SuMo: A Supportive Modeling Language Environment for Guided Model Transformations

Nick DiGennaro, Matthew Stephan, and Eric J. Rapos
Computer Science & Software Engineering
Miami University
Oxford, Ohio, USA
{digennnj,stephamd,rapose}@miamioh.edu

***Abstract*—Adoption of model-driven software engineering is limited by the steep mastery curve of leading approaches and their associated technologies. To help combat this, we research and develop a supportive modeling language and environment, SuMo, that ensures modelers are able to produce valid models, model transformations, and generated artifacts in their development projects. SuMo includes a custom modeling language definition, a structure- and semantic-enforcing live modeling environment to support model creation, a transparent code generation engine that includes model element mapping to better integrate modeling and programming languages, and a guided model-to-model transformation engine to help users through the complex task of transformation specification. Each of these guided approaches to model design and transformation development are included in a single, web-based, environment to avoid complex configurations, which often cause issues for novice developers. We conduct a systematic evaluation that assesses SuMo's code generation and model-to-model transformation processes independently. We conclude that both types of model transformations developed using SuMo produce valid and correct outputs in all cases.**

***Index Terms*—modeling languages, live modeling, language customization, model transformations, code generation, structural modeling, modeling tools, supportive modeling**

I. Introduction

Model-driven software engineering (MDSE) has seen a continual increase in usage in industry [1], educational contexts [2], and research [3]. This sparks a need for appropriate processes to ensure the development and maintenance of valid modeling artifacts. Further, model transformations are some of the more complex modeling concepts, thus a suitable modeling language and corresponding environment would prove useful in the creation and editing of models to ensure valid model transformations can occur.

Model transformation typically takes one of two forms [4]. **Model-to-Text (M2T) Transformations** allow for conversion to some textual representation of the graphical model. The most basic form of M2T transformation comes in model serialization, where the models are converted to textual format for file storage; however, this type of transformation is not the most useful application. In order to leverage M2T transformations fully, developers aim to leverage a specific subset known as Code Generation. Through mappings from model elements to source code grammar elements, modeling tools are able to transform input models to executable source code in a given target language [5]. In contrast, **Model-to-Model (M2M) Transformations** allow the conversion from one type of model to another, or from one version to another. In both transformation types, the inputs and outputs must conform to a given specification; in this case their respective meta-models. In M2M transformations, the mappings between elements are far more variable and require a user to specify their own mapping, typically using a transformation specification language, such as ATL [6]. Regardless of which type of transformation is applied, the notion of conformance is central to model transformations. Any input model must conform to a specification in order to be a valid input to the transformation, and any output produced by the transformation must conform to its target definition to be useful in the resulting domain. Supporting applications of model transformation must ensure this conformance on both sides of the transformation.

A. Motivation

Existing implementations of model transformation tools leave much of the transformation specification process up to the user with minimal guidance and feedback. This open form specification process can lead to malformed transformation rules that result in invalid transformations or non-conforming instance models. Due to the complex conceptual nature of model transformations, this inability to ensure valid transformations may be a detriment to the widespread adoption of MDSE techniques. Through a lack of proper guidance and live modeling feedback, novice users may be unaware of the issues in their transformations, and this could lead to reinforcement of improper modeling practices. Issues with tool implementations have contributed to student frustration in the adoption of MDSE techniques [7]. To address such issues and facilitate continued adoption and growth of MDSE, we devise a guided supportive modeling language and accompanying environment that aim to provide novice and experienced modelers with the guidance and feedback to produce accurate and valid software systems through the use of MDSE techniques, specifically structural modeling, code generation, and model-to-model transformations. This aligns with the vision proposed in our past work in our proposal and justification for an Instructional Modeling Language [8] and techniques aimed at Agile MDSE [9].

B. Contributions

In researching and developing our guided and supportive modeling process, specifically relating to structural model transformations, we aim to answer two research questions. **RQ1:** Does the provision of a supportive live-modeling environment, coupled with a transparent code generation engine, ensure consistent and valid outputs? **RQ2:** Does the inclusion of responsive guidance and feedback allow for consistent generation of valid model-to-model transformations?

To this end, we develop **SuMo**, our process for **Su**pportive **Mo**deling, which makes the following contributions,

- a custom UML variant, along with a supportive web-based live-modeling environment, which enables the creation of structural models for use in transformations;
- a transparent code generation process that clearly maps model artifacts to resulting code, facilitating a stronger integration of models and source code;
- a guided model-to-model transformation engine that directs users through transformation processes;
- a demonstration of the effectiveness of the above contributions through a systematic evaluation.

A summary of closely related works is presented in Section II. We present an overview of the browser-based modeling environment that implements SuMo in Section III, followed by the specific details of the SuMo language and process in Section IV. SuMo was evaluated using the process outlined in Section V. Finally, we conclude this paper with some discussions of the project and its future in Section VI.

II. Related Work

Various other modeling languages and tools exist with similar functionality (e.g., Sirius/Acceleo, Modelio, Rational Software Architect); however, the common difference between them and SuMo is their lack of guidance and support, which is provided through our process and language design. Some of the more closely related works are presented in this section.

While the Eclipse Modeling Framework (EMF) [10] provides functionality for meta-model creation, the tool itself has a steep learning curve for novice users [7] and has a core functionality that lacks many of the desired features of an end-to-end modeling tool. Specifically, to leverage instance model creation, users must also install and become familiar with the Graphical Modeling Project (GMP)[1] to be able to edit their instances graphically. Our SuMo process and corresponding realization include both meta-modeling and instance modeling in one web-based environment. While the Eclipse Modeling Project does offer a number of options for model transformation, such as ATL [6], Epsilon [11], JET [12], and several others, they also involve incorporating additional tools into the framework with added dependencies, learning curves, and version issues. SuMo incorporates code generation and model transformations into the same environment as its structural model editor, so users can focus on producing consistent models and transformations without needing to context switch between various complex tools. Our customized UML variant that we employ as SuMo's modeling language allows users to master the principle concepts rather than needing to understand the full complexity of these other implementations. Lastly, our web-based implementation of SuMo provides a level of access and live-modeling support not present in EMF, allowing us to make advances in the state-of-the art for modeling languages and environments.

