API2MoL: Automating the building of bridges between APIs and Model-Driven Engineering

Javier Luis Cánovas Izquierdo[1,2,*], Frédéric Jouault[2], Jordi Cabot[2], Jesús García Molina[1]

[1]Department of Computers and Systems, Facultad de Informática, University of Murcia, Murcia 30071, Spain. `{jlcanovas, jmolina}@um.es`

[2]Atlanmod, INRIA & École des Mines de Nantes. La Chantrerie 4, rue Alfred Kastler B.P. 20722 - F-44307 Nantes, France. `{javier.canovas, frederic.jouault, jordi.cabot}@inria.fr`

[*]Corresponding author: Tel.:+34868884642. Fax: +34868884151

Structured abstract

Context. A software artefact typically makes its functionality available through a specialized Application Programming Interface (API) describing the set of services offered to client applications. In fact, building any software system usually involves managing a plethora of APIs, which complicates the development process. In Model-Driven Engineering (MDE), where models are the key elements of any software engineering activity, this API management should take place at the model level. Therefore, tools that facilitate the integration of APIs and MDE are clearly needed.

Objective. Our goal is to automate the implementation of API-MDE bridges for supporting both the creation of models from API objects and the generation of such API objects from models. In this sense, this paper presents the API2MoL approach, which provides a declarative rule-based language to easily write mapping definitions to link API specifications and the metamodel that represents them. These definitions are then executed to convert API objects into model elements or vice versa. The approach also allows both the metamodel and the mapping to be automatically obtained from the API specification (bootstrap process).

Method. After implementing the API2MoL engine, its correctness was validated using several APIs. Since APIs are normally large, we then developed a tool to implement the bootstrap process, which was also validated.

Results. We provide a toolkit (language and bootstrap tool) for the creation of bridges between APIs and MDE. The current implementation focuses on Java APIs, although its adaptation to other statically typed object-oriented languages is straightforward. The correctness, expressiveness and completeness of the approach have been validated with the Swing, SWT and JTwitter APIs.

Conclusion. API2MoL frees developers from having to manually implement the tasks of obtaining models from API objects and generating such objects from models. This helps to manage API models in MDE-based solutions.

Keywords: Application Programming Interface, Model-Driven Engineering, Domain-Specific Languages

1. Introduction

The concept of API (Application Programming Interface) is essential in software engineering as an expression of the principle of information hiding. The services that a supplier software asset offers to client applications are exposed through an API, a specification that hides implementation details and describes how to properly use services (at least their signature) and the kind of results they provide. API specifications come in different shapes depending on the kind of software asset. For example, class interfaces are used to specify object-oriented libraries while WSDL descriptions are applied when defining web services.

APIs are at the heart of several influential software development paradigms such as Service Oriented Architecture (SOA) or component-based development. They also play a key role in Web 2.0 in the construction of new web applications such as *mashups,* which integrate data and applications from different sources (e.g., Twitter, GoogleMaps, or Youtube). In fact, building any application usually involves managing a plethora of APIs to access different software assets such as: basic infrastructures (e.g., operating system, databases, or middleware), general-purpose or domain-specific libraries, frameworks, software components, web services, and even other applications. A good API is a competitive advantage for companies since they are the facade to the services they offer.

On the other hand, Model Driven Engineering (MDE) is becoming one of the most popular software engineering paradigms nowadays. MDE emphasizes the use of models to raise the level of abstraction and automation in all software engineering activities. Models have shown their potential for improving the quality and productivity of new software developments, reengineering of legacy systems and dynamically configuring running systems [1]. In any of these types of applications, MDE solutions normally involve the manipulation of APIs, especially object-oriented APIs in which this work is focused. For instance, two of the most common uses are the generation of API code from models in forward engineering, and the reverse engineering of legacy artefacts using APIs in model-driven reengineering. APIs are also used to access and modify data on existing applications (e.g., data in Web 2.0 applications) at runtime from MDE solutions. This runtime interaction requires that API objects can interoperate with MDE solutions, that is, API objects should be converted into/ generated from models. While the generation of API code from abstract models can be automated by using techniques such as template languages (e.g., XPand [2] and MOFScript [3]) and reverse engineering techniques for API code have been proposed [4, 5], the interoperability between APIs and MDE has received little attention and there are not tools that facilitate it.

Such tools providing API-MDE interoperability are an example of bridge between two technical spaces [6], and they should support two basic operations: i) obtaining models from a set of objects which are accessible through an API (e.g., Swing [7] Java objects or Twitter accounts), and ii) generating API objects from models. These bridging tools allow API manipulations to be performed at the model level, using a high-level view of the API, and offering a homogeneous treatment of all APIs involved in the software system at hand. As an example, models obtained from API objects could be used in scenarios such as web interoperability and Graphical User Interface (GUI) manipulation at runtime. For example, when models are obtained from web data, the interoperability and content aggregation between web applications is facilitated by applying MDE techniques on the model obtained. Similarly, models at runtime could be used to dynamically manage GUI API objects (e.g., Swing and the Standard Widget Toolkit (SWT) [8]), providing more maintainable and changeable solutions than traditional solutions based on statically generated code [9]. Automating the building of these bridging tools would facilitate the management of API models in MDE-based solutions since developers would be liberated from having to manually implement the tasks of obtaining models from API objects and generating such objects from models.

In this sense, this paper presents the API2MoL approach aimed at automating the implementation of API-MDE bridges. API2MoL is based on a rule-based declarative language to specify mappings between the artefacts of a given API (e.g., API classes in object-oriented APIs) and the elements of a metamodel that represents this API in the MDE technical space. Thus, a mapping definition provides the information which is necessary to build a bridge for a concrete API specification and metamodel. These API2MoL mapping definitions are bidirectional since the API2MoL engine uses them in both the process of creating a model out of API objects and the process of instantiating API classes from models. However, for large APIs, the definition of an equivalent metamodel and the specification of the mappings between the two can be time-consuming. To avoid this problem and ensure the applicability of our approach, API2MoL includes as well a bootstrap process able to automatically create both artefacts (i.e., metamodels and mapping definitions) from the inspection of the API.

API2MoL is, to the best of our knowledge, the first generic proposal to deal with the integration of MDE and APIs which automates the creation of the API-MDE bridge. Our proposal includes a complete prototype of a toolkit (language engine and bootstrap tool) focused on Java APIs, although an adaptation of the approach to deal with APIs for other statically-typed object-oriented languages such as C# could be easily implemented. This implementation has been empirically validated using the JTwitter [10], Swing and SWT APIs.

The rest of the paper is organized as follows. Section 2 presents the main challenges involved in integrating API with MDE. Section 3 presents the API2MoL language and Section 4 shows its execution semantics. Section 5 describes the bootstrap tool which generates both the metamodel and the mapping definition. Section 6 describes the process used to validate the approach. Section 7 outlines the implementation of the tools and Section 8 describes the state of the art with regard to the integration of APIs into MDE. Section 9 finalizes the paper and shows future work.

2. Overview of the approach

This section presents a high-level view of the main elements of our approach. As most bridges between two technical spaces, an API-MDE bridge should be bidirectional [6], that is, it should support both the creation of models from API artefacts and the generation of API artefacts from models. Throughout this paper, we will use the term *injection* to refer to the former process (we will say that the existing API objects are *injected into* an equivalent model) and the term *extraction* to refer to the latter (we will say that new API objects are *extracted from* the contents of the model).

In what follows we generically define both the injection and extraction processes for a bridge between object-oriented APIs and MDE. Object-oriented APIs are probably the most widely used in software development. In these APIs, the API artifacts are classes which are instantiated at runtime. In the rest of the paper we will focus on Java APIs but extension to other object-oriented APIs is straightforward. Once the injection and extraction processes are defined, we motivate the need of a language to express mappings between an API specification and an API metamodel, and finally we analyze how the construction of the bridge could be automated.

As preliminary concepts, we review first the correspondence between the standard four-level metamodeling architecture [11] and the instantiation levels for an API, in order to clarify the basis on which the concept of API-MDE bridge is built. In MDE, the instantiation relationship between a model and its metamodel is referred to as a "conformance" relationship, i.e., we say that "a model conforms to its metamodel". Likewise, we indicate that "an object graph conforms to its API" to express that API objects are instances of classes of a concrete API specification and that links between them conform to what is defined in the API fields or methods. For example, the Swing API contains classes such as `JButton` or `JFrame`, which can be instantiated at runtime for any program using the API to render a GUI with buttons and frames. Therefore, and although the concept of metaclass is not supported in the same way by the different object-oriented languages, it is possible to abstract the details and consider that object-oriented programming follows the same conformance (i.e., instantiation) levels as the four-level metamodeling architecture (see Figure 1). The model level in MDE corresponds to the API object level, the metamodel level corresponds to the API class level, and finally, the meta-metamodel level (e.g., Ecore) corresponds to the API metaclass level, which is defined at the level of the programming language used to implement the API (e.g., classes supporting reflection in Java).

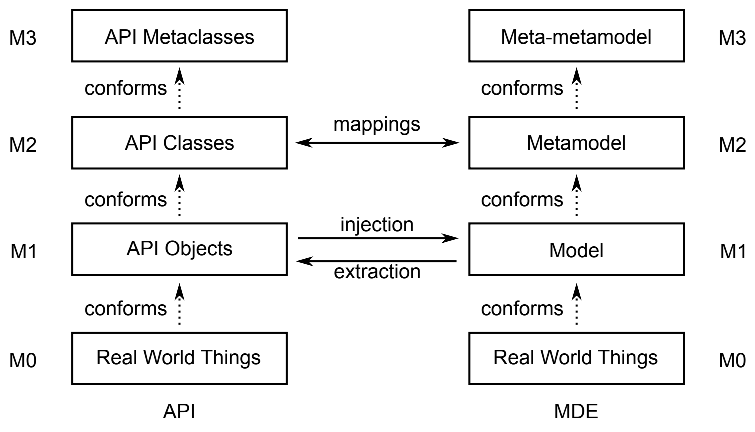


Figure 1. Bridge between API and MDE technical spaces.

2.1. From APIs to Models: Injection process

When injecting a model, it is first necessary to create a metamodel to represent the API. This metamodel will contain a metaclass for each class in the API in order to ensure that all the information provided by the API can be represented in the injected models. The level of abstraction of the metamodel is therefore the same as the API. Nevertheless, if desired, model transformations can be applied to represent the contents of this model at a higher abstraction level. The API object graph (i.e., the snapshot of a specific interaction between a program and the API) will therefore be expressed as a model conforming to the API metamodel.

The injected model should contain an element for each source API object in the program. These model elements conform to the metaclasses representing the API classes to which the source API objects conform, and they are initialized by API methods that return the objects' data (e.g., getter methods). Figure 2 illustrates the injection process by means of a simple example that only involves a `JLabel` object of the Swing API. For the sake of clarity, throughout this paper API classes are stereotyped as «APIClass» and their instances as «APIClassInstance», whereas metaclasses are stereotyped as «Metaclass» and their instances as «MetaclassInstance». The `JLabel` metaclass corresponds to the `JLabel` API class and the model element `m1`, which is an instance of the `JLabel` metaclass, is created to represent the API object `l1`. The features of `m1` (e.g., `text`) are initialized by using the getter methods of the `JLabel` API class (e.g., `getText`). Thus, to perform this injection process, the mapping information between the API classes and the metamodel metaclasses must be known, as illustrated in Figure 1 at M2 level.

