Towards an Interoperable Model-driven Automated Assessment System for Computer Science Education

Markus Hamann
markus.hamann1@tu-dresden.de
Chair of Software Technology
Technische Universität Dresden
Dresden, Germany

Sebastian Götz
sebastian.goetz1@tu-dresden.de
Chair of Software Technology
Technische Universität Dresden
Dresden, Germany

Uwe Aßmann
uwe.assmann@tu-dresden.de
Chair of Software Technology
Technische Universität Dresden
Dresden, Germany

Abstract

In recent years, the number of computer science students has steadily risen, but the time for educators to give feedback to the students remains the same. Because of this predicament, many systems for the automatic or semi-automatic assessment of student tasks were developed. A fundamental problem of this development is that most of these assessment systems deploy a different type of data representation, which leads to a lack of interoperability between the approaches. This hinders the reuse of teaching materials that need to match the targeted system, leading to situations in which instructors need to recreate their materials. In this work, we aim to close this gap by introducing a model-driven approach called *Assisted Assessment*. The approach uses a technology-independent assessment model to bridge instructors' and assessment systems' technical spaces, helping instructors to transform their material for various systems. We introduce *Assisted Assessment* by describing the scenario of an undergraduate course for software technology and how the approach can help to manage the different available assessment systems.

CCS Concepts

• **Applied computing** → **Interactive learning environments**; • **Software and its engineering** → *Design languages*; **Interoperability**; *Model-driven software engineering*.

Keywords

model-driven engineering, education, student assessment

ACM Reference Format:
Markus Hamann, Sebastian Götz, and Uwe Aßmann. 2024. Towards an Interoperable Model-driven Automated Assessment System for Computer Science Education. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3652620.3687775

MODELS Companion '24, September 22–27, 2024, Linz, Austria

ACM ISBN 979-8-4007-0622-6/24/09
https://doi.org/10.1145/3652620.3687775

1 Introduction

During the last years, the number of computer science students is steadily increasing. At the same time, each instructor's time for giving feedback to their students remains the same. Moreover, in some regions, the number of students willing to work as student assistants in education also decreases due to the increasing competition with student jobs in the industry. As a result, many different kinds of automatic or semi-automatic systems to support teachers in their interactions with their students have been presented in recent years [2, 3, 6, 7, 13]. These systems focus on automated assessment of exercises, expressive feedback to the students, teaching support, or a combination of the former.

A problem arising from this diversity is the lack of interoperability between the approaches. Each system employs a different type of data representation (i.e., meta-model). The data sets are often created manually for each system and for each task at hand. Reuse of teaching data sets across different approaches is often either impossible or implies extensive manual efforts. The process of creating new or translating existing teaching material is time-consuming and allows errors and inconsistencies.

To overcome this interoperability problem, we propose a model-driven approach using an intermediate assessment model, called *Assisted Assessment*. The intermediate assessment model serves as a bridge between the technical space of assessment systems and the problem space of instructors using these systems. By separating these spaces and bridging them with a technology-independent model, each space can be varied independently without interfering with the others. In this work, we present the following vision:

- Defining an assessment model holding assessment-critical information in a technology-independent format
- Using a meta-assessment model to minimize the effort of creating the assessment model
- Describing a workflow how to use the assessment model in the given approach

We exemplify our approach with a demonstration for the automated assessment of modeling tasks to be solved by students in the field of domain models. In this case study the intermediate assessment model is connected with multiple assessment systems. The assessment model connects to a rule-based assessment engine, is used to generate support artifacts for manual or machine-learning-based assessment, and to generate documentation about the utilized assessment and grading schemata.

2 State of the Art

The development of assessment systems for modeling and programming tasks in education has been an active field of research for many years. In 2023, a literature review from Ullrich et al. [13] analyzed publications from 1998 to 2021. They found a total of 110 publications on automatic assessment of conceptual models and identified several techniques for their technical realization.

The most common technique are *model comparison approaches* (78), which compare an expert or ideal model to the student model. A recent example for this class is the approach from Jebli et al. [6], which directly compares the student with the tutor model.

Under normal conditions, migration between systems that follow the model comparison approach consumes few resources since the instructor only needs to recreate the expert model in the new system's model format. It is often assumed that the migration is trivial if the model formats are standard formats like UML diagrams, but that might not always be the case. For example, problems could occur if a highly specialized, non-standardized model, such as domain models, or certain UML extensions, such as profiles, are used in the task. While the migration process may not be overly resource-intensive, switching systems necessitates to recreate all expert models of a task collection. Then, instructors cannot directly reuse their teaching materials from one system to another.

The disadvantage of the pure model comparison approach is that only direct comparisons are possible, which are often not expressive enough to represent enough of a task's solution space. Therefore, model comparison systems often utilize rule-based techniques, too. Ullrich et al. [13] also mentioned an overlap between model comparison and rule-based approaches. Examples are the approach of Bian et al. [2], which compares models with the help of multiple rule-based algorithms, and Thomas et al. [12], which assess the submitted models based on similarity rules.

