MOLEGA: Modeling Language for Educational Card Games

Kaylynn Borror
borrorkn@miamioh.edu
Computer Science & Software Engineering, Miami University
Oxford, Ohio, USA

Eric J. Rapos
rapose@miamioh.edu
Computer Science & Software Engineering, Miami University
Oxford, Ohio, USA

Abstract

Domain-specific modeling languages abstractly represent domain knowledge in a way that users can more easily understand the model content without technical expertise. These languages can be created for any domain, provided the necessary knowledge is available. This research uses educational game design as a demonstration of the power of domain-specific modeling. Games are useful tools in supplementing the traditional education of students, however, many educators often do not possess the design or technical skills to develop a custom game for their own use. MOLEGA (the Modeling Language for Educational Card Games) is a domain-specific modeling language that provides a guided model design environment for these users. Using MOLEGA, users can create visual models, inspired by UML class diagrams, to represent their desired card game, based on two selected variants. User models are then used to generate executable source code for a mobile-compatible, browser-based game that can be deployed on a server by following the provided instructions. MOLEGA is evaluated for validity and correctness using a suite of example models.

CCS Concepts: • **Software and its engineering → Development frameworks and environments**; **Domain specific languages**; **Visual languages**; • **Applied computing → Interactive learning environments**.

Keywords: domain-specific modeling, domain-specific modeling languages, game design, educational games, code generation, web applications, model-driven software engineering

1 Introduction

Model-driven software engineering (MDSE) uses models to abstractly represent software systems throughout the engineering process [3]. This approach to software design and development provides an abstract representation of a complex problem and solution. One of the major benefits provided through MDSE is the use of domain-specific modeling languages (DSML). DSMLs abstractly represent software systems inside a domain in a way where non-technical users can more easily understand the information the model is presenting, but do not require advanced technical knowledge of programming and engineering skills [5]. One specific domain that has the ability to leverage the power of DSMLs is that of educational game design.

Many people who want to use games to their benefit often do not possess the design or implementation skills necessary to write code for a game. Educators often fall into this category, wanting to use games to enhance their students' learning experiences but not being experienced developers themselves. Having a way for an educator to create their own classroom aids without the knowledge on how the aids work would be extremely helpful for the educator. This environment is a perfect demonstration of the power of DSMLs.

This paper presents MOLEGA (the **Mo**deling **L**anguage for **E**ducational Card **Ga**mes), a DSML that allows educators and other users to create web-based card games for usage in classrooms or similar settings. User models are created using our MOLEGA web editor, which provides a guided and supportive live-modeling environment that consistently ensures that only valid models are created. From these models, fully functional code for the game can be generated in a format where it can be deployed in a web-based environment with minimal technical knowledge required, following detailed instructions. This model-to-text code generation provides functional code without the user needing any understanding of the model transformation process.

Domain-specific modeling has gained traction due to its ability to facilitate the development of technical systems by those with limited technical expertise. However, limited research exists on the usage of DSMLs in educational games. Specifically, no published research exists for the usage of DSMLs for card-based educational games.

While there are previous instances of code generation engines that transform from class-based languages, little research exists for code generation from models to web-based languages, such as Javascript. This work aims to further expand the application of DSMLs to a new target, both in terms of domain and technical implementation.

While DSMLs aim to solve many issues surrounding a lack of technical expertise, many languages are often still very complex, leading to barriers in their adoption. To combat this, we also believe that a DSML must provide its users with adequate feedback and guidance during the modeling and code generation processes such that they are able to consistently generate functional code without having to worry about minor semantic or syntactic details.

To demonstrate further applications of domain-specific modeling, this paper poses two research questions:

- **RQ1:** Can domain-specific modeling be used to create web-based educational card games?
- **RQ2:** Does a guided framework ensure the generation of consistently correct executable game code?

In responding to these research questions, we also make the following contributions:

- creation of a DSML for the definition of custom web-based educational card games
- implementation of a model-to-text code generation engine to produce executable web-based code
- implementation of a complete web-based framework integrating the DSML and code generation processes, applied to two example game types
- systematic evaluation of the code generation process covering all aspects of the two example games

2 Background & Related Work

Given the nature of the venue, a base understanding of DSMLs and language design is assumed. However, it is important to also understand the chosen application domain of education games, as well as examine some of the closest related works. This section will explore these topics in detail.

2.1 Educational Game Design

Educational games, or "edugames," are a type of game that are used to aid in learning. While traditional games have the goal of creating a solely enjoyable experience, edugames have the primary goal of educating the players while also offering an enjoyable experience as a secondary effect. To serve their purpose, edugames must follow both traditional game design principles and pedagogical principles. This can be difficult, since game designers and education experts often do not possess enough knowledge about each others' domain areas to work completely independently [2].

Card-based activities are a common educational tool, often seen in the form of memorizing flash cards. However, card-based games have also been shown to have a positive effect on learning. A 1998 study involved teaching students about gastrointestinal physiology through the use of modified versions of Go Fish and Gin Rummy [9]. A similar study in 2011 required pharmacy students to play games based on the same two card games, three times each over a six-week period. The pharmacy study found that the student participants had an overwhelmingly positive reception to the card games and felt that it contributed to their learning [1]. More recently, a custom variant of the popular game *Cards Against Humanity* has been used in teaching engineering ethics [4].

2.2 Related Work

Work by Prasanna [10] and the DSML GLiSMo [12] [13] are related to this work. For both works, DSMLs were created to represent different aspects of game development, such as choices that the player character can make at each stage of the game, along with areas where the insertion of mathematical problems is valid. Neither of these languages offer any code generation capabilities in their published research. Rather, they are meant to be used as visualization tools for a user to follow along with in order to understand the progression of game events from beginning to end.

