

This is a repository copy of *An Online Education Platform for Teaching MDE*.

White Rose Research Online URL for this paper:

<https://eprints.whiterose.ac.uk/217659/>

Version: Accepted Version

Proceedings Paper:

Barnett, Will, Zschaler, Steffen, Boronat, Artur et al. (2 more authors) (2023) An Online Education Platform for Teaching MDE. In: Proceedings - 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS-C 2023. 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS-C 2023, 01-06 Oct 2023 Proceedings - 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS-C 2023 . Institute of Electrical and Electronics Engineers Inc. , SWE , pp. 114-121.

<https://doi.org/10.1109/MODELS-C59198.2023.00035>

Reuse

This article is distributed under the terms of the Creative Commons Attribution (CC BY) licence. This licence allows you to distribute, remix, tweak, and build upon the work, even commercially, as long as you credit the authors for the original work. More information and the full terms of the licence here:

<https://creativecommons.org/licenses/>

Takedown

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

An Online Education Platform for Teaching MDE

Will Barnett

King's College London

London, United Kingdom

Email: will.barnett@kcl.ac.uk 

Steffen Zschaler

King's College London

London, United Kingdom

Email: steffen.zschaler@kcl.ac.uk 

Artur Boronat

University of Leicester


Leicester, United Kingdom

Email: artur.boronat@leicester.ac.uk 

Antonio Garcia-Dominguez

University of York

York, United Kingdom

Email: a.garcia-dominguez@york.ac.uk 

Dimitris Kolovos

University of York

York, United Kingdom

Email: dimitris.kolovos@york.ac.uk 

Abstract—The setup and configuration of Model-Driven Engineering (MDE) tools is not straightforward because the MDE tooling landscape is highly fragmented. Also, many MDE tools are research prototypes with limited documentation. In an education setting where the aim is to teach MDE, having to spend time setting up and configuring tools reduces the amount of time learners have available to focus on the concepts being taught. Although certain tools, such as Epsilon and Umple, offer web-based playgrounds for their specific tools, they do not cover the full range of MDE activities. By generalising and extending the Epsilon Playground, we have created an education platform that can support a variety of MDE tools and be configured by teachers to use for their learning activities. We provide an overview of the platform's architecture and give an example of the tool and activity configurations using an Epsilon Validation Language (EVL) activity. We demonstrate the support for multiple tools with an Object Constraint Language (OCL) example and discuss key design decisions and the plan for future work. We hope that the education platform described here will provide opportunities for collaboration on the creation and dissemination of learning resources for the teaching of MDE.

Index Terms—MDE, education, frameworks

I. INTRODUCTION

Model-Driven Engineering (MDE) is a paradigm where models play a central role in the development of a software system. Over the last couple of decades MDE has been an area of active research with advancements in techniques and tools. In terms of education, there is a consensus that MDE is a complex subject to teach. Setting up teaching environments for MDE poses a challenge due to the complexity and availability of suitable tools [1] [2], hampering teaching activities.

Some MDE languages and tools now provide playground environments [3] [4] [5] that allow a user to experiment with the language in a web-browser without having to install any tools locally. This means that the user can concentrate on learning the language and its concepts. Playgrounds also offer a means to share examples created via a link.

The benefits of the playground concept can contribute towards the solution of making teaching environments easier to set up. When teaching MDE, however, tools are not used in isolation. Existing playgrounds, in contrast, only support

a single tool with no direct means of interoperability or customisation.

We propose developing a web-based education platform that can support a variety of MDE tools and be easily configured for a variety of learning activities. The platform should provide a foundation for extensions to support education-specific features that facilitate teaching MDE. Such a platform could be used in conjunction with MDE Open Education Resources [6] [7] to enhance the learners' experience of using OERs.

We present an education platform architecture and an implementation as a revision to the Epsilon Playground [3]. Our architecture generalises the hard-coded support for Epsilon to make the platform extensible in its support for additional MDE tools without the need to modify the platform's front-end source code. The platform adds: support for additional tools by way of tool services, type conversion of inputs to tool service functions, integration with GitHub repositories for persisting users' progress, and support for the customisation of panel layouts. The platform can be set up and run on a per institution basis. A global instance could be offered in the future, subject to available resources.

The paper is structured as follows: Section II identifies related work on web-based tools for teaching MDE. Section III introduces the education platform users and the scenarios that describe their interaction with the platform. Section IV presents a motivating example that is used in the remainder of the paper. Section V presents the platform's architecture. Sections VI and VII describe the tool and activity configurations, respectively. Section VIII describes the tool function invocation and type conversion protocol which enables tool integration. Section IX demonstrates new platform features through a non-epsilon Eclipse OCL tool example. Finally, Section X summarises our contributions and plan for future work.

