FeatureLanguage: Automatic Generation of Application Backend for Model-Based Programming Course Projects

Erica De Petrillo
Department of Electrical and Computer Engineering
McGill University
Quebec, Canada
erica.depetrillo@mail.mcgill.ca

Gunter Mussbacher
Department of Electrical and Computer Engineering
McGill University
Quebec, Canada
gunter.mussbacher@mcgill.ca

***Abstract*—University programs in software engineering or computer science increasingly include foundational courses in model-driven engineering. Building a substantial application through a term-long group project is one hands-on, practical way to learn the concepts taught in these courses. While learning by example can be very beneficial, providing students of these model-based programming courses with solutions in the form of complete working applications can be a real challenge due to time and resource constraints. In this paper, we argue that by specifying high-level requirements using our FeatureLanguage, we can completely generate the backend (i.e., Controller and Model) of a Model-View-Controller (MVC) application suitable for a university-level course. The proposed FeatureLanguage is an extension of a domain model with a specification of the different features the application should be able to accommodate as well as the constraints that need to be enforced. First, we discuss the FeatureLanguage, followed by an explanation of the different transformations from the FeatureLanguage to the backend code. We demonstrate that the complete backend can be generated and compare a generated MVC application with its handwritten counterpart. We argue that it is also feasible to completely generate a Controller test suite following a behaviour-driven development approach as well as the frontend of the MVC application, which we will explore in future work.**

***Index Terms*—Model-Driven Engineering, Domain Model, Model-Based Programming, MVC Application, Model Transformation**

I. Introduction

Model-Driven Engineering (MDE) is used for engineering numerous systems of ever-increasing complexity [25]. Software engineering or computer science university programs often require their students to take at least one model-based programming course, and because of the distinctive nature of this subject, these courses usually take a hands-on approach. It is more and more common for students registered in these courses to be expected to produce, at the end of the semester, a Model-View-Controller (MVC) application using model-driven engineering. Sometimes, these types of projects are even done in an iterative manner, giving students the opportunity to learn from their mistakes throughout the semester. To do this, course instructors must be able to provide a sample solution so that their students may learn from example. However, implementing an MVC application from scratch can be quite time-consuming, so having to do that every time the course is offered (and the project changes) is simply not viable.

We define FeatureLanguage, a domain-specific language that aims to help course instructors or teaching assistants generate entire MVC applications using the material they already typically prepare for these types of projects. While efforts exist to generate boilerplate code for the backend [3] [17] as well as for functional tests, there is currently no tool specifically for our purpose. Using only a domain model and the instructions provided to students (namely the list of constraints to be respected and the desired features), it is possible to create a FeatureLanguage model within minutes. From this, we can currently generate the entire backend of the MVC application, specifically the Model and Controller layers, without manually writing any code. This is possible thanks to novel model-to-code transformations as well as leveraging existing ones. In the future, we hope to expand this process to also generate the View layer as well as the test suite.

In the remainder of this paper, Section II presents background information crucial to the understanding of the FeatureLanguage and the transformation pipeline. Section III expands on a motivating example. Sections IV and V respectively explore the FeatureLanguage metamodel, and explain the transformation pipeline. We then touch upon related work in Section VI, before concluding the paper in Section VII.

II. Background

The heart of model-driven engineering is, as the name suggests, models. For our generation purposes, we use domain models designed with Umple [14], and their data is then processed using Acceleo transformations [1]. Below, we will define all three of these pieces of the puzzle. But first, let us take a deeper look at MVC applications [23], as these are the basis for this generation initiative.

A. MVC Applications

An MVC application is an application that is separated into three layers, namely the Model, the View, and the Controller,

```
namespace ca.mcgill.minimalresto.model;

class MinimalRestoApp {
  1 <@>- * Table tables;
  1 <@>- * Order orders;
}

class Table {
  unique immutable Integer number;
  1 <@>- 0..* Seat seats;
  enum Location { Indoors, Patio };
  Location location;
  lazy Integer maxNumberSeats;
}

class Seat {
  boolean isArmChair;
}

class Order {
   Date date;
   Time time;
   autounique number;
   * orders -- 0..1 Table tables;
}
```

Listing 1: Umple Model of MinimalRestoApp

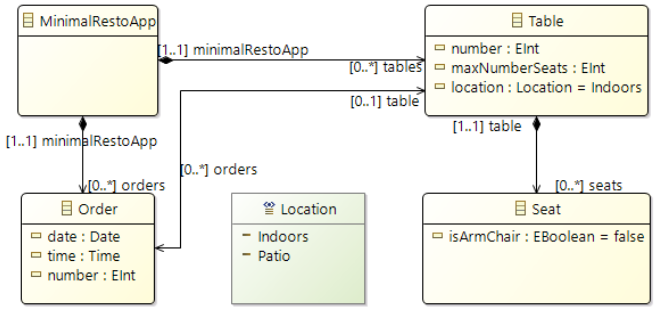

Fig. 1: Minimal Restaurant Application Domain Model. Note that, in contrast to Umple, Ecore cannot declare Table.number as unique and immutable, Order.number as autounique, and Table.maxNumberSeats as lazy.

which can be seen on the right of Figure 3. The View is the User Interface (UI) with which the user interacts. The Controller is the brain of the operation. It performs actions on the application's objects. These actions can be broken down into four categories: Create, Read, Update, and Delete, or CRUD for short. The Model contains the different classes of the objects that are used in the application. These objects are persisted in the database. The View communicates with the Controller, and the Controller communicates with the Model, but in general, the View should not have access to the Models. Thus, when the Controller must pass a Model to the View (in the case of Read actions for example), it is done through transfer objects (TO). Transfer objects are a representation of Model classes that do not expose the makeup of the Model to the View, and by extension, to the user.