Another significant difference from existing model transformation tools is that our solution implements model-to-model transformations using a guided approach that does not require the user to learn a new specification language, such as ATL [6]. Instead our live editor provides drop-downs and simple text entry options to ensure they are able to consistently produce well-formed transformations that always result in valid transformed model outputs.

Umple [13]–[15] is a web-based framework for model-driven development of object-oriented systems that allows users to specify their system design using UML class diagrams and ultimately generate Java code for use in production systems. However, Umple does not provide the ability to create instance models based on the users' meta-models, nor does it feature any model-to-model transformation functionality. Both of these are core features and affordances of SuMo. Umple also involves an intermediate textual representation of the class diagrams, which requires users to become familiar with an additional syntax, and distracts from the abstraction provided by graphically modeling systems. In contrast, SuMo uses only a graphical modeling language, thus removing the need to learn an additional language syntax.

We originally proposed a vision for an instructional modeling language with similar motivation [8]. However, SuMo is more general purpose and instead aims to provide users with consistent guidance and feedback to produce valid models and generated artifacts in every instance. We draw inspiration from their intended use of a constrained UML subset in order to focus on the supportive modeling aspects rather than technical robustness.

III. Browser-Based Live-Modeling Environment

In this section, we describe the technical context of SuMo, which enables the provision of guidance and support. Specifically, the browser-based live-modeling environment that we develop our research goals within. We chose a browser-based application to remove technical and installation barriers to its use. We aimed to develop a customized modeling environment to provide users support during the development of their models and model-transformations. We discuss our research and main contributions in the form of our supportive approaches in Section IV.

While the environment itself is not our main contribution, it is necessary in the development of our model transformation processes. To leverage the power of model transformations, a user must first be able to produce models of various types as inputs to their transformations. Further, the supportive and guided aspects that we build into the modeling environment

[1] https://www.eclipse.org/modeling/gmp/

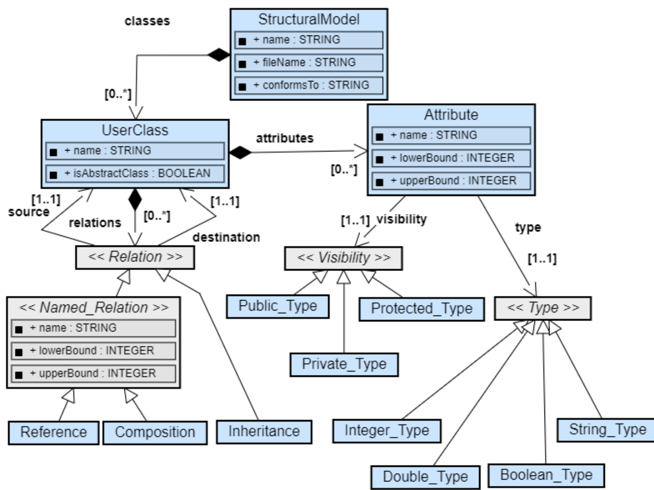

Fig. 1. Language definition meta-model for Structural Models - a customization of UML Class Diagrams

form the basis of our supportive web-based process contribution.

The remainder of this section establishes the context and environment that we developed to enable our guided transformation processes. Specifically, we present the structural live model editor, a functional integrated development environment (IDE) to interact with the generated code, and a model transformation specification application.

A. Structural Live Model Editor

The first aspect of the environment enables users to develop models that can be leveraged in various types of applications while also being conducive to both educational and Agile contexts. We opted to initially focus on structural models, similar to those represented as Class Diagrams in the Unified Modeling Language (UML), as they are more commonly used in model transformations, based on the current set of applications available. Instead of using standard UML, we developed a simplified customization of UML class diagrams to focus on modeling concepts rather than full expressability. Specifically, we opted to include only classes (standard or abstract), attributes based on four primitive data types (integer, double, string, and boolean), and three types of relations between classes (inheritance, reference/association, and composition). We chose a custom variant over a UML profile to avoid unnecessary complexity for users. We show our language definition meta-model, defined using our own editor, in Figure 1. Since one of the main aspects of model transformations relates to the conformance relationship between meta-models and their instances, we ensure that SuMo is capable of producing meta-models and conforming instance models. This essentially required the creation of two similar, but distinct, model editors.

To leverage the successes of existing approaches, it is important that we present users with elements found in leading model editors such as the Eclipse Modeling Framework (EMF). Specifically, we included a modeling pane, a dynamic

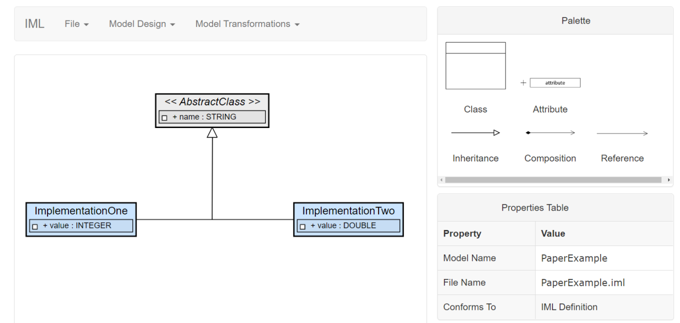

Fig. 2. Meta-model editor showing the modeling pane, palette, and properties table

palette, and a properties table. For instance model editing, we additionally included a meta-model conformance indicator to provide dynamic feedback to the user.