Rule-based approaches were described in 41 of the publications in [13]. They use rules of different complexity to assess student models. A newer example, Boubekeur et al. [3], uses several simple heuristics, like the number of classes and their relationships. Striewe et al. [11] use graph queries to perform more complex checks on a model instead. Alternatively, the rules themselves can be included inside an implementation [3] or exist as separated artifacts [11]. Depending on the circumstances, these approaches can require an extensive number of rules depending on the details and alternatives that need to be checked. Most systems found in the literature use their distinct rule format, generally with different limitations and notations. Notably, also for rule-based systems there is no easy way to migrate the teaching materials from one approach to another.

In recent years, there have also been a number of approaches to include *machine learning* in the field of automatic assessment. One example would be the modeling assessment system from the project Artemis [7], which used past evaluations to present suggestions for new solutions. Another example is again Boubekeur et al. [3], who include machine learning to improve the performance of their heuristic approach. Machine learning results often improve by having consistently labeled and structured input parameters. Since each approach comes with its own set of input parameters, instructors would need to prepare the learning data for each system anew. Since creating consistent parameter sets can be time-consuming, this can hinder the usage or migration between different systems.

Such approaches exist in the field of assessing programming exercises, too. Xiang et al. [14] presented an approach to directly match a student program with the program of a tutor. Calderon et al. [4] presented an approach that uses rules for expected outputs to assess programming assessments. Lastly, Artemis [8] and INLOOP [9] are examples of programming assessment systems using tests to assess the quality of a student's programming skills. Since these approaches are similar to their modeling counterparts, their problems are similar, too. In the case of comparing two programs, an instructor can migrate to different systems without recreating the programs since the programming language stays the same. Targeting different languages is preferable when assessing an algorithm with this approach. This leads to the same problem as in the modeling space since instructors must recreate their programs in the new language and cannot directly reuse the existing materials. Also, the rule-based programming assessment approaches are similar to their model-space counterparts since the rule language would depend on the system used. When employing tests for comparison (*test-based approach*), instructors often face the same advantages and disadvantages as when using the comparing approach. When instructors use the same implementation language, they can easily migrate the examples and tasks to or use another system without any problems, as described above. On the other hand, instructors must recreate their teaching material for each system when providing the same task for different implementation languages.

All these systems, which have been developed for several years, use their own meta-model. This hinders the reuse of teaching materials since materials, programs, as well as models, need to match the meta-model of the targeted system, so that instructors need to recreate their materials. An approach does not yet exist that tries to unify the different representations, for instance, by means of metamodel mappings [10], or at least by the provision of support tools to simplify the migration between the system. Even worse, if models should be migrated between distinct technical spaces with different meta-languages [1], no support exists. In this work, we aim to close this gap by introducing a novel model-driven, generic approach that enables the interoperability between existing systems, while the instructor only specifies the teaching material once.

3 Assisted Assessment

3.1 Running Example: Domain Model

Teaching how to create domain models from a textual description is part of many computer science curricula. To support the educators, a model assessment system providing automated feedback to student solutions is used. Somewhen, the educators plan to switch to two other assessment system. Until then, the old assessment system used ideal or expert models, as shown in Figure 1. The new assessment systems use a rule-based language as shown in Listing 1 and an instructor hint system shown in Listing 2 to check all relevant model elements in a student's solution. Without our approach, the educators face considerable efforts to manually translate their teaching materials from the old to the new assessment system.

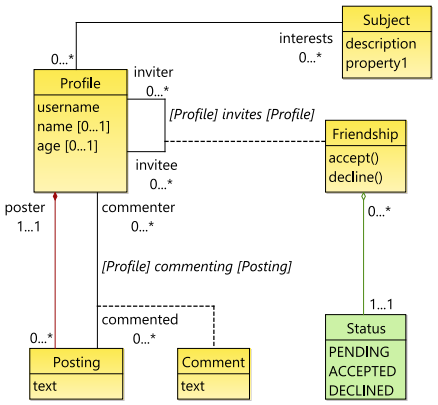

Figure 1: Visual representation of an expert or ideal model that is used as a task-specific input material for the *Assisted Assessment* workflow. The model represents a domain model of a simplified domain of a *social network*.

3.2 A Model-driven Pipeline for Assisted Assessment

This work introduces a three-staged approach using model-driven concepts to assist instructors in generating assessment artifacts for a wide variety of assessment systems, called *Assisted Assessment*. Figure 2 depicts the workflow of the Assisted Assessment approach, including the three technical spaces: one for the instructor, one for each assessment system and one for the intermediate assessment model. The approach separates the instructor's technical space from the assessment system's technical space by isolating the assessment system's input representation from the task-specific information in the instructor's teaching material. For this purpose, the approach introduces a new third, *bridging technical space* that centers around the assessment model.

