The OpenPonk modeling platform

Peter Uhnák Robert Pergl

Department of Software Engineering
Faculty of Information Technology
Czech Technical University in Prague
Czech Republic
{uhnakpet,robert.pergl}@fit.cvut.cz

Abstract

In this paper we present OpenPonk: a free, open-source, simple to use platform for developing tools for conceptual modeling: diagramming, DSLs, and algorithms operating on the models and diagrams, such as automatic layouting, model transformations, validations, etc.

This project differentiates itself from the current efforts by providing completely free and open-source live development environment, which is simple to learn, use, and extend.

There are already several plugins and extensions that bring several notations and algorithms, some of which are presented in this paper, alongside the overview of the core of the platform, and how they integrate with each other. We also present a comprehensive project case study utilizing OpenPonk.

Keywords OpenPonk, modeling, visualizations, Pharo, DynaCASE, Roassal, UML, BORM

1. Introduction

In this paper we present OpenPonk (formerly known as DynaCASE) – an emerging modeling platform implemented in the live environment Pharo[2].

In all engineering endeavours, engineers utilize various types of diagrams that help them analyze and design their complex systems; civil engineers use CAD tools, software engineers use IDEs and CASE tools, and enterprise engineers use CABE tools. Also there are many research groups that focus on research in some of the aspects of these tools.

Development of such tools is very demanding, because a lot of effort has to be invested into creating the foundation of the tool such as graphical visualization, interaction of graphical objects, persistence, layouting, and general user interface. To do these projects in high quality requires big budgets and teams. However, there are often small or mid-sized research groups and individual practitioners who have an idea that they would like to implement, whether their own modeling notation, specific algorithms, model transformations, simulations, etc. If they attempt to implement their tool from the scratch, the resulting product ends up often subpar and isolated from other tools, consequently not used in the end by a larger audience, thus wasting the ideas and invested resources. Additionally, resources that should have been invested into the research itself have to be wasted on reimplementing solved problems.

This is why we started the OpenPonk platform. We want to give designers of tools a platform which solves recurring tasks mentioned above, so they may focus on the core of their needs.

Our aim is not to replace or compete with existing industry-standard solutions for standardized notations, such as UML and Enterprise Architect, although there is some overlap of functionality. The main focus is to provide platform for tool building for the long tail of non-standard custom models and algorithms for both research and business.

The core of our tool and current extensions are available as open-source under the MIT license[1].

1.1 Goals and objectives of the paper

The purpose of this paper is to present the current state of OpenPonk. In the first part, we present a high-level overview of the *core* architecture and design behind OpenPonk. In the second part we overview the general approach and architecture of user-provided *plugins*, and how they connect to the core. In the third part, we present several plugins and extensions that we currently provide: simulating finite-state automata, modeling business entities using BORM, describing models with custom DSLs, and live model manipulation powered by MetaLinks. In the fourth part we present a use case study of using OpenPonk as a foundation for existing research efforts and the support for UML Class Diagrams

[1] https://openponk.github.io

with round-trip engineering for CORMAS. Finally, we discuss related solutions that are currently available.

2. Architecture and design

In this section we present the core architecture of the system and high level view of the *Plugin Architecture*, which is explained in greater detail in the following section.

2.1 Terminology note

We further refer to the foundation of the platform as the *core*, and to user-supplied models, notations, and components as *plugins*.

Figures or *visual elements* are entities drawn on canvas, usually representing a particular model element.

Finally, we use the term *user* to refer to *user-developer*, i.e. a person that is using our platform as a basis to create or extend a plugin/model/notation. For a user that is simply using our platform and its extensions as a tool, we will use the term *tool-user*.

2.2 Connections between models, view, and controllers

OpenPonk is foremost a meta-modeling platform. Our aim is to provide the foundation for building modeling tools for creating and operating on a wide range of models, especially models that have a direct visual representation – *notation*. To represent the model and manipulate it through the notation, we in principle follow the model-view-controller (MVC) design [19], however due to possible model limitations we limit the direct interaction between the model and view and use controllers instead, as depicted in Figure 1.

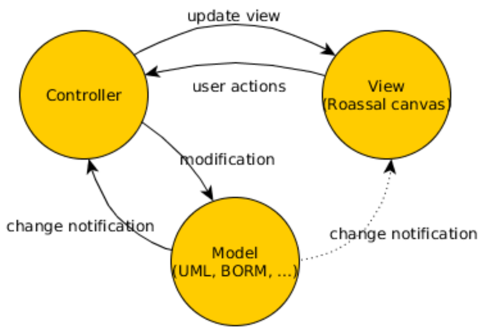


Figure 1. MVC design with limited communication between model and view

2.2.1 Model

The model is the most straightforward part, as it describes mere meta-models such as UML meta-model, BORM meta-model or Finite-State Automata meta-model.

2.2.2 View

The primary[2] view visualizes the notation of the meta-model. We have chosen Roassal visualization engine[1]. Roassal is a vector-based engine providing a support for a wide range of needs, such as chart and graph drawings, map visualizations, etc. It is also used by the Moose reengineering platform [13] for system complexity visualizations.

Importantly for us, Roassal provides a lower-level API for creating elementary shapes (e.g. ellipse, rectangle, line) and interactions (e.g. moving elements, resizing elements, zooming) that can be combined by the implementer to create appropriate notation elements. Furthermore, the core of Roassal is extensible. We created a range of new elements and interactions, many of which were contributed back to Roassal itself.

2.2.3 Controllers

Controllers are responsible for interpreting user-triggered signals coming from the view (adding a new figure, renaming an element, etc.) and for propagating updates to and from the model (meta-model instance). In the original MVC design[19], the view is directly observing the changes in the model and adopting them. We have diverged from this approach to handle non-observable models and to limit to the logic complexity in the view.

A lot of complex logic is hidden in the core controllers, thus simplifying the logic of the user-supplied ones.

2.2.4 Other components

The created models are organized in so-called *projects*. A project is a set of models and diagrams that are opened and stored together. The primary use case of projects is grouping together models describing various aspects of the modeled system.

OpenPonk additionally provides a set of prepared GUI components, and visual extensions that can enhance the produced tool.

3. Plugin Architecture

The concrete meta-models and their notations are developed as so-called *plugins* by inheriting general classes of the core. Such plugins are independent of each other and can be distributed separately.

Basic properties of every plugin are described in a subclass of *Plugin* class as shown in Figure 2. At the moment, specifying only essential properties is necessary: the name of the plugin, the containment (top-level) meta-model class (i.e. the model that represents the whole diagram), and a controller class for the containment model. Optionally, the user can specify an icon (used in various places of the GUI), version of the plugin, and serializer used for file-system persistence.

[2] Later we introduce secondary views, such as tree-views.