Users are able to develop their own meta-model based on our UML customization, and opt to create instance models that conform to their own model by switching to the instance model editor. In doing so, the dynamic palette displays classes and relations from their user-defined meta-model. This is a feature that is not present in existing model editors, at least without substantial additional effort on the part of the user, such as the UI definitions in GMP.

To research and realize these editors, we used the open source JointJS[2] framework. By leveraging its customizable element definitions, we created a graphical model editor that represented our specific UML customization. We present an example of the meta-model editor in Figure 2. This example shows two classes in inheritance relations with an abstract class as a means of showing classes, attributes, and relations having been added to the model pane.

Our live-modeling environment includes some quality-of-life features that enable robust modeling in an end-to-end fashion. Specifically, we offer the opportunity to import/export models in an XML format for artifact sharing, produce images of models, save and open models on server-based accounts, and several other familiar features common to existing frameworks. From the structural modeling page, users are able to generate corresponding source code automatically, which is displayed to them in our web-based IDE.

B. Integrated Development Environment

While there are several web-based IDEs for editing and executing source code, it is crucial that our modeling environment incorporates our own implementation to provide users maximal benefit from code generation. When users generate source code from their structural models, we are able to immediately display their generated code in a fully functional IDE. Because each model typically contains many class objects, the resulting generated code contains an equal number of source code files. The main function of the IDE with respect to SuMo's guided model transformations is its ability to facilitate immediate interaction with the generated code, corresponding

[2]https://www.jointjs.com/opensource

```
Home   File ▾   Edit ▾   View ▾   Run ▾   Help ▾
+   <   >   AbstractClass   ImplementationOne   ImplementationTwo
1  /*
2   * ==========================
3   * ||      AbstractClass     ||
4   * ==========================
5   * || [0..1] + name : STRING ||
6   * ==========================
7   */
8
9  package iml.paperexample;
10
11 public abstract class AbstractClass {
12
13     /**
14      * @lowerBound 0
15      * @upperBound 1
16      */
17     public String name;
18
```

Fig. 3. An example of generated code shown the custom IDE (cropped)

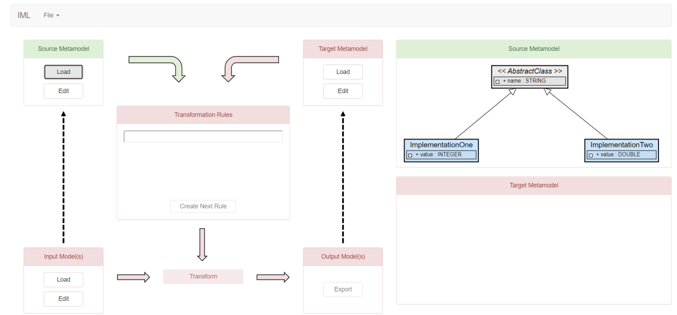

Fig. 4. An example of the model transformation specification environment with a source meta-model selected

to the model elements. We discuss this link between models and the generated code further in Section IV-B.

We realized our IDE through Ace.js[3], an existing framework for JavaScript-based code editing that includes a set of standard IDE features. We enable users to build and compile their code by running the compiler in a Docker instance. Similar to the model editor, we were determined to include the standard features and components found in leading IDEs. Specifically, we included a central text editor allowing multiple tabs, a console for compilation and execution, and a project explorer to view saved code projects. We present an example of the IDE in Figure 3, which illustrates the code generated from the example model in Figure 2.

C. Model Transformation Specification Interface

The final aspect of our supportive modeling process is the guided model transformation specifications. Correspondingly, our environment needs to support our goal of exploiting a users' familiarity of basic concepts. Thus, it is important that our interface represents the foundational knowledge base of model transformations. As such, we mirrored a widely-accepted overview diagram for explaining model transformations based on a leading text on MDSE [2] and others [4] to inspire our layout and design. We present our model transformation interface in Figure 4. In this example, the user has selected their source meta-model, which is shown to the user during the specification process for a quick lookup of model contents. This figure shows our realization of the overview diagram as an interactive specification interface where users can specify source and target meta-models, choose their input instances, specify mapping rules, trigger the transformation, and export the resulting transformed models. While the use of this interface has potential benefits in educational contexts, it offers functionality suitable for robust implementations as well.

Beyond the layout of the interface, the tightly guided approach to the specification of transformation rules was crucial in the design of our approach. Unlike other model transformation approaches, which typically leverage custom transformation languages, our approach uses a specification wizard to guide users through the process by having them provide mappings, values, and other specifications through modals and popups. By removing the free-form inputs, we remove the possible introduction of malformed specifications that could otherwise occur. We discuss this further in Section IV-C.

[3]https://ace.c9.io/

IV. SuMo - Supported Modeling and Transformations

To effectively guide users through the various applications of model transformation, we provide support in three distinct ways. First, by *providing consistent feedback during the design of their meta- and instance models*, we ensure that the inputs to any model transformation are both well formed and adhere to standard modeling practices. Further, we assure they meet the criteria for code generation. Second, by *enabling a transparent and consistent code generation process*, not only are we ensuring the user is consistently provided valid and functional generated code, we provide additional features to support the correspondence between model and code artifacts. Finally, through our *guided transformation specification engine*, we ensure that any model transformations specified by the user produce valid transformations that will always conform to the target meta-model. We discuss each of these goals herein.

A. Consistent Feedback for Supportive Model Design

To reap the benefits and power of model transformations, a user must first be able to produce valid and meaningful models, both at the meta-model and instance-model levels. To support this prerequisite, SuMo provides support in our live-modeling environment through a number of specific facets, all of which rely on our custom modeling language included in the SuMo environment. In the abstract sense, this takes the form of consistent live feedback and preventing "breaking" changes. We now discuss the specific support subprocesses within SuMo: type checking, relation cycle prevention, dynamic conformance checking, detailed error messages, and stable state assurance.