B. Domain Models and Umple

Domain models are specified with class diagrams and used to populate the Model layer of an MVC application. Umple [14] is a modeling tool which simplifies model-driven development by streamlining domain model design. In Listing 1, we can see the Umple model which is equivalent to the MinimalRestoApp domain model discussed in Figure 1. We first start by defining a namespace, which corresponds to the package name for the Model layer of our MVC application (line 1). Afterwards, we describe each class (e.g., Table in line 8) in a class block. Inside each block, we list the corresponding attributes and the associations.

An attribute declaration is made up of an attribute type and an attribute name (e.g., Integer number in line 9) and usually represents a simple, single-valued piece of data. Umple has several built-in attribute types: Integer, Boolean, and String are some of the most commonly used. If we need an attribute type that does not correspond to any of the built-in ones, we may use the enum keyword, and define an enum by giving it a name and its accepted values, as can be seen for the Location attribute in Table (line 11). An attribute may be unique, which means that each instance of a class must have a different value for that attribute, (e.g., number in line 9). An attribute can also be autounique, which is essentially the same as unique, apart from the fact that the attribute is set automatically at instantiation, without any input from the user (e.g., number in line 23). These unique or autounique attributes are usually used to differentiate one instance from another. Attributes can also be lazy, which means that they are not needed in the class constructor. They are initialized to 0 for integers, false for booleans, and null for anything else [15]. An example of that is maxNumberSeats in Table, which does not need to be initialized when a Table is first created (line 13). Attributes can also be immutable (line 9), which means that once they have been set, they may not change for the rest of the instance's lifecycle.

An association declaration is made up of the multiplicity of the class it is declared in, followed optionally by that class's role name in that association. Different symbols are used for bidirectional associations ($--$), unidirectional associations ($->$), and compositions ($<@>-$)). Bidirectional associations are for when both classes are aware of each other, unidirectional associations are for when only one class is aware of the other, and compositions are for when a class is contained by another. See line 10 for an example of composition between Table and Seat and line 24 for an example of bidirectional association between Table and Order.

Note that the example does not include generalization as this concept is left for future work on FeatureLanguage.

C. Model Transformations

Model transformations are used for a specific purpose, take in input models, and create output models, based on a specification [22]. Model transformations are at the core of initiatives like the Model-Driven Architecture (MDA) [8]. We may transform a model into another model during a model-to-model transformation with technologies like ATL [2]. We may instead transform a model into text during a model-to-text transformation with technologies like Acceleo [1]. This is essentially what happens when we generate a Controller class from our FeatureLanguage model. Indeed, we use the information provided in the FeatureLanguage model to fill in a

template that creates a Java file for the Controller. This is done using Acceleo, which leverages OCL [10] queries to allow us to retrieve the information we need from the FeatureLanguage model.

III. Motivating Example

Consider a group project for an introductory class on model-driven engineering, typically taken during the first or second year of a university program in Software Engineering, Computer Engineering, or Computer Science. Students would be required to create an MVC application for some sort of management software, like a reservation or asset management system for example. The instructor would provide a description of the problem domain, a list of features the solution should be able to fulfill, as well as the constraints posed on some domain concepts. The instructor would also provide Gherkin scenarios [6] to facilitate the testing of the application (more specifically the Controller code). The skeleton code generated from the Gherkin scenarios by Cucumber [4] would be provided to the students as well. Students would be expected to complete this project throughout the term, in groups of about six people. Overall, they would be required to complete around 16 features, like adding or updating an asset. The project would also require the implementation of about 25 constraints, and about 150 unit tests derived from several Gherkin scenarios.

Currently, the implementation of the project is minimally automated, and most steps of the project are done manually. Students start by creating a domain model with Umple [14], which will allow them to generate the Model code. Then, they implement step definitions for the provided Gherkin scenarios and constraints, followed by the Java implementation of the Controller to pass all Gherkin scenarios. Afterwards, they can move on towards designing the View. This is done through specifying a layout with SceneBuilder, which in turn generates layout code in XML format, and manually implementing the JavaFX [7] Controller. Lastly, even though a sample persistence layer is provided, students are still required to adapt it to their project in order to save the data to JSON files.

While the importance of doing this hands-on work cannot be neglected for students, being able to automate this work could be beneficial, for instructors and students alike. For example, an instructor teaching this kind of course could be greatly aided by an application that could automatically generate the solution to this management software project. Being able to automate the creation of Gherkin scenarios would significantly reduce the effort required to invent a new problem description for each course offering. Furthermore, students would be provided a more complete and thorough solution from which they would be able to learn. Using the instructions provided for the project, namely the features and constraints, as well as a domain model, there is the possibility of automating not only the generation of the Model (as is currently the case) but also the full Controller implementation. While generating the Gherkin scenarios, the full implementation of the step definitions, and the UI could also be achieved, this is outside the scope of this paper and left for future work.

To motivate the proposed FeatureLanguage, let us look at a simple restaurant management application. The MinimalRestoApp system showcases all key modeling features used by students of model-based programming classes, except generalization: classes, attributes with different types, enumerations, as well as associations and compositions with different multiplicities. Furthermore, attributes may be designated as lazy and instances may be identified by a unique or autounique attribute or by a list index. The domain model of the MinimalRestoApp in Figure 1 contains four classes, namely Table, Seat, Order, and the MinimalRestoApp root class. The attributes of the Table and Order classes have the following six constraints:

- The Table's reference number must be greater than 0: Table.number > 0
- The Table's reference number must be unique
- The Table's reference number must be immutable
- The Table's maxNumberSeats must be lazy
- The Table's location must be either "Indoors" or "Patio"
- The Order's reference number must be autounique

The first constraint would be provided by the instructor in the problem description. The fifth constraint refers to the enumeration definition. The remaining constraints correspond to the Umple unique, immutable, lazy, and autounique keywords, respectively. Hence, the last five constraints are derived from the Umple domain model.

While Table instances and Order instances are uniquely identified by their number attributes and the root class MinimalRestoApp is a singleton by definition and does not require an identifying attribute, Seat instances are identified by their index in their Table's list. While Umple allows the specification of identifying attributes, it does not support the explicit designation of a list index for the purpose of uniquely identifying instances in the list.