Zahari et al. proposed an extension to the GLiSMo DSML, called FA-GLiSMo [14]. This extended DSML intends to represent educational adventure games while adopting elements to encourage Flow Theory: a learning theory that describes the state of complete engagement to an activity. FA-GLiSMo intends to build upon GLiSMo's drawbacks, aiming to embed elements in the learning theory into educational games represented by the language.

Eterovic et al. offer an abstract visualization of the connections between Internet of Things (IoT) technologies [7]. This approach, based on UML diagrams, allows both technical and non-technical users to configure the plan of their own IoT systems. This language was tested through the use of human interaction with evaluations done on two types of user groups: those who had UML experience and no IoT experience and those who had experience in neither topic.

SharpLudus is a code generation environment intended for generating action-adventure games through the use of domain-specific languages (DSL) [8]. This environment's DSL, SLGML, is focused around defining the game world, allowing the representation of elements like rooms and their design, non-player characters and their actions, and specifications for when a player character lives or dies. SharpLudus generates C# classes in response to receiving valid SLGML diagrams.

MOLEGA differs from these related works in several ways. Unlike previous works that define DSMLs for edugames, this research not only defines a DSML for a different type of edugame (i.e., card games), but also incorporates a code generation algorithm which allows a user to use the DSML to represent a game they want to exist, then to actually be able to create it. Rather than allowing a user to create a game's objects and the flow of gameplay, the MOLEGA allows the user to specify any type of game included in the DSML's metamodel, along with the ability to customize a variety of features involved with the chosen game type.

3 MOLEGA - Modeling Language for Educational Games

MOLEGA is a DSML that allows users to create models representing educational card games. The MOLEGA web editor environment provides a completely guided model creation experience, which includes metamodel conformance checking, dynamic type checking, the prevention of breaking changes, and descriptive error messages. This section begins

by discussing MOLEGA's target games and concludes with a broader look at the language framework.

3.1 Target Games

In our initial implementation of MOLEGA, two different game types were chosen to represent varied functionality: Community Judge and Relations, discussed below.

The rules of Community Judge type games are almost identical to those of Cards Against Humanity [1] or Apples to Apples [2]. During a game turn, one player is designated the Judging player, drawing a question card which displays a prompt, and all other players must play one response card from their hand. After all other players have submitted their card choices, the Judging player chooses which of the played cards they feel best fits their card's prompt. The player who played the chosen card gains a point and the Judging player title is moved to the next player. One round has passed when all players have had a chance to judge. The player with the most points at the end of a certain number of rounds, or the player who reaches a certain score first, wins.

The rules of Relations games are a modified mixture of Gin-Rummy and Go Fish. During a player's turn, they can choose cards in their hand that are related to one another. They can do this as many times as they see fit during their turn. The player can also click on their opponent's area to see their related card collections at any time. When they have made all of their decisions, the current player can then either pass the turn or discard a card in their hand while passing their turn. This power then moves on to the next player in line. The player with the most points at the end of a certain number of rounds, or the player who reaches a certain score first, wins.

Community Judge is a good choice for the first type of game due to the popularity of nearly identical games like Cards Against Humanity, a variant of which has been used in teaching engineering ethics [4]. Additionally, this type of game doesn't have a strict rule structure, since the winner of a turn is determined by player opinion and not an in-game mechanic. This makes it easier to customize the content on both card decks. Relations, however, being a mixture of games with stricter rule structures, needs a little more attention to ensure that the game behaves correctly to player input. This is done by making sure the related cards are listed correctly in the card file. While the card file setup may be a little more complex than Community Judge, Relations is another valid target for MOLEGA. Previous literature shows that when used in an educational context, modified versions of Go Fish and Gin Rummy are beneficial for student learning [1] [9].

As it is not possible to create a DSML to model every possible educational card game variant in order to answer **RQ1**, the selection of these two types of games, along with their included variants, aims to provide a representative sample that is sufficient in supporting the research question. By choosing two significantly different game types with multiple variations, MOLEGA serves as a proof of concept realization and demonstration of the power of domain-specific modeling to represent educational games without the need to provide full coverage. The contents of the game cards, regardless of the game type, are provided by CSV files, which are filled in by the user after generation.

The implementation of these target games is done in a Node.js environment. To handle client-server interactions simply and smoothly, the Socket.io library [3] is used. Node.js was chosen for this reason. While Socket.io has been adapted for other languages such as Java, C++, and Python, it was originally written as a Javascript framework.

The games are coded in a way where both the mobile and browser versions are readable and scaled to size. The difference between browser play and mobile play can be seen in Figure 1, Figure 2a, and Figure 2b. Figure 1 displays the game interface for a computer browser player, while Figures 2a and 2b display the game interface for a mobile player, where the first figure shows the game table and players while the second is scrolled down on the mobile device, showing the player's hand of cards.