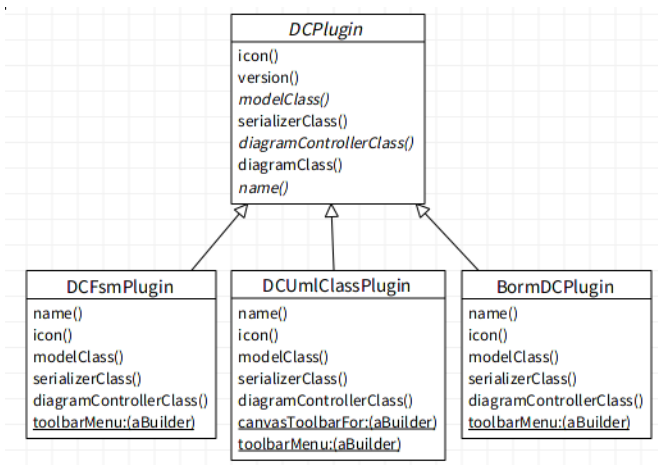


Figure 2. Plugin description classes

3.1 Model creation

The meta-model infrastructure is the backbone of a plugin, and all other components[3] revolve around it. One of our requirements for the platform is the ability for the users to use their models. This means not only standard models such as UML, but also custom models developed by the user to address their specific business requirements.

A meta-model typically consists of a set of implemented classes that represent subjects from the modeled domain. Figure 3 shows an example model of a Finite-State Automata (FSM) meta-model.

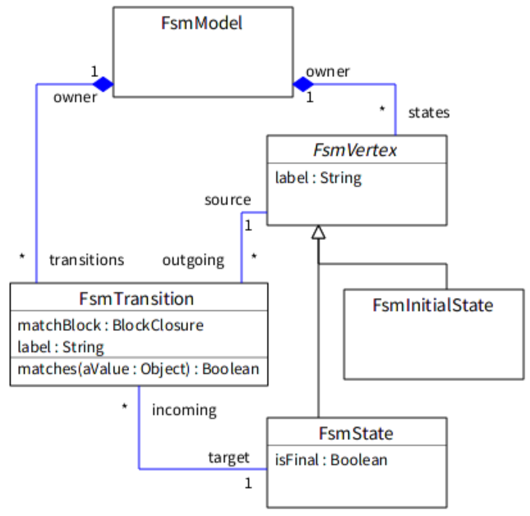


Figure 3. Example Finite-State Automata meta-model

Integrating models that have been developed independently of our tool however presents an integration challenge, as the model may inhibit some more advanced properties of the platform. More precisely, many components, e.g. the notation visualization in particular, require a mechanism through which changes in the model can be observed so that they can be updated accordingly. As we noted earlier, to address this need we delegate where possible the burden of updates to the controllers. Such an approach does inhibit some parts of the platform, as updates must go through the controller (using command pattern or through direct calls), or the user must explicitly inform the controller that changes to the model have been made. For the standard approach where the user creates and manipulates the model through the provided utilities (e.g. palette, canvas, form editor) this does not present a problem, as the interactions are automatically wrapped.

We use this approach for our UML plugin that utilizes the FAMIX[7] meta-model as the basis for the meta-model. Although we have extended the FAMIX infrastructure with additional classes required for UML, we have not modified the model itself; in fact neither was FAMIX created with our tool in mind, nor does it provide sufficient mechanisms for observing changes. Despite this, we were able to successfully create a diagramming plugin as the platform is powerful enough to handle the problems properly.

Naturally, if the model does provide the necessary mechanism, not only can the implementation be simplified as some burden is removed from the controllers, but additional possibilities open up.

3.2 Creating visual elements

Meta-models that we are particularly interested in have an accompanying visual diagram notation.

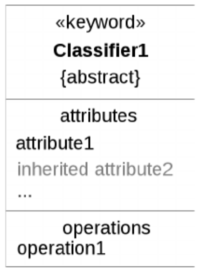
Notation	Diagram elements
 «keyword» Classifier1 {abstract} attributes attribute1 inherited attribute2 ... operations operation1	UMLClassifierShape (Classifier) - UMLKeywordLabel (Classifier) - UMLNameLabel (Classifier) - UMLLabel (Classifier) - UMLCompartment --- UMLLabel --- UMLLabel (Property) --- UMLLabel (Property, inherited) --- UMLLabel - UMLCompartment --- UMLLabel --- UMLLabel (Operation)

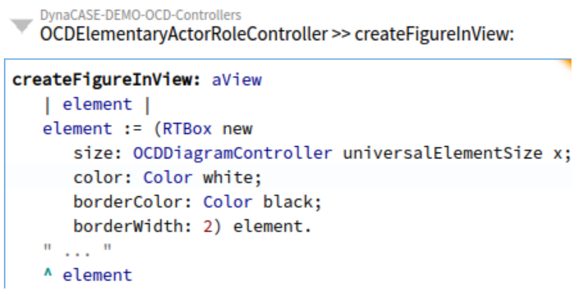
Figure 4. Composition of visual elements

In Figure 4 we can see a UML Class composed from several *primitive* shapes as prescribed by UML Diagram Interchange[14].

The creation of the visual element is stored in the appropriate controller in the`#createFigure` method. The only requirement of the method is that the returned object understands `renderIn: aRoassalView`. That way, the method can both directly use Roassal API and return a Roassal element [Figure 5], or add an intermediate layer [Figure 6].

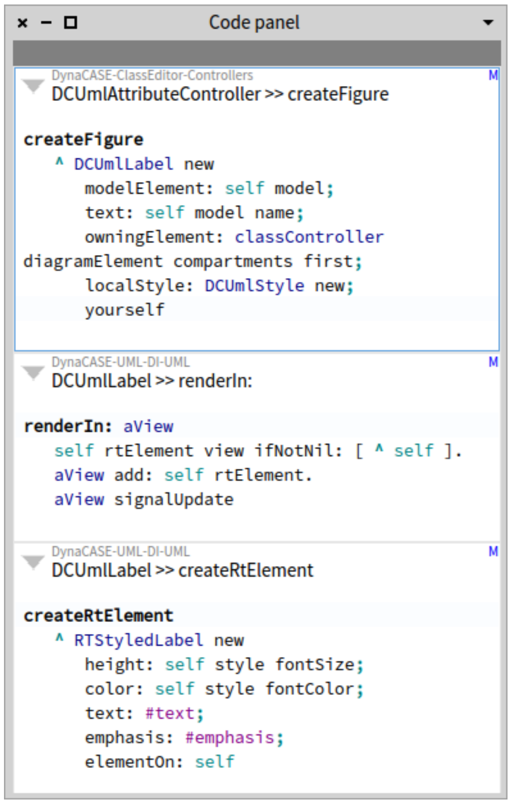
When the model changes, the figure typically has to reflect the new properties of it. The view however is not completely redrawn when an update is required, instead only the concerned parts are updated. To tell the platform what and

[3] e.g. visualization, simulation


DynaCASE-DEMO-OCD-Controllers
OCDElementaryActorRoleController >> createFigureInView:

```
createFigureInView: aView
   | element |
   element := (RTBox new
      size: OCDDiagramController universalElementSize x;
      color: Color white;
      borderColor: Color black;
      borderWidth: 2) element.
   " ... "
   ^ element
```