For each of the three non-root classes, we can Add, Remove, Display, or Update instances. It is worth mentioning that in the case of the Seat class, these manipulations must be done pertaining to a particular Table instance. Let us look in greater detail at these four types of features.

A. Add

Assume we want our MinimalRestoApp to offer the Add Table feature. To fulfill it, the addTable method must be added to the Controller. The details of the implementation of such method can be seen in Listing 2. Notice that the parameters are only made up of non-lazy and non-autounique attributes. Furthermore, no multi-valued associations are present in the parameters. We first retrieve the MinimalRestoApp root object from the MinimalRestoAppApplication[1] class, as all other objects are contained in it directly or indirectly (line 3).

[1]The implementation of the Controller assumes the existence of two helper classes: (1) the MinimalRestoAppApplication class to start the application and hold the root object, and (2) the MinimalRestoAppPersistence class to save application data. These are almost fully generic classes that are more easily generated than the Controller class and hence not the focus of this paper.

```
public static String addTable(int number, String location) {

    MinimalRestoApp root = MinimalRestoAppApplication.
          getMinimalRestoApp();

    if (!(number > 0)) {
        return "The table number must be greater than 0.";
    }

    Location parsedLocation;
    try {
        parsedLocation = Location.valueOf(location);
    }
    catch (Exception e) {
        return "The table location must be Indoors or Patio.";
    }

    try {
        new Table(number, parsedLocation, root);
    }
    catch (RuntimeException e) {
        return "The table number must be unique.";
    }

    try {
        MinimalRestoAppPersistence.save();
    }
    catch (RuntimeException e) {
        return e.getMessage();
    }

    return "";
}
```

```
public static String removeTable(int number) {

    try {
        Table table = Table.getWithNumber(number);
        table.delete();
    }
    catch (RuntimeException e) {
        return e.getMessage();
    }

    try {
        MinimalRestoAppPersistence.save();
    }
    catch (RuntimeException e) {
        return e.getMessage();
    }

    return "";
}
```

```
public static TOTable getTable(int number) {

    if (Table.hasWithNumber(number)) {
        return convertToTOTable(Table.getWithNumber(number));
    }

    return null;
}
```

Listing 2: addTable (left), removeTable (top right), and displayTable (bottom right) Controller Methods

Then, we validate that all parameters fulfill their respective constraints (lines 5-15). This may require input to be cast to the correct type (e.g., in the case of enumerations) (line 11). We then attempt to create the Table object by calling the Model code generated by Umple (line 18). If the call succeeds, we persist the change (line 25). At any point, if an error arises, we return it in the form of a String.

B. Remove

Following the same logic, we likely wish to be able to Remove a Table as well. The removeTable method, which is illustrated in Listing 2, first starts by attempting to find the desired Table instance (line 4), given the parameter and using a method generated by Umple. If it is successful, it then deletes said instance (line 5) before persisting the changes (line 12). As with addTable, if at any point an error arises, it is returned as a String.

C. Display

Manipulating Table instances is absolutely useless if we cannot access them. Therefore, we will also need to Display a Table. The getTable method can be examined in Listing 2. Notice that unlike the previous two methods, we do not return a String but rather a transfer object for Table, or TOTable for short. We start by checking if a Table with the desired reference number (i.e., the parameter) exists (line 3). If it does, we retrieve this Table and convert it to a TOTable object (line 4). If it does not, we return null. Due to space constraints, the body of the converter method and the specification of the transfer object are not shown. A transfer object is an immutable object that only contains the information needed by the UI. It does not expose classes from the generated Model package to the UI – as returning a Table object would – to enforce a strict interpretation of the MVC pattern. In the course project, Umple is used to specify each transfer object.

D. Update

Last but not least, being able to Update a Table is needed to change some attribute values over time. The updateTable method resembles a lot the addTable method. First, the parameter list is similar, except that an additional parameter is needed for the identifying attribute of Table (i.e., number) – one for the existing value and one for the new value, that lazy attributes are included, and that immutable attributes are not included. Second, they both return a String in the event that an error is thrown, and third, both start by checking for parameter constraints and typecasting to the correct type in the case of enumerations. However, before performing the update, we must retrieve the Table instance with the desired reference number, much like in the removeTable method. Then, we can update all necessary parameters by calling the corresponding setter methods generated by Umple, and persist the changes as in the addTable and removeTable methods.

This paper will show that by using the already-provided features and constraints, as well as by creating an Umple domain model, it will be possible to generate the contents of the Controller, including the converter methods as well as the Umple specification for the transfer objects.

IV. Metamodel

The desired generation is achieved through use of FeatureLanguage, a textual representation of a traditional domain model supplemented with additional information. The details of the FeatureLanguage metamodel can be viewed in Figure 2. The root class, FeatureLanguage, and its associations indicate that FeatureLanguage models are separated into five main sections, namely Concepts, Constraints, Keys, Properties, and Features. The following paragraphs will explain the different metaclasses in greater detail.

