ABSTRACT

INTUITIVE MODEL TRANSFORMATIONS:
A GUIDED FRAMEWORK FOR STRUCTURAL MODELING

by Nicholas John DiGennaro

As research in Model-Driven Software Engineering (MDSE) continues to innovate the software engineering process, a gap has been created between its potential benefit and actual use. This gap exists due to MDSE education being held back by tools that are not well-suited for learning and use by beginners. To teach new developers MDSE concepts before moving into industrial tools, MDSE educators need a lightweight, model-first, end-to-end teaching tool. The Instructional Modeling Language (IML) is proposed as a solution to fit these needs. This thesis implements for the IML Framework one of the foundational aspects of MDSE: Model Transformations. The ability to apply model transformations to large collections of models contributes to software reuse, allowing developers to dedicate efforts elsewhere. Furthermore, the early work of conceptually modeling systems has a larger payoff as a result of model transformations being used in the code generation process. These benefits are some of the most influential factors for adopting MDSE and must be properly emphasized in education. Model transformations in IML provide students with a guided experience and clear explanations of the model transformation processes. In turn, students are provided with a solid foundation in MDSE and its benefits.

# INTUITIVE MODEL TRANSFORMATIONS:
# A GUIDED FRAMEWORK FOR STRUCTURAL MODELING

A Thesis

Submitted to the
Faculty of Miami University
in partial fulfillment of
the requirements for the degree of
Master of Science
by

Nicholas John DiGennaro
Miami University
Oxford, Ohio

2021

Advisor: Eric J. Rapos, PhD
Reader: Matthew Stephan, PhD
Reader: Hakam W. Alomari, PhD

This Thesis titled

INTUITIVE MODEL TRANSFORMATIONS:
A GUIDED FRAMEWORK FOR STRUCTURAL MODELING

by

Nicholas John DiGennaro

has been approved for publication by

The College of Engineering and Computing

and

The Department of Computer Science & Software Engineering

---

Eric J. Rapos, PhD

---

Matthew Stephan, PhD

---

Hakam W. Alomari, PhD

# Table of Contents

# List of Tables

# List of Figures

# Acknowledgements

I would first like to thank my thesis advisor Assistant Professor Eric Rapos of the Department of Computer Science and Software Engineering at Miami University. His passion and enthusiasm for this project was contagious. Working with someone as committed as he is motivated my efforts throughout this thesis.

I would also like to thank Nicholas Gerard, David Sorkin, Matthew Sorkin and Ben Harendza for their work in implementing supporting software. With special thanks to Nicholas Gerard, this team provided a framework for enabling a seamless experience when using the tools implemented by my work.

I would also like to acknowledge Professor Matthew Stephan of the Department of Computer Science and Software Engineering at Miami University as the first reader of this thesis. I am grateful to his impactful comments at the beginning stages of the project.

Finally, I would like to express my love and gratitude to my parents and fiance. Without their support throughout this process I would not have been able to maintain my efforts and accomplish this work. Thank you.

Author

Nick DiGennaro

# Chapter 1

# Introduction

Software engineering as a discipline encompasses every step in the software creation process. From requirements and design, to quality assurance and deployment, a software engineer's main goal is to program and produce working software. With so many steps before code creation, many things can go wrong.

Badly defined system requirements are among one of the main reasons software systems fail [4]. Since software modeling focuses on accurately and abstractly representing domain knowledge as part of system design, this is a step in the right direction. However in standard applications of modeling, the models do not serve as primary artifacts and are often not updated and become obsolete and useless [5]. To remedy this, models need to be treated as primary artifacts in software engineering. This necessity, along with the demonstrated effectiveness of Model-Driven Software Engineering (MDSE) practices [1] is the perfect driver for the development of tools and techniques to support continued adoption of this revolutionary paradigm.

The idea of MDSE is to use common software engineering modeling processes and directly relate them to executable code. This allows for straightforward conversions between models and code. This conversion process is known as a Model Transformation (MT). From a model, software engineers can apply a model transformation and automatically generate reliable and usable code. Helping software engineers through the model transformation process is the motivation for this thesis. Providing engineers with intuitive, transparent and guided model transformation tools is important in promoting the adoption of MDSE as a paradigm.

## 1.1 Motivation

As MDSE gains more attention from software engineers and other IT professionals, problems with adoption of the methodology are beginning to show themselves. Once a professional is convinced of the MDSE process, a big problem is standing in their way: education. Many tools have already been implemented that allow code generation and include the other advantages of MDSE. Unfortunately, these tools are extremely complex and confusing. They provide engineers with lots of power, but at a steep learning curve. New MDSE developers might give up before they can even successfully install these tools, let alone create a working model. The goal of a larger ongoing project that this thesis contributes to is to fix this educational gap by implementing a new, lightweight, educational tool, the Instructional Modeling Language (IML).

IML will be used to educate software engineers new to MDSE. As a subset of the Unified Modeling Language (UML), IML is accessible to students of MDSE, while still providing them with the robust benefits of MDSE. The work completed in this project focuses on allowing users to

complete structural model transformations. In order to support these goals, this thesis implements Model-to-Text (M2T) and Model-to-Model (M2M) transformations in such a way that the concepts are central to their application, and the processes are intuitive and transparent. The concepts of M2T and M2M transformations are further explored in Chapter 2 Background & Related Work.

In order to effectively demonstrate the intuitive model transformations, IML first needed a framework, including a model editor. As discussed, simply installing MDSE tools is a barrier to developers, therefore, it was decided that the collection of MDSE tools should be implemented using a web framework. The only access requirement for IML is a web browser.

## 1.2 Contributions

In support of the larger project goals, this thesis makes the following significant contributions to the state-of-the-art of MDSE:

- A web-based modeling tool that allows users to create and edit graphical models while enforcing proper modeling practices through well-defined error messages (Chapter 3).
- An implementation of transparent structural model-to-text transformations via Java code generation, including readable and commented code with visual links to the graphical models (Chapter 4).
- A clear and guided model-to-model transformation application with a fully capable engine for performing model transformations between user defined metamodels (Chapter 5).
- A systematic validation of each model transformation engine using a sample set of input models compared against manually derived expected outputs (Chapter 6).

### 1.2.1 Overview

The remainder of this thesis is presented as follows. Chapter 2 provides sufficient information for understanding the MDSE processes that were implemented in this project, as well as an exploration of, and comparison to, related work on modeling tools and model transformations.

Chapters 3, 4 and 5 describe each technical contribution in-depth. Beginning with the Web-Based Model Editor in Chapter 3, followed by Code Generation in Chapter 4 and Model-to-Model Transformations in Chapter 5.

The thesis is concluded by presenting the results of evaluation for each contribution in Chapter 6. This chapter also presents a discussion on the evaluation and the research contributions. Finally, Chapter 7 presents some threats to validity, followed by potential future work and a summary of the thesis.

# Chapter 2

# Background & Related Work

The following chapter provides relevant background work related to the project. This chapter also includes current related works for comparison.

## 2.1 Background

The background for this document includes work from the following fields:

- Model-Driven Software Engineering
- Educational Language Design

### 2.1.1 Model-Driven Software Engineering

#### Understanding Terminology

Model-Driven Software Engineering (MDSE) has a few similar fields associated with the domain. While they are similar, understanding the differences is necessary to better understand MDSE. For example, Model-Driven Engineering (MDE) is often used interchangeably with MDSE, however, there is a difference. MDE focuses on models driving the engineering process while MDSE focuses on models driving the *software* engineering process. Despite MDSE being a slight modification of MDE, they are still used in place of each other in many academic works. [6] [7] This thesis uses MDSE instead of MDE but, the mild equivalence between these two terms is important to understand when looking over references made in this paper.

Another term associated with MDSE is Model-Driven Software *Development* (MDSD). Software development is the process of creating and maintaining software. This is different from software engineering. Software engineering encompasses the entirety of software creation from architecture and design to implementation. Development and engineering overlap on implementation but engineering focuses the creation process.

The final term to consider is Model-Based Software Engineering (MBSE). MBSE can be seen as a subsection of MDSE. In MBSE, after the requirements of a system are gathered, these can be analyzed and used to make models of the system. These models can then be given to developers to help in implementation. In MDSE, models are primary artifacts of a system and are used to generate code.

To summarize the previous paragraphs and produce one definition, MDSE is the process of using a defined metamodel of a problem domain to create models. The models produced from this

Table 2.1: Terminology Comparison.

| **Term 1** | **Term 2** | **Difference** |
|---|---|---|
| MDSE | MDE | MDSE specifies models driving the software engineering process while MDE specifies models driving the engineering process more generally |
| MDSE | MDSD | MDSD focuses on the development of software while MDSE specifies the entire process from requirements to implementation to maintenance. |
| MDSE | MBSE | MBSE uses models of a system to help analyze a system while MDSE uses models as the driver for development and implementation. |

metamodel are possible solution systems for the problem domain. The main driver of the software engineering process is the model. The model is used in every step from design to implementation to testing. See Table 2.1 for a graphical comparison of terminology.

For a better understanding of modeling software systems, Figure 2.1 is displayed to depict a simple university/school system using the Eclipse Modeling Framework (EMF). There is a base class Person defined. Professor and Student then extend to this class defining a parent/child relationship. Furthermore, a relationship is defined between Student and Class forcing a Class to have at least 12 Students. A similar relationship is defined between Professor and Class forcing every class to have at least one professor. Additionally, this relationship defines that a class can have no more than 3 professors.

Defining these relationships in a system is a concept in MDSE referred to as metamodeling. Once the metamodel for this system is created, developers can begin defining concrete implementations of Students and Professors. Metamodeling is discussed in depth in the following section.

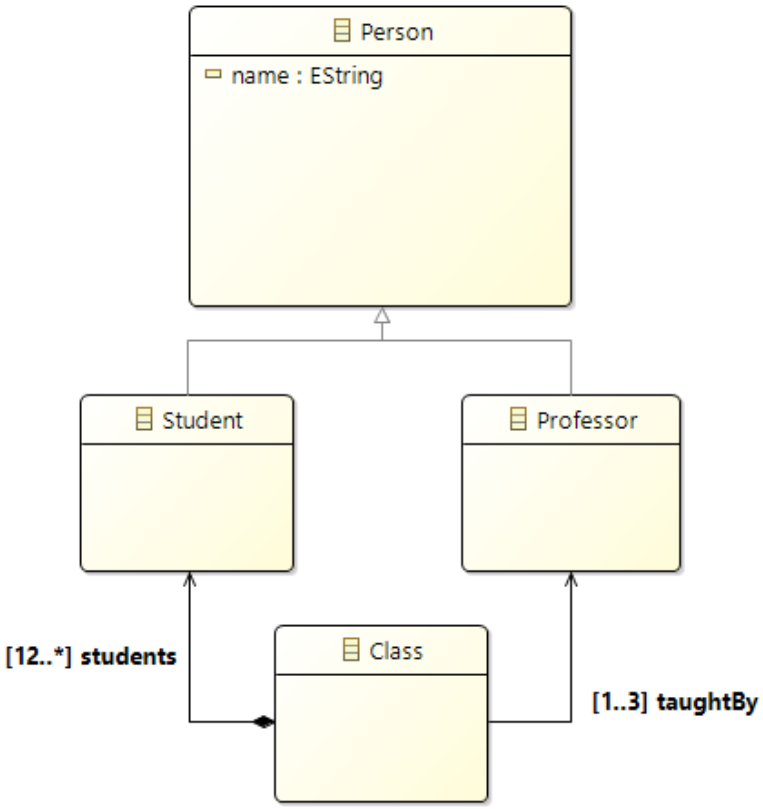



Figure 2.1: Example Model of Simple University System

**Metamodeling**

Thomas Kühne observes that the prefix "meta" is added to a word whenever a process is applied twice. He provides the example of meta-learning. An instance of meta-learning could be learning about learning strategies [8]. In simpler terms, learning about learning. Kühne explains though, applying a process twice is too general of a definition for using "meta". He proposes a situation where a model is created. Then, the model is edited and updated. Even though the process of modeling was applied twice to the same model, metamodeling did not occur. Kühne gives 3 constraints for a metamodel relationship between models. He says the relation must be acyclic, anti-transitive and level-respecting [8]. More simply, there must be a directed relation from a model to its metamodel that describes it is an *instance of* the metamodel. And, there must be another directed relation from the metamodel to its model stating the model is a *type of* the metamodel.

Metamodeling finds similarities between solutions to similar problems. For example, Point of Sale (PoS) systems are commonly used in storefronts to process monetary transactions between the place of business and the customer. There are many different implementations of PoS systems but they are often very similar. One system may hold a SQL Server database of the current in-store stock while another system uses a MariaDB database. Each uses a different database language but each

implementation uses a *database* to keep track of in-store stock. This is the level of abstraction that is used to relate different solutions. At this level, concrete implementation details are not necessary for the database, or any other component of the system. The metamodel for the PoS domain should contain only an abstraction for the database. The database language is left out in the model. This is because the metamodel must capture each possible PoS system. Therefore, the metamodel must only specify commonalities between PoS systems, like the existence of a database, and leave out more concrete differences, like the database language used in implementation.

Once a metamodel is defined, this allows the instantiation of instance models. A metamodel defines an application domain and provides the structure for all the instance models possible in that domain. Recalling the example of PoS systems, the metamodel for that domain provides the structure for all possible PoS systems. Using the elements provided by the PoS system metamodel, a user can define a specific PoS system instance. Instead of using generic elements, the user can define specifics. For example, the PoS instance model can be described to have a SQL Server database, instead of just a database. It is important to note that the instance model must follow the rules set by the metamodel. These rules constrict the elements that can be included in the instance model and can even constrict how the elements are implemented. For example, the metamodel for PoS systems may not only include a database element, but required one. This would mean every instance model made in the PoS system metamodel must include a database in the system. If all of these metamodel rules are satisfied, the instance model is said to conform to the metamodel. Creating a metamodel to define instance models that can be used to solve different problems in problem domain is known as Domain-Specific Modeling. This concept is discussed in the following section.

**Domain-Specific Modeling Languages**

Usually when metamodels are defined, they are used to create Domain-Specific Modeling Languages (DSML). DSMLs are used to abstract a specific problem space with the use of models. Referring to the earlier example, the specific problem space would be PoS. To understand DSMLs it helps to first understand Domain-Specific Languages (DSL). As explained in a survey by van Deursan et al. [9], a DSL is a programming language restricted to a single domain. The purpose of creating a language for a specific domain is to harness more power over the given problem domain. The difference between DSL and DSML is DSMLs define a *modeling* language to create power over a problem domain. A report by Ethan Jackson [10] describes DSMLs more precisely saying a domain is first described by a model of computation (MoC). A MoC is a machine type specific to the given domain. An MoC can be considered a metamodel because it establishes relationships between each model in that domain. A model made based on the MoC is a software system for that domain. Thus, "programming languages for particular MoCs are called domain-specific modeling languages" [10].

Recently, there have been attempts by researchers to define a metamodel for DSMLs. Nordstrom et al. [9] says modeling environments are made specifically for a problem domain, however, there is a set of commonly found modeling concepts. Using these concepts it may be possible to create a model for the modeling environment, also know as, a metamodel. Tolvanen and Rossi [11] attempted an implementation of a metamodeling tool. The tool requires a developer to define a

domain within the tool. Once the domain is defined, the tool can be used to create models within that domain. Code can then be generated from these models.

Tolvanen and Rossi's tool includes the concept of code generation based on models. This concept is important in DSML and MDSE as a whole, therefore, this is discussed in depth in the next section.

### Model Transformations & Code Generation

Creating models of systems also allows for *Model Transformations (MT)*. A MT takes a model of a system within a domain and transforms it into a model of a domain that is the same, similar, or different. To use the definition of Sendall, MTs, "take one or more source models as input and produce one or more target models as output, following a set of transformation rules." [12].

Using more precise language to describe MTs, consider the metamodels, MM and MM'. For each of these metamodels, conforming instance models can be created. Consider model M, defined by metamodel MM and model M', defined by metamodel MM'. If a transformation specification can be created between metamodel MM and metamodel MM', then model M can be transformed into model M' [1]. Figure 2.2 provides a diagram to display this process.

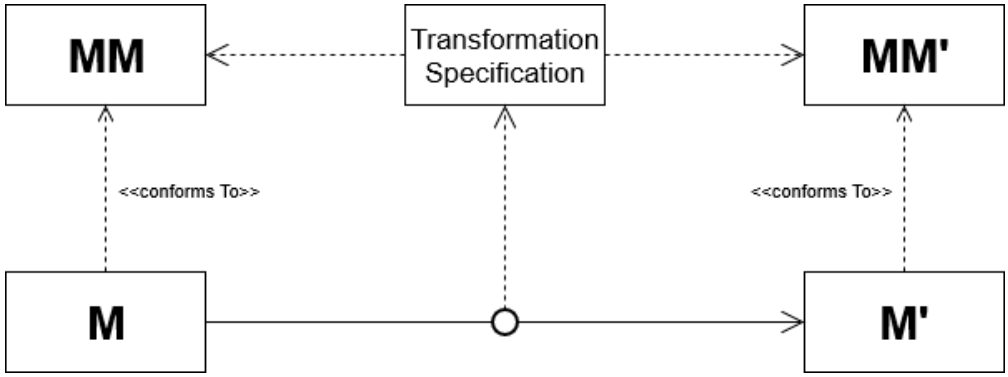



Figure 2.2: Model transformation from M to M'. Adapted from Model-Driven Software Engineering in Practice, figure 8.1 (a) [1].

It is important to remember figure 2.2 is at a high level of abstraction. To make this concept more concrete consider a survey done by Lin [13] where they envision source control becoming a widely used application of MTs. They refer to this as Model-Centric Version Control (MCVC). The process of MCVC would be utilized in software development to compare an archived version of a file to the current version of the file [13]. The difference between MCVC and current source control would be the ability to see the differences with visual representations instead of highlighting additions and deletions in code. When comparing MCVC to figure 2.2 it may help to consider model M' as the current file and model M as the archived file so there is a perception of working backward.

*Code generation* is the process of transforming the model of a system into code. Furthermore, code generation can be seen as a type of model transformation. For example, models can be created using graphical representations, but they can also be created using textual representations. A

metamodel of graphical representations includes defining a set of symbols that can be combined to create a graphic of a model. A textual metamodel includes defining grammar and syntax that can be used to create a textual representation of a model. If a translation can be found between the graphical symbols of a metamodel and the grammar and syntax of a textual metamodel then a model transformation can be performed to move between graphical and textual representations.

The concept of transforming graphics to text can be taken a step lower. Consider the previous example displayed in figure 2.1. Using this example, if the metamodel MM is replaced with some modeling language and metamodel MM' is replaced with some coding language the application becomes more apparent. If translations from symbols to text can be described, then models created in the modeling language can be translated into source code.

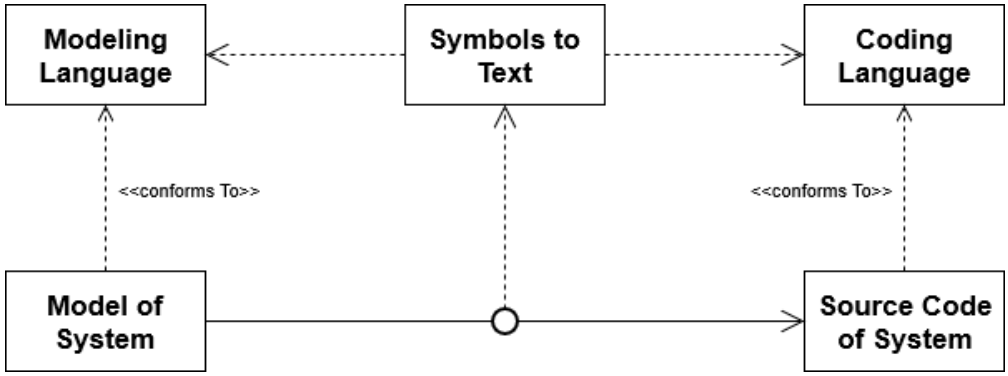



Figure 2.3: Code Generation. Adapted from Model-Driven Software Engineering in Practice, figure 8.1 (a) [1].

**Behavioral Modeling & Code Generation**

Code generation has already been mentioned in this paper but this is still a developing topic in MDSE. The code generation previously considered can be seen as early attempts at code generation. These early attempts typically only create code that is referred to as the skeleton of a program. Skeleton code involves basic structure such as method headers and class definitions. This does not contain behavioral code. For example, code inside a method that performs operations would be considered behavioral. Buchman's Action Language for Foundational UML (ALF) [14] allows for textual modeling of program behavior. Even though this model is textual and not graphical, it still extends the effort in behavioral modeling. This is important to facilitate the use of models throughout the software engineering process.

Attempts at behavioral modeling must use state machines. State machines are imperative for behavioral modeling because they allow computer scientists to specify the various states of a program and how the program transitions through these states depending on stimulus. Lamport describes state machines as a set of states, a set of initial states and the next-state relation for each state [15]. The next-state relations are how programs know when to transfer from one state to the next. These transitions are triggered by an external stimulus, as mentioned earlier. More generally, the state of a program is identified by the values of its variables. When a stimulus is received, these values are

evaluated and a decision is made. This can be compared to code by considering a method call as a stimulus. When this method is called, an if statement evaluates some of the current values of the state and chooses a path of execution depending on the evaluation. This may cause execution that changes variable values and transitions the program into a new state.

There are software tools that have implemented behavioral modeling and code generation. Papyrus-RT and Simulink are popular tools for creating state machines with a graphical interface. Both are intended to be used to model real time systems. A real time system is software that must take a continuous stream of input and make decisions depending on the input received. Using the models of these systems, code can be generated. The main difference between these tools is how they were implemented. Papyrus-RT is designed as a plugin for the popular IDE Eclipse. Therefore, the code generated from Papyrus-RT is in Java. Simulink is standalone software that is used to generate C code.

To gain a better understand of Papyrus-RT, Hili created a paper in which Papyrus-RT is compared to UML-RT [16]. UML-RT is another real time DSML that is an extension of the Unified Modeling Language (UML). Furthermore, an example of Simulink is provided through this work [17]. In this paper, Simulink is used alongside of MATLAB to create data center simulations. The simulations were intended to mirror a data center under different circumstances, such as at high traffic times.

**Model Management**

The processes of MDSE that have been covered in this paper are not complete. The software engineering process includes the development process of a system. This was touched on briefly when defining terms but not covered in entirety. This is because Software Change Management for MDSE is still maturing.

What makes software change management different from other processes in software engineering is a piece of software must be compared to an older version of itself. Therefore, these versions must be kept over time, compared, and properly displayed. The solution for this in traditional software engineering is to simply display the lines of code that were added or deleted. With MDSE, changes in a software system are displayed at the architectural level, and as behavioral modeling matures, at the behavioral level. This could drastically increase the understanding of consequences from changing software.

Recently, academics have been focusing efforts on keeping track of software change from models and properly displaying these changes. In Owe's recent publication [18] they created a modeling tool for distributed systems. This tool is supposed to allow developers to modify their systems however they deem necessary and still be supported by verification and re-verification from the tool.

To bring a perspective using the pervasive tool EMF, Brun [19] created a tool, EMF Compare. This tool is an extension of EMF to provide model comparisons in EMF. More generally, the paper defines model comparisons falling into three categories, calculation, representation and visualization. These categories describe the steps for comparisons respectively from the algorithm used in comparison, to the output of that algorithm and how that output can be made readable by humans. These categories are defined to illustrate how EMF Compare handles them for each step.

Moving away from Eclipse based model management solutions, Rapos presented a thesis in model management support using Simulink [20]. The basis of the thesis is to use model management for testing. By understanding how a system changes in its lifespan, Rapos proposes this information can be used determine the impact on test cases. The outcome of the thesis is three tools that work together to manage the model evolution of systems made using Simulink.

### 2.1.2 Educational Language Design

Programming languages are made to manipulate computers to create solutions to software problems. Educational Languages are also created to solve software problems except they are specifically made for new programmers. The design of these languages prioritizes features that make code easier to understand and utilize instead of making run-time more efficient, for example.

In the book chapter by Mendolsohn [21] it describes three criteria that teachers must consider when choosing a language to teach their students. The first is described as the language level which must consider how much time will be spent on learning programming or learning how to apply programming concepts. The second criteria considers whether skills learned from the language can be applied to a different context. Lastly, the chapter identifies teaching style which must decide whether a teacher will be hands on or allow students to explore and learn on their own. While each of these considerations focuses on decisions to be made by a teacher, they can be used when designing a programming language.

An example of an educational language created by Holt [22] is Turing. This programming language was made to balance teaching programming and serious development. For example, Turing was designed to use syntax similar to Basic because the syntax of this language is simple and "clean". This allows new programmers to learn programming, as identified in the earlier paragraph, as a criteria to consider for an educational language. Turing also makes considerations for more experienced programmers. This allows new programmers to begin learning and evolve to more serious development within the same language. This evolution is not a criteria that is considered by Mendolsohn.

### 2.1.3 Software to Support Teaching Goals

Using software to support teaching is a concept that exists outside of computer science and software engineering settings. Figuring out how to effectively integrate computers and software in the classroom has roots deeper than teaching MDSE [23]. And, there is an overlap of psychology whenever teaching practices come into question [24] [25]. When teaching humans psychological science is used to find the most effective teaching practices. This allows developers to implement or supplement those practices.

For example, Penuel [26] wrote about the co-design process. This process involves developers, researchers and teachers in the development of educational software. The process is similar to agile development, requiring developers and teachers to evaluate the design of multiple prototypes. Penuel wrote that the researchers were surprised by the teaching styles in classroom. By simply observing classroom sessions, the researchers realized teaching is a mix of traditional lectures and hands on activities instead of being all student centered. Without observing the teachers in practice,

the researchers would have been incorrect in their initial assumptions of the psychology of teaching. More tools and related work are considered in the following section.

## 2.2 Related Work

This section looks at the state-of-the-art of MDSE, focusing on tools that support MDSE and software created to support teaching.

### 2.2.1 Tools to Support MDSE

The following section introduces current and pervasive MDSE tools that are related to the IML project. Tools used in metamodeling are presented first and can be compared to IML's web-based model editor. The next section discusses model transformation tools and can be compared to the code generation and M2M transformation implementation in IML. The final section introduces other education modeling tools that can be compared to the entire IML project.

#### Metamodeling Tools

Recall that Tolvanen and Rossi attempted an implementation of a metamodeling tool [11]. Their tool MetaEdit+ allows users to define DSMLs and then use these DSMLs to create models. From these models code can then be generated. More recent papers [27] focus less on defining a metamodel for metamodeling and more on finding translations between two metamodels. The work done by Zhao et. al. takes in two DSMLs and attempts to find cross sections between the domain. This approach allows DSMLs to remain independent of each other in specification because they do not need to conform to a larger metamodel. Similar to IML, MetaEdit+ implements the same MDSE process of metamodeling, instance modeling, code generation. However, IML is able to do this within one tool. MetaEdit+ is actually a suite of tools. MetaEdit+ Workbench is needed for metamodeling and MetaEdit+ Modeler employs Workbench metamodels for instance modeling. IML is able to handle metamodeling or instance modeling without having to install a new tool, or even change the web application.

Eclipse Modeling Framework (EMF) is one of the most pervasive tools for metamodeling in MDSE. EMF provides a modeling framework for creating applications within Eclipse. Using EMF Core, developers can specify a metamodel. Using the metamodel, an instance model of an application can be made. Then, code generation is possible using the instance model. The code that is generated is Java. This is because EMF is implemented using Eclipse, which is most notable for developing Java code. [28]. As described with MetaEdit+, IML places simplicity at its focus when compared to EMF. Both tools are more expressive than IML, but this is as intended. Students of MDSE should use IML to learn proper processes. Once learned, students can migrate to tools like EMF and MetaEdit+ to create robust software.

