CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF MASTER'S THESIS

Title:	Live Visualization of Epidemiological Models
Student:	Bc. Jan Blizničenko
Supervisor:	Ing. Robert Pergl, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2017/18

Instructions

1. Study the Kendrick DSL [1] and model and the OpenPonk platform [3].
2. Design a UI on top of Kendrick in order to simplify the process of building epidemiology models using techniques of live programming [2]. The behavior of a simulated epidemiology model and its visualization must be responsive to modifications of sub-models or model parameters.
3. Implement the UI in the OpenPonk platform.
4. Perform all relevant tests and demonstrate your work on a case study provided by UPMC/IRD (cooperating institution).
5. Document your solution.

References

[1] https://ummisco.github.io/kendrick/
[2] https://www.microsoft.com/en-us/research/project/live-programming/
[3] https://openponk.github.io

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague January 23, 2017

Czech Technical University in Prague

Faculty of Information Technology

Department of Software Engineering

Master's thesis

Live Visualization of Epidemiological Models

Bc. Jan Blizničenko

Supervisor: Ing. Robert Pergl, Ph.D.

June 29, 2017

Acknowledgements

My thanks belong to my supervisor Ing. Robert Pergl, Ph.D., to Dr. Serge Stinckwick and to Dr. Nick Papoulias for making it possible for me to create this interesting thesis. I thank also to Bc. Peter Uhnák, main developer of OpenPonk platform which proved to be great base for my implementation and to Pharo platform community for all those great tools.

My last thanks go to my family, my friends, my girlfriend and her two dogs for their support during making this thesis, as well as during my whole studies.

Declaration

In Prague on June 29, 2017 ....

Citation of this thesis

Blizničenko, Jan. *Live Visualization of Epidemiological Models.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstract

Working with modular models, where any model may consist of other models, would benefit from a way of visual modeling that provides the user with live feedback of results of changes made on any model or any of its modules. Unfortunately, epidemiology, used as a case study in this work, reveals that current tools do not provide a proper way to work with such modular models with live visualizations of results.

This thesis presents a solution by providing models with diagram-based notations that are transformed to and from models of external execution back-end systems.

The tool resulting from this case study is based on OpenPonk platform and is used for Kendrick epidemiological modeling system as the back-end.

Keywords Modeling, Modularity, Live programming, Epidemiology.

Abstrakt

Práce s modulárními modely, kde může jakýkoliv model sestávat z jiných modelů, by se usnadnila možností vizuálního modelování, při kterém by byly výsledky změn provedených na modelu okamžitě zobrazovány uživateli, a to ať už by se jednalo změny na samotném modelu nebo jakémkoli jeho modulu. Epidemiologie, která je v této práci použita pro případovou studii, však ukázala, že žádné současné nástroje dostatečnou možnost práce s takovými modulárními modely s živou vizualizací neposkytují.

Tato práce nabízí řešení pomocí vytváření modelů s notacemi založenými na diagramech, které jsou transformovány do a z modelů externích backendových systémů.

Nástroj, který vznikl jako případová studie, je založen na platformě OpenPonk a je využit pro epidemiologický modelovací systém Kendrick.

Klíčová slova Modelování, Modularita, Live programming, Epidemiologie.

Contents

Introduction **1**
Motivation and objectives 1
Problem statement 2

1 Analysis and related work **3**
1.1 Related work and tools 3
1.2 Architecture of OpenPonk 4
1.3 Analysis of Kendrick models and work-flow 8

2 Proposed solution **15**
2.1 Solution description 15
2.2 Workflow comparison of DSL and the tool 21

3 Detailed implementation **25**
3.1 Models and controllers 25
3.2 User interface 32
3.3 Model generating and bridge 35
3.4 Simulations and other kinds of output 40

4 Implementation for Kendrick **43**
4.1 Model-view-controller class structure 44
4.2 Project and plugins 48
4.3 Persistence and serialization 48
4.4 Bridge 49

5 Testing **51**
5.1 Unit tests 51
5.2 User scenarios 52

6 Discussion and future work **53**

6.1 Simulations 53
6.2 Linking 54
6.3 Highlighting changes 55
6.4 Optimization 55
6.5 Other notes 56

Conclusion **59**

Bibliography **61**

A Glossary **65**

B Acronyms **67**

C Contents of CD **69**

List of Figures

1.1 MVC architecture of OpenPonk 6
1.2 Structure of OpenPonk (formerly DynaCASE) model classes . . . 6
1.3 Screenshot of OpenPonk graphical user interface 8
1.4 Diagram of OpenPonk Graphical user interface 9
1.5 Kendrick classes and their variables important for the case study . 11
1.6 Visualization of simulation of “SEIRS” model with two strains . . 13

2.1 Screenshot of the tool with definition diagram on the left and resulting diagram on the right 16
2.2 Core of modularity extension of OpenPonk, along with usage for Kendrick 17
2.3 Example of live updating when a definition model is changed . . . 20
2.4 Example of an incomplete “SEIRS” concern’s definition model linked to its “SIR” concern’s resulting model 20
2.5 Screenshot of the tool visualizing Kendrick ’“SEIRS”-“Biological”’ model when modifying its “SEIRS” concern 23

3.1 Class diagram of traits for object models in modular diagramming 28
3.2 Class diagram of links and corresponding controllers 29
3.3 Object diagram of linking of Kendrick concerns with broken reference 30
3.4 Cropped screenshot of main GUI parts with labels 33
3.5 Class diagram of main structure of GUI parts 34
3.6 Linking buttons and dialog windows for adding models and creating links 36
3.7 Actions inside a bridge when a model object is changed, added or removed 38
3.8 Simulation of Kendrick “SIR” model with three species, while modifying “SIR” concern 41

4.1 Connection of model classes in package OpenPonk-Modularity-Kendrick 45

4.2 Detailed model and other classes in package OpenPonk-Modularity-Kendrick 46

Introduction

Motivation and objectives

Modular models are defined as models where a model can consist of other models or model parts [1]. This work generalizes such description in a way that any model or model part may consist of or be based on any other models or model parts.

If a model consists of other models or model parts, those models are further called modules of the model. Working with such modular models would benefit from a tool, which could provide a way of visual modeling with live feedback after any user-made modification of the model or any of its modules.

Visual live-modeling tools such as OpenPonk [2] (formerly DynaCASE) implemented on Pharo platform has no notion of modularity. Such limitation greatly reduces possibilities of live exploration of modular models that can be performed through visual notations. The user, which is usually a researcher or professional outside informatics, is limited to either monolithic modeling, or describing each modeling part with no connection to the other, thus no live visualization of results when any of such part is modified or replaced.

Motivation for this work originated with the epidemiological modeling system Kendrick [3], which is used here as a case study. Kendrick allows complex modular modeling, however, currently the creation of Kendrick's models is limited to textual form. This is done either by programming in an API for Pharo [4] implementation of Kendrick or by using a textual domain specific language (the Kendrick DSL). So far there is no support for visual modeling and notation, something that could potentially benefit to users.

Unlike traditional composition in object-oriented programming language (though traits or aspects) that usually produce a simple union of named entities from their parts, the model of Kendrick produces a set of named entities more complex than Cartesian product [5]. Because of that, visual feedback of the composition of "whole-parts" structure becomes even more important.

In case of the Kendrick DSL, resulting models can be lively edited and simulated. This adds another requirement regarding the interaction between live modular modeling as described above and the live simulation and other use of such resulting models.

This thesis is organized in multiple chapters. First chapter contains precise problem statement – fundamental questions needed to be answered. Next chapter shows analysis and discussion about related work and tools. After that, general solution is proposed, followed by description of implementation, extending the OpenPonk and using it for Kendrick as a case study. The thesis is concluded by discussion about possible drawbacks and future work.

Problem statement

Aim of this work is to provide an answer to following fundamental questions:

Question 1 How can a monolithic modeling platform (such as OpenPonk) be extended to support modular exploration?

Question 2 How can such a modular platform integrate with execution backends (such as Kendrick) to provide for e.g. live simulation feedback?

The solution should provide answers to these questions and to prove the solution in practice, a case study has to be made by implementing a prototype of a tool based on OpenPonk for Kendrick models.

Chapter 1

Analysis and related work

1.1 Related work and tools

There are multiple works and tools that are focused on subsets of the problem, like modular modeling, visual live programming, modeling of epidemiological models and similar, although there is no known solution and a tool that sufficiently combines all these requirements together and no works that clearly answer questions expressed above.

The epidemiological modeling system Kendrick [3] is implemented in Pharo smalltalk [4] and is main motivation and base for this case study. It supports advanced modular modeling, can display diagram with its states and transitions when inspected via Pharo tools, but has no visual notation that allows to modify those models and even the existing diagram provides too incomplete information to allow understanding it. To compare differences between two models, instances of both models need to be inspected and searched for differences. Comparing of simulation visualization is way simpler, because all it needs is to leave the previous graph opened, but even to open it, too many actions are required from the user – no such feedback is live.

There are also many examples of advanced diagramming tools. OpenPonk modeling platform [2], formerly "DynaCASE", is a diagramming tool which provides great means for extensibility by new kinds of models and functions. Its structure even allows reusability of its code and whole parts for derived projects. OpenPonk provides ways of modifying models using its GUI and to immediately visualize any changes to the model from any source, including direct model modifications using programming or DSL. However, it is monolithic in nature and does not provide a way to work with modular diagrams and models, where content of one model may rely on entities from another model, that might have been produced as a result of combining various other modules. Thanks to that extensibility and reusability, along with built-in support and usage of announcements of model changes, the tool created as the case study is derived from the OpenPonk platform.

Other diagramming tools, like Enterprise Architect [6] or Modelio [7], can provide only specific kinds of modularity. Enterprise Architect, and many other tools, provide a way to work with hierarchical models, where one model becomes an element in another model and can even provide simulations of such hierarchical models, but there are no means, or even use cases, to show any kinds of live results based on changes in sub-models – results such as displaying the whole model as it would be without hierarchy, or any live simulation results, especially if these results should be produced by an external back-end. Modelio supports modularity in terms of shared entities across multiple models and diagrams and, unlike Enterprise Architect, is open source, but other downsides are similar with Enterprise Architect and other such tools.

In epidemiology, the GLEAMviz [8] tool allows diagram based modeling and display of advanced simulations of global epidemic and mobility models. Despite the same domain with Kendrick, the model that GLEAMviz uses is very domain-specific and does not allow modular composition and therefore, not even the tool itself is capable of modular modeling. Furthermore, current implementation of the tool does not provide any kind of live feedback when models are changed.

From other fields of expertise, there is a ProMoT tool [9] for biology systems. The tool is made with visual modular modeling in mind, but its integration with external back-ends does not result in a live feedback modeling and editing cycle required in this work.

Regarding live feedback, there are many cases of solutions and prototypes for live programming like interactive visual programming language Scratch [10, 11] and research regarding live programming in Microsoft corporation [12, 13, 14], both textual and visual. These solutions are centered around an idea of a single playground. Such playground provides results of a small monolithic script or are very limited in possibilities and therefore these solutions fail to provide any means for live modular modeling needed for this work. There is an interesting recent exception in this category – a work in progress on visual language GP [15] based on Pharo [4] and Scratch [10, 11]. The GP language is being developed by ViewPoints Research Institute and it proposes a new modular system for general-purpose block languages.

Other notable works and tools for various systems and domains include a work regarding modular modeling of auditory processing [16], web application for modular reactor systems [17] and CPN Tools [18] for modeling and validating hierarchical Petri nets, however, even these lack some of requirements stated above.

1.2 Architecture of OpenPonk

An important part of this thesis is a case study – to implement a tool based on OpenPonk platform [2] and use it for Kendrick modeling system.

The OpenPonk platform is an extensible tool implemented on Pharo platform [4] that is based on smalltalk programming language. This is useful for the case study, since both OpenPonk and Kendrick are all implemented on the same platform and even share same dependencies like visualization engine Roassal [19, 20] used for drawing simulation results in Kendrick [3] and displaying diagrams in OpenPonk [2].

Pharo also proved to me to be very useful for prototyping and exploration of applications in early stages of development thanks to its dynamic nature and simple and open structure [21].

The OpenPonk has been created with extensibility in mind – extensibility either by new features or by new kinds of models and modeling notations. Such extensibility is needed trait for prototyping a tool based on OpenPonk, especially with requirement of usage with Kendrick that has no visual modeling notation so far.

There are many other extensible tools, but most of them offer the extensibility at the cost of features and in some cases, these tools are nothing more than vector-based drawing tools that enable defining sets of shapes for each type of diagram [22].

Despite all important features and possibilities of OpenPonk, it should be noted that OpenPonk itself is, in some regards, still in an early phase of development and lacks some options related to user experience present in many diagram modeling tools and even those drawing tools, like selection of multiple elements, keyboard key shortcuts, proper user interface for creating new projects and models, ability to remove models in user interface and some advanced features like shared entities. Thankfully, none of it does present a problem for this case study, since these options are either not crucial or can be added in the tool implemented as a case study.

Architecture of OpenPonk is based on MVC design pattern. Models of OpenPonk have entity for the model itself and separate ones for model elements that the model contains. This model structure is displayed in figure 1.2. Each model and model element – collectively called model objects – has its own controller, which separates these model elements from views, as shown in figure 1.1. These models are displayed as diagrams on a Roassal [19] view consisting of those figures. Modifications of models and model elements is usually done via GUI similar to other modeling tools, but OpenPonk also offers other kinds of model manipulation, like direct modifications using programming code, any extensions or even other tools. To provide visual feedback of model changes, controllers subscribe for required announcements. Whenever there is a change of model from any source, controllers are notified and Roassal figures and related GUI parts are updated to reflect all such changes. This way of announcing changes is very useful for live visual modeling.

To extend the OpenPonk for usage with new kinds of models and modeling notations, several classes have to be provided. Most important ones are related to the MVC architecture.

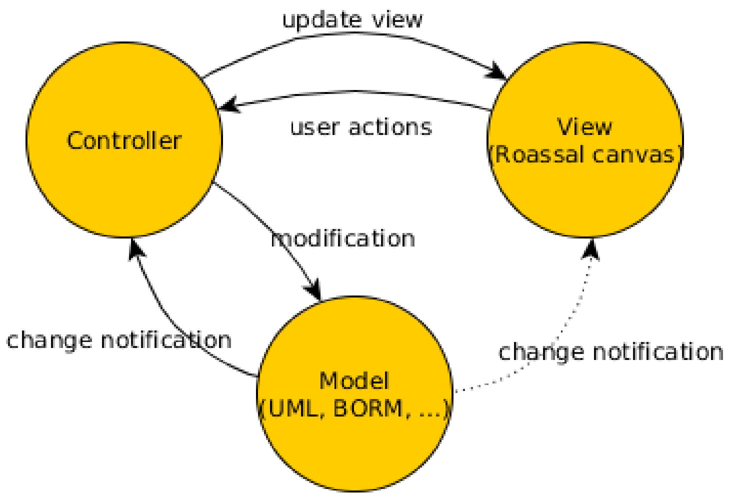


Figure 1.1: MVC architecture of OpenPonk

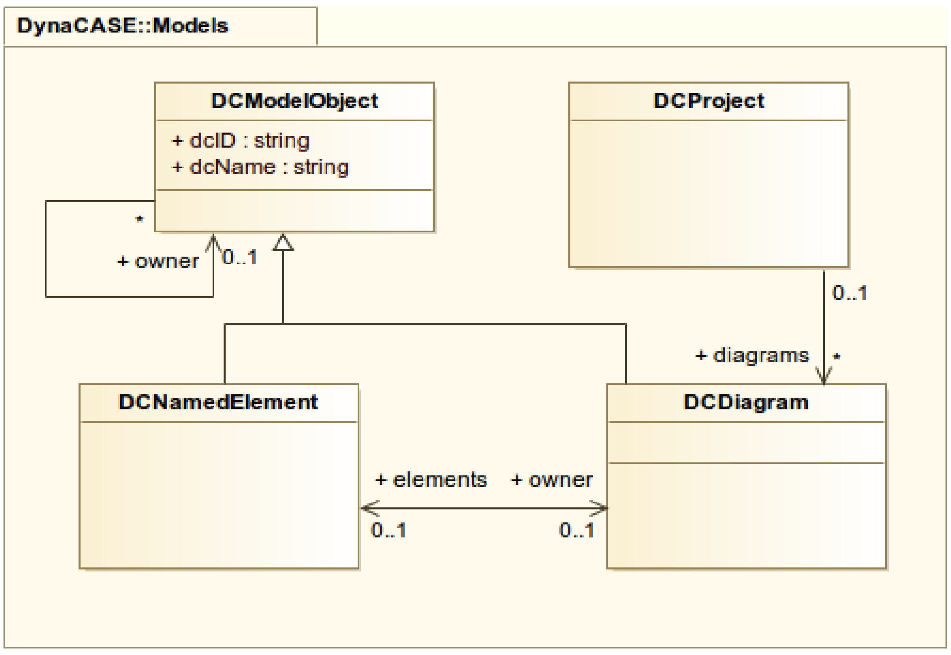


Figure 1.2: Structure of OpenPonk (formerly DynaCASE) model classes

Model Structure of classes for the model itself and all its elements, including relationships.