To ensure the validity of the data represented by the models, we must ensure that any values entered by the user satisfy all modeling-language constraints. This includes basic type checking to ensure the values are of the correct data type, bounds checking to ensure lower bounds are always less than or equal to the upper bound, preventing multiple inheritance, and other basic error checking. This, seemingly, simple guidance goes a long way to ensure that any model produced

contains only valid values. If an invalid value is entered by a user in the properties table at any point, our process includes providing a detailed error message describing the issue and reverting the model to its most recent stable state.

The next type of guidance relates to the dynamic checking of modeling concepts pertaining to cycles of relations. There are cases where adding a relation between two classes will result in a cycle (such as a Class inheriting from itself). Rather than allowing the addition of the relation and producing an error when processing the model, SuMo informs the user immediately through an error message and reverts the model back to a stable state.

Another sub process within SuMo is providing immediate feedback to the user when they are creating instance models that conform to a provided meta-model that ensures the validity of all model elements. The first feature that supports this goal is the dynamic generation of the palette. Since the palette contains the Classes and Relations specified in the meta-model, our process allows the user to add only valid elements to their instance model. The second feature is a dynamic conformance checker that reports any issues of non-conformance to the user in the meta-model conformance pane. Examples of non-conformance include the absence of required attribute values or relations and the surplus/deficit of relations outside of specified bounds. As with the other features, we constructed the messages to be as detailed as possible to best guide the user on how to correct the issues of non-conformance.

Through each of these model guidance features, SuMo prioritizes ensuring users are provided with clear error messages that provide sufficient detail to correct the issues. One example of this includes identifying when invalid values, based on data type, are entered for attributes. Further, our process does not allow for the creation of models in an unstable state. Each time a breaking change is introduced, in addition to providing the detailed message, our process includes reverting back to the most recent stable state. Due to the immediate nature of the error checking, only single-step atomic changes will be rolled back, immediately following the message describing the error. The only exception is in the case of meta-model non-conformance, which does not equate to an unstable model. In this case, SuMo allows the user to rectify these issues on their own. However, an instance model with conformance issues cannot be used to generate code. Thus, SuMo requires an instance model to be in conformance with its meta-model before proceeding with code generation.

B. M2T: Transparent Code Generation

Code generation for meta-models and instance models must be handled in different manners, as they conceptually represent different levels of abstraction. As meta-models represent all possible instances within a domain, this equates to defining the structural data representations for each of the various classes. However, instance models represent a single representation of one instance, equating to providing specific values to each of the classes' attributes. Independent of the model type, the code generation process ensures that the user is provided with a single source code project with appropriately named directories that match the model name, further ensuring correspondence between the model and generated code. We present SuMo's general approach in the form of generation of Java code for meta-models and instance models, followed by our intentional aspects of the process that enable a transparent code generation experience. It was necessary to implement our own custom code generation engine due to the absence of the transparency features desired for SuMo in existing environments.

Generating the necessary code for a meta-model means creating an equivalent Java class for each class in the user-defined meta-model. To this end, our process entails producing independent *.java* files for each class, complete with all of their respective attributes and outbound relations. To achieve this, our process involves the code generation engine traversing the model, class by class, and creating a textual representation of each class following a series of static mapping rules. The general process for constructing each class is as follows,

1) add the class definition, including any *abstract* or *extends* notations
2) add attribute definitions for each attribute in the model class, based on the properties
3) add two constructors for the class: one default constructor with no parameters and one constructor with parameters for all attributes
4) add a *toString* method for the class, responsible for printing the class details in a readable format
5) add *getters* and *setters* for each of the attributes in the class

This general process ensures that every element from the user-defined meta-model is present in the generated code. The guaranteed one-to-one mapping enables the generalization of this process to any given meta-model, ensuring that our process is applicable in all cases. A similar process is applied by EMF.

Since the instance relies on the definitions provided by the meta-model that it conforms to, to generate instance model code our code generation process and engine must first ensure that all class definitions from the meta-model have been generated as we described. Once complete, SuMo traverses the input model to create a final *.java* file that houses the instantiation of the model elements. The general process for generating instance model code is as follows,

1) create a class definition based on the instance model's name
2) add a main method to the class to be used to instantiate all of the model's objects
3) add a call to the default constructor for each class present in the instance model, providing a unique identifier for each class using an incremental counter approach
4) for each newly constructed instance, use the generated setter methods to instantiate the object with the attribute values provided in the instance model
5) after all classes are instantiated with the attribute values,

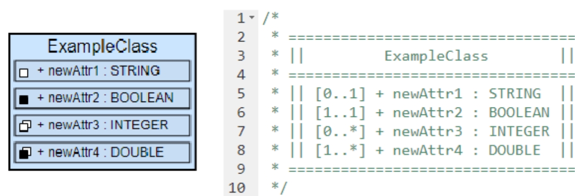

Fig. 5. Example class with corresponding generated ASCII-art representation

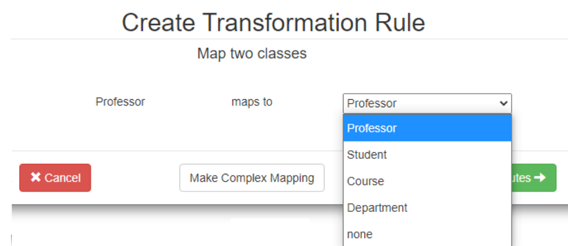

Fig. 6. An example class mapping with fixed LHS and drop-down RHS

add the relations from the instance model by calling the appropriate setter method for the relation's source object and assigning the relation's destination object

As with our process for meta-model code generation, this iterative and exhaustive method for generating instance model code ensures that all elements present in the input model are directly mapped to the resulting code. This process is unique to our code generation process as other approaches do not employ both types of code generation in a single facility.