II. RELATED WORK

In addition to the increasing number of playgrounds, there are web-based versions of IDEs such as Eclipse [8] [9] and

Visual Studio Code [10]. Some tools are only web-based; examples include AToMPM [11] and Freon [12].

There are no widespread web-based model formats for online editing and interchange of models that can be used to help integrate different tools of an education platform. An early initiative includes LlonWeb [13] however it is in the specification stage.

A web-based platform for the MontiCore language workbench based on JupyterLab [14] has been used for teaching the tutorials of a conference and lectures on the use and engineering of Domain Specific Languages (DSL). The platform's focus is on Monticore based DSLs and does not cover broader MDE or other tools.

An important consideration for an education platform is its cost to host and run must not be prohibitively expensive for education providers. The Epsilon Playground [3] architecture makes use of Functions-as-a-Service (FaaS) for its backend functions allowing on-demand scalability and minimal running costs when the platform is not being used.

III. SCENARIOS

The education platform has three user roles: Learner, Teacher, and Tool Provider. Learners access the platform to complete activities created by a Teacher. Teachers create lessons to deliver to their students as activities on the platform, and they make available the activity files from a location accessible to their students (e.g. on a web server or Git repository). Tool Providers create platform services for their existing tools. Activities use the platform tool services to perform MDE functions, such as model-to-model transformations. Tool Providers are responsible for making their tool services available: for example, by deploying them to a cloud hosting service.

The following descriptions give typical scenarios for each of the user roles.

A. Learner

Complete Activity A student follows the link from the teaching organisation's Virtual Learning Environment (VLE). They work through some of the activities but do not complete them before the tutorial is over. They save the activity and note the generated link to resume the activity at a later time.

Resume Activity A student follows the link that they generated at the end of their last tutorial, and the platform activities are reloaded at the previously saved state. The student completes the activities and saves their changes.

Export Activity A student decides to use the work they completed for the tutorial as a starting point for a new project. They follow the last link that they saved and the platform activities reload at the previously saved state. They export the activity's project files as a ZIP archive containing an IDE-compatible project. The student opens the project in a local IDE and begins adapting and extending their work.

B. Teacher

Create Activity A lecturer starts creating an example DSL project to demonstrate model-to-model transformations. They do not complete the example. Later, the lecturer finishes the example project and creates a configuration file for the education platform activities referring to the configuration file reference documentation. They push the example project and activity configuration to their teaching repository. The lecturer checks the activity they created by navigating to the web address of the deployed platform.

C. Tool Provider

Add tool A software engineer decides to integrate their DSL tool into the education platform. Using the platform documentation and examples, they plan the tasks they need to complete. They implement the back-end services, which provide the key functions of the tool, and deploy them to a cloud service. The software engineer creates the platform tool configuration referring to the platform reference documentation. They create the grammar files and run a local instance of the platform front-end to verify they are happy with the appearance of the available tool panels. The software engineer uploads the grammar files and tool configuration file to the front-end web server, and checks their tool is functioning correctly.

From these scenarios, we derive the following requirements for the education platform:

- R1** A student shall be able to start an activity given a URL.
- R2** A student's modifications to an activity shall be persisted at their request.
- R3** A student shall be able to obtain a copy of the files that are in an activity including any modifications made by the student.
- R4** A teacher shall be able to create activities.
- R5** A teacher shall be able to customise the visual appearance of activities without having to modify the source code of the platform.
- R6** The platform should minimise the workload for a teacher creating new activities.
- R7** A tool provider shall be able to extend the platform without having to modify the common front-end source code of the platform.
- R8** The platform should minimise the workload for a tool provider creating a tool for the platform.

Now that we have introduced the platform, the following section presents an example activity.

IV. RUNNING EXAMPLE

We re-use the Epsilon Validation Language (EVL) [15] example from the Epsilon Playground built-in example [3] and adapt it to our education platform as an activity. The EVL activity is used as a running example to demonstrate the education platform's functionality in Sections V to VII. The platform is available at this¹ Git repository for local de-