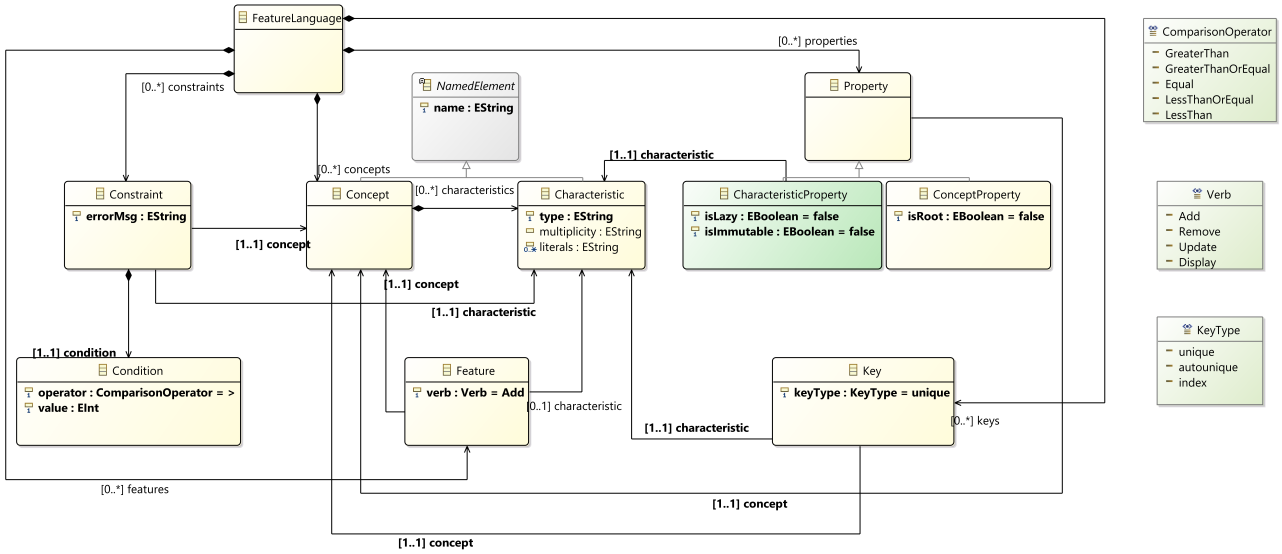

Fig. 2: FeatureLanguage Metamodel

A. Concepts and Characteristics

Concepts and Characteristics exist to make a domain model available in FeatureLanguage, and hence, they are considered read-only. A Concept in FeatureLanguage is equivalent to a class in a domain model (e.g., Table). Each Concept has an identifying name and a set of Characteristics. Characteristics are used to represent attributes (e.g., number, maxNumberSeats, location) and associations (e.g., seats, orders). This design choice was made in order to simplify the Controller generation. Since attributes as well as associations can be parameters in a class's Controller methods, modeling both as Characteristics made for easier data accessing. When representing a simple attribute, a Characteristic has a type (e.g., EInt) and an identifying name (e.g., number). Note that type is not a class in FeatureLanguage as for our purposes, only the type name is necessary for generation, so forgoing giving it its own class simplified the language model. If that attribute's type happens to be an enumeration, then the Characteristic also contains a list of all possible literals in the enumeration (e.g., Indoors and Patio). If the Characteristic represents an association though, it has a multiplicity instead (e.g., 0..*). All classes, attributes, enumerations, and relationships from a domain model are represented in the Concepts section of the FeatureLanguage model. It is assumed that the domain model is valid (i.e., it is a well-formed class diagram). Hence, the aforementioned simplifications are possible. Similarly, multiplicity can be simplified into a String instead of modeling separately lower bound and upper bound.

The remaining classes covering Constraints, Keys, Properties, and Features describe how the domain model is extended with the information required to generate the Controller, and eventually the Gherkin scenarios, step definitions, as well as UI. Note that some Keys and Properties could be derived from an Umple domain model and hence included in the definition of Concepts and Characteristics. However, this is not done so that the FeatureLanguage can also be used together with a domain modeling tool that does not support them.

B. Constraints

A Constraint in a FeatureLanguage model specifies an additional Condition that might apply to some Characteristics of a Concept: for example, a numerical ID such as a Table number that must be greater than 0. Each Constraint applies to one of a Concept's Characteristics, and contains one Condition and one error message. The error message is that which the Controller should return in the event the Constraint is not respected. The Condition is made up of an operator and a numerical value. Note that in the future, we envision to extend this model to support arbitrary OCL constraints by leveraging OCLinEcore's capabilities instead of the basic comparisons that are currently supported.

C. Keys

In a domain model, all classes but the root class need an identifier to allow a user to tell apart their different instances. This identifying value is represented as a Key in FeatureLanguage, and all Keys are listed in the Keys section. There are three types of Keys: unique, autounique, and index.

Unique and autounique Keys are for Concepts that are identified by one of their Characteristics, which is either set by the creator of the instance or automatically generated, respectively. In the example in Figure 1, the number of a Table is declared as unique and the number of an Order as autounique.

The index KeyType is meant for Concepts that do not have an identifying Characteristic of their own, but are rather identified through their position in another Concept's list (e.g., seats of a Table).

D. Properties

The Properties section contains two types of Properties, namely Concept Properties and Characteristic Properties. A Concept Property is used to declare which Concept is to be treated as the root of this system (e.g., MinimalRestoApp). This is the only Concept that will not have a Key, and it is assumed to be a singleton. Characteristic Properties are used to declare if one of a Concept's Characteristics is lazy (e.g., maxNumberSeats) or immutable (e.g., a Table's number).

E. Features

The last section contains a list of all Features that should be executable in the system. There are four types of Features: Add, Remove, Display, and Update. Add features are for creating instances. We may create instances of Concepts that are not related to any other Concept (apart from the root Concept), like creating a Table, but we may also create instances of Concepts that are indeed related to other Concepts, like adding a Seat to a Table. Remove features are essentially the opposite of add features, deleting instances of Concepts that are or are not related to other Concepts. Display features retrieve information from the application, and can also be applied to both Concepts and Characteristics. Update features can be used to either modify an entire Concept or to change the value of one of its Characteristics.

F. FL Model

FeatureLanguage (FL) models use the .fl extension and follow minimal formatting. The FL model for the MinimalRestoApp motivating example can be seen in Listing 3.

Each Concept is defined using the concept keyword followed by its name (e.g., concept Table on line 5). All Characteristics are listed below, in an indented fashion (e.g., int number on line 6 specifies an attribute). For Characteristics that represent associations, the name is derived from the role name in the domain model, and the multiplicity is represented using the same convention as domain models (e.g., Seat seats 0..* on line 9). For Characteristics that represent enumeration attributes, literals are listed within curly braces and space-separated (e.g., Location location { Indoors Patio } on line 11).

Then, each section is delimited using its associated keyword. In the Constraints section of this example, the Constraint placed on Table's number is shown, alongside the error message that would be displayed in the event that Table's number would be less than 0 (line 25).