To provide modelers with an obvious equivalency between their graphical model and the generated Java code, thus furthering and supporting the integration of the models and source code, it is beneficial for our process to include a code generator that adds a header comment to every source code file that serves as a visual link back to the source model. To this end, we constructed an ASCII-art generator to build the class object as it appears in the model. One nuance is, rather than using the iconography for cardinality, we represent the upper and lower bounds as a range in the comment. An example of this mapping between a class from the model editor and the generated ASCII art file header is in Figure 5.

In addition to the header comments, SuMo's code generation incorporates detailed documentation to link the code back to the original model in several other ways. For example, for each attribute, we generate comments that include its upper and lower bounds specified by the model, as these elements are not directly represented in the generated executable code. To aid in the readability and transparency of instance model generation, our engine sections the code into 3 main blocks based on the high-level tasks it achieves. Each block of code is preceded with a comment explaining the purpose of the code that follows. The three block comments are as follows,

- `// Instantiate model objects.`
- `// Set object attributes.`
- `// Implement relations between model objects.`

Within each of the blocks, we put spaces between each object when handling its attributes and relations to further support the transparency and readability of the generated code. Essentially, our goal is that as little information as possible is lost in the conversion from the graphical model to the generated code, and that each element in the generated code can be easily traced back to the original source model.

C. M2M: Guided Model Transformations

Arguably, model-to-model transformations are one of the more complex tasks associated with MDSE. This is typically compounded by novices having to learn a new syntax to specify transformation rules, which can lead to malformed rules and invalid transformations. SuMo's solution to an effective model transformation specification is to take a guided approach, providing the user with as much structure and predefined content as possible. In addition to our UI based on the overview diagrams, SuMo also guides the user through the model transformation specification by highlighting remaining required tasks in red and completed tasks in green. We showcase this in Figure 4. This complements the real power of our transformation specification process, the rule specification mechanisms.

Rather than an open-rule specification platform, SuMo guides the user through the creation of necessary rules to produce a valid mapping. This is done by pre-populating the left-hand side (LHS) of applicable rules with the elements of the source meta-model such that the user is informed of which elements must be mapped. Rather than leaving the right-hand side (RHS) of the rule completely open, our SuMo process offers the user a choice via a drop-down of all valid options from the target meta-model. As such, it is simply a matter of choosing the intended mapping target from a list. This includes the option to map to no element in the target meta-model, should the transformation dictate removing elements entirely. Currently, SuMo does not support composing multiple elements from the LHS to a single element on the RHS, but this is something we consider for future iterations. An example mapping specification for a class mapping is shown in Figure 6. Here we see the LHS fixed with the current model element from the source meta-model (Professor) and a prepopulated drop-down with all allowable mappings from the the target meta-model on the RHS. The same approach is also used for attribute and relation mappings within already mapped class pairs.

To facilitate the expressiveness required to produce relevant and meaningful transformations, SuMo allows for more complex mappings than the simple 1-to-1 mappings we have described thus far. Thus, our transformation engine provides options for complex conditional mappings for each element. Rather than allow only simple rules such as `Professor maps to Professor`, users can add conditions such as `if Professor.ID.contains("_") then Professor maps to none, else Professor maps to Professor`, which would remove all professors with an ID containing an underscore and ignoring those without. Instead of having the user learn the syntax of these rules, SuMo's engine provides drop-downs and text boxes to streamline the process. The

Fig. 7. An example conditional mapping rule shown in the SuMo wizard

engine allows for additional `else if` clauses. This example rule describing the removal of some professor objects, created using the editor, is in Figure 7. The use of a wizard-based approach rather than employing existing constraint languages, such as OCL [16], means modelers do not need to worry about mastering yet another language, while still providing similar expressability.

Another feature necessary for robust model transformation is the ability to assign values to attributes that cannot be derived from attributes in the source models, but rather are based on some set of application rules. Consider an example where the source meta-model represents a person's child as one of two classes, `Son` or `Daughter`, but the goal of the target meta-model is to merge these both to a single `Child` class with an attribute that indicates their gender. To do this, it would require a rule of the form `Son maps to Child AND Child.gender="male"` and an equivalent rule for the other class. Our engine continues the application of SuMo's guided approach by providing the option to assign values, which are checked for valid assignments based on type, using input text boxes.

The final important feature of SuMo's model transformation process and our corresponding engine that supports effective model transformation is the ability to ensure the creation of valid output models through *enforced target meta-model conformance*. That is, before producing an output model, SuMo applies the dynamic conformance checking algorithms provided by the instance model editor to ensure the resulting model conforms to the target meta-model. If there are no conformance issues, the model is generated and exported by the engine for use. However, if conformance errors are present, our intelligent approach determines the specific issues and applies solutions to ensure conformance. This involves the application of any combination of three adjustments to the model, presented below. Each adjustment is known to be correct based on the language definitions of the structural models, and the built-in conformance checking algorithm.

Attribute Instantiation: If, through the application of rules and processing of the input model, there are attributes in the resulting model that are not instantiated and the target meta-model requires a value assignment, our transformation engine must determine the appropriate instance value by either applying the default value from the meta-model or the type-default for the attribute type.

Relation and Object Addition: If through the application of rules and processing of the input model there are classes with fewer outbound relations than allowed by the target meta-model, our transformation engine instantiates default instances of the required classes to meet the constraint requirement and adds relations between the source and new targets.