Figure 5. Returning Roassal elements/shapes



DynaCASE-ClassEditor-Controllers
DCUmlAttributeController >> createFigure

```
createFigure
   ^ DCUmlLabel new
      modelElement: self model;
      text: self model name;
      owningElement: classController
diagramElement compartments first;
      localStyle: DCUmlStyle new;
      yourself
```

DynaCASE-UML-DI-UML
DCUmlLabel >> renderIn:

```
renderIn: aView
   self rtElement view ifNotNil: [ ^ self ].
   aView add: self rtElement.
   aView signalUpdate
```

DynaCASE-UML-DI-UML
DCUmlLabel >> createRtElement

```
createRtElement
   ^ RTStyledLabel new
      height: self style fontSize;
      color: self style fontColor;
      text: #text;
      emphasis: #emphasis;
      elementOn: self
```

Figure 6. Using an intermediate layer

how should be updated, the user implements a *#refreshFigure* method in the accompanying controller. In this method updates such as changing text content, colors, adding and removing subelements, etc. can occur. The *#refreshFigure* is automatically called by the platform after a model change was detected, this can either be a result of the platform observing changes in the model (if possible), or when the model was modified through one of the platform's editing interfaces. With this approach the user does not need to concern themselves when an update should occur.

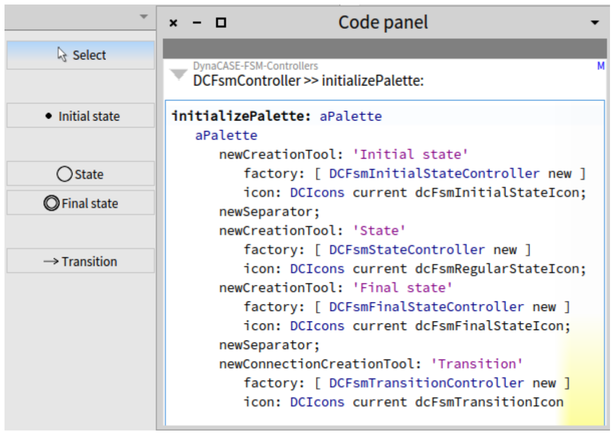
In addition, if the model provides sufficient observation granularity, the figure can directly observe the model and update accordingly.

3.3 Controllers

Controllers are the glue between the model and views. The most common approach is to have a controller for each model element (e.g. a `Method` and a `MethodController`, `Classifier` and a `ClassifierController`, etc.). Depending on the appropriate granularity, additional responsibility may be taken (e.g. `MethodController` also handling `Parameter` model elements).

Apart from responsibility of creating the view described in the previous sections, two more UI-related interfaces are provided:

1. Each model element has typically a different set of properties that are modified by the tool-user. Therefore, each controller is free to override the `buildEditor:` method and to specify a custom Form consisting of the appropriate form elements (e.g. input text, droplist, checkbox), and the binding between the model and the form elements. This form automatically opens when the tool-users selects a model element.
2. Each diagram (notation) is usually accompanied by a different palette. The responsibility of the *Diagram Controller* (the master controller for the diagram) is to implement the palette specification [Figure 7]. The platform will then handle the actual creation of the appropriate objects when the tool-user selects one of the items and interacts with the canvas.



DynaCASE-FSM-Controllers
DCFsmController >> initializePalette:

```
initializePalette: aPalette
   aPalette
      newCreationTool: 'Initial state'
         factory: [ DCFsmInitialStateController new ]
         icon: DCIcons current dcFsmInitialStateIcon;
      newSeparator;
      newCreationTool: 'State'
         factory: [ DCFsmStateController new ]
         icon: DCIcons current dcFsmRegularStateIcon;
      newCreationTool: 'Final state'
         factory: [ DCFsmFinalStateController new ]
         icon: DCIcons current dcFsmFinalStateIcon;
      newSeparator;
      newConnectionCreationTool: 'Transition'
         factory: [ DCFsmTransitionController new ]
         icon: DCIcons current dcFsmTransitionIcon
```

Figure 7. Palette specification

3.3.1 Connecting elements and live validation

Models elements, and their visual representations rarely live by their own. Instead, they are connected through references or compositions. The connection is accommodated by four functions implemented in the controllers: *#canBeSourceFor:*, *#canBeTargetFor:*, *#addAsSourceFor:*, *#addAsTargetFor:*; the full signature being *receivingController* For: aNewController*.

The purpose of the *#canBe*For:* methods is to decide whether the receiving controller accepts the argument, if they return *false*, the connection cannot proceed and the #*addAs*For:* methods will not be called.

The #*addAs*For:* methods contain the behavior associated with connecting the elements. For containers only #TargetFor:* is required. If, however, the created element is a binary association (typically an edge), the first (source) controller will implement the #SourceFor:* methods, and the second (target) controller the #TargetFor:* methods.

In addition, the platform automatically uses the result of #*canBe*For:* to display visual feedback on top of the visual element, such as green or red overlay if the element can or cannot be connected [Figure 8].

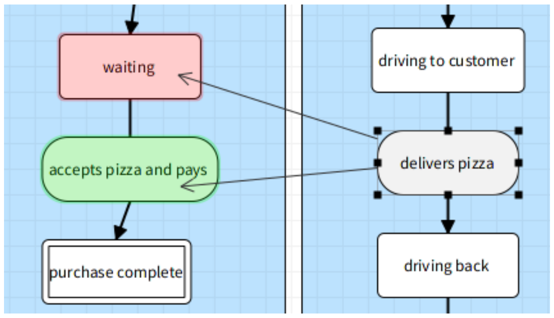
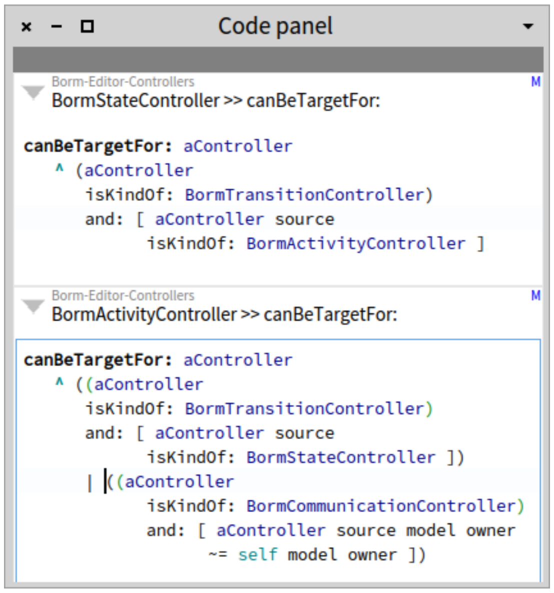


Figure 8. A communication can end in an activity (oval shape), but not in a state (rectangle shape) in BORM



Borm-Editor-Controllers
BormStateController >> canBeTargetFor:

```
canBeTargetFor: aController
    ^ (aController
      isKindOf: BormTransitionController)
      and: [ aController source
           isKindOf: BormActivityController ]
```

Borm-Editor-Controllers
BormActivityController >> canBeTargetFor:

```
canBeTargetFor: aController
    ^ ((aController
      isKindOf: BormTransitionController)
      and: [ aController source
           isKindOf: BormStateController ])
      | ((aController
           isKindOf: BormCommunicationController)
           and: [ aController source model owner
                ~= self model owner ])
```

Figure 9. Code describing live validation

Figure 9 shows that `StateController` can only be a target for a transition originating from an activity, whilst `ActivityController` can be also a target for a communication from an activity from a different participant (owner).

