NTNU
Norwegian University of
Science and Technology

Jigsaw EMF Editor

Oddvar Hungnes

Master of Science in Computer Science
Submission date: February 2016
Supervisor: Hallvard Trætteberg, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Jigsaw EMF Editor

Oddvar Hungnes

Supervisor: Hallvard Trætteberg

***Abstract*—In our previous paper, Jigsaw Language Toolkit [1], we have shown that the jigsaw puzzle metaphor can be used to visualize syntactical constraints in languages with complex type systems and static type checking.**

There are several successful jigsaw-based programming languages, but we believe that the jigsaw-based visual syntax can also be successfully applied to general-purpose modeling.

In this paper, we present the Jigsaw EMF Editor, which is a jigsaw-based visual editor for the Eclipse Modeling Framework. We subject the editor to both user testing and evaluation against a selection of existing models. This helps us learn more about the strengths of the jigsaw-based visual syntax, and what kinds of models it is well-suited for.

I. Introduction

In the Jigsaw Language Toolkit paper, [1] we implemented a programming toolkit in Java [37] for creating visual domain specific languages based on the jigsaw puzzle metaphor. The central aspect of the jigsaw puzzle metaphor, as popularized by the Scratch programming system, [43] is interaction with blocks that snap together in a manner similar to how a jigsaw puzzle is assembled. Each block has a number of protrusions and depressions along its border, named tabs and slots, respectively. Figure 1 illustrates some of the basic terminology we have established.

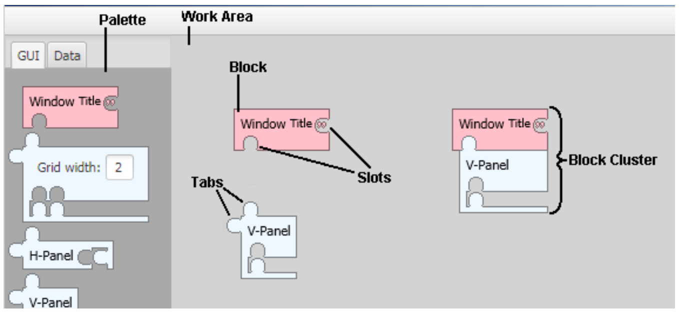


Fig. 1: The jigsaw toolkit and its terminology. Blocks can be copied from the palette and assembled into block clusters by connecting their tabs and slots.

A key feature of jigsaw-based visual languages is that the syntax restrictions of the language are explicitly visualised as the shapes of the tabs and slots on the blocks. This enables users to immediately understand what sort of connections can and cannot be made. Visual programming systems built around the jigsaw metaphor can be made very simple to use. Typically, they present a pure drag-and-drop interface, and the available programming blocks are usually available in a palette. Considering Don Norman's Design Principles, [3] this provides an outstanding presentation of constraints, visibility, feedback, consistency and affordance.

The modeling discipline is concerned with creating models of domains of interest using various modeling languages, such as UML. [51]

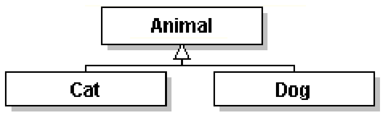


Fig. 2: An very simple UML class diagram. The language of UML class diagrams is a graph-based modeling language; Classes and their relationships are expressed using nodes and edges.

UML aims to unify the modeling practice with a general purpose visual modeling language. However, modeling can be applied to many problems, and it is unreasonable to assume that there exists a one-size-fits-all solution to the modeling problem. Sometimes a domain-specific language(DSL) is necessary. [33]

Graph-based languages such as the UML class diagram are widely used among professional developers, but other kinds are gaining popularity among novices. We believe that it is worthwhile to look into other types of visual languages as well. Jigsaw-based languages perform well as programming languages, but they are also expressive enough to be applied to general-purpose modeling.

Our goal is to make visual languages based on the jigsaw metaphor available to the modeling discipline, and conduct some initial research into what kinds of models jigsaw-based languages may be well-suited to. In this paper, we describe the design, implementation and evaluation of a general-purpose jigsaw-based DSL editor for EMF models.

The structure of the paper is as follows.

- Related work - we study the state-of-the-art in both the jigsaw-based visual programming discipline and the modeling discipline.

- Methodology - we specify the requirements for the editor and how it will be evaluated.
- Design - we describe the conceptual problems and solutions we find when adapting the jigsaw-based syntax to the abstract models.
- Implementation - we describe key features of our software implementation, and some of the challenges we face.
- Results - we describe the results of evaluating the editor.
- Discussion - we discuss our results, including benefits and drawbacks of the editor compared to other model editors.
- Conclusion - we briefly sum up our most important achievements and further work.

II. Related Work

In this section, we review existing literature and software pertaining to data modeling and the jigsaw puzzle metaphor, establishing a conceptual framework.

Much of our focus will be on the Eclipse Modeling Framework(EMF) in particular. EMF has a living community of developers, modelers, and users, making it a good platform for our editor.

A. Domain-Specific Languages

DSLs are formal languages developed for a particular domain. They are useful in many situations, but some of the main points [53] are:

- The DSL can serve as a formal language for more precise communication with domain experts.
- A DSL is typically easier to learn and simpler to read and write than a programming language or another general purpose language such as UML [51] or XML. [52] This allows people who are not programming experts to participate more in the production of a project.
- A well-designed DSL can help improve productivity by being a more efficient language for its task than a general purpose language.
- DSLs can help provide runtime flexibility in a software project. Configurations and behaviours can be expressed using the DSL instead of being hard-coded.