¹<https://github.com/mdenet/educationplatform-docker>

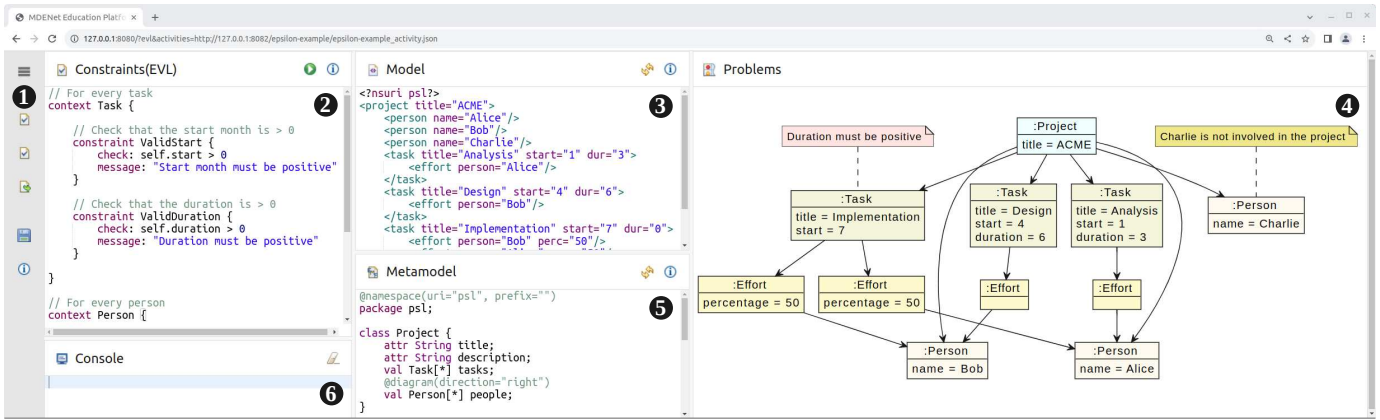


Fig. 1. The education platform Epsilon EVL Example.

velopment and testing. The repository documentation includes a video demonstrating the running activity.

Figure 1 shows the interface that learners see when competing the activity. There are five panels: constraints (EVL) ②, model ③, problems ④, metamodel ⑤, and console ⑥. The contents of ② are the constraints to check against the model and its metamodel from ③ and ⑤, respectively. The contents of ④ display the result of running the constraints for the model which is triggered when the learner clicks the run button that is on ②.

V. ARCHITECTURE

The education platform uses the architecture of the Epsilon Playground [3] as its foundation with additions and modifications to fulfil the requirements from Section III. The distinguishing feature of the platform's architecture its support for arbitrary tools without modification, whereas the Epsilon Playground's tool support is hard-coded.

There are four software components that make up the education platform: MDENetPlatform, ToolManager, ActivityManager, and ToolService. The components and their relationships are shown in Figure 2.

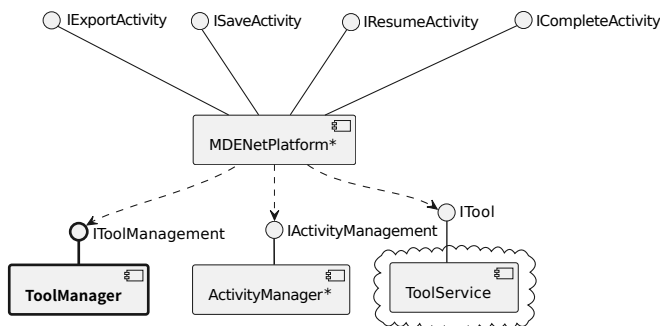


Fig. 2. Education platform components. Bold text indicates new components and "*" indicates modified components.

The ToolManager is a new component for providing arbitrary tool support. It is responsible for fetching and processing

the tool configurations that define a tool's panels, and fetching tool resources which include highlighting rules and icons.

The top-level component is the MDENetPlatform. It provides the front-end user interface and handles responses from ToolServices. It is a modified version of the playground's top-level component with support for arbitrary tools.

The ActivityManager component is responsible for fetching and processing the activity configuration, and fetching files displayed in an activity's panels. It is a modified version of the playground's Example Manager that has been restructured around learning activities and adds support for the customisation panel layouts.

The final ToolService component is responsible for handling requests to tool functions from users completing activities. Each tool provides its own service so an activity can make use of many ToolServices. They make-up the backend of the education platform and are deployable to a cloud platform.

To support saving of activity progress and use of private Git repositories, the education platform supports the use of a token server for GitHub authentication.

The education platform needs tools for activities to use, and activities for learners to complete. In Sections VI and VII, we present the configuration for the EVL example.

VI. TOOLS

Tool services provide the functionality that the installed tools on a developer's local machine environment would normally provide: for example, model-to-model transformation, text generation, or model validation. They make up the backend of the education platform. A tool service comprises a tool function and static resources.

The tool function provides an endpoint that conforms to the tool interface specification. The static resources a tool provider must create include: a tool configuration file, highlighting rules, and icons. The tool configurations are independent of activities, so a teacher only needs to reference a tool by its URL to use it in an activity they are creating.

A. Configuration

A tool configuration file defines the tool functions and the panels that are available for a platform activity to use. Listings 1 to 3 show the tool configuration for the EVL example.