Controller Structure of classes for each model class that acts as its controller. There are diagram controllers and element controllers.

View Figures to display in a Roassal [19] view on a canvas, possibly with some other elements of the user interface.

A model needs a single class for the whole model itself and instance of such class contains all model elements. It also needs classes for each type of elements or relationships between those elements. Basic functionality for the model and elements is provided by an abstract class `OPModelObject`. For relationships, there is a subclass `OPEdge` which enhances it by having source and target.

There is usually a single controller for each subclass of model object. One for the model itself – subclass of `OPDiagramController` and one for each element – subclasses of `OPElementController` and `OPRelationshipController`.

All controllers have to answer class of their model object to know which controller should be created for instances of those model classes. Controllers need to implement methods related to model object creation. Whenever user tries to create a model object, controllers are asked whether their models can serve as a container for it or whether that model object can be connected to their models, namely whether the element can be a source or a target of the relationship that user wants to create. By default, they answer false. To change that, controller has to implement methods `canBeTaretFor:` and `canBeSourceFor:`, along with `addAsTargetFor:` and `addAsSourceFor:` for any other case than adding the element as one item to a simple container. Note that whole model is one such simple container.

Each element controller also provides a figure for its model object. These figures are Roassal [19] objects that will be displayed on Roassal canvas view.

OpenPonk also requires a plugin – subclass of `OPPlugin` that provides basic information about that kind of model, like name, class of the model, class of the diagram controller and, possibly, an icon for GUI.

To be able to save the project in a textual form and then load it back, classes related to serialization and persistence should be provided too.

User interface of basic OpenPonk is created using Spec [23] and consists of following basic parts as displayed in figures 1.3 and 1.4.

Navigator Column with a tree of all models and its elements.

Editor Each model has its own editor and it consists of other parts.

Canvas Main drawing area where figures are displayed.

Palette Column with buttons for creation of elements.

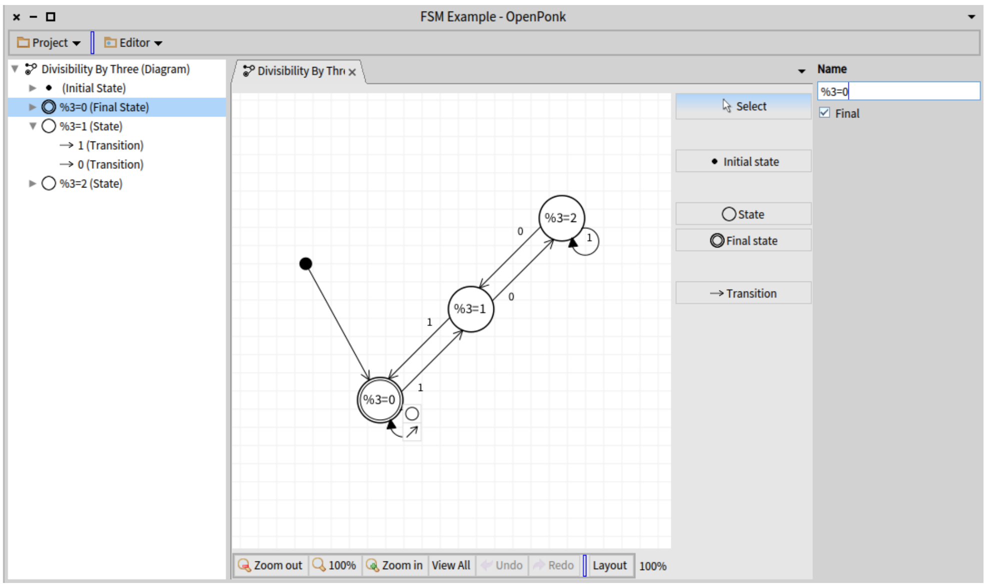


Figure 1.3: Screenshot of OpenPonk graphical user interface

Form Section with information and input controls related to currently selected model element.

These UI parts can be used for derived tools if needed. To modify them by subclassing, multiple references to their classes throughout the whole tool need to be replaced.

The navigator provides an interactive list of all models, elements, relationships and other model objects to quickly explore or even modify them and also to open models as diagrams in tabs.

Models, visualized as diagrams, are contained in a project (`OPProject`). If any new model is added to the project (note that this is not possible via GUI in current state of OpenPonk – only by programming code), it is not immediately opened as a diagram in new tab, but simply added to the navigator. If a user double clicks on its navigator entry, new editor is created for it and added as new tab.

1.3 Analysis of Kendrick models and work-flow

Kendrick models are of two kinds. Concerns are represented by `KEModelPart` class and models are represented by `KEModel` class. Concerns are not complete models that can be, for example, simulated. Instead, each concern provides one aspect of the model. Models, on the other hand, are complete and can be

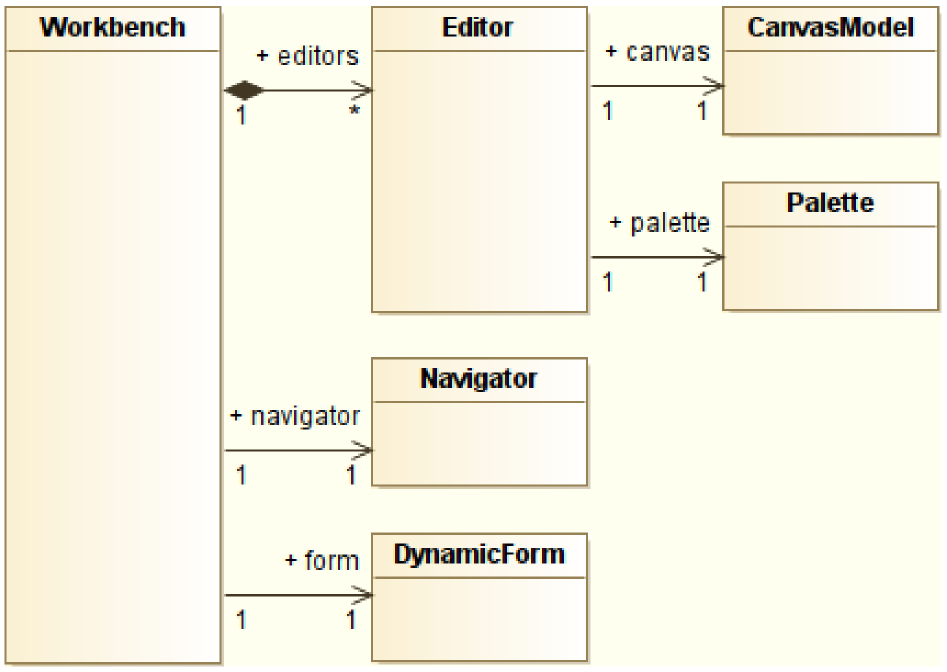


Figure 1.4: Diagram of OpenPonk Graphical user interface

simulated. Model can be defined directly or can be composed out of various concerns.

Before analyzing Kendrick any further, different kinds of modularity should be described first.

First one is based on hierarchical structure. In hierarchical models, whole model can serve as an element of another model. Examples are hierarchical Petri nets [24] and hierarchical state machines [25]. There are tools and solutions for working with hierarchical models [26], although none offers modeling with live feedback on modular models created by complex algorithms. This kind of modularity is not present in Kendrick, thus will not be part of the case study, therefore need for further work may be expected to support hierarchical models.

Second kind of modularity is not, in fact, modularity by traditional definitions [1], but extensibility. Models are extensible, if one model can be based on another one and provide various changes in comparison with it, without changing the base model. Extensibility of models is present in Kendrick in form of extending concerns by other concerns.

Last kind of modularity is based on models defined by collection of its modules and, possibly, additional parameters. Although this kind of modularity can be very simple if final models are mere sums or Cartesian products of its modules, there are more complex systems like Kendrick [5].

Kendrick provides its core model and concern classes with API, but also provides a DSL. The API allows the user to create concerns and models using Pharo programming. The user works directly with instances of these classes (`KEModel` and `KEModelPart`) and other ones like `KETransition` and `KEPopulation`. When concern is created this way, and extended by other concern, the same instance can be reused for different concern or model. All Kendrick classes and their variables important for the case study are shown in figure 1.5. Despite that it provides all possible functionality, many epidemiological researchers and experts do not know smalltalk or programming at all. For that reason, DSL is provided.

DSL is a textual way to define models and concerns with no requirement of knowledge of any programming language or class names. The Kendrick DSL system remembers all concerns and models by their names given by a user and these names are used to refer to these models for any future use. DSL code can be split into multiple files which do not need any connection, except of execution in specific order, unlike with instances of classes that need to be provided to following code.

In core Kendrick model classes, models and concerns have attributes, parameters, equations and transitions as instance variables.

Attributes are, for example, states or species. Attributes are represented as dictionaries, where types of attributes are keys and names are values. Key might be symbol `#state` or `#species` and values might be "S", "I", 'mosquito' etc.

Parameters are also dictionaries. Key is name of the parameter and value is a number or block of code.

Transitions are instances of `KETransition` which have a source, target and probability. Source and target is defined as dictionary, where keys are types of attributes and for each one there is exactly one value – name of the attribute. For example, source of transition might be a dictionary with key `#state` and value "S" and another key `#species` and value 'mosquito'. The probability might be a number or block that uses names of parameters.

Equations are strings containing differential equations of the mathematical model, but these equations can be transformed into transitions, they therefore do not appear to be necessary to support in early stages of development.

All model and concern classes provide an API to get, add or replace content of any of four variables described above by executing proper programming code.

Concerns (class `KEModelPart`) do not have any additional instance variables and functionality, but instead provide more API methods to delay a transition or split a status. Both of these modifications can be made even by sequences of other API calls, but delaying a transition and splitting a status is apparently used often enough that even the tool should provide such shortcut.

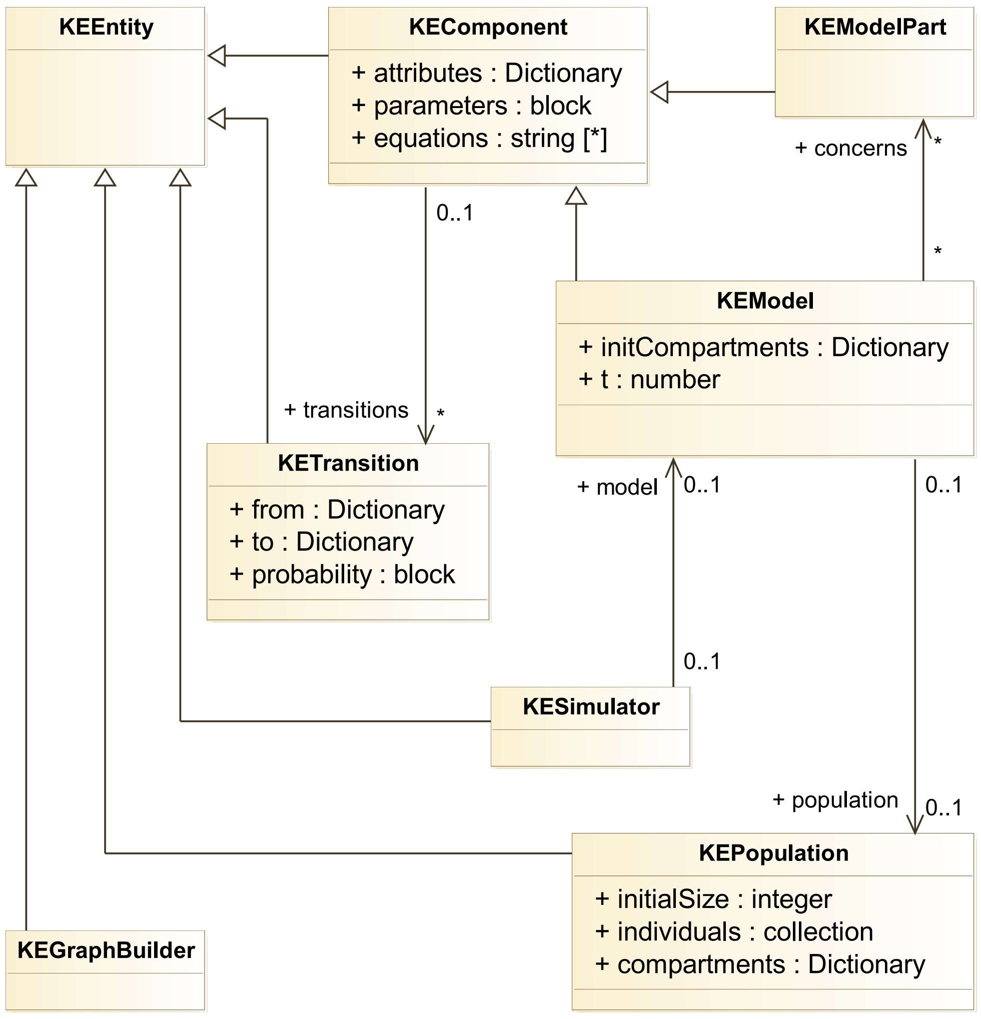


Figure 1.5: Kendrick classes and their variables important for the case study

To delay a transition, the source, target and probability of original transition has to be provided, along with new state and probability. If there are states “S” and “I” connected by transition with probability `lambda` and user wants to delay it with probability `gamma` and state “E”, following changes happen:

- State “E” is created,
- transition from “S” to “I” is removed,
- transition from “S” to “E” with probability `lambda` is created and
- transition from “E” to “I” with probability `gamma` is created.

To split a state, the name of state and collection of values has to be provided. These values can be names or indexes. Note that Interval class is also a collection, therefore interval from 1 to 1000 can be used instead of an explicit list. If there are states “S”, “I” and “R”, transition from “S” to “I” with probability `lambda` and from “I” to “R” with probability `gamma` and user wants to split the status “I” with indexes 1 and 2, following changes happen:

- States “I1” and “I2” are created,
- transition from “S” to “I” is removed,
- transition from “I” to “R” is removed,
- transitions from “S” to “I1” and to “I2” are created with probability `lambda` and
- transitions from “I1” and from “I2” to “R” are created with probability `gamma`.

Note that original state “I” is not removed, but since it does not have any transitions anymore, it does not affect the model.

In addition to variables in common superclass of models and concern, the `KEModel` class has three more important variables: `initCompartments`, `population` and `concerns`.

The `population` defines a population on which the epidemic is investigated. Variable `initCompartments` contains information about initial distribution of population and variable `concerns` contains a collection of integrated concerns, although the collection of concerns is not used in any important place in current implementation of the Kendrick.

To define the initial distribution of population, overall size of population is provided and then each compartment gets own amount of specimen. Compartment is one node created by Cartesian product of attributes. For example, if there are states “S”, “I” and “R” and biological species ’human’ and ’mosquito’, one compartment could be human at state “S”.

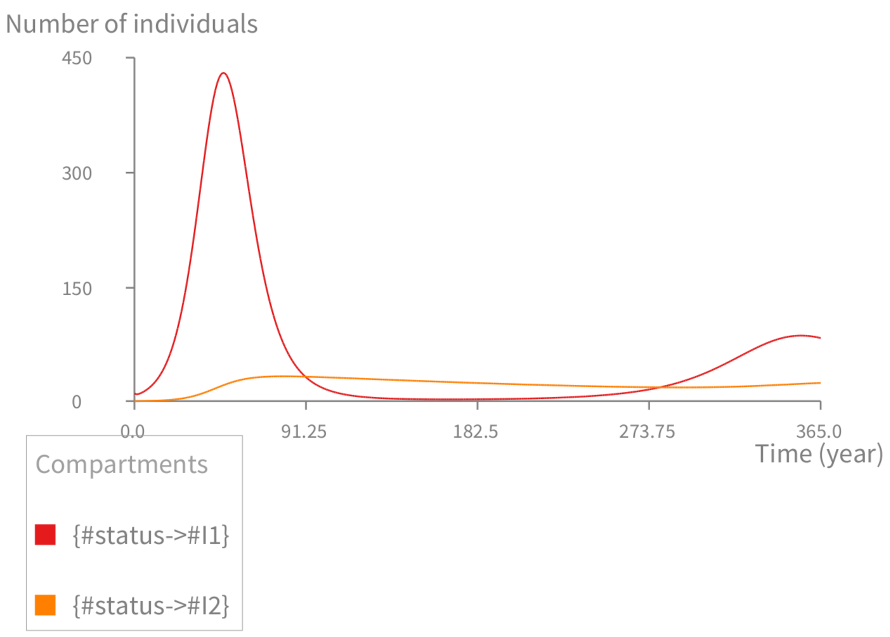


Figure 1.6: Visualization of simulation of “SEIRS” model with two strains

To simulate such model, simulator is executed on the model. Note that the simulation alters the object. To preserve the instance unchanged, deep copy has to be created. To the simulator, instance of `KESimulator`, user provides algorithm (or selects predefined one), parameters for the algorithm and the model.

For drawing results or other drawable outputs of the simulation, subclass of `KEGraphBuilder` receives required values from the simulator and can open Roassal [19] canvas with the visualization. Visualization of simulation of “SEIRS” model with two strains (with “I” status split into two) over time is shown in figure 1.6. Such visualization should be displayable even in the case study tool and there might even be a requirement to display results and other simulation data in textual or tabular form.