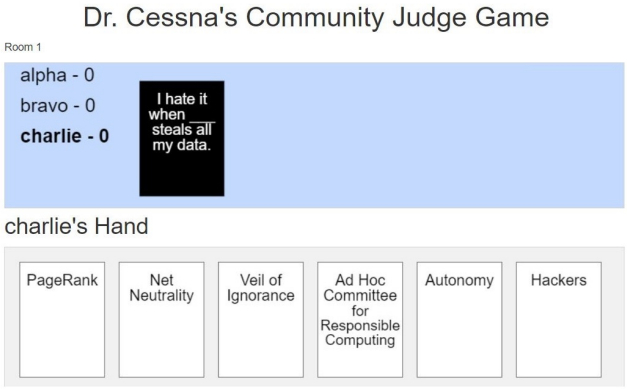


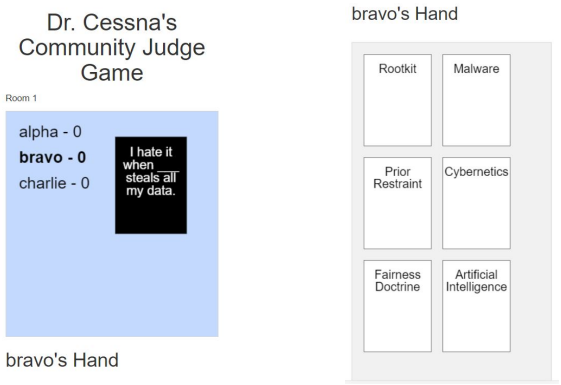
Figure 1. Browser Community Judge Play

To know what sort of customizable attributes that the modeling language should include, these target games were created first. By having example targets complete and working, it was possible to find the variation points between different versions of the various games, which informed the language design.

3.2 MOLEGA Framework

MOLEGA, (***Mo**deling **L**anguage for **E**ducational Card **Ga**mes*) is a DSML and environment for the development of the two target types of games. MOLEGA supports customization

[1] https://cardsagainsthumanity.com/
[2] https://www.mattelgames.com/games/en-us/family/apples-apples
[3] https://socket.io/



(a) Table View **(b)** Hand View

Figure 2. Mobile Version of Community Judge

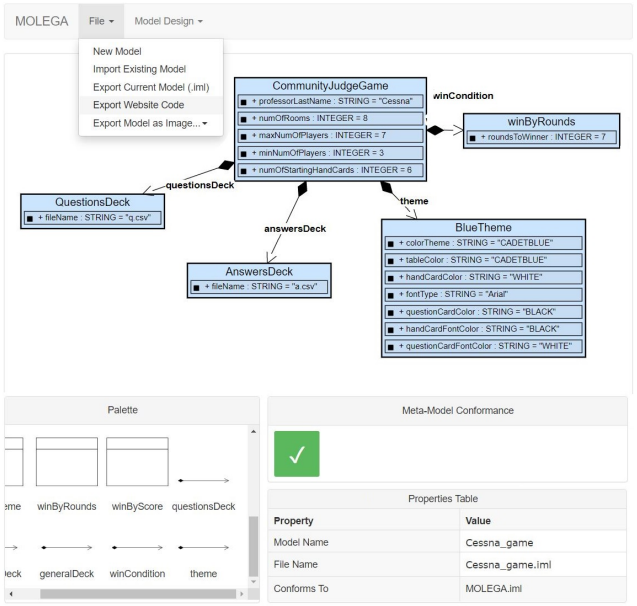


Figure 3. MOLEGA Web Editor with File Menu Icon Expanded (control panels typically appear to the right, as seen in Figure 5)

of both game setup elements as well as game content and several rules.

Beyond simply being an editor for MOLEGA models, the framework provides guided model-time support to users to help ensure the models they produce are valid and consistent, thus ensuring only correct code is generated for use. This support is provided in several ways. First, the editor provides a constant conformance check to ensure that all rules are being met for the game under development. Should a user fail to meet any requirement of the meta-model, the Meta-Model Conformance pane indicates this with a red X (rather then the normal green check) and provides a detailed message indicating how to resolve the issue. If a model contains conformance errors, the code generation functionality is disabled, preventing the environment from generating invalid code. Beyond this conformance checking, the MOLEGA framework provides type checking to ensure valid values are provided for attributes; not only does it check for type matching, but it provides context-sensitive value checking as well, such as ensuring the minimum number of players is always less than or equal to the maximum number of players. Finally, the editor ensures that any breaking change is prevented, the error is identified, and the change is rolled back to the most recent stable state. This ensures that the model is rarely in a state where incorrect code could be generated.

The web editor layout and *File* menu options are seen in Figure 3. More details of the editor contained in the MOLEGA framework are discussed in Section 4.2.

4 Domain Specific Modeling Language Design

To answer **RQ1**, we defined a DSML to represent the web-based educational card games discussed in Section 3.1. This section describes the design and implementation of that language, and is organized by explaining the metamodel design for MOLEGA, followed by the web editor and environment for MOLEGA and the design choices associated with it.

4.1 Metamodel Design

MOLEGA's metamodel was created using a modified version of UML class diagrams provided by the Instructional Modeling Language (IML) [6, 11]. MOLEGA's metamodel was designed to represent the two chosen target games, which can be seen in Figure 4, and is discussed in further detail in this section.

This metamodel's design assists in answering **RQ1**. DSMLs are meant to be useful in accurately representing domain software in an abstract way. Attributes in the model should encompass the domain it is representing. For MOLEGA to be a useful DSML, attributes for the different class types were determined by first creating the example target games. Each example game contains variation points at which customization of the system can occur. These include colors of specific in-game elements, the way that a player wins a game, and several others. The classes and attributes of MOLEGA were designed based on those variation points.