A DSL can be said to consist of an abstract syntax, semantics, and one or more concrete syntaxes. [54]

1) Abstract Syntax and Semantics: An abstract syntax defines the structure of the data that the language is able to represent. It is purely structural; it does not specify what its representation might look like in the language. [55] As an example, the abstract syntax for a programming language typically defines the structure of an abstract syntax tree. In modeling languages, the abstract syntax is typically given in the form of an abstract syntax graph definition known as a *metamodel.* [54]

For clarity, the term *instance* is sometimes used to denote models, to separate them from their metamodels. [64]

The abstract syntax is supplemented with a specification of language semantics, which defines what the data should be interpreted to mean, or how it should be used. For example, the semantics of an interpreted programming language is concerned with how an interpreter should execute the statements and evaluate the expressions contained in the abstract syntax tree. [55]

2) Concrete Syntax: A concrete syntax defines what the representation of data looks like in the given language. Multiple concrete syntaxes can exist for the same abstract syntax and semantics. For example, even if the following expressions are use different concrete syntaxes, they represent the exact same mathematical idea: [55]

- English: The sum of twenty and fourteen
- Reverse Polish Notation: 20 14 +

A concrete syntax may also be visual in nature, or pertain to any other modality. Traditionally, the following categories of concrete syntaxes are commonly used:

- Textual syntax, where the language is expressed using linear sequences of symbols. Typically, the symbols are characters from the Latin alphabet.
- Graph-like syntax, where the language is expressed using visual symbols and links, [56] representing the nodes and edges of a graph.
- Syntaxes using other conventional GUI widgets such as tree views, [57] lists, tables and forms. [58]

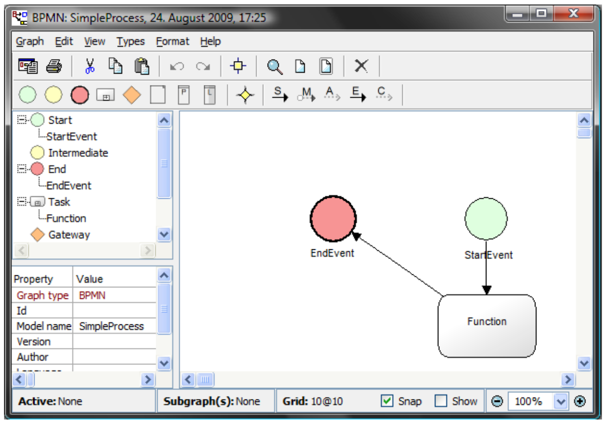
A mapping between the concrete syntax and abstract syntax is required in order to keep the separate representations in sync. In the context of modeling languages, the set of rules involved in this mapping is called the mapping model. [59]

The mapping can be defined in either or both directions. For example, a textual programming language has a parser which translates the textual representation into an abstract syntax tree. However, there is usually no need to go the other way, so the reverse mapping may be omitted.

In programming language design, there are established techniques and tools which work well and make the process vastly more structured and efficient. [60] It is reasonable to suggest that similar techniques and tools should be used in the design of DSLs, rather than building the entire framework from the ground up.

B. MetaEdit+

MetaEdit is a framework for creating domain-specific languages and tools. It was initially released in the early 1990s [61], and has evolved into the more modern MetaEdit+, which is maintained by MetaCase. MetaEdit+ has the features of a modeling framework, enabling developers to create metamodels, as well as integrated modeling tools for creating graphical DSLs and working with model instances. [63] A screenshot is shown in Figure 3.



Property	Value
Graph type	BPMN
Id	
Model name	SimpleProcess
Version	
Author	

Fig. 3: A screenshot of the MetaEdit+ modeler. [65]

Metamodels in MetaEdit+ are defined using graphs, objects, properties, ports, roles and relationships, constituting the GOPPRR modeling language. [66]. Thus, the metamodel defines a graph-based language. A concrete syntax can be automatically created using default symbols for each element. However, developers are able to replace symbols with customized graphics or text.

MetaEdit+ offers facilities for model transformations, placing constraints on models, and generating source code. Code generation can be configured using the MERL language, which MetaCase claims to be expressive enough to target any programming language.

SOAP [67] can be used for generic communication with the models, and they can be serialized using standard XML.

C. Generic Modeling Environment

The Generic Modeling Environment (GME) is a configurable toolset which supports the creation of graphical domain-specific modeling environments. [62] These modeling environments are generated through interpretation of domain-specific metamodels. GME is component-based. The environments are assembled from a set of components, including user interface elements, interpreters and model storage facilities. Developers may develop their own add-on components without modifying GME itself.

GME metamodels are defined using a complex set of modeling concepts, focusing on the decomposition of models into sub-models. A GME metamodel can be divided into multiple abstraction layers. For example, when working with electronic circuits, there might be one diagram editor for working on the logic gate level and one for working on the transistor level.

Like MetaEdit+, GME maps each element in the metamodel to a graphical element, e.g. a node with a given shape, or some user interface widget like a text field.

Contraints in GME are supported through a predicate expression language derived from OCL. [15] Models can be serialized to a proprietary binary format, a database format based on MS SQL, [16] or the developer can define their own model storage add-on.

D. Eclipse Modeling Framework

Most famously, Eclipse is known for being a powerful integrated development environment(IDE) for Java programming. The Eclipse Platform is, however, a fully generic framework for building IDEs. Through its plugin-based architecture, it is possible to implement an IDE for virtually anything. [4] The Eclipse Modeling Framework(EMF) resides at the core of the Eclipse Modeling Project [10], which seeks to facilitate model-driven development in the context of Eclipse. Despite the name, however, EMF can be used separately from the Eclipse Platform.