In comparison with this structure of classes and their API, downside of DSL for the case study tool might be lack of accessors of used classes, including inability to get any instance of `KEModel` or `KEModelPart`, and inability to preserve current state of all concerns and models without generating it all again, because a concern gets modified even by integration into a model or extending by other concern. This problem is, to some extent, present even in `KEModelPart` class, because it does not do a deep copy when extended, but thanks to its accessors, such problem can be bypassed. The greatest problem may lie in simulations, since simulation modifies the provided model, which is, however, not wanted if it needs to be reused. Thankfully, even that can be solved using accessors.

Although Kendrick does provide a way to preserve a model and model part in unchanged state even if used for other models and simulations, there may be various back-end systems that do not offer such possibility.

Chapter 2

Proposed solution

2.1 Solution description

One reason for live feedback requirement is a need to allow users to modify a definition of any modular model and to see what is the impact on the resulting model immediately after each change. Furthermore, user needs to see not only changes made in the topmost model itself, but also changes in any module or module of the module etc. Finally, the same applies also for replacing one module by another or adding new module to existing model.

To help a user get the live visual feedback for any made changes, graphical user interface of such modular modeling tool would consist of two separate views. A left view used for defining models and the right view for providing results. Both views use models displayed as diagrams. Such user interface has been implemented for the tool based on OpenPonk as shown in figure 2.1. These results and outputs are updated every time the user makes any kind of change, thus providing the live feedback.

The user can select any model on the left and any model on the right, thus can make changes to a concern and watch how it affects the resulting model, or can switch concerns that the model integrates and immediately see what is the impact on the model.

Although the solution focuses on diagrams, not all models are made with diagramming in mind. If the user works with an API or a DSL, there might not be a way to undo some changes in existing instances of the model without modifying and re-evaluating all the code again, thus creating completely new instance of a model. For the use case of live diagramming, more incremental updating is required. For example, in Kendrick, even during interactive sessions, once a concern (module) is integrated into a model, there is no way to undo the integration other than rebuilding the whole model. Furthermore, the integration may even make changes to the integrated concern instance, thus changing even the diagram of that concern.

One way to overcome such issues is to heavily modify such modeling system

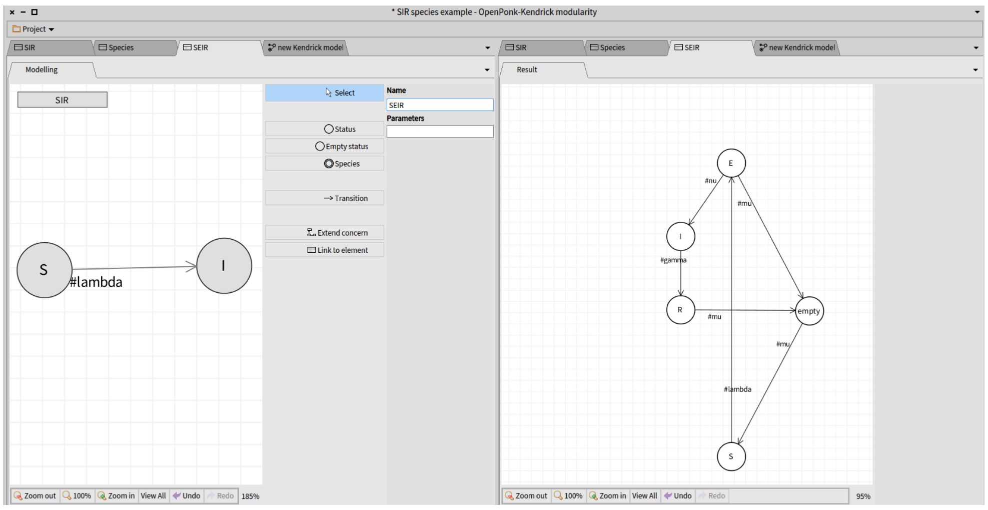


Figure 2.1: Screenshot of the tool with definition diagram on the left and resulting diagram on the right

and its DSL to accommodate for these needs of live diagramming. However, such extensive changes might not always be possible or preferable for all back-end systems. This solution proposes to create intermediate, diagram-friendly models, which are used by the diagramming tool for defining the model on the left side of the screen and displaying the composition results on the right side. Models created especially for defining models on the left side of a screen are called definition models and those created to display results on the right side of a screen are called resulting models. Single set of diagram-friendly models might be used for both defining and displaying results, but there might be a need for separate definition models and resulting models. In case of Kendrick, a single model suffices at the current state of implementation.

The case study tool is based on OpenPonk not only by inspiration, but also by directly subclassing and using parts of the existing OpenPonk infrastructure. That way proved that this solution can be applied for tools derived from already existing ones. OpenPonk classes are used as a base for most of aspects of the tool, specifically model elements, projects, controllers and user interface. Most of classes used for the solution are subclasses of OpenPonk core abstract classes and some of them even subclasses of OpenPonk finite state machine classes, especially those used for Kendrick. If needed, even completely custom classes may be used, if they provide similar functionality.

How such extension of OpenPonk has been made, along with usage for Kendrick, is shown in figure 2.2. Note here that the implementation includes additional controllers figures and Spec-related classes [23], which are described

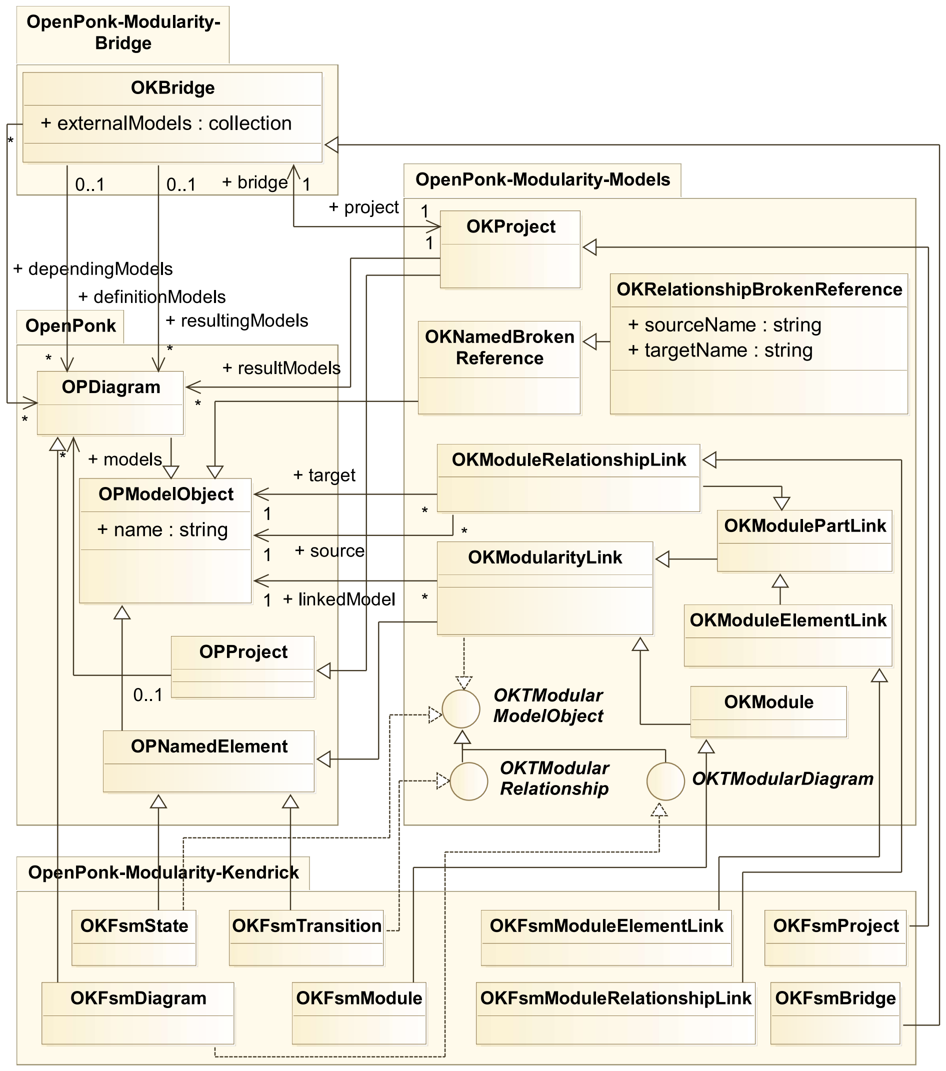


Figure 2.2: Core of modularity extension of OpenPonk, along with usage for Kendrick

in next chapter.

To connect definition and resulting models while extending and composing models, new kind of model elements have been created. These elements are called links in this work and can be used as parts of a model just as any other element. Links do not have any identity on their own, but provide a reference from the definition model to the resulting model or its elements. Such links are used only in definition diagrams and never appear in resulting ones. These links can have various semantics and impact on the model they are created in.

There are two kinds of links – module links, which reference resulting model itself as a whole and element links, which reference a single element (or a relationship) in one of linked modules. Adding a module link to the diagram provides an information that this model extends or is composed of another one and simply adding such link may change the resulting model. Element links, on the other hand, do not change any result on its own and the information about just adding such link to the diagram is not even forwarded to the external model in any way. All such links, however, can have attributes on its own. For example, Kendrick models have API for delaying transitions between states and when such transition is delayed, there is another state created and put between original ones and original transition is replaced by new ones. Element links do have necessary attributes for this delay, specifically name of new state and probability of new transition, which user can fill in. Also, to add a new state and create a transition from old state to new state, a user creates a link to the state from extended model, a new state in current model and a transition from the link to the new state. Although such semantics of links are used as default in the tool and are used for Kendrick in the exactly same way, meaning of adding a link to the definition model can be altered so that even adding a module link does not mean anything or adding an element link does. Also, by default, only one module link to a single resulting model should be created, but it needs to be altered for subclasses of systems with hierarchical models.

Whenever a user makes any change, the tool calls a bridge component, which handles communication with the execution back-end (for e.g. Kendrick). The bridge provides external models and other outputs from definition models and then provides resulting models from external models. External model is usually an instance of a class of the execution back-end, but can be provided by the back-end even in other forms like XML. One example of external model is an instance of `KEModel` or `KEModelPart`. Such external models cannot be directly modified by a user or the tool itself – only by the bridge. Simulations and other outputs provided by the back-end are saved in the resulting model and then used by user interface.

Bridge is called after a change in definition models that are displayed as diagrams on the left side of GUI and can be changed from different sources (including a user). Resulting models are displayed on the right side of GUI and cannot be changed, only replaced by the project when bridge creates new ones.

To explain it further, let's use Kendrick concerns as an example. If the user makes a change in a definition diagram for Kendrick "SIR" concern (model part), the bridge creates an instance of the Kendrick model part with all the information from the diagram. Then it goes on to create the resulting diagram from this Kendrick model. If there are any links to original "SIR" resulting model, it means that there are dependent models or model parts. Since new resulting diagram has been generated, the old one is replaced by this new one in the project and in user interface. If there are no depending models, updating requires no further action of the project or views. However, if there are any depending definition models, their links now point to the old, already replaced model and its elements. In this case, if there is for example a “SEIRS” concern defined as extension of “SIR”, all links in “SEIRS” definition model need to be updated to point to this new model. The bridge takes the new resulting model with same name as the model to which the module link is pointing to now and switches these models in the link. Right after that, element links in “SEIRS” definition model are updated in a similar way to point to elements of this new resulting “SIR” diagram by searching for elements with same names as previous ones. Since updating links is in fact change of the “SEIRS” model, its resulting model needs to be updated too, so after updating links, the whole process is repeated for this depending model (“SEIRS” model in this example). This is all recursively repeated for all depending models, so if any concern or model depends on this “SEIRS” model, its links get updated and new resulting model is generated too.

In the case of relationships without names, those are linked by names of source and target elements. Similarly, any other kind of attribute or structure might be used for such identification, but such alternatives have not been fully explored during designing this solution and therefore are not even directly supported by current state of the implementation.

As a last step, the project asks the bridge for resulting diagrams that were removed or created and replaces old ones by new ones in the user interface. In the example of “SIR” and “SEIRS” concerns, original tabs with their resulting diagrams are replaced by new ones. A setback of this way of updating is that the tool does not remember which element has been selected (clicked-on) by the user, nor even changes to their layout on the canvas done by dragging them to new positions. Most important steps of this process are shown in figure 2.3. Note that this figure does not show that the whole process repeats for the “SEIRS” definition diagram.

While updating links to new model, there may not be any corresponding element, or even whole model, to link to. The model or element could have been removed or renamed. In that case, bridge points these links to broken reference objects, causing no resulting model to be generated in the bridge until the model object required by the link reappears or after a user fixes the problem by providing name of new model object to link to. Links pointing to broken references are further called invalid and model object are called validly linkable to if a link is valid if it points to that object. Example of links with invalid references is shown in figure 2.4.

If a resulting model is not generated, it is instantly removed from the right side of the user interface. When all problems get fixed and resulting model is generated again, it reappears on its original position.

Links are considered invalid in a few more cases.

First one occurs when element it links to does not have a name. Such

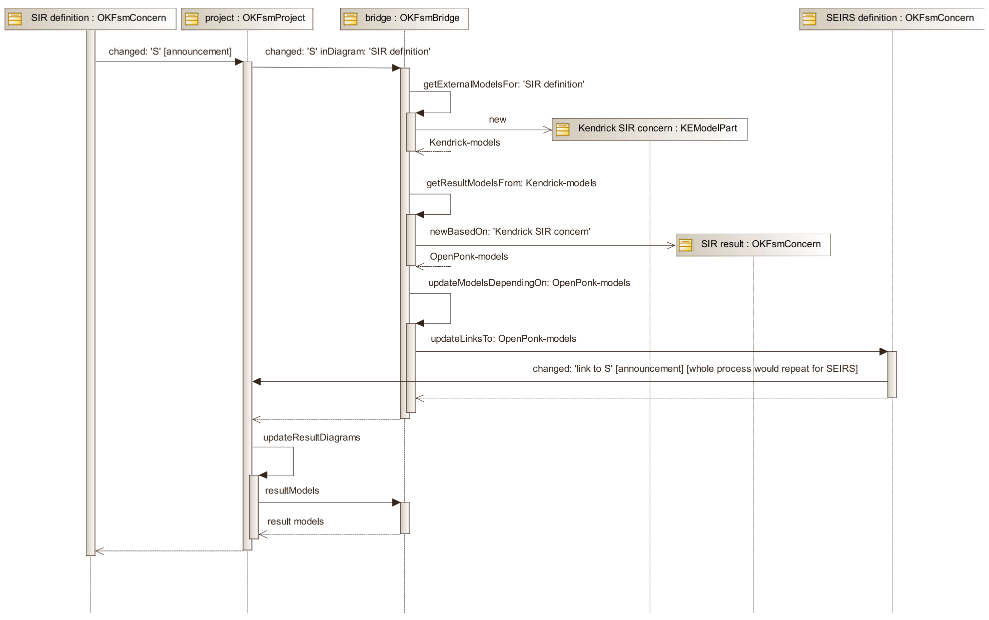

SIR definition : OKFsmConcern
project : OKFsmProject
bridge : OKFsmBridge
SEIRS definition : OKFsmConcern
changed: 'S' [announcement]
changed: 'S' inDiagram: 'SIR definition'
getExternalModelsFor: 'SIR definition'
new
Kendrick SIR concern : KEModelPart
Kendrick-models
getResultModelsFrom: Kendrick-models
newBasedOn: 'Kendrick SIR concern'
SIR result : OKFsmConcern
OpenPonk-models
updateModelsDependingOn: OpenPonk-models
updateLinksTo: OpenPonk-models
changed: 'link to S' [announcement] [whole process would repeat for SEIRS]
updateResultDiagrams
resultModels
result models

Figure 2.3: Example of live updating when a definition model is changed

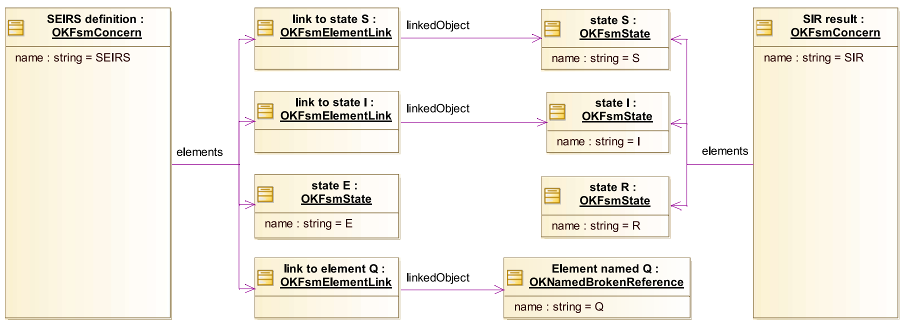


Figure 2.4: Example of an incomplete “SEIRS” concern’s definition model linked to its “SIR” concern’s resulting model

element is not considered as a valid object to link to, because it might not be findable after generating new resulting models. In some kinds of models, such elements might not be valid even in the external back-end model and for that reason, do not even become part of resulting models.