A less pervasive tool, JetUML, aims to create "lightweight software modeling" [29]. This tool is important to mention because its goal is to create a lightweight model editor. The tool uses a subset of UML with the initiative to minimize the use of UML semantics. Initially, the tool restricts users

to the most basic UML elements. This is supposed to allow models to be used as quick sketches, which is where the tool diverges from MDSE. Users are given the ability to enable more features of UML as they see fit. There are no MDSE features like instance modeling or code generation. While JetUML is no MDSE tool, its lightweight design is important to note. IML is similar to JetUML because each implement a subset of UML for simplicity. While JetUML is well-suited to educate software engineers, it is no position to demonstrate MDSE techniques. IML and JetUML share a motivation for simplicity, but IML applies this motivation to the processes of MDSE.

**Model Transformations**

Model transformations also need support using tools. These tools require the ability to specify the rules to transform a model of one domain to the model of another domain. For example, Jouault et. al. created a DSL, ATLAS Transformation Language (ATL) [30]. This tool allows users to make model transformations within the Eclipse framework. The driver behind the tool is a rules based DSL. This means transformations are described in a programming language syntax. Instead of directly displaying models, ATL uses other methods of abstraction such as trees and tables. This created a more complex tool to learn but allowed Jouault to leverage common useful characteristics of programming language design. For example, included in the language are called rules. These are essentially method definitions. They are run by using the name of the called rule and they can even take arguments. Including these feature allows the reuse of code when solving similar problems.

ATL is not the only model transformation tool. A survey of model transformation tools done by Stephan and Stevenson [31] considers three tools, ATL, KerMeta and EMF Model Transformation Framework (EMT). They evaluated each of the tools based on expressiveness, ease of use and modularization. Expressiveness describes the ability of a tool to perform complex transformations. This is related to providing programmers with the ability to make complex transformations. The ease of use of a tool describes the design and learning curve of the tool. This is related to the complexity of the interface. Stephan and Stevenson focus on the trade offs between using a graphical interface or a text based language like what is utilized in ATL. Lastly, modularization focuses on the ability to build constructs that can be reused to solve similar transformation problems. For ATL, Stephan and Stevenson decide that the language based interface improves the expressiveness of the tool but declines the ease of use. They came to a similar conclusion for KerMeta, except this tool uses object-oriented language commands, whereas, ATL and EMT are rule-based languages. Finally, Stephan and Stevenson decide EMT's graphical interface far exceeds the ease of use of the other tools. However, they find EMT is missing many features and lacks expressiveness. Ultimately they decide ATL has the best balance of expressiveness, ease of use and modularization.

Acceleo is another model transformation tool. This is a templated-based transformation tool that applies the Object Management Group's (OMG) Meta-Object Facility (MOF) Model to Text Language (MTL). Users of the tool can define a code template and apply it to an EMF created model. The code template allows users to equate model elements with code from any programming language. Once applied, the code template will generate the corresponding code from the EMF model. Obviously, this tool is more dynamic than IML's code generator because of its ability to generate any coding language. However, it is far more complex. MDSE students hardly understand the concept of model transformations and would likely struggle with creating a code

template for code generation. IML readily serves its purpose when compared to Acceleo. Students will gain understanding of the model transformation process and observe its benefits. After mastering IML model transformations, students will find an easier transition to more complicated tools like Acceleo.

Moving beyond model transformation tools, a paper by Varró et al. [32] attempts to formalize model transformations mathematically. They focus on the correctness and completeness of model transformations. The paper looks at model transformations from two views they describe as, the model dependent approach and the Metamodel/grammar dependent approach. The first approach looks at specific cases of model transformations and attempts to check the correctness of that instance. Whereas, the Metamodel/grammar dependant approach tries to prove the correctness of the transformation rules between two metamodels. It is obvious that formal processes defined in this paper could be useful for ensuring the correctness of model transformations performed by the previously mentioned tools.

In fact, Stephan and Cordy wrote a survey of current model comparison techniques that can be used for model transformation testing [33]. They define model transformation testing as the process of performing a set of transformations to receive an output set of models. The actual output set can then be compared to an expected output set of the same transformations. This is related to model comparison for the obvious reason of deciding if a model is correct, but comparisons can be more complex. For example, model comparison could be used to discover the specific similarities and differences between models. This could be applied to fields like model clone detection or versioning, among other applications. A unique contribution of this paper is their introduction of homogeneous comparisons and heterogeneous comparisons. Homogeneous comparisons involve comparing two models conforming to the same metamodel. While heterogeneous comparisons consider two models, each conforming to a different metamodel. They argue that the heterogeneous comparisons can be useful for test-case generation. Stephan and Cordy used a tool called EMF Compare to perform a model comparison with models conforming to two different metamodels. This produced a list of differences that they argue would be a useful start for creating test-cases.

Stephan and Cordy are not the only researchers concerned with model transformation testing. A paper by Sahin et. al. [34] explains that the process of creating model transformations can be prone to bugs. In the paper they propose a tool that addresses model transformation testing. Their tool generates test cases based on the metamodel that the transformation is being constrained within. While the test-cases are generated using only homogeneous comparisons, as described by Stephan and Cordy, this paper displays there is work being done for model transformation testing.

Both Stephan and Cordy and Sahin et. al. mention the concept of versioning in their papers. Versioning introduces the concept of managing and storing the changing models as a system evolves. This is the same problem that current programming practices face. Typically programmers use tools like GitHub to keep track of the changes being made to code. In MDSE this problem is known as model management and cannot be solved by services like GitHub.

There is one last application suite to cover that has a tool for each of the tool types mentioned above. Visual Paradigm (VP) is a commercial MDSE application that attempts to include all MDSE processes and features inside one tool. From modeling, to code generation, to M2M transformations, VP has an implementation for each of these MDSE processes. IML shares this goal because

it aims to include educational tools for each MDSE process. However, VP is missing a model-driven testing tool, which IML plans to implement. One of VP's shortcomings is it has so many features, MDSE students would struggle finding where to start. This is a common issue with many of the robust MDSE tools. Additionally, VP is a commercial application, which means it costs developers money to use. While there is a free, community edition available, this version does not include important features like code generation. MDSE students would hit a pay wall before they could fully explore the benefits of modeling.

In addition to the state-of-the-art of MDSE tools, this background needs to include related works in MDSE education. The next section considers software created for this purpose.

**Educational Modeling Languages**

Educational Languages are powerful because they create a lower barrier to entry for new programmers. This concept is important for MDSE because it is a new field to Software Engineering that is looking to become more pervasive. To promote developers to utilize MDSE methodologies, it's imperative they have a low barrier to entry. Therefore, it is necessary to create useful educational languages for modeling languages. Currently, there are modeling languages available to developers, however, these languages are often too complex for new users. They require a curriculum guided by a teacher specialized in MDSE and the modeling language being taught.

There does exist an educational modeling language, Epsilon [35], created for the Eclipse IDE. This language is similar to the proposed educational language, IML [36]. Each language implements a broad spectrum for modeling, however, Epsilon has some drawbacks. The main drawback with Epsilon is it teaches modeling from the basis of programming concepts. A consequence of this is, Epsilon models must be defined using a textual interface that looks like a programming language. While this may feel natural to software engineers, this distracts from the purpose of modeling. Modeling is intended to abstract the complexities of system implementation. A better implementation of an educational modeling language would be model-first. This is exactly what the IML project intends to create. Models and MTs in IML are defined using a guided, graphical interface.

Along with being model-first, IML aids new model-driven software engineers by including tools that streamline the MDSE process and create a lower barrier to entry. The design of IML in its totality makes it a complete end-to-end modeling tool, unlike Epsilon. For example, IML's final feature set includes both structural and behavioral modeling frameworks. This eliminates the need to download and learn one tool for structural modeling, and another tool for behavioral modeling. To perform structural and behavioral modeling with Epsilon, students are required to download a modeling language for each type of model, whereas the final IML toolset (once complete) includes the full range of MDSE applications in its base offering. This means that IML allows students to define a metamodel of their system's domain, create specific structural instances, and add behavior to each instance using state-based models. Further, students are able to generate executable code for any system model as well as perform other MDSE activities such as model-based testing. A model-first, end-to-end tool like IML solves many of the barriers to adoption of MDSE.

Umple is another modeling tool with an educational focus [37]. This tool implements a modeling language based on UML class and state diagrams. The goal of Umple is to create a lightweight

implementation of a modeling language (similar to IML) as well as synchronize models and code. IML and Umple are similar because each have lightweight implementations of structural and behavioral modeling based on UML models. However, they are different because IML prioritizes models as the primary artifacts, whereas Umple places equal empahsis on the development of code and models, and introduces complexity through an added intermediate representation. An additional differentiation is, as previously expressed, IML is a complete end-to-end tool, capable of providing students with the full range of MDSE applications including metamodeling, model transformations, and model-based testing. While Umple also has implementations of structural and behavioral modeling, it does not allow users to compile and run their code within their web-based tool. Further, Umple does not provide support for metamodel conformance, nor the creation of instance models based on user defined metamodels.

### 2.2.2 Motivating Work

The motivating work for this background focuses more generally on advancing MDSE. Currently, MDSE struggles with the simple fact that it is not widely used in development. Recent work has provided powerful and useful tools to developers in the MDSE space but the barrier to entry remains high. The following motivational works focus on bringing MDSE to a wider audience.

In a case study on a one semester course in MDSE [38], Rapos designed assignment material in a three tiered manner. For each topic there was a low stakes lab assignment for introduction. This was followed by a personal assignment to improve understanding of the material. Then, a group project to provide mastery of the subject. While Rapos found this system was generally a success, there was an issue with the software tools available for teaching. Rapos points out that some of these technical difficulties came from school computers not being sufficiently prepared that semester, lacking the software needed to complete assignments. But, Rapos also considers the lack of one end-to-end teaching tool for MDSE. The requirement for installing multiple tools like EMF, Papyrus-RT and Simulink proved to be difficult for students with proper instruction, let alone developers looking to learn MDSE practices on their own. Any student of MDSE may retreat back to their current technology stack if installing software and getting different tools to work together proves to be too cumbersome. This is the main takeaway from Rapos' paper which provided inspiration for the next paper.

Following the one semester case study, Rapos writes another paper focused on providing the necessary MDSE teaching tool [36]. He coins the term Instructional Modeling Language (IML) for the new DSML aimed towards teaching. IML is intended to include "meta-modeling, model transformation, simulation, and code generation" [36] within one lightweight application. The goal of including each of these features is to provide students with exposure to the full range of MDSE processes within a simplistic tool. Rapos' vision allows for students to quickly and easily begin developing entry level applications with MDSE when compared to a set of tools that require more expertise. This thesis is the first step in realizing this vision for the Instructional Modeling Language.

# Chapter 3

# Web-Based Model Editor

The first phase of the IML structural model transformation project was the implementation of a web-based model editor. This editor is an MDSE model editor. MDSE model editors are not to be confused with drawing tools like diagrams.net [39]. Drawing tools are similar to MDSE tools because both provide users with abstract representations of software systems. However, drawing tools are different because models created with those applications are not primary artifacts, nor do they require conformance to specific design rules. Drawing tools allow you to complete entirely nonsensical models which does not aid in the understanding of MDSE concepts. MDSE tools create models that become the software system due to their direct translation to code. As discussed in Chapter 2, the Eclipse Modeling Framework (EMF) [28] is an example of an MDSE model editor. This tool allows users to define a metamodel. Once defined, users can then define instance models that conform to that metamodel. The IML model editor uses this functionality as inspiration but focuses on simplicity, guidance, and usability.

## 3.1 JointJS

It is important to recognize that model editors are visual tools. This means computer graphics play a key role in implementation. Rather than creating a graphics engine, we decided to take advantage of the open source nature of the web and web development. There are multiple, open source modeling frameworks that can be easily integrated in a web application. After some consideration, it was determined the open source modeling framework, JointJS [40], would be the best for implementing the IML model editor.

Looking at some of the JointJS demonstrations (see figure 3.1), the framework displayed the ability to create UML class diagrams and UML statechart diagrams. With IML being a subset of UML, displaying these abilities proved JointJS could satisfy the visual needs of IML. Furthermore, JointJS has the ability to create custom elements. This was integral to implementing the model editor because the provided UML elements did not include enough functionality for dynamic model editing. For example, a class represented in UML consists of the name, its attributes and functions in separate, but connected, sections. The JointJS implementation of this did not have the ability to select these sections separately. The model editor needs this ability to edit individual attributes, or the class itself. Using the custom elements, we were able to implement the functionality to select these sections separately.

An example of custom element code can be see in figure 3.2. To create a custom element, developers define attributes of the element in the “attrs” sub-object. Then, what is drawn for these attributes is defined through the “markup” list of objects. In the IML editor, this helps create the

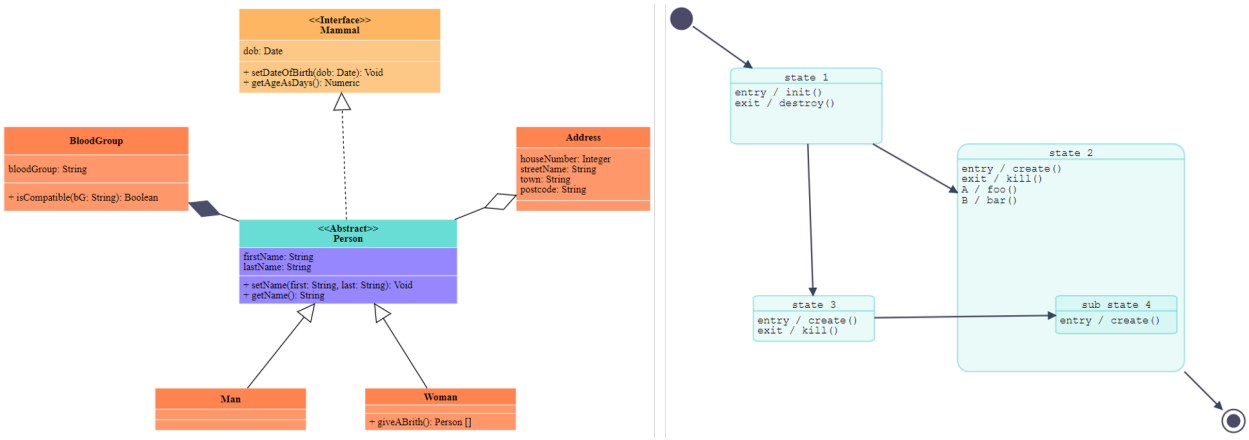



Figure 3.1: JointJS UML model demos. Obtained from [2].

```
var ImlClassElement = joint.dia.Element.define('iml.Class', {
    attrs: {
        classAttributeRect: { ... },
        classNameLabel: { ... },
    }
}, {
    markup: [{
        tagName: 'rect',
        selector: 'classAttributeRect'
    }, {
        tagName: 'text',
        selector: 'classNameLabel'
    }]
});
```

Figure 3.2: IML Class Pseudocode

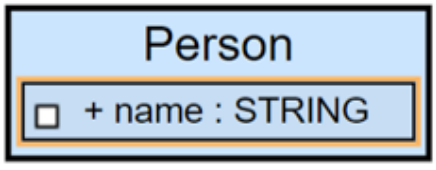



Figure 3.3: IML Class Example

```
1   var ImlInheritanceElement = joint.dia.Link.define('iml.Inheritance', {
2       router: { name: 'normal' },
3       markup: [
4           '<path class="connection" stroke="black" d="M 0 0 0 0"/>',
5           '<path class="marker-source" fill="black" stroke="black" d="M 0 0 0 0"/>',
6           '<path class="marker-target" fill="white" stroke="black" d="M 15 -7.5 0 0 15 7.5 15 -7.5"/>',
7           '<path class="connection-wrap" d="M 0 0 0 0"/>',
8           ...
9       ].join('')
10  });
```

Figure 3.4: JointJS Link Pseudocode

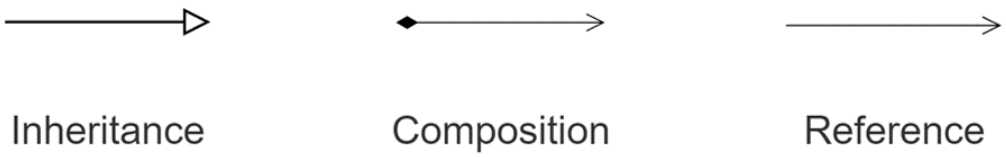



Figure 3.5: IML Relations

model element seen in figure 3.3. This class representation consists of two custom elements. A class element and an attribute element that is embedded in the class element. The attribute within the class is highlighted to indicate that it is selected. This was enabled by custom elements and allows the creation of context menus that depend on the element selected. Context menus are discussed later in this chapter.

Another advantage to JointJS is the link element. Links can be connected between other JointJS elements to represent a relationship. JointJS handles drawing and updating these links depending on the location of the attached elements. Similar to implementing custom elements, links can be extended to add custom arrowheads and labels. Some pseudocode demonstrating this can be found in figure 3.4. This code is different from the custom class because more traditional markup is used to define the link. The arrowheads are drawn using relative points, denoted by each pair of numbers in the "d" attribute on line 6. While this customization is merely a visual difference to JointJS, they are rather important to the IML model editor. Visual differences in IML have semantic meaning. Figure 3.5 displays each of the three IML relations. The white-filled, closed arrowhead indicates an inheritance relation. The relation with a black-filled, diamond source and an open arrowhead denotes a composition relation. While the open arrowhead without the black-filled, diamond source represents a reference relation. The meaning of these relations are defined later in this chapter. For now, it is simply important to know they were made possible by JointJS's flexibility.

## 3.2 Technology Stack

After choosing JointJS as the modeling framework, the next step was to choose the technologies that would surround the editor. Multiple web frameworks were considered like, ReactJS and Laravel. However, the decision was made to use a traditional web stack. This means the code for each web

application was written in HTML, CSS, and JS. These technologies were aided by the JS library jQuery and the CSS library Bootstrap 3. Surrounding the web applications is a website. The website code uses PHP to connect and pass information between web applications and query the server for data.

This technology stack was determined as the best option for a few reasons. First, we had the most experience developing in these languages. Some of us had experience with ReactJS but none were more confident in their abilities than with HTML/CSS/JS. Another reason considered was the longevity of IML. It has been stated that this is the first installment in a larger tool. Using tested and reliable technologies would ensure the project code would not become outdated and that new developers could easily contribute. Finally, again looking at the larger project, IML was to be hosted on a Linux server. Using HTML/CSS/JS lent itself best to working with simple operating systems like Linux, that usually employ languages like PHP to send information between the user and different pages on the server.

With all prerequisite decisions being made, development could begin. The first goal was to implement the web page structure. The next section explains how this was accomplished.

## 3.3 Web Page Structure

A mock-up was drawn of the structural modeling page before implementation. This can be seen in figure 3.6. The web page was to be split into two columns. The left column would contain the menu bar with the modeling pane below it. The right column would contain the model conformance options, palette and context menu, respectively. A majority of the screen real estate was committed to the modeling pane. This is where the user would be able to see and interact with their model which is why most of the UI is dedicated to its display. This is also were JointJS would be employed. Using HTML and Bootstrap, this page structure was implemented.

Bootstrap was integral in enforcing this layout. Bootstrap allows developers to define rows and columns that neatly organization a web page. Bootstrap also has many other CSS class definitions that clean up a web UI. Except for the menu bar, each window used the panel class that places HTML in a rounded box. The panel also offers the option to add a header that was used to title some of the windows. An example of each of these CSS classes can be seen in figure 3.7. This is a code snippet of the palette element. Notice how the top level div element is given the class, “row”. That class separates the element from the others in the right column. Further, this element takes advantage of the panel and panel-heading class to contain the palette icons and inform the user what the window is to be used for. Finally, the menu bar uses the Bootstrap “navbar” class to create and style the dropdowns. This is implemented in a similar manner to the classes in figure 3.7.

The final web page structure can be seen in figure 3.8. Notice how the model conformance options panel was removed from the final design. Originally, this was going to be used to let the user choose between metamodeling in IML or instance modeling with a user provided metamodel. Ultimately, it was deemed unnecessary. We decided that better functionality would be to determine metamodeling vs instance modeling depending on the type of model the user provides. Or if they did not provide a model, then metamodeling in IML is selected by default.

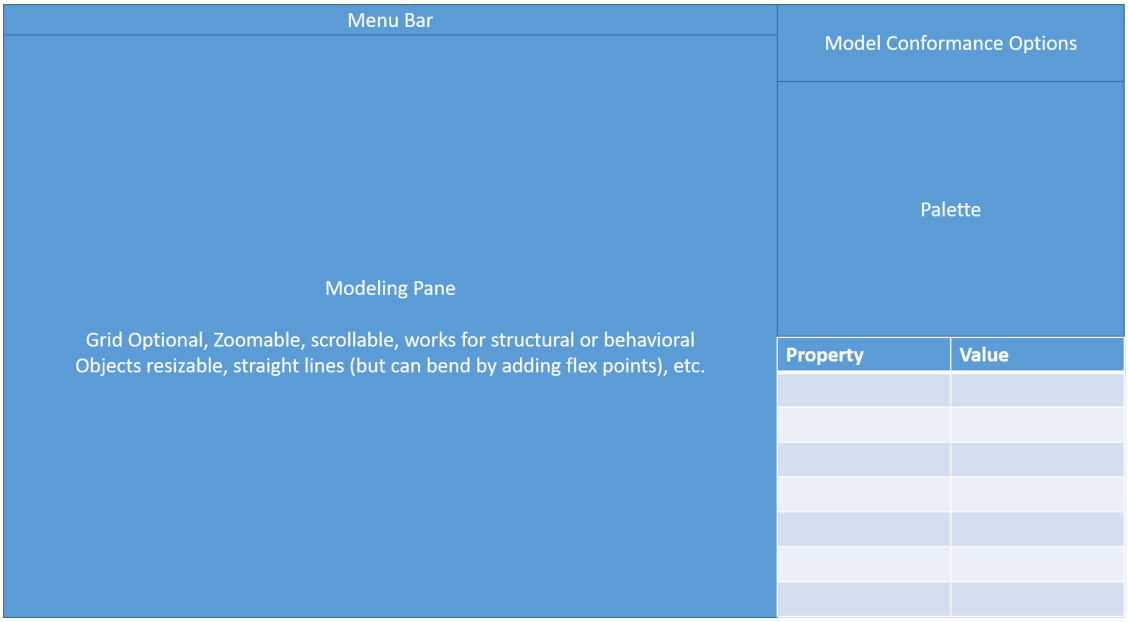



Figure 3.6: Model editor mock-up

```
<div class="row">
    <div style="height:300px; margin-bottom: 10px;" class="panel panel-default">
        <div class="panel-heading text-center">Palette</div>
        <div id="paletteIcons" class="palette"></div>
    </div>
</div>
```

Figure 3.7: Palette code snippet

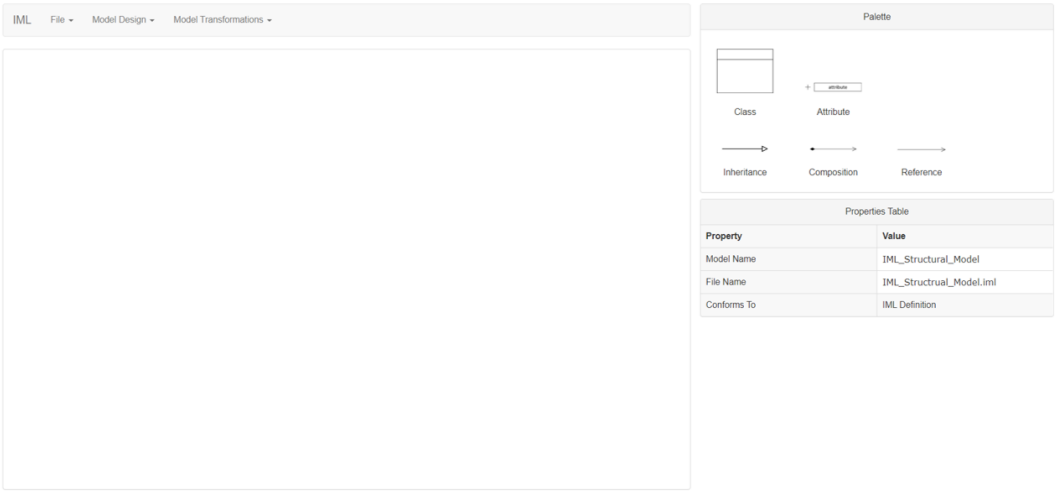



Figure 3.8: Final version of the model editor UI.

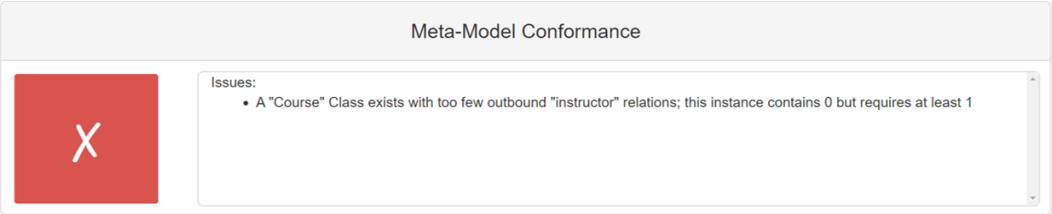



Figure 3.9: Example of Meta-Model Conformance panel when an instance model is out of conformance with its metamodel while instance modeling.

The image in figure 3.8 displays the model editor in metamodeling mode. This UI changes slightly when instance modeling. The Meta-Model Conformance panel is added in the right column. This panel gives the user guidance for implementing a conforming instance model. If the current instance model state does not conform to the metamodel, a red 'X' symbol is displayed with the conformance errors. These errors tell the user what is wrong with their instance model and what changes are needed to achieve metamodel conformance. An example of the Meta-Model conformance panel responding to a non-conforming instance model can be seen in figure 3.9. Once the instance model conforms to the metamodel, the red 'X' is replaced with a green check mark and the conformance issue text area is cleared.

With the final web page structure decided and implemented, we could begin to add functionality. The first component to be worked on was the modeling pane. This required implementation of the palette and back end model data structures. These implementations are explained in the following section.

## 3.4 Editor, Palette and Backend

To start, a JointJS model consists of two data structures, the paper and the graph. Compared to the Model-View-Controller design pattern [41], the paper can be seen as the view, while the graph can be seen as the model. Element changes and additions are typically made through the graph. And, the paper typically handles interactions with the HTML such as events or finding model elements using the corresponding HTML element.

Looking at other MDSE modeling tools, basic functionality allows users to choose items from a palette and add them to a model. This was the first thing to implement for the structural modeling page. As mentioned in section 3.1, JointJS allows developers to create custom model elements. Using this feature, the IML class and attribute elements were defined. The class element was implemented to consist of one box split into two rows. The first row is to hold the name of the class, while the second row is to keep track of each attribute added to the class. Similarly, the attribute element was defined to consist of a box with a single row displaying the details of the attribute. The attribute details are consistent with UML standards. The IML attribute displays its visibility, followed by its name, the variable type and a value, if provided. See figure 3.10 for a visual comparison between IML attributes and UML attributes. The UML notation is detailed in the OMG Unified Modeling Language specification document [3].