The assessment model (cf. Figure 2, result of stage 2), which holds all relevant information for a specific assessment, plays the central role in Assisted Assessment, since it serves as a bridge between the instructor's and assessment system's technical spaces. Assessment models are task-specific but independent of the technology of the targeted assessment system. The assessment models are generated in a task-specific second stage by the composition of the available assessment rules. Assessment rules are atomic parts of the assessment model that hold one assessment instruction each. They are described in detail in the next section. This process of creating assessment rules can be done manually for each task by the instructor (cf. Figure 2, see *Manual Creation*) or with the help of the first stage (cf. Figure 2, output of *Meta-Assessment Evaluator*). In the given example domain, the assessment model could store the assessment instructions needed for the assessment of the domain models submitted by students.

While the assessment model is task-specific, it also contains data common for many tasks. Therefore, a first stage of the approach was introduced which prepares, for all task-specific assessment in a specific field, a common set of rules, experiences, and other stable data. This common data can significantly reduce the effort required to create the assessment model for each task and helps to maintain it

Listing 1: Simplified Epsilon Validation Language rule [5] for the automatic assessment of a student's domain model.

```
...
context Property{
  constraint R_01_profile_age{
    check{
      var r1 = self.class <> null;
      var r2 = self.name.compareLabel("age");
      var r3 = self.class.name.compareLabel("Profile");
      var result = r1 and r2 and r3;
      return not result;
    }
    message{
      output( 1, "The concept of 'age' was found and is part of
                  the correct containing concept 'Profile'.");

      return 'correct : R_01_profile_age';
    }
  }
}
...
```

Listing 2: Grading instruction in the format required for modeling tasks in the assessment system Artemis [8]. Multiple instructions inside one criterion are possible, allowing instructions of several rule types for the target *Property 'age'*.

```
[criterion] Property 'age'
  [instruction]
  [credits] 1
  [gradingScale] correct
  [description] The property 'age' must be part of a class
                named 'Profile'.
  [feedback] The concept of  'age' was found and is part of
             the correct containing concept 'Profile'.
  [maxCountInScore] 1
```

if changes occur in the common data. As input to stage 1, we propose a meta-assessment model that includes the data shared across all tasks and uses it as a template in a model-driven pipeline. Combined with task-specific information, from the meta-assessment model most parts of the task-specific assessment model can be generated, which is beneficial, when the instructor needs to check for common task-unspecific errors or common convention mistakes. Though there are other ways an instructor could create rules with similar results, the meta-assessment model approach tends to create, in our experience, more uniform and readable assessment models. Defining meta-rules fosters reuse and enables the instructor to establish a base rule library for a specific task type. In the given example domain, existing expert domain models (cf. Figure 1) could be used as task-specific input teaching material to generate most of the assessment model.

The third stage produces the technology-dependent representation of the assessment model (cf. Figure 2, result of stage 3). This representation can vary, depending on the employed meta-models and meta-languages of the assessment system, and multiple representations are possible. If a rule-based assessment system is used as target system, the representation would be a rule-based language. For documentation, an overview with examples can be generated. In all these cases, a template engine can be utilized to generate the technology-dependent representation. An advantage of this approach is that, since the assessment model is stable, switching or expanding to a new assessment system for another target language or technical space can be done by supplying a new template. In the

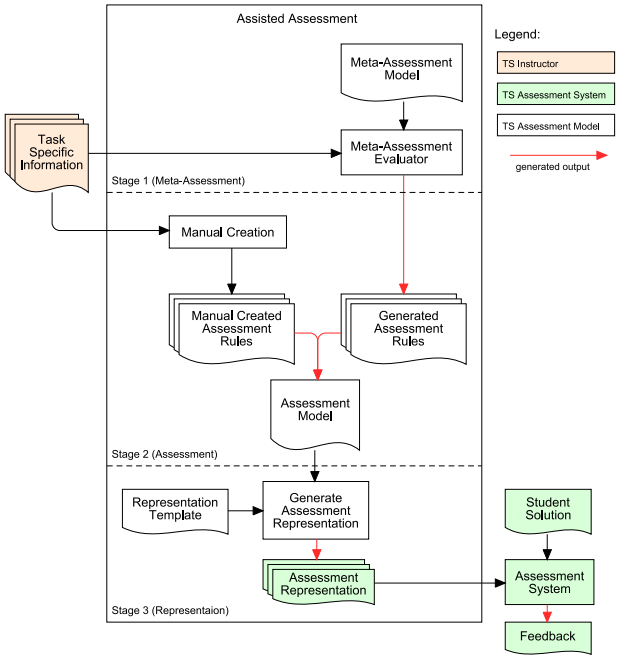

Figure 2: Description of the three-staged workflow of the *Assisted Assessment* approach. The technical spaces (TS) of the assessment system and the instructor lie primarily outside of the system boundaries and are bridged by the central *Assessment Model*.

given example domain, both technology-dependent representations of the assessment model candidates (cf. Listings 1 and 2) could be created from the same assessment model.

The proposed pipeline has multiple advantages. First, the assessment model and the generation of the assessment representation separate the instructor's technical space from the assessment system's technical space. This reduces constraints and limitations transferred from one space to the other. The stable bridging space (i.e., the assessment model) also allows for changing the technical space of the assessment system while keeping the instructors stable. Better reuse of existing teaching materials and the possibility of building up shared libraries of teaching materials, including assessment models, are further advantages of this approach.