Another one occurs if the link depends on another link on the same definition model, but that link gets invalid. There can be four main reasons for that:

1. Link to relationship is invalid if source or target element cannot be validly linked to (does not have a name in case of original elements or is invalid in case of links)

2. Link to any object is invalid if link to its whole model is invalid. For example, if there is a link to “SIR” model and link to state “S” from the “SIR” model and then the link to “SIR” becomes invalid, even link to state “S” becomes invalid.

3. Link to any object is invalid if link to its container is invalid. This is, however, generalization of second one, because even whole model (diagram) itself is specific case of container.

4. If user manually reconnects the link (in the case study tool it can be done by writing different name), the link gets immediately linked to a broken reference object with that name specified by user no matter if any element with such name exists, because it is not role of the model to search in other models for such linkable elements, or even search in project for other models. Renamed link notifies project (possibly by announcement), which calls a bridge to update links. Remember here that when links are updated, it is considered a change and whole resulting model generation process begins. This is the only case where links are updated before any creation of resulting models happens, along with the moment when new definition model is added to the project.

While bridge updates links, it, therefore, does not just relink current valid links to model objects from new resulting models that replaced old ones, but also relinks invalid links to new resulting models if the proper model object starts to exist and it also, as stated above, may relink valid links to broken references, thus making the link invalid.

2.2 Workflow comparison of DSL and the tool

To show differences of workflow using the API or DSL of Kendrick and the workflow with the diagramming tool, let’s use an example with Kendrick concerns and models.

2.2.1 Usage of Kendrick DSL

The Kendrick DSL code for creating a simple “SIR” concern, a “Biological” concern involving both human and bird population, a “SEIRS” concern extending the “SIR” concern, and finally a model with both the “SEIRS” and multi-species “Biological” concerns follows. Work with an API would be, to some extent, similar in nature.

```
KendrickModel SIR
Attribute: #(status -> S I R);
parameters: #(beta~lambda~gamma~mu);
Transitions:
#(S -- lambda~--> I #'.'
I -- gamma~--> R #'.'
status -- mu --> Empty #'.'
Empty -- mu --> S #'.').
KendrickModel Biological attribute:
#(species -> human~bird).
KendrickModel SEIRS
extends: 'SIR';
parameters: #(sigma~nu);
delay: #(sigma~, S -- lambda~--> I , E);
AddTransition: #(R -- nu --> S #'.');
AddTransition: #(E -- mu --> Empty #'.').
Composition SEIRSInfluenza
model: 'Influenza';
concern: 'Biological';
concern: 'SEIRS';
```

After this code has been executed and the user decides to switch from “SEIRS” concern to the original “SIR” concern, there is no way to do this incrementally. The above code needs to be edited and executed again to produce the new model. If the user wants to compare even basic structural changes in the previous and new model, the user needs to inspect both instances of Pharo objects. The same steps as above are needed for a case when a user decides to change the biological concern by adding a third specie, like a mosquito. Furthermore, by extending “SIR” by “SEIRS”, even the original “SIR” instance is irreversibly changed, so there is no way to reuse an existing instance for separate extensions. Whole code has to be executed again in all cases. Also, in DSL case, cached entities need to be reset.

2.2.2 Usage of the tool

In the Kendrick case study based on OpenPonk, a user would define these concerns and models using diagrams on the left side of the tool.

For each concern and model, a new definition model will be created on the left side of the user interface. User should start by creation of empty “SIR” definition model and adding its states and transitions. On the right side

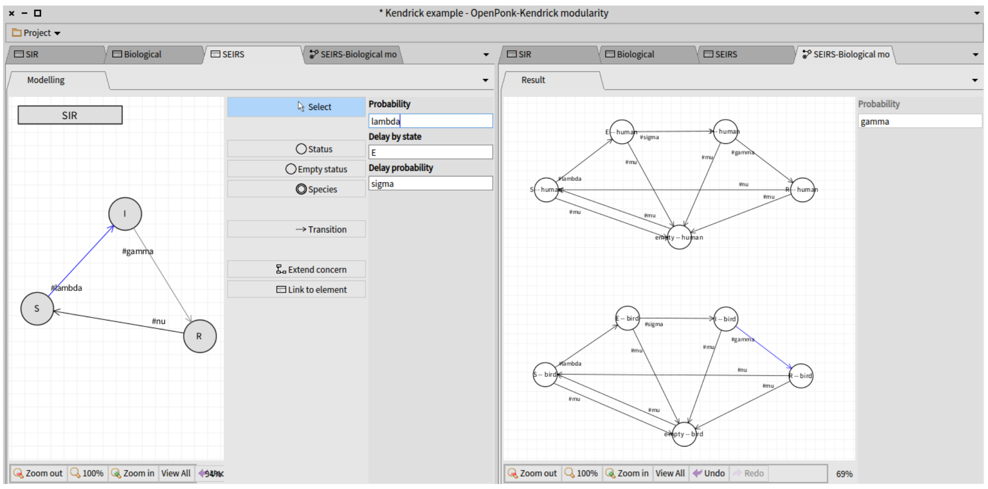


Figure 2.5: Screenshot of the tool visualizing Kendrick '"SEIRS"-"Biological"' model when modifying its "SEIRS" concern

of the UI, the user could see that the resulting "SIR" model looks same as definition one during every step of his or her work. As a second step, the user will create the "Biological" concern in similar way as the "SIR" one.

For "SEIRS", the user created new concern once again. Its resulting model is, as always after creation, empty. Then the user would not add any states directly, but instead adds a module link to the resulting model of "SIR", which causes that the "SEIRS" resulting model immediately looks the same as "SIR", because the model now extends it does not apply any additional changes. The user continues by creating a link to transition from "S" to "I" and link to state "R". Unlike module links, these element links do not add any new information by their creation itself and are there just to help with defining other changes. Finally, the user selects the link to the transition and defines that the transition should be delayed. Information about the delay is therefore part of the link object. The "SEIRS" model is completed by creating a transition from "R" link to "S" link.

As a last step, to create a final model with both concerns, user creates an empty model and adds links to both of those concerns.

If the user was modifying "SEIRS" concern, while watching whole resulting model, the user interface of the tool implemented for the case study would look like in figure 2.5.

To switch from "SEIRS" concern to "SIR" concern in a final model, it would take just a few clicks or keystrokes and the resulting model immediately changes with the proper concern.

Similarly, to add a specie to the "Biological" concern, the user just adds the specie element to the “Biological” definition model and the resulting model conforms to this change automatically and displays the proper diagram.

Chapter 3

Detailed implementation

The implementation can be divided into three sections, although all are connected together.

First one is extending the OpenPonk models and controllers for use with the solution for modularity.

Second part focuses on user interface, which is mostly created completely from the scratch, but main canvas and its controls are provided by subclasses of OpenPonk Spec [23] classes.

Last section describes how the OpenPonk-based tool has been used to implement Kendrick support as the case study.

Before that, little note about Pharo class naming is required. Pharo does not have namespaces. To overcome that, users of Pharo (developers programming in Pharo) usually use prefixes of class names [27]. Kendrick models use "KE" (like in `KEModelPart` class), OpenPonk uses "OP" (like in `OPProject` class) and for this OpenPonk-based tool, "OK" prefix is used as a reference to this case study – OpenPonk with Kendrick, although most of the code is made even for different models than Kendrick ones. These prefixes are usually followed by another derived and dependent packages, like in `OPFsm` package that provides finite state machines for OpenPonk. Similarly, since Kendrick definition and resulting models are based on the `OPFsm`, their names start by "OKFsm".

3.1 Models and controllers

Extending of models is always done hands-in-hands with controllers. There were three main goals for models and controllers.

1. Provide modified model objects for usage with modularity.
2. Create linking-related classes.
3. Create project class which can accommodate resulting models and overall modular work-flow.

3.1.1 General model objects

Usage for such modular tool has greater requirements on models and elements (generally called model objects). To comply with these requirements, traits with such functionality have been created.

First of all, each model object has to answer messages related to links, like whether the model object can be linked to and whether such link would be valid (i.e. not broken). Models should also provide information that they are original model objects, not just links. As an optimization, models and elements used for live updating have modified announcing to reduce multiple kinds of announcements after single user action, like for example adding an element to the diagram leading to at least two announcements – first one informing that element has been added and second one informing that the diagram has changed (because it contains new element). It would cause duplicate process of generating resulting models, so in these cases, one of those announcements is modified to show that it is duplicate one. The project then may ignore the duplicate one, or just remember it until next non-duplicate change happens.

To simplify linking, these models also answer what elements they depend on and, vice versa, what elements depend on this one. The reason for that is to automatically add links to all elements that have all dependencies already linked and to automatically add links to all elements the linked one depends on. Without that, link to transition could be added, but not links to source and target states of that transition.

Changes described in paragraphs above are there for all model objects, so for both whole models (diagrams) and for all elements (including relationships). There are, however, specific needs for diagrams and relationships.

Models can be linked to only if can be exactly identified. In case of most elements and diagrams, it means that they need to have a name. For relationships, the situation is different. Relationships are, in current implementation, linked by names of source and target elements, so the source and target elements need to have a name and relationships have to answer it correctly.

Diagrams do not exactly require many specific changes from general model objects, but some useful accessors are provided anyway, like getters for all links, all module links etc.

On controllers side, only diagram requires changes related to linking. Such diagram controllers have to reply controller class for given model class, but to simplify linking, those diagram controllers answer also link class for given model object. Diagram controllers also subscribe for new kind of announcement announced when links are updated. Diagram controllers also provide collection of model objects that can be linked to, using information from the project and model (only those existing model object that are not yet linked to may be linked to). Few changes are also needed to work with different GUI structure.

Since these modular model object do not have any new variables, just additional interface and functionality, all required changes described above are integrated into traits. These traits are meant to be used with model objects that are already usable for OpenPonk, but need these traits to be used even with this derived tool. Most important parts of these traits are shown in figure 3.1.

3.1.2 Projects

Special kind of model is a project. OpenPonk uses instances of class `OPProject` that has no subclasses. However, this tool requires that there are different kinds of project with different bridges and different possible kinds of models. For that, `OKProject` is provided as a subclass of `OPProject` and most importantly, this `OKProject` requires to have a subclass for any kind of project – like there is `OKFsmProject` for Kendrick projects. Such subclasses answer collection of enabled classes of models which can be created. To comply with this new requirement, a few persistence-related classes had to be made. First one is a `OKProjectDirectoryPersistence` (a subclass of `OPProjectDirectoryPersistence`), which is able to save information about type of project and second one is `OKPersistenceProjectReader`, which is able to read such information and create proper kind of project.

The `OKProject` also has additional functionality related to resulting models and bridge. First of all, the project now has not only collection of models, but also result models. It also points to its bridge it calls whenever there is a change in any model or element. Even the `OPProject` subscribes for some announcements from its models and elements, but `OKProject` subscribes for all announcements that may require updating links or calling a bridge.

To inform the user interface of updated resulting diagrams, the `OKProject` is announcing it. Whenever bridge provides new resulting models, `OKProject` sends announcement `OKProjectContentReplaced` to all subscribers. It is used in user interface to update displayed resulting diagrams on the right side of the screen.

3.1.3 Links and linking

Not only traditional elements will be parts of modular diagrams. New kind of elements are called links. Even for them, classes for models and controller had to be made, as well as handling of broken references had to be solved. Broken references are caused by removing models to which links point to.

Modularity links are elements, elements that need to be part of modular diagrams. To comply with these requirement, links are subclasses of general `OPNamedElement` class and use `OKTModularModelObject` trait. With that, they all inherit properties like name and owner and even more importantly, implementation of announcements like any other OpenPonk model element.

There are already prepared link classes for linking modules (whole diagrams), elements and relationships. These predefined ones can be used directly,

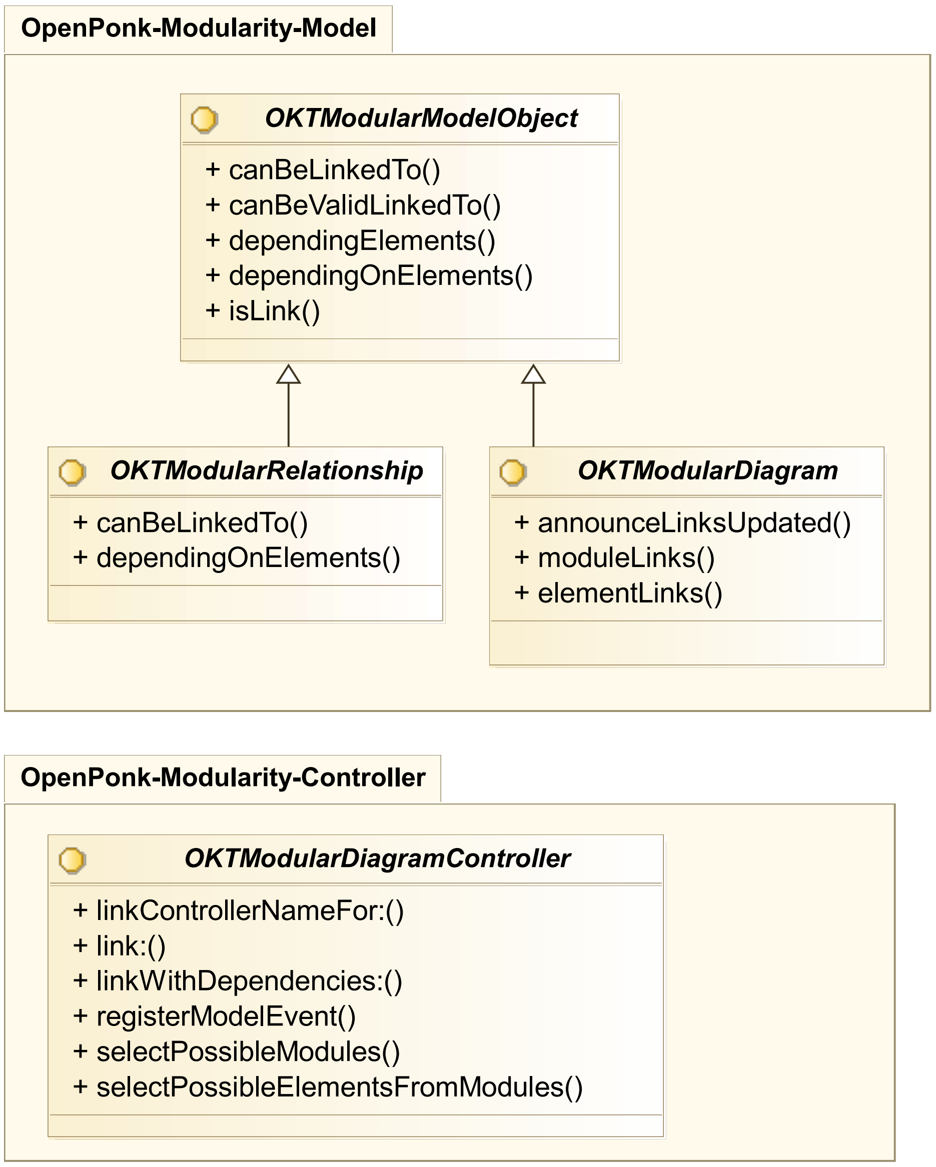


Figure 3.1: Class diagram of traits for object models in modular diagramming

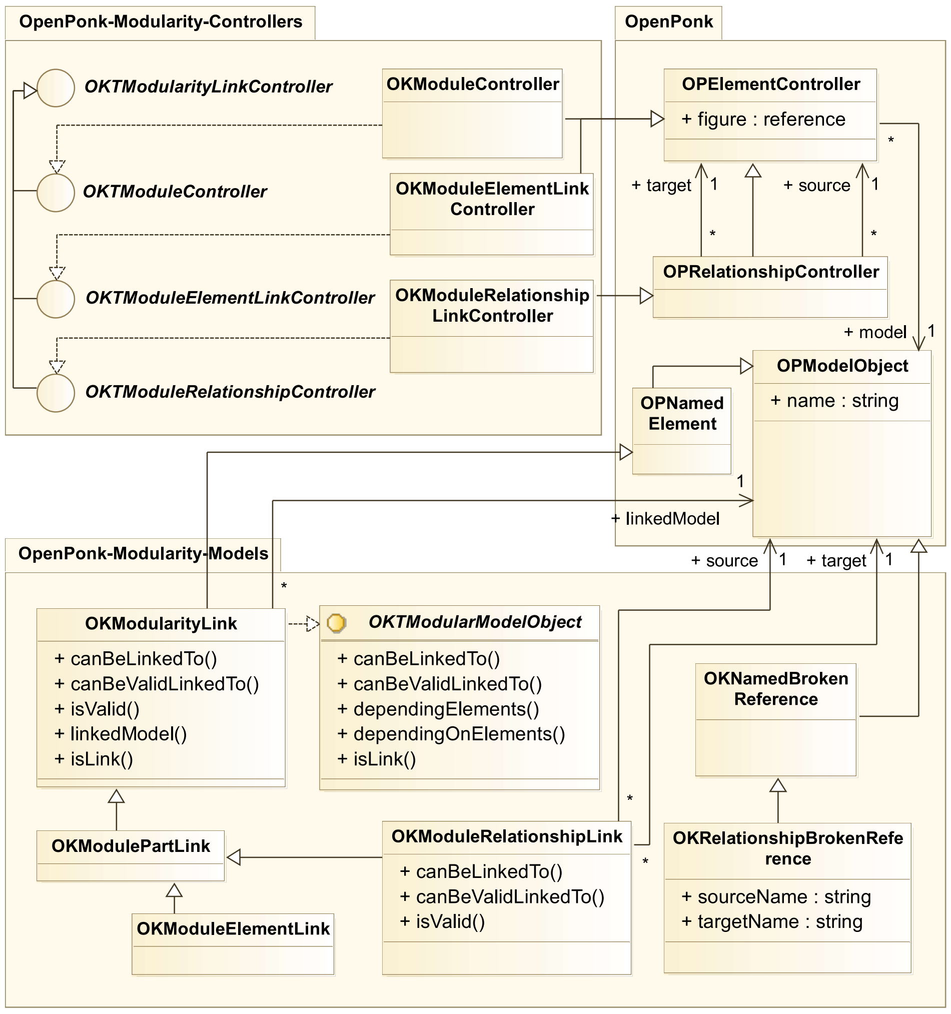


Figure 3.2: Class diagram of links and corresponding controllers

but are also meant to be subclassed for specific needs, like it is done for Kendrick.