The Keys section specifies the three types of Keys used in MinimalRestoApp. The specification of a Key requires a Concept along with its Characteristic, next to either the unique, autounique, or index keyword. For example, Table.number unique on line 28 and Order.number autounique on line 30 indicate that a Table and an Order are identified by their respective numbers, while Table.seats index on line 29 signals that an individual Seat can be accessed through its place in the list of the Table to which it belongs.

```
concept MinimalRestoApp                                                    1
  Table tables 0..*                                                        2
  Order orders 0..*                                                        3
                                                                           4
concept Table                                                              5
  int number                                                               6
  int maxNumberSeats                                                       7
  MinimalRestoApp minimalRestoApp 1..1                                     8
  Seat seats 0..*                                                          9
  Order orders 0..*                                                        10
  Location location { Indoors Patio }                                      11
                                                                           12
concept Seat                                                               13
  boolean isArmChair                                                       14
  Table table 1..1                                                         15
                                                                           16
concept Order                                                              17
  Date date                                                                18
  Time time                                                                19
  int number                                                               20
  MinimalRestoApp minimalRestoApp 1..1                                     21
  Table table 0..1                                                         22
                                                                           23
constraints                                                                24
  Table.number > 0 "The table number must be greater than 0"               25
                                                                           26
keys                                                                       27
  Table.number unique                                                      28
  Table.seats index                                                        29
  Order.number autounique                                                  30
                                                                           31
properties                                                                 32
  MinimalRestoApp root                                                     33
  Table.number immutable                                                   34
  Table.maxNumberSeats lazy                                                35
                                                                           36
features                                                                   37
  Add Table                                                                38
  Remove Table                                                             39
  Display Table                                                            40
  Update Table                                                             41
  Update Table.location                                                    42
  Add Table.seats                                                          43
  Remove Table.seats                                                       44
  Display Table.seats                                                      45
  Update Table.seats                                                       46
  Add Order                                                                47
  Remove Order                                                             48
  Display Order                                                            49
  Update Order                                                             50
```

Listing 3: FL Model for Minimal Restaurant Application

There are two types of Properties, those for Concepts and those for Characteristics. The Concept Property lets us declare exactly one Concept as the root, and is mandatory. In this example, MinimalRestoApp is declared as the root of the system in the Concept Property (line 33). A Characteristic Property lets us declare a Characteristic as immutable or lazy (e.g., Table.number immutable on line 34 and Table.maxNumberSeats lazy on line 35).

Lastly, we can see the Features section, which lists all the actions we should be able to perform in MinimalRestoApp. Some features act upon a Concept (like Add Table on line 38). Others act upon a Characteristic, be it an association Characteristic (like Add Table.seats on line 43) or an attribute Characteristic (like Update Table.location on line 42).

V. Transformation

To achieve the desired generation, our data will need to go through multiple states and undergo several transformations. An overview of this process can be seen in Figure 3. The blue dashed arrows represent new transformations discussed in this paper, while the black solid arrows represent third-party transformations that already existed. Arrows without any text represent handwritten work, like transcribing keys to the FeatureLanguage model for example. The blue dashed boxes represent the files created by the proposed system, namely the partially generated FeatureLanguage model as well as the fully

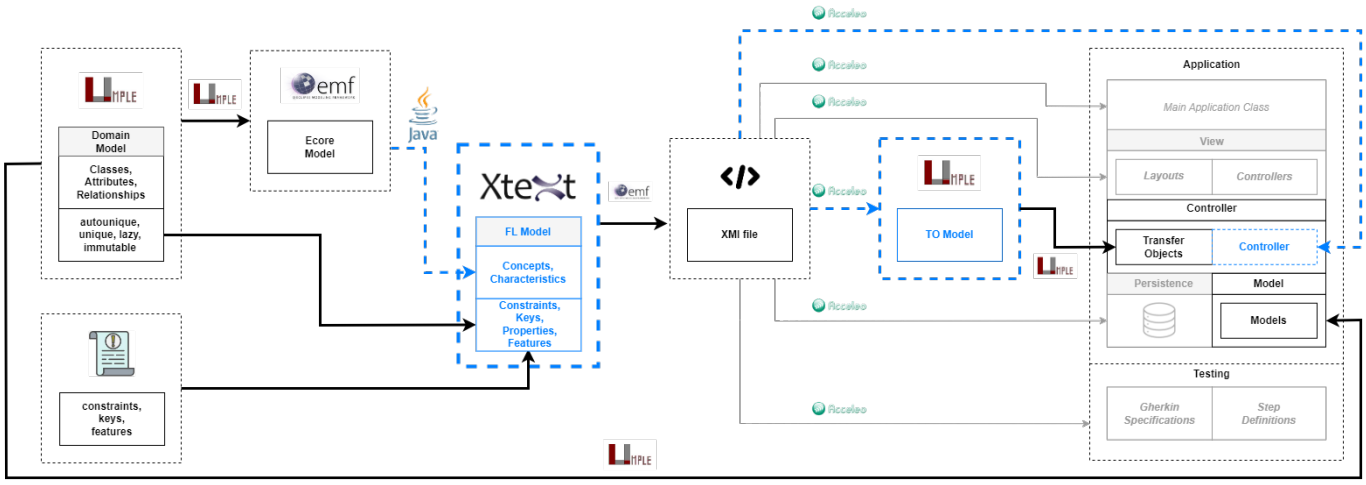

Fig. 3: Transformation Pipeline

generated Umple transfer objects model and the Controller class. All greyed out elements represent future work.