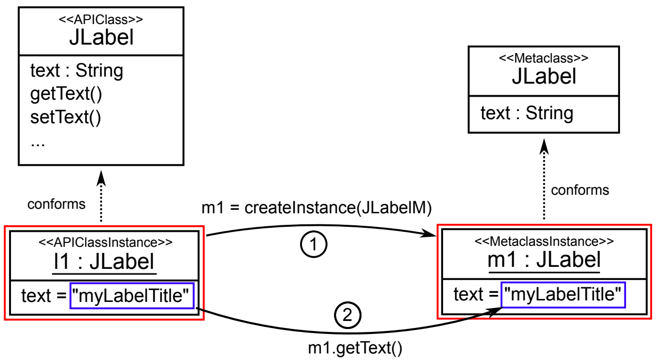


Figure 2. Model injection from API Java objects.

The model injection implies converting an object graph created by an object-oriented program into a model to be manipulated by an MDE-solution, as illustrated in Figure 2. This allows the application of MDE techniques to such objects at runtime. Some application scenarios are the following:

- Web data aggregation for integrating applications. For instance, models representing a Twitter account status could be transformed into LinkedIn models that could be used to automatically update the LinkedIn account based on the last tweet; using a model-based representation as a pivot for the interoperability between the tools would facilitate a lot the integration of other tools.
- Data analysis. The Portolan approach [12] injects models from the virtual servers of the cloud provided by a cloud computing vendor whose management is accessible via API. Once we have this model-based representation any available model-driven engineering technique could be used to implement analysis algorithms on this model and show them in an easy-to-understand graphical way to the cloud administrators in order to optimize the cloud performance.

2.2. From Models to APIs: Extraction process

The extraction process is applied in a similar way, but in the opposite direction. An extraction process has a model as its input, and generates an API object for each model element. These generated objects are initialized by invoking the methods provided by the API to manage object instances (e.g., setter methods). As before, the mapping information between the API specification and the metamodel is needed to identify the appropriate method for each model element feature. Figure 3 illustrates the extraction process for the same example shown in Figure 2. The **`JLabel`** API object is created from the instance of the **`JLabel`** metaclass. The **`text`** attribute of **`l2`** is initialized by using the setter methods of the **`JLabel`** API class (e.g., **`setText`**). Note that the proposed extraction process generates API objects which are directly created in memory at runtime, facilitating the changeability and maintainability of the API objects. Other possibility would be to generate source code including the set of calls to create and initialize the API objects.

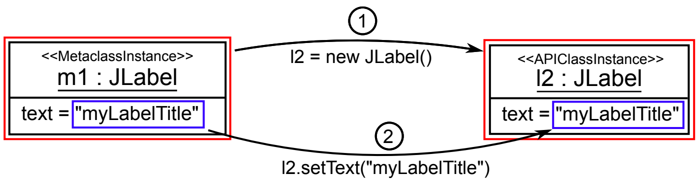


Figure 3. Model extraction into API Java objects.

Some possible model extraction scenarios are the following:

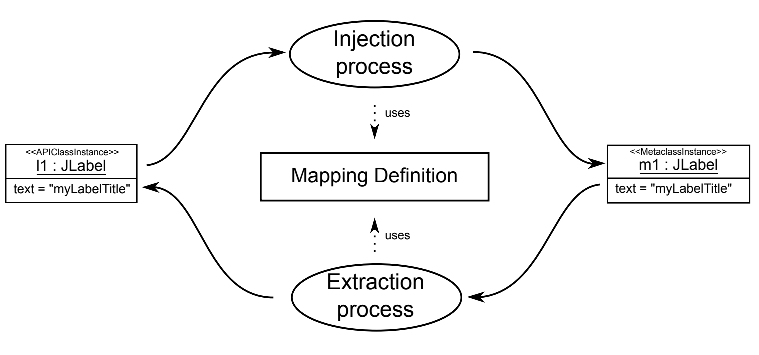
- Building GUI from models. For instance, the user could build a GUI model conforming to a GUI API (e.g., SWT or GWT [13]) metamodel, and the GUI would be rendered by executing the extraction process as supported by Wazaabi [9]. The elements of runtime GUI models act as proxies in charge of creating and manipulating the GUI objects. This scenario provides more maintainable and changeable solutions than traditional solutions based on statically generated code.

- Managing models at runtime. The process of extracting models can be combined with the injection process to manage models at runtime. In [14], models at runtime are used to represent information of the running system. These models are kept synchronized as the system execution progresses.
- On the fly reengineering. API objects can be converted into models as an intermediate step to either obtain objects for a different API or modify existing objects. Such combination would allow applying on the fly reengineering, where only runtime objects and models are managed, but no code. For instance, objects of a Swing GUI could be dynamically converted into SWT or GWT objects.

2.3. A mapping language to define both the injection and extraction processes

As we have seen before, any tool bridging APIs and MDE must know the mapping information between API classes and metamodel metaclasses in order to execute the injection and extraction processes. This knowledge could be hardcoded into the tool, but this would make such task complicated and specific for a concrete API. An alternative would be to provide a generic bridge parameterized by the mapping information for a specific pair <API, metamodel>, where the mapping would be expressed in some formalism. Bearing this idea in mind, we have defined the API2MoL approach aimed to automate the building of the injector and extractor for a given API, by providing a Domain Specific Language (DSL) for specifying the mappings between API classes and metamodel metaclasses.

Our DSL is a rule-based language that allows defining mappings declaratively. Thus, a mapping definition consists of a set of rules defined for a specific pair <API, metamodel>. It is worth noting that API2MoL rules are bidirectional and the same mapping definition can therefore be applied for both the injection and extraction processes. Figure 4a illustrates the API2MoL approach for the injection/extraction examples shown in Figures 2 and 3, whereas the Figure 4b shows the corresponding API2MoL mapping definition. The mapping definition includes only a rule (`JLabel : javax.swing.JLabel`) to specify the correspondence between the `JLabel` API class and the `JLabel` metaclass. The rule expresses how to inject/extract the `text` model attribute by including a section that specifies the methods provided by the API. This section contains a statement for each method used in the injection and extraction processes. Thus, to inject such attribute, the `getText` method must be called (statement `get getText()`), whereas to perform the extraction process, the `setText` method must be used (statement `set setText()`). Such a mapping definition would be interpreted by the generic bridge in order to inject `JLabel` objects into `JLabel` model elements, as well as extract `JLabel` objects from `JLabel` model elements.



```
JLabel : javax.swing.JLabel {
  text :
    get getText();
    set setText();
}
```

(b)

Figure 4. (a) The use of a mapping definition to perform the injection and extraction processes shown in Figures 2 and 3. (b) The mapping definition expressed in the API2MoL DSL.

Beyond this simple example, in Section 3 all the constructs offered by this language (i.e., several type of sections and statements) to adapt a mapping definition to the capabilities offered by an API are described.

2.4. Generating bridges automatically

Since APIs have normally a large number of classes, the DSL presented above represents only a partial improvement regarding the effort to develop an API-MDE bridge, as the creation of the metamodel and the mapping definition would still be tedious and time-consuming. To overcome this limitation, our approach includes as well a discovery process to automatically generate both the metamodel and the mapping definition from the inspection of the API classes, as showed in Figure 5.

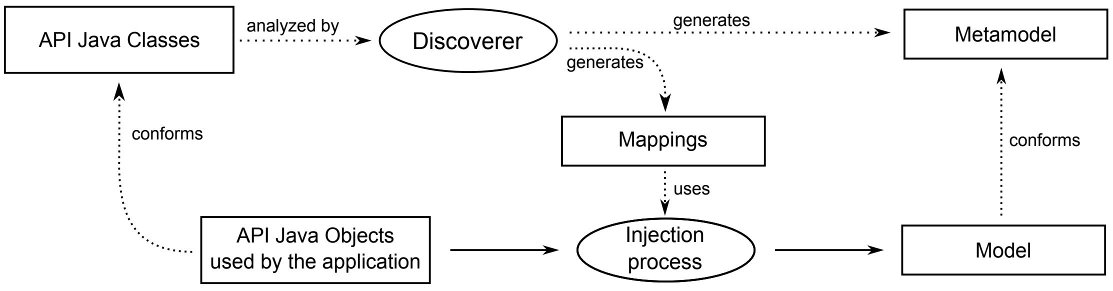


Figure 5. The process of discovering both the metamodel and the mapping definition. In the figure, mappings are used by the injection process but they could also be used by the extraction process.

Parsing and using reflection are two possible techniques to implement the discovery process. On the one hand, parsing tools (e.g., JDT or JastAdd) or byte code analyzers could be used to analyze the code of the classes of the API specification and generate both the metamodel and mapping definition. However, a parsing-based discoverer involves hardcoding AST traversals to analyze and understand the API code and it is only applicable when the API source code or byte code are available. On the other hand, a reflection-based discoverer analyzes the API by inspecting the API classes at runtime using the reflection capabilities supported by the programming language used to implement the API. The input of this discoverer would therefore be the reflection representation (i.e., a `Class` instance in Java) of each API class. The effort to develop a reflection-based discoverer is similar to that of using parsing tools since it would still be necessary to analyze and understand the reflection information of each object.

Nevertheless, since reflection capabilities are normally provided through a Reflection API, the reflection-based process can be automated taking advantage of the API2MoL language itself. Thus, it is possible to create an API2MoL mapping definition that obtains a reflection model describing the classes of any given API. This reflection model could then be used to automatically generate both the metamodel and the mapping definition for an API of the language considered. The generation process would be specified declaratively by means of model-to-model transformations, thus facilitating the process of defining the heuristics needed to discover both the metamodel and the mapping definition, as described in Section 5. Since the current API2MoL implementation deals with Java APIs, we have developed the discovery process using the Java Reflection API. We call "bootstrap process" to this discovery process, because we use API2MoL in order to discover the metamodel and mapping definition needed to apply API2MoL itself to an API.

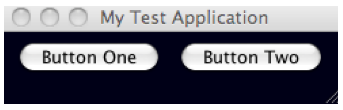
Since the heuristics incorporated to the discoverer may not deal with all API peculiarities, the bootstrap process might not generate the whole API metamodel and mapping definition, and the developer must therefore extend them manually. In these cases, the metamodel would be modified by using the corresponding meta-metamodel language (i.e., Ecore or MOF) while the mapping definition would be extended easily by using the API2MoL mapping language.

Once we have presented an overview of the approach, the following sections will describe their main elements: the API2MoL language and the bootstrap process.

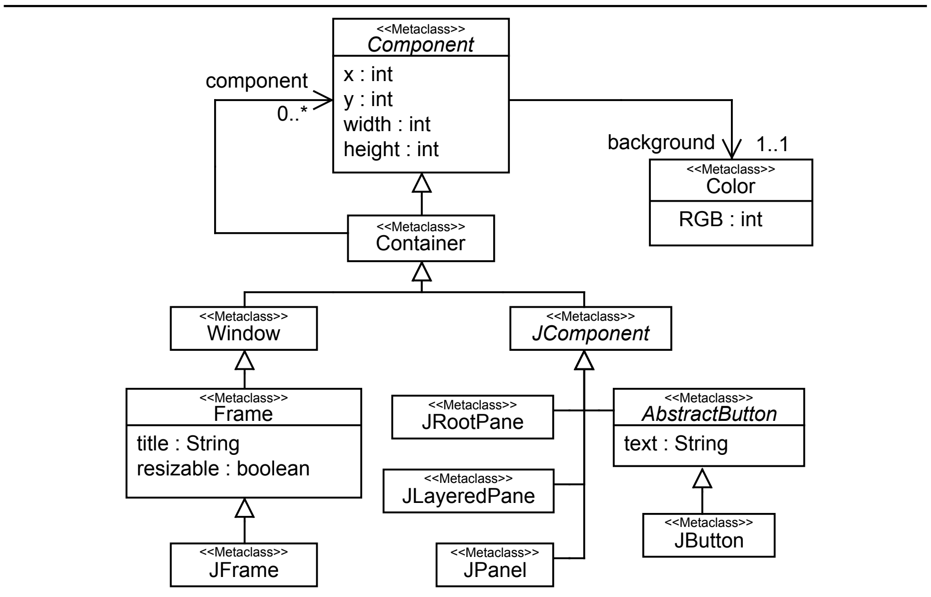
3. API2MoL mapping language

An API2MoL mapping definition consists of a set of mapping rules where each rule is responsible for defining the correspondence between a metamodel element (i.e., metaclass) and an API class, and specifies the mappings between the metaclass features and the API methods to be invoked when reading/writing those features, as indicated in the previous section. This information allows a model to be correctly injected from/extracted into the API objects. Obviously, the target API metamodel must already be available in order to write a mapping definition but Section 5 shows how this metamodel, and even the mapping definition themselves, could be automatically generated.