For each model element, there has to be controller, so link controllers are provided too and can be subclassed as well. In a case that user does not want to subclass these controllers, traits with same functionality are provided. In fact, those controller classes are just using those traits and have no additional functionality.

To better understand structure of these links and controllers, class diagram is provided as figure 3.2.

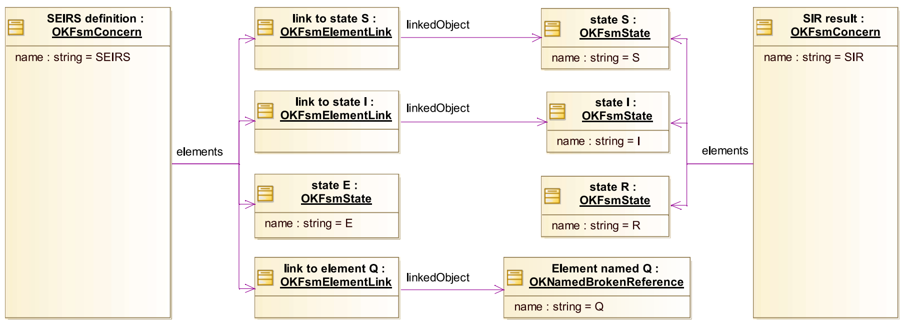


Figure 3.3: Object diagram of linking of Kendrick concerns with broken reference

Each link has a linkedModel, which points to the model object from a resulting diagram. If the object is no longer available, the bridge points the link to an instance of `OKBrokenReference`, which is a subclass of `OPModelObject` to answer to same messages, but most of them result in no action at all. With broken references, resulting models cannot be generated. To indicate that, each modular model object (not just links) answer whether can be linked to and, more importantly whether can be validly linked to. Element cannot be linked to if it is the broken reference or if it has no name, because linking is based on names of model object. Links also answer whether are valid, which is true if the element they link to can be validly linked to. In case of relationship links, also validity of source and target has to be checked.

Example of linking with broken references is shown in UML object diagram in figure 3.3.

To create a link, user clicks on a button for the correct type of link. In both cases, windows with list of possible model objects opens. These are model object that links can point to. List of possible model objects is provided by diagram controller. In case of linking Kendrick models and concerns, the controller returns all resulting models that do not already have such link from same model. In case of linking Kendrick elements, only those that can be validly linked to, are are not already linked to, are provided. After the user selects one item from the selection, the selection is returned to the diagram controller. The diagram controller provides a correct subclass of `OKModularityLink` for that class of model object. After that, the link is created with the linked model and added to the diagram in a same way like any other model object in OpenPonk and that also results in creation of proper controller.

To save the user some clicks, many links may be provided automatically based on dependencies of linked objects. Each model object used in the tool answers two collection of other model objects:

- objects that this one depends on and
- objects that depend on this one.

There cannot be any link to a model object whose dependencies do not have links to. User is, however, provided with option to add links even to such depending model objects.

To meet the criteria, required links are created automatically. Before creating the link which the user wants, the controller recursively checks for model objects this one depends one and creates their links too.

For example, if there is a container “A”, states “X” and “Y” inside the container and one transition between those states and user wants to create a link to the transition, controller (referred to as ’it’ in following list) creates links in following order:

1. It finds that the transition depends on states “X” and “Y”.
2. It finds that the state “X” depends on container “A”.
3. It finds that the container “A” depends on whole diagram, which is already linked (without that, linking of elements would not be even allowed).
4. It creates a link to the container “A”.
5. It creates a link to the state “X”.
6. It finds that the state “Y” depends on the container “A”, which is already linked.
7. It creates a link to the state “Y”.
8. It creates a link to the transition.

If there was no such functionality, the user would have to manually select the container, then those states and finally the transition, in this specific order.

To help the user even further, after any link is created, all depending model objects on linked object are linked too. For example, if the user added the container “A” from previous example first, all states will be linked right after that and since transition depends on those states, it is added to. Without that, there would be just the link to the container, but no elements displayed inside, which could lead to misunderstandings by users. However, even here the controller has to make checks whether all dependencies are linked first. For example, if the user added link to state “X”, the controller finds that transition depends on the state and tries to add link to the transition too, but it would not happen if there was no link to the “Y” state too, which the transition depends on as well.

3.2 User interface

3.2.1 Composition of main window

From the users point of view, the interface consists of two main parts. Left one for editing models and right one for displaying results. Each such part has multiple tabs, one for each model. On the left, there are tabs with definition models with editing controls. On the right, there are tabs with resulting models. Inside each of those tabs, there is another level of tabs. These inner tabs are there to enable the user to see multiple views of a single diagram. For example, one view could be focused on structure, like states and transition and other one on values of parameters or any information related to simulations.

Those single views are very similar with OpenPonk user interface. They consist of main canvas, along with a form. There is a difference on the left side and right side. On the right side, there is no palette, but only the canvas and form. Furthermore, the form has all the controls disabled, because right side is not made for editing. On the left side, there is not only canvas and form, but also a palette – column with buttons for element creation and other actions.

Many parts of interface are inspired by OpenPonk and `OKEditor` is even direct subclass of `OPEditor`. To better understand the hierarchy of Spec [23] classes used for composition of the whole window, let's first see the structure of most important classes in figure 3.5 and labeled screenshot in figure 3.4.

The main window is called a workbench (`OKWorkbench`) and contains two workspaces, a toolbar and may contain other parts related to the project or tool itself. There are two types of workspaces (`OKWorkspace`) – `OKEditWorkspace` for left part of a screen and `OKResultWorkspace` for right one. Each workspace contains multiple modelspaces which are switchable by a row of tabs. There is a exactly one modelspace (`OKModelspace`) for each model contained in a project. Such modelspaces contain modelviews (`OKModelview`). Each model can have multiple views, and for each one, there is one modelview. Modelviews are switchable by second row of tabs. Usually, there just a single modelview for the diagram and other attributes, but there might be other ones for simulations and other outputs. `OKModelview` has a subclass `OKDiagramModelview` which contains an instance of `OKEditor` or its subclass `OKViewer` for resulting models. Note that even modelspaces and modelviews have separate subclasses for left part of a screen and a right one to create proper kind of editor or viewer. In current implementation, there is also a view for Roassal [19] visualizations – `OKRoassalModelview`. It has no editor or viewer, but the Roassal view instead.

Workspaces are also reponsible for replacing tabs with resulting models when old ones are removed and new ones generated. The workspace does that by removing all those tabs and creating new ones, although replacing just changed ones would be more effective. Such replacing causes that if a user select and element in the resulting diagram, his selection is canceled every time new results are generated. It would also cause to switch currently focused tab

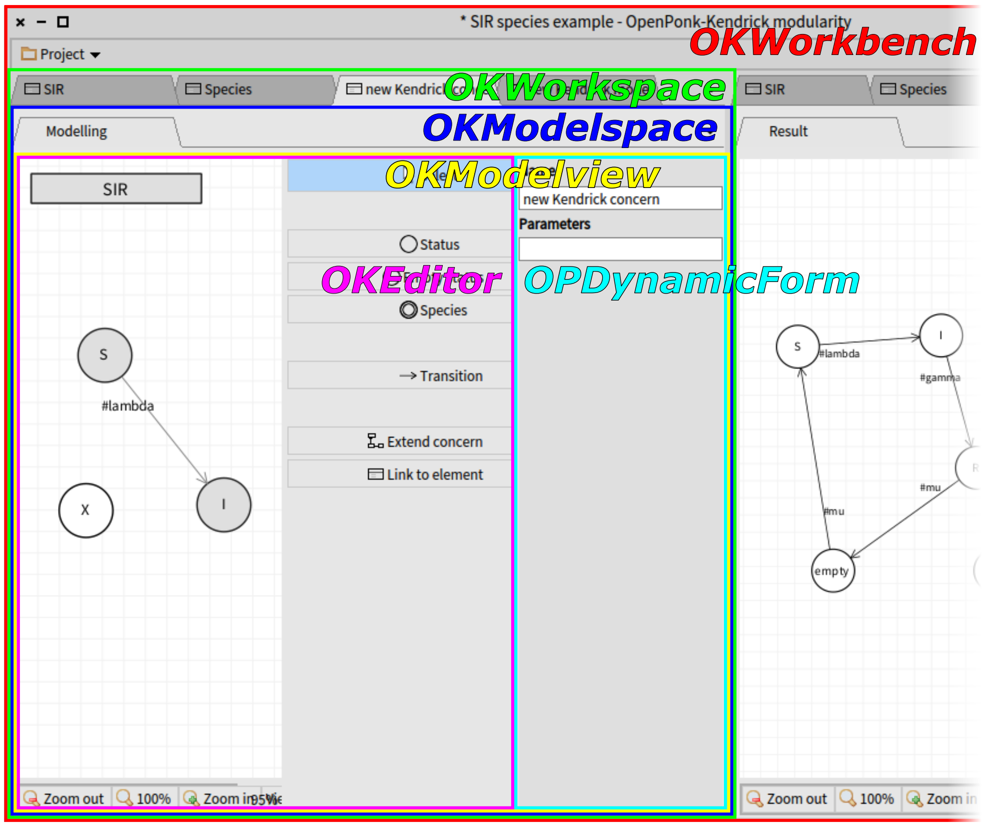


Figure 3.4: Cropped screenshot of main GUI parts with labels

to other one, but that is prevented by the workspace which remembers which one is focused and tries to find a new one with same name as previous one.

3.2.2 Creation of projects and adding models

OpenPonk does not allow to create a project with multiple kinds of diagrams using user interface, nor allows adding diagrams to existing project using user interface. Such functionality is, however, required for this tool.

If user decides to create a new project, he or she is presented with a list of possible options. In current state, only Kendrick project can be selected, but if the tool was used for multiple kinds of projects, this will be the place to select the right one. After the user clicks on one name of project kind, he or she is asked for a name and then new window with the tool itself is presented without any diagrams there. The window for project selection is named `OKNewProjectSelectionDialog` and it is referenced by `OKWorkbenchToolbar`.

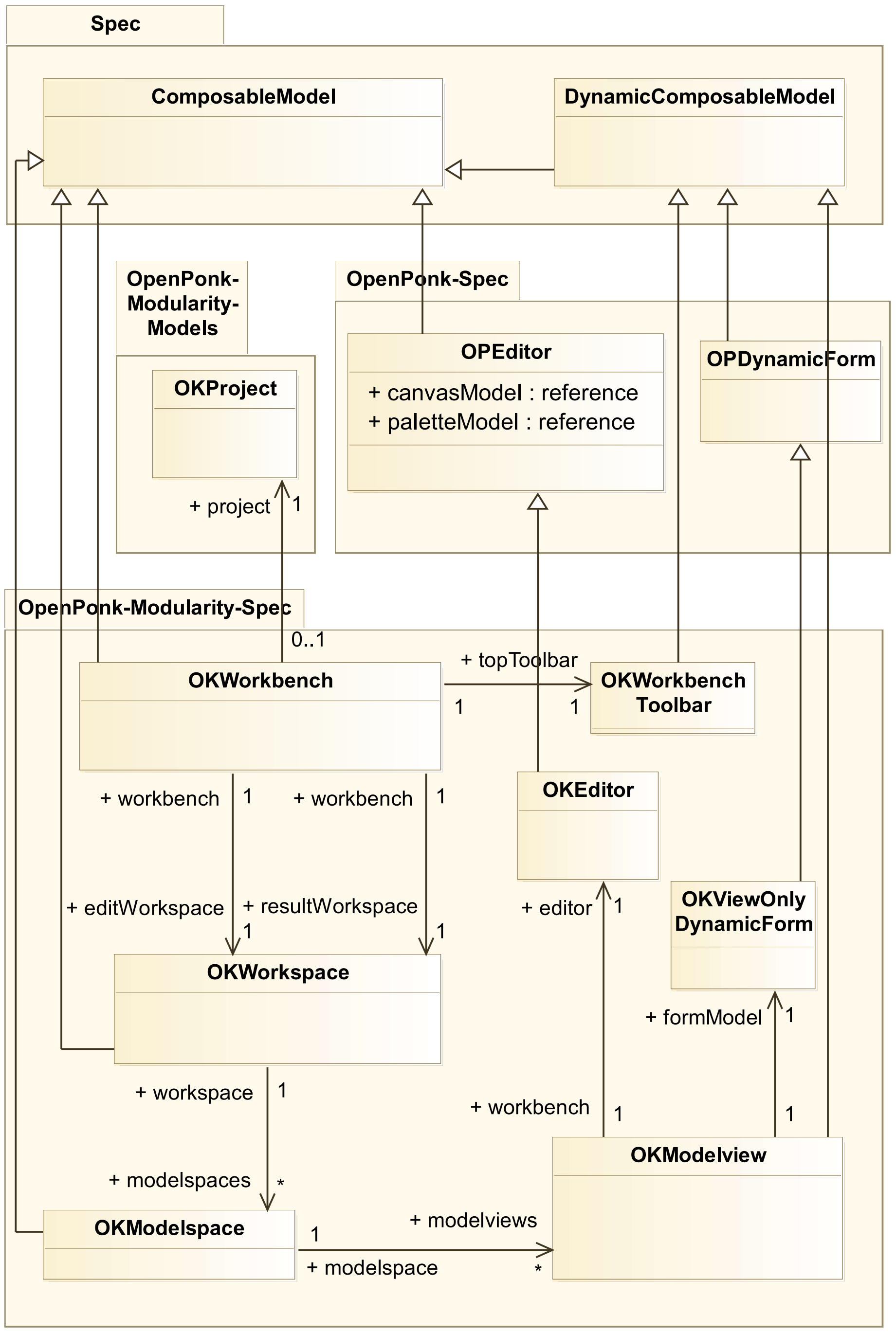


Figure 3.5: Class diagram of main structure of GUI parts

To create new model (diagram) inside the project, the `OKWorkbenchToolbar` has option to select the right kind of model. In very similar way as with project kinds, user is presented with list of model kinds provided by project controller. After the user clicks on one name of model kind, there is once again dialog window for name selection and then the diagram is added to the left part of the screen. If the empty model can be transformed into an external model using bridge, the empty diagram is added to the right side of the screen too. This window is named `OKNewModelSelectionDialog` and it is also referenced by `OKWorkbechToolbar`.

3.2.3 User interface for linking

The palette is a column with multiple buttons to add various elements to the model. For modular models, there are also buttons that provide linking options.

In unmodified OpenPonk, whenever user clicks on an element button, he is presented with option to click on a figure of model object (or whole diagram) where the selected element should be. If the user clicks on a diagram, it is added to the diagram itself and if the user clicks on a container, the new element is added to the container. The element is added to the very position user clicked on. Similarly, for relationships, user first clicks on source element and then on target element to create a relationship from the first one to the second one.