This metamodel design allows all customizable components of the target to be represented in any model generated by a user. For either of the classes that inherit from «Game», the listed attributes (professor's name, number of rooms, maximum number of players, etc.) can be customized in order to meet the user's needs. Similarly, in the various «Theme» classes, different colors of different pieces of the

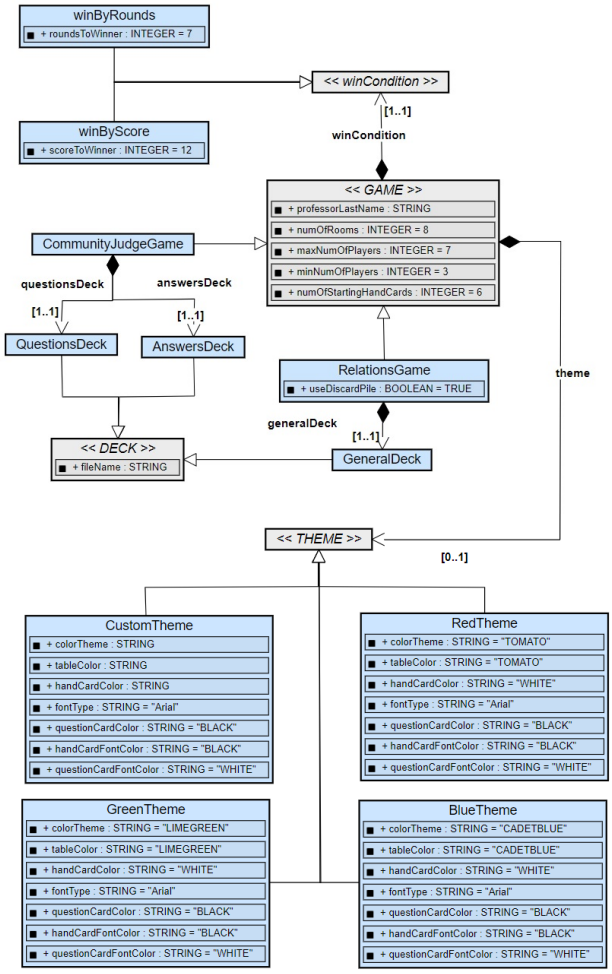


Figure 4. MOLEGA Metamodel

target are also fully customizable, with multiple attributes allowing multiple different colors.

4.2 Web-Based Model Editor

The web editor for creating MOLEGA model instances is a modified version of the structural modeling web UI for IML [6]. While the IML web editor allows for the import of any IML-type metamodels to use for creating models, the MOLEGA web editor has the MOLEGA metamodel as the default and only metamodel to use. MOLEGA's web editor can only be used to generate games as defined by MOLEGA's syntax. The UI for MOLEGA's web editor with no model created is seen in Figure 5, which shows the initial error explaining that at least one game object is needed to conform to the meta-model.

This decision was made for a few reasons. IMLs structural modeling editor already had a finished built-in model conformance check. This saved time in generating MOLEGA's web editor since more time could be spent on implementing error checking from elements such as user input into the

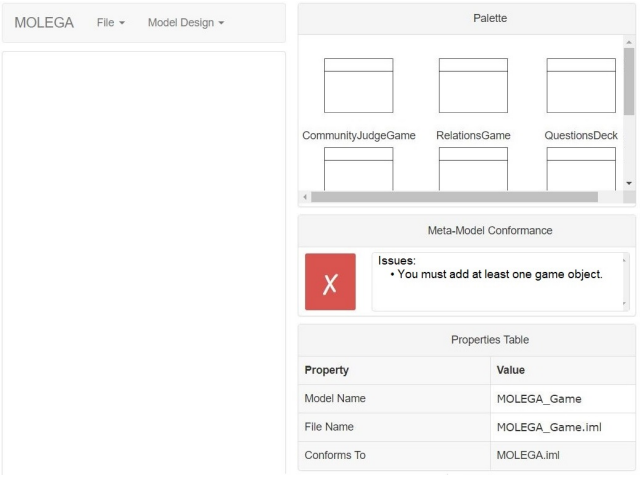


Figure 5. MOLEGA Web Editor (main model pane on left side scaled down for readability of other content)

model rather than building the entire conformance checking algorithm from scratch. The *Metamodel Conformance* panel in the editor makes identifying model errors convenient and allows for a dynamic report of errors rather than returning all model conformance issues at the time of code generation.

The browser-based nature of the editor also makes it a good decision for use. Users are not required to download a piece of software they may not be familiar with and any required plugins or modules to make it work. They can instead go to the website where MOLEGA is hosted and complete the entire process in one single framework. Since the target of the code generator is a web-based game, the modeling editor also being web-based keeps the inputs and outputs consistent.

Aside from restricting the IML editor to only support MOLEGA-defined models and related functions, other enhancements were made in order to make the MOLEGA modeling editor more robust. These modifications were made to ensure that the models created in the editor not only conform to the MOLEGA metamodel, but also that the semantic details inside the model accurately represent a valid game. For example, MOLEGA checks that only one game object is defined in the modeling environment at a time. Other modifications made to the web editor include checking that all numeric values are positive, making it impossible that the maximum number of players is less than the minimum number of players, and forcing any colors entered into the modeling environment to be in a format interpretable by a web program (e.g., a CSS or hexadecimal color code).

5 Code Generation

To answer the first part of **RQ2**, particularly the **generation** of the code, we present the details of our code generation processes. The entire code generation process is implemented

using Javascript within the MOLEGA framework, generating corresponding functional code based on the model of the game developed in the editor. This section covers the prerequisites for code generation as well as how the code generation process works.

5.1 Generation Prerequisites

In order for successful code generation to take place, a valid model must first be created in the modeling environment. A valid model is a model that completely conforms to the MOLEGA metamodel and its constraints, which is communicated to the user via the *Metamodel Conformance* panel on the model editor's UI. When this panel displays a red error marker (✗) and a list of issues, the model does not fully conform to the MOLEGA metamodel. Only when the panel displays a green success marker (✓), and the model is considered valid, is code generation possible.

A valid Community Judge model consists of four required components and one optional component. A Community Judge game must have a *CommunityJudgeGame* class, a *winCondition* class, a *QuestionsDeck* class, and an *AnswersDeck* class. The optional component is a *Theme* class. Meanwhile, a valid Relations model consists of three required components and one optional component. A Relations game must have a *RelationsGame* class, a *winCondition* class, and a *GeneralDeck* class. The optional component is a *Theme* class.