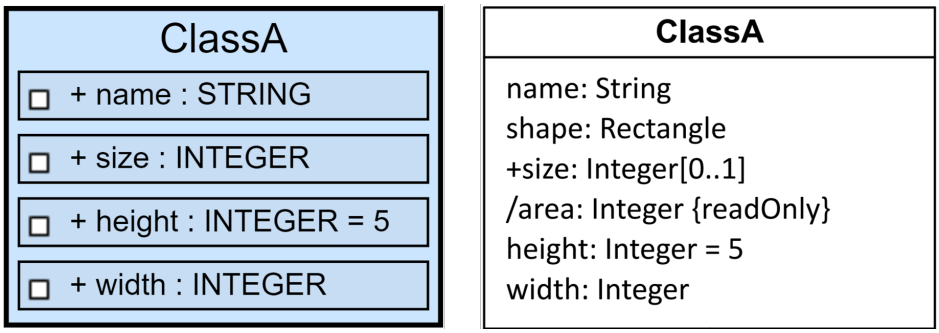



Figure 3.10: IML class (left) vs UML class (right). This image displays an example UML class from the OMG Unified Modeling Language specification [3].

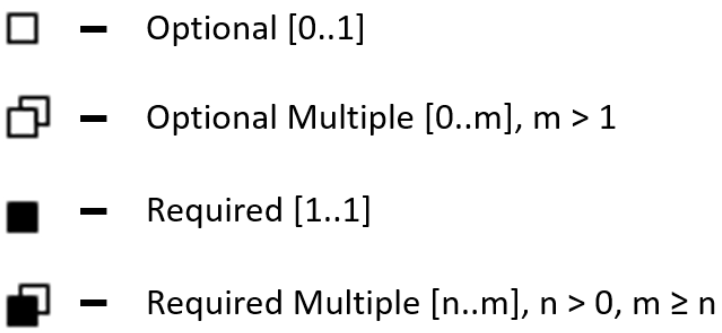



Figure 3.11: Key of visual bounds used in IML model editor.

Figure 3.10 shows a few attribute examples for both modeling languages. A difference between UML attributes and IML attributes is how bounds are represented. UML provides the notation, "[0..1]" to indicate an optional, singular attribute. IML uses a visual approach by displaying a single white box to indicate optional and singular. For other types of bounds, IML uses different combinations of white and black boxes to indicated optional vs required or singular vs multiple while UML would use different combinations of numbers. A key for IML bounding icons can be found in figure 3.11.

Other differences are displayed with the shape and area attributes in the UML class. Considering the shape attribute, it has a type of Rectangle. This type is not possible for IML attributes. As already stated, IML is a subset of UML and this is an example of a constraint related to this fact. IML attributes can only be one of the following types: String, Boolean, Double, or Integer. However, it is still possible to represent a class property of type Rectangle in IML. This would be accomplished by creating a new IML class called "Rectangle", introducing a relation from ClassA to Rectangle, and naming that relation "shape". This can be seen in figure 3.12. As a rule, only primitive variables can be class attributes in IML, while object variables within a class must be represented as relations. This is not a shortcoming of IML. UML can simply abstract away the Rectangle class because the diagram is only conceptual. Depending on the needs of the UML modeler, the Rectangle class may or may not have a concrete meaning. IML directly translates to code so Rectangle must be

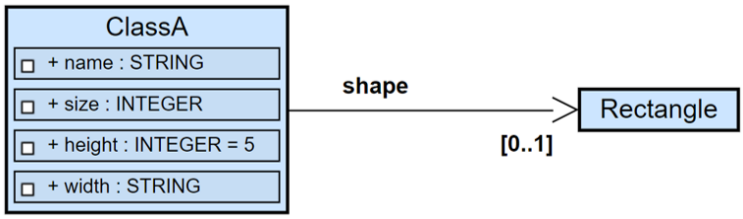



Figure 3.12: IML model displaying the ability to represent object attributes.

```
<td class="tg-xwyw">
    <img
        id="classIcon" height="80%" class="paletteIcon"
        draggeable="true" ondragstart="drag(this, event)" onclick="addClassModelToGraph()"
        src="Resources/ClassModelIcon.svg"
    >
    <br>
    Class
</td>
```

Figure 3.13: Palette icon snippet

explicitly defined as a class to properly generate the code for the system.

Now considering the area attribute of the UML class, the character '/' represents a "derived" property. According to the OMG UML specification, a derived attribute has a value that may be computed from other information in the program [3]. For example, area may be initialized by the multiplication of the properties height and width. This functionality is not currently possible in IML. While it may be implemented in the future, IML is an MDSE educational tool first. This means the included features focus on teaching MDSE concepts and derived attributes are not critical to this goal. Once IML has achieved more of its educational purposes, features like this may be considered in the future.

Next, the palette was implemented to let users choose which elements they want to add to the model. The palette is a simple implementation consisting of HTML elements that contain a picture to represent the model element and text to display the name of the model element. Each icon was given a unique id so the correct model element could be identified when users interact with the palette icons. The code for the class palette icon is displayed in figure 3.13 as an example. The id and CSS information is on line 3, the event handlers are on line 4 and the image source URI is on line 5.

When adding classes, users can simply click on the corresponding palette icon and it will be added to a default position in the model editor. Users can also click and drag the class icon to drop a class at the position of their cursor within the model editor. Adding attributes to the model follows a similar process. A big difference is an attribute must be embedded within a class, therefore, a class must be chosen when adding an attribute. Users can click and drag the attribute palette icon and drop it on the desired class. Users can also click on the attribute icon to trigger the "add-by-clicks" chain of events. This highlights the attribute icon and waits for the user to click on a class.

The dragging and dropping events are handled by HTML and JS. When a palette icon is dragged,

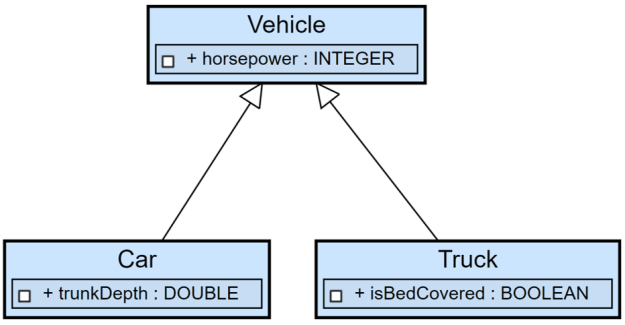



Figure 3.14: Class inheritance example

the browser triggers a drag event which is used to keep track of which HTML element is being dragged. Similarly, a drop event is triggered when the dragged element is dropped. This also identifies the HTML element that was dropped on. At first glance, using HTML elements to manipulate JointJS might pose a problem. However, as mentioned in the first paragraph of this section, the JointJS paper can act as an interface between the HTML document and the graph. When the dropped on element is identified, this can be passed to the paper method, "findView". This method then returns the corresponding graph cell (A graph "cell" refers to any element contained within the graph). Using this cell, it can be determined if the dropped on cell is an IML class, and subsequently, add an attribute to the IML class if determined valid.

Relations were briefly introduced in section 3.1 with figure 3.5 displaying them side-by-side. Adding their palette icons followed the same process as the icons for class and attribute. The process for adding a relation to a model is similar to attributes, but more involved. To make a valid relation, one prerequisite is the relation must have a source and target class. Thus, when adding a relation, we must have an event chain similar to adding attributes. This chain should allow the user to choose which relation to add, then pick its source and target classes. Just like attributes, this can be accomplished in two ways. First, relations can be added by dragging a relation palette icon and dropping it on a class. The dropped on class becomes the source for the relation. Then, unlike attributes, another class must be chosen to be the target of the relation. While choosing the target class, the relation is attached to, and moves with, the cursor until the event is finished or cancelled. This helps the user know there is more to be done with the add relation event. The second method of adding relations follows the "add-by-clicks" approach. The process is exactly the same as dragging and dropping except a relation is chosen and added to the source class by first clicking the relation icon and then clicking the source class. The functionality for choosing the target class is the same for the remainder of the event chain.

Relations also proposed a difficultly that classes and attributes did not. As stated before, the IML model editor is an MDSE editor, not a drawing tool. This means the model being created corresponds to valid code. Therefore when metamodeling, the metamodel must remain valid to the IML meta-metamodel at all times, or else the code generation would produce a system that can't be compiled. Relations are the first model element that could be used to produce an invalid system. Take the inheritance relation as an example. This relation is used to produce hierarchies of classes. Looking at figure 3.14 Car and Truck are separate classes that represent a unique classification of

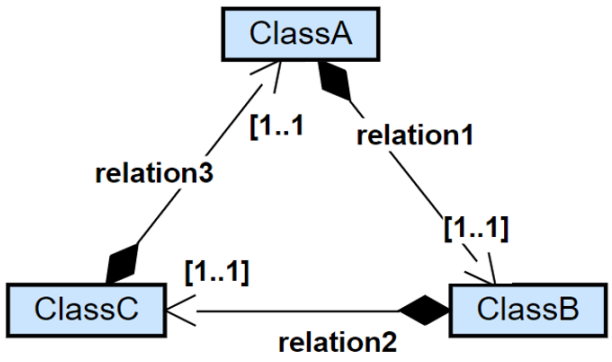



Figure 3.15: Example of a relation cycle. This figure was cropped because it is impossible to create a model of this type in IML.

objects. However, they are also types of vehicles. This can be represented by creating the Vehicle class, adding the attributes Car and Truck share to vehicle and establishing inheritance relations between each class and Vehicle. This concept is a core tenant of Object-Oriented Programming (OOP). The problems lies with the free-form nature of modeling. What is stopping the user from creating a cycle of inheritance between these three classes? If the modeler is using a drawing tool, the answer is nothing is stopping them. To a Java compiler, if a cycle (like the one found in figure 3.15) was found, it would produce a compiler error warning about an inconsistent hierarchy. Therefore, the IML model editor must check for invalid models as relations are being added for all relations.

To check for relation cycles, graph traversal algorithms were created using recursion. Stated generally, the source of the relation was added to an array that keeps track of visited classes. Then, the target of the relation is passed to a recursive method along with the visited array. This method would check if the target class has already been added to the visited array. If it has, then a cycle has been discovered. If not, then follow every outbound relation of the class. Repeat this process for every outbound relation until a cycle is found or the outbound relations have been exhausted. The pseudocode for this can be found in figure 3.16.

Relations present another code generation problem. Specifically, the inheritance relation introduces the concept of attribute overloading. This means, if two or more classes in the same inheritance hierarchy have attributes with the same name, then Java has to decide which address location to use for a value. While Java has a defined method for making this decision, it can be confusing, especially for new developers. With IML being primarily a teaching tool, focused on MDSE, the decision was made to disallow attribute overloading. This way, when users create a model and generate its code, the code is easier to understand and clearly maps to the visual model they created. Users can spend more time understanding MDSE and code generation and less time figuring out the Java compiler. When working on a model, if a user attempts to implement attribute overloading, their current action is cancelled and an error message is displayed explaining the cancellation.

While a user is building their model, a concurrent process is taking place in the back end. An object-oriented (OO) data structure is being implemented to represent their IML structural model.

```
function findCircle(target, visited) {
    if visited contains target {
        return true
    }

    visited.push(target)

    foundCircle = false
    foreach outboundRelation in target.outboundRelations {
        foundCircle = checkCircular(outboundRelation.target, visited)
    }

    return foundCircle
}
```

Figure 3.16: Find relation cycle pseudocode

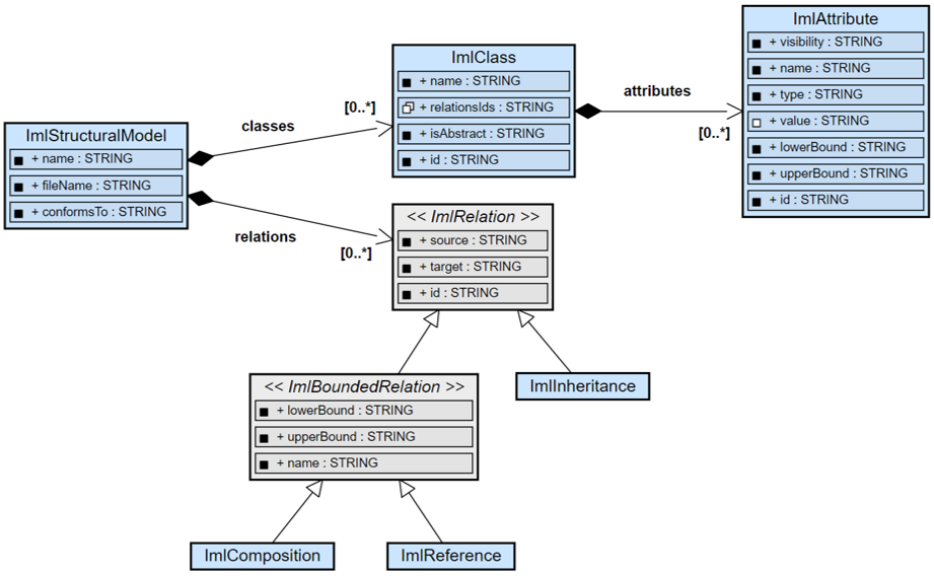



Figure 3.17: Iml Structural Model, structural model

Each class of the data structure was written in JS. JS classes follow similar class structures to Java and other OO languages. The IML metamodel class diagram of this system can be seen in figure 3.17. In simple terms, an IML Structural Model is composed of a set of class and relation objects. A class is then made of attribute objects. Additionally, each object has various attributes that help define the object. For example, the ImlStructuralModel class has 3 other primitive attributes: name, fileName and conformsTo. Another interesting aspect of this data structure is the inheritance hierarchy that exists with relations. In its most basic sense, a relation is defined by its source and target classes. However, more complicated relations exist like the ImlBoundedRelation. These relations have bounds and a name. They are used to defined attributes within a class that are made of objects, as opposed to, attributes that are primitives like an integer or boolean.

This concurrent back end process was implemented to enable model transformations and importing and exporting. Unpacking an ImlStructuralModel object was made easy by simply iterating over the class and relation arrays. Relation objects granted type enforcement and the use of "instanceof" to determine the current relation type. With primitive attributes attached to each object, finding values when they were needed was straightforward. To update the back end data structures, the properties table was implemented. The implementation process of this feature is explained in the next section.

## 3.5 Properties Table

In the IML structural model editor, the properties table was included to edit element attributes. Attribute in this context refers to properties that are intrinsic to an IML element. This is different from the attribute of a class. For example, a relation has at least two intrinsic attributes, source class and target class. These attributes make the element a relation and must be kept track of in the IML back end. An attribute of an IML class simply represents a variable in code. The properties table allows users to edit those intrinsic attributes and the table changes depending on context. When a class is selected, the list of class properties, and their current values, are displayed in the table. These values can be changed by clicking on the value to enter an edit mode. Any changes can then be saved by pressing the enter key, or changing the page focus. Likewise, if an IML attribute is selected, then the properties table updates to show the properties of the selected attribute. This also holds true for each relation element. If nothing is selected, then the attributes of the structural model are displayed. These properties can be edited as well. An example of the properties table is shown in figure 3.18. This table has the context of IML attribute "name", from figure 3.3.

To build the properties table dynamically, an HTML table is generating using the element selected, and the back end data structure that corresponds to that element. A loop is run around the element attributes which creates a row for every iteration. A row consists of two cells, the property name (left cell) and the property value (right cell). The cells in the left column are not editable. This is indicated by the grey background. The cells in the right column are editable. This is indicated by the white background. Again considering figure 3.18, if the empty cell next to the Value cell is clicked, edit mode will be triggered for the Value attribute. Edit mode creates a text box within the empty cell that can be used to enter a value. If the cell was not empty, the current value would appear in the text box. This allows the user to preserve the current value if they did not intend to

| Properties Table | |
|---|---|
| **Property** | **Value** |
| Visibility | PUBLIC |
| Name | name |
| Type | STRING |
| Lower Bound | 0 |
| Upper Bound | 1 |
| Value | |

Figure 3.18: Example properties table. The name attribute of the person class from figure 3.3 is selected.

change it. Figure 3.19 shows the properties table in edit mode for the Value attribute.

Figure 3.19 also shows one of two special cases for the edit mode. A button containing an ellipse can be seen to the right of the text box. Clicking this button produces a popup with a text box. Because attribute values could contain arrays, this popup gives the user an easy method for typing the array. Figure 3.20 shows the popup with some values. Users can enter their array values by comma separation. If a user was to use the text box in the properties table, they would have to include square brackets around their comma separated array values. The other edit mode special case can be seen in figure 3.21. This displays the edit mode for Upper Bound. Instead of a text box, Upper Box uses a number input box. This input only accepts integer values and can be incremented or decremented. The number input box is also used for the Lower Bound edit mode. The special case can be seen with the button containing the asterisk. The upper bound is special because it can represent the "many" portion of a relation. This means, the upper bound can be set so that an array can contain as many attributes as the user desires. If the lower bound is set to one and the upper bound is set to asterisk, this would be described as, "1-to-many". The attribute must have at least one value, but there is no limit beyond that. When clicked, the many button exits edit mode for Upper Bound and set its value to '*'. The many button does not exist for Lower Bound.

The final notable aspect of the properties table is the edit mode for Type, Visibility and other enumerated attributes. When the user enters edit mode for these properties, a drop down is created instead of a text or number input box. This drop down contains all possible values for that attribute. As stated earlier, an IML attribute can only be one of four types, BOOLEAN, DOUBLE, INTEGER and STRING. Subsequently, these are the only types included in the drop down and can be seen in figure 3.22. Every time these attributes are updated, the back end data structures are updated to match. Once a user is satisfied with the model they created, they can export their model into a '.iml' file. This file can be used later to import back into the model editor. The exporting and importing implementation is explained in the following section.

Properties Table

| Property | Value |
|---|---|
| Visibility | PUBLIC |
| Name | name |
| Type | STRING |
| Lower Bound | 0 |
| Upper Bound | 1 |
| Value | ... |

Figure 3.19: Example properties table in edit mode. The name attribute of the person class from figure 3.3 is selected.

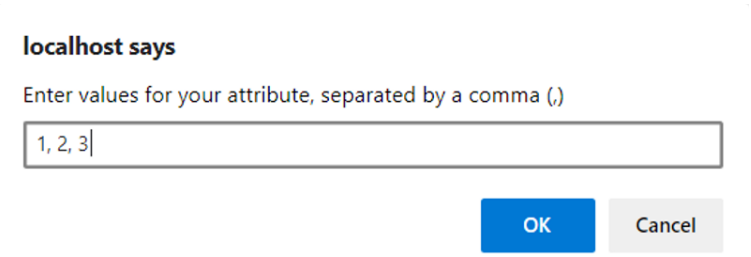



Figure 3.20: Attribute popup

Properties Table

| Property | Value |
|---|---|
| Visibility | PUBLIC |
| Name | name |
| Type | STRING |
| Lower Bound | 0 |
| Upper Bound | 1 * |
| Value | |

Figure 3.21: Upper bound edit mode

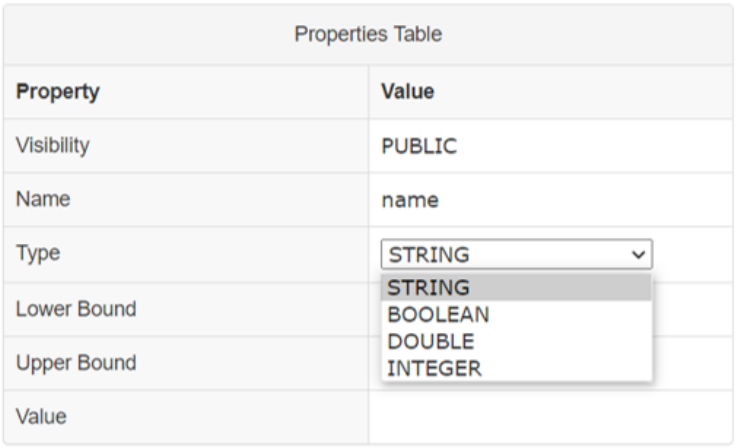



Figure 3.22: Type edit mode

## 3.6 Export and Import

Exporting and importing model files was an essential feature of the model editor. Without the ability to export and import a model, implementing the M2M transformations would not be possible. Models are exported into the ".iml" file format. This is essentially custom XML. Proposed in the paper, *IML: Towards an Instructional Modeling Language* by Rapos et. al. [36], it is stated that XML is widely used and accepted in many current modeling tools making it well suited for transitioning students into more complex applications. Furthermore, XML is widely used in web applications allowing us to leverage parsers for importing. XML syntax contains tags, denoted by the characters '<' and '>'. Between those characters is the name of the tag, followed by the attributes of the tag. Each attribute has a name that comes before the attributes value. The value is surrounded in quotation marks and is separated from the name by the character '='.

A tag can have either an opening tag or both an opening and closing tag. The opening tag is used by itself when it is only needed to represent one object with attributes. A set of opening and closing tags is used when an object contains other objects. The closing tag also uses the opening and closing characters '<' and '>'. This tag contains the name from the opening tag, preceded by the character '/'. When an object consists of a single opening tag, the opening tag is ended with '/>' instead of simply '>'. Finally, when a tag pair has sub-tags, style convention has that the sub-tag should be one tab to the right of the parent tag. While this is not necessary for parsing, it makes the file more readable. Readability in all areas is essential to IML and thus, the style convention is followed in a .iml file.

An IML metamodel can be seen in figure 3.23 and the corresponding .iml file is shown in figure 3.24. Similar to the back end data structures, the structural model file representation consists of a set of classes and a set of relations. In the file there is also an iml tag pair outside of the StructuralModel tag pair. This is for future versions of IML which will include corresponding behavioral models.

Referring back to the StructuralModel tag, there is a set of attributes that exists in the opening tag. These attributes correspond to the primitive attributes found in the ImlStructuralModel JS object. The sub-tags, Classes and Relations, correspond to the classes and relations data structures

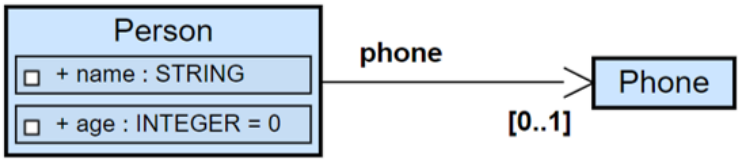



Figure 3.23: IML model representing a Person class with two attributes name and age. This model also includes an empty class Phone and a relation from Person to Phone.

```
<iml version="0.1">
    <StructuralModel name="PersonAndPhone" conformsTo="IML Definition" routingMode="simpleRoute">

        <Classes>
            <Class name="Person" isAbstract="FALSE" x="278" y="199" id="1">
                <Attribute visibility="PUBLIC" name="name" type="STRING" lowerBound="0" upperBound="1" position="1" />
                <Attribute visibility="PUBLIC" name="age" type="INTEGER" value="0" lowerBound="0" upperBound="1" position="2" />
            </Class>
            <Class name="Phone" isAbstract="FALSE" x="575" y="212" id="2" />
        </Classes>

        <Relations>
            <Relation source="1" destination="2" type="REFERENCE" name="phone" lowerBound="0" upperBound="1" />
        </Relations>

    </StructuralModel>
</iml>
```

Figure 3.24: The XML file that corresponds to the model found in figure 3.23

also found within the ImlStructuralModel JS object. This pattern is the same for all IML elements represented in the .iml file. Each attribute saved through the back end data structures is saved through XML attributes or sub-tags. A unique aspect of the saved data can be found with the source and destination attributes of relations. These are the id's of the corresponding classes. Class elements are loaded first and then are identified and connected using the id's saved in the relations. Normally, class id's are a 16-bit unique id but they have been changed in figure 3.24 for simplicity. Note, metadata related specifically to the model editor, such as element positions, has been omitted from the XML.

To generate the .iml file, each tag is created by concatenating strings and object attributes and outputting this to a file. The iml and StructuralModel tags are created first, outputting their primitive attributes according to XML syntax. Then, two loops are run around the classes and relations data structures, respectively. Each class and relation are outputted using the same process. One difference exists for classes that have IML attributes. If a class has one or more attributes, then the tag is converted from a single opening tag to a pair of opening and closing tags. The attributes are then generated and outputted within the corresponding class tags.

Importing a .iml file works in the same order. To accomplish this we took advantage of an XML parser. JQuery has a builtin XML parser that was used to read in the .iml files. Again, starting with the classes, each class and its attributes are read in and generated in the back end and in JointJS. After the classes are in memory, the relations can be read in and connected between their respective classes. One difference between importing and exporting is importing must consider the possibility of receiving either a metamodel or instance model. If a metamodel is uploaded, there is nothing to consider besides being a valid .iml file. The complexity lies with importing instance models. When an instance model is uploaded, its metamodel must be uploaded as well. This is needed to ensure instance model conformance, as well as, generate the palette for adding new elements to the imported instance. After an instance model is uploaded, the user is prompted to upload a specific

metamodel. This metamodel is identified to the model editor by looking at the instance models "conformsTo" attribute. This attribute contains the name of the metamodel file that the instance model conforms to. The user is prompted to upload this specific file, however, the filename is not the only check made for conformance. The instance model is validated to conform by the checkConformance method. This is necessary because a metamodel file could have the same name as the correct metamodel file, but contain a completely different metamodel.

This chapter detailed the implementation of the structural model editor. The model editor was determined necessary for implementing structural model transformations because users needed a simple and intuitive method for creating models to transform. The model editor was implemented using web technologies HTML, CSS and JS, with support from packages like JointJS, JQuery and Bootstrap. Using models from the web editor, users can experience some of the most important benefits of MDSE. One of these benefits is code generation. The following chapter provides an in-depth look at implementation of IML code generation.

# Chapter 4

# Code Generation

Model-to-Text (M2T) transformation is the general term for converting graphical software models into some textual representation. There is a subset of this process in MDSE that is particularly useful. This subset is known as code generation. Using the backend JS classes from the model editor, we were able to implement code generation for IML. Currently, IML can generate Java code from any metamodel and any conforming instance models. Java was chosen as the first target language due to its general popularity, as well as its prevalence in the modeling community thanks to Eclipse based MDSE tools. Code generation for IML can be broken into two processes, the generation process for metamodels, and the generation process for instance models. Generating metamodel code is explained first, followed by instance model generation.

## 4.1 Metamodel Code Generation

The work completed in this project implemented a structural model editor. This means the code generated by IML in the model editor is structural code. Structural code is related to data and *defining* program states, while behavioral code would *alter* data and the program state. This means most of the generated code is Java class files. More specifically, the generated code defines the state of an object. This could be an attribute, its type and initial value. The metamodel code generator does not generate methods besides getters, setters, constructors and printers. Any other methods would be considered behavioral and would need a behavioral modeling tool to properly represent.

To generate the structural Java code, the ImlStructuralModel class, mentioned in chapter 3, is used to iterate over the model elements. The code generation engine iterates over each class element and for each class element, it loops through the attribute and relation elements. As information is discovered about a class and related elements, the corresponding code is appended to a string. Starting with a simple example, figure 4.1 shows an IML metamodel with a single abstract class. When transformed, the engine generates an empty abstract Java class with the name “EmptyClass”. The generator knows the class is abstract by checking the ImlClass object’s isAbstract element. If true, then“abstract” is inserted in the class code string.

The abstract class example illustrates a design pattern used throughout code generation. There are many instances of optional code, like the abstract modifier, where a variable is always created


Figure 4.1: Empty abstract class

```
1   var abstract = ' ';
2   if(imlClass.isAbstract) {
3       abstract = ' abstract ';
4   }
5
6   ...
7
8   // Class header line
9   classCode += 'public' + abstract + 'class ' + imlClass.name + extendsString + '{\n\n';
```

Figure 4.2: Abstract class code snippet

for appending regardless of the abstract status of a class. The variable is initialized to a default value that is needed in the code string. This may be a space or an attribute value that could be edited later. Then depending on element values, the default string is updated to reflect what was found. Figure 4.2 displays a code snippet from the generation engine. On line 1, the variable "abstract" is initialized to a string with a whitespace character. This variable is used later to assemble the class header. Line 2 evaluates the current classes isAbstract attribute. If isAbstract is true, the "abstract" variable is reset to a string value that prints the abstract modifier to the class code. Line 9 shows the "abstract" variable in use. If isAbstract is false, then the needed space is added between the two Java keywords. If isAbstract is true, then the abstract modifier is correctly added between the keywords. The same process is followed for the "extendsString" variable.