We start with an Umple domain model – the same tool many model-based programming students are expected to use, which is transformed into an Ecore model using Umple's generation feature. Through a Java extraction algorithm, we take the data we need from the Ecore model and repackage it to fit the FeatureLanguage text format for Concepts and Characteristics. Then, the Constraints, Keys, Properties, and Features sections are filled in manually by the instructor (some need to be copied from Umple). Note that, instead of opting for a more compact representation, we decided to clearly separate the different sections to allow for iterative modification of the domain model through the extraction algorithm without overwriting the other sections. It is also possible to begin the generation process from the Ecore model or the FeatureLanguage file if desired. Once the FeatureLanguage model is complete, it is transformed into an XMI file, using an EMF-based text-to-XMI converter algorithm. Now that our data is in this format, it is ready to go through its last transformation, which will result in the generated Controller and the Umple transfer objects model. Using Acceleo, we create templates to automate the implementation of the Controller. Note that the choice of transformation technology was arbitrary, and that any other tool that allows model-to-model transformations could have been used. We also use templates to generate the Umple model for the transfer objects, which in turn will be used to generate the classes for the transfer objects. Similar Acceleo transformations will be used in the future to generate the View, Persistence, and Testing layers of the application, as well as the main application class.

A. Controller

First, we start with the file header, which loosely refers to package declarations, imports, and class names. We get the package name from the root Concept's name. For the imports, we first import the Application and Persistence classes whose names we also derive from the root Concept's name, then we cycle through all the Concepts and import the entire corresponding model for each Concept. It is important to mention that this might result in creating more imports than necessary, but it is an acceptable trade-off considering the time and resource savings generation can provide. In this header section, we also declare the Controller class name and its constructor, deriving them again from the root Concept's name.

Next, we move on to generating all the methods in the Controller. Each feature specified in the FeatureLanguage model must each have at least one corresponding method in the Controller. Moreover, converter methods from Concepts to their transfer object counterparts must also be implemented. Below, we will describe the transformations necessary to do so for each of the four Feature types.

1) Add Methods: There are two types of Add methods: Add Concept (like Add Table) and Add Concept.Characteristic where Characteristic is an association (like Add Table.seats), the latter being a variation on the first. We will therefore go into detail using the Add Table example. Figure 4 provides a schematic representation of the transformation from the FeatureLanguage model to the outputted code, showing the output in the third column and the corresponding source elements from the FL model and the FL metamodel in the second and first column, respectively. We first generate the name of the method using the Concept name (line 2). We also list all the relevant Characteristics as parameters, namely single-valued Characteristics that are neither autounique, lazy, nor refer to the root (lines 2-6). We can determine which Characteristics correspond to those criteria by using Keys and Properties. In this case, we only use number and location as parameters, since maxNumberSeats is lazy and all other Characteristics are multi-valued associations or refer to the root. The type of each parameter matches that of the Characteristic, except for Enumerations, where the Enumeration type (Location in this case) is replaced by String. Then, we get the root,

	FL Metamodel	FL Model	Controller
1			
2	Feature	Add Table	`public static String addTable(int number, String location) {`
3	Concept, Characteristic	concept Table (and respective characteristics)	
4	Key	Table.number unique	
5	ConceptProperty	MinimalRestoApp root	
6	CharacteristicProperty	Table.maxNumberSeats lazy	
7	ConceptProperty	MinimalRestoApp root	`  MinimalRestoApp root = MinimalRestoAppApplication.getMinimalRestoApp();`
8	Constraint, Condition	Table.number > 0 "The table number..."	`  if (!(number > 0)) {`
9	Characteristic	int number	`    return "The table number must be greater than 0";`
10			`  }`
11	Characteristic	Location location { Indoors Patio }	`  Location parsedLocation;`
12			`  try {`
13			`    parsedLocation = Location.valueOf(location);`
14			`  }`
15			`  catch (Exception e) {`
16			`    return "The table location must be Indoors or Patio.";`
17			`  }`
18			`  try {`
19	Feature	Add Table	`    new Table(number, parsedLocation, root);`
20	Concept, Characteristic	concept Table (and respective characteristics)	
21	Key	Table.number unique	
22	ConceptProperty	MinimalRestoApp root	
23	CharacteristicProperty	Table.maxNumberSeats lazy	
24			`  }`
25			`  catch (RuntimeException e) {`
26	Feature	Add Table	`    return "The table number must be unique.";`
27	Key	Table.number unique	`  }`
28			`  try {`
29	ConceptProperty	MinimalRestoApp root	`    MinimalRestoAppPersistence.save();`
30			`  }`
31			`  catch (RuntimeException e) {`
32			`    return e.getMessage();`
33			`  }`
34			`  return "";`
35			`}`

Fig. 4: addTable Transformation

MinimalRestoApp, which is done using the ConceptProperty which defines the root (line 7). If any of the Characteristics listed in the parameters have any Constraints attached to them, we verify that they are respected, and return an error if they are not. This is done by finding the Constraints that include the Concept we are trying to add (lines 8-10). In the case where one of the parameters should be typecast to an enumeration type, we implement a try/catch block to parse said parameter. In this example, we must parse the String location into a Location parsedLocation (lines 11-17). Now that all the preparation work is done, we can implement the main try/catch block, the one where we attempt to create the Table object. The list of parameters used in this constructor call includes the root, as well as the parameters in the AddTable method signature, with the exception of enumeration Characteristics that underwent parsing (lines 18-24). If something goes wrong, we return an error String: either the exception message or a specific message about the Concept Key if it is unique, as in this case with Table number (line 26). We finish off with the persistence block, which only requires using a ConceptProperty to find the root Concept, in this case MinimalRestoApp (line 29).

As previously mentioned, Add Concept.Characteristic methods follow a very similar logic. Notable differences include needing the Concept key in the parameters of the addCharacteristicToConcept method signature, and needing to retrieve the Concept object, as well as adding the Characteristic object to it. For example, for the method addSeatToTable, we would need the Table number to be part of the parameters. Then, we would need to retrieve that specific Table, create the new Seat, then add the Seat to that Table.