Relation Removal: Inversely, if through the application of rules and processing of the input model there are classes with more outbound relations than allowed by the target meta-model, our transformation must remove some relations to bring it into conformance. Since it is not desirable to delete the class elements themselves as the user may opt for different selections, our approach removes the requisite number of relations using a simple lexicographic-based heuristic.

Following SuMo's application of the above corrections across the full model, conformance checking occurs again to ensure the model now conforms to the target meta-model. If it conforms, the transformation is complete and the result is generated for the user. If there are still conformance issues at this point, which is unlikely given the strict guidance provided, the user is notified that the specification is invalid, and no output is produced. The SuMo process opts to produce no output over an invalid model as the latter would be more of an issue should the user continue using the resulting model within their software project.

V. Evaluation

To evaluate our two research questions, we aimed to evaluate the code generation and model transformation processes separately using independent experiments. Evaluation of the model-editor environment and guided support process was evaluated implicitly through its use in producing the artifacts used in the other two evaluation experiments. That is, by using the modeling environment to develop meta-models and instance models for use in code generation and model transformation, we were able to provide extensive internal testing of the tool that allowed for the discovery and correction of bugs present in the process. While it may be desirable to perform user testing in the future, we were more concerned with the evaluation of our research aspects rather than the tooling itself. Further, it allowed us to explicitly focus on answering our two research questions, which focus explicitly on the validity and correctness of the code generation and model transformation processes. We make all artifacts from our evaluation available on our public repository[4].

A. Code Generation

To effectively test the validity and correctness of the code generation process, thus answering **RQ1**, we opted for a path-coverage based approach. For each element in the language definition in Figure 1, we developed a test case to evaluate that specific portion of the code generator. To this end, we

[4]https://doi.org/10.5281/zenodo.5227220

enumerated all elements, along with their valid permutations to ensure path coverage for our code generation processes. By leveraging this path coverage criteria and testing each model element, independently of one another to reduce noise, we demonstrate the complete functional coverage of the SuMo code generation process. Through this process we identified the 51 cases in Table I.

For each of the functionalities, we created an input meta-model and a conforming instance model containing the elements required to demonstrate that functionality. To reduce noise, we built up tests incrementally when possible, first testing smaller portions, then using those models as the basis for functionality that builds upon the successful test cases. For example, we evaluated attributes without assigning values before evaluating the assignment of values. To minimize the number of tests required, we evaluated related features together in combined test models. This resulted in 13 explicit test models to evaluate the code generation process. Finally, we developed a single complex model that demonstrates the composition of the functionalities to evaluate code generation for a more robust and realistic model.

For each test model, we chose to evaluate the generated code on four metrics of success. The first metric pertains to the ability to generate code from the model. Since SuMo's code generation engine is designed to not generate code from a model that would produce invalid code, if any code is generated, this is the first marker of success. The second metric of evaluation relates to the correctness of the ASCII art in the source files. For each test model, we manually inspected each of the the 36 generated file headers. If the header contained an ASCII art header, and it accurately and correctly represented the class, that file was correct. A test was successful in this criterion if all files had correct representations in their generated file headers. The third metric measured the validity of the code generated by the engine. To evaluate this, we leveraged the Java compiler. If the code was compiled without error, we deemed this code to be valid, and this was the success marker for the test case as a whole. The final success criterion for a test case was whether the actual system output matched the expected outputs. Using an automated comparison of the expected and actual outputs, we obtained the final marker for success for each test case. To summarize, for a test case to pass, it must: a) generate source code, b) have valid ASCII art in all source code files, c) compile using the Java compiler with no errors, and d) have the actual and expected outputs match exactly. We represent these four criteria in the columns of Table I as they form the basis of our evaluation results.

We created expected outputs manually by producing text files that represented the console output that should result when executing the compiled code. In the case of this experiment, we augmented the code generator to add print statements at the end of the main methods so that the instances' objects would be printed to the console using their *toString* methods. This printing of object instances formed the basis of the expected outputs. To perform the automated comparison of expected and actual outputs, we used a Bash script to compile and run the generated code and redirect the console output to a file, which we stored for comparison. After executing all the code, we automatically compared the expected and actual outputs and recorded the results.

B. Model Transformations

Similar to the code generation approach, to effectively evaluate **RQ2** we aim to leverage a coverage approach to model transformations. However, instead of covering model elements, this evaluation covers all possible mapping combinations for each of the model element types. This yields 4 Class Level Tests, 5 Attribute Level Tests, and 7 Relation Level Tests. Additionally, we aimed to cover complex mappings through an additional 7 Complex Mapping Level Tests to ensure adequate handling of these more complex mapping rules. Finally, it was necessary to test the ensured conformance of our model transformation engine by providing scenarios that would intentionally produce invalid output models, and demonstrate that SuMo ensures they conform to the meta-model through application of SuMo's methods/subprocesses. This final category led to an additional 4 Conformance Level Tests. Through our mapping coverage approach we developed a total of 27 test scenarios, which we present in Table II.

For each of the 27 scenarios, we created four input elements: a source meta-model, a target meta-model, a set of mapping rules, and an input instance model. When possible we leveraged models that were evaluated previously by us, either through code generation, or models used in earlier model transformation tests. We used homogeneous model transformations, where the source and target meta-model are the same, in cases where the mapping coverage called for elements to map to itself.

To determine success, we chose two criteria for evaluation. The first is the ability to produce the resulting model. As with code generation, since the transformation engine will not produce an output if it does not conform to the target meta-model, the first success criterion is whether a model is produced as a result of the transformation process. The second criterion is a comparison of expected and actual outputs, using an evaluation process we describe herein. If both of these metrics, which are indicated by the columns in Table II, are successful, we consider the test case to have passed.

For each test scenario, we manually produced an instance model that conformed to the target meta-model to represent the expected output of applying the transformation rules to the input model. We used the 27 expected outputs for comparison against the observed outputs of the transformation process.