Listing 1 shows the top-level structure with four attributes: `id`, `name`, `version`, `author`, `homepage`, `functions`, and `panelDefs`. The comments prefixed with '#' indicate values that have been given their own listings for clarity. The `functions` attribute declares the tool functions that are available to an activity. Finally, the `panelDefs` attribute declares the panels available to an activity.

```
1 tool:
2   id: epsilon
3   name: Epsilon
4   version: 0.0.1
5   author: Eclipse Epsilon
6   homepage: https://eclipse.dev/epsilon/
7   functions:
8     # Listing 2
9   panelDefs:
10    # Listing 3
```

Listing 1. Tool configuration top-level

A tool can declare multiple functions. Each function declaration has attributes `id`, `name`, `parameters`, `returnType`, and `path`. A declaration must correspond to a tool service's function which are described in Section VI-B. Listing 2 shows the EVL tool configuration function declaration for our example.

```
1 - id: function-evil
2   name: evil
3   parameters:
4     - name: program
5       type: evil
6     - name: metamodel
7       type: emfatic
8     - name: model
9       type: flexmi
10    - name: language
11      type: text
12    instanceOf: metamodel
13   returnType: text
14   path: http://127.0.0.1:8070/services/
      RunEpsilonFunction
```

Listing 2. Tool configuration function definitions

The `parameters` attribute lists the inputs of the tool function. Each parameter must have a `name` and a `type`. The `type` attribute indicates the language the function accepts: it is used by the platform for validating the inputs provided from panels, and converting the inputs if needed. The available types are defined by the `language` attributes of all the panel definitions included by an activity. Parameters that are models and are an instance of a metamodel have an `instanceOf` attribute that references the `metamodel` parameter by its name.

The `returnType` attribute declares the type of the produced output. It is used by the platform for automatic type conversions and displaying the response appropriately. Finally, the `path` attribute is a URL to the function's endpoint, where requests from the platform are to be sent.

A tool can define multiple panel types to be instantiated by the panels of an activity's configuration (as in Listing 6). Each panel declaration has attributes `id`, `name`, `panelclass`, `icon`, `language`, and `buttons`. Listing 3 shows the EVL tool configuration panel declaration for our example.

```
1 - id: evil
2   name: evil
3   panelclass: ProgramPanel
4   icon: evil
5   language: evil
6   buttons:
7     - id: action-button
8       icon: run
9       actionfunction: function-evil
10      hint: Run the program (Ctrl/Cmd+S)
11     - id: help-button
12       icon: info
13       url: https://www.eclipse.org/epsilon/doc/
14         evil/
15       hint: EVL Language Reference
```

Listing 3. Tool configuration panel definitions

The `panelclass` attribute specifies the class of panel the platform must create. Three are supported: `ProgramPanel`, `ConsolePanel`, and `OutputPanel`. The `ProgramPanel` provides a text editor that supports syntax highlighting, and a diagram viewer that can show a graphical representation of the editor contents. The `ConsolePanel` displays text that can be cleared. The `OutputPanel` provides a viewer for multiple text files.

The `icon` attribute specifies the image file to use from the `images` css static resource file. The `language` attribute declares the language identifier that is used for type checking and conversions.

The `buttons` attribute defines the buttons that are displayed on a panel. All buttons have `id`, `icon`, and `hint` attributes. The `icon` attribute specifies the image file to use from the `images` css static resource file as the button's icon. The `hint` attribute specifies the tooltip text to display when the user hovers over the button.

The concrete functionality of the button depends on the attributes given. In the case of an `actionfunction` attribute, the specified function will be executed when clicked. In the case of a `url` attribute, the specified URL will be opened in a new tab.

B. Service

The tool service functions are stateless FaaS that conform to the platform's tool interface. The requests and responses utilise JSON, which means that a tool provider can use any technology supporting JSON to implement a tool function as long as they comply with the tool interface specification.

Table I specifies the tool interface request JSON keys and their expected value contents. Listing 4 shows a compliant request for our EVL example.

The `[function parameter name]` key represents the names of the tool function parameters for which there may be more than one. The parameters are file contents encoded as UTF-8

```
1 | {
```

TABLE I
TOOL REQUEST SPECIFICATION

Key	Value	Mult.
[function parameter name]	File contents required for the action	0..*
language	Tool language identifier	1..1

TABLE II
TOOL RESPONSE SPECIFICATION

Key	Value	Mult.
output	String	1..1
[*diagram*]	SVG image	0..1
generatedText	Output file text	0..1
error	Error log text	0..1

```

2 | "program": "// For every task\ncontext Task {\n
  |   n\t\n\t// Check that the start month is >
  |   0\d...",
3 | "metamodel": "@namespace(uri="psl", prefix="")\n
  |   \npackage psl;\n\nclass Project {\n\tattr
  |   String title;\n\d...",
4 | "model": "<?nsuri psl?>\n<project title="ACME
  |   ">\n\t<person name="Alice"/>\n\t<person
  |   name="Bob"/>\d...",
5 | "language": "evl"
6 | }

```

Listing 4. The request for the EVL example.

strings that are the inputs required to generate the function’s output. The values of each file will be the contents of a panel mapped by the playground activity configuration.

The language attribute is the language identifier of the source panel initiating the tool function. It can be used by the tool function to handle multiple languages as single service.