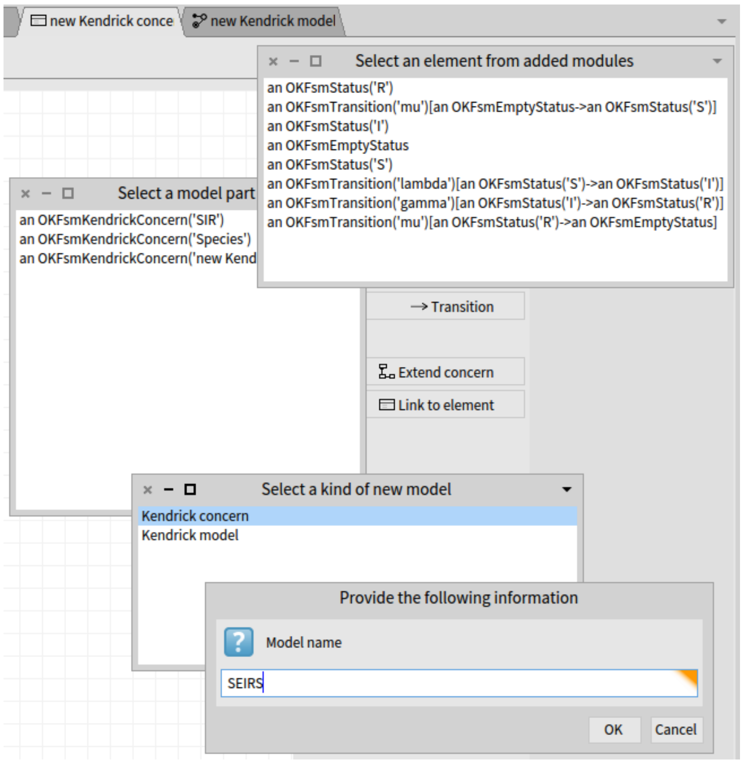
The same way of adding elements is preserved in this tool in all cases except of linking. In case of linking, clicking on the linking button in the palette behaves differently. If the button is clicked, the user is presented with a windows containing list of possible model objects that can be linked to. There could be multiple linking buttons for different purposes, all displaying different kinds of model objects. There is a class `OKSelectionDialog`, which has two subclasses – `OKModuleSelectionDialog` and `OKModuleElementSelectionDialog`. First one serves for selecting resulting models, like for example Kendrick concerns and second one serves for selection elements from already linked models, like states and transitions in those linked Kendrick concerns.

Selecting items from a list shown in figure 3.6 is just a very simple way to display these options. It could be replaced by more visual way to show those options, like selecting from the diagram itself.

3.3 Model generating and bridge

As described above, whenever there is a change in any definition diagram, its resulting diagrams and other outputs are generated and all definition models that depend on those resulting models get links updated, which also triggers a change announcement.

The OpenPonk has built-in announcement throughout its whole structure and where it does not, Pharo provides very easy way to implement announce-



```
an OKFsmStatus('R')
an OKFsmTransition('mu')[an OKFsmEmptyStatus->an OKFsmStatus('S')]
an OKFsmStatus('I')
an OKFsmEmptyStatus
an OKFsmStatus('S')
an OKFsmTransition('lambda')[an OKFsmStatus('S')->an OKFsmStatus('I')]
an OKFsmTransition('gamma')[an OKFsmStatus('I')->an OKFsmStatus('R')]
an OKFsmTransition('mu')[an OKFsmStatus('R')->an OKFsmEmptyStatus]
```

```
an OKFsmKendrickConcern('SIR')
an OKFsmKendrickConcern('Species')
an OKFsmKendrickConcern('new Kend
```

```
Kendrick concern
Kendrick model
```

```
Model name
SEIRS
```

Figure 3.6: Linking buttons and dialog windows for adding models and creating links

ments. To trigger whole model updating process, a changed model element announces its change. OpenPonk has three kinds of announcements related to model object: `OPElementChanged`, `OPElementRemoved` and `OPElementAdded`. These announcements have multiple subscriptions, specifically a container model object subscribes all its elements for changes and controllers subscribe in their model objects and project (instance of `OPProject`) subscribes for announcements of its models to mark itself dirty. The `OKProject`, a subclass of `OPProject`, subscribes for all those announcements mentioned above and whenever some of those happen, it calls a bridge to provide new resulting diagrams for the changed model.

To reduce duplicate generating of result models, there is a new subclass of `OPElementChanged` named `OKElementChangedDuplicate`. This one is announced instead of `OPElementChanged` whenever there is `OPElementRemoved` or `OPElementAdded` announced right before or after that for different model object in same model. The project then ignores this duplicate announcement regarding the result model generating.

Another new announcement is `OKLinkRenamed`. Whenever a user renames a link, the object that the link pointed to is replaced by broken reference object with the name provided by the user. It is replaced by broken reference even if any model object with such name exists, because it is not a role of element in a model to search in other models. To try to fix that link, the link announces `OKLinkRenamed`. The model is subscribed for all such announcements of its links and whenever that announcement happens, it announces it to. Finally a project is subscribed for that announcement from the model and in this case, the bridge is not asked to update resulting diagrams, but instead to update links in that model. It should be noted that updating links is a change and such change causes an announcement, so after links are updated, resulting diagrams get generated anyway with properly updated links.

The bridge remembers all relevant information in following variables.

`definitionModels` a collection with all definitions models in the project,

`externalModels` a dictionary where keys are definition models and values are collections of models provided by execution back-end like kendrick,

`resultingModels` a dictionary where keys are also definition models and values are collections of resulting models that should be displayed on right side of the user interface and finally

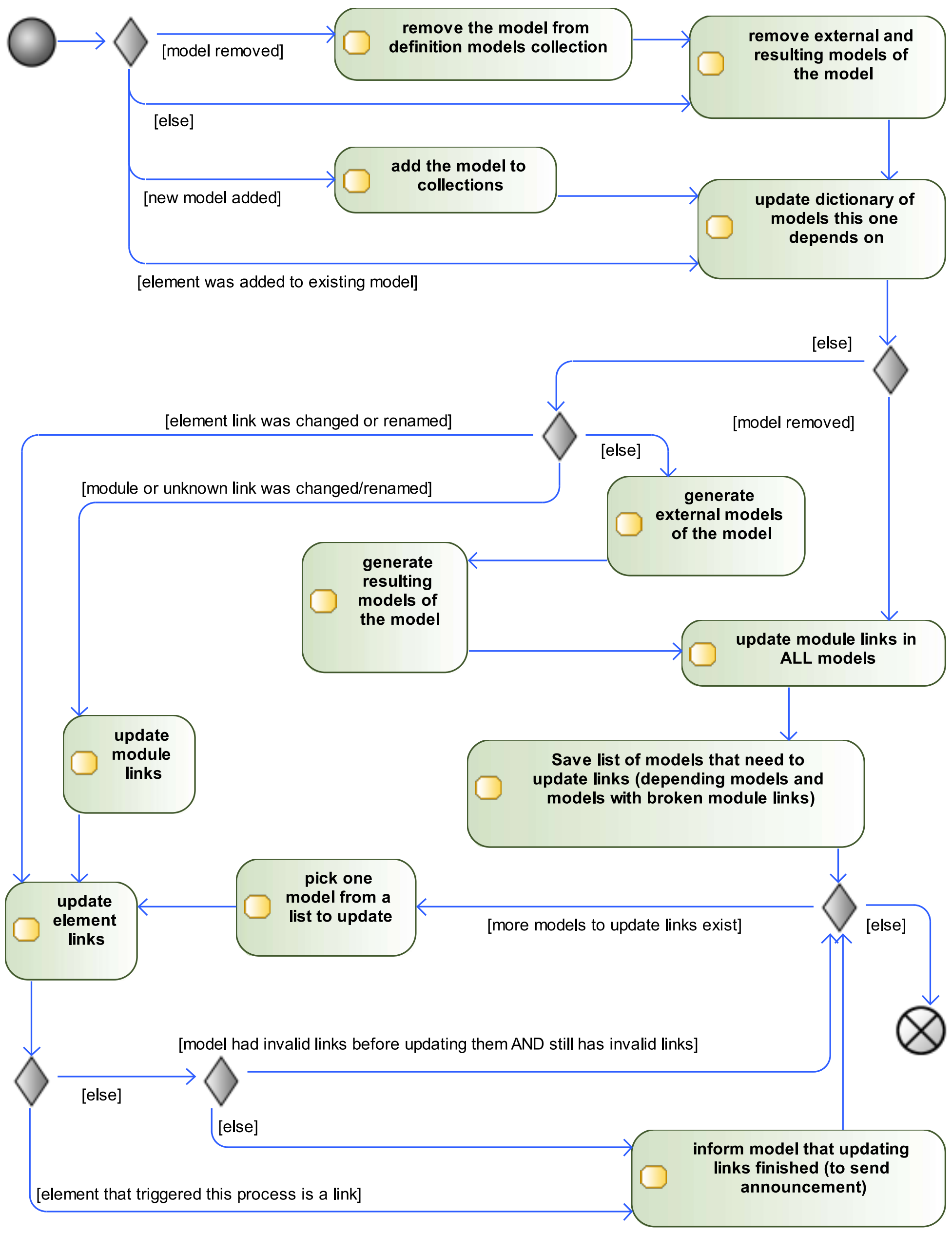
`dependingModels` a dictionary where keys are names of resulting models that any definition model depends on and values are sets of definition models that depend on resulting models with such name.

In depending models, keys are all names of models that appear in module links – links that point to whole resulting models, or broken references instead of resulting models. Values are sets of those definition models that contain such links.

For example, if there was a definition model "SEIRS" without any links and user would add module link to "SIR" resulting model, the "SEIRS" definition model would be added to the set in value with string "SIR" as key. If user renames the link from "SIR" to "RIS", the "SEIRS" definition model is removed from "SIR" and added to "RIS". On the other hand, if that "SIR" model itself gets renamed, nothing changes in this dictionary at all, because the "SEIRS" still depends on model with the same name, until its link is changed.

Whenever a bridge is asked by a project to provide new resulting models, actions that bridge performs are shown in activity diagram in figure 3.7.

The bridge needs to know which model object exactly was changed and what kind of change was it to properly react. Currently this information is used only for taking correct steps, but such information could be used even in further optimization, because different actions might be taken if whole diagram was changed or just a single element. This information about type of change is provided by a project. The project gets the information from announcement from the model.



remove the model from definition models collection
[model removed]
remove external and resulting models of the model
[else]
add the model to collections
[new model added]
update dictionary of models this one depends on
[element was added to existing model]
[else]
[element link was changed or renamed]
[model removed]
[else]
generate external models of the model
[module or unknown link was changed/renamed]
generate resulting models of the model
update module links in ALL models
update module links
Save list of models that need to update links (depending models and models with broken module links)
update element links
pick one model from a list to update
[more models to update links exist]
[else]
[model had invalid links before updating them AND still has invalid links]
[else]
[else]
inform model that updating links finished (to send announcement)
[element that triggered this process is a link]

Figure 3.7: Actions inside a bridge when a model object is changed, added or removed

By adding the model to collections is meant to add it to collection of definition models, create a key in dictionary of external models and a key in dictionary of resulting models. By updating dictionary of models this one depends on is meant to remove this model from all sets by all keys in `dependingModels` dictionary.

Note that removing definition models is not implemented as it is not fully implemented even in original OpenPonk, but all the necessary steps are designed in the activity diagram.

Although a model could autonomously send announcement after every change of any of its links, it would lead to announcing for every single link in the model which would cause to serious performance problems, because after every such change, whole transformation process would happen. Instead, the model is informed by the bridge that it finished updating its links and then the model announces it only once.

After resulting models are generated, or when the definition model is removed and therefore its resulting models are removed too, it is stated in the diagram that all modules links in all models are updated, instead of just depending ones. Since dependencies across models are based on names only, problem can happen if the model itself is renamed or removed and module links in all other models were not updated too. Let's have a "SIR" resulting model and a depending definition model that links to this "SIR" model. If the "SIR" is renamed to "SIR1", the bridge might search for any definition models depending on resulting model "SIR1", but none such can be found (because it links to "SIR", not "SIR1") and no links are updated. The definition model linked to "SIR" remains linked to the model already removed from the project. Nevertheless, this potential problem can be solved easily and for that, there are at least these two options:

1. Do not resolve dependencies just by names of linked models. Instead, remember dependencies by those resulting models themselves and whenever that resulting model should be removed or replaced from a project, check for definition models depending on this resulting model.

2. Whenever resulting models are removed (in case their definition model has been removed) or generated, update all module links in all diagrams. It ensures that models get invalidated whenever they should.

Although the second option is used in this implementation as it is the simpler one, first one would be just as valid one and might even lead to better performance.

To add dependencies on a model, e.g. to add definition models depending on resulting models to `dependingModels` dictionary, all names of models referenced by module links of current definition model are collected and added as keys with the current model added to the set that is a value with that key.

When updating links in a bridge, first module links and then element links are updated. Both are, however, updated similarly.

For module links, all models known to the bridge that can be validly linked to, e.g. have a name, are collected. Then all module links that do not point to any of those models get new model object to link to. If there is a resulting model with same name as the link points to now, the link gets pointed to this new resulting model. If there is no resulting model with such name, the link gets pointed to a broken reference with that name instead, thus making the link invalid.

Updating element links is very similar. All elements in all linked resulting models are collected first, e.g. All elements from such resulting models, for which there is a valid module link from a current model. Then all element links that do not point to any of those elements get new element to link to. If there is an element with same name as the link points to now, the link gets pointed to this new element. If there are no elements with such name, the link gets pointed to a broken reference with that name instead, this, also, making the link invalid.

Note that updating links could be potentially separated from the bridge, since updating links does not rely on the external back-end system.

This is all that the abstract `OKBridge` class provides. Main transformations of models are fully implemented by subclasses of the bridge for every single kind of execution back-end. One such execution back-end is Kendrick and implementation for the Kendrick is described in following chapter.

3.4 Simulations and other kinds of output

In user interface, every modelspace can have multiple modelview. Usually it is only one for diagram, but even multiple ones could be created.

Every OpenPonk-based model (class that uses `OKTModularDiagram` trait) answers collection of additional views. By default, it is an empty collection so only one modelview is created for a single model, however, the model can return multiple other views.

While the bridge is creating an external and resulting model, it may also create other kinds of outputs, like visualization of simulation. This is fully handled by subclasses of bridge for specific execution back-ends. The bridge then adds such outputs to the resulting model itself and it passes them to modelspace when asked for.

One example of simulation of Kendrick “SIR” model with three species is shown in figure 3.8. Note that even simulations can be displayed independently from actions on left side of a screen so that the user can see how simulation changes based on changes in modules.

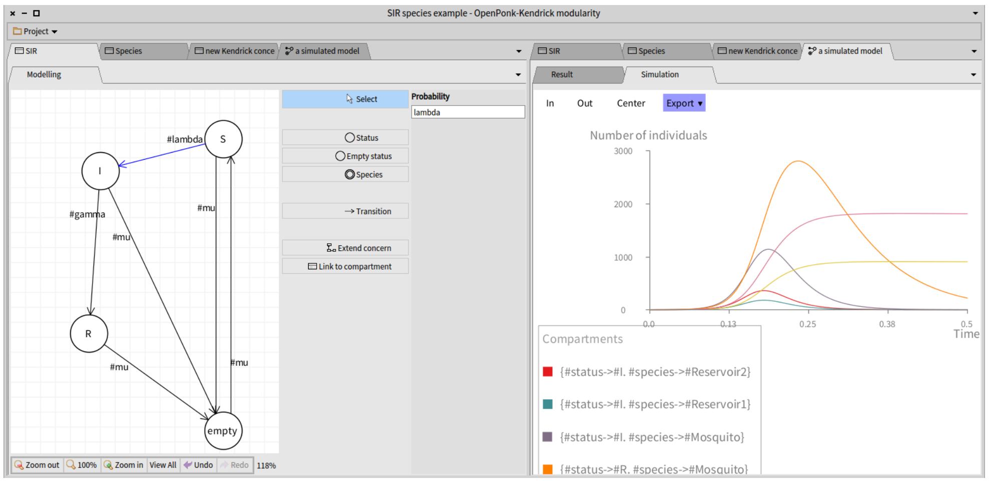


Figure 3.8: Simulation of Kendrick “SIR” model with three species, while modifying “SIR” concern

Chapter **4**

Implementation for Kendrick

Since Kendrick models and options are powerful and complex, only a small subset of Kendrick functionality has been implemented to the tool as a case study. List of supported features follows.

- Creation of models and concerns.
- Extending concerns and integrating concerns into models.
- Creating named states and species.
- Creating transitions with numeric or single-parameter probabilities.
- Delaying transitions by providing name of new state and transition probability.
- Splitting a single state into states named with list of suffixes or suffixes created from intervals with steps by 1.
- Creating a model's initial population amount.
- Giving parameters numeric or single-parameter probabilities.
- Displaying all information above in resulting models.
- Defining and displaying simulations of SIR-based models with species.

The list is very short in comparison with Kendrick features and even options of those simulations are very limited. To provide at least some options to define rest of important parts until a better way is provided in following work, an option to write arbitrary code has been given to the user, where the code is executed as a block on the Kendrick model as a last step of creation of the model in the bridge, similarly to writing Pharo core for the Kendrick API. That would not be possible in many programming languages, but in Pharo smalltalk [4] it was quite simple task due to its dynamic nature and meta-programming possibilities [21].

As stated in previous chapters, for each kind of back-end system, several components need to be created. To show an example of such components, this thesis uses Kendrick [3] as a case study.

Components required by the case study tool are a subclass of project and a subclass of bridge. It also requires models and notations suitable for diagram-based modeling and these are implemented like any other new kind of model for OpenPonk [2].

To create new kinds of models for OpenPonk, the developer has to provide all parts of the MVC architecture – model, controller and Roassal [19] figures. Furthermore, OpenPonk requires subclass of `OPPlugin` that provides basic information about that kind of model, like name, class of model, class of controller and, possibly, an icon for GUI. To be able to save and load a project, classes related to serialization should be provided too.

4.1 Model-view-controller class structure

Models, figures, controllers and other elements are based on OpenPonk finite state machines package.

Connection of model classes with other packages is shown in figure 4.1, where only the topmost classes (those without any superclass in same package) of a new OpenPonk-Modularity-Kendrick package are displayed.