Ecore is the metamodel provided by EMF. It is based on the EMOF standard, [12] which is essentially a subset of the UML class diagram, which has a good balance between simplicity and expressiveness.

Ecore supports *packages*, *classes*, *features*, *operations*, *references* and *attributes*. Classes form a hierarchy through inheritance [19].

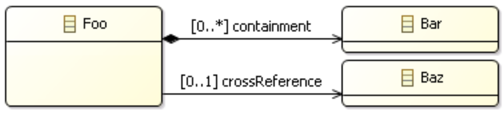


Fig. 4: Containment reference (line with black diamond and arrow) and a cross reference (line with arrow) in the EcoreTools [5] graphical editor.

Containment references (Figure 4) maintain a special role in EMF. Containment references are equivalent to composition associations in UML. A model is expected to form a *containment tree*, often with a single root.

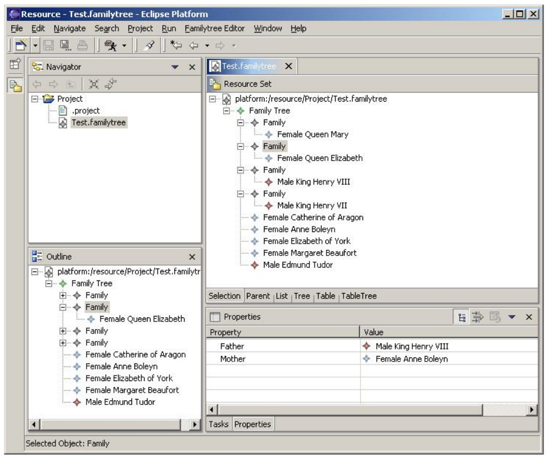
Names in Ecore are typically stylized with an 'E' prefix, e.g. EClass, EObject, and so forth.

EMF supports code generation, primarily targeting the Java programming language through the EMF.Codegen framework. The Java code generation is somewhat configurable through *genmodel* files, but the overall structure of the resulting code is always the same. The generated code exposes both a concrete Java API and a set of meta-objects, such as EMF package and class primitives, as well as object factories. The meta-objects enable efficient use of the reflective API, allowing libraries and applications such as editors to work with models in a completely generic way. Furthermore, a developer can choose to modify the generated code by hand, and, with proper care, they can ensure that the code generator will not conflict with their changes if the metamodel needs to be updated later in the development process.

As an alternative to generating code, developers can choose to employ Dynamic EMF [11], which is essentially a generic implementation of the reflective API. This allows an application to create models with no generated code, which is helpful in cases where code generation would be cumbersome or unnecessary. The tradeoff is that no hand-written code modifications are possible, and that dynamic instances cannot provide the same compile-time guarantees or type safety as generated code. However, since they share the same reflective API, the application can be written to be agnostic about whether the instance is dynamic or uses generated code.

By default, EMF includes support for serializing models using the XML Metadata Interchange(XMI) [13] format. XMI, by design, only includes the abstract model, e.g. expressions in the abstract syntax. Hence, if EMF is used to implement a DSL, then a separate concrete model is often used to store expressions in the concrete syntax. The concrete model typically contains information such as the locations of boxes in a graph-based language, and is sometimes simply called the *view* or the *diagram*. [25] This separation adds complexity, but it has the advantage that the same abstract model can be shared between multiple editors, even with different concrete syntaxes. In practice, this can be somewhat finicky, especially if the different views overlap. An editor may need to adapt or completely regenerate its concrete model if the shared abstract model is modified by another editor. By necessity, any such adaptation algorithm involves some amount of guesswork.

EMF supports generics: Model classes can have type parameters, which serve as placeholders for a concrete type. The type parameters may be replaced by concrete types, using type arguments, in a subclass or when an instance of the model class is created. This can affect the types of references and attributes on that instance. Generics in EMF are implemented in accordance with how generic programming is implemented in the Java programming language. [37] This means that, while type arguments introduced by a subclass are reifiable, type arguments introduced in the model instance are non-reifiable. [38] Non-reifiable types are useful in the generated API, but they are not included in the abstract model.



Property	Value
Father	Male King Henry VIII
Mother	Female Anne Boleyn

Fig. 5: A screenshot of an EMF tree editor. [6]

EMF supports form-based DSLs using trees, tables and property sheets. The tree editor is shown in Figure 5. A customized tree editor can be generated along with the model code, enjoying the use of generated model classes and the customizations described in the genmodel file. The tree editor can also be used to work with Ecore models without generated code, in a form known as the *reflective editor*, [24] which employs Dynamic EMF and the reflective API. The advantage of the reflective editor is that it can be deployed more quickly than the generated editor.

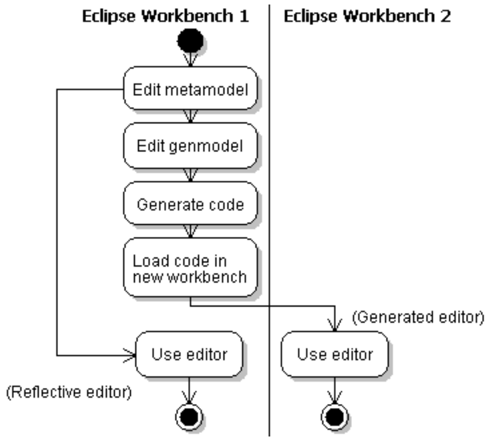


Fig. 6: The activities involved in deployment of reflective and generated EMF tree editors. Note the shorter deployment path of the reflective editor, and that the generated editor must be loaded into a new Eclipse workbench before use.