In order to illustrate the main concepts of the API2MoL DSL, we will use the Swing API as an example. Swing is an API based on the Abstract Window Toolkit (AWT) [15] that facilitates the development of GUIs for Java applications. For example, this API could be used to develop the simple GUI shown in Figure 6a, which is composed of a `JFrame` and two `JButton`s API objects. Figure 6b shows an excerpt of the API metamodel to which the models injected/extracted using API2MoL would conform. Note that this Swing metamodel mimics the Java classes of the Swing API. The metamodel therefore contains metaclasses for API classes (i.e., `JRootPane` or `JLayeredPane`) which are normally only used when instantiating a Swing object graph. Section 5 will show how we automatically derived part of this metamodel from the Java Swing API specification by applying the bootstrap process.



(a)



(b)

Figure 6. Swing example to illustrate the API2MoL language. (a) Swing example application to be injected/extracted (b) Excerpt of the Swing metamodel to which the injected models must conform.

A DSL consists of at least three elements [16]: abstract syntax, concrete syntax, and semantics. The abstract syntax (usually expressed as a metamodel) defines the concepts of the DSL and the relationships between them, and also includes the rules constraining the models that can be created with the DSL (typically known as well-formedness rules). The concrete syntax defines a notation (textual, graphical or hybrid) for the abstract syntax, and a translational approach is normally used to provide semantics. Both the abstract and concrete syntaxes are presented in this section while semantics of API2MoL are described in Section 4.

An excerpt of the API2MoL's abstract syntax metamodel is shown in Figure 7, whereas an example of its textual concrete syntax is shown in Figure 8. For the sake of clarity, Figure 7 does not include the metaclasses supporting the method and constructor overloading explained below. Figure 8a shows an excerpt of the grammar used for defining the concrete syntax, and Figure 8b presents the concrete syntax for the specific Swing example mentioned above. The complete version of both the abstract syntax and the concrete syntax can be downloaded from [17]. In what follows we describe the most important constructs of the language and their textual notation, while using the Swing example to illustrate them.

A mapping definition is represented by the `Definition` metaclass, which is the root element of the DSL metamodel. A mapping definition includes a `context` attribute, a *Default* section (`defaultMetaclass` reference) plus a set of mapping rules (`mappings` reference). A *context* is provided by one or more Java package names and they are used to delimit the injection process, that is, the set of classes to be considered. For example, the context in the Swing example is formed by `java.awt.*` and `javax.swing.*`, which are the packages for the Java classes used by Swing. Those Java objects that are not included in the context (i.e., unknown objects) will be injected according to the *Default* section. This section indicates the name of fallback target metaclass for all unknown objects. In addition, the section can specify the attribute name of the target metaclass which will store the class name of the unknown object for debugging purposes. In the Swing example, the *Default* section of the mapping definition specifies the `UnknownElement` metaclass and the attribute `type`.

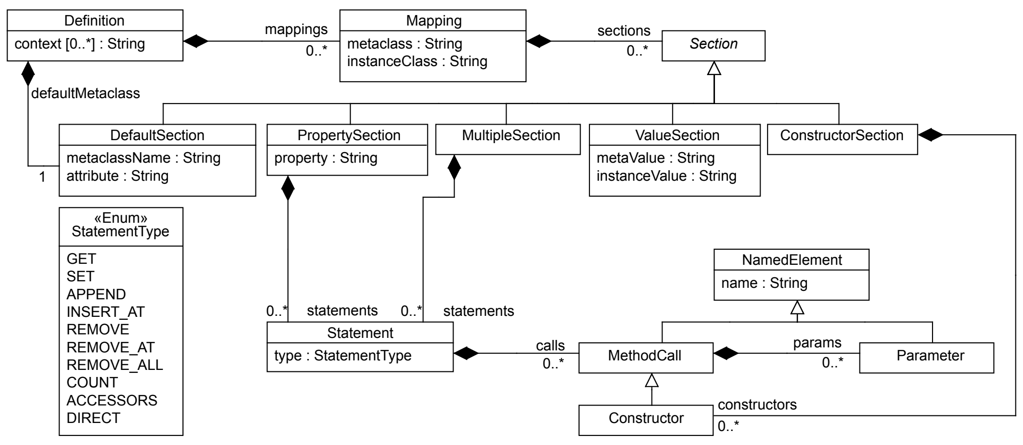


Figure 7. Excerpt of the API2MoL abstract syntax (the complete metamodel can be downloaded from [17]).

```
( <ContextSection> )? ';'
( <DefaultMetaclassSection> )? ';'

rule:
  contextSection?
  defaultMetaclassSection?
  mappingRule*

contextSection:
  '@' 'context' packageName (',' packageName)* ';'
  ;

defaultMetaclassSection:
  '@' 'defaultMetaclass' metaclassName '(' attributeName ')' ';'
  ;

mappingRule:
  metaclassName ':' className '{'
    ( defaultMetaclassSection | constructorSection |
      propertySection | multipleSection | valueSection)*
  '}'
  ;

constructorSection:
  '@' 'new' className '(' arguments ')' ';'
  ;

propertySection:
  propertyName ':' statement*
  ;

statement:
  statementType (methodName ('(' arguments ')')? )? ';'
  ;

statementType:
  'get' | 'set' | 'append' | 'insert_at' | 'remove' | 'remove_at' |
  'remove_all' | 'count' | 'direct'
  ;

multipleSection:
  '@' 'multiple' statement*
  ;

valueSection:
  objectValueName : modelValueName
  ;
```

(a)

```
@context javax.swing.*, java.awt.*;
@defaultMetaclass UnknownElement(type);

Component : java.awt.Component {
  x : get;
  y : get;
  width : get;
  height : get;
  background : accessors;
  @multiple
    set setBounds(x, y, width, height);
  …
}

Container : java.awt.Container {
  component:
    get;
    append add(element);
  …
}

Frame : java.awt.Frame {
  title : accessors;
  resizable : accesors;
  …
}

Window : java.awt.Container { … }

JFrame : javax.swing.JFrame { … }

JComponent : javax.swing. JComponent { … }

JRootPane : javax.swing.JRootPane { … }

JLayeredPane : javax.swing.JLayeredPane { … }

JPanel : javax.swing.JPanel { … }

AbstractButton : javax.swing.AbstractButton {
  text : accessors;
}

JButton : javax.swing.JButton { … }

Color : java.awt.Color {
  @new Color(RGB);
  RGB : get;
}
```

(b)

Figure 8. API2MoL concrete syntax. (a) Excerpt of the API2MoL grammar (the complete grammar definition can be downloaded from [17]). (b) Excerpt of the API2MoL concrete syntax for the Swing example.

Mapping rules consist of a header and a set of sections. The header specifies the Java class and the metaclass involved in the mapping (`metaclass` and `instanceClass` attributes of the `Mapping` metaclass). The Java class is identified through the use of its canonical name (e.g., the first rule of the example in Figure 8b specifies the mapping between the `Component` metaclass and the Swing Java class `java.awt.Component`). When a rule for a class is not available but such class is included in the *context*, a predefined rule is automatically applied by the language engine. Predefined rules map metaclasses and API classes (without the package prefix) which have the same names (e.g., if there was no rule for `java.awt.Component`, API2MoL would apply the predefined rule between this API class and the `Component` metaclass). These rules free developers from specifying mappings that are straightforward. Thus, an API2MoL mapping definition can combine both normal rules (i.e., rules defined by the developer for dealing with a particular mapping scenario) and predefined ones.

Sections define how the mapping is applied, that is, how the methods of the Java class indicated in the header must be invoked when reading/writing the metaclass features. There are five types of sections: property, default, multiple, constructor and value (subclasses of `Section` metaclass).

Property sections (`PropertySection` metaclass) specify a bidirectional mapping between a metaclass feature (i.e., attribute or reference) and an API class feature (i.e., mostly methods, but for some APIs, also fields when they are publicly available). These mappings are applied during both the injection and extraction processes (e.g., sections for the `title` and `resizable` features of the third mapping rule of the example), by simply changing the direction in which they are applied. Each *property* section specifies the name of the metamodel feature followed by a colon character and a set of *statements* (`Statement` metaclass) that describe the kind of access (e.g., *get* and *set* access) provided by the API to read/write that specific feature, along with the specific names of the methods (`MethodCall` metaclass) that implement that access in the API. It is possible to skip an explicit definition of the names of the affected methods since API2MoL can infer them from the statement type in most cases according to Java naming conventions. For example, for statements of the `GET` type, the `getX()` method is used as default where `X` is the mapping attribute. If the API uses a different method to obtain the value of `X` then it is necessary to specify the method name manually.

After analyzing several APIs, ten different types of *statements* have been identified, as listed in Table 1. These statements represent the typical kinds of methods provided by APIs to access and modify their internal objects, covering both the primitive-typed and collection features of those objects.

Type	Description
`GET`	Specifies the method to obtain the value for a metaclass property
`SET`	Specifies the method to assign a value to a one or more features of an API class
`ACCESSORS`	Specifies the existence of both SET and GET statements
`APPEND`	Specifies the API method used to add an elements to a collection
`INSERT_AT`	Specifies the API method used to include an element in a collection at a certain position
`REMOVE`	Specifies the API method to remove an element from a collection
`REMOVE_AT`	Specifies the API method used to delete an element from a collection at a certain position
`REMOVE_ALL`	Specifies the API method used to reinitialize a collection
`COUNT`	Specifies the API method in charge of counting the number of elements in a collection
`DIRECT`	Indicates that the attribute can be accessed directly. This statement is normally used to access public attributes

Table 1. Statements describing the type of access provided by the API.

Our Swing running example includes several of these statements: `GET`, `SET`, `ACCESSORS` and `APPEND`. For instance, in the first rule of the example, the `GET` statement is used in the `x`, `y`, `width` and `height` sections to obtain the value of these features. Since the API only includes the getter methods for these features, a `SET` statement for them is not provided (we will use a multiple section for them, see below). On the other hand, this rule uses the `ACCESSORS` statement for the `background` feature, since there exits both getter and setter methods in the API for such feature. The second rule of the example illustrates the use of `GET` and `APPEND` statements in the `component` section, which refers to a collection feature. The former is used to obtain the value to such feature whereas the latter allows adding a new element to the collection. The third rule of the example also illustrates the use of the `ACCESSORS` statement for both the `title` and `resizable` features.

The *Property* sections of a predefined rule only include a `GET` statement and a `SET` statement. Neither of them specifies the API method to be called, and are therefore inferred as explained before.