Next, the class attributes are generated. Attributes can come in two forms, primitive and object. Primitive attributes are found in the "attributes" property of an ImlClass, while object attributes are represented by ImlBoundedRelations, found in the "relations" property of an ImlClass. Primitive attributes are generated first, then relation attributes. The primitive attributes are generated first because they have a simpler implementation. Once the primitive attributes were generating properly, the relation attributes were implemented.

Each primitive attribute is found by iterating over the "attributes" class property. To generate the code for a primitive attribute, all necessary information is gathered from the ImlAttribute object. This includes the attribute: type, visibility, name, lower bound and upper bound. The attribute value is not obtained yet because some evaluation of the previously mentioned information needs to be completed to print the value according to Java standards.

The first consideration is the lower bounds. This information tells the engine if the attribute is optional or required. A lower bound of 0 indicates the attribute is optional. This is important because an optional attribute remains uninitialized unless a metamodel default is provided. If the lower bound is 1 or more, then the attribute is required and the type default may be considered. There are two types of defaults in IML code generation, metamodel and type. Metamodel defaults can be assigned to initialize attributes and takes precedent over a type default. If there is no value provided by a metamodel default, the type default must be used for a required attribute. In any case, if the lower bound is greater than 0, the corresponding attribute must be initialized with some value.

Next, the engine must look at the upper bound. A value of 1 means the attribute is a single value while a value of 2 or more indicates the attribute is an array. The upper bound value cannot

be less than 1. In the case of a single value, the attribute value is determined by the lower bound and metamodel default, as mentioned in the previous paragraph. However, if the upper bound is greater than 1, then an array has been discovered. To translate an array attribute into Java, many things must occur. To start, a flag must be flipped so the Array package imports are printed to the class later in generation. Next, string variables must be edited and keywords must be used. For example, the string variable saving the attribute type must be updated. A type value of "Integer" must now become "ArrayList<Integer>". Then, the values of the array (if provided, or required) must be passed into the "Arrays.asList()" method to create an ArrayList object of those values. The final result of a required array may look like:

ArrayList<Integer> arr = Arrays.asList(...array values...);

The last piece of information to consider is the type of the attribute. This information affects how the value appears in Java code because the Java compiler needs denotations to determine proper type value syntax. For example, boolean values must be "true" or "false" exactly, string values must be surrounded in double quotes or double values must include a decimal point. Furthermore, the type information is used for required attributes that have not been assigned a value or do not have a metamodel default value. The type default value must be used in this case. The code generation engine uses,"Enter a value", false, 0 and 0.0 for String, Boolean, Integer and Double type defaults, respectively. A generated class with a few different attributes is provided for example in figure 4.3.

The string default, "Enter a value", was used because empty string was already being used to indicate no value. When attribute values are updated in the model editor, they are passed through an HTML text input element. This element only returns strings of what was entered. To allow users to delete values and let attributes remain uninitialized, it was necessary to process an empty string from the text input as no value. Considering there is no data lost when converting from an uninitialized string to an empty string, this implementation did not pose a threat to the model editor. However, this did present a problem with model transformation. When converting an attribute of type double, integer or boolean, the resulting value may be a default value. Default values are different from uninitialized attributes. If a target string attribute has a lower bound greater than 0, then this attribute needs a value and cannot be uninitialized. Therefore, a new default string value was created for IML. This allowed the distinction between an uninitialized string and a string set to a default in IML.

Similar to primitive attributes, each object attribute is found by iterating over the "relations" class property. Object and primitive attributes are used for similar reasons in Java classes, therefore, their implementation in Java syntax is similar. The translation of object attribute bounds follows the same algorithm as primitive object bounds. The two types differ when printing the attribute values. To create an object in Java, the "new" keyword must be used in conjunction with a constructor from the desired class. Just like primitive attributes, an object attribute must be initialized if its lower bound is greater than 0. However, there is a small difference if the object attribute is optional. There is no way to provide a metamodel default value for an object attribute so an optional object attribute is always uninitialized. Just like primitive attributes, if an object attribute is an array, then "Arrays.asList" is employed and the same changes are made to the attribute type and array packages. Again, the difference in values can be seen here as the asList method contains a set of "new" objects. Another generated class with examples of object attributes is shown in fig 4.4.

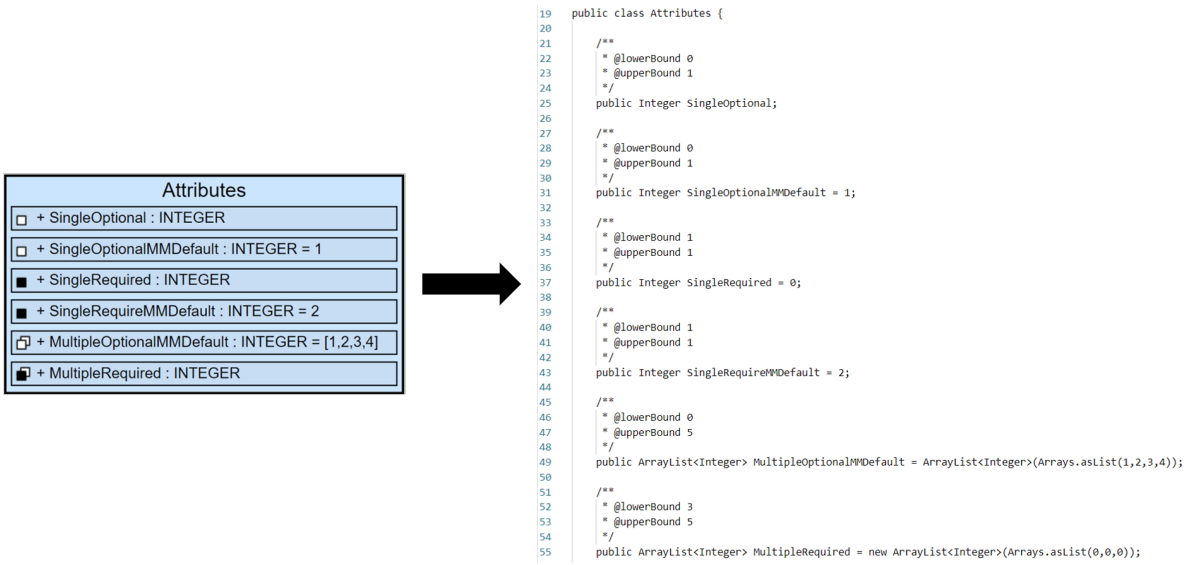


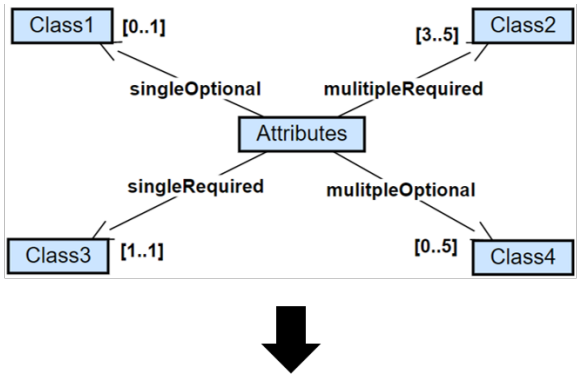

Figure 4.3: Primitive attribute generation examples

After the attribute value is decided for either attribute type, the whole line of attribute code is put together. Using the visibility, type, name and value variables, the entire line is generated. Examples of this can be seen on lines 25, 31, 37, 43, 49 and 55 of figure 4.3 or lines 23, 29, 35 and 41 of figure 4.4. Preceding these lines are comments referencing the lower and upper bounds of the attribute. These comments are included as a part of IML's efforts to generate clear and readable code. Users can more easily connect lines of code to model elements using this extra information. To generate the comments, the lower and upper bound variables from the object are inserted into strings. Each string is then combined to create the entire comment. This final comment string is then prepended to the line of attribute code and added to the class code.

Attribute initialization code is not the only code generated during the attribute and relation iterations. The body of the constructor and the getters and setters are also generated. A constructor in Java is a special, class method that initializes an object's attributes when the object is created. More information on Java constructors can be found in Oracle's tutorials and documentation [42]. Getters and setters are also class methods. These methods are supposed to be used as the main process for getting and setting attribute values after an object has been initialized.

To generate the constructor body, there is a string variable assigned before looping through both attribute types. Every time an attribute is discovered and after the attribute value has been created, the name of the attribute is used to generate the corresponding constructor line of code. This line is then appended to the end of the constructor body string. After iterating through the primitive and object attributes, the constructor body string is used with other variables to generate the entire constructor. An example constructor can be seen in figure 4.5.

Getters and setters are generated in a similar manner to the constructor body. After the attribute code is created, a string variable for the getter is created, followed by a string variable for the setter. Using the attribute's type and name, the entire method for both the getter and setter can be generated. In addition to the printers and constructor, this is the only time structural models




```java
public class Attributes {

    /**
     * @lowerBound 0
     * @upperBound 1
     */
    public Class1 singleOptional;

    /**
     * @lowerBound 1
     * @upperBound 1
     */
    public Class3 singleRequired = new Class3();

    /**
     * @lowerBound 3
     * @upperBound 5
     */
    public ArrayList<Class2> mulitipleRequired = new ArrayList<Class2> (Arrays.asList(new Class2(), new Class2(), new Class2()));

    /**
     * @lowerBound 0
     * @upperBound 5
     */
    public ArrayList<Class4> mulitpleOptional;
```

Figure 4.4: Object attribute generation examples

```java
public Attributes(Class1 singleOptional, Class3 singleRequired, ArrayList<Class2> mulitipleRequired, ArrayList<Class4> mulitpleOptional) {
    super();
    this.singleOptional = singleOptional;
    this.singleRequired = singleRequired;
    this.mulitipleRequired = mulitipleRequired;
    this.mulitpleOptional = mulitpleOptional;
}
```

Figure 4.5: Constructor generation example

```
Attributes: {
    primitiveAttr1: "defaultString"
    primitiveAttr2: 1.23
    primitiveAttr3: [1, 2, 3]
    objectAttr1: [
        ClassA: {
        }
    ]
    objectAttr2: [
        ClassB: {
        }
        ClassB: {
        }
        ClassB: {
        }
    ]
}
```

Figure 4.6: Pretty print for an object with primitive and object attributes.

can be used to fill out the body of a method. Because getters and setters are a common design pattern in OOP, the functionality of the methods can be assumed and created with minimal attribute information. Additionally, the JavaDoc comments of the getter and setter methods are created. Again, these are included to create clear and readable code for IML users. After a getter or setter is fully compiled for one attribute, it is appended to a string variable that keeps track of all getters or setters for a class. Similar to the constructor body, these variables are appended to the class code string later on. This was done so all getters could be grouped together towards the end of the class. This same reasoning is used for the setters.

As mentioned in the previous paragraph there is another set of methods that are generated in a class. The toString and prettyPrint methods can be completely generated with a structural model. These methods can be created because they are unlike most methods. They are intended to retrieve and display the state of an object. Because they are use for displaying state, instead of changing state, they can also be completely generated by structural code. Classes often include a toString but do not include a prettyPrint method. This was included to help users. Having the ability to see the state of an object allows the user to understand the connections between their model and its code. Implementing the prettyPrint method required the use of recursion to keep track of indention level. Every time an object attribute or an array is found, the tab level needed to be increased by 1. Creating this code follows a similar algorithm to attribute generation because it iterates over each primitive and object attribute again. An example of output from the prettyPrint method can be seen in figure 4.6.

Finally, the package statement is prepended to the class code using the name of the structural model and the Java package syntax. Then, an ASCII art comment is created for the class and prepended to the class code. The comment is created by running through the primitive and object attributes again. For each attribute, the bounds, visibility, name and type is encoded into one line. An example of object attributes in the ASCII art can be seen in figure 4.7. The ASCII art is included to continually reinforce the classes connection with the model. This comment adds another way for

```
1   /*
2    * ========================================
3    * ||              Attributes            ||
4    * ========================================
5    * || [0..1] + singleOptional : Class1    ||
6    * || [1..1] + singleRequired : Class3    ||
7    * || [3..5] + mulitipleRequired : Class2 ||
8    * || [0..5] + mulitpleOptional : Class4  ||
9    * ========================================
10   */
```

Figure 4.7: ASCII art for class code in figure 4.4.

the user to understand how the code and models work together. The ASCII art adds value because it is a condensed representation of the code. The user should be able to look at the corresponding model class element and find the similarities using the ASCII art.

After adding the ASCII art, the class code is ready to be added to the Java project. Using a JS package called JSZip, a file structure of the Java project package is created. The newly finished file is added at the lowest level of the file structure. Eventually, each class will be added to this file structure. Once all classes are added, the Java project will be zipped up and output to the user through the browser downloads. One exception to this is if the user is generating an instance model. In this case, the code for the instance model must be generated in addition to the metamodel code. Generating instance model code is explained in the next section.

## 4.2 Instance Model Code Generation

At the end of the metamodel code generation method, there is a call to a method that generates instance model Java code, if an instance model exists. When metamodeling and generating code, the instance model code generation method does not run because there is no instance model to generate. Users do have the option to model using their own metamodels. If a model is created with a user-defined metamodel, then an instance model exists and the code for it must be created when generating code. This means that every time an instance model is generated, so are the classes from the corresponding metamodel. It is necessary to generate these classes every time because the instance model relies on implementing objects defined by those class files.

When generating the code for a structural instance model, there is only one file that needs to be created. This is a class file consisting of a main method and Java statements that initialize, get and set object's and their values. The code is broken into 3 sections, object instantiation, attribute assignment and relation implementation.

To instantiate objects as variables, Java requires an object type, name and call to one of the object's constructors. A statement like this is needed for each object that is to be created. The code for this is generated by iterating over the classes attribute found in the ImlStructuralModel object. The object type is the class itself, therefore, the name of the class is used as the type. The

```
9   // Instantiate model objects.
10  Attributes attributes1 = new Attributes();
11  ClassA classa1 = new ClassA();
12  ClassB classb1 = new ClassB();
13  ClassB classb2 = new ClassB();
14  ClassB classb3 = new ClassB();
15
16
17  // Set object attributes.
18  attributes1.setPrimitiveAttr1("defaultString");
19  attributes1.setPrimitiveAttr2(Double.valueOf(1.23));
20  attributes1.setPrimitiveAttr3(new ArrayList<Integer>(Arrays.asList(1,2,3)));
21  attributes1.setObjectAttr2(new ArrayList<ClassB>());
22
23
24  // Implement relations between model objects.
25  attributes1.getObjectAttr2().add(classb1);
26  attributes1.getObjectAttr2().add(classb3);
27  attributes1.getObjectAttr2().add(classb2);
28  attributes1.setObjectAttr1(classa1);
```

Figure 4.8: Generated instance code

name of an object is decided by the name of the class and a counter. The class name is sent to lowercase and then inserted into a dictionary with the key being the name of the class. The value corresponding to each key is an integer counting the number of object seen of that type. This integer value is then appended to the lowercase version of the class name and becomes the name of the current object. Finally, the empty constructor of the class is used, as the object's attributes are set later. Furthermore, this allows any default values set in the class to be used, if desired. The call to the constructor is created by concatenating the name of the class with a string of open and closed parenthesis. The code for a generated instance model can be seen in figure 4.8, on lines 10-14.

Next, each object's attributes must be set according to the values found in the instance model. To do this, each class in the instance model is iterated over again using the same classes attribute from object instantiation. Object attributes were not set as they were instantiated because we decided the instance code would be more readable if the statements were grouped by similar actions. Readability is of high value to the IML project, so the extra effort needed to generate the code in this way was deemed worth the time. To keep track of which object name was assigned for a specific class, another map was employed during object instantiation. This map is named "declaredInstances" and has the IML class id as the key and the object name as the value. During attribute assignment the instance class could be used to find its corresponding object name. Next, the attributes property of the instance class is iterated over. This loop is used to set the primitive attributes. For each attribute, the correct value is retrieved from the IML attribute object. If a value is provided, this is used. If there is no value provided, the metamodel default is used. If there is still no value, then the type default value is used. Some checking of bounds and type are performed for the same reasons as in metamodel code generation. If arrays are present, then the list of val-

ues must be formatted properly and the arrays packages need to be imported. The type must also be checked to ensure proper formatting. Next, the object attributes for an instance class must be assigned by iterating over IML class relations variable. Like primitive attributes, object attributes use setters to assign the attribute. The difference is the setter must be passed a "new" object like in object instantiation. This requires the "new" keyword to be used with a class constructor for the object type. An example of primitive and object attribute assignment can be seen in figure 4.8, lines 18-21.

Finally, the relations between each object must be created. This part of instance code generation uses the getters and setters to retrieve and set attributes with the objects assigned in object instantiation. This is done by iterating over the classes, and for each class, their relations. We find relations by looking at the relation's source. If the source of the relation is the current instance class, then that relation must be implemented as an object attribute of the current class. Once a relation for a class is found, the bounds of that relation need to be checked. The bounds may changed how an object is related to another object. If the relation has an upper bound of 1, then the attribute can be set using only the setter for that attribute. If the upper bound is greater than 1, then the relation is an array. To add an object to a relation that is an array, the attribute for that relation must first be retrieved by the attribute's getter and then the ArrayList "add" method must be used to insert an object into the relation array. The "declaredInstances" map is used again in this algorithm to find the object variable names that correspond with the current instance class and the related instance class. These object names are then concatenated into a string with the relation attribute name and the proper Java statement is created. An example of relation assignments can be seen in figure 4.8, lines 25-28.

This chapter explains the process for generating Java code for an IML metamodel and instance model. To generate code, an algorithm traverses the ImlStructuralModel object and looks at each class, attribute and relation element. The algorithm uses the defined set of rules for generating Java code from IML models and constructs the corresponding Java class files. If an instance model is present, then instance model classes are applied to make Java objects based on the code generated by the metamodel. As discussed in the beginning of this chapter, code generation is a subset of M2T transformations. M2T transformations are not the only type transformation implemented in this project. The following chapter explains how M2M transformations are implemented in IML.

# Chapter 5

# Model-to-Model Transformations

A Model-to-Model (M2M) transformation is the process of defining transformation rules between two metamodels and then applying these rules to an input instance model to produce an output instance model. More specifically, the input instance model must conform to one of the metamodels. If it does, then the output instance model will conform to the target metamodel, assuming the transformation rules are valid. Using the models created by the web-based model editor, we were able to implement this process in IML. The implementation of this process can be broken into 3 parts, creating the model transformations page, implementing the ability to define transformation rules, and creating the transformation engine. Each of these steps are described in the following sections.

## 5.1 Model Transformations Page

M2M transformations are similar to code generation because both processes accept a model created in one domain and transform it into a model of another domain. In the case of IML code generation, the first domain is an IML model and the second domain is Java code. For IML M2M transformations, both domains are IML models. Furthermore, they are user-defined IML models. This introduces a new problem that was not present with code generation. During code generation, the second domain is constant. The model is always transformed into Java, or some other predefined programming language. This means that transforming from IML to Java always follow the same rules. In the case of M2M transformations, the two domains can change. This means the transformation rules between the domains do not always follow the same rules. These new rules must be defined by the user. For this reason, the model transformations page was implemented in the IML project. The model transformations page allows users to upload two metamodels, define the transformation rules between them and apply those rules to instance models.

The structure of the model transformation page follows a similar visual pattern to the diagram presented in Chapter 2 figure 2.2. This was done intentionally to show the user the flow of the model transformation process as well as draw parallels with conceptual materials and our realization. The final UI is shown in figure 5.1. This image displays the UI when a user first opens the web page. Conversely, figure 5.2 displays how the UI changes after all models have been uploaded, all rules are defined and after a successful transformation has been applied. This UI was implemented using the same technologies used for the model editor (HTML, JavaScript, JQuery, Bootstrap 3, JointJS).

The HTML document for this page is broken into two columns using Bootstrap. One column for defining the model transformation and another column for displaying the source and target metamodels. The column for creating a model transformation is broken down further into 3 rows.

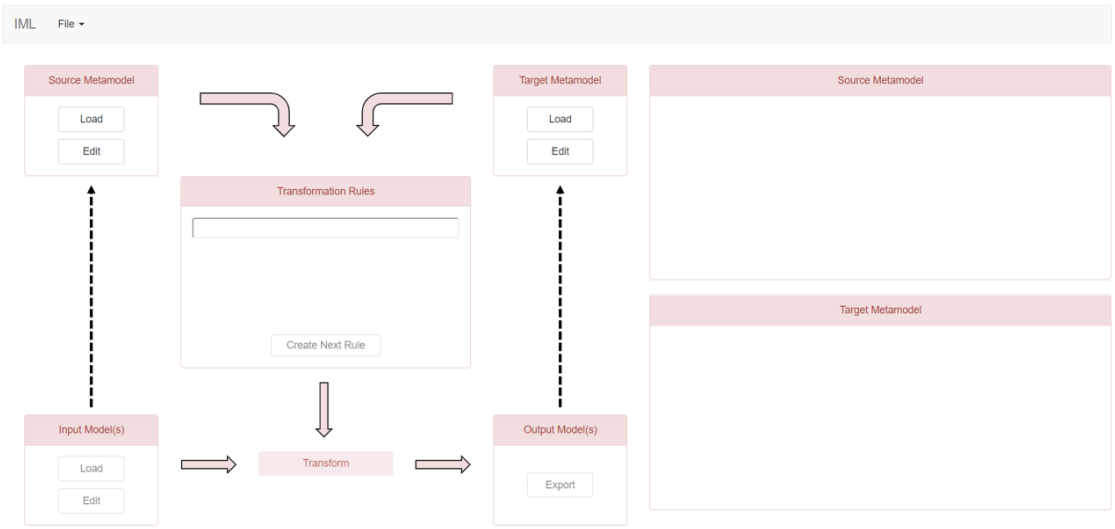



Figure 5.1: Model transformation UI before user interaction

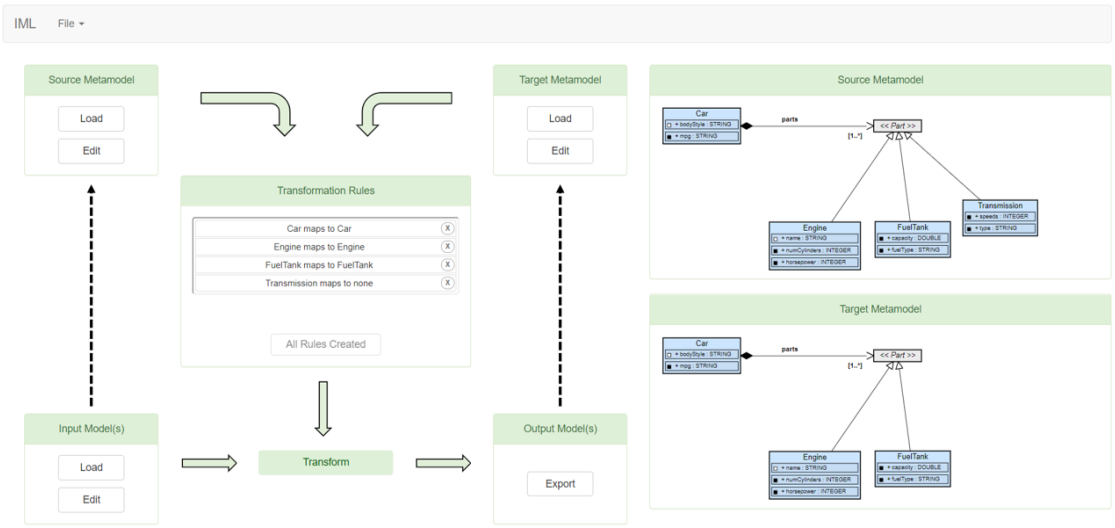



Figure 5.2: Model transformation UI after completed transformation

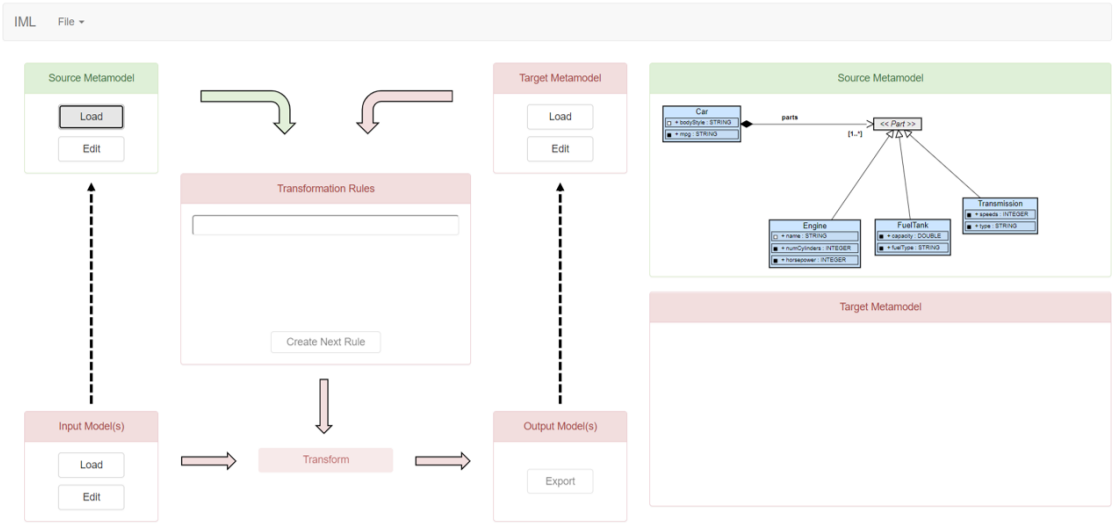



Figure 5.3: Model transformation UI after successful source metamodel upload.

The first row contains the panels for uploading the source and target metamodels. These panels are in the first and last column of the row and begin colored red to indicate the user must interact with them. After a metamodel is successfully uploaded using the "Load" button, the corresponding panel turns green. The "Edit" button is included in this UI but is not functional yet. Eventually, this button is used to change a metamodel that has been uploaded. When clicked, the "Edit" button opens the uploaded metamodel in a model editor on a new browser tab. Once edited and saved, the updated model is used in the model transformation. This functionality was not implemented in this project because it falls outside of the scope of performing model transformations. The second and third columns of the first row are arrows beginning at the metamodel panels and ending at the transformation rules panel. These arrows are intended to show the user they must begin a model transformation by uploading the metamodels before defining rules. Both arrows begin red like the metamodel panels. Each turn green when the corresponding metamodel panels turns green. Figure 5.3 shows a successfully source metamodel uploaded while the target metamodel panel is still waiting.