What may be unusual is that this validation does not fully rely on the model, instead the checks are made against the controllers. We have chosen this approach, as during more complex creation not all information may be available in the model, and the new model element is typically not yet connected with the present model, that is, sometimes we cannot decide whether an element can be connected only after it has been already connected. Thus complex meta-model structures can force the user to create only a partially valid model.

But the purpose of the live validation is not to always have fully valid model, instead it is a quick and cheap[4] to prevent common errors, similarly to a code editor warning a user of a missing semicolon or invalid syntax. For a full model validation, the user is free to implement a more powerful validation checker operating on the full model, as may be seen in Figure 10 for the case of an OntoUML validation editor[10].

4. Extensions and notations

We design the platform to be extensible and usable in different contexts and scenarios. To illustrate the wide possibilities, we present several quite different extensions that have been developed on top of the platform.

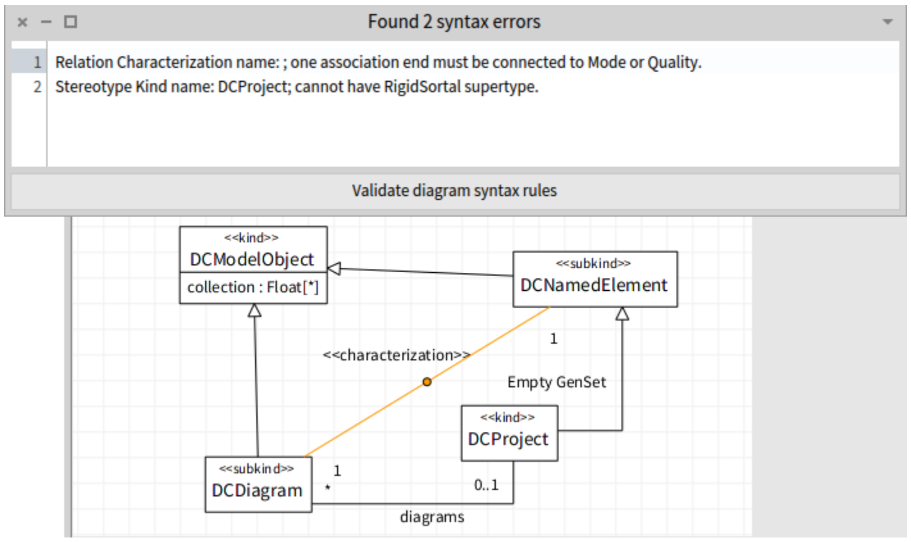
4.1 Model editing and live scripting

There are two principal ways in which a tool-user can modify meta-model instances. The first one is through the editors and tools provided by OpenPonk. This approach is common to majority of modeling tools (whether implemented in Java, C++, JavaScript, or Pharo). The popularity of this stems mainly from its ease-of-use, as the user is guided and hand-held by the tool's graphical interface, so the user does not accidentally corrupt the model. Moreover, the user does not need to be completely familiar with the actual meta-model implementation.

The second way is the ability to programatically manipulate the model, which offers interesting possibilities. Advanced modeling tools built in non-live environments (such as Java) address this by providing a special manipulation language that enables the user to query and manipulate the model, such as the Epsilon Object Language[5] for the Eclipse platform. There are, however, major downsides to this approach: implementing such a query language requires additional engineering effort on the part of the tool developer, while learning the query language requires additional effort on the part of the tool user. Furthermore, the language can be limiting in its capabilities and its usability, as without proper tooling support, aiding the user in creating, debugging, and working with the language may be limited.

We have mitigated this problem by choosing the Pharo live programming environment[2]. Pharo can be compared to a programming language, IDE, and an operating system rolled into one. In this platform, both the (tool-)user and the developer have direct access to objects in the system. This means that instead of using a specially crafted language, the user can directly use the Pharo language with all the

[4] Requires no extra effort from the tool-user.



Found 2 syntax errors

1 Relation Characterization name: ; one association end must be connected to Mode or Quality.
2 Stereotype Kind name: DCProject; cannot have RigidSortal supertype.

Figure 10. Full model validation of OntoUML diagrams

powerful tools and support available in Pharo for regular development.

Naturally, in some contexts, providing an alternative way and a specific manipulation language may be beneficial, however by choosing Pharo we already have a powerful language applicable for all meta-models without any extra engineering effort required.

4.2 Observing model changes with MetaLinks

As we noted earlier, to be able to directly manipulate the meta-model instances and to be able to immediately see the changes in the diagram, the meta-model code base has to offer an API through which changes can be observed. Doing so, however, requires modification of the code, which in some cases may either not be possible, or it may introduce unreasonable overhead (e.g. from performance point of view).

To address this issue, one can make use of MetaLinks [4]. MetaLinks are a recent addition to Pharo that allows installing additional behavior to methods[5] without modifying the code itself. Therefore one can define code extensions that would announce changes to the model, and those extensions would be then installed to the methods on demand, without affecting the original code base. As the MetaLink's interface is very low level, we have developed an open-source toolkit that would simplify the most common use cases, by the name of *MetaLinks-Toolkit*[6]. Albeit this toolkit is still in its early stage, one can utilize it to ease the MetaLinks integration. Figure 11 shows a code installing observations to a code base. After the installation, when the specified attributes have been modified, the objects fire announcement instances that can be observed. We have experimentally used this toolkit on our BORM meta-model, however our aim is to provide such augmentations for the FAMIX meta-model.

[5] To individual nodes of the method's abstract syntax tree.

[6] Available online: https://github.com/peteruhnak/metalinks-toolkit

```
observations := {
    MTObservationSet
        target: MTElement
        change: #(name owner uuid)
        add: #()
        remove: #().

    MTObservationSet
        target: MTContainer
        change: #()
        add: #(add:)
        remove: #(remove:)
}.
ci := MTMetaLinksChangeInstaller new.
ci install: observations.
element := MTElement new.
container := MTContainer new.

"Now we can use normal Announcement propagation"
element when: ValueChanged do: [ :oldValue :newValue | ... ].
container when: ValueAdded do: [:newValue | ... ].
container when: ValueRemoved do: [:newValue | ... ].
```

Figure 11. Installing MetaLinks-based observations

4.3 Simulating Finite-State Automata

In this section, we demonstrate an unanticipated integration of a finite-state automata simulator (FSA).

When an integration is anticipated, an API can be implemented that properly handles the needs and abstractions for the to-be-integrated tool. Handling all possible cases, however, requires many layers of abstraction, which not only have to be implemented on the platform-side, but, more importantly, every time a user wishes to create an extension. Thus, apart from solving the inherent complexity of the problem, the user is forced to address incidental complexity of the platform, as well.