The *Default* section (`DefaultSection` metaclass) of a rule is similar to the *Default* section of a mapping definition but is applied to those classes which are subclasses of the class specified in the rule and for which a normal rule has not been defined. This section is normally used to avoid the use of predefined rules for such subclasses or when the class of the rule header cannot be instantiated. For example, if the first rule of the example had included a *Default* section, any Java subclass of `Component` without a normal rule would have been injected according to the contents of the *Default* section for `Component`.

Multiple sections (`MultipleSection` metaclass) are used only in the extraction process when the API defines methods that deal with more than one feature at the same time. A *Multiple* section is specified in the mapping definition by the `@multiple` keyword followed by one or more statements, whose declaration format has been explained previously. For example, the first rule of the example defines a *Multiple* section with which to specify that the `x`, `y`, `width` and `height` features can be set together by using the `setBounds` method.

The default constructor (i.e., the constructor without parameters) is normally used in the creation of Java objects from metamodel elements during the extraction process. However, it is sometimes necessary to specify a particular constructor that is available in the API. *Constructor* (`ConstructorSection` metaclass) sections are used when the default constructor is not available. These sections are specified by the `@new` keyword followed by the constructor method (`Constructor` metaclass), which must be used. For instance, the last rule of the example includes a *Constructor* section which specifies the constructor to be used for the `Color` object.

API2MoL offers support for enumeration values by means of *Value* sections (`ValueSection` metaclass). These sections are used to define a mapping between a metamodel enumeration value (`metaValue` attribute) and an enumeration value (`instanceValue` attribute) in the programming language (in our case, Java), and they are only used as part of a type of special rule called `enum`. Figure 9 shows an example of an `enum` rule which maps `DialogType` values and a Java enumeration type which is defined by constants declared in the `JRootPane` class of the Swing API. This rule includes several *Value* sections to map each `DialogType` value into a `JRootPane` value. The rule also specifies the type of the values, which in this case is integer (i.e., the Java enumeration type used in the Swing Java API).

```
enum DialogType : int {
  PLAIN    : javax.swing.JRootPane.PLAIN_DIALOG;
  WARNING  : javax.swing.JRootPane.WARNING_DIALOG;
  QUESTION : javax.swing.JRootPane.QUESTION_DIALOG;
  …
}
```

Figure 9. Example of `Enum` mapping rule and `Value` sections.

It is worth noting that API2MoL supports overloading in methods and constructors. When specifying a method in a statement, the argument type can be indicated in squared brackets to select the appropriate overloaded method, if any. For instance, if `Frame` class offered the methods `setTitle(String)` and `setTitle(Object)` and the first one must be used, the statement would be `setTitle([String])`. The same support is offered to constructors. For instance, if a `Color` class could be instantiated by using both the `Color(int)` and `Color(Object)` and the first one must be used, the constructor section must specify the `Color([int])` constructor. If a method/constructor is overloaded and the argument types are not indicated, API2MoL will select that method/constructor whose type conforms to the metamodel feature, which is the default behaviour.

Note that the bidirectionality of a mapping definition depends on the sections included in the mapping rules, which can restrict the behaviour of either the extraction or injection processes. For instance, the first rule of the Swing example defines a bidirectional mapping for the `Component` metaclass since every metaclass feature can be read/written from/to the Java class (the `background` feature has the `ACCESSORS` statement and the `x`, `y`, `width` and `height` have `GET` statements along with a multiple section to set them). However, in some cases the features of an API class may have a special access and the corresponding mapping rules will only include sections to perform either the extraction or injection process (e.g., read-only properties can only by injected).

4. Execution of API2MoL mapping definitions

In this section we describe how the API2MoL engine executes a bidirectional mapping definition. As explained in Section 2, our approach deals with API objects at runtime, which allows the developer to execute mapping definitions on-the-fly, thus facilitating its development and testing. However, accessing and creating API objects at runtime requires the use of a reflection library in the target programming language. In our case, the current implementation of API2MoL works with the Reflection Java API. Adaptations to other reflection APIs are straightforward.

Below, we define the procedural semantics of API2MoL language by describing how injection and extraction processes are performed for Java APIs.

4.1. The Injection Process

The injection process is applied on all in-memory API objects of an execution snapshot of the program to obtain the corresponding model representation. These API objects are arranged in an object graph including normally a root object which is the element starting the injection process (if there are several root objects, the process is repeated for each of them).

Given an API object, we obtain its class type and find the rule whose header matches that class. This rule provides the information concerning the metaclass in the metamodel that corresponds to the API object class. A new instance of that metaclass is created to represent the input API objet in the model. For example, in order to inject the `JFrame` object from the Swing example, the rule whose header is `JFrame:javax.swing.JFrame` will be located and an instance of the `JFrame` metaclass will be created. Note that for each API class, a transformation definition can only contain a rule that matches it, which can be either a normal or a predefined rule.

Once the metaclass has been instantiated, the next step consists in initializing its features (i.e., attributes and references) by invoking the appropriate methods on the API object to retrieve the corresponding values. The methods that must be called depend on the statement information (i.e., `GET` or `ACCESSOR` statement) included in the *Property* section of each feature. Since a rule only defines the mapping for the features declared in its header metaclass, the initialization of the inherited metaclass features also involves locating the *Property* sections declared in the rules corresponding to the superclasses of this metaclass. Each feature is thus initialized with the value returned by invoking the corresponding getter method on the instance of the API class. For example, when injecting a `JFrame` metaclass instance, rules for `Frame`, `Window`, `Container` and `Component` metaclasses will be located and their `GET` statements applied (e.g., `x` and `y` in the `Container` metaclass) because they are superclasses of the `JFrame` metaclass.

Two situations may arise when initializing a feature of a metaclass instance, depending on its type. If the type of the feature is a primitive type, the value returned by the getter method is directly assigned to the feature (e.g., the `title` feature of `Frame` metaclass). However, if the type is an API class, the value returned by the getter method is in its turn injected by recursively following the same process (e.g., since the `background` feature type of `Component` metaclass is `Color`, injecting `Color` will cause the execution of the rule whose header is `Color : java.awt.Color` and so on). The execution of a rule can therefore *trigger* other rules, so given an API object, the API2MoL execution mechanism also injects all the objects that are directly or indirectly connected to it. It is important to note that the infinite recursion is avoided by using a cache, where the runtime object reference identifies both the model element and the API object.

For example, given a `JFrame` object, the injection process returns a graph of instances of metaclasses, as can be seen in Figure 10, which shows an excerpt of the model injected from the Swing application example in Figure 6a. This model conforms to the Swing metamodel shown in Figure 6b. For simplicity, the values of certain attributes and references are not shown. Note how the structure of the object graph is automatically injected from the Swing application along with the buttons and background color.

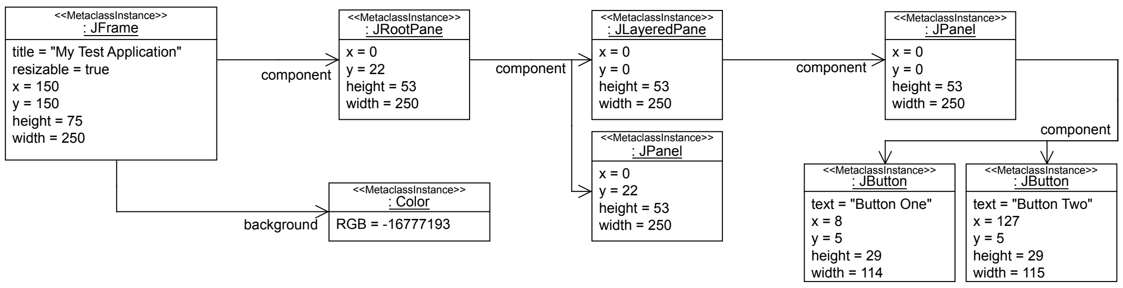


Figure 10. Excerpt of the Swing model injected from the Swing example.

4.2 The Extraction Process

The procedure for the extraction process is fairly similar. In an extraction process, an API2MoL definition is used to determine the API objects to be extracted from the model elements (i.e., instances of a metaclass in the metamodel). Likewise the injection process, the extraction process starts using the root model element as start element.

Given a model element, its metaclass is first obtained and then the rule which matches such metaclass is located. The API class to be used to instantiate the API object is inferred by the rule and the API object is then created by using either the default constructor (e.g., the instantiation of the `Frame` API class) or that defined in the *Constructor* section (e.g., the instantiation of the `Color` API class, which uses the `Color(RGB)` constructor).

Once the API object has been created, its fields then have to be initialized by extracting the values of the corresponding features of the model element by applying the mapping. Unlike the injection process, the statements used to extract the metaclass features are either `SET` (also considered by the `ACCESSOR` statement) or `APPEND`, depending on the type of the field of the API class (i.e., whether or not it is a collection). For example, when extracting a `JFrame` object, the `background` field will be extracted by using the `SET` statement, whereas the `component` field will be extracted by applying the `APPEND` statement. On the other hand, as with the injection process, since a rule only defines the mapping for the fields declared in its header class, the initialization of the inherited fields of the API class leads to the location of either the *Property* or *Multiple* sections declared in the rules corresponding to the superclasses of this class. The fields involved in *Multiple* sections are first extracted and the rest are then considered.

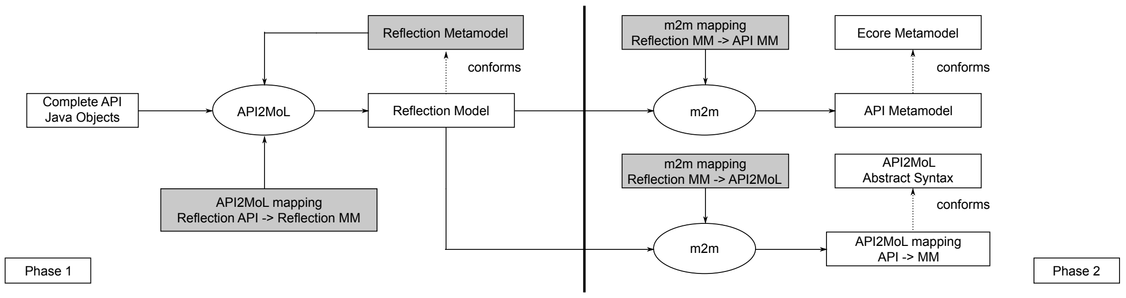
When extracting fields involved in a *Multiple* section, the statement to be applied is first obtained and then the involved method is used to initialize the fields. Once the *Multiple* sections have been executed, the rest of fields are extracted by locating its *Property* section and the involved method to initialize such field.

When initializing a field, the method to be called to perform the `SET`/`APPEND` statement must be parameterized using the values of the features of the model element, and two situations may arise depending on their types. If the type is primitive, the value of the metaclass feature is directly used as a parameter of the `SET`/`APPEND` method (e.g., the `title` field, which uses the setter method). If the type is a metaclass, the value of the metaclass feature is in turn extracted by executing the rule whose header matches the metaclass of this value, and the extracted value is used as a parameter (e.g., the `background` field, whose type is a `Color` API class, will cause the execution of the rule whose header is `Color : java.awt.Color` and the `SET` statement is then applied). As with the injection process, the execution of a rule can *trigger* other rules in the extraction process and the infinite recursion is also avoided by using a cache indexed by the runtime object reference. For example, given a `JFrame` metaclass instance, which is the root of the model shown in Figure 10, the extraction process returns a graph of class instances with the same structure that correspond to the view shown in Figure 6a.