2) Remove Methods: Remove methods are noticeably simpler than Add methods. For Remove Concept methods, the parameters only include the Concept Key. For example, the method removeTable would only have the parameter int number. Remove Concept.Characteristic (where Characteristic is an association) methods work very similarly but need both the Concept and Characteristic Keys: the method removeSeatFromTable would only need the int number key for Table and the int index key for Seat. Then, there are only two try/catch blocks. In the first block, we attempt to find the Concept or Characteristic instance we want to remove. The values for this call are derived from the Concept or Characteristic and their respective Keys. For example, we could find a Table with the provided number, or find a Seat, with both the provided Table number and index. If the instance is found, we delete it. All thrown exceptions are returned in the form of a String. The second block commits the change to the persistence layer, and follows the exact same pattern as that of Add methods.

3) Display Methods: There are three types of Display methods. First, we can Display Concept, like Display Table. This is the most straightforward type, and the only required parameter is the Concept's Key, in this case the Table's number. The other two types of Display methods concern Display Concept.Characteristic features. All Display

Concept.Characteristic features will have a displayCharacteristicFromConcept method, where Characteristic is an association. For example, we could displaySeatFromTable, which would take a Table number as parameter, as well as a Seat index. However, for Characteristics that have an upper bound multiplicity greater than 1, we may also have a displayCharacteristicListFromConcept method. We know that a Table can have multiple Seats, so we will generate a displaySeatsFromTable method, with only the Table number as a parameter, in order to display all Seats at once. In all cases, we first start by trying to retrieve the Concept that corresponds to the supplied Key, much like the two previous method types, so we would look for the Table with the provided number. If none is found, then we just return null. If we were looking to display the entire Concept, then we return the converted form of the instance, as the View should not have access to Model types. For displayTable, we would therefore return TOTable. The data necessary for this call is retrieved through the Concept and its Key. If we were instead trying to display a Characteristic, we would then check if the Characteristic exists in the Concept instance. For example, we would look for a particular Seat in the found Table, and convert it to TOSeat if found. If we were trying to display a list of Characteristics, like all the Seats of a Table, we would simply return a list of TOSeats.

4) Update Methods: There are also three types of Update methods. First, we have methods for Update Concept features where the Concept has a unique or autounique Key. For example, Update Table would require as a parameter the Table number to access the Table we are trying to update, as well as all other single-valued associations, non-immutable, and non-autounique Characteristics that do not refer to the root. These parameters follow the same logic as those of the Add methods, so enumeration types are converted to Strings. Second, we have methods for Update Concept.Characteristic, where the Characteristic is not an association. For example, we could have a method to only update the location of a Table. In this case, we only need the Table number and the new value for the Characteristic as parameters. Note, however, that these Update methods modifying only one Characteristic will also be named updateConcept (e.g. updateTable in this case), and will leverage overloading. Third, we have methods for Update Concept.Characteristic, where the Characteristic is a Concept with an index Key. This is the type of method we would need to Update Table.seats. The required parameters would then be the Table number of the Table the Seat belongs to, the index Key of the Seat we are trying to update, as well as all other single-valued association, non-immutable, and non-autounique Seat Characteristics. The body of Update methods resembles greatly that of Add methods, as we first must check for Constraints, and then we typecast if needed. We then attempt to retrieve the Concept instance, as in the Remove methods, and then set the Characteristics corresponding to each parameter. Finally, we persist the changes.

5) Converter Methods: In the Display methods, converting from a Concept to its transfer object was touched upon. Indeed, these conversion methods must be generated as well. For each Concept, we return a new TOConcept using the TOConcept constructor. Each parameter in that constructor corresponds to a Characteristic. For non-association Characteristics, we simply use a getter to retrieve them. For association Characteristics, we also use a getter, but we convert the result using its own converter method. For example, if we were to convertToTOTable, we could use table.getNumber() to get its number, and convertToTOSeat(table.getSeats) to get a list of TOSeats.

B. Transfer Objects Model

While the Controller file constitutes the bulk of the generation, it is not very useful without the transfer objects for all Concepts. Instead of generating the classes for the transfer objects ourselves, we instead generate the Umple model which can then be used to generate the classes. The namespace is found using the root Concept, in this case MinimalRestoApp. For each Concept, we create a corresponding class with all its Characteristics. For non-association Characteristics, they are rewritten verbatim, apart for some slight changes in types. For example, the TOTable class would have Integer number as an attribute, instead of int number, and String location instead of Location location. For association Characteristics, they follow the regular Umple syntax, but use directed associations only (e.g., * –> TOSeat 0..* seats). The transfer objects currently provide all the information in the domain model but are expected to only provide the needed information in the future when the Display method, which makes use of the transfer objects, will be more customizable.

VI. Related Works

Currently, generating the Model layer of an MVC application is quite standard, as many different frameworks offer that possibility. Indeed, using Umple [14], Eclipse Modeling Framework (EMF) [5], or PyEcore [11], it is possible for users to completely generate an entire set of Model classes. Frameworks like Spring [12] or Java Enterprise Beans [16] can also be used to accelerate development time when implementing MVC applications, as they can help reduce the amount of boilerplate code one must write.

For our purposes, we would need a tool to generate an entire Controller layer from an Umple domain model, which does not exist as of now. However, that is not to say efforts to generate controllers do not exist, like JHipster [17] which generates full web applications but does require some customization of the generated code for business logic. EMF on Rails [21] is an example for the generation of web applications based on Ecore models, resulting in skeleton code for Spring Roo [13] from which a full web application can be generated. We will discuss two other examples in greater detail below. Do note that Controller generation using artificial intelligence is out of scope for this paper though.

Entity Framework [18] is an object-relation mapping framework used within the .NET framework. Amongst its many features, Entity Framework, leveraging its scaffolding engine, allows the generation of boilerplate CRUD controller code

by creating a data model (similar to a domain model) and a database context [19]. While the Controller is functional, it is quite rudimentary: additional business logic and error management must be added in by hand [20].