The second row of the model transformation column contains 3 columns. The first and last columns contain dashed arrows starting at the input models and ending at the metamodels. The first dashed arrow begins at the input model panel and ends at the source metamodel panel to illustrate to the user that the input model must conform to the source metamodel. The second dashed arrow is used for the same reason but to indicate this relationship with the output model and target metamodel. The second column of the second row of the model transformation column is a panel that contains the transformation rules defined between two uploaded metamodels. To begin creating rules, users must click the "Create Next Rule" button. When first landing on the model transformation page, the "Create Next Rule" button is greyed out. This can be seen in figure 5.1. This is done to force the user to upload two metamodels before creating rules. It is not possible to define rules without two metamodels, therefore, limiting interaction with the button was designed to reduce user confusion. When only one metamodel is uploaded, the "Create Next Rule" button

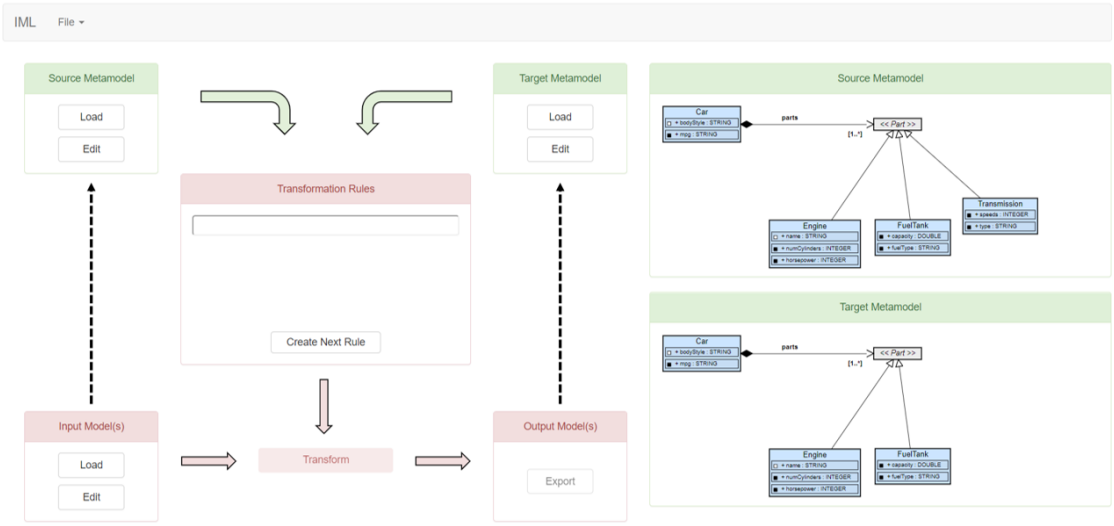



Figure 5.4: Model transformation UI after successful source and target metamodel upload.

is still greyed out. This functionality can be seen in figure 5.3. The button is only functional when both metamodels have been uploaded. An example of this can be seen in figure 5.4. Even with the button functional, the rules panel is still colored red. The rules panel turns green once the first rule is defined. This can be seen in figure 5.5. Finally, when all possible rules are defined, the rules panel remains green but the "Create Next Rule" button becomes greyed out. At this point, the button text changes to "All Rules Created". Again, this is done to reduce user confusion by not allowing any more interaction than is possible for a model transformation. This stage of the transformation rules panel can be seen in figure 5.2. The process for creating a rule has been left out of this description because it is described in section 5.2.

The third row of the model transformation column also contains 3 columns. The first and last columns hold the instance model panels. The input instance model panel has similar functionality to the metamodel panels. The one difference is the input instance model panel begins with its buttons greyed out. This can be seen in figure 5.1. Users are prevented from uploading an input instance model until a source metamodel has been uploaded. This functionality was implemented to ensure the input model conforms to the source metamodel and conformance cannot be checked until there is a metamodel to check against. Once the source metamodel is uploaded, the "Load" button becomes functional and the input instance model panel has the same functionality as the metamodel panels. The output instance model panel works differently from the other model panels. This is because no output model is uploaded. The output model is produced as a result of applying transformation rules to the input instance model. This panel remains red and its button greyed out for the entire model transformation specification process. It only becomes green and functional when a successful model transformation has been completed. This can be seen in figure 5.2. When the output instance model panel is functional, users can click the "Export" button and the output instance model .iml file is downloaded immediately. Lastly, the second column in the third row of the model transformation column contains the "Transform" button and three arrows. The "Transform" button begins greyed out. It remains disabled until an input instance model has

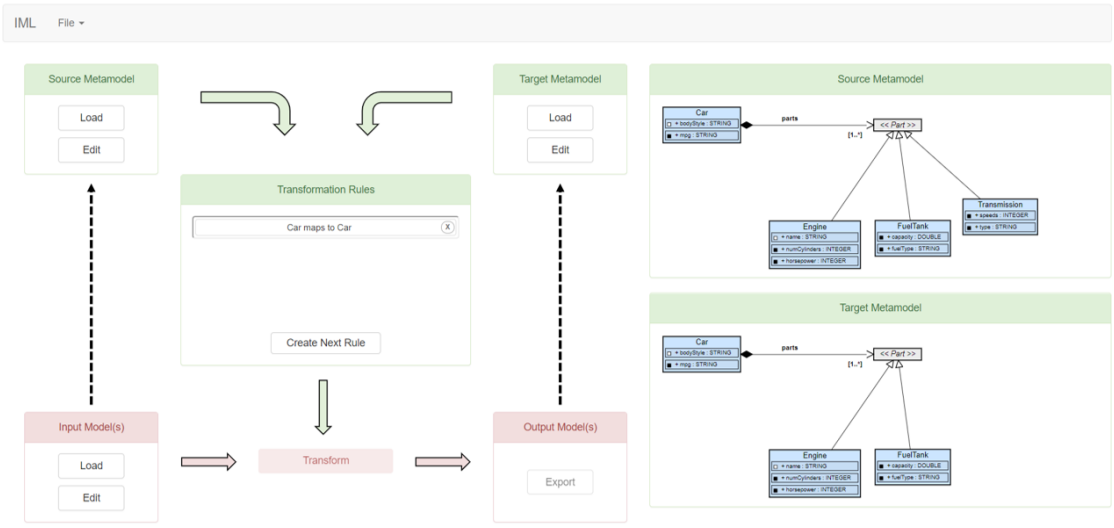



Figure 5.5: Model transformation UI after one defined rule.

been uploaded and at least one transformation rule has been defined. The arrows operate similarly to the metamodel arrows. They turn green when their corresponding element turns green. For example, when the input instance model turns green, the first right facing arrow also turns green. A picture of this situation can be seen in figure 5.6. The downward arrow corresponds to the rules panel while the second right facing arrow corresponds to the output instance model panel.

The second column of the model transformation page contains the metamodel display panels. Like the other elements, these panels begin colored red and turn green as the user fills in the model transformation. These panels are connected to the source and target metamodel panels in the same way the source and target metamodel arrows are tied to those panels. The one difference is when a metamodel is uploaded and the panel turns green, an IML model of the metamodel is displayed in the panel. These displays are included in the UI to help the user define transformation rules. Seeing the classes, attributes and relations of both metamodels allows the user to connect elements visually when presented with rule definition options. Instead of having the metamodels open in other browser tabs, all the information they need is located on one screen. The displays are implemented using JointJS. A JointJS paper is implemented for each of the displays. When a metamodel is uploaded, the import methods are leveraged from the model editor. These papers only have zoom and drag interactions activated. An example of both metamodel displays with metamodels can be seen in figure 5.4 or figure 5.2.

Each element of the model transformation UI has been explained in this section. The UI was created with a focus on making model transformations easy for the user. The user is guided by arrows and color as they define a model transformation. This description of the UI did not include how users can define transformation rules. The next section of this chapter describes this process.

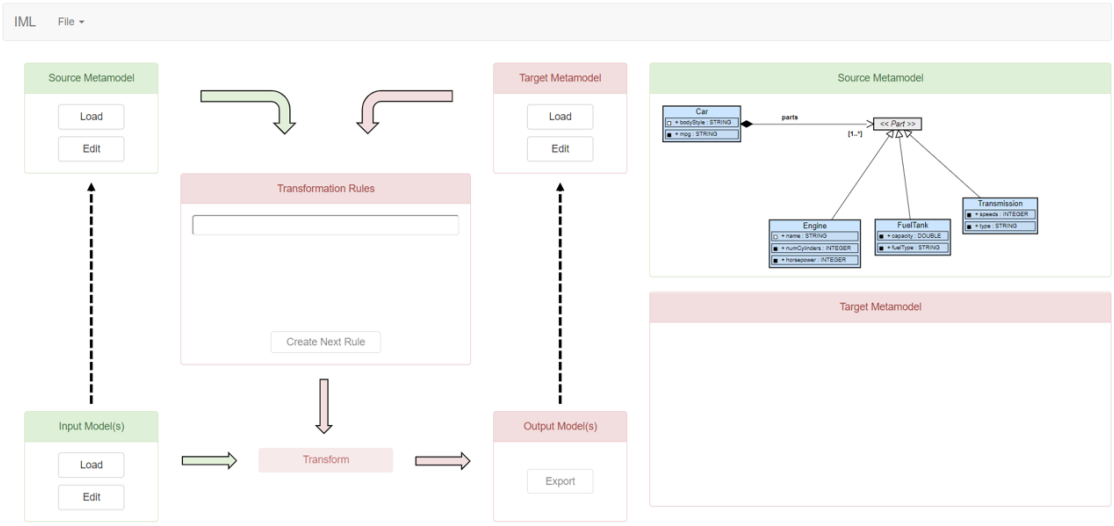



Figure 5.6: Model transformation UI after successful input instance model upload.

## 5.2 Mapping Rules

Defining mapping rules between two metamodels enables conforming instance models to be transformed between the metamodels. The process of defining those rules is captured in the IML model transformations page. The IML approach to defining transformation rules focuses on making it simple for beginner users. This process gives users a strong guide for defining rules. Modeling raises the level of abstraction for a software system, thus, the M2M transformation process is also an abstract concept. For this reason, we designed a process that constrained rule definition as much as possible, while still showing users the capabilities of M2M transformations. This constrained process, along with the heavy guided experience, allows users to focus on the core concepts rather than worrying about complex model transformation language syntax. By entirely removing an additional language requirement, our implementation focuses on the foundational knowledge over rigid definitions. Further, by providing users with clear feedback about the rules they create, and only providing valid options, IML streamlines the model transformation process.

There are two types of rules in IML M2M transformations, simple mappings and complex mappings. Mappings can be made between two classes or two attributes (Note: a relation is considered an attribute in this context). To begin defining a rule, users must upload a source and target metamodel. After this, users can select the newly enabled “Create Next Rule” button. When clicked, a modal appears similar to the one seen in figure 5.7. This modal allows a user to begin a transformation rule by mapping a class in the source metamodel, to a class in the target metamodel with a simple class mapping. The source class is static, while the target class can be any class in the target metamodel. Figure 5.8 shows the target class options. This is because there must be a rule for every object that may be encountered in the input instance model. The rule for that class tells the transformation engine how to generate the correct output for an object of that class. When defining the second rule, a new class from the source metamodel is chosen. Source classes are chosen in the order they are saved in the back end data structures. This was determined the best method for

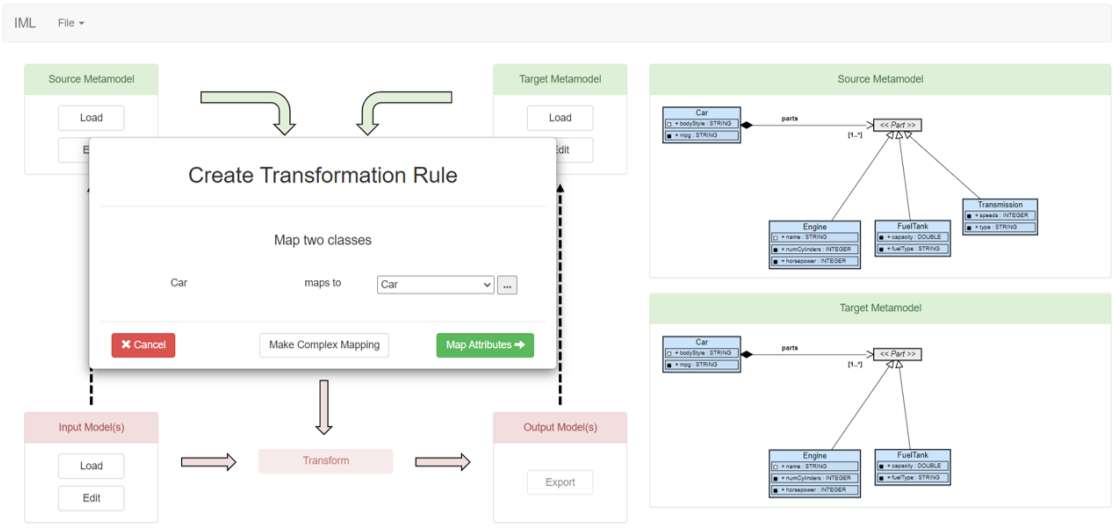



Figure 5.7: Map classes modal.

choosing classes because the order of the rules is insignificant as long as they stay in order and the classes can be easily removed from the data structures created from the import methods. In addition to mapping to a target class, class rules can have a target of "none". "None" is used when objects of a source class are not be transformed into a class in the target. For example, referring to the metamodels in figure 5.7, the "Transmission" class in the source metamodel should likely be mapped to "none" because it does not show up in the target metamodel. Changing between metamodels with small mutations, like this, is a common use case for model transformations.

To move on to the next step of the process, users click the "Map Attributes" button. This declares the rule for the "Car" class as a simple class mapping. After clicked, the user is presented with another modal to define an attribute mapping. This modal is nearly the same as the simple class mapping modal except the title and subtitle have been updated to "Map Class Attributes" and "Mapping Car to Car", respectively. This can be seen in figure 5.9. These text cues are providing the user with the context of the mapping rule. The user should now understand they are mapping the attributes of the Car classes in each metamodel. Similar to the class mappings, the attribute mappings have a static source attribute with a set of choices for the target class attribute. The target class attributes drop down has the name of all of the attributes found in the target Car class, including the "part" relations. This can be seen in figure 5.10. Referring back to code generation, relation elements are seen as object attributes while attribute elements are seen as primitive attributes. This is the same case here. Object and primitive attributes are allowed to be mapped together to widen or narrow a property.

To finish an attribute mapping, the user clicks the "Map Attributes" button. This displays the next source attribute to map. When the last source attribute to map is displayed, the text of the "Map Attributes" button is changed to "Finish" to indicate there are no more source attributes to map. This can be seen in figure 5.11.

After clicking "Finish", the success modal appears to the user. This is shown in figure 5.12. The success modal aims to provide the user with a conclusion for the rule definition process. The

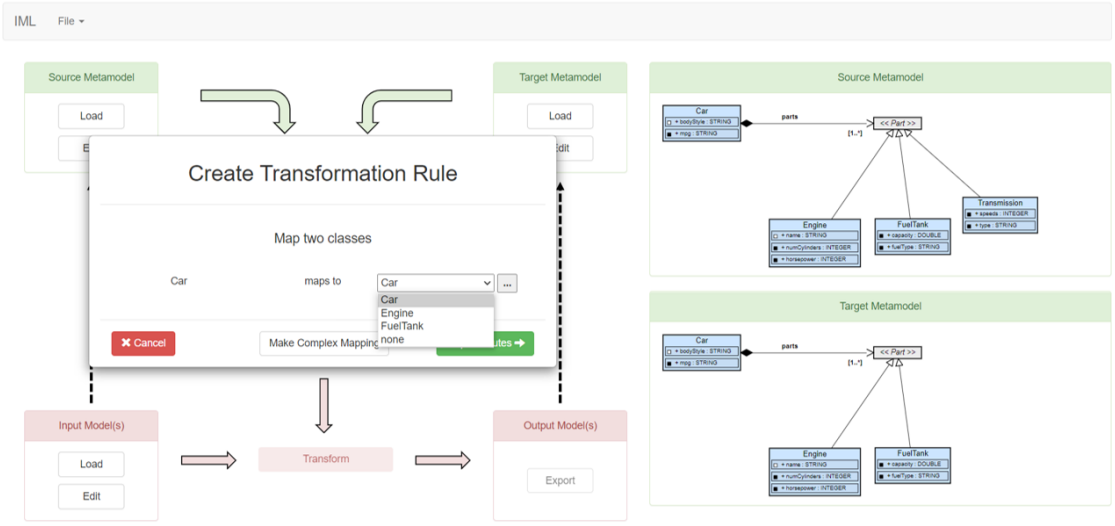



Figure 5.8: Map classes modal with drop down.

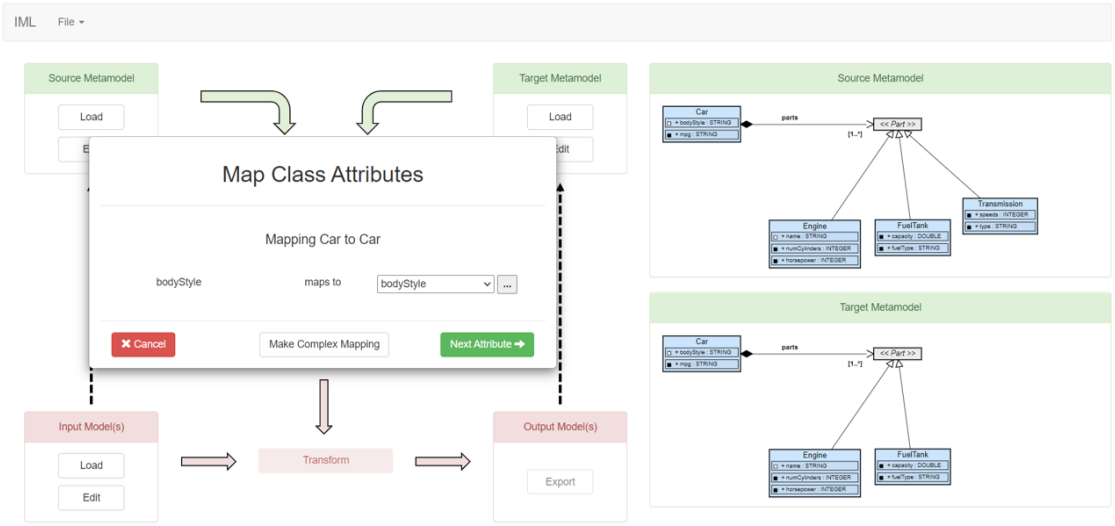



Figure 5.9: Map attributes modal.

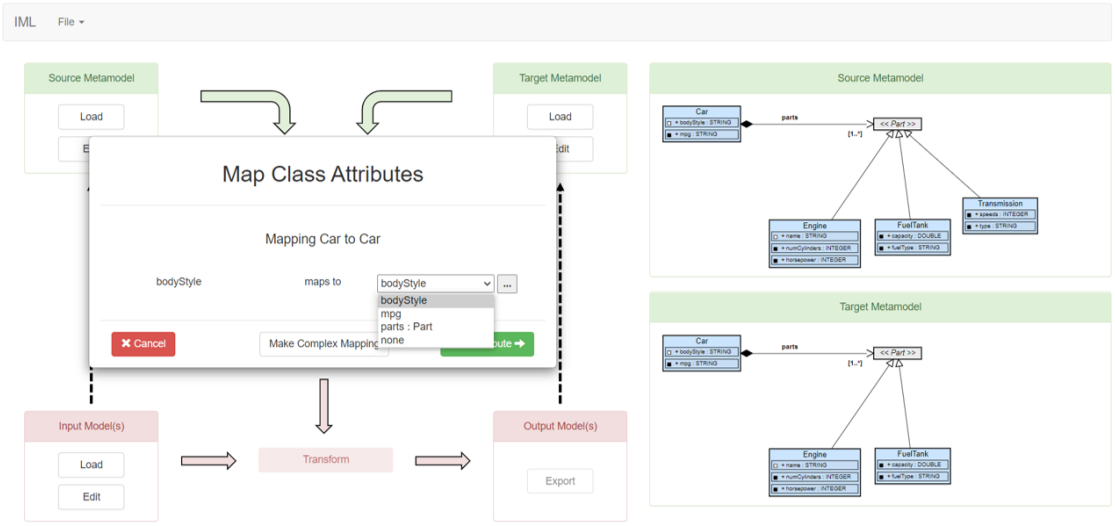



Figure 5.10: Map attributes modal with drop down.

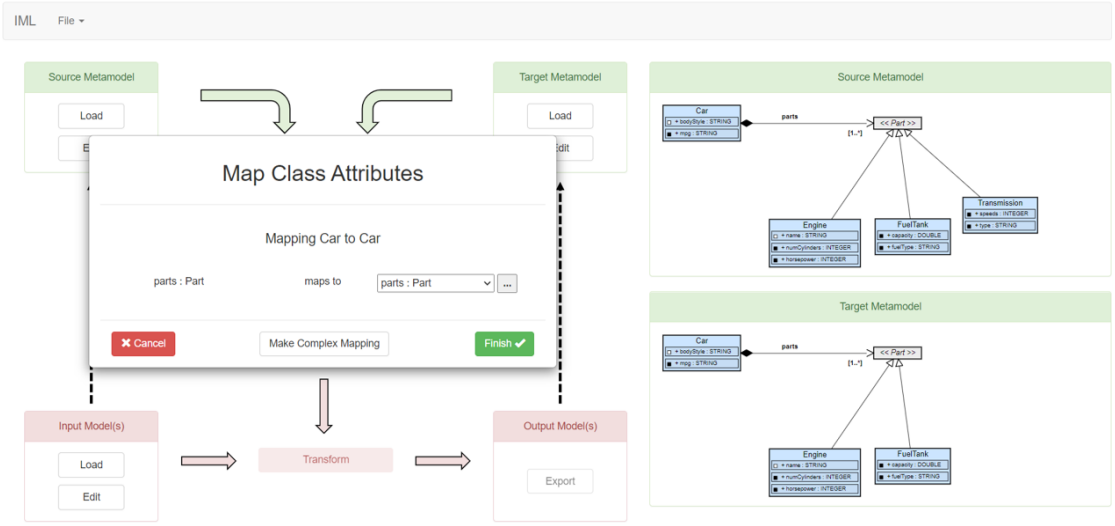



Figure 5.11: Map last attributes of a class mapping.

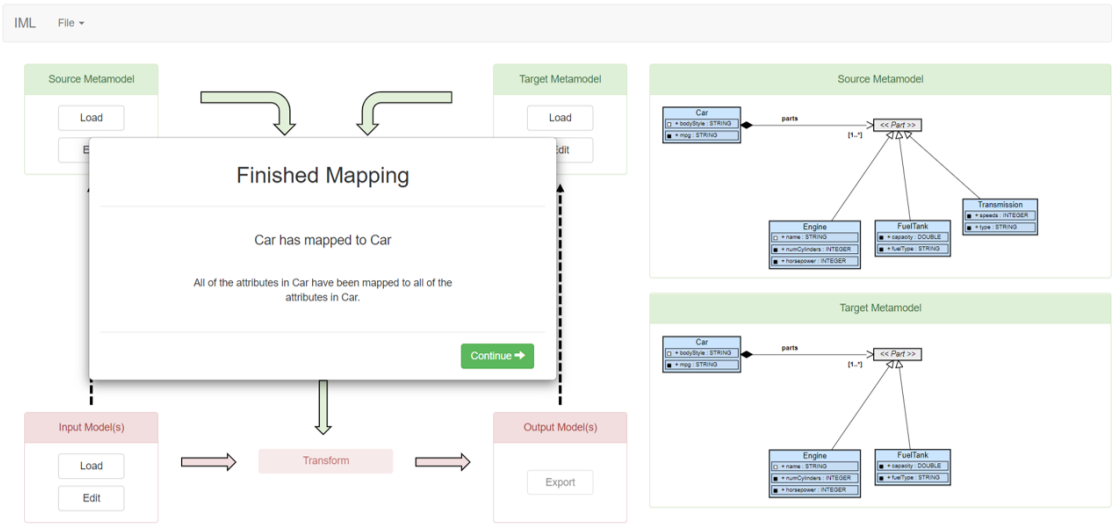



Figure 5.12: Mapping success modal.

end of the rule mapping is clearly stated with the modal title “Finished Mapping”. The subtitle of the modal restates which class have been mapped together. And, the body of the rule states if the attributes have been mapped together. This is important for classes that are mapped to “none”. In this case, the subtitle says, “Car has mapped to none” and the body says, “Instances of Car are not included in transformed models.”. This is displayed in figure 5.13.

Besides simple mappings, users can define complex class and attribute mappings. Complex mappings allow users to define mappings that occur depending on one or more conditions. These conditions are based on the value of the attributes in the current instance object. For example, a user may want to delete certain objects from their instance model if it contains a specific attribute value. Figure 5.14 displays a concrete example of this situation. In English, the modal says that if an object has a bodyStyle attribute with the value of “Sedan”, then Car should map to “none”. Otherwise, Car maps to Car. This means if an object is discovered to have a bodyStyle attribute with the specified value, then that object is not transformed. All other Car objects are transformed.

More conditions can be added to the modal by clicking the “Add else if” button at the bottom, center of the complex mapping modal. Figure 5.15 shows a condition modal with 2 more conditions added. When more conditions are added, the modal extends downward until it hits a maximum extension. Then, the modal adds a vertical scroll bar to access the conditions that cannot be seen. Additionally, the added conditions can be removed. A complex mapping must have at least one condition but there is no maximum requirement. The complex condition shown in figure 5.15 is the same as the mapping in figure 5.14 except it checks for additional body styles to leave out of the transformation.

Complex attribute mappings work in a similar fashion to complex class mappings. The main difference is the mappings involve source and target attributes instead of classes. This is shown in figure 5.16. The example shown in figure 5.16 states that if an “mpg” attribute of a car object equals “0”, then do not map the “bodyStyle” attribute. Otherwise, map “bodyStyle” to “bodyStyle”. Else if statements can be added to complex attribute mappings in the way as complex class mappings.

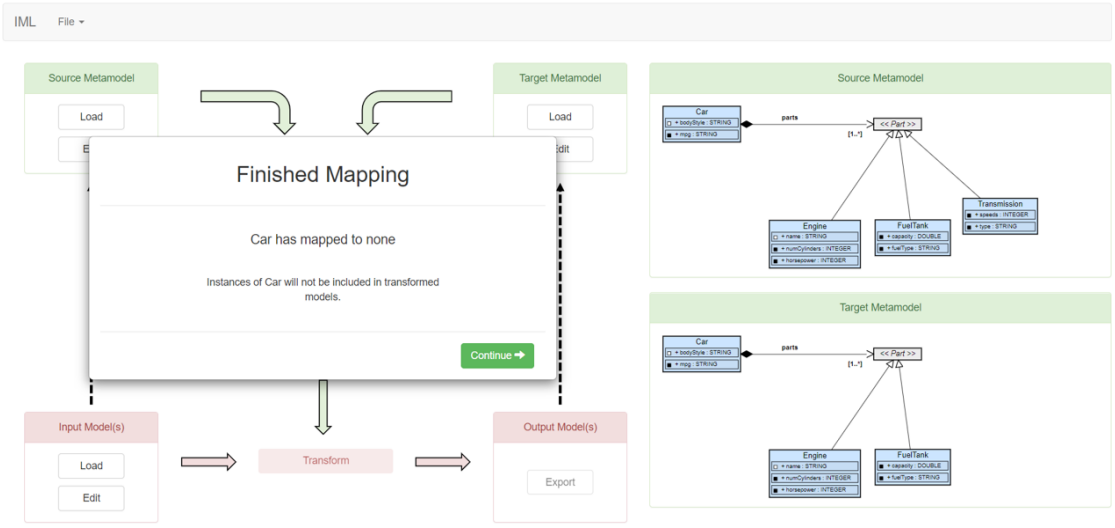



Figure 5.13: Mapping to none success modal.