To run the evaluation, we used the model transformation engine to apply the rules and produce outputs, one scenario at a time and stored the resulting models. Following the production of the 27 observed outputs, we performed a manual comparison by two authors independently to verify their equivalence. In this manual, point-by-point, inspection, we ignored non-semantic changes, such as position within the model. We considered correctness in this case to be when the actual and expected outputs matched based on our criteria.

C. Results & Discussion

To answer **RQ1**, we rely on the results of our code generation evaluation. In particular, we identified 51 scenarios represented by 13 test cases to provide sufficient coverage, each evaluated on four criteria. The results of this evaluation are in Table I, where each row represents a scenario and the four criteria are shown in the resulting columns. Through this experiment, we demonstrated that every test scenario produced valid code that also included links to the original source models by way of ASCII art headers. 100% of the test cases passed, including the additional complex example, providing confidence in our code generation engine, and providing a positive answer to the research question. Through the SuMo process, our code generation engine is able to consistently produce valid and correct code for every valid input model.

To answer **RQ2**, we rely on the results of our model transformation evaluation. In this experiment we identified 27 specific scenarios to provide sufficient coverage of model transformation, mapping types and model element types, and execute a transformation for each scenario. The results of this experiment are in Table II, where each row represents the transformation scenario and the results columns represent the success criteria. As with code generation, we found that every test scenario passed in both criteria, thus providing a 100% correctness rate for the model transformation engine. Through the consistent guidance and feedback provided during model transformation specification, the SuMo process ensures the consistent generation of valid outputs in a model-to-model transformation for all evaluated input models.

In our evaluation, we were able to answer both research questions in the affirmative. This provides solid evidence that the SuMo guided and supportive process and M2M and M2T model transformation engines are consistently able to produce valid outputs. These positive results indicate that the guidance provided by SuMo prevents the creation of invalid models for use in code generation and model transformation. Additionally, our guided and supportive approach model transformation specification ensures that only valid outputs can be produced for future use. The SuMo process we developed is intended to work with our intentionally constrained UML customization. However, we predict in general that the provision of guidance and feedback throughout supportive modeling and transformation processes will improve the overall quality of the resulting model transformations.

TABLE I
CODE GENERATION EVALUATION RESULTS

Functionality Covered	T	G	A	C	M
Class Level Tests					
Normal Class	ClassEmpty	✓	1/1	✓	✓
Abstract Class	ClassAbstract	✓	1/1	✓	✓
Attribute Level Tests					
Integer Attribute	AttributesType	✓	1/1	✓	✓
Double Attribute					
String Attribute					
Boolean Attribute					
Integer Attribute with Value	AttributesType AndValue	✓	1/1	✓	✓
Double Attribute with Value					
String Attribute with Value					
Boolean Attribute with Value					
Integer Type Default	AttributesType AndDefaultValue	✓	1/1	✓	✓
Double Type Default					
String Type Default					
Boolean Type Default					
Public Attribute	Attributes Visibility	✓	1/1	✓	✓
Protected Attribute					
Private Attribute					
Single Optional (0..1)	Attributes Bounds AndArray Values	✓	1/1	✓	✓
Single Required (1..1)					
Multiple Optional (0..m)					
Multiple Optional (0..*)					
Multiple Required (1..m)					
Multiple Required (1..*)					
Multiple Required (m..n)					
Multiple Required (m..*)					
Relation Level Tests					
Inherit from Abstract Class	Relations Inheritance Primitives	✓	3/3	✓	✓
Inherit Integer					
Inherit Double					
Inherit String					
Inherit Boolean					
Inherit Reference	Relations Inheritance Objects	✓	6/6	✓	✓
Inherit Composition					
Multiple Inheritance					
Single Optional (0..1)	Relations Reference Bounds	✓	8/8	✓	✓
Single Required (1..1)					
Multiple Optional (0..m)					
Multiple Optional (0..*)					
Multiple Required (1..m)					
Multiple Required (1..*)					
Multiple Required (m..n)					
Multiple Required (m..*)					
Single Optional (0..1)	Relations Composition Bounds	✓	8/8	✓	✓
Single Required (1..1)					
Multiple Optional (0..m)					
Multiple Optional (0..*)					
Multiple Required (1..m)					
Multiple Required (1..*)					
Multiple Required (m..n)					
Multiple Required (m..*)					
Self Reference Relation	Self Reference	✓	2/2	✓	✓
Self Composition Relation	Self Composition	✓	2/2	✓	✓

Legend: **T -** Test Case Name
G - Code Successfully Generated?
A - ASCII Art Correctly Generated? (X / Y files correct)
C - Code Successfully Compiled?
M - Matched Expected Output

VI. CONCLUSION

To conclude we present several retrospective views of this research and aim to adequately frame our results. Specifically, we acknowledge threats to the validity of our approach, discuss potential future work to expand this process, and finally present a summary of key results and conclusions.

A. Threats to Validity

The first threat to validity in this research is related to the fact that, while one of the main goals of this work was based on easing the burden of the model transformation process, we opted not to perform any user studies to assess this aspect. Instead, we chose for this evaluation to focus solely on the validity of the transformations as a means of supporting the SuMo processes. While a user study would further support this claim, we deemed it most appropriate to first conduct an internal validation and evaluation before recruiting and experimenting with users.

Another potential threat is the manual evaluation processes in our methods. Specifically, the manual creation of expected outputs in both experiments, and the manual comparisons we performed for the ASCII art and M2M transformation output evaluations. To mitigate these threats, two independent members of the research team evaluated the expected outputs for correctness. We employed the same method for the point-by-point evaluations of the ASCII art and M2M transformations.