5. The Bootstrap Process: Automatic Generation of API Metamodels and Mapping Definitions.

As explained in Section 2.4, we have defined a bootstrap process based on API2MoL itself that discovers the structure of the desired API elements and generates (almost completely) both the desired API metamodel and the mapping definition. Thanks to this bootstrap process, the developer simply needs to complement the mapping of those few API elements not covered by this process. Whereas this section describes the bootstrap process, the following section shows some empirical results of its completeness.

The process is composed of two phases, which are shown in Figure 11: (1) the API classes are represented as a model that conforms to a metamodel of the Reflection API of the language and (2) once this model is obtained, two model-to-model transformations are applied in order to generate a specific metamodel for the API and the corresponding mapping definition. Thanks to this, we can automatically obtain these two components for any API, as long as this API is implemented using a language that supports reflection.



Reflection Metamodel
conforms
Complete API Java Objects
API2MoL
Reflection Model
API2MoL mapping Reflection API -> Reflection MM
Phase 1
m2m mapping Reflection MM -> API MM
m2m
Ecore Metamodel
conforms
API Metamodel
m2m mapping Reflection MM -> API2MoL
API2MoL Abstract Syntax
conforms
m2m
API2MoL mapping API -> MM
Phase 2

Figure 11. The bootstrap process (handcrafted artefacts in gray boxes).

The first phase (phase 1 in Figure 11) uses API2MoL to obtain a model that represents the structural information of the input API (i.e., metadata describing the API classes). This is achieved by applying an API2MoL injection process that maps the reflection API into a Reflection metamodel created by hand. Since API2MoL currently supports Java APIs, we have handcrafted a simple Reflection Java API metamodel, which is shown in Figure 12 and represents the mains concepts of the reflection Java API (e.g., class, method, attribute, etc). We have also written a mapping definition for injecting reflection models conforming to this metamodel. The mapping definition only contains the **`GET`** statements needed to perform this process, which suffice to generate a reflection model of any Java API.

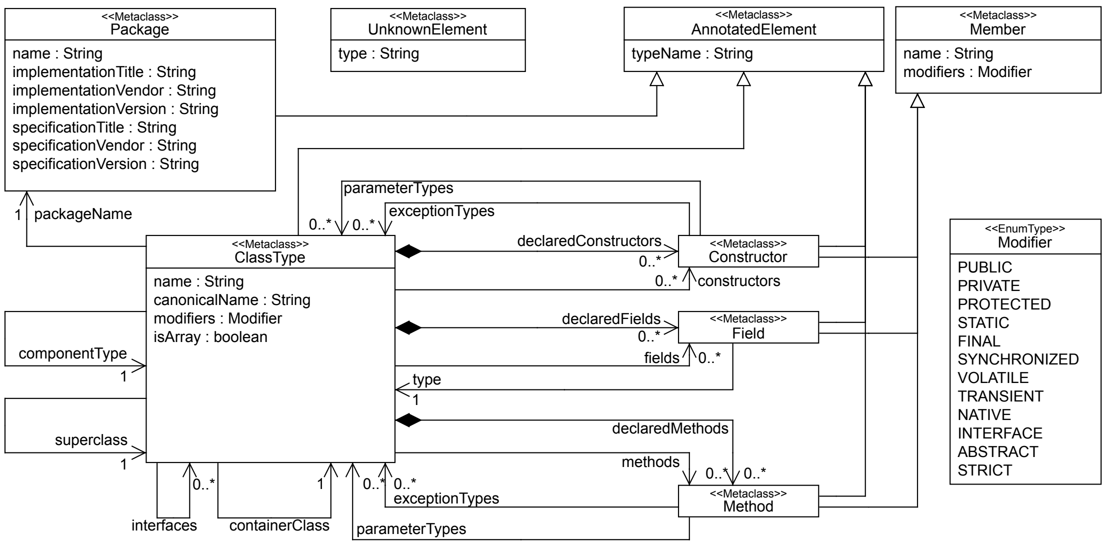


Figure 12. Excerpt of the Reflection metamodel used for describing Java API classes.

The second phase (phase 2 in Figure 11) applies two model-to-model transformations (specified using the ATL language [18]) that receive the reflection model as input and generate (1) the desired API metamodel and (2) the API2MoL mapping definition (expressed as a model conforming to the API2MoL metamodel). For some APIs, the generation may not be complete since they may present certain particularities that need a special treatment. We have identified a set of mappings and heuristics which cover almost every API feature, as will be shown in Section 6, although there are some features which are not still discovered. Nevertheless, the small percentage of these situations makes always worthy the application of our bootstrap process to kick-start the process. The two model-to-model transformations used to discover the API metamodel and the API2MoL mapping definition for a specific API are described as follows.

5.1. Discovering the API metamodel

The transformation used to generate the API metamodel maps each **`ClassType`** of the reflection model (Figure 12), which represents a Java API class, into a new metaclass in the target metamodel. These metaclasses have one feature for each field of the source API class (**`declaredFields`** reference) with public accessibility (i.e., the field has the public access modifier or a getter method). More specifically, each field is mapped into either an attribute or a reference depending on their type. If it is a primitive type, it is mapped into a metaclass attribute of the same type (e.g., an integer field is mapped into an integer attribute). However, if the type refers to another class in the API (i.e., it refers to another `ClassType` in the reflective model), it is mapped into a reference referring to the metaclass derived from mapping the `ClassType` element.

The `interfaces` and `superclass` references of the `ClassType` elements are used to define the hierarchical structure of the API classes. The former refers to the implemented interfaces and the latter refers to the superclass (both represented as `ClassType` elements), since Java language does not provide multiple class inheritance. However, as the metamodel does support multiple class inheritance, each element of both references is used as superclasses of the resulting metaclass.

Figure 13 illustrates how the above mappings are applied by using a simple example based on the Swing API. Figure 13a shows the reflection model (i.e., an instance of the Reflection Metamodel) obtained by injecting the `JButton`, `AbstractButton` and `Insets` Swing API classes in phase 1. In this figure, each API class is represented by a `ClassType` instance element. The `ClassType` whose `name` is `AbstractButton` thus contains two fields (`text` and `margin`) and two methods (`getText` and `getMargin`). The API metamodel obtained from the application of the previous mappings is shown in Figure 13b. As can be observed, each `ClassType` representing an API Class is mapped into a metaclass and the class fields have been mapped into either a metaclass attribute (i.e., the `text` field of the `AbstractButton` metaclass) or a metaclass reference (i.e., the `margin` reference of the `AbstractButton` metaclass). The hierarchical structure has also been discovered (`JButton` metaclass is a subclass of `AbstractButton` metaclass).

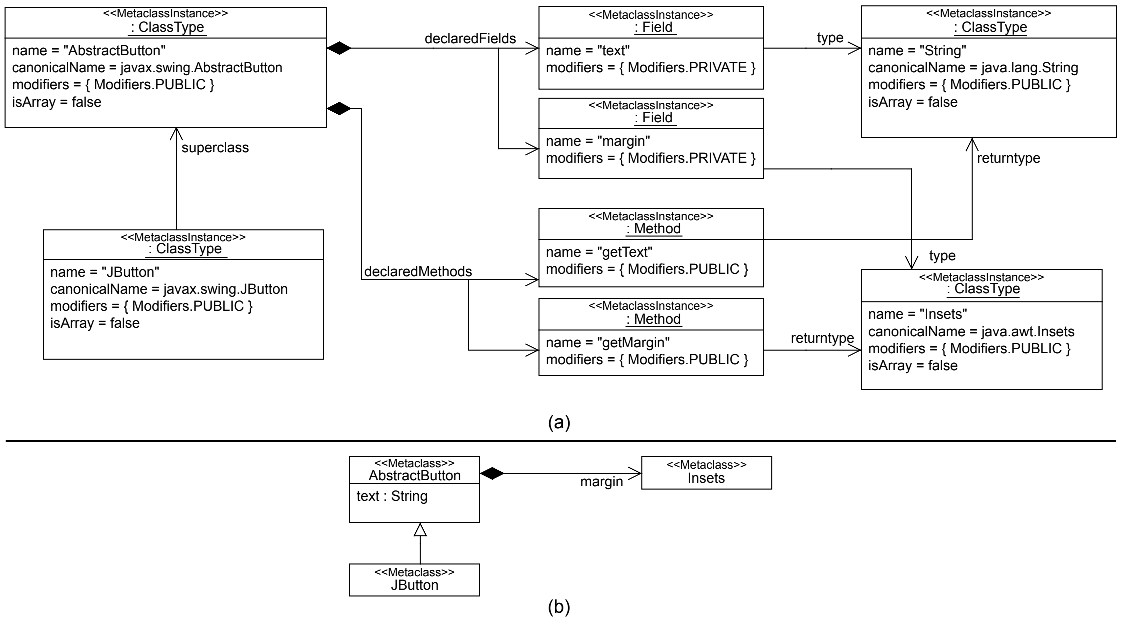


Figure 13. (a) An excerpt of the Swing reflective model and (b) the corresponding API metamodel discovered.

In addition to mappings expressing how to discover the structure of the API metamodel from the reflective model, two heuristics have been applied in order to complete the discovery process. The first heuristic deals with *accessor* methods which return information derived from certain fields (i.e., derived or calculated attributes). This heuristic analyzes the names of these methods in order to discover new metaclass features in the API metamodel which are not actually represented as class fields. These features can also be either an attribute or a reference, depending on the return type of the method. For example, if the `ClassType` named `AbstractButton` contained a calculated attribute called *perimeter* represented by the `getPerimeter` method whose return type is integer, a new integer attribute called `perimeter` would be included in the `AbstractButton` metaclass.

A second heuristic is applied when the class field is of collection type. Without applying the heuristic, these fields are first mapped into a multivalued metaclass feature, but it is still necessary to discover the type of the elements contained in the collection in order to generate either a multivalued attribute or a multivalued reference. For example, if a `ClassType` contains a field whose type is a list of string values, it must be mapped into a multivalued metaclass attribute whose type is `String`, whereas if the type of the field is a list of `ClassType` elements, it must be mapped into a multivalued reference referring to the corresponding metaclass that results from mapping this `ClassType`. In order to drive the generation of the metaclass features, this heuristic relies on the information concerning generics in the reflection model to discover the collection type. In cases in which generics are not used, API2MoL is currently unable to discover the element type of collections.

5.2. Discovering the API2MoL mapping definition

The transformation used to generate the mapping definition creates, for each `ClassType` in the reflection model, a new mapping rule between the `ClassType` and its corresponding metaclass in the generated API metamodel. Such rule includes the *Property* sections needed to inject/extract the metaclass features. Each *Property* section also contains the necessary statements, depending on the available API methods. For the moment, `SET`, `GET`, `DIRECT` and `APPEND` statement types are supported. They are added to the rule according to these heuristics:

- A `SET` statement is added if a setter method exists for the corresponding field of the API class.
- A `GET` statement is added if a getter method exists for the corresponding field of the API class.
- An `APPEND` statement is added if the field is a collection and an `ADD` method exists for the corresponding field of the API class.
- A `DIRECT` statement is added if the visibility of the field of the API class is public (i.e., it is directly accessible)

Figure 14 presents the API2MoL mapping definition corresponding to the reflection model shown in Figure 13a. The definition includes the mappings between the Swing API classes considered in the example reflection model and the API metamodel discovered. As can be seen, three mapping rules are added, one for each pair of API class and metaclass. Moreover, the mapping rule of the `AbstractButton` metaclass contains the *Property* sections for both the `text` and `margin` features. In this case, since the class in the example only provides the getter method, the *Property* sections only contains the `GET` statement type.

```
AbstractButton : javax.swing:AbstractButton {
  text :
    get;
  margin :
    get;
}

JButton : javax.swing.JButton { }

InSets : java.awt.Insets { }
```

Figure 14. An excerpt of the API2MoL mapping definition discovered.