5.2 Transformation Process

The code generator is triggered by a Javascript function when the "Export Website Code" menu option is clicked. A psuedocode listing of the code generation process is seen in Figure 6. The first step in this process is to transform the model in the editor into a format where the contents can be parsed for class and attribute names and values. The serialization of the model leverages the existing IML algorithm. This serialization is then parsed into an xml format using jQuery's built-in *parseXML()* function in order to use the xml tags to navigate the model's elements rather than manually parsing the String output.

To ensure that the classes being parsed are connected to one another using the appropriate composite relationships, the *Relation* tags are especially important. Each *Relation* tag is composed of multiple elements, the two most important for this task being the *source* element and the *destination* element. The *source* element lists the id for the source class of the composite, in the case of MOLEGA models this class is always either a *CommunityJudgeGame* class or a *RelationsGame* class. The *destination* element's value is the id for the connecting class. If a class exists in the model, but is not connected then there is no *Relation* tag with the id of that class. However, if the class is connected, then the id will be present in a tag. This is important for determining which classes in the model conform to the metamodel (i.e., are connected with composites) and which classes are free-floating in the modeling editor, as free-floating classes are not considered conformance errors in accordance to the MOLEGA metamodel.

The code generator stores the ids of models in the active, non-floating classes to ensure that only these classes are represented in the code generated. Attributes contained in the model are sorted into one of three String variables: a *style* variable, an *app* variable, or a *constants* variable. The *style* variable contains all text meant to make up the style.css file for the code output, which mostly includes background colors of webpage elements and other colors that do not change. The *app* variable contains the global variables required for the server to run the game code, which consists of attributes such as the number of rooms, the player limits for each room, and the file names for the CSV card files. Finally, the *constants* variable contains the client side variables which are independently kept for each connecting client. These include the card color attributes and the font which is used for displaying information to the client. When all active classes have been parsed, the above String variables are then used to fill in values in template source code files in order to be added to a zip file for export. This is done using JSZip [4], a Javascript-specific library for generating and modifying zip files.

While the custom elements come directly from the input model, several other files required for the game to function correctly are fixed for any input model. The code generator only generates Community Judge-specific files for models that have the *CommunityJudgeGame* class. Similarly, Relations-specific files are only generated for models that contain the *RelationsGame* class. The collection of templated and fixed files is done using XMLHttpRequest objects to access the bodies of these files on the server. To avoid these asynchronous requests from ending after the export has occurred, these requests are nested on one another to ensure that all required files are included before a full export occurs.

6 Evaluation

To answer the second part of **RQ2**, regarding whether the code generated in the previous section is consistently **correct and executable**, we must formally evaluate the code generation process. To achieve this, we focused on the correctness of the generated code in comparison to the expected output. A systematic evaluation plan was chosen to test and verify both positive and negative error cases. This section first explains the design of the MOLEGA evaluation, followed by the results and a short discussion of the evaluation.

6.1 Experimental Design

The design of MOLEGA's evaluation is split into two major categories: valid and invalid models. These two categories could then further be split again by specifying the target type

[4] https://stuk.github.io/jszip/

```
activeClassIds = [], constants="", style="", app=""

for relation in all Relation elements:
  if ! activeClassIds includes relation.destination:
    activeClassIds.push(relation.destination)
  if ! activeClassIds includes relation.source:
    activeClassIds.push(relation.source)

for class in all Class elements:
  if activeClassIds includes class.id:
    if class.name includes "CommunityJudge":
      save "CommunityJudge" as gameType
    else if class.name includes "Relations":
      save "Relations" as gameType
    for attr in class.attributes:
      format each attr in "var " + attr.name + " = "
          ↪ + attr.value + ";" format
      place attribute in the correct String (
          ↪ constants, style, app)
if no Theme class exists:
  apply default theme to constants and style

if gameType == "Community Judge":
  generate Community Judge-specific static files
else:
  generate Relations-specific static files

generate remaining files (readme, package.json,
    ↪ constants, style)
export zip file
```

Figure 6. Code Generation Pseudocode

to be tested: Community Judge-type models and Relations-type models. This taxonomy exhaustively covers all possible model categorizations able to be developed using MOLEGA.

For valid models, three boolean criteria were checked: “Generate?", “Correct?", and “Does it Run?". The first success criterion relates to whether or not any code is generated from the model. For the second criterion, the result is considered correct if it generated the expected code. If code is generated but not correct, then this means the game is not customized properly in the way that the user intended.

In invalid models, only one criterion was tested: “Generate?". In these tests, the passing answer to this question should be “No". Since invalid models should not generate code, they have no need to be correct. Essentially, since these models are created with intentional errors, they should not allow code generation, and instead present the user with an error. Thus a result of “No” for this criterion indicates a successful evaluation. In each of these cases, detailed error messages explaining how to resolve the errors and enable successful code generation are provided to the user.

For each valid model, one element was tested at a time to ensure that the component change in the model accurately resulted in the corresponding target code. For example, the base model for a Community Judge game has a game object, a win condition of *winByRounds*, a *QuestionsDeck*, and an *AnswersDeck*. In this model, all default values are kept the same while adding valid input to the default missing attributes in order to make the model valid. This follows a simple mutation testing approach to easily identify the location of any errors introduced by a subsequent test evaluation.

To test that generated code is in fact correct, the expected output for each model transformation was created manually. While tedious, this ensures that the expected files align with what a human user would expect to encounter rather than relying on another machine-generated file that could be generating incorrect content. Any files that are intended to be customized were manually created to compare with the generated files. A bash script simplified this comparison task: unzipping the generated code and comparing all custom files in the generated code to the manually created expected files, ignoring formatting to avoid flagging changes with no functional impact. If a conflict arises, the script notes as such along with which files are not the same. No matter if all tests pass or fail for that model, after completing one generated zip file, the script then moves on to the next one, only requiring one execution of the script to check all valid model code generations. As long as the generated zip file and the expected output directory have the same name then the correct files are checked for discrepancies.