NestJS is a framework to help create NodeJS server-side applications. It aims to speed up the development process by providing resources to automate repetitive programming [3]. It can create controllers, services, and transfer objects, but this technique still requires business logic to be implemented by hand. However, using the NestJS CRUD microframework [9], it is instead possible to generate a fully working CRUD controller and service for an entity. While the process is very short and hence time saving, it is not a good alternative for teaching purposes as we cannot see any logic in the code, so it would be hard for students to learn from example.

To ensure a more thorough analysis of the current research landscape, we examined the MoDRE Workshop papers from the last 12 years as well as the available Educators' Symposium (EduSymp) papers of the MODELS conferences of the last 19 years. In over 100 MoDRE papers and about 80 EduSymp papers, we found no attempt at Controller generation. We did, however, find a handful of related generation efforts. Most notably, we found research on generating functional tests from Business Process Model and Notation (BPMN) diagrams [24]. From these BPMN diagrams, the authors generated Gherkin scenarios for functional test cases, which is akin to our future vision of generating unit test cases through Gherkin scenarios from our FeatureLanguage.

VII. Conclusion

Learning by example can be a very effective strategy, especially when it comes to model-driven engineering. While courses focusing on this subject often require students to implement an MVC application themselves, it can be very difficult to provide full sample solutions due to time and resource constraints.

Using FeatureLanguage, course instructors will be able to generate the backend of an MVC application with minimal time and resource investment. Using a domain model, as well as the project constraints and features, it will be possible to define a FeatureLanguage model in minutes. Afterwards, the transformations described in this paper will take care of outputting the Controller layer. It is evident that generated code cannot be as elegant as handwritten code (e.g., unnecessary import statements, multiple try/catch blocks instead of one). However, this is a minor downside considering the huge gain in productivity and efficiency that such an approach could have. Moreover, the goal of a sample solution is to show how model-based programming works, rather than to teach students how to write concise code. Therefore, the benefits instructors and students alike could experience from using FeatureLanguage for sample solution generation remain unchanged.

In the future, we hope to expand FeatureLanguage by supporting generalization. We also look forward to creating a transformation to generate the View of an MVC application as well as the testing suite composed of Gherkin scenarios and their corresponding implementations. This will make it easier for instructors to create more complete project descriptions that include Gherkin scenarios. In order to further validate our code generation, we plan on implementing by hand some MVC applications (e.g., the course projects from the last years) and comparing them with their generated counterparts. This way, we will confirm that a set of sample solutions suitable for University courses can be generated using FeatureLanguage, saving time and resources for instructors, and providing a more thorough learning experience for students.

While it is possible to expand the scope of FeatureLanguage to non-educational projects in the future, significant changes may have to be made due to assumptions about project complexity and required support of non-functional requirements such as security and performance.

References

[1] Acceleo. Available at https://eclipse.dev/acceleo/ (2024/31/03).

[2] ATL. Available at https://eclipse.dev/atl/ (2024/04/04).

[3] CRUD generator (TypeScript only). Available at https://docs.nestjs.com/recipes/crud-generator (2024/22/03).

[4] Cucumber. Available at https://cucumber.io/tools/cucumber-open/ (2024/21/03).

[5] Eclipse Modeling Framework (EMF). Available at https://eclipse.dev/modeling/emf/ (2024/27/03).

[6] Gherkin Syntax. Available at https://cucumber.io/docs/gherkin/ (2024/23/03).

[7] JavaFX. Available at https://openjfx.io/ (2024/21/03).

[8] MDA - The Architecture of Choice for a Changing World. Available at https://www.omg.org/mda/ (2024/03/04).

[9] NestJs CRUD. Available at https://github.com/nestjsx/crud (2024/22/03).

[10] Object Constraint Language. Available at https://www.omg.org/spec/OCL/2.4/PDF (2024/04/04).

[11] PyEcore Documentation. Available at https://pyecore.readthedocs.io/en/latest/ (2024/27/03).

[12] Spring Boot. Available at https://spring.io/projects/spring-boot (2024/27/03).

[13] Spring Roo. Available at https://github.com/spring-attic/spring-roo (2024/05/05).

[14] Umple. Available at https://cruise.umple.org/umple/ (2024/21/03).

[15] Umple Attribute Definition. Available at https://cruise.umple.org/umple/AttributeDefinition.html (2024/22/03).

[16] What Is an Enterprise Bean? Available at https://docs.oracle.com/javaee/6/tutorial/doc/gipmb.html (2024/27/03).

[17] What Is JHipster? Available at https://www.jhipster.tech/ (2024/03/04).

[18] R. Anderson et al. Entity Framework. Available at https://learn.microsoft.com/en-us/aspnet/entity-framework (2024/21/03).

[19] T. Dykstra et al. Tutorial: Get started with EF Core in an ASP.NET MVC web app. Available at https://learn.microsoft.com/en-us/aspnet/core/data/ef-mvc/intro?view=aspnetcore-8.0 (2024/21/03).

[20] T. Dykstra et al. Tutorial: Implement CRUD Functionality - ASP.NET MVC with EF Core. Available at https://learn.microsoft.com/en-us/aspnet/core/data/ef-mvc/crud?view=aspnetcore-8.0 (2024/21/03).

[21] R. López-Landa., J. Noguez., E. Guerra., and J. de Lara. Emf on rails. In *Proceedings of the 7th International Conference on Software Paradigm Trends - ICSOFT*, pages 273–278. INSTICC, SciTePress, 2012.

[22] L. Lúcio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G. Selim, E. Syriani, and M. Wimmer. Model Transformation Intents and Their Properties. *Software & Systems Modeling*, 15, 07 2014.

[23] S. Smith. Overview of ASP.NET Core MVC. Available at https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-8.0&WT.mc_id=dotnet-35129-website (2024/31/03).

[24] P. von Olberg and L. Strey. Approach to generating functional test cases from bpmn process diagrams. In *2022 IEEE 30th International Requirements Engineering Conference Workshops (REW)*, pages 185–189, 2022.

[25] J. Whittle, J. Hutchinson, and M. Rouncefield. The State of Practice in Model-Driven Engineering. *IEEE Software*, 31(3):79–85, 2014.