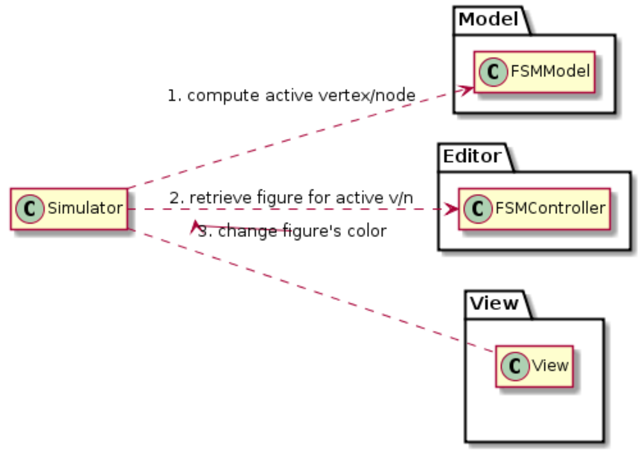


Figure 12. Simulator accessing different parts of the editor

The FSA simulator uses direct read-only access to the model and read/write access to the view. The model is used to compute the next step of the simulation, but no changes to the model are required. On the other hand, to visualize the progress through the simulation, the simulator directly modifies the view. It asks a diagram controller (responsible for managing the view) for an appropriate view element for the currently active node or edge and then changes the visualization (Figure 12) through the low-level Roassal API, or with the aid of existing OpenPonk-Roassal extensions.

In such an approach, both the FSA editor[7] and the OpenPonk platform are unaware of a third-party tool accessing their parts, therefore no explicit action is required of them (i.e. providing a special API).

Naturally, this approach requires the third-party tool to properly clean up after themselves, as the platform does not know what changes were made and what changes should be reverted.

Although such a third-party tool may inadvertently break the model or the view, we consider such risk acceptable, as this approach shortcuts otherwise complex implementation to bare essentials, which we find especially valuable for research and experimenting with and prototyping emerging ideas, which is one of the core aims OpenPonk. A more sophisticated and robust solution can always be introduced later.

4.4 DSLs for BORM and Class Diagrams

Although the tool-user creates a model primarily through the diagramming interface, there are other ways to create the model: importing from a different tool (via an exchange format such as XMI), transforming from other representation (reengineering source code), manually creating the required classes (via a programmable API), using a domain-specific language and possibly others. Choosing the best way is heavily context-dependent, therefore many tools will provide multiple options for the user.

[7] The plugin(s) implementing the FSA meta-model and the notation.

To address this need, we have introduced a domain-specific language for two of our meta-models: BORM[9] and UML Class Diagrams[14].

Even though both models have visual notations, some users may prefer to describe their model using text, and possibly only adjust the layout of the resulting visualization. This is not a new approach and many tools provide this functionality, such as PlantUML[18]. Unfortunately, they often stop at the production of the visualization and there is no real model behind it, which the user can manipulate, transform, validate, etc. Our aim is to provide a DSL for model creation, not just visualization creation.

For the implementation of our DSLs we have chosen PetitParser[20]. PetitParser enables an easy creation of composable grammars by describing the grammar directly with Smalltalk code in a regex-like syntax.

For the need of editing the model via a DSL, we have introduced a DSL editor to the OpenPonk core. This editor is directly connected with the currently opened Workbench Editor, and the user can modify the model through it; saving the text in the editor updates the model, and refreshing the editor produces the textual representation of the model, as seen in Figure 13. The DSL editor synergizes with the classical diagramming interface, as the user can arbitrarily switch between them. As describing a model with text lacks certain diagram information, most notably the layout position, we have introduced a set of layouting algorithms to OpenPonk[16] that are automatically applied to the diagram; thus the user always sees a readable diagram instead of a stack of overlapping elements.

5. Practical case study: UML Round-trip engineering for ABM platform CORMAS

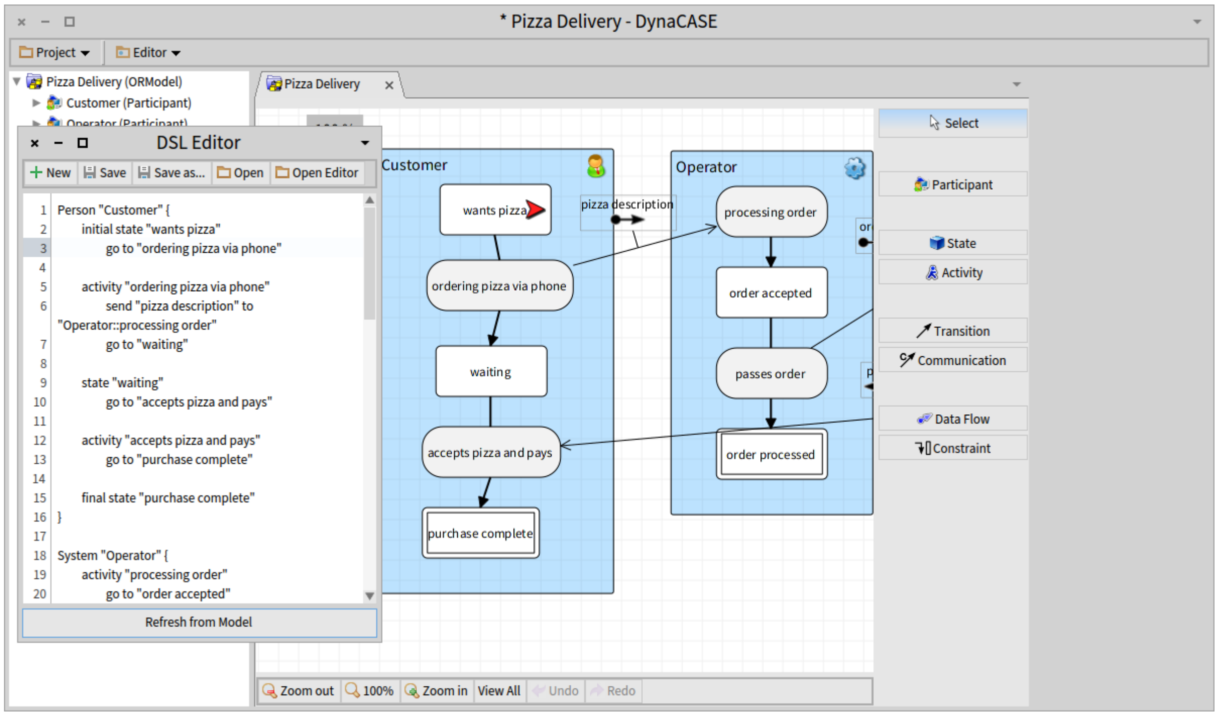
As our aim is to provide a platform for tool building, here we present a case study where OpenPonk was successfully used.