Naturally, there are disadvantages that come with every model-driven technology stack, like the higher complexity of the approach and especially the steeper learning curve for beginners. Also, minor changes in the assessment model may be needed when changing to a completely different assessment type. However, we believe that increased flexibility, reuse, and effort reduction in a running pipeline outweigh the drawbacks significantly.

3.3 The Assessment Model

The Assisted Assessment approach aims at minimizing the work needed to create, maintain, and reuse assessment artifacts. For this purpose, a central model is defined, the assessment model that includes all the information needed to maintain the assessment

Listing 3: Example assessment rule of the assessment model in textual form. The domain of this rule is a domain model. The rule checks for a correctly modeled element in the student solution. In this case it searches for a property named *'age'* in a class named *'Profile'*.

```
1  Rule R_01_profile_age{
2    type: correct  scope: single  check: presence
3    query{
4      target: age:Property
5      query{
6        c:Class -:contains-> p:Property;
7        check{ (p.name == "age") and (c.name == "Profile")
8      }
9    }
10   score: 1
11   feedback: "The concept of 'age' was found and is part of
12              the correct containing concept 'Profile'."
13 }[
14     "The property 'age' must be part of a class named 'Profile'."
15     author: "Markus Hamann"
16     diagram{ @startuml
17              class Profile{
18                age
19              }
20              @enduml}
21 ]
```

across multiple artifacts and assessment types. The assessment model includes multiple assessment rules, and the entirety of all rules form the assessment, when they are applied to a model. To realize the assessment model, we investigated existing assessment systems and their input formats, to design the assessment rules to match most assessment systems. As first step, we have focused on model assessment systems, but the approach is likely to be transferable to programming assessment systems.

We found that assessments usually comprise four categories of data. For better understanding, Listing 3 shows an example of an assessment rule in form of a textual domain-specific language. This assessment rule can be seen as the technology-independent version of the assessment rule shown in the example domain (cf. Listings 1).

The first category describes the rules' 'general' attributes. The first attribute is the name or identifier of the rule. In the example, the rule is called *'R_01_profile_age'* (Line *1*), a hint that this rule is the first rule for a concept *'profile_age'*. In Line *2* of the example, there is also information about the *type* of rule, its *scope*, and if the *presence* or *absence* of the rule query should be *check*ed. To understand these three attributes better, the second category needs to be explained first.

The second category of a assessment rule in the assessment model is the 'query' (Lines *3* to *8*). A query is comprised of a target and a constraint to be checked. The *target* describes the element the rule should check. That can be an element of an existing expert solution, a concept, or other goals. The target splits into an identifier and a target type. The example rule describes the target as *'age:Property'*. That information tells the instructor that the rule does a check on the concept *'Property'* and for the specific concept element named *'age'*. The check itself is done on the solution a student submits. The *query* then describes how the rule should try to match the elements of the submitted solution. Generally, every language that can define a constraint, test, or other types of assessment can be used inside the query. Such languages include programming languages, test frameworks, existing rule languages, graph query languages, or

other executable languages. Since the approach's primary goal is to separate the technical space of the assessment system from the instructor space, a technology-independent language, like graph query languages, should be used to target different assessment systems. In our experience, a graph query language is the best choice to define a query since it is the most versatile. They can be used for model comparison by simply checking for the existence of expert model elements in the student solution (similar to our example rule), but they can also be complex enough to describe more complex rules of rule-based languages. Tests for programming exercises can also be described by using graph query languages. The query in our example (cf. Listing 3, Lines *6* to *7*) uses a simplified pseudo query language to check whether the domain model submitted by the student as solution contains a property named *'age'* within a class *'Profile'*. In rule-based assessment systems [11], a graph query language is often used to define rules or constraints, but in these cases, the language is always part of the technical space of the assessment system. With the query language's inclusion in the assessment model rule, the language is now part of the new bridging technical space described in Figure 2.

The first category's last attributes (Line *2*) can now be described in detail with information on how the rules query works. First, the instructor can define custom *type*s, which specify the class of problems the rule checks. In our example, the instructor defined the type as *'correct'*. This indicates that the query checks if an element in the solution submitted by a student is defined in a way the instructor defines as correct. The instructor can freely define the types of a rule. However, in our experience some types commonly used are *correct*, *alternative*, *incomplete*, *error*, *missing*, and *info*. If the sequence of rule execution is essential, these standard types can be used to plan the rules' execution time. Usually, this is done by executing the rules first, which checks for a more complete solution. Next, the instructor can define the *scope* as either *single*, *global*, or *group*. In single-scope rules, the query stops on the submitted solution's first match and returns the rules feedback. A match is found if the rules query returns a successful result for a submitted solution's element. This scope type also specifies that after a successful match, all other single-scoped rules with the same target are stopped from being executed. In the case of the given example, it means that after the shown rule matches an element in the submitted solution, all other single rules with the same target *'age:Property'* will not be executed. An example of such a rule would be a single-scope rule checking for an alternative (type: *alternative*) solution for the same target. This prevents successfully matching elements in the submitted solution multiple times with multiple rules, searching for different variations of the same target concept. The scope type also stops the execution of rules on the previously matched element in the submitted solution. After a match, the rule of the example would stop other single-scope rules from trying to match against the *'age'* element in the submitted solution, for example, other similar rules searching for the same target type of *'Property'*. This prevents multiple independent rules from matching against the same element in the target solution. Global-scoped rules apply to the whole submitted solution. They do not stop the execution of other rules in any case and can not be blocked by other rules. Examples of this rule type are *info* rules checking for mistakes in conventions or bad practices and sending feedback on any match in the submitted solution. Finally, group-scoped rules are special rules that check if a group of other rules is satisfied. A rule that sends feedback on a *missing* element in the submitted solution in case all other rules with the same target fail to match is a prime example of a group-scope rule. The query of a group-scoped rule only includes a list of the rule identifiers; if one of these rules is matched, the group-scoped rule is matched, too. In most circumstances, these three types should cover the typical rule applications in assessments.