To ensure that valid models produced *executable* web game code, after passing the code generation and correctness tests, each generated game was launched in a browser. This final step evaluates the “Does it Run?" criterion. While a game file with code correctness should theoretically launch and play with no issues, physically launching each game gives extra assurance that each valid model creates a working target.

6.2 Results

Tables 1a through 1d display the results of all tests conducted. These tests were generated in order to exhaustively cover all possible cases, both valid and invalid. Table 1a displays the valid Community Judge models, and Table 1b shows the valid Relations models. Table 1c lists test cases addressing invalid Community Judge models, and Table 1d shows the invalid Relations models. With every single test case having passed, we produce a final result of **100% success** across all criteria. We have made the MOLEGA environment, along with our evaluation dataset, publicly available [5].

6.3 Discussion

All tests for valid models passed. These tests evaluated the code generator’s ability not only to export code in response to a valid model, but also that the code generated was correct and capable of execution. This supports an affirmative answer to both of this paper’s research questions. Since the purpose of these verification tests was to ensure that the code

[5] https://doi.org/10.5281/zenodo.5167719

Table 1. Evaluation Results for the 4 Experiments

(a) Valid Community Judge Model Results

Valid Community Judge			
Test Case	**Generate?**	**Correct?**	**Does it Run?**
Base Model			
Base Valid Model	✓	✓	✓
Class Changes			
Use Blue Theme	✓	✓	✓
Use Red Theme	✓	✓	✓
Use Green Theme	✓	✓	✓
Use Custom Theme (CSS, all valid)	✓	✓	✓
Use Custom Theme (hex, all valid)	✓	✓	✓
Use Win By Score	✓	✓	✓
Extra Theme (Not Connected)	✓	✓	✓
Extra Win Condition (Not Connected)	✓	✓	✓
Extra Questions Deck (Not Connected)	✓	✓	✓
Extra Answers Deck (Not Connected)	✓	✓	✓
Attribute Changes			
Change Prof Name	✓	✓	✓
Change Num of Rooms	✓	✓	✓
Change Max Players	✓	✓	✓
Change Min Players	✓	✓	✓
Change Num Starting Cards	✓	✓	✓
Change Deck Name (Questions)	✓	✓	✓
Change Deck Name (Answers)	✓	✓	✓
Change Color Theme	✓	✓	✓
Change Table Color	✓	✓	✓
Change Hand Card Color	✓	✓	✓
Change Font	✓	✓	✓
Change Question Card Color	✓	✓	✓
Change Hand Card Color	✓	✓	✓
Change Question Card Font Color	✓	✓	✓
Change Rounds to Winner	✓	✓	✓
Change Score to Winner	✓	✓	✓

(b) Valid Relations Model Results

Valid Relations			
Test Case	**Generate?**	**Correct?**	**Does it Run?**
Base Model			
Base Valid Model	✓	✓	✓
Class Changes			
Use Blue Theme	✓	✓	✓
Use Red Theme	✓	✓	✓
Use Green Theme	✓	✓	✓
Use Custom Theme (CSS, all valid)	✓	✓	✓
Use Custom Theme (hex, all valid)	✓	✓	✓
Use Win By Score	✓	✓	✓
Extra Theme (Not Connected)	✓	✓	✓
Extra Win Condition (Not Connected)	✓	✓	✓
Extra General Deck (Not Connected)	✓	✓	✓
Attribute Changes			
Change Prof Name	✓	✓	✓
Change Num of Rooms	✓	✓	✓
Change Max Players	✓	✓	✓
Change Min Players	✓	✓	✓
Change Num Starting Cards	✓	✓	✓
Change Use Discard Pile to False	✓	✓	✓
Change Deck Name (General)	✓	✓	✓
Change Color Theme	✓	✓	✓
Change Table Color	✓	✓	✓
Change Hand Card Color	✓	✓	✓
Change Font	✓	✓	✓
Change Question Card Color	✓	✓	✓
Change Hand Card Color	✓	✓	✓
Change Question Card Font Color	✓	✓	✓
Change Rounds to Winner	✓	✓	✓
Change Score to Winner	✓	✓	✓

(c) Invalid Community Judge Model Results

Invalid Community Judge			
Test Case	**Generate?**	**Correct?**	**Does it Run?**
Missing Classes			
Missing Game	✗	N/A	N/A
Missing Questions Deck	✗	N/A	N/A
Missing Answers Deck	✗	N/A	N/A
Missing Win Condition	✗	N/A	N/A
Missing Attribute Values			
Missing Prof Name	✗	N/A	N/A
Missing Questions Deck Name	✗	N/A	N/A
Missing Answers Deck Name	✗	N/A	N/A
Missing Values from Custom Theme	✗	N/A	N/A
Missing/Deleted Other Attributes	✗	N/A	N/A
Surplus Relations			
Surplus Themes	✗	N/A	N/A
Surplus Win Conditions	✗	N/A	N/A
Surplus Questions Decks	✗	N/A	N/A
Surplus Answers Decks	✗	N/A	N/A
Surplus Games	✗	N/A	N/A
Invalid Inputs			
Max Players Less Than Min Players	✗	N/A	N/A
Both Decks the Same File Name	✗	N/A	N/A
Attributes Zero or Negative	✗	N/A	N/A
Win Rounds Zero or Negative	✗	N/A	N/A
Win Score Zero or Negative	✗	N/A	N/A
Invalid CSS/Hex Color	✗	N/A	N/A
Invalid Font Type	✗	N/A	N/A
Invalid Composite Connection	✗	N/A	N/A