In collaboration with Cirad RU Green – a research group focused on addressing the needs of environmental research and sustainable agriculture – we have developed a UML Class Diagram Editor with round-trip engineering support for CORMAS agent-based modeling (ABM) platform[3] as a plugin for OpenPonk[17].

The users of CORMAS follow the participatory modeling (PM) approach in which the users (also known as stakeholders) collaborate on analyzing, creating, and implementing the domain model and the implementation solution.

For CORMAS, the stakeholders consist primarily of researchers and experts in the target domain; both, however, being seldom experienced programmers.

This presents a real challenge, as even though they are capable of creating the necessary models (the stakeholders pass a UML modeling course), they do not posses the suf-



```
Person "Customer" {
    initial state "wants pizza"
        go to "ordering pizza via phone"

    activity "ordering pizza via phone"
        send "pizza description" to
"Operator::processing order"
        go to "waiting"

    state "waiting"
        go to "accepts pizza and pays"

    activity "accepts pizza and pays"
        go to "purchase complete"

    final state "purchase complete"
}

System "Operator" {
    activity "processing order"
        go to "order accepted"
```

Figure 13. DSL editor on a BORM model

ficient programming knowledge and experience to correctly implement the model. For this reason, we have incorporated a support for round-trip engineering into our UML Class Diagram Editor. The objective is not to fully automate the process, but to aid with one of the challenging parts – creating the class structure from the model and vice versa.

The user can create the necessary model expressed in the UML notation with the diagram editor; for the needs of CORMAS we focus on a subset of UML notation typically used in class diagrams – classes, attributes, methods, associations, generalizations. The generator generates the necessary classes and methods in Pharo Smalltalk from the created model. Because the CORMAS platform is implemented in the VisualWorks Smalltalk (VW), we use the Smalltalk Interchange File Format[8] to exchange the source code between VisualWorks and Pharo, although direct code generation for VW has also been considered. For associations, we have chosen a straight-forward approach of providing accessors and add/remove methods[9], so that update on one side automatically updates the opposite sides. Other approaches have been researched [6]. These adhere more closely to the UML specifications, especially on maintaining visibility and multiplicity constraints of the associations; we found them needlessly complex in the context of the CORMAS platform, however for general-purpose forward engineering they may be suitable.

[8] SIF. Available online: https://github.com/peteruhnak/sif/blob/master/docs.md

[9] Add/remove methods are used for collection-based associations – association ends with multiplicities > 1

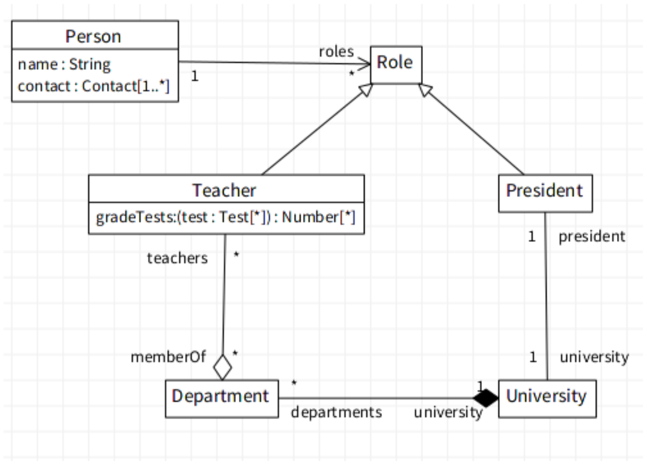
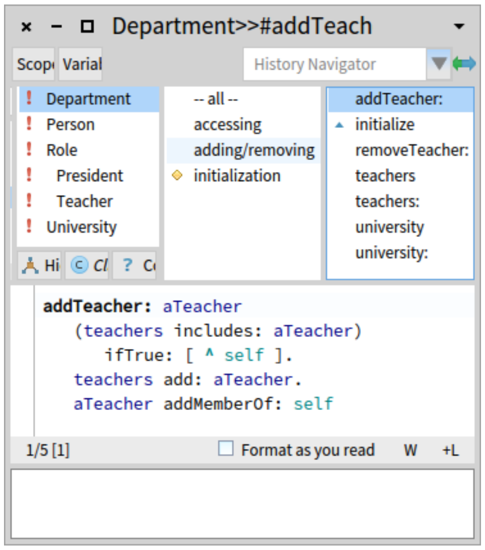


Figure 14. Example model

One of the important implemented features is the ability to reverse-engineer a model. In an ideal Model-Driven Development (MDD) scenario, only models are ever being manipulated and the code is always fully generated [11]. This, however, requires a very powerful modeling platform that is capable of describing every aspect of the software with models, which we currently do not provide. If there are modifications to the source code being made, the developers start to create discrepancies between the code and the model. It is possible to manually update the model each time the code has been made, that however poses the risk of complete abandonment of the original model if the discrepancies

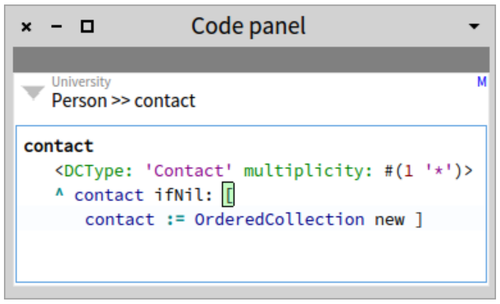


```
addTeacher: aTeacher
    (teachers includes: aTeacher)
        ifTrue: [ ^ self ].
    teachers add: aTeacher.
    aTeacher addMemberOf: self
```

Figure 15. Classes and methods generated from the model

pile up beyond a threshold. Instead, we allow the developer to regenerate the model directly from the code.

Providing a generic reverse-engineering support is a challenging task, especially in dynamic languages, as the real types and constraints are known only during runtime, and reverse engineering certain constructs – associations in particular – may not be possible at all. As the models for CORMAS are not generic and therefore we can constrain the problem, we have provided an alternative solution. During the source code generation we add additional information to the source code — namely we add pragma annotations to the generated methods. They add meta-information that would be otherwise lost, as there is conceptual gap between the model and the code. With such approach during the reverse-engineering, we can reuse this meta-information to aid with the regenerated model. By putting the meta-model information directly alongside the related code, we also lower the probability of introducing discrepancies, as a developer modifying the code will be more likely to also update the meta-information because it is already part of the modified code.



```
contact
    <DCType: 'Contact' multiplicity: #(1 '*')>
    ^ contact ifNil: [
        contact := OrderedCollection new ]
```

Figure 16. A generated method with additional meta-information stored in a pragma

6. Graphical user interface

The graphical user interface of the application is implemented using the Spec[22] framework.

Figure 17 shows the composition of GUI elements. The top-level window of the OpenPonk application is a *Workbench*. A workbench is always tied to a single project and has the responsibility of containing other parts of the GUI.

For each model in the project, an *Editor* can be opened within the workbench. The workbench organizes the editors in tabs, therefore several editors/tabs can be opened at once, although only one can be visible at a time.