Outside of this attributes, two optional features are supported. Blocking attributes could be used to fine-tune rules if a rule needs to operate outside the definition of the three scope types. The attribute *block* can be used to define if the rule stops the execution of another rule on a successful match. The options for this attribute are:

- **none** The rule does not stop the execution of other rules.
- **target** The rule does stop the execution of rules with the same target.
- **element** The rule does stop the execution of rules trying to match the element the rule matched against.
- **both** The rule behaves like the target, and the element option is activated.

The other attribute called *blocked* can define if other rules can stop the rules execution. A rule's execution can be stopped by no other rule (*none*), rules that stop by target (*target* or *both*), or rules that stop by element (*element* or *both*). For example, single-scoped rules, by default, are blocking *both* and are blocked by *both*. Global-scoped rules are blocking *none* and are blocked by *none*. If these two scope types do not fit a rule, instructors can use the *block* and *blocked* attributes to override the scope behavior. The *check* attribute, the last one, plays a crucial role in determining rule matches. It defines whether the rule should be considered matched if the query successfully returns on an element of the submitted solution (*presence*) or if the rule is matched if the query has not returned successfully even once in the entire solution (*absence*). The example (cf. Listing 3) gives an example of a *presence* rule, which should match with an element. An example of an *absence* rule would be a rule that searches for a design pattern or best practice in the submitted solution and returns feedback if the pattern is not found.

The third category includes 'feedback', such as the *score* rewarded for matching the rule and a *feedback* message. The 'documentation' category, the last one for each rule, provides additional information about the rule. A *description* or other documentation data could be included here as key-value pairs. Examples are *version* information, *authors*, or other details.

Which data available in the rules or assessment model is used, and how it is used depends highly on the provided functionality of the targeted assessment system. So, while some systems can not use all the data provided by the model, the goal of the assessment model is to provide enough information to be used with most of the common assessment systems and assessment types. Since the transformation to the input representation of the targeted system (cf. Figure 2) is highly dependent on the targeted system features, the transformation needs to be designed and implemented manually once for each targeted system.

Listing 4: Example rule of the meta-assessment model that could be used to create the rule from Listing 3. The template slots could be filled with data extracted from a solution model prepared by the instructor.

```
Meta-Rule R_01{
  type: correct  scope: single  check: presence
  query{
    target: Property
    query{
      c:Class -:contains-> p:Property;
      check{ (p.name == "<target.name>")
             and (c.name == "<target.container().name>")}
    }
  score: 1
  feedback: "The concept of '<target.name>' was found and
             is part of the correct containing concept
             '<target.container().name>'."
}[
    "The property '<target.name>' must be part
     of a class named '<target.container().name>'."
    author: "Markus Hamann"
    diagram{ @startuml
             class <target.container().name>{
               <target.name>
             }
             @enduml }
]
```

3.4 The Meta-Assessment Model

The meta-assessment model also includes all discussed data but with some modifications to allow task-specific data to be inserted into its meta-rules. To better showcase the usage of a meta-assessment rule, Listing 4 showcases a meta-rule with simple template slots. Together with the expert domain model seen in the example domain (cf. Figure 1), it could be used to generate the task-specific assessment rule of Listing 3.

The *target* plays a pivotal role in the meta-assessment model, serving as an anchor for the task-specific data. This key feature allows for the effective integration of task-specific data into the model. The meta-assessment model only includes the *target type*, while the target element is left empty (cf. Listing 4, Line *4*). During the generation process, a task-specific rule is systematically generated for every task-specific data element corresponding to the target type. It also forms the target of the assessment model rules from the task-specific data element's identifier and the previous target type (cf. Listing 3, Line *4*).