Often, the tree editor is unsuitable or insufficient. The EMF.Edit framework allows the reuse of many features from the tree editor, such as label providers, table cell factories and the command framework. The latter may significantly simplify the implementation of undo and redo functionality. EMF is not a complete DSL development framework, but it does provides the foundation for interoperability and code reuse between complex DSL editors. EMF-based DSL development frameworks usually treat the Ecore model as an abstract syntax definition and specify a separate model to be used as the concrete syntax.

E. EMFText

EMFText is a framework for developing textual DSL editors (Figure 7) for EMF models. The grammar is written the Concrete Syntax Specification Language(CS), [26] which is based on EBNF. [29]

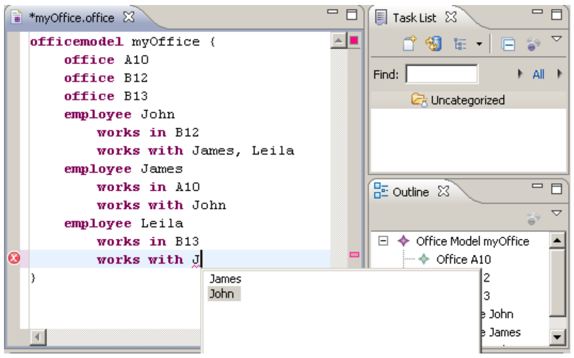
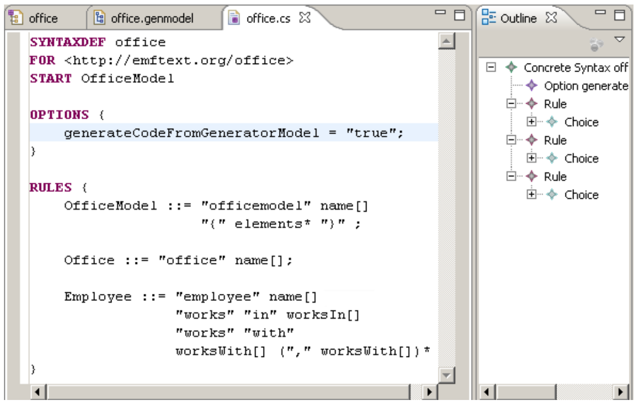

```
officemodel myOffice {
    office A10
    office B12
    office B13
    employee John
        works in B12
        works with James, Leila
    employee James
        works in A10
        works with John
    employee Leila
        works in B13
        works with J
}
```

Fig. 7: A screenshot of an EMFText editor. [28]

The bulk of a CS file consists of a set of rules, each corresponding to a model class. CS is able to express any context-free grammar, as well as the Java programming language itself. EMFText provides a large number of sample grammars in the *Concrete Syntax Zoo.* [27]

The CS editor (Figure 8) can take an Ecore model as input, thus providing auto-completion and validation for the grammar. An EMFText editor can then be generated from the CS file. The generated EMFText editor requires that EMF model code has been generated in order to successfully compile, but it is possible to add an option to the CS file that results in EMF model code being automatically generated. The generated editor can then be loaded into a new instance of the Eclipse workbench as a plugin, or it can be embedded in a different application.


```
SYNTAXDEF office
FOR <http://emftext.org/office>
START OfficeModel

OPTIONS {
    generateCodeFromGeneratorModel = "true";
}

RULES {
    OfficeModel ::= "officemodel" name[]
                    "{" elements* "}" ;

    Office ::= "office" name[];

    Employee ::= "employee" name[]
                 "works" "in" worksIn[]
                 "works" "with"
                 worksWith[] ("," worksWith[])*
}
```

Fig. 8: A screenshot of an EMFText CS grammar. [28]

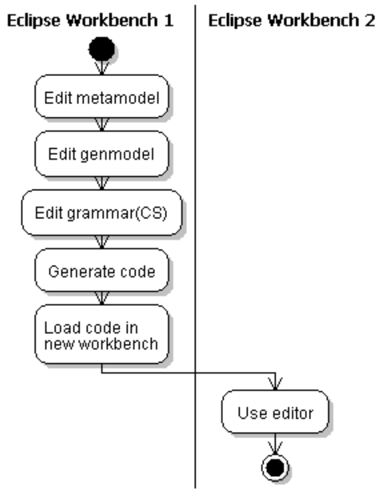


Fig. 9: The activities involved in deployment of EMFText editors.

The CS file is used to generate a two-way mapping between the concrete syntax and the abstract syntax, where the abstract syntax is given as an EMF model. The two directions of transformation are handled separately: The *parser* handles the text-to-model transformation, while the *printer* handles the model-to-text transformation.

In this approach, if the user makes an edit to the text, then the only obvious way to synchronize it with the abstract model is to discard the model and run the parser again on the entire text. The same is true for the other direction.

Although optimizations are possible, this is the natural way to handle the mapping for text-based DSLs, and the two-way mapping in EMFText enables interoperability with other EMF editors. Of course, if the grammar allows for multiple ways to express the same thing, then information may be lost and the exact original representation cannot be faithfully reproduced by the printer. As a trivial example, the exact number of whitespace characters used to separate two lexemes is not reproducible, but it is possible to provide hints as to how the printer should handle such choices.

F. XText

XText is another framework for developing textual DSL editors. [30] Like EMFText, XText uses an EBNF-like grammar language. A key difference, however, is that it does not require an existing Ecore model. Instead, the syntax tree of the parser is generated as an Ecore model and can be used directly as the metamodel of the language. It is also possible to automatically generate a default grammar from an existing Ecore model, although the default grammar will normally need to be hand-edited.