(d) Invalid Relations Model Results

Invalid Relations			
Test Case	**Generate?**	**Correct?**	**Does it Run?**
Missing Classes			
Missing Game	✗	N/A	N/A
Missing General Deck	✗	N/A	N/A
Missing Win Condition	✗	N/A	N/A
Missing Attribute Values			
Missing Prof Name	✗	N/A	N/A
Missing General Deck Name	✗	N/A	N/A
Missing Values from Custom Theme	✗	N/A	N/A
Missing/Deleted Other Attributes	✗	N/A	N/A
Surplus Relations			
Surplus Themes	✗	N/A	N/A
Surplus Win Conditions	✗	N/A	N/A
Surplus General Decks	✗	N/A	N/A
Surplus Games	✗	N/A	N/A
Invalid Inputs			
Max Players Less Than Min Players	✗	N/A	N/A
Relations Attributes Zero or Negative	✗	N/A	N/A
Win Rounds Zero or Negative	✗	N/A	N/A
Win Score Zero or Negative	✗	N/A	N/A
Invalid CSS/Hex Color	✗	N/A	N/A
Invalid Font Type	✗	N/A	N/A
Invalid Composite Connection	✗	N/A	N/A

generation process consistently produces valid and correct code, the fact that all tests passed demonstrates that not only can domain-specific modeling be used for educational card games (**RQ1**), but that our method is capable of consistent correctness in all cases. Moreover, since all generated code successfully ran, this code can also be claimed as executable.

In addition, all tests passed for invalid models as well. Since these tests were intended to end in a non-generation scenario, the presence of all "X" answers to the "Generate?" criterion qualifies as a pass. Since **RQ2** investigates the correctness of the generated code, the prevention of generating any code when the model is incomplete or incorrect preempts the possibility of generating incorrect code. If a model is incomplete or does not conform to its metamodel, then no transformation should occur to ensure that only consistently correct code is generated. This behavior is confirmed through our experiments and results in Tables 1c and 1d.

With respect to **RQ1**, through the evaluation experiments dedicated specifically to **RQ2**, we demonstrate that MOLEGA can be used to create web-based educational card games, confirming that domain-specific modeling is a useful tool for this application. MOLEGA is capable of representing all possible game variants, indicating that there are no aspects lost through the abstraction provided through modeling.

With respect to **RQ2**, the evaluation confirms that through the guided editor and strict metamodel conformance requirements, every valid model where a game was expected to be produced led to consistently correct and executable code. Furthermore, any models that contained invalid or missing game elements would not lead to broken or incomplete code, preventing users from attempting to use code that would not work. In summary, through this evaluation, it is shown that not only is it possible to use domain specific modeling to create educational games (**RQ1**), but the provision of our strongly guided modeling approach ensures that only correct and executable code will ever be provided to users (**RQ2**).

7 Conclusion

To conclude, we first present a look at some threats to the validity of our implementation and evaluation within the context of our approach. Following this, we will present some potential areas of future work to expand on the project's contributions. Finally, we summarize the contributions of our work made to the area of domain-specific modeling.

7.1 Threats to Validity

MOLEGA is limited to representing two specific rulesets for the chosen target games. Considering the sheer number of card game rulesets that exist, this is an initial limitation to this work. Currently, MOLEGA's metamodel is simplified to accommodate only the selected games, however, due to the abstract nature and the modularity of the class-style models, it is possible to extend to other card games, providing more variation points. While adding a new multiplayer game type or a new kind of win condition would be rather simple, updating the code generator to include this new variant would require significant refactoring, since each new ruleset would need to be created from scratch. Essentially, the bottle neck for this would be in the code generation process, rather than the modeling language definition.

While our evaluation validates the functionality of the web-based framework and code generation processes, it does not involve any user studies for testing the usability, usefulness, or enjoyment of either the web editor or the target games generated. Without this level of detailed evaluation, we are unable to make any claims about the usefulness of our approach, only the validity of the code generation process, thus making our work more of a proof-of-concept rather than a fully robust implementation.

Evaluation experiments for this project were conducted by comparing the generated code to manually created expected outputs. As with any work with manual process, the possibility of human error is introduced. However, without a benchmark to compare against, this was the best option to demonstrate validity. There does not currently exist an automated way of creating expected code outputs for a system model, as that is the focus of our work, to reduce the risk. One way this threat was reduced is by reusing as many previously validated outputs wherever possible.

7.2 Future Work

We plan to conduct user studies using the MOLEGA framework to further validate its usefulness as a tool for use by educators, as well as evaluating the resulting games. By designing two further evaluation experiments, we plan to further demonstrate the benefits of the MOLEGA framework. First, we plan to evaluate MOLEGA's use by educators in various disciplines to ensure non-technical users are able to effectively produce games using the modeling environment. As part of this experiment we will collect both quantitative and qualitative data in the form of user surveys. The second expanded experiment will see the use of generated code by students in several classes to play generated games and provide feedback about their experiences. This evaluation focuses more on the resulting games, but is an important factor further supporting **RQ1**. This expanded evaluation would move beyond asking "Does it Run?" to a more robust evaluation of "Is the Game Playable?".

Another area of future work relates to expanding the target games supported. Currently, the two games provided were chosen due to their popularity and educational support, but further games would only expand the contributions of our framework. Refactoring the current MOLEGA metamodel to enable both multiplayer and single-player games, or to allow the combination of game rulesets would increase the customizable aspects of this algorithm, allowing more control over the output to the user creating a model.