Table II specifies the tool interface response JSON keys and their expected value contents. Listing 5 shows a compliant response for our EVL example.

```

1 | {
2 |   "validatedModelDiagram": "<?xml version
  |     =\ "1.0\ "
3 |     encoding=\ "UTF-8\ " standalone=\ "no\ "?>
4 |     <svg>
5 |       ...
6 |     </svg>",
7 |   "output": ""
8 | }

```

Listing 5. The response for the EVL example.

The response `output` key value is textual output from the execution of the tool function: for example, the console output. A response key that contains the keyword ‘diagram’ is expected to be a diagram in Scalable Vector Graphics (SVG) image format. The `generatedText` key value is the output file contents generated by the tool function as a string. The `error` key value is the textual log of any errors that occur during execution of the function. The `output` key of a response is mandatory while all other keys are optional. Every response key can only be specified once.

VII. ACTIVITIES

Education platform activities are created by a teacher to demonstrate learning objectives from the course syllabus being taught. An activity is presented to the learner as a single web page with a collection of panels. The activities available to a learner are listed in a navigation menu on the left-hand side of the page. For our EVL example shown in Figure 1, there are three activities indicated by the buttons in the menu ①. The first three menu buttons are the activities, and the remaining two are the save and about buttons that appear on every page.

There are three steps to a teacher creating an activity:

- 1) Create the project files.
- 2) Create the activity configuration file.
- 3) Make the activity available.

The project files make up the majority of an activity. They can be created using the local development environment for any of the tools supported by the platform. For our example, the Eclipse IDE can be used to edit and debug the activity project until it meets the learning objectives of the activity.

Following the creation of the project files, the platform must be configured so that the project is presented as activities the teacher designs by creating the activity configuration file.

A. Configuration

An activity configuration file created by a teacher defines the presentation of project files within the education platform. The file formats accepted by the platform are YAML [16] and JSON [17]. Two kinds of common attributes used throughout the platform configuration files are `id` and `name`. An `id` uniquely identifies the object that it is an attribute of. A `name` or `title` is the text to use in user interfaces for the object. Listings 6 to 9 show the activity configuration for our example.

Listing 6 shows the top-level structure. An activity has seven attributes: `id`, `title`, `icon`, `tools`, `layout`, `actions`, and `panels`. The comments prefixed with ‘#’ indicate values that have been given their own listings for clarity. The `icon` attribute specifies the image file to use from the images css static resource file for the icon in the activity menu ①.

```

1 | activities:
2 | - id: evl
3 |   title: Validate Project
  |     Plan
4 |   icon: evl
5 |   tools:
6 |     - http://127.0.0.1:8070
  |       /epsilon_tool.json
7 | layout:
8 | # Listing 7
9 | actions:
10 | # Listing 8
11 | panels:
12 | # Listing 9

```

Listing 6. Activity configuration top-level

The `tools` attribute specifies the tool service URLs for all of the tools that are used by an activity. Upon loading an activity, tool configurations that contain panel definitions are fetched from the tool urls. The `layout` attribute specifies the relative positions of the panels. The `actions` attribute specifies the mapping of panels to the input parameters of tool functions which are requested when a button is pressed. Finally, the `panels` attribute defines the panels and the project file that are displayed.

An activity must define a layout that has an `area` attribute with a two-dimensional array of panel IDs. The rows and columns of the array naturally map to the layout of the panels displayed by the platform. Listing 7 shows the EVL activity configuration layout specification.

```

1 | area:
2 | - [ panel-evl,      panel-model, panel-problems]
3 | - [ panel-console, panel-mm,      ]

```

Listing 7. Activity configuration layout

The `area` attribute specifies an array that has two rows and three columns. The `evl`, `model`, and `problems` panels are displayed on the top row with the console and metamodel panels being displayed on the bottom row as shown in Figure 1. Any empty positions in the `area` array are filled in with the panels of the same column by the platform when the panels are laid out.

An activity can define multiple actions. Each action definition has attributes `source`, `sourceButton`, `parameters`, and `output`. Listing 8 shows the EVL activity action configuration for our example. The `source` and `sourceButton` attributes specify the button that the mapping applies to. The `source` is a panel id from the `panels` attribute of the same activity configuration file.

```

1 | - source: panel-evl
2 |   sourceButton: action-button
3 |   parameters:
4 |     program: panel-evl
5 |     flexmi: panel-model
6 |     emfatic: panel-mm
7 |   output: panel-problems

```

Listing 8. Activity configuration actions

The `sourceButton` is a button id from the `buttons` attribute of the panel's definition in its tool configuration file. The `parameters` attribute (lines 6 to 10) maps each parameter of a tool function to the ID of the panel whose contents will be used as input. For example, the `program` parameter is mapped to `panel-evl` for its input.

The `output` attribute (line 12) specifies the panel that will display the result of the tool function. In this example, it is the Problems panel ④.