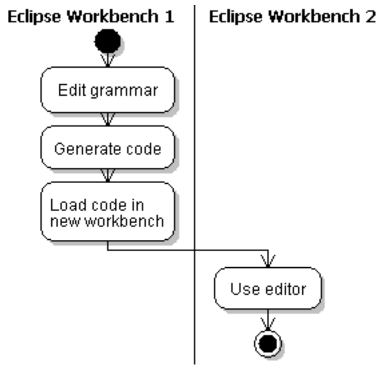


Fig. 10: The activities involved in deployment of XText editors. Note that it is also possible to use a separate metamodel, making the deployment more similar to that of EMFText (Figure 9).

Like EMFText, XText editors can be embedded in non-Eclipse applications. It also offers web browser support and IntelliJ IDEA [20] integration.

XText is a popular framework which enjoys widespread use. It is natural to assume that this is, at least in part, due to the flexibility and relative ease of deploying XText editors.

G. Sirius

Sirius is a framework for creating graphical DSL editors for EMF models, designed to run in Eclipse (Figure 11). It is built on the Graphical Modeling Framework(GMF) [31], which in turn leverages the Graphical Editing Framework(GEF) [32] and EMF. Knowledge of GMF and GEF are optional since they are normally hidden under a layer of abstraction. Sirius supports three *dialects* out of the box: Diagrams, tables and trees. However, it is possible to create new dialects as extensions. [34] It is also possible to create multiple viewpoints into the same EMF model, so that a single model can be edited using different Sirius-based editors.

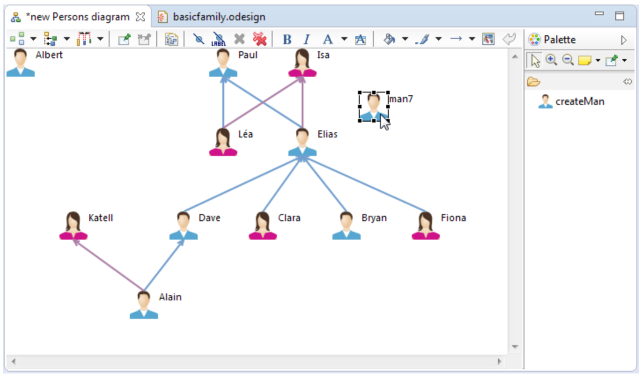


Fig. 11: A screenshot of a Sirius diagram editor. [35]

A Sirius workbench is defined using a viewpoint specification project(VSP), [35] which takes an Ecore model as input and serves as its mapping model (Figure 12). A viewpoint specification is a hierarchical structure which provides a large amount of flexibility. For example, a diagram viewpoint specification contains nodes and edges, which typically represent model objects and their relations. The modeler is expected to assign a style to each node, which defines its visual characteristics such as the shape and color.

The mapping from the model elements to the nodes and edges is implemented using a query language such as Acceleo. [36] This allows the concrete model to differ significantly from the abstract model.

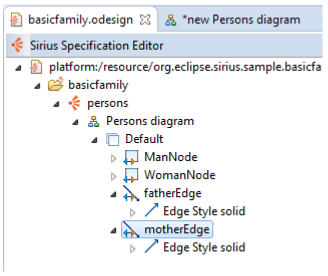

basicfamily.odesign
*new Persons diagram
Sirius Specification Editor
platform:/resource/org.eclipse.sirius.sample.basicfa
basicfamily
persons
Persons diagram
Default
ManNode
WomanNode
fatherEdge
Edge Style solid
motherEdge
Edge Style solid

Fig. 12: A screenshot of a partial Sirius viewpoint specification model. [35]

Sirius viewpoint specifications are interpreted; [35] no editor code needs to be generated, and viewpoint specification files can be reloaded in a running Sirius workbench. However, the editor does require that model code has been generated and loaded into the Eclipse workbench. Furthermore, developers are expected to build the viewpoint specifications from scratch, which can be a large amount of work.

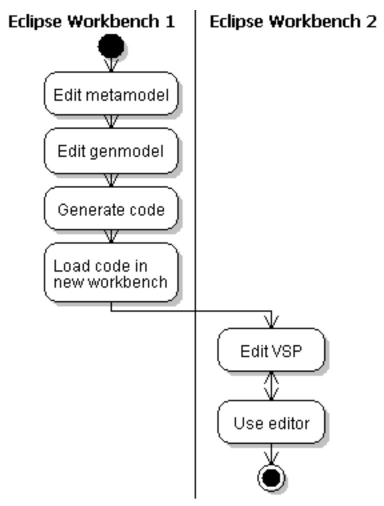


Fig. 13: The activities involved in deployment of Sirius editors. The viewpoint specification project can be edited in the same Eclipse instance as the running editor, even while the Sirius editor is in use.

H. Jigsaw-based languages

Visual languages based on the jigsaw puzzle metaphor are characterized by blocks that snap together in a manner similar to how a jigsaw puzzle is assembled. The blocks can be moved around using drag and drop, and these languages are excellent at visualizing the constraints of the language; The user can easily tell which blocks are available from a palette and how they may be assembled by inspecting their shapes.

1) Scratch: Scratch [43] is a visual programming system based on the jigsaw puzzle metaphor for developing simple digital games and animated art. It is popular among novice users and has seen success as an educational tool. [44] Scratch is largely responsible for popularizing the jigsaw puzzle metaphor, although it was inspired by the older Starlogo. [45] A screenshot of Scratch can be seen in Figure 14.