6. Validation

The API2MoL approach has been validated to assess its correctness and completeness. The validation process has been carried out throughout the development of the tool supporting it. First, once the mapping language was designed and implemented, we verified the correctness of the injection and extraction processes with the Swing API, as explained below. Next, after we finished the implementation of the bootstrap process, we checked the completeness of both the language and the bootstrap process with three Java APIs: Swing, SWT and a Twitter API called JTwitter, thus allowing us to validate our approach with several real applications. However, since there are a number of different APIs, it is important to note that some specific API peculiarities could have not been considered and the validation results could differ. Figure 15 illustrates both processes (i.e., development and validation) applied in the implementation of API2MoL.

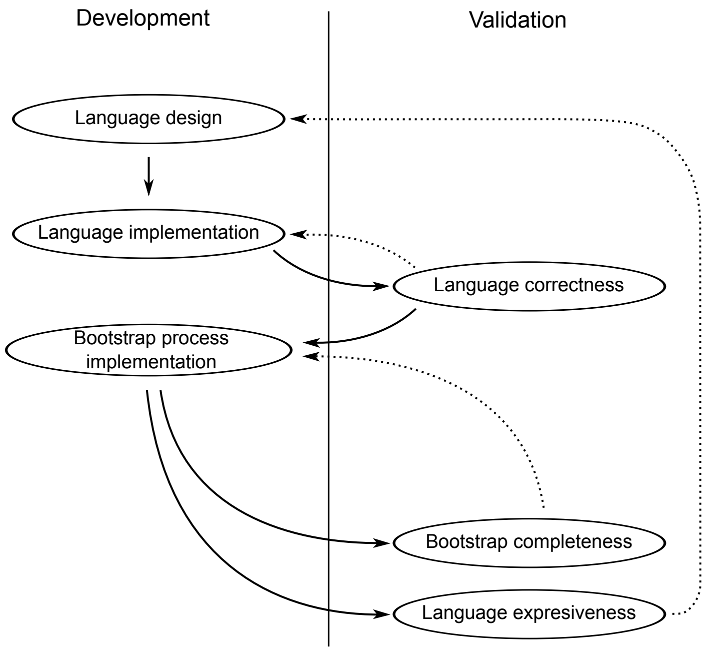


Figure 15. The development and validation tasks followed in API2MoL (dotted lines indicate that errors were found).

Given a set of API objects of an application (e.g., the set of GUI objects of a Swing application), the strategy we followed to check the correctness of the injection and extraction consisted in: (1) applying the injection process to the set of API objects in order to obtain an initial model; (2) applying the extraction process to such model obtained in order to generate a new set of API objects; and (3) reinjecting the API objects generated from the initial model, obtaining a new model representing the new input set of API objects. As a result of this process, two injected models representing the same set of API objects are obtained. If the injection and extraction processes are correct, these models must be identical. To perform the comparison we used EMFCompare [19]. The process is illustrated in Figure 16. We successfully applied this process using as test sets several Swing applications specially designed to check every language section and statement (these applications can be downloaded from [17]), obtaining the same resulting injected models for all of them.

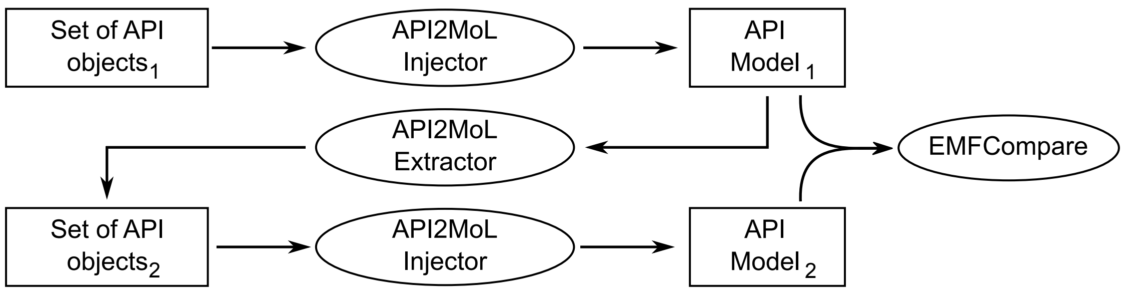


Figure 16. The process applied to verify the correctness of the injection and extraction processes.

In the second validation phase, as said above, we aimed at determining the completeness of both the API2MoL language and the bootstrap process. For this, we used different APIs: Swing, SWT and finally JTwitter. The meaning of "completeness" is different for the language and the bootstrap process. When validating the language, completeness refers to whether the language is sufficiently expressive to cover all the possible kinds of mappings between API classes and API metamodel elements, whereas in the case of the bootstrap process, it refers to whether both the automatically discovered API metamodel and API2MoL definition are complete. In what follows we describe the results for each test API and we finish the section with a summary of the completeness for both the language and bootstrap process part.

Swing API

Since the Swing API was used to test the correctness of the language, we used the same mock applications to check its completeness. We check the mapping definition used in each application and we did not find a lack of expressiveness.

Once the language had been validated, the bootstrap process was applied to automatically generate the Swing metamodel and the corresponding API2MoL definition. The injection and extraction processes were then executed again for each mock application and the results were compared to those handcrafted previously. We performed a manual comparison of both the metamodels (i.e., the bootstrapped metamodel and the handcrafted one) and mapping definitions (i.e., the bootstrapped definition and the handcrafted one) to detect the missing parts in the injection/extraction process when using only the bootstrapped elements.

Although the generated mapping definitions sufficed to cover simple applications, it was necessary to slightly extend the results of the bootstrap process due to limitations related to the discovery of specific constructors of the API class objects. In Swing, some class objects (e.g., `BevelBorder` or `EmptyBorder`) are constructed by specifying the initialization values as constructor parameters since there are no setter methods for these values. In this case, specific API2MoL *Constructor* sections had to be manually added to the mapping rules of some elements.

Due to the size of the Swing metamodel (it includes more than one thousand elements), and that of the mapping definitions, more details on the Swing tests cannot be included here. Any readers who are interested in these aspects can download them from [17].

SWT API

Similarly to Swing, SWT is another API which allows developers to create GUIs for Java applications. Whereas Swing has been built into Java technology and is therefore completely portable, SWT has the advantage of being implemented as a native application, thus improving performance and compatibility. With regard to the API structure, SWT differs in some aspects such as the creation and management of widgets.

The application used to validate API2MoL with SWT is a well-known example included in the API called `ControlExample`. This application uses all the widgets and layouts provided by the API. Thus, when checking the completeness two problems were found: the discovery of (1) public attributes and (2) specific constructors of the API objects. The former is particularly related to the structure of the SWT layout class, which defines a set of public attributes used to configure the graphical layout (i.e., there are no get or set methods). The language completeness was extended to include a new type of statement, called `DIRECT`, in order to support direct access to the public fields of Java classes. The bootstrap process was also modified to enable it to discover such statements and generate them automatically in the API2MoL definitions.

The second problem is the same as that of the Swing case and mainly affects widget elements, whose constructors normally receive the parent element. The solution adopted was the same as that shown in the previous case, and it was necessary to add manually specific API2MoL constructor sections to the mapping rules of some elements. As before, more details on the test are available at [17].

JTwitter API

Finally, we tested both the language and the bootstrap process with JTwitter, an open-source Java API for Twitter. JTwitter allows the twitter user account's data, such as followers, friends, and statuses, to be managed. This third test example also allows us show the usefulness of API2MoL in integrating the Web 2.0 application domain with model-driven techniques.

Both the language validation and the API2MoL bootstrap process were successfully applied to the JTwitter API using a Twitter test account named `api2moltest`. Figure 17 shows an excerpt of the metamodel discovered in the bootstrap process, which was completely and correctly generated so it was not necessary to modify. The metaclasses shown provide a representation of the main classes managed by the API, which correspond with the main concepts of the Twitter domain. The `Twitter` metaclass represents a twitter account and contains the information related to the twitter user. For example, it refers to the last user status information (i.e., last tweet) by means of the `status` reference whose type is `Twitter.Status`, users who are being followed are referred to by the `friends` reference whose type is `Twitter.User`, and user's private messages are referred to by the `directMessages` reference and represented by the `Twitter.Message` metaclass.

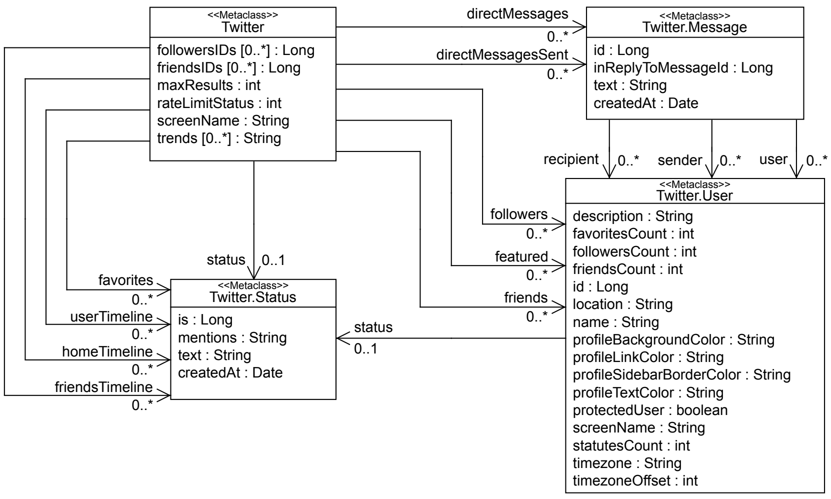
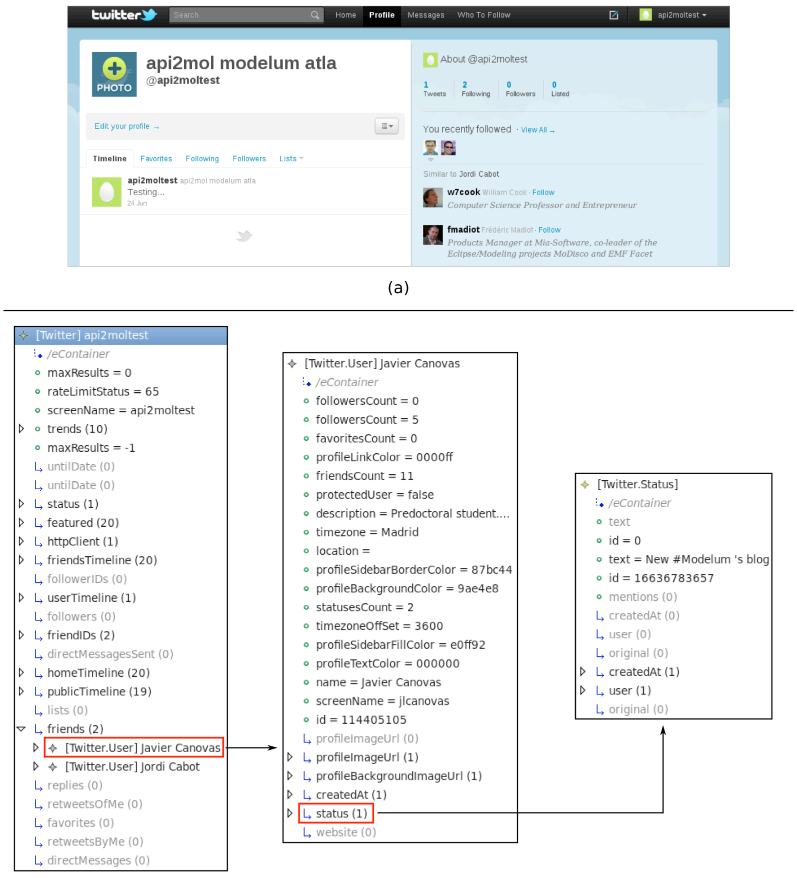


Figure 17. Excerpt of the discovered metamodel from the JTwitter API.