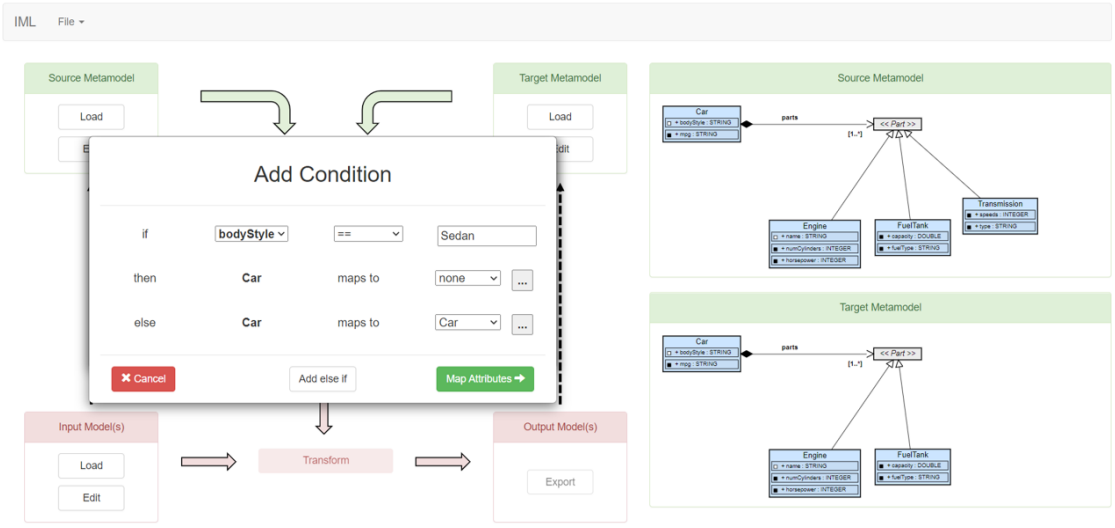



Figure 5.14: Complex class mapping modal.

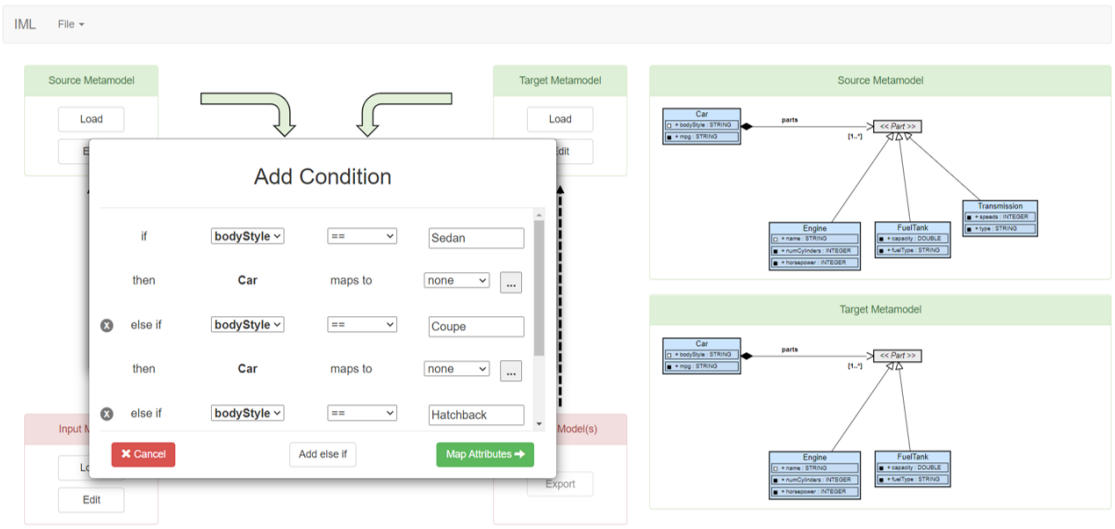



Figure 5.15: Complex class mapping modal with multiple conditions.

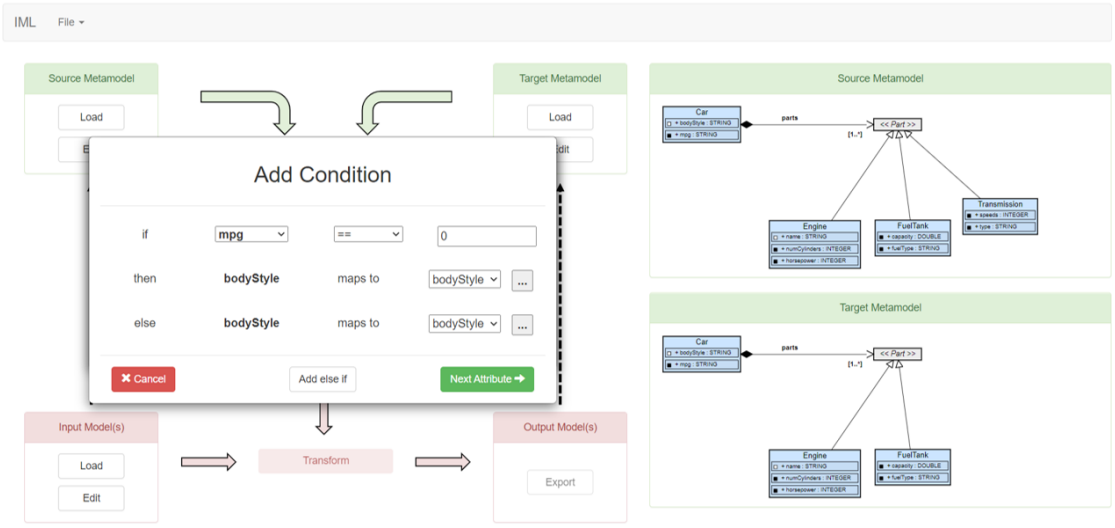



Figure 5.16: Complex Attribute mapping modal.

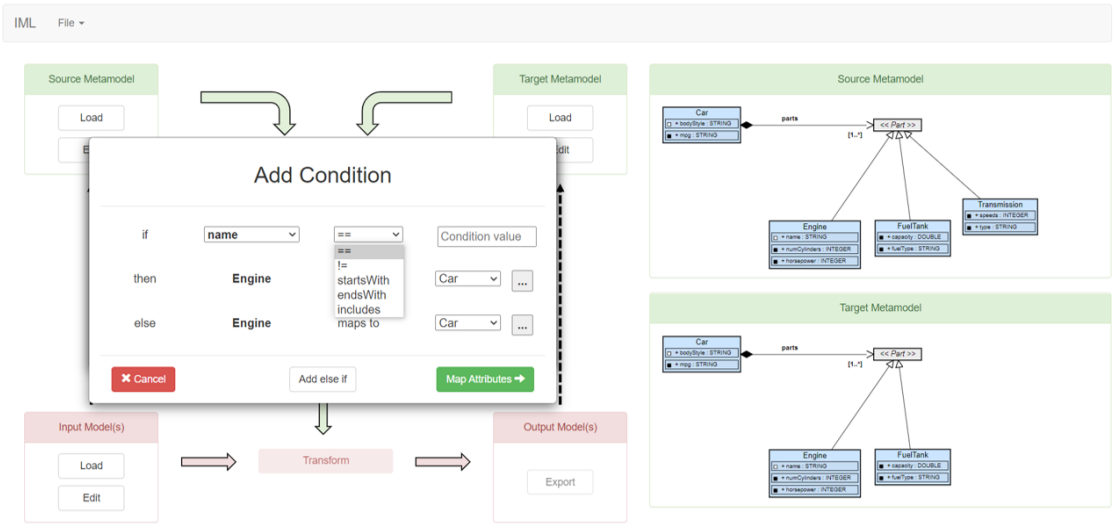



Figure 5.17: String operators.

The operator used for comparison in a complex mapping condition can be changed. A drop down of possible operators for that attribute type is provided for the user. An example this drop down can be seen in figure 5.17. These operators update depending on type of the chosen attribute. Figure 5.18 shows the available operators for integers and doubles. Conditions based on boolean attributes are unique. Boolean attributes are already true or false. We took advantage of this by making the operator a static equals operator and moving the drop down to the condition value. The drop down includes the values true and false. This can be seen in figure 5.19.

After a transformation rule has been successfully defined. All modals close and the rule is displayed in the transformation rules panel. Examples of this have been shown in figures 5.5 and 5.2. If a complex class rule has been defined, the rule display includes each of the possible target classes. Using the complex class mapping of "Car", figure 5.20 shows the rule display. This display indicates the "Car" could be mapped to the target "Car" or "none". If there were more possible classes mapped, these would be displayed in the order that they were created in the complex mapping modal. The attribute mappings are not displayed on the model transformation page.

Lastly, when defining mappings, users can enforce attribute assignments. These allow the user to assign class attributes with specific values. These can assigned with class mappings or attribute mappings. To implement attribute assignments on a class mapping, user can click the "..." button located to the right of the target classes drop down. When clicked, the attribute assignment modal appears. This can be seen in figure 5.21. The attribute assignment for classes allows the user to provide a value for each attribute. When the user has entered their assignment values, they can click the "Assign" button. This hides the attribute assignment modal and bring the class mapping modal back into focus with one change. The "..." now appears with a black background and white text to indicate the values have been set. An example of this can be seen in figure 5.22. After assignment, the user is still able to interact with the "..." button. If clicked, the assignment modal reappears. This time, the current values of the attribute assignments is displayed and the "Clear" button is enabled. This can be seen in 5.23 From here, users can enter new values and click "Assign". This

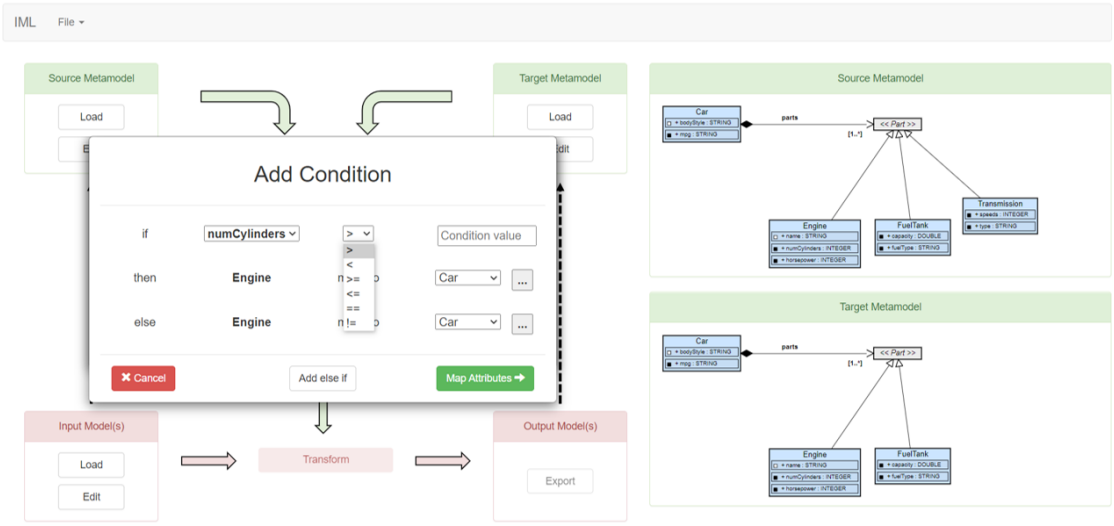



Figure 5.18: Integer and double operators.

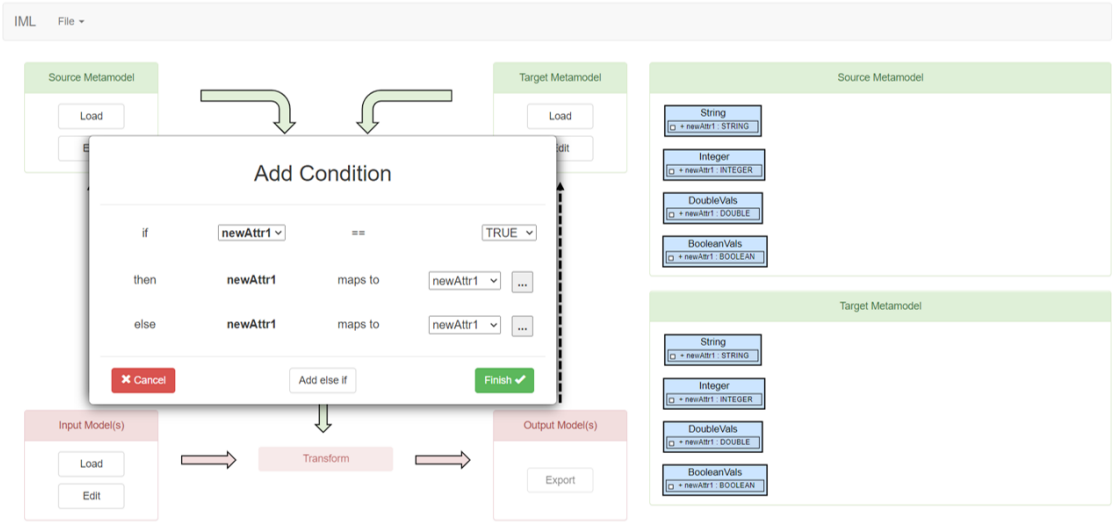



Figure 5.19: Boolean attribute condition.

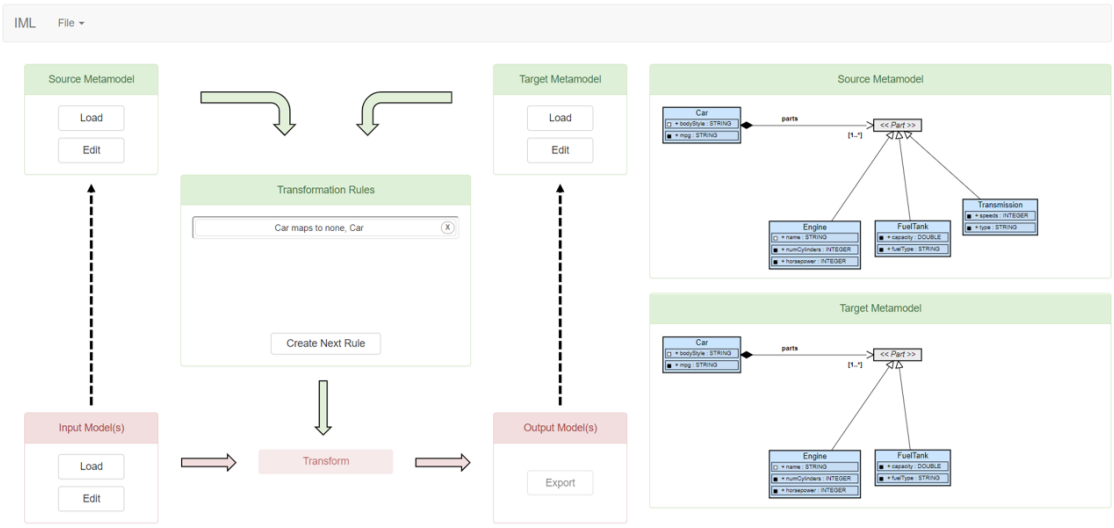



Figure 5.20: Complex rule displayed in transformation rules panel.

replaces the current values. Or, the user can choose to remove the assigned values by clicking the "Clear" button. If clicked, the attribute assignment modal disappears and that class mapping modal is returned to its original state. The attribute assignment modal also resets.

Attribute assignments can be created for specific attributes. This occurs within the attribute mapping modal. The functionality of the attribute assignment modal is exactly as it is with class attribute assignments except only the target instance attribute can be set. The attribute assignment modal in this situation can be seen in figure 5.24. There is another difference with this attribute assignment modal. The title has a note under it. This note says "Assigning values now overwrites Class level assignments". This note has appeared because of the attribute assignments that were created at the class level. This is to warn the user that any values set with a class level attribute assignment is overwritten with an attribute level attribute assignment. This note does not appear if there are no class attribute assignments.

Complex mappings also have the ability to implement attribute assignments. The "..." button to implement the assignments appears next to each mapping for each condition. The functionality works exactly as it does for class and attribute assignments, respectively.

The user's work is finished once they upload their models and defined their mapping rules. The next step is to apply the rules to their input instance model and generate an output instance model. This task is completed by the transformation engine. The algorithms that make up the engine are presented and explained in the following section.

## 5.3 Transforming Instance Models

Similar to code generation, transforming instance models requires iterating over the elements of an IML model and generating the equivalent in the target domain. In the case of M2M transformations, the input instance model is the only model to loop through. Pseudocode of the transformation

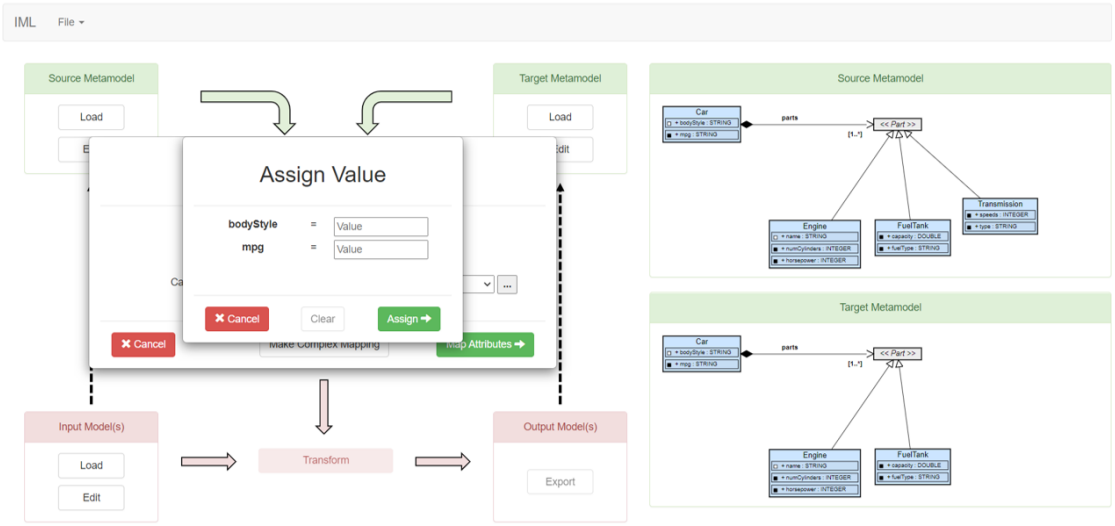



Figure 5.21: Class attribute assignment modal.

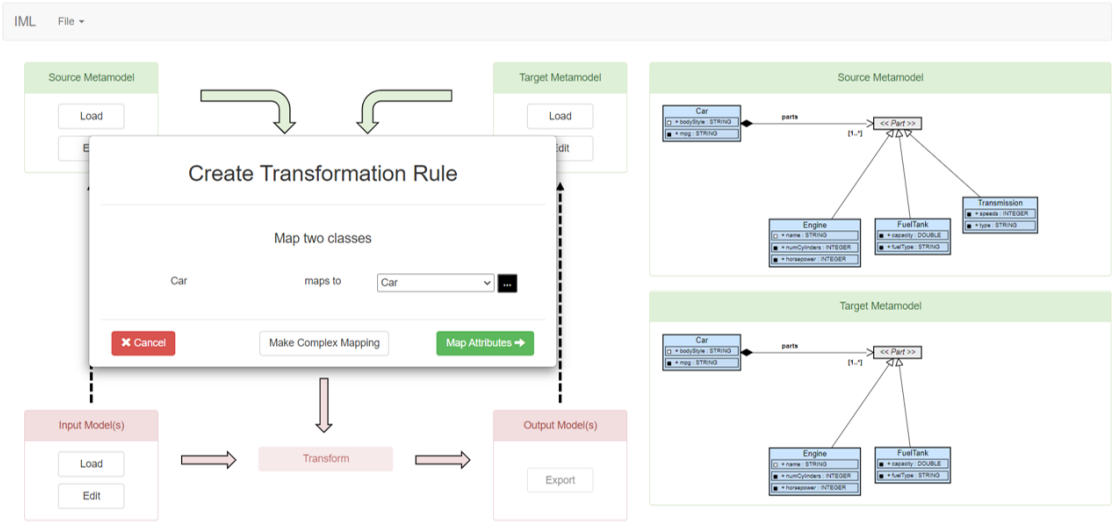



Figure 5.22: Class mapping with saved attribute assignment.

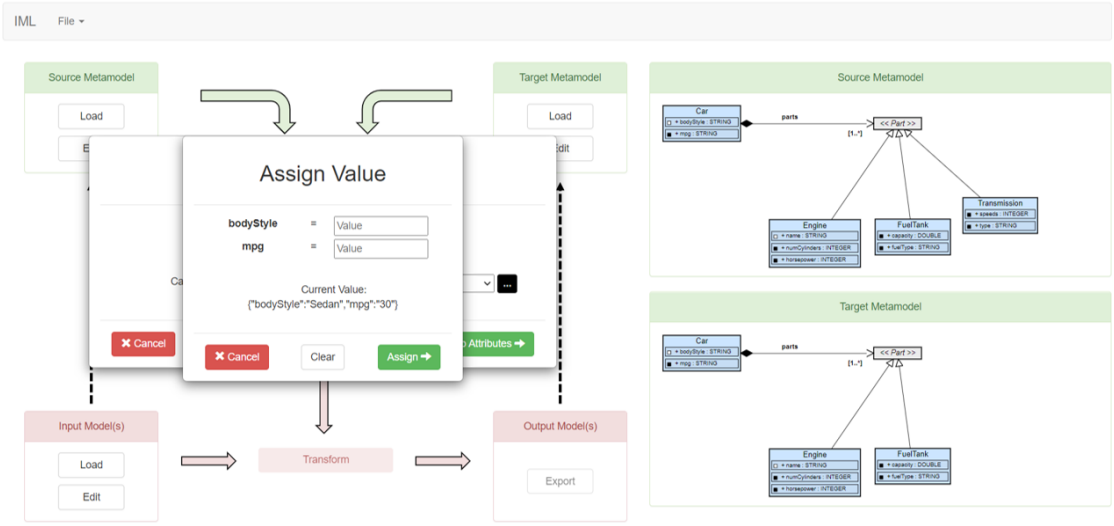



Figure 5.23: Attribute assignment modal with existing attribute assignment.

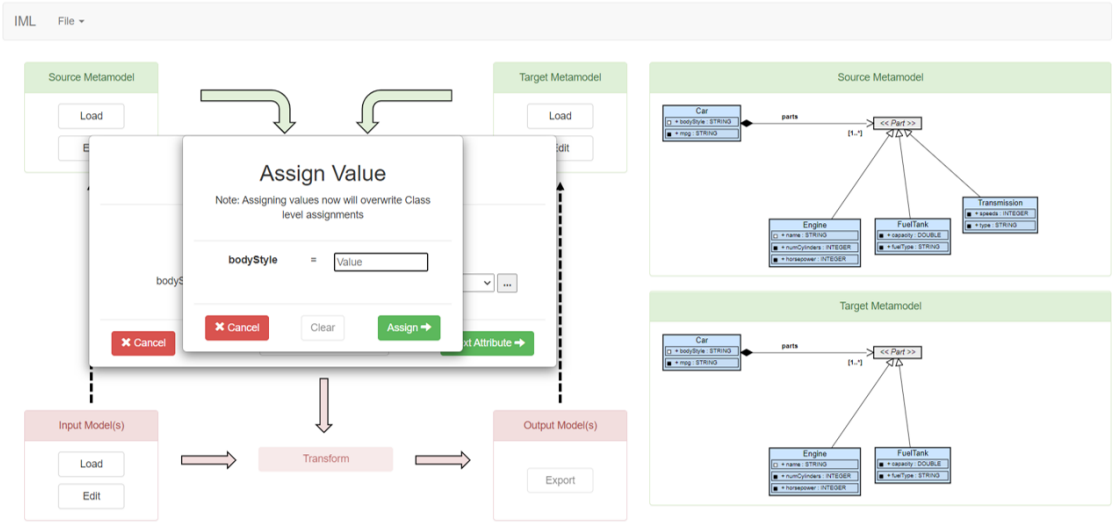



Figure 5.24: Attribute assignment modal.

```
function transformInputInstanceModel () {
    foreach(classMapping in classMappings) {
        foreach(class in inputInstaceModel) {

            if(classMapping applies to class) {

                Transform class using class mapping data.

                foreach(attributeMapping in classMapping.attributeMappings) {

                    Transform each attribute using attribute mapping data.

                }
            }

        }
    }

    // All classes, with their attributes, have been added to the target instance model.

    foreach(relation in sourceInstanceModel.relations) {
        foreach(relationMapping in relationRules) {

            if(relationMapping applies to relation) {

                Transform each relation using relationMapping mapping data.

            }

        }
    }
}
```

Figure 5.25: Transformation engine pseudocode.

algorithm is shown in figure 5.25.

The algorithm is made of two main loops. The first loop iterates over each class mapping created by the user. Inside of this loop, the instance model classes are looped through as well. Every iteration of the model classes loop checks if the current rule applies to the current class. If the rule applies, then the class must be transformed according to the details of the rule. After the source instance class has been converted into a target instance class, the attributes of the source class must be transformed. To apply these transformations, the attribute mappings for that class must be used. These mappings are saved as a part of the class mapping data structure. This allows them to be easily obtained during a class transformation. Once the attributes are translated into attributes of the target instance class, the class transformation is complete. The next class found in the input instance model is checked to see if it applies.

Each class mapping is compared to every class in the input instance model. This must be done because it is unknown how many classes apply to any given rule. Furthermore, this does not result in duplicate transformations because there is only one rule for each class in a transformation. Therefore, only classes that should be transformed by a rule, are transformed by that rule.

The second main loop of the transformation engine iterates over each relation in the input instance model. Then, for each relation, a loop is run through for each relation mapping. One important distinction here is the introduction of the relation mapping. This was not discussed previously

because it does not apply at a conceptual level of model transformations. The previous section explains relations to be considered attribute mappings. While this is still correct, implementation would not allow for this level of abstraction. When transforming relation elements, we must consider the source and target classes of a relation. For this reason, all target instance classes must be created before any relation can be transformed. This allows the transformed relation to be properly attached to its source and target classes. Without ensuring that all class have been transformed, there is no way to guarantee a class exists when it is time to transform a relation. Thus, the relation mapping data structure was created to be able to apply these transformations after the class and attribute mappings.

There are a few considerations that are abstracted away by the pseudocode. Transforming attribute values is one of them. When applying an attribute mapping, the attribute types must be taken into consideration. For example, an integer attribute might be mapped to a string attribute, or a double may be mapped to a boolean. For these cases, a value conversion algorithm was created. Where possible, common type conversion were implemented. Such as, integers parsing to strings or a double value of "1.0" converting to "TRUE" for a boolean. Default string values are the only to break from convention. The default for a string in IML is "Enter a value". The reason for this was already provided in section 4.1.

Another consideration is the output instance model must conform to the target metamodel. Despite the guided nature of IML model transformations, there are many cases where it is possible to generate a nonconforming instance model. So many that we were afraid the user would have trouble performing model transformations. Considering this is an educational tool, we want the user to explore the benefits of MDSE with as little frustration as possible. For example, a user is only required to define at least 1 rule before performing a model transformation. If the user leaves out a rule for a class that is a part of a required relation, the output model is out of conformance. In these cases, the transformation engine attempts to fill in the output instance model with just enough default objects and attributes to make a conforming instance model. For example, a transformation and its input and output instance models are provided in figures 5.26, 5.27 and 5.28, respectively. The transformation specification leaves out a rule for mapping "Department" even though one Department object is required for the "Course" class. This does not present a problem in IML model transformations because the requirement for "Department" is detected and a default object is inserted in the output instance.

Output model conformance must also be considered in the context of bounded attributes. For example, an attribute with a lower bound of 2 may be mapped to an attribute with a lower bound of 3. In this case, the source attribute may only have 2 values to provide for the target which needs at least 3. If this situation is detected, the transformation engine appends default values to the source array until the lower bound of the target attribute is satisfied. Alternatively, a source attribute may have an upper bound that exceeds the upper bound of the target. In this case, the source attribute might have too many values to provide the target attributes. If this situation is detected, the engine removes values from the end of the source array until the upper bound of the target attribute is satisfied.