The second difference of the meta-assessment model is the existence of template slots in the *query*, *feedback* message, *score*, and *documentation* fields (cf. Listing 4). The template slots follow the syntax of `<target.x>` where `x` is an expression that can be an attribute, a (self-defined) function, or a chain of both. The starting point for each expression inside a template slot is the rule's *target*, which is computed as discussed in the previous paragraph. Then, the element data will be inserted in the rule through the template slots by computing the return value of the expression `x`. A template engine can be used to do this step. In the example meta-assessment rule, the expression `target.name` (Line *11*) is computed to the property name *'age* and the expression `target.container().name` (Line *13*) is computed to the name of the containing element *'Profile'*. Naturally, the expression `x` needs to follow the task-specific input material's data- and function structure. Also, more powerful constructs like loops or if-clauses are possible depending on the template engine. Large amounts of task-specific assessment rules can so be created from a relatively low amount of meta-assessment rules by this simple mechanism.

The task-specific data must be provided in a format that can be used in the template engine. This requirement typically necessitates the data to be in a structural form, which can include a wide range of formats such as XML, XMI, JSON, or most programming or model language data formats. Prime examples of such a format are model files in the XMI format of the Eclipse Modeling Framework (EMF). Figure 1 shows the visual representation of a domain model created with the help of EMF and their meta-language Ecore. An instructor can utilize such a model as input for the template engine (cf. Figure 2, process 'Meta-Assessment Evaluator'). This process will generate the rule of Listing 3 and other similar rules (*username*, *name*, *text* for *Posting* and *Comment*, and *description*) from the meta-rule from Listing 4.

Naturally, since the template engine needs to transform material from the instructor's technical space, the format of task-specific material can vary greatly. Template engine implementations or configurations would be prepared once for common storage formats like XML, XMI, JSON, or EMF models. At the moment, a prototype for EMF models is already being successfully tested. For specialized data formats, an individual implementation of the template engine or a transformation program to EMF or XML would be needed.

4 Assessing Domain Models with Multiple Assessment Systems

The next section explains an example currently under evaluation to show the possibilities of the proposed approach. It follows the example domain introduced in Section 3.1. In an undergraduate course for software technology, a large part of the course is analyzing and designing domain models and other analysis models. These models follow a subset of the UML meta-model. For example, technical details and model elements introducing them, such as interfaces, generic classes, capabilities, and datatypes, are removed from the meta-model. To help the students prepare for the exams, multiple older exam tasks for domain models were revisited, and are prepared as exercise and self-studying material. An infrastructure in the course already exists for the automatic assessment of programming exercises. That infrastructure uses an assessment system called Artemis [8]. Artemis also allows for the assessment of modeling tasks, but only in a non- to semi-automatic way [7]. The instructor can prepare grading instructions for the evaluators that grade the assessments. These instructions can then quickly assess an element in the student solution by dragging it onto the element. The definition of an automatic assessment pipeline in Artemis is flexible since it employs shell scripts. Therefore, we can exploit this to easily integrate command-line applications. Thus, an automatic assessment system for modeling tasks can be integrated.

To support the students and showcase the possibilities of the approach, we discuss three possible use cases for the assessment of the domain models:

(1) **Automatic Assessment**
To enable students to always access a feedback system, like they are used to in the programming exercises, an automatic feedback system is used via the exercise pipeline of Artemis.

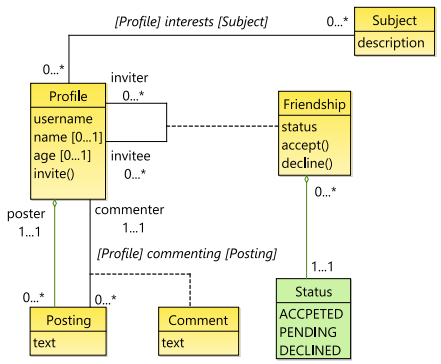

Figure 3: Visual representation of a student solution trying to recreate the domain model of a simplified domain of a *social network* (cf. Figure 1).

(2) **Non-Automatic Assessment**
Some people see automatic assessment skeptically, e.g., for hands-on exercises or situations like exams and graded homework. For this, Artemis's non-automatic modeling task pipeline can be used.

(3) **Documentation**
For the assessment of paper exams, student questions in meetings, and similar situations where information about an assessment is needed, documentation should be provided to ensure a common grading and assessment schema.

A rule-based validation engine for models is a common choice for automatic assessment. The requirement is that it can be loaded and called from the Artemis exercise command-line pipeline. We chose EVL (Epsilon Validation Language), a scripting language for model validation from the Epsilon language family [5], because of its simplicity and low overhead. Listing 1 shows the generated validation code from the rule shown in Listing 3. The representation also includes a generated main script to run all validations and generate an output in JSON format. The main script is then called by the pipeline if a student solution is available. The assessment pipeline then returns the results as feedback and a score to the student via the Artemis user interface.