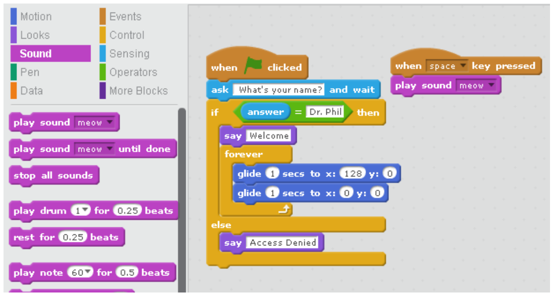


Fig. 14: A screenshot of the programming tab in Scratch.

The Scratch community has created a substantial number of modifications [46], including projects like Web Blox [48], which reuses the visual syntax of Scratch for other languages. However, Scratch has a number of limitations that make its syntax unsuitable as a general purpose visual syntax for DSLs. It only has a few basic types of blocks [47], so it cannot visualize the constraints in a language with a complex type system. The blocks can only participate in containment structures; there is no proper support for cross references. There is little flexibility in how a block can be structured, so if an a block participates in many different associations with other blocks, then the layout becomes awkward very quickly.

2) Google Blockly: Google Blockly, seen in Figure 15, is a framework designed for developing editors for visual programming languages based on the jigsaw puzzle metaphor. MIT App Inventor [49] is one example which uses Google Blockly to define the logic of Android applications. However, there are also projects that use it for domains other than programming, such as BlocksCAD. [50] The visual syntax of Google Blockly is very similar to Scratch, although it does introduce a few modifications. The block layout is slightly more flexible, but the type system is even simpler, with only statement blocks and expression blocks available. Again, it is unsuitable as a general framework for DSLs.

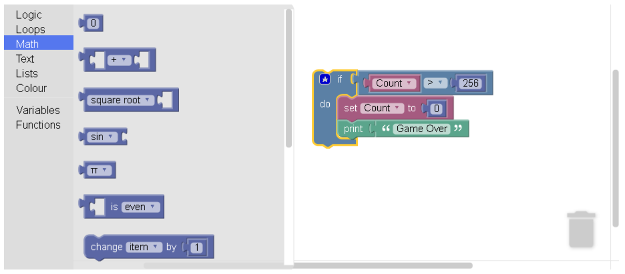


Fig. 15: A screenshot of the Google Blockly demo application.

3) Jigsaw Toolkit: The Jigsaw Toolkit (Figure 16) is a framework for implementing editors for DSLs based on the jigsaw puzzle metaphor. [1] It extends the expressiveness of jigsaw-based languages in a way that offers support for complex type systems, multiple references to a single block, generics, and a relatively flexible layout.

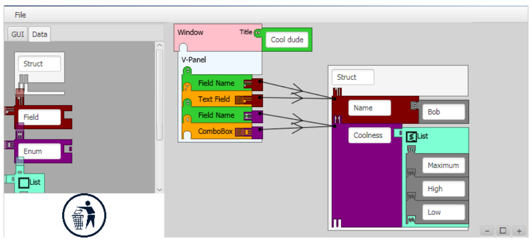


Fig. 16: A screenshot of the Jigsaw Toolkit window builder demo application. The left column contains a *palette* split into two tabs. The right column contains a *work area.*

The Jigsaw Toolkit visualises the type system using the shapes of tabs and slots. Each type has a particular shape associated with it. A slot and a tab with the same type are displayed as inverted images of each other.

A block may have a tab in its upper-left corner, which symbolises the type of the block itself. It may also have any number of slots along its down- and right-facing edges, whose shapes determine what types of blocks may be connected to them. A block can be assumed to fit into a slot that faces either downward or rightward, meaning the orientation of its tab will be different depending on where it is connected. When it is not connected to any block, the tab orientation is indeterminate. By default, this case is actually represented as two tabs with different orientations, although only one is usable at a time. This is illustrated in Figure 17.

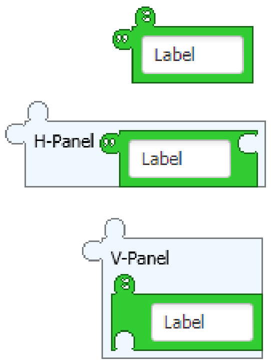


Fig. 17: Blocks display two tabs until they are connected to a slot.

The shape of any particular type inherits the *cuts* from the shapes of the supertypes, as illustrated in Figure 18. This is intended to communicate that a block with the subtype can be inserted in a slot with the supertype, but not the other way around.

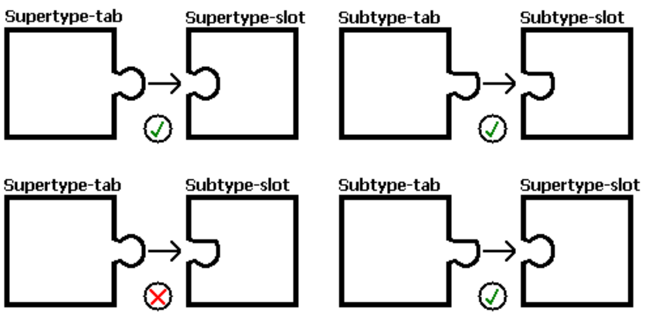


Fig. 18: The shapes of the tabs and slots are constructed to conform to how subtype relationships work. Note how a subtype tab can fit into a supertype slot, even if the supertype tab doesn't fit into the subtype slot: Their borders would overlap.

Blocks and slots may have parameterized types (Figure 19). This enables applications to make use of generic type checking, [42] although the functionality is somewhat limited. Importantly, a tab and a slot must have the same set of type parameters in order to fit together visually. In EMF, however, a subtype can define a different set of type parameters than its supertype, making them visually incompatible. However, we assume that in most practical cases, this will not be a major concern.