This chapter provided implementation details for IML M2M transformations. To perform a M2M transformation, a source and target metamodel must be provided to the IML M2M trans-

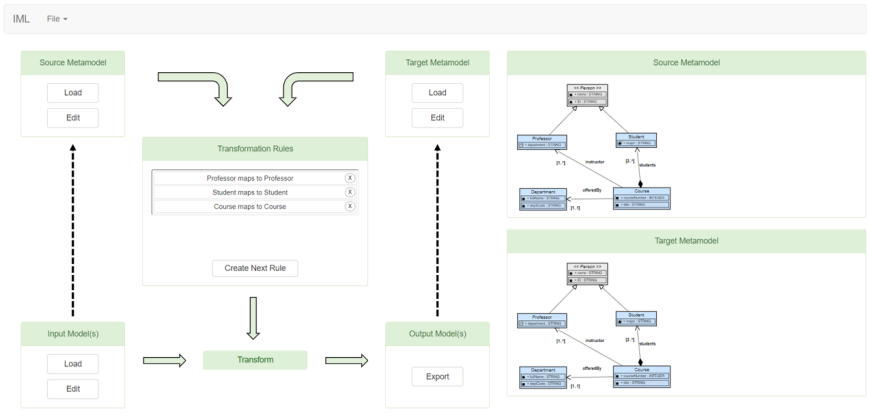



Figure 5.26: Example transformation missing a rule.

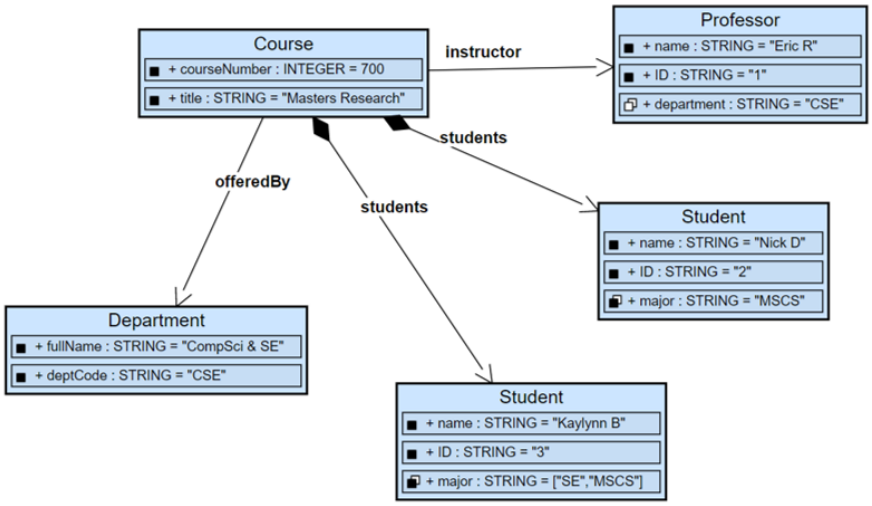



Figure 5.27: Input instance model for missing rule.

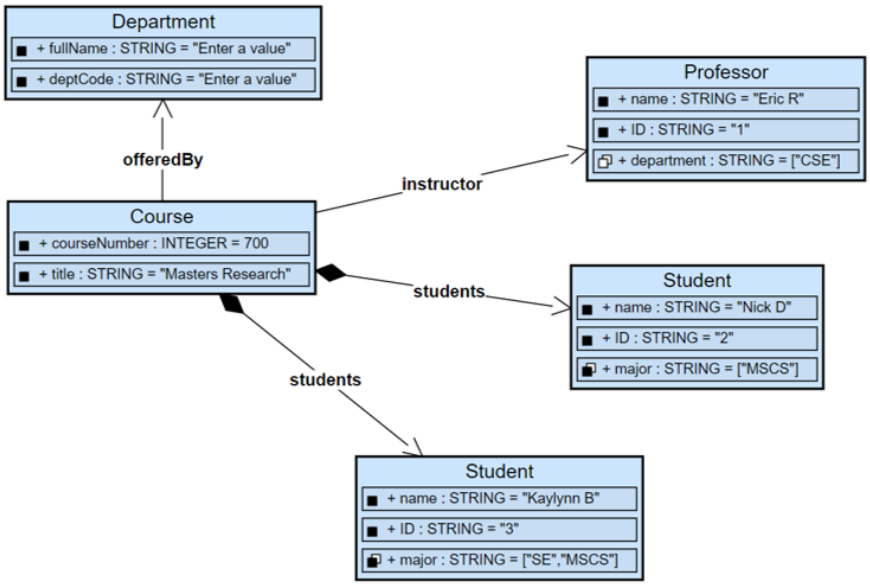



Figure 5.28: Output instance model for missing rule. Department class has been added by the transformation engine.

formation tool. An instance model that conforms to the source metamodel must also be provided. After defining transformation rules using IML's guided transformation modals, users can perform a transformation and generate an output instance model. This output model is transformed from the input instance model and conforms to the target metamodel. It is simple enough to say that these transformations work but they must also be verified to be correct. The next chapter provides an evaluation of the M2M transformation engine, as well as the model editor and code generation engine.

# Chapter 6

# Evaluation

This chapter presents the evaluation of the web-based model editor, code generation and M2M transformations implemented as part of the IML project. While the goals for these contributions focus on their intuitive and transparent nature, these evaluation experiments seek only to evaluate the correctness and validity of the transformations as technical processes. The assessment of their educational merit is future work along with a high level usability study of the complete framework.

First, the experimental design of each evaluation process is explained. Then, the results are presented, followed by a discussion of these results.

## 6.1 Experimental Design

The following section provides the experiment design for evaluating each of the contributions made to the IML project. The evaluation process for the web-based model editor is explained first. Then, the process for code generation and M2M transformations is explained.

### 6.1.1 Web-Based Model Editor

To extensively evaluate the web-based model editor, the ideal method would be running a complete user study. However, running a study like this requires extensive time and resources. Running an educational user study can take years to simply gather enough participants with appropriate longevity of data.

Instead, to evaluate the web-based model editor, we decided it would be sufficient to use the evaluation of code generation and M2M transformations as an assessment of the model editor's capabilities. To evaluate the model transformations requires the creation of a variety of models to cover a large set of possible outputs from each transformation engine. Showing the ability to produce the model test inputs for the other evaluation sets demonstrates the initial viability of the web-based modeling tool as an editor. Through our usage in evaluation, several tool design bugs were identified and corrected.

### 6.1.2 Code Generation

In software engineering (SE), there is a subset of activities called quality assurance. Within quality assurance, there is a concept known as test coverage. The most basic type of test coverage is attempting to execute every line of code in a software program. Keeping track of each line executed,

engineers can run their tests and definitively say their code is, “95% covered”, as an example. To evaluate code generation for IML, a form of test coverage is employed.

The main different between MDSE and SE is models drive development. Therefore, test coverage in MDSE must cover models instead of lines of code. More specifically, MDSE tests must cover model elements. To evaluate the IML code generation engine, models were created to use each model element and generate the corresponding code. Additionally, model elements have different types of implementations. For example, a class element could be concrete or abstract. These different implementations were also enumerated for testing.

With all software testing, there is an expected output that can be compared to an actual output to determine if a test passed or failed. In the case of IML code generation, the tests output Java code. One method for deciding if a test passed would be to compare the actual Java code files against a set of expected Java code files. While this would be sufficient, it is not necessary. The goal of MDSE is to use abstraction to make viewing code unnecessary, or at the very least an infrequent activity. Therefore, we decided to compare expected system output against actual system out. This is made possible by using the toString and prettyPrint methods that are created in each class file with code generation. Instead of comparing the Java code files directly, the generated Java code was compiled, run and the console output was compared to an expected console output. One advantage to this method is the Java compiler is leveraged to verify that the generated code is valid and free of any syntax errors. There is no need to check the Java files for proper syntax because the Java compiler does this automatically and correctly. Another advantage to this method is it proves the code generator is semantically correct, instead of simply syntactically. This is then more than a test of the code generator, it is a test of the generated code. Proving the generated code works properly is the ultimate goal and is a transitive evaluation of the code generator.

To implement this evaluation process, the code generation was slightly altered. Print statements of each object were added to the end of the instance model code to get a system output. The altered code generator was then used for each test model. Before compiling and running the code for a test model, a text file of the expected system output was created manually. After creating the expected output file, the generated code was imported into the Eclipse IDE. The code was then compiled and run to generate the console output. This output was then copied into a text file so that it could be compared to the expected output text file. Using a Jupyter notebook, a python program was created that traversed a file structure to open the expected and actual output files and compare them line by line. The lines of each file are striped of the whitespace at the end of the line. This is the only way the files were altered before comparison. After the initial execution of comparisons, some differences were observed. Some of the differences were identified to be objects printing in a different order than they were created in the expected output. These differences did not show anything wrong with the code generator, so the expected object outputs were rearranged to allow line comparisons to be completed automatically. Additionally, some typos in the expected output files were identified on the initial execution. Again, these were determined to not pose a risk to the evaluation so they were corrected. After these corrections, each expected output file was shown to be equivalent to the actual output files. The outcome from these tests are provided in a table in the results section for code generation.

In addition to covering each element, one complex test was completed to show multiple ele-

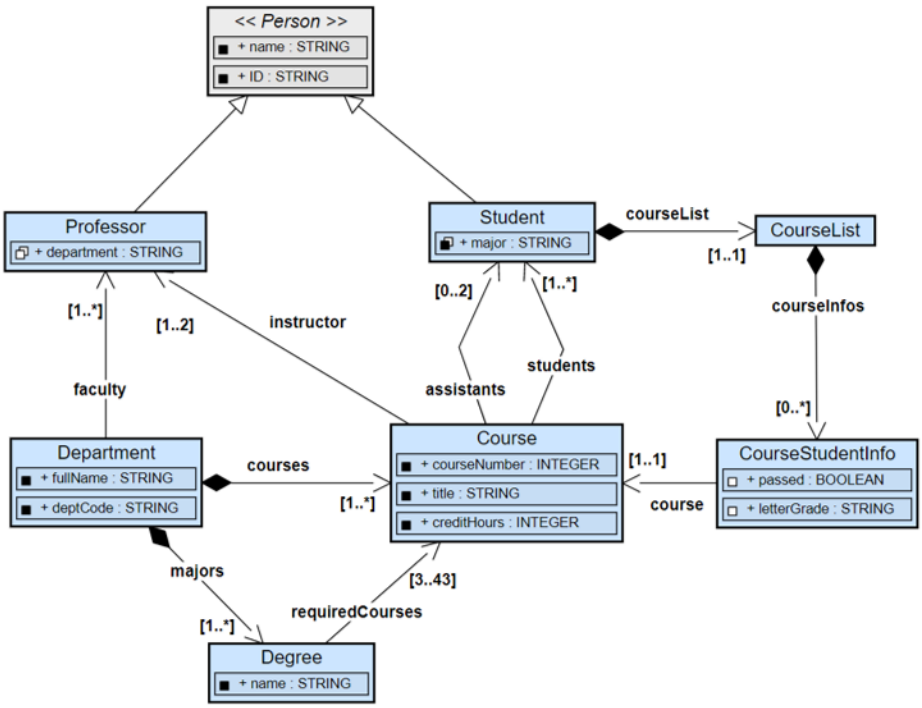



Figure 6.1: Metamodel representing the department of a college.

ments working together in code generation, as well as to form a practical running example. The metamodel and instance model for this are presented in figures 6.1 and 6.2. Using these models, test output was generated in the same manner as every other test. The only difference for this test is only the actual and expected output for the Department object was compared. This was done because Department can be considered the root object of the model. This means implementing a Department object forces the implementation of every other object in the model. Therefore, when the Department object is printed, so is every other object in the model.

In order to perform a high-level evaluation of one of the transparency features of our code generation process, we also aimed to ensure that the generated ASCII art representations were correctly implemented and displayed. As this is a secondary validation of the generated code, we opted to perform a random sampling spot-check inspection of the generated code files, and examined the file headers for the presence and correctness of the ASCII representation of the IML class figure.

To accomplish this evaluation, one class file from the output of each code generation test was chosen to be evaluated. This file was chosen by ordering the files alphabetically and assigning each file an integer id based on that order and then generating a random number. Beginning at 1, the first file in alphabetical order was assigned its id. The id integer was incremented for each subsequent file, until there were no more files left in the Java package. It is important to note that the class file for the generated code relating to the instance model was not given an id. Instance model code does not represent any specific class element in the metamodel, therefore, it does not have an ASCII art representation. Another consideration is if there was only one IML class in a model, then there was

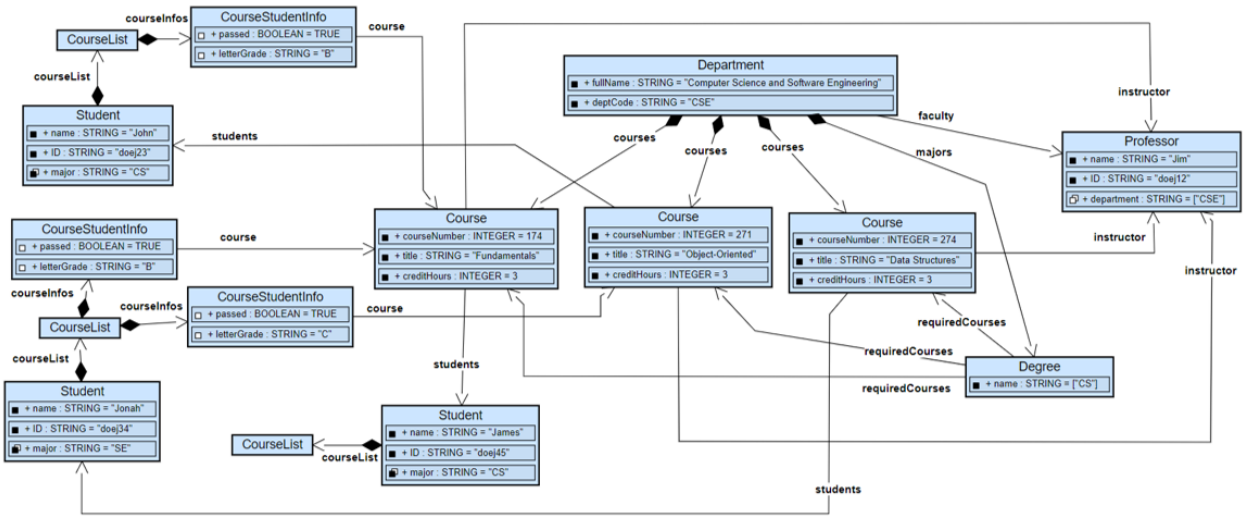



Figure 6.2: Instance model representing a simple version of the department of Computer Science and Software Engineering at Miami University.

only one Java class file to chose from. In this case, there was no random sampling. The only valid Java class file was chosen for evaluation.

Once a file was selected, the file was opened and the ASCII art was visually compared to the corresponding IML class in the metamodel for that test. For the ASCII art to be properly generated, the name of the class should appear as the header of the ASCII art. Additionally, each attribute should be included in the same order as they appear in the IML class model and all the properties of each attribute should match what is in the model. These properties include the, lower bound, upper bound, visibility, name, type and default value, if provided. Also, composition and reference relations should appear in the ASCII art. These will always be printed after the primitive attributes, however, there is no order to the relations themselves. The properties that should appear for the relations are the same as the properties that appear for primitive attributes. One difference is the type associated with the relation should be the name of the target class of the relation, instead of one of the primitive types. If there are no attributes in the IML class, then only the name of the class should appear in the ASCII art. There is no distinction for an abstract class in the ASCII art representation.

### 6.1.3 Model-to-Model Transformations

Similar to the evaluation for code generation, evaluating M2M transformations applies a test coverage approach. Looking at each element, tests were created that would apply all the simple mappings possible for an element of that type. After exhausting the types of elements, tests were created to exercise transformations of complex mappings. Complex mappings are simply conditional simple mappings. Therefore, the tests created for these mappings were designed to show the different types of complex mappings instead of creating a complex mapping for each element. Finally, the evaluation tested the transformation engine's ability to assist the user with target metamodel con-

formance issues. There are many cases where an output instance model does not conform to its target metamodel after applying the transformation rules. In these cases, the transformation engine attempts to resolve the non-conformance by adding default values and elements or removing unnecessary ones.

To determine if a test pass or failed, expected output instance models were created to compare against the actual transformation output. One method for determining if the models are equivalent would be to compare the model files directly. This method was not chosen because the model files contain metadata that is unrelated to model structure. Even identical models would not be equivalent with a simple line-by-line file comparison. To test, the actual and expected model files were opened using the model editor. These models were then compared manually. If the two models had the same number of classes of each type, the same relations and relation names, and each class had identical attributes and attribute values, then the models were determined to be equivalent.The outcome from these tests are provided in a table in the results section for M2M transformations.

Similar to code generation, one complex model was created for M2M evaluation to show multiple elements working together and continue the practical running example. The metamodel seen in figure 6.4 was created for the complex test. The model found in 6.1 is used as the source metamodel. Our new target metamodel adds new classes and changes the level of abstraction for multiple elements. The transformation rules were defined between the two metamodels and applied to the instance model seen in figure 6.2.

## 6.2 Results

### 6.2.1 Code Generation

For presenting the results of the code generation evaluation, table 6.1 was created. This table contains 3 columns, Test Case, Test Description and Test Result. There are header rows for each model element, followed by the tests for that element. The Test Case column contains the name of a test. The Test Description column contains a short description of the corresponding test. And, the Test Result column contain the words “Pass” or “Fail”. To test, the code for an instance model and metamodel was generated. This code was compiled and run to produce a console output. A test was considered to pass if the actual output of the software system matched the expected output that was created manually, before code generation. The word “Pass” is used in the Test Result column to indicate that the corresponding test passed, while “Fail” indicates that the test failed.

Considering the complex model transformation, the first 40 lines of console output for the instance model can be seen in figure 6.3. The console output for this complex example is provided in Appendix A. The actual and expected output files were compared using the same python algorithm and were determined to be equivalent.

With respect to the spot-check validation of the generated code headers, all of the selected Java source files contained an accurate ASCII art depiction of the corresponding IML model class. We found this to be a sufficient demonstration of this transparent code generation feature. A table displaying each test can be seen in table 6.2. The first column, Test Case, uses the name of the test from the first code generation evaluation to signify that these files were used to run the test.

Table 6.1: Code generation evaluation

| Test Case | Test Description | Test Result |
|---|---|---|
| **Class** | | |
| Empty | Creates a single class with no attributes. | Pass |
| Empty Abstract | Create an abstract class. Tested through "Inheritance Primitives". | Pass |
| **Attribute** | | |
| Type And Value | Implement variable of each type and initialize variables using values provided by user. | Pass |
| Type And Default Value | Initialize variables using metamodel and type defaults. | Pass |
| Visibility | Declare variables using each type of visibility. | Pass |
| Bounds And Array Values | Declares variables to cover the attribute bounds. Includes optional vs required and single vs array. | Pass |
| **Relation** | | |
| Inheritance Primitives | Displays ability for classes to inherit primitive attributes from another class. Tests abstract class creation. | Pass |
| Reference Bounds | Declares reference relations to cover the relation bounds. Includes optional vs required and single vs array. | Pass |
| Reference Self | Displays ability for reference self relations. | Pass |
| Composition Bounds | Declares composition relations to cover the relation bounds. Includes optional vs required and single vs array. | Pass |
| Composition Self | Displays ability for composition self relations. | Pass |
| Inheritance Objects | Displays ability for classes to inherit relations from another class. | Pass |

```
Department: {
    fullName: "Computer Science and Software Engineering"
    deptCode: "CSE"
    faculty: [
        Professor: {
            name: "Jim"
            ID: "doej12"
            department: ["CSE"]
        }
    ]
    courses: [
        Course: {
            courseNumber: 174
            title: "Fundamentals"
            creditHours: 3
            instructor: [
                Professor: {
                    name: "Jim"
                    ID: "doej12"
                    department: ["CSE"]
                }
            ]
            assistants: [
            ]
            students: [
                Student: {
                    name: "James"
                    ID: "doej45"
                    major: ["CS"]
                    courseList: [
                        CourseList: {
                            courseInfos: [
                            ]
                        }
                    ]
                }
            ]
        }
        Course: {
            courseNumber: 271
```

Figure 6.3: Instance model compiled code output.

Table 6.2: ASCII comment evaluation. The Empty Abstract test was unable to be used because the test was evaluated with the set of files generated for the Inheritance Primitives test, therefore, it has no generated code. The Type and Default Value test was not used because the same metamodel was use for the Type and Value test, therefore, it would be redundant to perform this test. .

| Test Case | Chosen File | Test Result |
|---|---|---|
| Empty | Vehicle.java | Pass |
| Empty Abstract | N/A | N/A |
| Type And Value | Vehicle.java | Pass |
| Type And Default Value | N/A | N/A |
| Visibility | Car.java | Pass |
| Bounds And Array Values | AttributeArrays.java | Pass |
| Inheritance Primitives | Car.java | Pass |
| Reference Bounds | ExhausePipe.java | Pass |
| Reference Self | Vehicle.java | Pass |
| Composition Bounds | UsbPort.java | Pass |
| Composition Self | Vehicle.java | Pass |
| Inheritance Objects | Part.java | Pass |

Chosen File is the second column and it provides the file from the Java project that was chosen to be evaluated. Cells in this row contain "N/A" if the files were unable to be used for testing the ASCII art. Explanations of the not applicable tests can be found in the table description. Finally, Test Result is the last column and displays whether or not the corresponding test passed. This column contains "Pass" if the test passed, "Fail" if the test failed and "N/A" if the corresponding files were unable to be used.

### 6.2.2 Model-to-Model Transformations

For presenting the results of the M2M evaluation, tables 6.3 and 6.4 were created. These tables contain 3 columns, Test Case, Test Description and Test Result. There are header rows for each model element, followed by the tests for that element. The Test Case column contains the name of a test. The Test Description column contains a short description of the corresponding test. And, the Test Result column contains the words "Pass" or "Fail". A test involved creating two metamodels and one instance model. Then, transformation rules were defined depending on what was being tested. The tests in table 6.3 focused on evaluating rules involving different elements. Alternatively, table 6.4 concentrated on evaluating different types of rules and nonconformance situations. A test was considered to pass if the actual output instance model matched the expected output instance model that was created before the transformation. "Pass" in the Test Result column indicates that

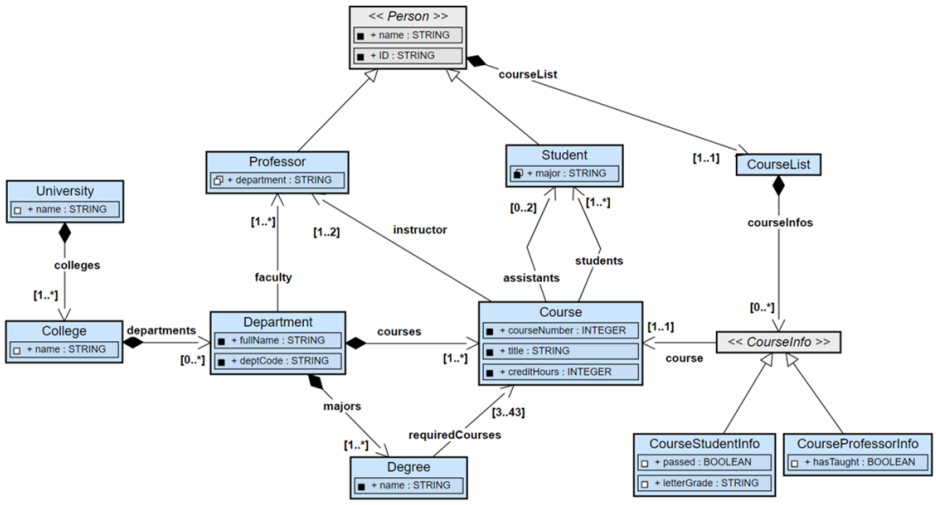



Figure 6.4: Metamodel representing a university created by altering the metamodel in figure 6.1. The University and College class were added. The courseList relation was raised a level in abstraction so that the Professor class could inherit the relation. Additionally, the CourseStudentInfo class was lowered a level in abstraction to allow a similar class for the Professor class.

the corresponding test passed, while "Fail" indicates that the test failed.

In addition to the results table, the expected and actual output models for the complex transformation are displayed by figures 6.5 and 6.6, respectively. Using the same method as the other M2M tests, the expected and actual instance models were compared visually and determined to be equivalent, thus passing the test. Besides the placement of some classes and relations, there is only 1 difference with the output instance model when compared to the input instance model. A CourseList object was added and related to the Professor object using the courseList relation. This displays the new ability to add CourseList objects to Professor objects implemented by the target metamodel. Additionally, when the output instance model is imported into the model editor, the University and College objects are now available in the palette. An image of the new palette can be seen in figure 6.7. This highlights a practical application of model transformations because model development can continue with the new features of the target metamodel.