An activity can define multiple panels. Each panel definition has attributes `id`, `name`, `ref`, and `file`. Listing 9 shows the EVL activity configuration panel definition for our example.

```

1 | - id: panel-evl
2 |   name: Constraints (EVL)
3 |   ref: evl
4 |   file: psl-evl

```

Listing 9. Activity configuration panels

The `ref` attribute is a reference to a panel definition ID from one of the tools configuration files specified by the `tools` attribute in Listing 6. The reference determines the type of panel that is created and the functions available to it. Finally, the `file` attribute specifies the path, relative to the configuration file location, of the file the panel will display.

VIII. TYPE CONVERSION AND FUNCTION INVOCATION

A problem that increases with the addition of multiple tools is the proliferation of different input types to tool service functions. One approach is for the tool functions to provide versions of the function that accept the different input types. However, this does not scale as each tool function has to handle every combination of type available in the platform, resulting in increased code duplication and coupling.

An alternative approach is to support the conversion of types using dedicated conversion functions when a tool function is requested. This reduces the types that tool functions need to support. To achieve this the platform must:

- 1) Compare the input value types against the expected tool function types.
- 2) Determine the conversion functions to use and call them.
- 3) Call the requested tool function.

The Epsilon Playground architecture used as the platform foundation provides no support for either 1 or 2 and fixed support for 3. Therefore, we add support for 1 and 2 and extend 3 to support the invocation of tool functions from declarations. The following pseudocode shows how the platform handles type conversion when a tool function is triggered by a user clicking on a run action button.

```

1: params: map of activity action parameters to their type
   and value
2: tf: object describing a tool function
3: for all p in params do
4:   tfp ← tf.getParam(p)
5:   if p.type ≠ tfp.type then
6:     if ¬tfp.hasMetamodel() then
7:       cr ← convert(p.value, p.type, tfp.type)
8:     else
9:       cr ← convertIncludingMetamodel(
           p.value, p.type,
           mm.value, mm.type, tfp.type)
10:    end if
11:    requestData[p.name] ← cr
12:  else
13:    requestData[p.name] ← p.value
14:  end if
15: end for
16: call(tf.id, requestData)

```

When a run action button is pressed, a `params` object is created using the corresponding button's action from the activity configuration file. The `params` object maps function names to a value and type. The value is the input to the tool function and is the contents of a displayed panel that is specified by the configuration file's activity parameter to panel mapping.

For each of the parameters in `params`, Line 3, the platform checks to see if the parameter type matches the corresponding tool function's parameter type, Line 5, to determine if any type conversion is necessary. If the types match, the value `params` is inserted into the `requestData` array on Line 13. If the types do not match, the platform tries to convert the input value to

a type that matches the tool function’s parameter type using a conversion function.

Models are an instance of their metamodel, so to convert a model, the metamodel is required by the conversion function. Since metamodels do not have such a dependency, the platform handles each of these cases separately. On Line 6 the parameter is checked for a metamodel dependency. If there is no dependency, the platform converts the input parameter using the `convert()` function. This function has three parameters: input value, input type, and target type. If there is a dependency, the platform converts the input parameter using the `convertIncludingMetamodel()` function. This function has five parameters: input value, input type, metamodel value, metamodel type, and target type. The result of either conversion is assigned to `cr` on Lines 7 or 9 that is inserted into the `requestData` array on Line 11.

Following all the parameters in `params` being processed, the `requestData` variable holds inputs to the tool function with the types it expects. The tool function is finally called by the `call()` function on Line 16. The `call()` has two parameters, the id of the tool function and an array containing the parameters and their values.

To minimise the complexity of the platform type conversion, only direct conversions using a single function are considered so functions are not chained. This means that there needs to be a suitable conversion function available for the conversion to be successful and the tool action function called.

IX. EXAMPLE: USING A DIFFERENT CONSTRAINT LANGUAGE

A goal of the education platform is to support multiple tools and configurable activities for teaching MDE. We demonstrate that this goal has been achieved using an Eclipse OCL tool and activity example. The Object Constraint Language (OCL) [18] can be used to validate well-formedness of a model. We developed a platform tool service using the Eclipse OCL implementation’s [19] standalone API. The education platform and OCL tool service are available to run locally from this² Git repository.

The OCL tool service has three inputs: the OCL constraints to check, the model, and the metamodel. The model is in the XML Metadata Interchange format (XMI) [20] and the metamodel is in the Ecore XMI format [21, p. 20]; these correspond to the `ecore` and `xmi` types.

To show the type conversion of the education platform in our example, we provide the activity’s model in the Epsilon Flexmi [22] format that the OCL tool function does not support³. This means that the education platform must convert the model via its type conversion functionality prior to calling OCL tool function.