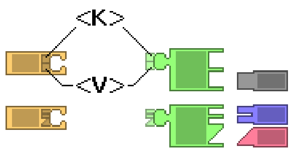


Fig. 19: How a generic type with two type parameters <K> and <V> would be represented in the Jigsaw Toolkit. The shapes associated with the type arguments are rendered as add-ons to the raw type's shape.

Type arguments can be assigned by dragging *type chips* onto *concrete type boxes*, as seen in Figure 20. A limitation of this is that the concrete type cannot itself be a parameterized type. It must be a simple type. To assign parameterized type arguments, a more sophisticated user interface element will certainly be needed. However, we again assume that this will not be a major concern in most cases.

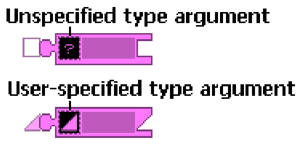


Fig. 20: A type argument can be provided by dragging a type chip onto the black concrete type box. The type chip for a given type can be obtained from any block of that type, and is shaped like a block's tab. When no type argument is specified, the type of the slot becomes the upper bound of the type parameter.

Cross references can be created by using a special type of block called a *pointer block*, illustrated in Figure 21. The pointer block holds a reference to a particular block and can be substituted for that block in any slot. In other jigsaw-based visual languages, blocks can only be referenced by a single container block.

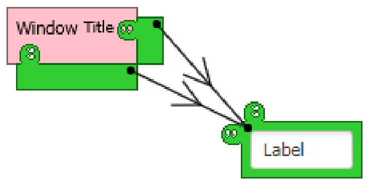


Fig. 21: Pointer blocks in the Jigsaw Toolkit. The pointer blocks have pointer arrows pointing to the referenced block. It is possible to disable the pointer arrows or install application-defined GUI widgets on the pointer blocks.

III. Methodology

In this section, we describe how our work is evaluated.

A. Requirements

Our editor must conform to a set of requirements, which are listed at the end of this section. In the first part, we explain the choices we make in order to arrive at the given set of requirements.

The editor has different stakeholders, which can be categorized as follows:

- Developers - People who write and modify code for the editor. This includes us and eventual future developers.
- Modelers - People who configure the editor for a given domain.
- End users - People who use the editor to build model instances.

The different stakeholders have different needs, and these should be accommodated as best as possible within the constraints of the project.

The editor can be seen as a mapping mechanism between an abstract EMF model and a jigsaw-based concrete model. The Jigsaw Toolkit is used to implement the concrete syntax.

For textual editors, we have seen that the mapping mechanism is implemented as a parser and printer. Typically, the parser is run periodically, reconstructing the entire abstract model. The abstract model is ephemeral and can be discarded before re-parsing, and only the concrete model is stored on disk. For graph-based editors, the mapping is continuous, so that the two models are always in sync. Typically, both models are explicitly stored with links between them.

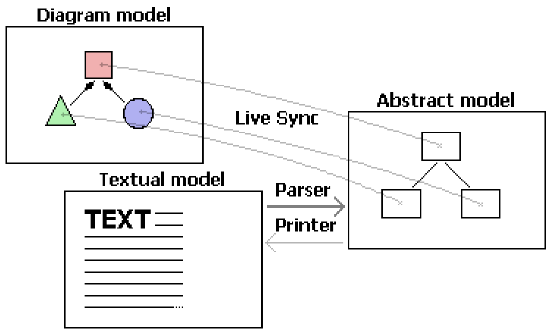


Fig. 22: For textual editors such as EMFText, it is common to first perform an edit, then run the parser (or printer) to rebuild the abstract (or concrete) model. For graph-based editors such as Sirius, it is more natural to continuously keep the models in sync.

It is natural for our editor to follow the established approach for graph-like editors such as Sirius, since it shares more traits with them than with textual editors. The editor should continuously keep the blocks in sync with the EMF model and vica versa.

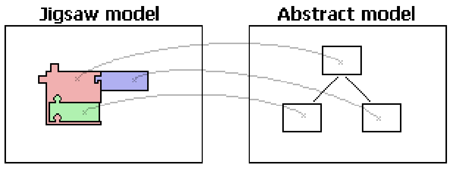


Fig. 23: Our editor should perform live transformations to always keep the two models in sync.

Modelers will expect that the mapping is somewhat flexible, and that the editor can be configured for their domain. To facilitate this, we intend to use a mapping model (Figure 24). Broadly speaking, for graphical DSLs, we have seen two approaches. The first approach is to let each element in the abstract syntax map to some graphical symbol, as in MetaEdit+ or GME. The second approach is to define a more detached graphical structure, which maps to elements in the abstract syntax using queries, as in Sirius. The Sirius approach is much more flexible, as it allows the graphical structure to differ significantly from the abstract structure. However, this approach may also be more labor-intensive for both developers and modelers, since the mapping model becomes a much more complex data structure.

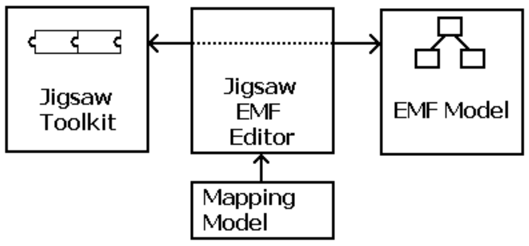


Fig. 24: The Jigsaw EMF Editor fulfills the role of connecting the Jigsaw Toolkit to an EMF model. A mapping model should be used to provide flexibility in how the mapping is performed.