The mapping definition discovered in the bootstrap process contains the mappings and statements needed to perform both the injection and extraction processes. It was not therefore necessary to make any changes to the mapping definition.

Both the discovered metamodel and the mapping definition were used to inject a model which describes the Twitter test account. Figure 18a shows the test account used, and Figure 18b shows an excerpt of the injected model, which contains a **`Twitter`** instance, that refers to one **`Twitter.User`** instance, which in turn refers to a **`Twitter.Status`** instance that represents the status of one of the user's friends.



api2mol modelum atla
@api2moltest

api2moltest api2mol modelum atla
Testing...
24 Jun

About @api2moltest
1 Tweets | 2 Following | 0 Followers | 0 Listed

You recently followed

Similar to Jordi Cabot

w7cook William Cook
Computer Science Professor and Entrepreneur

fmadiot Frédéric Madiot
Products Manager at Mia-Software, co-leader of the Eclipse/Modeling projects MoDisco and EMF Facet

(a)

```
[Twitter] api2moltest
  /eContainer
  maxResults = 0
  rateLimitStatus = 65
  screenName = api2moltest
  trends (10)
  maxResults = -1
  untilDate (0)
  untilDate (0)
  status (1)
  featured (20)
  httpClient (1)
  friendsTimeline (20)
  followerIDs (0)
  userTimeline (1)
  followers (0)
  friendIDs (2)
  directMessagesSent (0)
  homeTimeline (20)
  publicTimeline (19)
  lists (0)
  friends (2)
    [Twitter.User] Javier Canovas
    [Twitter.User] Jordi Cabot
  replies (0)
  retweetsOfMe (0)
  favorites (0)
  retweetsByMe (0)
  directMessages (0)
```

```
[Twitter.User] Javier Canovas
  /eContainer
  followersCount = 0
  followersCount = 5
  favoritesCount = 0
  profileLinkColor = 0000ff
  friendsCount = 11
  protectedUser = false
  description = Predoctoral student....
  timezone = Madrid
  location =
  profileSidebarBorderColor = 87bc44
  profileBackgroundColor = 9ae4e8
  statusesCount = 2
  timezoneOffSet = 3600
  profileSidebarFillColor = e0ff92
  profileTextColor = 000000
  name = Javier Canovas
  screenName = jlcanovas
  id = 114405105
  profileImageUrl (0)
  profileImageUrl (1)
  profileBackgroundImageUrl (1)
  createdAt (1)
  status (1)
  website (0)
```

```
[Twitter.Status]
  /eContainer
  text
  id = 0
  text = New #Modelum 's blog
  id = 16636783657
  mentions (0)
  createdAt (0)
  user (0)
  original (0)
  createdAt (1)
  user (1)
  original (0)
```

(b)

Figure 18. API2MoL injection process applied to a Twitter account by using the JTwitter API. (a) The twitter test account (b) An excerpt of the injected model

Language completeness summary

The API2MoL mapping language covered the great majority of the mappings required in the tests excepting for the `DIRECT` statement in the case of the SWT API. Once this statement was incorporated to the mapping language, the language completeness is 100% for the involved APIs, since it is expressive enough to cover all the possible kinds of mappings required for each API compared.

Bootstrap completeness summary

Table 2 shows the level of completeness obtained by applying the bootstrap process. The tests were performed once the new `DIRECT` statement had been added to the language. For each API we compared the number of generated elements with the real number of elements. For the API2MoL definitions we have considered rules and sections, whereas classes, attributes and references have been considered for the API metamodels. The number of missed elements in the bootstrap process has been calculated through a manual inspection.

API	Mapping definition			API metamodel		
	Bootstrapped	Real	Completeness	Bootstrapped	Real	Completeness
Swing	1116 mappings 1053 sections	1116 mappings 1240 section	100% 84,9%	1197 classes 446 attributes 486 references	1197 classes 446 attributes 486 references	100% 100% 100%
SWT	689 mappings 798 sections	689 mappings 1096 section	100% 72,8%	708 classes 492 attributes 251 references	708 classes 492 attributes 251 references	100% 100% 100%
JTwitter	53 mappings 160 sections	53 mappings 160 sections	100% 100%	71 classes 76 attributes 45 references	71 classes 76 attributes 45 references	100% 100% 100%

Table 2. Completeness of the API2MoL bootstrap process. Comparison between bootstrapped and real versions of both the mapping definition and the API metamodel used in several API examples

As can be observed, the API metamodel generation process is complete for the three APIs considered. This is principally due to the fact that the mappings and heuristics identified in the bootstrap process cover the vast majority of the API peculiarities. The API2MoL definition generation process also discovered all the rules in the three API tests and the vast majority of sections. However, for some APIs it was necessary to manually add *Constructor* and *Multiple* sections which are specific to the API, as was explained previously for Swing and SWT. According to the comparison, 15.1% of sections have been added for Swing and 27.2% in the case of SWT. In both cases, the API2MoL mapping language facilitated the task of finishing the mapping definitions.

7. Tool Support

Both the API2MoL language and the injection and extraction processes have been implemented on top of the Eclipse platform, as described in this section (they can be downloaded from [17]). Figure 19 outlines the architecture of the API2Mol Engine. A brief description of each component is shown as follows:

- The *API2MoL injector*, which is in charge of performing the injection process.
- The *API2MoL extractor*, which is responsible for extracting models from API objects.
- The *API2MoL projector,* which offers common services to both the *injector* and the *extractor*. Its main task is to create models from the API2MoL definition, which conforms to the API2MoL metamodel. It also uses a *model manager* and a *reflection helper*.
- The *model manager* allows both the injector and the extractor to manage models generically (e.g., model element creation or feature initialization).
- Finally, the *reflection helper* is used to reflectively manage the calls to the API methods.

As shown in Figure 19, given a textual API2MoL definition, the *API2MoL projector* is in charge of transforming it into a model, which conforms to the API2MoL metamodel. This API2MoL definition model is then used by either the *API2MoL injector* to convert API objects into models or by the *API2MoL extractor* to convert models into API objects.

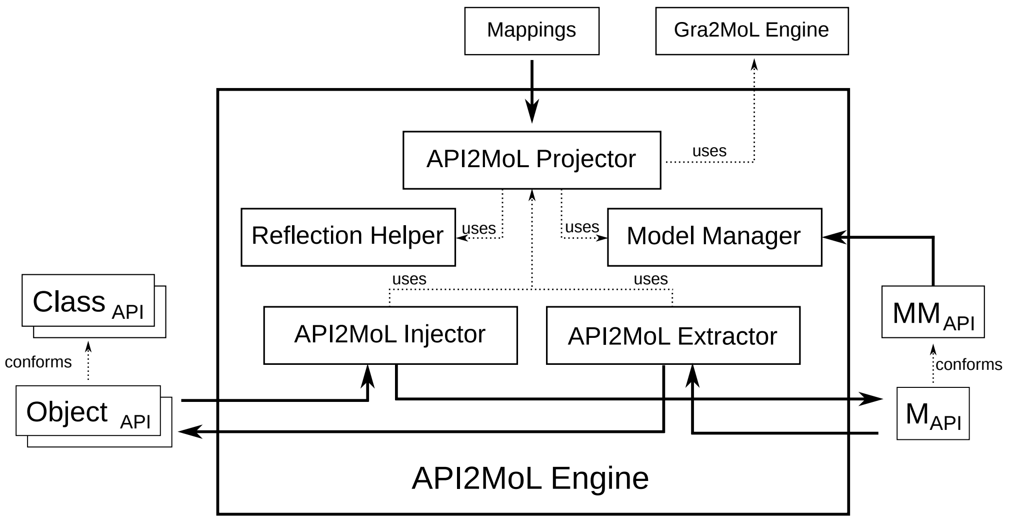


Figure 19. API2MoL architecture.

The API2MoL language can be created by using any of the available tools for the definition of textual DSL, such as XText [20], EMFText [21], TCS [22] or Gra2MoL [23]. Since TCS and Gra2MoL have been developed by the groups involved in this research work, they were the candidates used in order to ease the creation of the DSL. We finally chose Gra2MoL to create the concrete syntax of the language, but we have also started experimenting with TCS (notably to enable extraction of generated API2MoL models to textual syntax). The Gra2MoL engine is therefore used to parse the text of an API2MoL definition and transform the concrete syntax tree obtained into an abstract syntax model. The *projector* thus acts as a front-end, which receives an API2MoL definition from the user and provides the *injector* and *extractor* components with the corresponding model, by using the *Gra2MoL engine* to inject the model.

Model manager and *reflection helper* are auxiliary components which provide services related to the model and reflection API management. *Model manager* uses the infrastructure provided by MoDisco [24] to provide support with which to manage injected/extracted models independently of the metamodel to which they conform. On the other hand, the *reflection helper* allows the Reflection API of the language (i.e., Reflection Java API) to be accessed transparently.

8. Related work

To the best of our knowledge, ours is the first generic approach to build API-MDE bridges. A related approach is [4, 25], where a methodology to create framework-specific modeling languages (FSML) is presented. A FSML allows developers to represent domain-specific concepts provided by framework APIs. The abstract syntax of a FSML is similar to the API metamodel discovered by API2MoL. However, unlike API2MoL, such abstract syntax must be defined manually as well as the injector and extractor of models conforming to it.

Other close tools to API2MoL are those that: i) inject ontologies from source code [5], ii) inject models from source code, such as MoDisco [24] and Gra2MoL [23], or iii) use runtime models to dynamically create and manipulate application's artefacts, such as graphical user interfaces in Wazaabi [9] or systems providing a management API in SM@RT [14]. In what follows we compare our approach with these three kinds of related work.

The approach presented in [5] allows obtaining a common ontology for a set of APIs which share the application domain. These ontologies can be used in reverse engineering tasks such as program understanding and quality checking. However, the approach does not provide API-MDE bridges since the corresponding injector/extractor for specific APIs is not generated.

MoDisco (Model Discovery) is an extensible framework for model-driven reverse engineering whose objective is to facilitate the development of injectors (*discoverers* in the MoDisco terminology) in order to obtain models from legacy systems and use them in modernization use cases. XML and Java discoverers are available. In the case of the Java discoverer, it uses the JDT API to create models out of the Java code. However, MoDisco discoverers ignore API interactions when performing the model injection process and only work when the source code is available. API2MoL could be integrated with

MoDisco to provide more complete discoverers capable of obtaining models that consider the interaction of the legacy systems with the diverse APIs available.

Similarly, Gra2MoL (and related approaches such as TCS) is a DSL for obtaining models from source code by means of text-to-model transformations. It has been especially tailored to address the problem of model injection, thus making this task easier and more productive. The language provides a powerful query language for concrete syntax trees, and mappings between source grammar elements and target metamodel elements are expressed by rules similar to those found in model transformation languages. Gra2MoL can be regarded as an alternative approach for the development of MoDisco discoverers. Like MoDisco, Gra2MoL does not provide any support for APIs, so API2MoL could also be used to enrich the model injection process in the same way as that described for MoDisco.