TABLE II
MODEL TRANSFORMATION EVALUATION RESULTS

Functionality Covered	Transform?	Match?
Class Level Tests		
Class to Self	✓	✓
Class to Another Class	✓	✓
Class to Class with Assignment	✓	✓
Class to None	✓	✓
Attribute Level Tests		
Attribute to Self	✓	✓
Attribute Type Conversions	✓	✓
Attribute Counting Conversions	✓	✓
Attribute to None	✓	✓
Attribute with Assignment	✓	✓
Relation Level Tests		
Relation to Self	✓	✓
Relation to Another Same Type	✓	✓
Relation to Different Type	✓	✓
Relation Counting Conversion	✓	✓
Relation to Attribute	✓	✓
Attribute to Relation	✓	✓
Relation to None	✓	✓
Complex Mapping Level Tests		
Complex Class Mapping (if-else)	✓	✓
Complex Class Mapping (if-elseif-else)	✓	✓
Complex Class Mapping with None	✓	✓
Complex Attribute Mapping (if-else)	✓	✓
Complex Attribute Mapping (if-elseif-else)	✓	✓
Complex Attribute Mapping with None	✓	✓
Complex Mapping with Multiple Assignments	✓	✓
Conformance Level Tests		
Filling In Unmapped Required Attributes	✓	✓
Filling In Unmapped Required Relations	✓	✓
Removing Extra Attributes	✓	✓
Removing Extra Relations	✓	✓

B. Future Work

SuMo currently focuses solely on structural models based on our customized version of UML class diagrams. However, much of the expressability afforded by MDSE approaches comes from the ability to model system behavior in addition to structure. Thus, we have plans to extend our modeling environment and code generation facilities to include a similar UML custom variant based on statecharts, specifically in the style UML-RT statecharts and capsule diagrams. By adding additional behavioral capacities, we hope to expand the usefulness/applicability of the SuMo approach. Another planned extension is to allow for code generation to additional languages, including user-defined targets.

To assess the effectiveness of these methods in facilitating understanding and applying model transformations, it is beneficial to perform user studies that target these aspects. It is our goal, once we complete behavioral features and extensions, to plan and perform a comprehensive user study, potentially in a classroom setting. Users will provide feedback on the process, and we can compare SuMo with existing tools based on quantitative and qualitative metrics.

C. Summary

Through our research and development of **SuMo**, our Supportive Modeling process, which we realized via a supportive modeling language and live-modeling environment, and two separate transparent and guided model transformation engines, we were able to assess two crucial research questions. Regarding **RQ1**, we determined that our code generation process was able to generate valid code in 100% of input models. Further, the generated code was found to be automatically documented with elements linking directly to the input models, aiding in transparency of the code generation process. In this experiment, validity meant code both compiled and produced output that exactly matched the expected output. With respect to **RQ2**, we found that the strongly guided model-transformation specification engine was also able to produce valid results in 100% of the input models in our evaluation. This definition of validity includes a demonstration that even incomplete transformation specifications still produce valid outputs due to our conformance assisting algorithms. Every input model and transformation specification produced correct and valid output, without issue. We believe this work is an important step towards supportive live-modeling. By facilitating an exploratory modeling process and environment, through our customized UML variant and guided approaches, we support the creation and maintenance of high-quality model artifacts.

REFERENCES

[1] J. Hutchinson, M. Rouncefield, and J. Whittle, "Model-Driven Engineering Practices in Industry," in *International Conference on Software Engineering*. Waikiki, Honolulu, Hawaii: ACM, 2011, pp. 633–642.

[2] M. Brambilla, J. Cabot, and M. Wimmer, "Model-Driven Software Engineering in Practice," *Synthesis lectures on software engineering*, vol. 3, no. 1, pp. 1–207, 2017.

[3] A. Bucchiarone, J. Cabot, R. F. Paige, and A. Pierantonio, "Grand Challenges in Model-Driven Engineering: an Analysis of the State of the Research," *Journal of Software and Systems Modeling*, pp. 1–9, 2020.

[4] K. Czarnecki and S. Helsen, "Feature-Based Survey of Model Transformation Approaches," *IBM Systems Journal*, vol. 45, no. 3, 2006.

[5] Z. Hemel, L. C. Kats, and E. Visser, "Code Generation by Model Transformation," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2008, pp. 183–198.

[6] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A Model Transformation Tool," *Science of computer programming*, vol. 72, no. 1-2, pp. 31–39, 2008.

[7] E. J. Rapos, "We'll Make Modelers Out of 'Em Yet: Introducing modeling Into a Curriculum," in *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2018, pp. 130–134.

[8] E. J. Rapos and M. Stephan, "IML: Towards an Instructional Modeling Language," in *Proceedings of the International Conference on Model-Driven Engineering and Software Development*, 2019, pp. 417–425.

[9] S. W. Ambler, "Agile Model Driven Development is Good Enough," *IEEE Software*, vol. 20, no. 5, pp. 71–73, 2003.

[10] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[11] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The Epsilon Transformation Language," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2008, pp. 46–60.

[12] Eclipse Consortium *et al.*, "Java Emitter Templates (JET)," 2003.

[13] O. Badreddin, "Umple: a Model-Oriented Programming Language," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, 2010, pp. 337–338.

[14] M. A. Garzón, H. Aljamaan, and T. C. Lethbridge, "Umple: A Framework for Model Driven Development of Object-Oriented Systems," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2015, pp. 494–498.

[15] T. C. Lethbridge, V. Abdelzad, M. H. Orabi, A. H. Orabi, and O. Adesina, "Merging Modeling and Programming Using Umple," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2016, pp. 187–197.

[16] J. Cabot and M. Gogolla, "Object Constraint Language (OCL): a Definitive Guide," in *Int'l school on formal methods for the design of computer, communication and software systems*. Springer, 2012.