To showcase this process, we present a hypothetical scenario. An instructor utilizes a meta-assessment model, including rules like Listing 4 and the expert solution shown in Figure 1, to create an assessment model for the task 'social networks'. Listing 3 shows a rule checking for the existence of the property *'age'* in a student solution. After generating an EVL script program with validation rules like Listing 1 out of the assessment model and including it in the automatic assessment pipeline, the instructor opens the task for students. Figure 3 shows a student solution trying to recreate the expert solution of Figure 1. After uploading it to Artemis, the automatic assessment pipeline using the generated EVL program assesses the solution and returns the feedback seen in Figure 4. In the excerpt of the feedback seen in the figure, it can be seen that the assessment models contain rules for checking the student solution for *correct*, *alternative*, *incomplete*, and *missing* elements. It can also be seen that the student was mostly right about their domain model

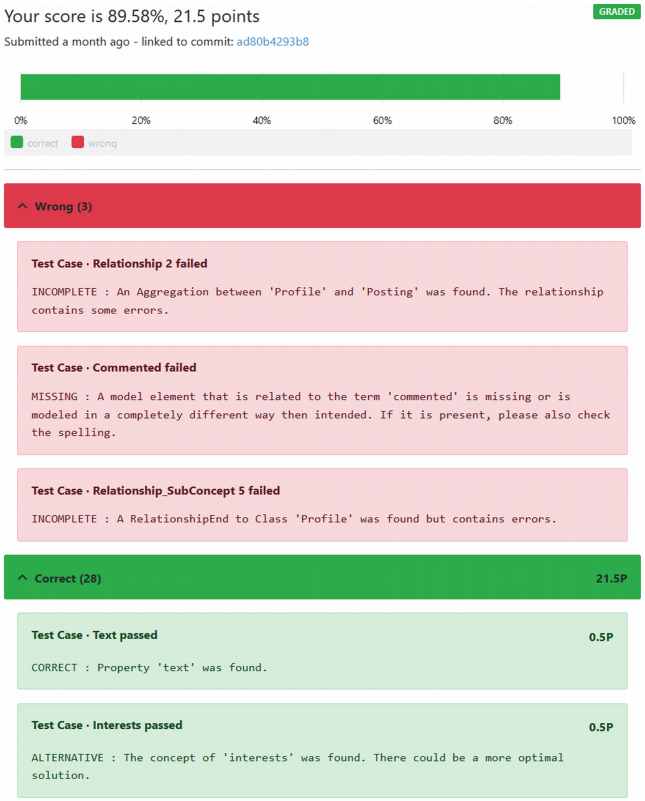

Figure 4: Excerpt of the feedback provided by Artemis for the student solution of Figure 3. The feedback was generated by a EVL representation of an provided assessment model.

and received a high score. With this feedback, the student can now continue their work on the task.

Alternatively, student solutions can be assessed manually. The model assessment system in Artemis works based on a grading instructions system for evaluators. Here, the student's solution is shown to an evaluator. The evaluator can now drag and drop prepared grading instructions to the student model elements or, in edge cases, create their own on the fly. The instructions can be prepared beforehand and are imported for each task by an instructor. Our approach can be used to generate, from the same assessment model as the example before, the list of usable instructions for the evaluators. Listing 2 shows an example of the instructions generated from rule shown Listing 3. As a further note, Artemis can also use this process to train a similarity pipeline on the assessments to achieve a semi-automatic assessment. Standardized instructions can help to increase the learning speed and results of the training.

Lastly, on most occasions, having documentation for the assessment on hand is advantageous. One occasion could be a student asking questions about their assessment results. Another would be giving out a guide for a new evaluator who needs to grade an exam. Often, the creation of documentation artifacts for assessments is skipped due to time constraints. In that case, Markdown code can easily be created from the assessment model to create a new documentation or integrate information into an existing one.

The compiled Markdown documentation can then be distributed to tutors and exam evaluators, increasing the consistency of their assessments and helping them guide their students.

To prove the usefulness of our approach, we are currently conducting a case study in the context explained at the beginning of this section. To help the students prepare for the undergraduate course exams, the course instructors reworked three old exam tasks into exercise tasks for the automatic pipeline of Artemis. Students of undergraduate courses often struggle to use common UML modeling tools while limiting themselves to the reduced feature set of domain models. To resolve that problem, a custom modeling editor was designed and distributed to the course students. It uses a domain model meta-model defined in Ecore and allows the students to create domain models natively (cf. Figure 3). These models are stored in the EMF XMI format. As task-specific data, the instructor provides an expert solution designed on the given domain description. Since the editor for domain models is already available, instructors reuse it to create expert solution models. Figure 1 is an example of such an expert solution model. The case study uses a meta-assessment model of 80 rules. The meta-assessment evaluator (cf. Figure 2) generates approximately 250-300 assessment rules for the assessment model per task. This number is enough to cover the task's solution space without the need for manually generated assessment rules. These generated rules define checks for *correct*, *alternative*, *incomplete*, *missing*, and *info* solutions for each element in the student model based on each element in the task-specific expert solution. Also, a grading schema for domain models is provided to ensure consistent scoring and feedback on all tasks. The main focus of the case study lies in Artemis's automatic pipeline, which is designed as mentioned in the paragraph on the rule-based validation engine earlier in this section. Further publications will present the case study's results and the whole technological stack implemented for it.