The editor contains subwindows necessary for the display and manipulation of the model's notation – the *Canvas* (Roassal View) showing the visualization itself, a *Palette* providing a set of buttons for adding new items to the canvas, and an extensible bottom toolbar providing some manipulation buttons with easy access to the tool-user. Most of the GUI windows provide an API through which the user can manipulate them and describe their content.

Navigator: a tree-like view that visualizes the instances of meta-model elements in the project associated with the workbench; *Form*: a form editor for the currently selected meta-model instance capable of modifying its various properties; and finally a top toolbar that can be extended by plugins.

6.1 Connecting with plugins

Every window that is a part of the Workbench can be modified in some way by a plugin. Each window provides its own API appropriate to the situation.

The *Navigator* will ask the plugin's definition file about the structure of the model and how the model should be properly displayed, this consists primarily of specifying how to access the child nodes, element names, and icons as shown in Figure 18.

Both the *Workbench* menu (the toolbar in the top part of the Workbench), and *Editor* menu (underneath the canvas) can be extended. To extend any of the menus, a *pragma*[10] has to be added to a class-side method. OpenPonk will scan for methods containing the pragmas and call the methods when appropriate, as displaying the menu extensions can be restricted only show only if an editor of a particular plugin is selected. By choosing a pragma to implement this behavior, we have added extra flexibility as not only the plugin itself can customize the menus (the most common use case), but also any extra (or third-party) utility, without interfering with the plugin itself (e.g. a simulator window adding *"Open simulator"* button).

7. Related Work

Several related works exist; we split them into three groups and address each individually: (i) stand-alone tools and en-

[10] Pragma is a method annotation that can be located in the system via reflectivity.

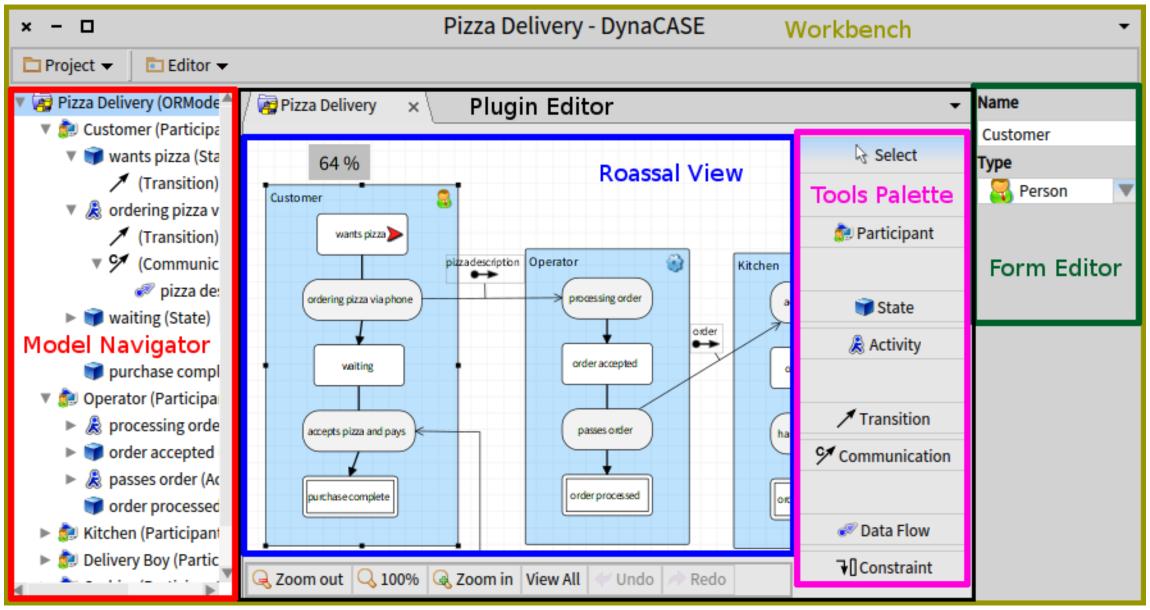
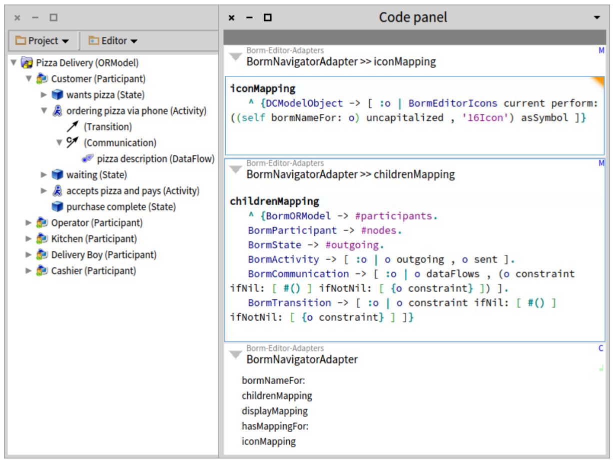


Figure 17. GUI overview



```
iconMapping
   ^ {DCModelObject -> [ :o | BormEditorIcons current perform:
((self bormNameFor: o) uncapitalized , '16Icon') asSymbol ]}
```

```
childrenMapping
   ^ {BormORModel -> #participants.
   BormParticipant -> #nodes.
   BormState -> #outgoing.
   BormActivity -> [ :o | o outgoing , o sent ].
   BormCommunication -> [ :o | o dataFlows , (o constraint
ifNil: [ #() ] ifNotNil: [ {o constraint} ]) ].
   BormTransition -> [ :o | o constraint ifNil: [ #() ]
ifNotNil: [ {o constraint} ] ]}
```

Figure 18. Specification of Navigator for BORM

vironments, (ii) Eclipse-based tools, and (iii) Eclipse itself. The first group contains stand-alone tools and environments that do not rely on any outside solution. MetaEdit+ and Enterprise Architect are typical representants of this group.

Enterprise Architect (EA)[21] offers an industry-grade MDA solution for UML and UML-based models (such as BPMN). EA provides support for the complete development life-cycle including requirements engineering, modeling, source code generation, round-trip engineering, testing, and more. While OpenPonk has been created with similar use-cases in mind, we provide it for free, as open-source, and platform independent (EA supports only Windows). By choosing Pharo live environment, OpenPonk can also offer significantly more interactive options.

MetaCASE[12] provides a solution for designing custom domain-specific languages (concepts, rules, notations, code generators) with its MetaEdit+ Workbench aimed at expert developers, and a MetaEdit+ Modeler aimed at model users. The same comments hold as for EA.

The second group of tools includes a wide spectrum of modeling tools such as Modelio, Papyrus, OpenCABE (our original BORM tool[15]). These tools are created on the Eclipse platform through its Graphical Modeling Project (GMP), which includes Graphical Modeling Framework (GMF) and Graphical Editing Framwork (GEF) [8]. Such tools usually focus on a single model family and they are limited in their meta-modeling abilities.