All model classes with most important methods and few other important classes mentioned in further sections are displayed in figure 4.2. This diagram, however, does not focus on connections to other packages

Kendrick model structure consists of a concern and a model. Concerns and models need separate model classes in OpenPonk with one common subclass – `OKFsm`.

Concern is represented by a class `OKFsmKendrickConcern` and model by a class `OKFsmKendrickModel`. The `OKFsm` has three additional variables to those in `OPFsm` and `OPModelObject` – `externalKendrickModel`, `parameters` and `additionalCode`. Variable `externalKendrickModel` is used for resulting models where the original Kendrick model is saved. Variable parameters is a dictionary where values of parameters can be defined. Variable `additionalCode` contains a string that is executed in a block on Kendrick model after it is generated.

To provide simulations, `OKFsmKendrickModel` also contains collection of Roassal views in a single variable, as well as other variables and method for defining parameters of simulation along with a name, population and other usual information. Views are provided to resulting model by a bridge.

To access these variables, various accessing methods are provided, including ones to parse parameters from a single string or returning a block with

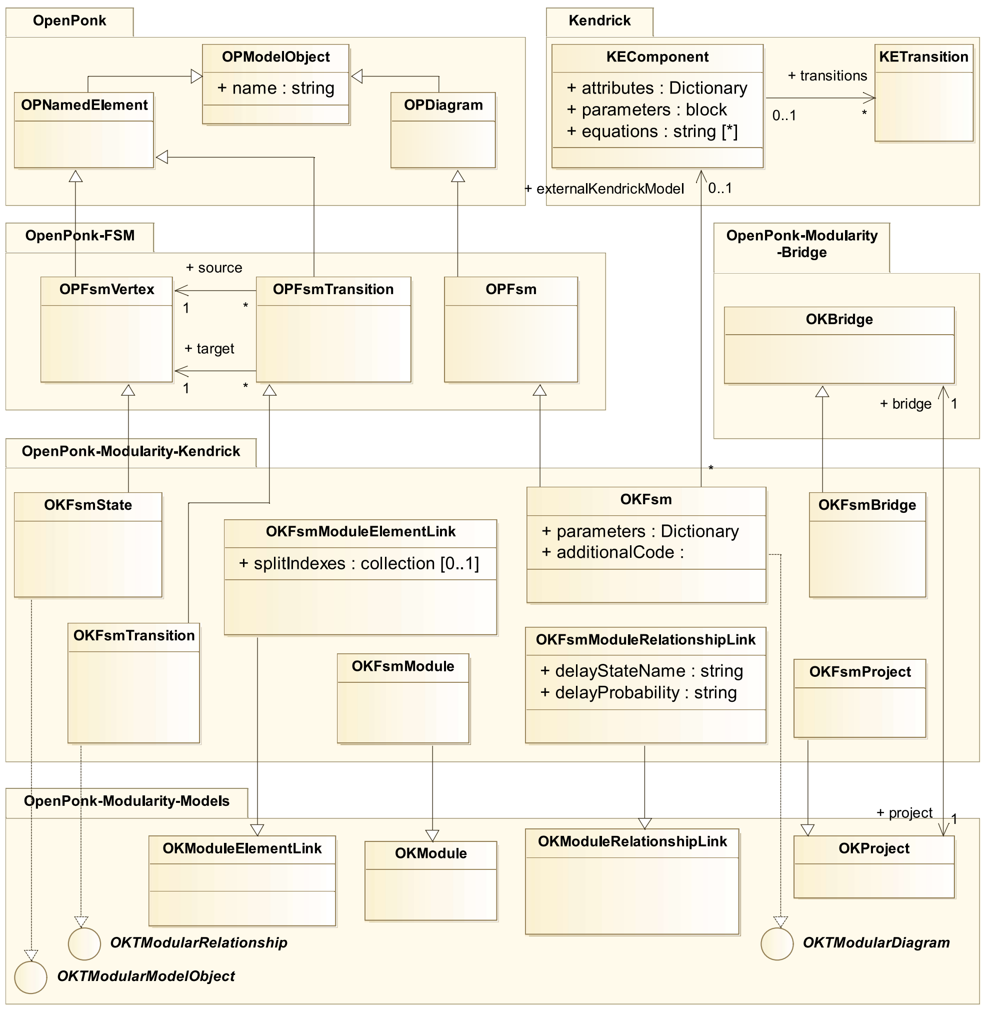


Figure 4.1: Connection of model classes in package OpenPonk-Modularity-Kendrick

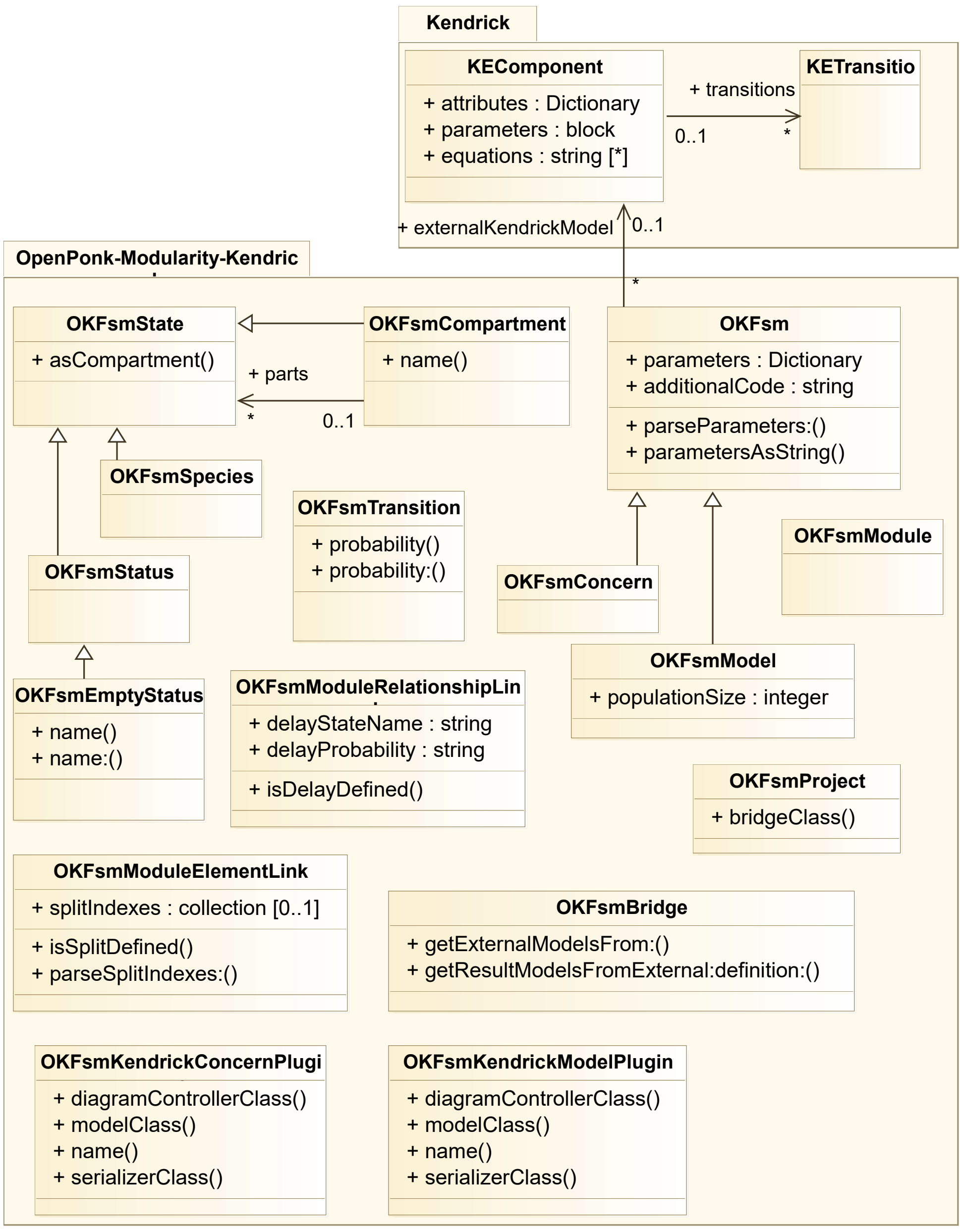


Figure 4.2: Detailed model and other classes in package OpenPonk-Modularity-Kendrick

the additional code.

As all models used in the tool, OKFsm uses OKTModularDiagram trait to provide basic modular functionality.

In addition to that, OKFsmKendrickModel has a populationSize variable for a single numeric value.

States are subclasses of OPFsmVertex – an abstract class of OpenPonk finite state machines. All states answer its attribute type, which can be, for example, state, species or joined compartment. OPFsmVertex has multiple subclasses: OKFsmStatus, OKFsmSpecies, OKFsmCompartment, and special subclass of OKFsmStatus – OKFsmEmptyStatus with disabled renaming. Implementation of the compartment is based on a single collection of parts which are other elements. For example, a compartment can have a collection with instance of OKFsmStatus and OKFsmSpecies, each with its name. For displaying in diagram, it answers its name like joined strings of those parts, but a bridge can access separate parts.

OKFsmTransition is another model class, which has a probability. Since probability is displayed in a same way as state machines transition name and Kendrick transitions do not have any names but are defined by source and target elements, the name variable is reused for this purpose, until additional functionality is required.

Since links are model elements too, they have subclass for Kendrick with additional functionality. Element link – OKFsmModuleElementLink – has variable with information about indexes or names for splitting a status. It has also various new functionality, including returning related elements (which are connected transitions), answering its attribute type, which is, in fact, attribute type of linked element. Relationship link – OKFsmModuleRelationshipLink – has variables for delaying the state they link to.

Each model object above has its own controller class. These classes usually just answer the class of their model object, except of controller of transitions which has modified way of refreshing figures to display probability properly and it also answers to buildEditorFrom: message to add the probability entry to an instance of OPDynamicForm. Some controllers have probability entries too and some answer canBeTargetFor: and #canBeSourceFor: messages. Specifically, instance of OKFsmController can be target for module links, instance of OKFsmKendrickConcernController can also be target for states and element links and finally, instances of OKFsmStateController and OKFsmModuleLementLinkController can be targets or sources for transitions and relationship links.

OKFsmController, which uses OKTModularDiagramController trait, has multiple more methods. First of all, it defines its default layout. It also adds entry to the form to enable providing values to parameters for the Kendrick model. The OPDiagramController, controller of the model itself, references its editor. However, when this editor is set via accessor, it checks whether its type is `OPEditor`. This method had to be replaced in `OKFsmController` to remove the check, because type of editor in this tool is `OKEditor`.

All diagram controllers in OpenPonk are responsible for implementing a method `modelToControllerMapping`, which provides a dictionary with model object classes as keys and controller classes as values. `OKFsmController` has new, but similar, method – `modelToLinkControllerMapping`, which provides a dictionary of model object classes with proper link controller ones.

Controllers for states also replace creation of figures for Roassal canvas by new figures for states and species. These new figures are subclasses of abstract finite state machines class `OPFsmAbstractStateFigure` and are very similar to finite state machine states, both by implementation and looks on canvas.

Diagram controllers provide palette entries as well. Palette contains buttons for adding new elements.

In case of concern, palette contains buttons for creating Status, Empty status, Species, Transition, module link for extending concern and element link from the extended concern. Concern controller also answers different possible modules from general modular controller, since it does not list any modules if there is one already, because only one concern can be extended.

In case of model, palette has only a single button – for integrating a concern. It also adds new items to the form: population size text input for integers and additional code button that opens window for editing the code. Furthermore, the controller replaces method that answers possible elements from modules, because model is not allowed to create element links, just to integrate already prepared concerns.

4.2 Project and plugins

The case study tool requires own subclass of `OKProject` for each type of project, so `OKFsmProject` is provided that answers its bridge class – `OKFsmBridge`.

Concern plugin provides its diagram controller class, its model class (`OKFsmKendrickConcern`), an icon, name and serializer class. Model plugin is very similar with proper classes.

4.3 Persistence and serialization

For persistence support, serializer classes are provided, which answer methods `materializeModelFrom:` and `serializeModel:` and use subclasses of `OPFsmGraphML` class which is based on XML format and parsing.

These subclasses provide a way to save and load all elements, parameters and other variables that are supported by models and elements.

4.4 Bridge

Last, but very important, part, is a bridge. `OKFsmBridge` is a direct subclass of `OKBridge` and it provides two methods required of all subclasses of the `OKBridge`. One to get external models from definition model and other to get resulting models from external one.

In case of Kendrick, only one external model is generated from definition one and only one resulting model is generated from external one.

Creating external model starts by validating all links. If any link is invalid, the model is not generated. Then instance of the Kendrick model or concern class is created and the bridge takes one step at a time to add all possible information from the definition model to the external model.

It adds modules – extended concern in case of concern or integrated concerns in case of model. Then it has to make a deep copy of all transitions in case of concern extension, because those would otherwise remain same as in the extended concern and would cause that the extended concern would be unintentionally modified. If it is a model, population with provided size is created. Next, all original states (those that are not links) get added to the model or concern followed by transitions.

When all the simple elements are created, information from links are processed. First, any relationship links with defined delays are used to do that delay in the Kendrick model. Implementation of delaying is inspired by its implementation in Kendrick's DSL. After that, similar actions are made for splitting states and then, parameters are forwarded to the model.

As a last step, additional code provided by the user is executed with the Kendrick model as an argument. In case of an error, a debugger window is opened, but when the user wants to proceed, it allows him to do so, but the model does not get generated.

To provide resulting model from external one, even the original definition model is used to copy its name for the instance of `OKFsmModel` or `OKFsmConcern`. When creating the instance, the name is provided along with the Kendrick model or concern to save it for further use.

First the Kendrick concern or model is checked for attributes – whether there are any states and species. If there are none, empty model is answered because if there are no attributes, not even transitions, population and other variables are there.

After that, attributes are transformed into instances of `OKFsmStatus` subclasses. If there is only one key in attributes dictionary, i.e. only states or only species, the corresponding elements are immediately created. If there are more (in current implementation, only states with species can be there), compartment model element is created instead and provided with instance of `OKFsmState` and `OKFsmSpecies` as its parts.

States are followed by transitions, where all transitions except of those with zero probability are added to the diagram.

Creation of the model itself is finished by population size and parameters are forwarded to the model.

Finally a simulation is executed, Roassal view of results is build by the Kendrick and the Roassal view is added to the model. The model then provides this view to modelspace and modelview when asked to.

Chapter 5

Testing

The tool and its usage for Kendrick is tested in two ways. First are automatic unit tests and second are user scenarios supported by executable examples.

5.1 Unit tests

The implementation consists of the base of the tool itself and usage for the Kendrick. For Kendrick, almost all the code is there just to provide base for saving information of those models or to create Roassal figures. Such classes are not tested by unit tests, but examples are focused on them instead. The same applies for the Kendrick subclass of the bridge, because it heavily relies on the Kendrick itself as the execution back-end.

In the core of the tool itself, code is either taken from the OpenPonk by subclassing, or newly created for the tool. Most new crucial logic lies in the bridge, therefore it is thoroughly tested by unit tests. Links, models and controllers are mostly very simple classes that usually serve just to save information or to provide constant answers based on their classes and testing them is therefore not as crucial as with the bridge, especially because there is a great chance that their functionality will be modified in the future. Other parts, like user interface or link figures, are tested by user scenarios.

As noted above, most important part which may cause multiple kinds of problems is the bridge because of its complexity. To create a proper unit tests, several new classes have to be provided. These so called "dummy classes" are overly simple classes that mock behavior of their real counterparts, but all is usually static and scripted just for those tests.

Namely, there is `OKDummyBridgeSubclass`, `OKDummyModel`, `OKDummyLink`, `OKDummyElement`, `OKDummyProject` and a few others.

`OKBridge` is an abstract class and it needs a subclass to work. For that, `OKDummyBridgeSubclass` has been created and it returns always the same collection of models if links are valid.

Similarly, `OKDummyProject` returns always the same collection of models and `OKDummyModel` has always the same simple element (`OKDummyElement with the same name`).

The `OKDummyLink` a little more sophisticated, as its linked object can be altered to test updating links. `OKDummyModel` also has multiple modifiable variables that can be modified by individual test so that this model answers data that are needed to test the proper aspect of the bridge.

The set of tests itself contains various tests from a simple creation with a project to testing of updating links and generating resulting models when the model that this one depends on is changed in various ways.

There are also unit tests for links and new functionality of model objects. Since some of that functionality is provided by traits, “dummy” classes that use these traits had to be created too.

5.2 User scenarios

All user scenarios use Kendrick models to test project and model creation, functionality of Spec classes [23], providing proper models to the user interface by a project and also displaying figures on a canvas. These scenarios are usually supported by examples that prepare all required models and elements to save time, if their creation has been previously tested successfully.

Chapter 6

Discussion and future work

Although most of the fundamental parts of the problem were solved, there are a few unsolved parts, parts that might not work in certain conditions and implementation-related problems that require future work.