Wazaabi is a tool for building GUIs for different technologies, such as SWT, JSF and Swing. The GUI models built by developers are dynamically processed by an engine in order to render the corresponding user interface. Unlike API2MoL, Wazaabi is only useful for forward generation (i.e., producing GUIs from models but not the other way round) and can only deal with SWT, JSF and Swing APIs. Moreover, API2MoL adds a level of automation because the tools needed to obtain models are automatically generated from the mapping definition. The part of Waazabi which generates GUIs from models could be replaced by an API2MoL extraction process.

The SM@RT approach allows a synchronization engine to be generated between a running system and its model. The inputs of the generation process are the metamodel of the running system to which the model must conform and an access model describing how to synchronize the model elements with the running system management API (e.g., JMX API). [26] describes an automated approach with which to infer the input metamodel, based on parsing the source code by using the JMX API of JEE systems. On the other hand, the access model is actually the mapping definition between the metamodel and the API elements but, unlike the API2MoL approach, it is defined imperatively by means of code templates (i.e., for each model element a code template specifies the code needed to manipulate the API element). Note that APIs have a great number of elements and that a considerable effort would be required to write the mapping with this approach. However, API2MoL offers a more automated approach which allows both the API metamodel and the mapping definition to be generated in almost their entirety. Moreover, it is easy to add both the missing metamodel elements and the mapping rules in a declarative manner.

Outside the MDE technical space, other methods for API manipulation have also been proposed, such as in software reengineering processes, in which an API migration is usually performed to adapt the source code of an application which uses a particular API to use a different one. This adaptation is normally tackled by means of developing wrappers for the target API [27, 28]. API2MoL could be used for this purpose as well, facilitating the API migration scenario by means of automating the interactions with the APIs by first expressing the APIs as models and then defining the API mappings at model level.

9. Conclusion and Future Work

We have presented API2MoL, an approach to bridge the gap between APIs and MDE and facilitate their integration in software engineering scenarios such as data aggregation for integrating applications and managing models at runtime. With API2MoL, API artefacts can be transformed into models and vice versa. Thanks to the API-MDE bridges automatically created by this approach, developers are liberated from having to manually implement the tasks of obtaining models from API objects and generating such objects from models. Therefore, API2MoL may improve the productivity and quality of the part of the MDE application that deals with the APIs.

API2MoL is based on two main elements: a declarative rule-based mapping language and a bootstrap process. The mapping language allows designers to specify the relationships between the API classes and a metamodel. This mapping information is then used to drive both the injection and extraction process. A bootstrap process has also been implemented in order to automatically generate the metamodel and the mapping definition for a given API. This bootstrap is essential in an API-MDE integration since APIs are normally large, and a manual definition of these elements would thus be error-prone and time-consuming. The correctness, expressiveness and completeness of both the language and bootstrap process have been verified with several APIs. Both have also been completely implemented.

At the moment, API2MoL is being used as core component of two new software products currently under development by two French software companies. In both projects, API2MoL plays the role of facilitating the interoperability between the APIs of several competing products. The key contribution of API2MoL is to provide end-users with a single entry point to access the products and to exchange information between them. Instead of having to created point to point bridges, API2MoL is used as a pivot for all of them.

As further work, we are currently using API2MoL with other APIs such as JDT, LinkedIn, Twitter4J and java-twitter in order to illustrate its applicability in more scenarios and to test the results when the tool is applied to different APIs for the same application (e.g., Twitter). We are also working on extending API2MoL to cover non object-oriented APIs, such as web service descriptions. In fact, the JTwitter example shown in the paper is a first approach towards using API2MoL in the context of web applications. In this respect, we are particularly interested in using API2MoL to facilitate the interaction at the model level between applications and external web services (which could be regarded as a type of external APIs) and to facilitate both the injection and extraction processes from the data provided by these services. Moreover, since a crucial issue for developers is to assess the level of API compatibility when new versions of APIs are released, API2MoL could also be used to facilitate such task by comparing the bootstrapped metamodel of different API versions. We are also planning to use API2MoL to discover the abstract syntax of internal DSLs developed as fluent interfaces [29], whose concrete syntax is provided by an API which specifies the methods offering the DSL functionality.

Finally, to improve the current implementation strategy (which only deals with in-memory runtime objects) we plan to support the *Java Debug Interface* (JDI) in order to be able to access in-memory objects which are not directly accessible via reflection because they are not in the same Java virtual machine as API2MoL (e.g., objects of a J2EE deployed application are only accessible by communicating with the application server in which they are deployed).

Acknowledgment

This work has been partially supported by Spanish Ministry of Science and Innovation (grant TIN2009-11555), Regional Government of Murcia (project 129/2009 and grant 15389/PI/10), and up to 80% of it has been co-supported by INFO and FEDER (grant PCTRM 2007-2010). Javier Luis Cánovas Izquierdo enjoys a doctoral grant from the Fundación Séneca.

References

[1] J. Bezivin, M. Barbero, and F. Jouault; “On the Applicability Scope of Model Driven Engineering”, in 4th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software, (MOMPES '07). 2007

[2] XPand language reference. Available from: http://www.openarchitectureware.org/pub/documentation/4.0/r20_xPandReference.pdf. December 2010

[3] MOFScript. Available from: http://www.eclipse.org/gmt/mofscript. December 2010

[4] M. Antkiewicz, K. Czarnecki and M. Stephan. “Engineering of Framework-Specific Modeling Languages”. IEEE Transactions on Software Engineering, vol 36, no. 6 (2009) pp. 795-824

[5] D. Ratiu, M. Feilkas, J. Jürjens. “Extracting Domain Ontologies from Domain Specific APIs”, in 12th European Conference on Software Maintenance and Reengineering (CSMR’08) 2008, pp. 203-212

[6] I. Kurtev, J. Bezivin and M. Aksit, “Technological spaces: An initial appraisal”, in International Conference on Cooperative Information Systems, Industrial track (CoopIS’02) 2002

[7] Swing. http://java.sun.com/products/jfc/tsc/articles/architecture. March 2011

[8] SWT. Available from: http://www.eclipse.org/swt. December 2010

[9] Wazaabi. Available from: http://wazaabi.org. December 2010

[10] JTwitter. Available from: http://www.winterwell.com/software/jtwitter.php. December 2010

[11] MDA Specifications. Available from: http://www.omg.org/mda/specs.htm. December 2010

[12] Portonlan project. Available from: http://code.google.com/a/eclipselabs.org/p/portolan/. December 2010.

[13] GWT. Available from: code.google.com/webtoolkit/. July 2011

[14] H. Song, Y. Xiong, F. Chauvel, G. Huang, Z. Hu and H. Mei, “Generating Synchronization Engines between Running Systems and Their Model-Based Views”, in proceedings of Models conference Workshops (2009), pp. 140-154.

[15] AWT. Available from: http://java.sun.com/products/jdk/awt/. July 2011

[16] A. Kleppe, “Software Language Engineering: Creating Domain-Specific Languages Using Metamodels”, Addison-Wesley Professional. 2008

[17] API2MoL website. Available from: http://modelum.es/api2mol. December 2010.

[18] F. Jouault, F. Allilaire, J. Bezivin and I. Kurtev, “ATL: A model transformation tool”, in Science of Computer Programming, Volume 72, Issues 1-2, Special Issue on Second issue of experimental software and toolkits (EST) (2008), pp. 31-39.

[19] C. Brun and A. Pierantonio, “Model Differences in the Eclipse Modelling Framework”, in European Journal for the Informatics Professional (CEPIS UPGRADE), Issue 2, 2008.

[20] XText. Available from: http://www.eclipse.org/Xtext. December 2010.

[21] F. Heidenreich, J. Johannes, S. Karol, M. Seifert and C. Wende. “Derivation and Refinement of Textual Syntax for Models”, in 5th European Conference on Model Driven Architecture Foundations and Applications (ECMDA-FA’09) (2009), pp. 114-129.

[22] F. Jouault, J. Beezivin, and I. Kurtev, “TCS: a dsl for the specication of textual concrete syntaxes in model engineering”, in International Conference on Generative Programming and Component Engineering (GPCE) (2006), pp. 249-254.

[23] J. Cánovas and J. García Molina. “A Domain Specific Language for Extracting Models in Software Modernization”, in 5th European Conference on Model Driven Architecture Foundations and Applications (ECMDA-FA’09), LNCS 5562 (2009), pp. 82-97, (downloadable from http://adm.omg.org/adm_info.htm#white papers).

[24] H. Bruneliere, J. Cabot, F. Jouault and F. Madiot. “MoDisco: A Generic And Extensible Framework For Model Driven Reverse Engineering”, in demonstration track of the 25th International Conference on Automated Software Engineering (ASE’10) (2010), pp. 173-174.

[25] M. Antkiewicz and K. Czarnecki. “Framework-Specific Modeling Languages with Round-Trip Engineering”, in proceedings of Models conference (MODELS’06) (2006), pp. 692-706.

[26] H. Song, G. Huang, Y. Xiong, F. Chauvel, Y. Sun, H. Mei, “Inferring Meta-models for Runtime System Data from the Clients of Management APIs” , in proceedings of Models conference (MODELS’10), pp. 168-182. 2010.

[27] T. Tonelli Bartolomei and K. Czarnecki and R. Lämmel. “Swing to SWT and Back: Patterns for API Migration by Wrapping”. Draft published online 4 May 2010.

[28] T. Tonelli Bartolomei and K. Czarnecki and R. Lämmel and T. van der Storm. “Study of an API migration for two XML APIs”, in postproceedings of Software Language Engineering (SLE’09). Springer. 2009.

[29] M. Fowler. “Domain-Specific Languages”. Addison Wesley, 2011.

Figure Captions

Figure 1. Bridge between API and MDE technical spaces.

Figure 2. Model injection from API Java objects.

Figure 3. Model extraction into API Java objects.

Figure 4. (a) The use of a mapping definition to perform the injection and extraction processes shown in Figures 2 and 3. (b) The mapping definition expressed in the API2MoL DSL.

Figure 5. The process of discovering both the metamodel and the mapping definition. In the figure, mappings are used by the injection process but they could also be used by the extraction process.

Figure 6. Swing example to illustrate the API2MoL language. (a) Swing example application to be injected/extracted (b) Excerpt of the Swing metamodel to which the injected models must conform.

Figure 7. Excerpt of the API2MoL abstract syntax (the complete metamodel can be downloaded from [17]).

Figure 8. API2MoL concrete syntax. (a) Excerpt of the API2MoL grammar (the complete grammar definition can be downloaded from [17]). (b) Excerpt of the API2MoL concrete syntax for the Swing example.

Figure 9. Example of `Enum` mapping rule and `Value` sections.

Figure 10. Excerpt of the Swing model injected from the Swing example.

Figure 11. The bootstrap process (handcrafted artefacts in gray boxes).

Figure 12. Excerpt of the Reflection metamodel used for describing Java API classes.

Figure 13. (a) An excerpt of the Swing reflective model and (b) the corresponding API metamodel discovered.

Figure 14. An excerpt of the API2MoL mapping definition discovered.

Figure 15. The development and validation tasks followed in API2MoL (dotted lines indicate that errors were found).

Figure 16. The process applied to verify the correctness of the injection and extraction processes.

Figure 17. Excerpt of the discovered metamodel from the JTwitter API.

Figure 18. API2MoL injection process applied to a Twitter account by using the JTwitter API. (a) The twitter test account (b) An excerpt of the injected model

Figure 19. API2MoL architecture.

Table Captions

Table 1. Statements describing the type of access provided by the API.

Table 2. Completeness of the API2MoL bootstrap process. Comparison between bootstrapped and real versions of both the mapping definition and the API metamodel used in several API examples