²<https://github.com/mdenet/educationplatform-docker>

³Technically, the OCL implementation supports any EMF resource and the Flexmi parser is implemented as an EMF resource. This means that the OCL tool function could be refined to load EMF resources more generally, rather than explicitly convert between the exact types.

In order to carry out the type conversions suitable conversion functions must be provided. Therefore, as part of developing our example an `EmfaticToEcore` function and the Epsilon tool was extended to include a `FlexmiToXmi` function. The network features of the Chrome web browser were used to verify the requests made by the platform to its tool services.

Figure 3 shows the OCL activity with its four panels: model ①, OCL constraints ②, metamodel ③, and console ④. The library example used is from the Eclipse OCL user documentation tutorial [23, p. 3]

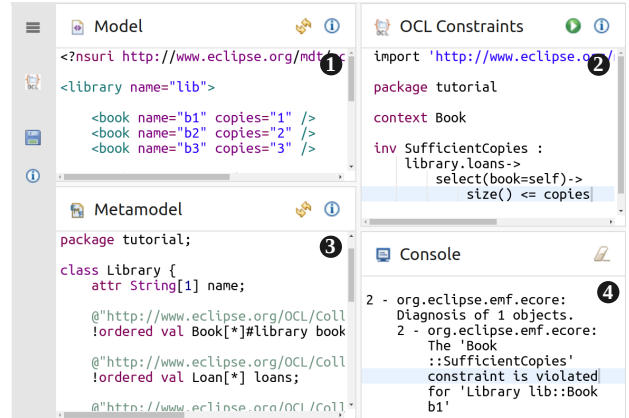


Fig. 3. The education platform Eclipse OCL example.

When the action button is clicked on ② the platform determines the conversions that need to be carried out to be able to call the OCL function and then calls them as required.

Figure 4 shows the calls the platform makes to the tool services. In total three functions are called. First, `EmfaticToEcore` to convert the metamodel, because the Epsilon tool service’s `FlexmiToXmi` function does not accept the metamodel in Emfatic format as used in ③. Second, the

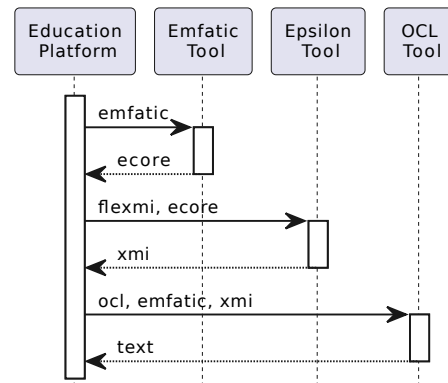


Fig. 4. Type conversions the platform makes for the OCL example.

`FlexmiToXmi` function is called to convert the model in ① to the format that is accepted by the OCL function. Finally the OCL function is called that returns the validation result as text that is displayed by the platform in ④.

X. CONCLUSIONS AND FUTURE WORK

In this paper, we presented an online education platform for teaching MDE that generalises and extends the Epsilon Playground to support multiple tools and customisable activities. The two examples of education platform activities can be used to show that some of the requirements from Section III have been met.

R1 and **R4** are fulfilled by the first running example that introduces the new configuration file format. **R5** and **R7** are fulfilled by the second example that demonstrates an additional constraint language tool with no changes to the platform front-end code.

The examples demonstrate **R3** and **R8** have been partially met. For **R3**, the loading of files has been demonstrated, however not the retrieval of persisted user modifications (though this is possible using an included token server). For **R8**, the automatic type conversion measure implemented to satisfy the requirement has been shown.

To ensure that all the requirements have been met, further validation needs to be completed. For **R6** and **R8** HCI aspects need to be considered, and for **R2** and **R3** persistence of user modifications to activity must be demonstrated.

To support our goal of the platform being used for teaching MDE it would be beneficial for additional tools to be developed by providers. Having a greater number of tools available increases the diversity of the platform so it can offer a richer learning experience. To this end, there is ongoing work developing a tool service for YAMTL⁴, and adding support for language workbenches like Xtext⁵ and Langium⁶.

As future work, we plan to complete the support for language workbenches as they offer a new category of activity that can be utilised for teaching important MDE concepts. To increase the diversity of tools offered by the platform, we will collaborate with tool providers to develop new platform tool services. We also plan to improve the usability of the platform by engaging with the MDE community to create open education resources that use it, and develop editors for configuration files to assist users in creating activities and tools. Finally, we will investigate the support for graphical languages that use frameworks like Sirius⁷.

ACKNOWLEDGMENTS

Zschaler and Barnett's contribution was partly funded by the UK Engineering and Physical Sciences Research Council (EPSRC) through the MDENet grant (EP/T030747/1).

REFERENCES

- [1] A. Bucchiarone, J. Cabot, R. F. Paige, and A. Pierantonio, "Grand challenges in model-driven engineering: an analysis of the state of the research," *Software and Systems Modeling*, vol. 19, pp. 5–13, 2020.

