objects	Getter/Setter
Tom	TR	Only for trees	Java text	Java objects (trees)	Flexible
Eclectic	MT	Yes	Bytecode	Java objects	Flexible

Table 1: Comparison of approaches. SE : Synchronization Engines, GT : Graph Transformation (in-place), TR : Term Rewriting (for trees) and MT : Model Transformation.

SiTra [1] is a simple approach to write model transformations in Java. It provides a Java interface that all implemented rules has to follow, and a *Transformer* class which executes the defined rules. Hence, this approach provides only a very light support for model transformations, which have to be encoded in Java, and there is no support to handle EMF models.

Other approaches based on Virtual Machines include the EMF Transformation Virtual Machine (EMFTVM) [16], or ATL [12], which provides dedicated Virtual Machines for model transformations. Our approach has the advantage of facilitating the integration of model transformation languages and GPLs based on the JVM. Also, the scheduling mechanism based on continuations of IDC is more flexible allowing e.g. streaming transformations.

Regarding API description languages, the approaches to discover API metamodels proposed in [11] and [15], as well the Framework Specific Modeling Languages (FSMLs) proposed in [2] are complementary to our work.

9 Conclusions and Future Work

In this paper, we have presented an approach for the seamless integration of model transformation and general purpose programming languages, like Java. The approach enables the manipulation of Java objects as if they were modeling elements, in a transparent way. For this purpose, we use a mapping model, which describes both the meta-model against which the transformation is defined, and the mapping to the API to be used. The approach enables the use of transformations at runtime, and its seamless integration in programming projects.

In the future, we plan to extend Eclectic with further specialized languages. We also plan to provide a more complete support for streaming transformations

and to provide means to model API protocols (method dependencies), which induce a certain scheduling of transformation rules. With respect to efficiency, we aim at improving the implementation to deal with some cases where we cannot yet generate direct JVM calls, but reflection needs to be used.

References

1. D. H. Akehurst, B. Bordbar, M. J. Evans, W. G. J. Howells, and K. D. McDonald-Maier. SiTra: Simple Transformations in Java. In *MoDELS'06*, volume 4199 of *LNCS*, pages 351–364. Springer, 2006.
2. M. Antkiewicz, K. Czarnecki, and M. Stephan. Engineering of framework-specific modeling languages. *IEEE Trans. Softw. Eng.*, 35(6):795–824, Nov. 2009.
3. J.-C. Bach, X. Crégut, P.-E. Moreau, and M. Pantel. Model Transformations with Tom. In *LDTA'12*, 2012.
4. G. Blair, N. Bencomo, and R. B. France. Models@ run.time. *IEEE Computer*, 42:22–27, 2009.
5. H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot. Modisco: a generic and extensible framework for model driven reverse engineering. In *ASE'10*, pages 173–174, New York, NY, USA, 2010. ACM.
6. W. D. Clinger, A. Hartheimer, and E. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, 1999.
7. J. S. Cuadrado. Towards a family of model transformation languages. In *ICMT*, volume 7307 of *LNCS*, pages 176–191. Springer, 2012.
8. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, October 1991.
9. M. de Mol, A. Rensink, and J. J. Hunt. Graph Transforming Java Data. In *FASE'12*, volume 7212 of *LNCS*, pages 209–223. Springer, 2012.
10. E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.
11. J. L. C. Izquierdo, F. Jouault, J. Cabot, and J. G. Molina. Api2mol: Automating the building of bridges between apis and model-driven engineering. *Information & Software Technology*, 54(3):257–273, 2012.
12. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008.
13. B. Meyer. The Triumph of Objects (Invited Talk). In *TOOLS'12*, 2012.
14. H. Song, G. Huang, F. Chauvel, Y. Xiong, Z. Hu, Y. Sun, and H. Mei. Supporting runtime software architecture: A bidirectional-transformation-based approach. *Journal of Systems and Software*, 84(5):711–723, 2011.
15. H. Song, G. Huang, Y. Xiong, F. Chauvel, Y. Sun, and H. Mei. Inferring meta-models for runtime system data from the clients of management apis. In *MoDELS (2)*, volume 6395 of *LNCS*, pages 168–182. Springer, 2010.
16. D. Wagelaar, M. Tisi, J. Cabot, and F. Jouault. Towards a general composition semantics for rule-based model transformation. In *MoDELS'11*, volume 6981 of *LNCS*, pages 623–637. Springer, 2011.
17. Wazaabi: An open source EMF based dynamic declarative UI framework. `http://wazaabi.org/`.