5 Conclusion

The automatic assessment of tasks in computer science education is still an active research topic. With the still-rising number of students enrolled in universities each year, new forms of education support are developed constantly. However, the research often results in new assessment systems with their own technology stack. In contrast, due to the resources needed to create assessment tasks, instructors often strive to stick to one technology stack. This work proposes a new model-driven approach to separate and bridge the changing technical space of the assessment systems and the technical space of instructors. Separating the two technical spaces allows each one to vary without interfering with the other, facilitating reuse and the buildup of open resource databases. The work proposes the use of an assessment model as a bridging technology. This model holds all the information required for an assessment task in a technologically independent way. The model can then be used to generate technology-specific data sets for targeted assessment systems. The instructors can use their technological stack to fill the assessment model in the bridging technical space. Also, a meta-assessment model was described that can be used to generate task-specific assessment rules from common task-unspecific meta-rules to help with the creation of a complete assessment model. An example of the assessment of domain models was given to showcase the usage of the approach. Here, the assessment systems, an automatic rule-based assessment, a manual assessment via grading instructions, and assessment documentation for other use cases are discussed. The assessment model could be used to provide these assessment systems with assessment data using their data representation. We plan to provide details on the complete example in connection with the full implementation and the results of an ongoing case study in a future publication.

Acknowledgments

The European Social Fund (ESF Plus) and the German Federal State of Saxony have funded this work within the project ProSECO (100687967). This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - SFB 1608 - 501798263.

References

[1] Jean Bezivin and Ivan Kurtev. 2005. Model-based Technology Integration with the Technical Space Concept. In *Proceedings of the Metainformatics Symposium*. Springer-Verlag, Berlin, Heidelberg.

[2] Weiyi Bian, Omar Alam, and Jorg Kienzle. 2019. Automated Grading of Class Diagrams. *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 700–709. https://doi.org/10.1109/MODELS-C.2019.00106

[3] Younes Boubekeur, Gunter Mussbacher, and Shane McIntosh. 2020. Automatic assessment of students' software models using a simple heuristic and machine learning. *Proceedings - 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS-C 2020 - Companion Proceedings*, 84–93. https://doi.org/10.1145/3417990.3418741

[4] Kevin Calderon, Nicolás Serrano, Carmen Blanco, and Iñigo Gutierrez. 2024. Automated and continuous assessment implementation in a programming course. *Computer Applications in Engineering Education* 32 (1 2024). Issue 1. https://doi.org/10.1002/cae.22681

[5] Eclipse Foundation Inc. 2012. Eclipse Epsilon. https://eclipse.dev/epsilon/, [Accessed: 01 June 2024].

[6] Rhaydae Jebli, Jaber El Bouhdidi, and Mohamed Yassin Chkouri. 2024. A Proposed Algorithm for Assessing and Grading Automatically Student UML Diagrams. *International Journal of Modern Education and Computer Science* 16 (2 2024), 37–46. Issue 1. https://doi.org/10.5815/ijmecs.2024.01.04

[7] Stephan Krusche. 2022. Semi-Automatic Assessment of Modeling Exercises using Supervised Machine Learning. *Proceedings of the 55th Hawaii International Conference on System Sciences*, 871–880. https://doi.org/10.24251/HICSS.2022.108

[8] Stephan Krusche and Andreas Seitz. 2018. ArTEMiS: An Automatic Assessment Management System for Interactive Learning. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 284–289. https://doi.org/10.1145/3159450.3159602

[9] Martin Morgenstern and Birgit Demuth. 2018. Continuous Publishing of Online Programming Assignments with INLOOP. *Software Engineering (Workshops)*, 32–33.

[10] Steffen Staab, Tobias Walter, Gerd Gröner, and Fernando Silva Parreiras. 2010. Model Driven Engineering with Ontology Technologies. In *Reasoning Web. Semantic Technologies for Software Engineering, 6th International Summer School 2010, Dresden, Germany, August 30 - September 3, 2010. Tutorial Lectures*, Uwe Aßmann, Andreas Bartho, and Christian Wende (Eds.). Lecture Notes in Computer Science, Vol. 6325. Springer, 62–98. http://dx.doi.org/10.1007/978-3-642-15543-7

[11] Michael Striewe and Michael Goedicke. 2011. Automated checks on UML diagrams. *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, 38–42. https://doi.org/10.1145/1999747.1999761

[12] Pete Thomas, Kevin Waugh, and Neil Smith. 2007. Learning and automatically assessing graph-based diagrams. *Beyond Control: learning technology for the social network generation. Research Proceedings of the 14th Association for Learning Technology Conference*, 61–74.

[13] Meike Ullrich, Constantin Houy, Tobias Stottrop, Michael Striewe, Brian Willems, Peter Fettke, Peter Loos, and Andreas Oberweis. 2023. Automated Assessment of Conceptual Models in Education: A Systematic Literature Review. https://doi.org/10.18417/emisa.18.2

[14] Chengguan Xiang, Ying Wang, Qiyun Zhou, and Zhen Yu. 2024. Graph semantic similarity-based automatic assessment for programming exercises. *Scientific Reports* 14 (12 2024). Issue 1. https://doi.org/10.1038/s41598-024-61219-8