⁴<https://dl.acm.org/doi/10.1145/3239372.3239386>

⁵<https://eclipse.dev/Xtext/>

⁶<https://langium.org/>

⁷<https://projects.eclipse.org/projects/modeling.sirius>

- [2] F. Ciccozzi, M. Famelis, G. Kappel, L. Lambers, S. Mosser, R. F. Paige, A. Pierantonio, A. Rensink, R. Salay, G. Taentzer, A. Vallecillo, and M. Wimmer, "How do we teach modelling and model-driven engineering? a survey," in *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2018, p. 122–129. [Online]. Available: <https://doi.org/10.1145/3270112.3270129>
- [3] D. Kolovos and A. Garcia-Dominguez, "The epsilon playground," in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2022, p. 131–137. [Online]. Available: <https://doi.org/10.1145/3550356.3556507>
- [4] M. Rudolph. (2023, 01) The langium playground – typefox blog. [Online]. Available: <https://www.typefox.io/blog/langium-playground>
- [5] I. Eclipse Foundation. Sirius — sirius webhome. [Online]. Available: <https://www.eclipse.org/sirius/sirius-web.html>
- [6] A. Bucchiarone, A. Vazquez-Ingelmo, G. Schiavo, A. Garcia-Holgado, F. Garcia-Penalvo, and S. Zschaler, "Designing learning paths with open educational resources: A case study in model-driven engineering," in *18th Iberian Conference on Information Systems and Technologies*, Mar. 2023.
- [7] A. Bucchiarone, A. Vázquez-Ingelmo, G. Schiavo, S. Barandoni, A. García-Holgado, F. J. García-Peñalvo, S. Mosser, A. Pierantonio, S. Zschaler, and W. Barnett, "Towards personalized learning paths to empower competency development in model driven engineering through the encore platform," ser. MODELS '23, 2023.
- [8] I. Eclipse Foundation. (2020, 03) Eclipse orion. [Online]. Available: <https://projects.eclipse.org/projects/ecd.orion>
- [9] —. Theia - cloud and desktop ide platform. [Online]. Available: <https://theia-ide.org/>
- [10] Microsoft. (2023, 07) Visual studio code for the web. [Online]. Available: <https://code.visualstudio.com/docs/editor/vscode-web>
- [11] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin, "Atompm: A web-based modeling environment," vol. 1115, 2013, Conference paper, p. 21 – 25.
- [12] J. Warmer and A. Kleppe, "Freon: An open web native language workbench," in *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, 2022, p. 30–35. [Online]. Available: <https://doi.org/10.1145/3567512.3567515>
- [13] M. Voelter, F. Tomassetti, and N. Stotz. (2023, 05) Lionweb. [Online]. Available: <https://github.com/LionWeb-org>
- [14] J. C. Charles, N. Jansen, J. Michael, and B. Rumpe, *Teaching the Use and Engineering of DSLs with JupyterLab: Experiences and Lessons Learned*, ser. GI-Edition. Proceedings / Gesellschaft für Informatik. Bonn: Gesellschaft für Informatik e.V., Jun 2022, vol. P-324, pp. 93–110, datenträger: CD-ROM. [Online]. Available: <https://publications.rwth-aachen.de/record/861221>
- [15] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, *On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 204–218. [Online]. Available: https://doi.org/10.1007/978-3-642-11447-2_13
- [16] I. dot Net, T. Müller, P. Antoniou, E. Aro, and T. Smith. (2021, 10) Yaml ain't markup language (yaml™) revision 1.2.2. [Online]. Available: <https://yaml.org/spec/1.2.2/>
- [17] Ecma, *ECMA-404: The JSON data interchange syntax*. Geneva, Switzerland: Ecma International – European Association for Standardizing Information and Communication Systems, December 2017. [Online]. Available: https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf
- [18] "Object constraint language," Feb. 2014, version 2.4. [Online]. Available: <https://www.omg.org/spec/OCL/2.4/PDF>
- [19] E. Foundation. (2023, 06) Eclipse ocl™ (object constraint language). [Online]. Available: <https://projects.eclipse.org/projects/modeling.mdt.ocl>
- [20] "Xml metadata interchange," Jun. 2015, version 2.5.1. [Online]. Available: <https://www.omg.org/spec/XMI/2.5.1/PDF>
- [21] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed. Pearson Education Inc, 2008.
- [22] D. Kolovos and A. de la Vega, "Flexmi: a generic and modular textual syntax for domain-specific modelling," *Software and Systems Modeling*, 2022. [Online]. Available: <https://doi.org/10.1007/s10270-022-01064-3>
- [23] C. Damas, A. Sánchez-Barbudo Herrera, A. Uhl, E. Willink *et al.*, *OCL Documentation*, Eclipse Foundation. [Online]. Available: <https://download.eclipse.org/ocl/doc/6.4.0/ocl.pdf>