Finally, the Eclipse platform itself (alongside its GMF and GEF frameworks) provides a strong foundation for modeling and meta-modeling tools – that is not only the development of the models themselves, but also the creation of additional tools and extensions operating on the models. Our OpenCABE[15] tool used for BORM modeling has been developed in Eclipse, so we have a very strong experience in this area. Eclipse platform is a very complex artefact with a steep learning curve. However, this would not be critical if the big investment in the development pays off in the form of resulting flexibility and ease of enhancements. Unfortunately, in our experience, it is not the case. On the contrary, the bigger the platform code base, the harder is to extend it with additional models, visualizations, simulations and other algorithms. After 6 years of development, we were not able to give the platform to students to easily implement their conceptual modeling ideas. With OpenPonk, we already had several successful student projects. As for the reason of this situation, we blame the *incidental complexity* of the Eclipse platform. As a more in-depth analysis is out of scope of this paper, let us just put a hypothesis that this incidental complexity comes from the limited dynamic and reflective possibilities of the Java platform; on the other hand, live, dynamic possibilities of Pharo enabled us to significantly limit the code base to the *inherent complexity* of the problem.

8. Summary, conclusion and future work

OpenPonk project is the flagship of our research group, as we deal with various conceptual modeling notations, making models and performing algorithms on them. The project was initiated because of the lack of suitable free, open-source, simple, and extensible platform. The architecture of OpenPonk has been inspired by our extensive experience with the Eclipse platform. We took the best architectural ideas and stripped off the fat and gore by implementing it using the Pharo live programming environment.

The platform has been designed as highly modular – a minimal core extended by plugins and extensions. We have been also very keen about *separation of concerns* – model, view and controller are separated and various options for acquiring models are open: drawing diagrams, importing from an interchange file, reverse-engineering source code, DSL parsing, and other. The separation of concerns enables existing meta-models to be enhanced and used in OpenPonk without modifying their code.

OpenPonk stands for "dynamic": the meta-model and model development is highly interactive thanks to the live nature of Pharo. Querying and manipulating models on-the-fly is "for free".

OpenPonk is still in an early stage in many aspects. However, several quite diverse projects were successfully implemented using the platform, which demonstrate the possibilities. Several bachelor and master students were already able to acquire a working knowledge of OpenPonk and implement their ideas. This is very encouraging for us, as our ultimate goal is to offer a playground that will be loved by students, researchers and practitioners. With every new project, the platform matures and offers more for everybody. The adoption by community is very important in this point and it will direct the future development. The development of the core itself focuses on providing a stable minimalistic, yet powerful, foundation for tools building via plugins development.

As for the current endeavours, we cooperate with ForMetis Enterprise Engineers, a Dutch consulting and development company, to implement simulations and validations for enterprise engineering models and we are in a close contact with INRIA Lille Nord Europe and the University of Antwerp, who are interested in cooperation.

Acknowledgments

Currently, the development of OpenPonk is sponsored by ForMetis Consultants[11], our Dutch industrial partner. The development of the round-trip engineering support and partially the UML Class Editor for ABM CORMAS has been financed by RU Green CIRAD[12]. The development of the MetaLinks Toolkit has been sponsored by Synectique[13] and ESUG through it's mobility support program[14].

References

[1] Alexandre Bergel. *Agile Visualization*. 2016. URL `agilevisualization.com`.

[2] A. Bergel, D. Cassou, S. Ducasse, J. Laval, and J. Bergel. *Deep into Pharo*. Square Bracket, [S.l], 2013. ISBN 978-3-9523341-6-4.

[3] P. Bommel, N. Becu, C. Le Page, and F. Bousquet. Cormas, an Agent-Based simulation platform for coupling human decisions with computerized dynamics. 2015. https://agritrop.cirad.fr/576753/2/CormasforIsaga2015.pdf.

[4] M. Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, 2008.

[5] Dimitris Kolovos, Louis Rose, Antonio Garcia-Dominguez, and Richard Paige. *The Epsilon Book*, volume 20. 2016.

[6] Dominik Gessenharter. Implementing UML associations in Java: a slim code pattern for a complex modeling concept. In *Proceedings of the Workshop on Relationships and Associations in Object-Oriented Languages*, RAOOL '09, pages 17–24, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-549-9. doi: 10.1145/1562100.1562104. URL `http://doi.acm.org/10.1145/1562100.1562104`. 00008.

[7] S. Ducasse, N. Anquetil, M. U. Bhatti, A. C. Hora, J. Laval, and T. Girba. MSE and FAMIX 3.0: an interexchange format and source code model family. 2011. URL `https://hal.inria.fr/hal-00646884/`.

[8] Eclipse. *Graphical Modeling Project*. 2016. URL `https://eclipse.org/modeling/gmp/`.

[9] Martin Podloucký and Robert Pergl. Towards Formal Foundations for BORM ORD Validation and Simulation. pages 315–322. SCITEPRESS - Science and and Technology Publications, 2014. ISBN 978-989-758-027-7 978-989-758-028-4 978-989-758-029-1. doi: 10.5220/0004897603150322.

[10] Matúš Vološin. Vizualizace instancí OntoUML modelů. Diplomová práca. Praha: České vysoké účení technické v Praze, Fakulta informačních technologií, 2016.

[11] S. J. Mellor and M. J. Balcer. *Executable UML: a foundation for model-driven architecture*. Addison-Wesley, Boston ; San Francisco ; New York, 2002. ISBN 978-0-201-74804-8.

[12] MetaCase. MetaEdit+, 2016. URL `http://www.metacase.com/`.

[11] http://formetis.nl/

[12] http://ur-green.cirad.fr/

[13] http://synectique.eu/index.php

[14] http://esug.org/wiki/pier/Promotion/Mobility

[13] O. Nierstrasz, S. Ducasse, and T. Grba. The story of Moose: an agile reengineering environment. *ACM SIGSOFT Software Engineering Notes*, 30(5):1–10, 2005. URL `http://dl.acm.org/citation.cfm?id=1081707`.

[14] OMG. OMG Unified Modeling Language (UML) 2.5, Mar. 2015. URL `http://www.omg.org/spec/UML/2.5`.

[15] R. Pergl and J. Tůma. OpenCASE a tool for ontology-centred conceptual modelling. In *Advanced Information Systems Engineering Workshops*, pages 511–518. Springer, 2012.

[16] Peter Uhnák. Layouting of Diagrams in the DynaCASE Tool. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

[17] Peter Uhnák and Pierre Bommel. Facilitating the Design of ABM and the Code Generation to Promote Participatory Modelling. 2016.

[18] PlantUML. PlantUML Language Reference Guide, 2016.

[19] S. T. Pope and G. E. Krasner. A Cookbook for Using Model-View-Controller User Interface Pradigm in Smalltalk-80. 1988.

[20] L. Renggli, S. Ducasse, T. Grba, and O. Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, 2010. URL `https://hal.archives-ouvertes.fr/hal-00746253/`.

[21] Sparx Systems. Enterprise Architect, 2016. URL `http://www.sparxsystems.com/products/ea/index.html`.

[22] B. Van Ryseghem, S. Ducasse, and J. Fabry. Seamless composition and reuse of customizable user interfaces with Spec. *Science of Computer Programming*, 96:34–51, 2014.