7.3 Summary

In specific domains, such as educational game design, models are used to help non-technical experts represent domain knowledge. While not directly involved in game design, educators can benefit from the accessibility of games for use in their classrooms. However, educators often do not have the technical skills to create these games themselves. Through DSMLs, these potential users can leverage MDSE to obtain functional games, needing to only specify the custom aspects of their game via a graphical editor.

MOLEGA's framework includes a graphical editor, possessing the ability to represent two types of card game while providing various customizable features to the games. Most importantly, the content on the cards in the game is customizable, allowing games generated by MOLEGA's code generator to be used not only as generic card games, but also as educational tools by replacing the default content with specific educational content. This generated code can be hosted with minimal technical support and can be accessed by any device with browser capabilities.

The code generation process has been evaluated using a systematic approach for both valid and invalid models. For models that are valid, code is shown not only to be successfully generated, but also correct when compared to a manually created expected output, and capable of running on a server. For models that should not generate code, they fail to run the code generator and give the user an error message as intended. These tests ensure that not only MOLEGA can be used to represent and generate web-based educational card games, but also that any code generated by the algorithm is consistently correct and executable in a server environment.

While it requires minimal technical knowledge or support to host the generated game code, MOLEGA provides an approach to custom web code generation that requires little technical knowledge to generate a result. Our work shows that a DSML can be used to represent and create web-based educational games. It also demonstrates that given our guided framework, this code generated is found to be valid, correct, and executable in all cases.

References

[1] Sean M. Barclay, Meghan N. Jeffres, and Ragini Bhakta. 2011. Educational Card Games to Teach Pharmacotherapeutics in an Advanced Pharmacy Practice Experience. *American Journal of Pharmaceutical Education* 75, 2 (March 2011), 7 pages. https://doi.org/10.5688/ajpe75233

[2] Francesco Bellotti, Michela Ott, Sylvester Arnab, Riccardo Berta, Sara de Freitas, Kristian Kiili, and Alessandro De Gloria. 2011. Designing serious games for education: from pedagogical principles to game mechanisms. In *European Conference on Games Based Learning*. Academic Conferences Ltd, Athens, Greece, 26–34.

[3] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, Williston, VT, USA. https://doi.org/10.2200/S00751ED2V01Y201701SWE004

[4] Daniel D Burkey and Michael F Young. 2017. Work-in-Progress: A 'Cards Against Humanity'-style card game for increasing engineering students' awareness of ethical issues in the profession. In *Annual Conference & Exposition*. ASEE, Columbus, Ohio, USA, 11 pages. https://doi.org/10.18260/1-2--29190

[5] H. Cho, J. Gray, and E. Syriani. 2012. Creating visual Domain-Specific Modeling Languages from end-user demonstration. In *2012 4th International Workshop on Modeling in Software Engineering (MISE)*. IEEE, Zurich, Switzerland, 22–28. https://doi.org/10.1109/MISE.2012.6226010

[6] Nicholas John DiGennaro. 2021. *Intuitive Model Transformations: A Guided Framework for Structural Modeling*. Master's thesis. Miami University. http://rave.ohiolink.edu/etdc/view?acc_num=miami1618913067752324

[7] Teo Eterovic, Enio Kaljic, Dzenana Donko, Adnan Salihbegovic, and Samir Ribic. 2015. An Internet of Things visual domain specific modeling language based on UML. In *2015 XXV International Conference on Information, Communication and Automation Technologies (ICAT)*. IEEE, Sarajevo, Bosnia and Herzegovina, 1–5. https://doi.org/10.1109/ICAT.2015.7340537

[8] André WB Furtado and André LM Santos. 2006. Using domain-specific modeling towards computer games development industrialization. In *The 6th OOPSLA workshop on domain-specific modeling (DSM06)*. Citeseer, ACM, Portland, Oregon, USA, 1–14.

[9] Cynthia M. Odenweller, Christopher T. Hsu, and Stephen E. DiCarlo. 1998. Educational card games for understanding gastrointestinal physiology. *Advances in Physiology Education* 20, 1 (December 1998), 7 pages. https://doi.org/10.1152/advances.1998.275.6.S78

[10] Akhila Tirumalai Prasanna. 2012. *A Domain Specific Modeling Language for Specifying Educational Games*. Master's thesis. Vrije Universiteit Brussel.

[11] Eric J. Rapos and Matthew Stephan. 2019. IML: Towards an Instructional Modeling Language. In *MODELSWARD*. SciTePress, Prague, Czech Republic, 417–425. https://doi.org/10.5220/0007485204190427

[12] Niroshan Thillainathan, Holger Hoffmann, Eike M. Hirdes, and Jan Marco Leimeister. 2013. Enabling Educators to Design Serious Games – A Serious Game Logic and Structure Modeling Language. In *Scaling up Learning for Sustained Impact*, Davinia Hernández-Leo, Tobias Ley, Ralf Klamma, and Andreas Harrer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 643–644. https://doi.org/10.1007/978-3-642-40814-4_92

[13] Niroshan Thillainathan and Jan Marco Leimeister. 2014. Serious Game Development for Educators - A Serious Game Logic and Structure Modeling Language. In *6th International Conference on Education and New Learning Technologies, Barcelona*. IATED Academy, Barcelona, Spain, 1196–1206. https://www.alexandria.unisg.ch/233433/

[14] Ana Syafiqah Zahari, Lukman Ab Rahim, Nur Aisyah Nurhadi, and Mubeen Aslam. 2020. A Domain-Specific Modelling Language for Adventure Educational Games and Flow Theory. *International Journal on Advanced Science, Engineering and Information Technology* 10, 3 (2020), 999–1007. https://doi.org/10.18517/ijaseit.10.3.10173