6.1 Simulations

Although the tool provides a solution for displaying simulations and other kinds of outputs in multiple views that are switchable by a second row of tabs, only implemented outputs are usual diagrams and Roassal views [19]. Other kinds of views yet need to be implemented. Parameters for the simulations are provided by definition model and such parameters are set by the user on the same place as name of the model and other information.

Just as simulation definition is part of definition model, all outputs related to a single model are parts of the model. Specifically, all such other outputs are saved in a variable of a resulting model. That does not present a problem for displaying in other views, but there is one alternative to such multiple-view-based approach.

It should be noted that these views were created not only with simulations and Roassal views in mind, but also to be able to define models not only by diagrams, but also by DSL. Whenever user modified a diagram, new DSL would be generated and whenever the DSL is changed, diagram would be redrawn. This is, however, not yet fully designed and implemented.

Other option is to have simulations completely separate, on the same level as discrete models. The simulation definition and result would be handled in similar ways as definition models and resulting models, and they will similarly link to their resulting models. Whenever a resulting model that is linked to from simulation is replaced by new one, the simulation will also be generated and displayed the same way as diagrams of models. This option will require creating new kinds of entities for such inputs and outputs on various levels of the tool itself and its implementation for Kendrick models, because in OpenPonk and the case study tool, all parts presume that there are only models with elements and other similar properties. In a bridge, this kind of entity will be transformed to external entity or multiple entities (like simulation and visualization) and then transformed back to entity that contains the visualization or other outputs that will be displayed instead of a model.

Although this option could be time consuming during both design and implementation, there is one great benefit: Independence of the model and the simulation. The same simulation could be quickly used for different model and multiple simulations of the same model could be easily created and compared.

6.2 Linking

There is one potential problem with current way of linking: model kinds that do not use names for all crucial elements and those that allow duplicate names.

This is all caused by the fact that all parts of linking and finding dependencies is based on names of model objects. Links can be subclassed to provide different kind of identification, like if models had unique ids that would be preserved after generating new resulting models or like it is already used for relationships, that do not have a name, but their source and target elements do and the source and target element names therefore identify the relationship. That can, of course, cause a problem in models that allow multiple relationships from same element to other same element.

Another problem could be with refactoring that influences multiple dependent models. The unanswered question in this case is how should such links change in case of renaming an element in definition model. After going through the process of generating new resulting models, there is currently no sure way to indicate that element “A” is not there anymore because it has now name “B”. Furthermore, in process of generating resulting models, names could be changed. For example, in simple model constructed by Cartesian product of all elements in modules, those names could be “A - B” to indicate an element “A” from first module with element “B” from second module. If the element “A” is renamed to “C”, resulting element would be “C - B”. To properly refactor such names, the tool, or a component that will do the refactoring, would need to know how the external back-end produces names. Those names do not necessarily have to be created by simple join of original names, but by more complex procedure which causes inability to backtrack the new name to names of original elements from which was this one created. If such rename was needed when using any kind of DSL, this all will be up to the user and his knowledge of the system to know how renaming will affect dependent models to make all the changes in their definition by hand before using the DSL code. Similar user-made actions are currently needed in the tool, otherwise link is marked invalid and resulting model is not generated.

Switching from using names to different kind of identification will cause problems dependent on whether only element names are not present or are duplicate, or even whole models could have duplicate names. In case of elements, only different links would need to be provided, similarly as relationship links are. If there were models where even definition and resulting models couldn't have a name or will have to allow duplicate ones, multiple changes will be required in links, `OKBridge` class itself and also in `Workspace` class, which uses model names during replacement of resulting models.

6.3 Highlighting changes

Important part of motivation for this work and the tool was to provide a way to explore possibilities of a model and to see how different settings and composition affect results. In current tool, the results (more specifically, resulting models) are always up-to-date with all definitions of models and modules. This way of providing live feedback could be greatly improved in future work.

After every replacement of resulting models, these models will be checked for changes. These changes will be displayed to the user in, for example, color notation or list of changes. If there was a new element, it would be colored green. If an element has been changed, it could have blue or yellow color. If an element was removed, it would still be present in the diagram, but with red color.

The user could be given an option whether to show only changes made by his or her last action or whether to keep all changes displayed until reset button is pressed.

Such improvement is based on following main modification:

1. Provide an ability to keep previous versions of resulting models,
2. create a way to visualize such changes, including displaying removed elements, on the canvas,
3. create proper controls for the user interface and
4. each kind of model would have to provide a way to compare previous model with new one in proper way.

Although these modification are not trivial to design and implement, such improvement could considerably ease exploration of models and modules.

6.4 Optimization

One problem with the implemented tool is its performance. Many user-made changes cause the tool to hang for several seconds. There are multiple reasons for that and multiple options to improve the situation.

Whenever any resulting model is generated, the bridge checks all models for invalid module links. This is to overcome a problem that after a resulting model has its name changed, depending models are not generated because they depend on model with different name that this new one has. This way of solving this particular problem could be altered, if the bridge remembered which previous resulting models are no longer amongst new resulting models and therefore update only diagrams that are depending on the old resulting model. That way, the dictionary with depending models would not use names as keys, but these resulting models themselves.

There are many other places in code which could be improved by caching of values instead of repeated enumeration. Most of these parts are in methods of `OKBridge` class and `OKTModularDiagramController` trait.

A great source of hangs of user interface are changes of names and other textual data. OpenPonk models, to provide fastest possible visualization on canvas, send `OPElementChanged` announcement with every keystroke in the text field. That it fine for a monolithic tool with just a single dependent view, but it causes serious performance issues in this case study tool where multiple models depend on a single one.

Most problems described above could be solved by a complex system of transforming as a background task. Whenever user makes a change, the tool will indicate that it is redrawing results and other models, but it will do that in different process and the user interface would still be responsive. Furthermore, if there were multiple changes of the same model in a short time, previous process of transformation could be stopped and replaced by new one. It should be noted that OpenPonk itself is not ready for such asynchronous changes while user works with user interface and there could be multiple secondary problems that would need to be solved.

6.5 Other notes

Any user with multiple screens would benefit from having more than two parts of the screen. If there was such requirement, the tool could be modified for displaying multiple results at the same time, like both resulting model and its simulation.

Although the tool and all the solutions were made with all possible kinds of models in mind, even when using similar solution in different tool, all has been tested only on a very small part of Kendrick functionality in the OpenPonk-based tool. Both Kendrick and OpenPonk are implemented on Pharo platform [4] and therefore even this tool is implemented in Pharo. Specifically, it is tested mostly in Pharo 6 on Windows 10.

The tool has not been tested with hierarchical models. For such kind of modularity, the model used as an element would be represented by module link that will be displayed in a similar way as element links are in the implementation for Kendrick. No further changes should be required, but it was not yet proved.

The implementation for Kendrick also does not yet fully support persistence, i.e. it is not able to save all information contained in models and their elements.

As the last note, since the graphical user interface of the tool consists of two parts of screen, both with own controls, diagrams and possibly other graphics or data, the tool is not very well suitable for smaller screens. The screen is recommended to be at least 1600 px wide.

Conclusion

There are many complex modular modeling systems that would benefit from a tool that offers modular modeling with live visualization of resulting models and other results, to allow exploration of models and model parts in more dynamic way than using an API or DSL.

This work offers a solution that requires creation of additional diagram-friendly models and notations that provide proper visual modeling. The solution proposes a graphical user interface with two main parts. Left half of a screen serves for creating, defining and modifying models and other parameters, which get transformed into external models and outputs by external back-end execution systems. Models on left half of a screen are called definition models. Right half of a screen serves for displaying and visualizing various outputs, like simulations, images and resulting models that are transformed from external models. The user can select any resulting model or output on the right while editing any definition model on the left.

As a case study, a tool based on OpenPonk [2] modeling platform has been implemented and in this new tool, small subset of features of Kendrick [3] epidemiological modeling system has been implemented, including diagram-friendly notations and own bridge. Kendrick, therefore, serves as an execution back-end system for this new tool.

For communication with the back-end system, a component called a bridge has been created. This component handles all communication with the back-end system and handles links between definition and resulting models. Each type of a project has its own bridge that implements communication with the external back-end its own way.

Because of that, even proprietary back-end systems written in other programming language or cloud ones can be used. On the other hand, it requires complete creation of those diagram-friendly models and ways to transform them in the bridge.

Most of main objectives of the work were satisfied, but the solution is not yet fully suitable for all possible kinds of models, especially not those without element names, with multiple relationships of same kind and direction between same elements and those allowing multiple elements of same kind with identical names. The solution has also not been proved on anything else than a single case study based on OpenPonk and small subset of Kendrick functionality. The implemented tool itself needs further work in many respects, like optimization, user interface and support for Kendrick features. Despite all such imperfections, this work, hopefully, pushed forward research related to live visual modeling of modular models and will prove to be a solid base for further work in that area.

Bibliography

[1] Miller, T. D.; Elgard, P. Defining modules, modularity and modularization. In *Proceedings of the 13th IPS research seminar, Fuglsoe*, 1998.

[2] Uhnák, P.; Pergl, R. The OpenPonk Modeling Platform. In *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies*, IWST'16, New York, NY, USA: ACM, 2016, ISBN 978-1-4503-4524-8, pp. 14:1–14:11, doi:10.1145/2991041.2991055. Available from: `http://doi.acm.org/10.1145/2991041.2991055`

[3] Anh, B. T. M.; Stinckwich, S.; et al. KENDRICK: A Domain Specific Language and platform for mathematical epidemiological modelling. In *The 2015 IEEE RIVF International Conference on Computing Communication Technologies - Research, Innovation, and Vision for Future (RIVF)*, Jan 2015, pp. 132–137, doi:10.1109/RIVF.2015.7049888.

[4] Black, A.; Ducasse, S.; et al. *Pharo by Example*. Square Bracket Associates, 2009, ISBN 978-3-9523341-4-0. Available from: `http://pharo-project.org/PharoByExample`

[5] Bui, T. M. A.; Papoulias, N.; et al. Explicit Composition Constructs in DSLs: The case of the epidemiological language KENDRICK. In *Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies*, ACM, 2016, p. 20.

[6] Sparx Systems. Enterprise Architect [software version 13.0.1310]. 2017. Available from: `http://www.sparxsystems.com.au/products/ea/trial.html`

[7] Modeliosoft. Modelio [software version 3.6.01]. 2017. Available from: `https://www.modelio.org/downloads/download-modelio.html`

[8] Broeck, W. V. d.; Gioannini, C.; et al. The GLEaMviz computational tool, a publicly available software to explore realistic epidemic spreading

scenarios at the global scale. *BMC Infectious Diseases*, volume 11, no. 1, 2011: p. 37, ISSN 1471-2334, doi:10.1186/1471-2334-11-37. Available from: `http://dx.doi.org/10.1186/1471-2334-11-37`

[9] Mirschel, S.; Steinmetz, K.; et al. PROMOT: modular modeling for systems biology. *Bioinformatics*, volume 25, no. 5, 2009: pp. 687–689.

[10] Maloney, J.; Resnick, M.; et al. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, volume 10, no. 4, 2010: p. 16.

[11] Resnick, M.; Maloney, J.; et al. Scratch: programming for all. *Communications of the ACM*, volume 52, no. 11, 2009: pp. 60–67.

[12] Microsoft. Live Programming. Mar 2014, visited on 2017-05-05. Available from: `https://www.microsoft.com/en-us/research/project/live-programming/`

[13] McDirmid, S. Usable live programming. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, ACM, 2013, pp. 53–62.

[14] Burckhardt, S.; Fahndrich, M.; et al. It's alive! continuous feedback in UI programming. In *ACM SIGPLAN Notices*, volume 48, ACM, 2013, pp. 95–104.

[15] Ohshima, Y.; Mönig, J.; et al. A module system for a general-purpose blocks language. In *Blocks and Beyond Workshop (Blocks and Beyond)*, IEEE, 2015, pp. 39–44.

[16] Patterson, R. D.; Allerhand, M. H.; et al. Time-domain modeling of peripheral auditory processing: A modular architecture and a software platform. *The Journal of the Acoustical Society of America*, volume 98, no. 4, 1995: pp. 1890–1894, doi:10.1121/1.414456, `http://dx.doi.org/10.1121/1.414456`. Available from: `http://dx.doi.org/10.1121/1.414456`

[17] Hale, R. E.; Cetiner, S. M.; et al. Update on Small Modular Reactors Dynamic System Modeling Tool Web Application. Technical report, Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States), 2015. Available from: `https://info.ornl.gov/sites/publications/files/Pub53985.pdf`

[18] Jensen, K.; Kristensen, L. M.; et al. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, volume 9, no. 3-4, 2007: p. 213.

[19] Bergel, A.; Cassou, D.; et al. *Deep Into Pharo*. 2013, ISBN 978-3-9523341-6-4, 209-240 pp. Available from: `http://files.pharo.org/books-pdfs/deep-into-pharo/2013-DeepIntoPharo-EN.pdf`

[20] Bergel, A. *Agile Visualization*. Square Bracket Associates, 2016.

[21] Callaú, O.; Robbes, R.; et al. How (and why) developers use the dynamic features of programming languages: the case of smalltalk. *Empirical Software Engineering*, volume 18, no. 6, 2013: pp. 1156–1194.

[22] Auer, M.; Tschurtschenthaler, T.; et al. *A flyweight uml modelling tool for software development in heterogeneous environments*. IEEE, 2003.

[23] Fabry, J.; Ducasse, S. *The Spec UI framework*. 2 2017, ISBN 978-1-326-92746-2. Available from: `http://files.pharo.org/books-pdfs/spec/2017-01-23-SpecBooklet.pdf`

[24] Aalst, W. van der. Hierarchical Petri nets [online presentation]. [visited 2017-05-07]. Available from: `http://cpntools.org/_media/book/hcpn.pdf`

[25] EventHelix.com. Hierarchical State Machine [online]. [visited 2017-05-07]. Available from: `http://www.eventhelix.com/realtimemantra/hierarchicalstatemachine.htm`

[26] Harie, Y.; Wasaki, K. Formal verification of safety testing for remote controlled consumer electronics using the Petri net tool: HiPS. In *Consumer Electronics, 2016 IEEE 5th Global Conference on*, IEEE, 2016, pp. 1–5.

[27] Uhnák, P. Changing class prefixes [online]. 5 2016, [visited 2017-05-24]. Available from: `https://www.peteruhnak.com/blog/2016/05/01/changing-class-prefixes/`

Appendix A

Glossary

back-end See execution back-end.

bridge A component that handles all the communication between the tool itself and execution back-end.

concern An alternative name for Kendrick model part – an object that provides information to any model that integrates it.

definition diagram A definition model displayed as a diagram.

definition model A model created for the tool to serve for defining external models by a user and is displayed on left side of a screen as a diagram.

element A model object that is part of a model, but is not standalone model itself.

element link One kind of a link that contains a reference to an element from a resulting model.

execution back-end A system that provides all the infrastructure and logic required for creation of its models. One example of such back-end is Kendrick. Also called an external back-end.

external model A model that is not directly used for any diagrams in the tool and is provided by an execution back-end. It is usually an instance of a class of the execution back-end, but can be provided by the back-end even in other forms like XML.

external back-end See execution back-end.

link An element in definition model that does not have any identity on its own, but instead contains a reference to a model object from a resulting model or references the resulting model itself.

model object A model, model part, element, relationship or any other similar object that is part of internal structure of definition and resulting models.

module A model or model part used to compose another model or model part.

module link One kind of a link that contains a reference to any whole resulting model.

resulting diagram A resulting model displayed as a diagram.

resulting model A model created for the tool to serve for displaying external models on right side of a screen as a diagram.

Appendix B

Acronyms

API	Application Programming Interface.
CTU	Czech Technical University in Prague.
DSL	Domain specific language.
FIT	Faculty of Information Technology.
GUI	Graphical user interface.
MVC	Model-view-controller.
UI	User interface.
UML	Unified Modeling Language.
XML	Extensible Markup Language.

Appendix **C**

Contents of CD

Enclosed CD contains

- LaTeX text file and other related source files,
- the thesis in PDF format,
- Pharo 6 image with implementation executable by Pharo virtual machine,
- complete executable Pharo 6 environment with the implementation and
- documentation and user testing scenarios.

- `readme.txt` .... the file with CD contents description
- `impl` ... the directory of implementation source codes and executable files
  - `doc` .... the directory of documentation
  - `exe` the directory of complete Pharo 6 platform with the implementation
    - `linux` .... the directory of complete Pharo 6 platform with the implementation executable on most Linux distributions
    - `osx` .... the directory of complete Pharo 6 platform with the implementation executable on OS X
    - `win` .... the directory of complete Pharo 6 platform with the implementation executable on Windows
  - `img` the directory of standalone Pharo 6 image with the implementation
  - `scenarios` .... the directory of user testing scenarios
- `src` .... the directory of LaTeX source codes of the thesis
  - `assignment.pdf` .... assignment of the thesis in pdf format
  - `MT_Bliznicenko_Jan_2017.bib` used bibliographical sources in BibTeX format
  - `MT_Bliznicenko_Jan_2017.tex` the LaTeX source code files of the thesis
- `text` .... the directory with text of the thesis
  - `MT_Bliznicenko_Jan_2017.pdf` ..... text of the thesis in PDF format