## 6.3 Discussion

Each test case for evaluating the code generator passed. With these results, we can say that we have covered the code generation for each model element. Remember, this evaluation was performed to evaluate the validity of the code generation engine. With one of the main goals of IML being to demonstrate the benefits modeling can provide to the development of software systems, ensuring each model element is capable of generating valid code is important for providing users with a seamless experience that justifies the use of MDSE. Enabling the user to create models of systems and generate code with little frustration displays to the users what MDSE can provide when they incorporate it in their development processes. While this evaluation primarily targets the validity

Table 6.3: M2M evaluation, part 1. Class, Attribute and Relation tests.

| Test Case | Test Description | Test Result |
|---|---|---|
| **Class** | | |
| Class to Self | Same metamodel for source and target. Map one class to itself. | Pass |
| Class to Another Class | Different metamodel for source and target. Map a class to a different class. | Pass |
| Class to Class with Assignment | Map two classes and display ability to implement a class assignment. | Pass |
| Class to None | Map a class to none. | Pass |
| **Attribute** | | |
| Attribute to Self | Same metamodel for source and target. Map one attribute to itself. | Pass |
| Attribute Type Conversions | Map each attribute type to every other attribute type. | Pass |
| Attribute Counting Conversions | Map single attributes to attribute arrays and vice versa. Map arrays of conflicting bounds. | Pass |
| Attribute to None | Map an attribute to none. | Pass |
| Attribute with Assignment | Map to attributes and display ability to implement attribute assignment. | Pass |
| **Relation** | | |
| Relation to Self | Same metamodel for source and target. Map one relation to itself. | Pass |
| Relation to Another Same Type | Map a relation of each type to a different relation of the same type. | Pass |
| Relation to Different Type | Map a relation of each type to a different relation of a different type. | Pass |
| Relation Counting Conversion | Map single relations to relation arrays and vice versa. Map arrays of conflicting bounds. | Pass |
| Relation to Attribute | Convert an object attribute (relation) to a primitive attribute. | Pass |
| Attribute to Relation | Convert a primitive attribute to and object attribute (relation). | Pass |
| Relation to None | Map a relation to none. | Pass |

Table 6.4: M2M evaluation, part 2. Complex Mapping and Assisting in Conformance tests.

| Test Case | Test Description | Test Result |
|---|---|---|
| **Complex Mapping** | | |
| Complex Class Mapping (if-else) | Create complex class mapping with 1 condition. | Pass |
| Complex Class Mapping (if-elseif-else) | Create complex class mapping with 2 or more conditions. | Pass |
| Complex Class Mapping with None | Create complex class mapping that includes mapping a class to none. | Pass |
| Complex Attribute Mapping (if-else) | Create complex attribute mapping with 1 condition. | Pass |
| Complex Attribute Mapping (if-elseif-else) | Create complex attribute mapping with 2 or more conditions. | Pass |
| Complex Attribute Mapping with None | Create complex attribute mapping that includes mapping an attribute to none. | Pass |
| Complex Mapping with Multiple Assignments | Create a complex mapping that implements 2 or more class or attribute assignments. | Pass |
| **Assisting in Conformance** | | |
| Filling In Unmapped Required Attributes | Demonstrate attribute values are added to an attribute array if its transformed array does not satisfy the attribute's lower bound. | Pass |
| Filling In Unmapped Required Relations | Demonstrate attribute values are removed from an attribute array if its transformed array does not satisfy the attribute's upper bound. | Pass |
| Removing Extra Attributes | Demonstrate relations and its target class are added to a relation array if its transformed array does not satisfy the relation's lower bound. | Pass |
| Removing Extra Relations | Demonstrate relations and its target class are removed from a relation array if its transformed array does not satisfy the relation's upper bound. | Pass |

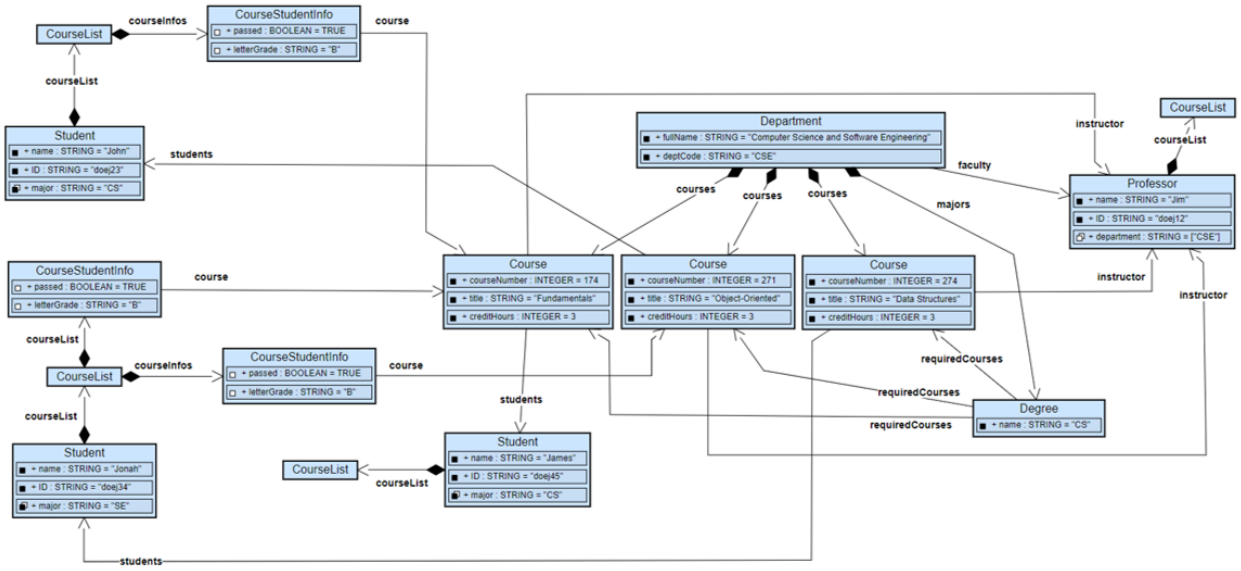



Figure 6.5: Expected output instance model when transforming from figure 6.1 to figure 6.4

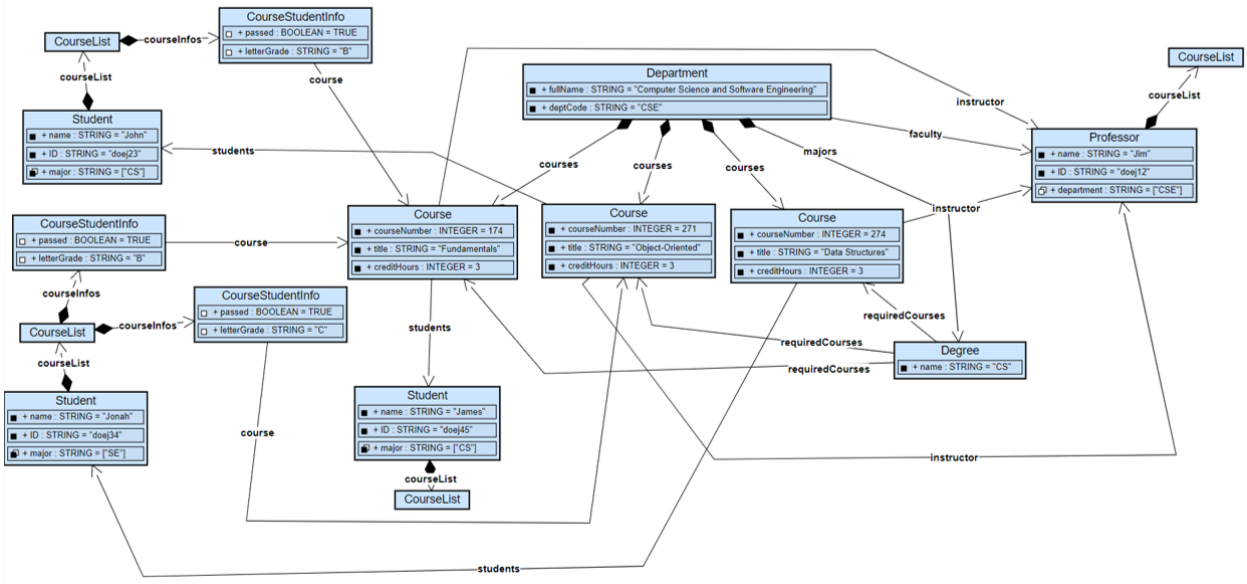



Figure 6.6: Actual output instance model when transforming from figure 6.1 to figure 6.4

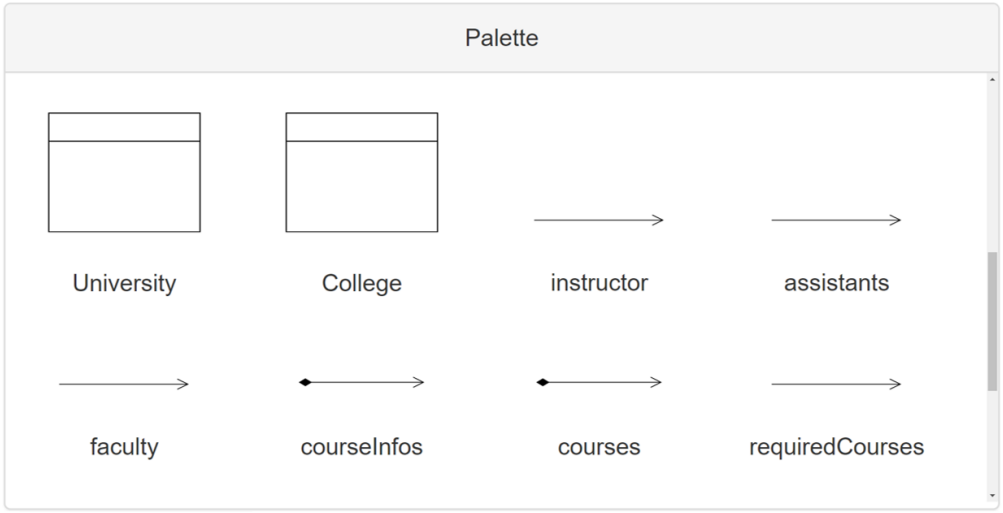



Figure 6.7: Image of the palette made by the metamodel in figure 6.4. This is used to display the University and College palette elements.

of the code generation, this supports the general goal of usable tools, as confidence in the generated code helps users become more comfortable with the processes, and promotes the adoption of MDSE. Further, our random sampling of ASCII art correctness demonstrates that this feature was correctly implemented and should support the goal of including transparent and intuitive code generation as part of the IML framework.

The model editor also shares in the effort to provide users with a good experience. For example, without the guided model creation features, it is possible that some models would generate perfectly valid code, but then result in compile/runtime errors when compiled/executed. Without our detailed guidance, a user could create an infinite recursive loop by creating a relation circle of required relations. Creating one object in the relation circle continually requires new objects in the circle until a stack overflow occurs. The model editor checks for these situations and prevents them from being modeled. While this is obviously a modeling issue, allowing the code generator to create code with runtime errors may confuse and frustrate the user. Preventing these frustrations could be the difference for convincing a software developer to change their methods and incorporate MDSE. It is these design decisions that ultimately support the goals of intuitive code generation and model transformations, which we evaluate from a technical standpoint for validity.

Each of the M2M transformation evaluation tests also passed. The comprehensive test suite covered all the possible mappings for each element type and all of the types of mappings that can be made. The process for creating the transformation rules is unique to MDSE tools. The strongly guided nature of defining rules in IML M2M transformations allows users to make model transformations quickly and easily. Similar to code generation, the goal of this process is to enable users with as little frustration as possible. These tests also showed how the model transformation engine will assist the user when their output instance models do not conform to the target metamodel by inserting/removing model elements to ensure only valid models are produced. This is important to performing model transformations effortlessly and maximizing the benefits seen by the user.

As discussed, the evaluation for the model editor was completed through the process for evaluating code generation and M2M transformations. The model editor was able to create a wide variety of models that were needed to test the second and third contributions. Creating these models did push the model editor over the edge sometimes. When this happened, we were able to discover bugs in the model editor that would not have otherwise been found. Any issues with the editor were subsequently fixed. A large amount of user frustration has been avoided by discovering these issues during evaluation rather than after deployment and has resulted in a more robust model editor.

Everything about IML was created to introduce MDSE to students and new developers by minimizing frustration and maximizing benefit. IML goes above and beyond in this aspect because it provides clarity within the benefits. For example, the code created for each attribute also includes model specific information through detailed and expressive commenting. Providing this information connects the user back to the model allowing them to clearly observe how their modeling efforts affect the code. The model editor and M2M transformation tool continue this effort through student specific features like clear error messages, suggested mappings, and visual guidance. The M2M tool UI enables performing transformations while also educating the user through its explicit design to resemble the seminal overview diagram for model transformations (as seen in Figure 2.2). Users can read about model transformations in their textbook with diagrams aiding in explanation. Then, they can move to the IML M2M transformation tool and see a similar diagram through the UI. This will allow students to more easily transfer knowledge into practical use.

# Chapter 7

# Conclusion

First, this chapter explores some threats to validity of IML and the model transformation project. Next a discussion on work that should follow the completion of this project is presented. Finally, a summary of the work completed in this project is provided.

## 7.1 Threats to Validity

Evaluating the M2M transformations used a manual comparison of the actual and expected output model files. When employing human review, there is always a risk of error. We decided to use human review because the metadata included in an .iml model file makes every file unique. For example, every model element has a unique id, automatically making the actual and expected output files different. It would require creating a parser to compare files automatically. Creating a parser for identifying relevant data to compare would require much more effort than necessary, given that the models can easily be compared visually in the IML model editor.

As stated in the code generation evaluation, some of the expected output files had to be altered to allow for automated comparison. Altering the expected model files introduces the risk of malpractice. This risk was deemed necessary because the alterations allowed for the actual and expected files to be compared programmatically, which ensures equivalence. Unlike the M2M transformation evaluation, comparing these output files manually would require a human to compare the files character-by-character. This type of human comparison would introduce more risk than is introduced by rearranging the expected output file for automated comparison. Therefore, it was decided that reordering the expected output file was necessary. To be abundantly clear, the only changes made to the expected outputs after their initial creation was the order that certain elements were printed, as this is based on internal orderings that could not possibly be known initially. This rearrangement, while it could be seen as a threat, is necessary to facilitate automatic comparisons for evaluation.

The IML framework only includes the attribute types of, String, Double, Integer and Boolean. It would have been possible to include more but it was decided this was unnecessary. The included set of types offers users enough diversity to save any type of primitive data they might need. For example, there is not a character type because this could be recreated by creating strings of length 1. Additionally, if there is an attribute type the user needs, it can implemented as an object using another class and adding attributes. Then, relating that new object to the original class.

It was mentioned earlier that the M2M transformation engine helps the user by fixing nonconformance issues. This does present a threat to properly educating students on the level of detail they must consider when making a transformation specification. Learning IML model transformations

initially could give the user false expectations when they migrate to other existing MDSE tools. This threat is considered to be outweighed by the benefits of providing an easy-to-use educational modeling tool. Some developers new to MDSE may never reach using these other tools without the help of IML. To mitigate this threat further, instructors using IML model transformations should temper their students expectations by explaining its natural simplicity for the purposes of education.

## 7.2 Future Work

This project implemented tools for modeling and performing model transformations for structural models in IML. Structural models can only be used to represent a portion of software systems. The work to be completed after this project could use the same tools and technologies to implement behavioral modeling and model transformations. Behavioral models will allow students to see MDSE can be used to generate an entire software program, instead of just creating starter code.

Model transformations are not the only advantage to MDSE. Future versions of IML could include Model Based Testing (MBT). Implementing MBT in IML will show students the benefits of MDSE across the software development life cycle. Displaying that the early efforts of modeling will give benefits throughout the life of a software system, will be essential for convincing more developers of the process.

The above discussions present future work for the IML project as a whole, however, with respect to future work on the structural modeling and model transformations, there are specific enhancements that could be considered. Specifically, an extended evaluation with users providing feedback on the experiences would only stand to further validate its usefulness as an introductory modeling tool. Additionally, we could consider the inclusion of an expanded feature set in the model editor to incorporate additional data types and relation types. While the current UML subset employed is sufficient, an extension would only serve to make our implementation more robust.

## 7.3 Summary

Model transformations are integral to MDSE and its benefits. Without them, software engineers would lose the ability to generate code and translate their programs across domains. Teaching model transformations to students of MDSE is incredibly important for expanding the use of MDSE practices. IML was created to give students a window into the possibilities of MDSE, and provide them a framework that supports the understanding of concepts without being burdened with complex technical issues. With this project, a web-based modeling tool, structural code generation and M2M transformations were implemented in IML in a manner consistent with this goal. IML can now be used to teach students the possibilities created through model transformations by leveraging our transparent and guided model transformation processes.

The model transformations and their implementations have been evaluated using a test coverage approach. Each type of IML model element was applied to an IML model and converted to code, or another model. The possible element attributes were exhausted for each element in these tests. Each test was presented in a table and shown to have passed. One test case was provided for each

evaluation as an example.

With the work completed in this project, students can simply open the IML model editor in their browser and begin modeling. With one click, students can turn their models into Java code, ready to be compiled. With a few more clicks, students can navigate to the M2M transformations page. Using this tool, students can transform their instance models into a completely new domain, or simply update them systematically to represent changing requirements. Instead of creating a new instance model by hand, students can define rules between two metamodels and generate conforming instance models automatically. IML will be used to teach the next generation of software developers. This first installment in the IML project will be used for years to come to teach developers more effective methods of software engineering.

# Appendix A

# Code Generation Complex Example Actual Output

```
Department: {
  fullName: "Computer Science and Software Engineering"
  deptCode: "CSE"
  faculty: [
    Professor: {
      name: "Jim"
      ID: "doej12"
      department: ["CSE"]
    }
  ]
  courses: [
    Course: {
      courseNumber: 174
      title: "Fundamentals"
      creditHours: 3
      instructor: [
        Professor: {
          name: "Jim"
          ID: "doej12"
          department: ["CSE"]
        }
      ]
      assistants: [
      ]
      students: [
        Student: {
          name: "James"
          ID: "doej45"
          major: ["CS"]
          courseList: [
            CourseList: {
              courseInfos: [
```

```
          ]
        }
      ]
    }
  ]
}
Course: {
  courseNumber: 271
  title: "Object-Oriented"
  creditHours: 3
  instructor: [
    Professor: {
      name: "Jim"
      ID: "doej12"
      department: ["CSE"]
    }
  ]
  assistants: [
  ]
  students: [
    Student: {
      name: "John"
      ID: "doej23"
      major: ["CS"]
      courseList: [
        CourseList: {
          courseInfos: [
            CourseStudentInfo: {
              passed: true
              letterGrade: "B"
              course: [
                Course: {
                  courseNumber: 174
                  title: "Fundamentals"
                  creditHours: 3
                  instructor: [
                    Professor: {
                      name: "Jim"
                      ID: "doej12"
                      department: ["CSE"]
                    }
                  ]
```

```
                    assistants: [
                    ]
                    students: [
                      Student: {
                        name: "James"
                        ID: "doej45"
                        major: ["CS"]
                        courseList: [
                          CourseList: {
                            courseInfos: [
                            ]
                          }
                        ]
                      }
                    ]
                  }
                ]
              }
            ]
          }
        ]
      }
    ]
  }
}
Course: {
  courseNumber: 274
  title: "Data Structures"
  creditHours: 3
  instructor: [
    Professor: {
      name: "Jim"
      ID: "doej12"
      department: ["CSE"]
    }
  ]
  assistants: [
  ]
  students: [
    Student: {
      name: "Jonah"
      ID: "doej34"
      major: ["SE"]
```

```
courseList: [
  CourseList: {
    courseInfos: [
      CourseStudentInfo: {
        passed: TRUE
        letterGrade: "C"
        course: [
          Course: {
            courseNumber: 271
            title: "Object-Oriented"
            creditHours: 3
            instructor: [
              Professor: {
                name: "Jim"
                ID: "doej12"
                department: ["CSE"]
              }
            ]
            assistants: [
            ]
            students: [
              Student: {
                name: "John"
                ID: "doej23"
                major: ["CS"]
                courseList: [
                  CourseList: {
                    courseInfos: [
                      CourseStudentInfo: {
                        passed: true
                        letterGrade: "B"
                        course: [
                          Course: {
                            courseNumber: 174
                            title: "Fundamentals"
                            creditHours: 3
                            instructor: [
                              Professor: {
                                name: "Jim"
                                ID: "doej12"
                                department: ["CSE"]
                              }
```

```
                    ]
                    assistants: [
                    ]
                    students: [
                      Student: {
                        name: "James"
                        ID: "doej45"
                        major: ["CS"]
                        courseList: [
                          CourseList: {
                            courseInfos: [
                            ]
                          }
                        ]
                      }
                    ]
                  }
                ]
              }
            ]
          }
        ]
      }
    ]
  }
  CourseStudentInfo: {
    passed: true
    letterGrade: "B"
    course: [
      Course: {
        courseNumber: 174
        title: "Fundamentals"
        creditHours: 3
        instructor: [
          Professor: {
            name: "Jim"
            ID: "doej12"
            department: ["CSE"]
          }
        ]
```

```
                    assistants: [
                    ]
                    students: [
                      Student: {
                        name: "James"
                        ID: "doej45"
                        major: ["CS"]
                        courseList: [
                          CourseList: {
                            courseInfos: [
                            ]
                          }
                        ]
                      }
                    ]
                  }
                ]
              }
            ]
          }
        ]
      }
    ]
  }
]
majors: [
  Degree: {
    name: "CS"
    requiredCourses: [
      Course: {
        courseNumber: 274
        title: "Data Structures"
        creditHours: 3
        instructor: [
          Professor: {
            name: "Jim"
            ID: "doej12"
            department: ["CSE"]
          }
        ]
        assistants: [
        ]
```

```
students: [
  Student: {
    name: "Jonah"
    ID: "doej34"
    major: ["SE"]
    courseList: [
      CourseList: {
        courseInfos: [
          CourseStudentInfo: {
            passed: true
            letterGrade: "C"
            course: [
              Course: {
                courseNumber: 271
                title: "Object-Oriented"
                creditHours: 3
                instructor: [
                  Professor: {
                    name: "Jim"
                    ID: "doej12"
                    department: ["CSE"]
                  }
                ]
                assistants: [
                ]
                students: [
                  Student: {
                    name: "John"
                    ID: "doej23"
                    major: ["CS"]
                    courseList: [
                      CourseList: {
                        courseInfos: [
                          CourseStudentInfo: {
                            passed: true
                            letterGrade: "B"
                            course: [
                              Course: {
                                courseNumber: 174
                                title: "Fundamentals"
                                creditHours: 3
                                instructor: [
```

```
                        Professor: {
                          name: "Jim"
                          ID: "doej12"
                          department: ["CSE"]
                        }
                      ]
                      assistants: [
                      ]
                      students: [
                        Student: {
                          name: "James"
                          ID: "doej45"
                          major: ["CS"]
                          courseList: [
                            CourseList: {
                              courseInfos: [
                              ]
                            }
                          ]
                        }
                      ]
                    }
                  ]
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}
CourseStudentInfo: {
  passed: true
  letterGrade: "B"
  course: [
    Course: {
      courseNumber: 174
      title: "Fundamentals"
      creditHours: 3
      instructor: [
        Professor: {
```

```
                    name: "Jim"
                    ID: "doej12"
                    department: ["CSE"]
                  }
                ]
                assistants: [
                ]
                students: [
                  Student: {
                    name: "James"
                    ID: "doej45"
                    major: ["CS"]
                    courseList: [
                      CourseList: {
                        courseInfos: [
                        ]
                      }
                    ]
                  }
                ]
              }
            ]
          }
        ]
      }
    ]
  }
  Course: {
    courseNumber: 271
    title: "Object-Oriented"
    creditHours: 3
    instructor: [
      Professor: {
        name: "Jim"
        ID: "doej12"
        department: ["CSE"]
      }
    ]
    assistants: [
    ]
```

```
students: [
  Student: {
    name: "John"
    ID: "doej23"
    major: ["CS"]
    courseList: [
      CourseList: {
        courseInfos: [
          CourseStudentInfo: {
            passed: true
            letterGrade: "B"
            course: [
              Course: {
                courseNumber: 174
                title: "Fundamentals"
                creditHours: 3
                instructor: [
                  Professor: {
                    name: "Jim"
                    ID: "doej12"
                    department: ["CSE"]
                  }
                ]
                assistants: [
                ]
                students: [
                  Student: {
                    name: "James"
                    ID: "doej45"
                    major: ["CS"]
                    courseList: [
                      CourseList: {
                        courseInfos: [
                        ]
                      }
                    ]
                  }
                ]
              }
            ]
          }
        ]
```

```
            }
          ]
        }
      ]
    }
    Course: {
      courseNumber: 174
      title: "Fundamentals"
      creditHours: 3
      instructor: [
        Professor: {
          name: "Jim"
          ID: "doej12"
          department: ["CSE"]
        }
      ]
      assistants: [
      ]
      students: [
        Student: {
          name: "James"
          ID: "doej45"
          major: ["CS"]
          courseList: [
            CourseList: {
              courseInfos: [
              ]
            }
          ]
        }
      ]
    }
  ]
}
```

# References


[1] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*, volume 1. 09 2012.

[2] client.io, March 2021.

[3] Object Management Group (OMG). OMG Unified Modeling Language (OMG UML), Version 2.5.1. OMG Document Number formal/2017-12-05 (`https://www.omg.org/spec/UML/`), 2017.

[4] Robert N Charette. Why software fails [software failure]. *IEEE spectrum*, 42(9):42–49, 2005.

[5] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013.

[6] Douglas C. Schmidt. Model-driven engineering, February 2006.

[7] Stuart Kent. Model driven engineering. In Michael Butler, Luigia Petre, and Kaisa Sere, editors, *Integrated Formal Methods*, pages 286–298, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[8] Thomas Kühne. Matters of (meta-) modeling. *Software & Systems Modeling*, 5:369–385, 2005.

[9] G. Nordstrom, J. Sztipanovits, G. Karsai, and A. Ledeczi. Metamodeling-rapid design and evolution of domain-specific modeling environments. In *Proceedings ECBS'99. IEEE Conference and Workshop on Engineering of Computer-Based Systems*, pages 68–74, March 1999.

[10] Ethan Jackson. The software engineering of domain-specific modeling languages: A survey through examples. Technical Report ISIS-07-807, Institute For Software Integrated Systems (ISIS), March 2008.

[11] Juha-Pekka Tolvanen and Matti Rossi. Metaedit+: Defining and using domain-specific modeling languages and code generators. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 92–93, New York, NY, USA, 2003. ACM.

[12] S. Sendall and W. Kozaczynski. Model transformation: the heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, Sep. 2003.

[13] Yuehua Lin, Jing Zhang, and Jeff Gray. Model comparison: A key challenge for transformation testing and version control in model driven software development. In *OOPSLA Workshop on Best Practices for Model-Driven Software Development*, volume 108, page 6, 2004.

[14] Thomas Buchmann and Alexander Rimer. Unifying modeling and programming with alf. In *SOFTENG*, 2016.

[15] Leslie Lamport. Computation and state machines. April 2008.

[16] N. Hili, J. Dingel, and A. Beaulieu. Modelling and code generation for real-time embedded systems with uml-rt and papyrus-rt. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 509–510, May 2017.

[17] M. Levy, D. Raviv, and J. Baker. Data center simulations deployed in matlab and simulink using a cyber-physical systems lens. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0465–0474, Jan 2019.

[18] Jia-Chun Lin Olaf Owe and Elahe Fazeldehkordi. A flexible framework for program evolution and verification. In *International Conference on Model-Driven Engineering and Software Development - MODELSWARD*, 2019.

[19] Cedric Brun and Alfonso Pierantonio. Model differences in the eclipse modelling framework. *UPGRADE*, 2008.

[20] Eric J. Rapos. *Supporting Simulink Model Mangement*. PhD thesis, Queen's University, 2017.

[21] Paul Brna Patrick Mendelsohn, T.R.G. Green. *Psychology of Programming*, chapter 2.5 - Programming Languages in Education: The Search for an Easy Start, pages 175–200. Academic Press, 1990.

[22] R.C Holt and J.R. Cordy. The turing programming language. *Communications of the ACM*, 1988.

[23] Rupert Wegerif. The role of educational software as a support for teaching and learning conversations. *Computers & Education*, 43(1):179 – 191, 2004.

[24] Edward L Thorndike. *The principles of teaching: Based on psychology*. Routledge, 2013.

[25] Sue Duchesne and Anne McMaugh. *Educational psychology for learning and teaching*. Cengage AU, 2018.

[26] WILLIAM R. PENUEL, JEREMY ROSCHELLE, and NICOLE SHECHTMAN. Designing formative assessment software with teachers: An analysis of the co-design process. *Research and Practice in Technology Enhanced Learning*, 02(01):51–74, 2007.

[27] Ludovic Apvrille Hui Zhao and Frédéric Mallet. Meta-models combination for reusing verification techniques. In *International Conference on Model-Driven Engineering and Software Development - MODELSWARD*, 2019.

[28] Eclipse modeling framework (emf), March 2019.

[29] Martin Robillard. Lightweight software modeling with jetuml, December 2010.

[30] Jean Bézivina Frédéric Jouaulta, Freddy Allilairea and Ivan Kurtevb. Atl: A model transformation tool. *SCICO*, 2008.

[31] Matthew Stephan and Andrew Stevenson. A comparative look at model transformation languages. 01 2009.

[32] A. Pataricza D. Varró, G.Varró. Designing the automatic transformation of visual languages. *Journal of Science of Computer Programming 44*, 2002.

[33] Matthew Stephan and James R. Cordy. A survey of model comparison approaches and applications. *MODELSWARD 2013 - Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development*, 01 2013.

[34] Dilan Sahin, Marouane Kessentini, Manuel Wimmer, and Kalyanmoy Deb. Model transformation testing: a bi-level search-based software engineering approach. *Journal of Software: Evolution and Process*, 27(11):821–837, 2015.

[35] Epsilon, May 2019.

[36] Eric J. Rapos and Matthew Stephan. Iml: Towards an instructional modeling language. In *International Conference on Model-Driven Engineering and Software Development - MODELSWARD*, 2019.

[37] Omar Badreddin. Umple: a model-oriented programming language. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 337–338. IEEE, 2010.

[38] Eric J. Rapos. We'll make modelers out of 'em yet: Introducing modeling in to a curriculum. In *International Conference on Model-Driven Engineering and Software Development - MODELSWARD*, 2018.

[39] diagrams.net, March 2021.

[40] client.io, March 2021.

[41] *Model-View-Controller Pattern*, pages 353–402. Apress, Berkeley, CA, 2009.

[42] Oracle. Providing Constructors for Your Classes. (`https://docs.oracle.com/javase/tutorial/java/javaOO/constructors.html`), 2020.