One option would be to implement our editor as a jigsaw-based Sirius dialect. We decide against this for a number of reasons. Firstly, it is unknown whether this would save or cost us time within the scope of this project. A large amount of additional preliminary research would be necessary, and the additional project dependencies would weigh us down to some extent. Secondly, it would lock us to the Sirius way of doing things, which has its own shortcomings, such as not being able to work with dynamic model instances. Thirdly, while the Sirius architecture would surely provide benefits, it is unclear how much value it would actually bring.

The jigsaw-based syntax naturally lends towards representing containment hierarchies such as EMF models. Furthermore, we do not immediately see any strongly compelling reasons for creating a jigsaw-based representation which significantly deviates from the structure of the model. It is possible that this will turn out to have significant value, but it should be implemented as a response to an actual need.

Thus, our mapping model should favor simplicity, being more similar to those in MetaEdit+ and GME.

We have seen that different EMF editors require different deployment patterns, some more convenient than others. Each of the ones we have reviewed require some form of code generation before they can be used, with the exception of the reflective tree editor. Sirius is special in that is allows the metamodel to be configured on the fly, while XText makes it possible to implicitly define the metamodel through the grammar.

We believe that ease of deployment is an important quality for modelers, and that it is worth taking appropriate steps to ensure that the editor can be deployed in a way that is convenient for them. The editor should be able to work both with and without generated model code, like the EMF tree editor. As with Sirius, The modeler should be able to edit the mapping model while the editor is running, but the editor should also be able to generate a complete default mapping model.

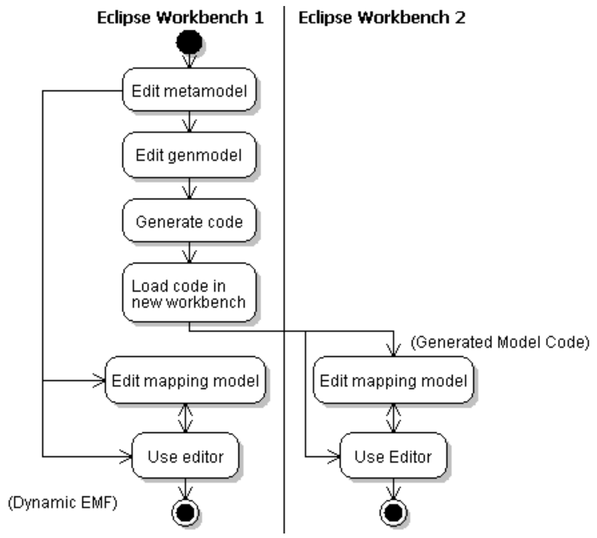


Fig. 25: It is possible to provide a relatively high amount of flexibility in the deployment pattern by making the right design decisions. The editor should be able to use models with and without generated model code. A default mapping model should be generated, so that creating your own mapping model from scratch is not necessary. It should be possible to edit the mapping model while the editor is running.

1) EMF Integration: The editor should be able to work with EMF models and offer basic interoperability with other EMF-based tools.

Requirement F 1. The editor must be able to work with Ecore models.

Requirement F 2. The editor must be able to save and load EMF model instances using the xmi format.

Requirement F 3. The editor must be able to represent root objects as blocks or work areas.

Requirement F 4. The editor must be able to represent both cross references and containment references.

Requirement F 5. The editor must support editing of object attributes.

Requirement F 6. The editor should be able to represent model elements with generic types.

2) Mapping Model: It should be possible to customize the editor using a mapping model. The mapping model should leverage the flexibility of the Jigsaw Toolkit.
Requirement F 7. The mapping model should define which model elements should be visible in the editor.
Requirement F 8. The mapping model should define where a structural feature appears on a block and how it is oriented.
Requirement F 9. The mapping model should allow for customization of labels for each class.
Requirement F 10. The mapping model should define whether objects of a given class should be represented as a block or a work area.
Requirement F 11. It should be possible to define which model elements are in the palette.
Requirement F 12. It should be possible to define a palette with multiple tabs.

3) Usability: In order to encourage adoption, the editor must be convenient to use. This applies to both the DSL development aspect and the modeling aspect.
Requirement F 13. The editor should be able to work with models with and without generated model code.
Requirement F 14. The mapping model should be initialized with useful defaults.
Requirement F 15. It should be possible to change the mapping model without requiring a restart of the editor.
Requirement F 16. The editor should be able to save and load layout information, such as the locations of blocks.
Requirement F 17. The editor should support undo and redo.

B. Model Evaluation

The editor is evaluated against a selection of EMF models. This serves two purposes. Firstly, it aids in improving the design of the editor by exposing limitations of the editor itself. Secondly, it helps us determine the characteristics of a domain that is well-suited to the jigsaw puzzle metaphor.

The following test models are used:

- A library model. [7]
- The Ecore metamodel.
- A family model. [35]
- A model of a quiz domain.

The rationale for model selection and the evaluation methodology is explained in greater detail in Section VII.

C. Survey

The editor is tested by EMF users, who are asked to fill out a survey. The survey covers multiple aspects of the project.

Firstly, the survey acts as a usability test. Secondly, the survey provides guidance on how to improve the editor, by asking the respondents about their expectations for an editor such as ours. Thirdly, the survey indicates whether the respondents believe that the editor and the jigsaw-based concrete syntax have the potential to be useful.

The survey design is explained in greater detail in Section VII-E, and the full survey is included in Appendix B.

IV. User Interface Design

In this section, we describe the solutions to conceptual problems regarding the design of the editor's user interface. Specifically, we look at how the different EMF constructs can be meaningfully represented through the Jigsaw Toolkit.

A. Object representations

An EMF model instance consists of objects which form a containment hierarchy rooted in a resource. The most straightforward approach is to let the resource be represented by a work area, and let the objects be represented as blocks.

However, in many models, the resource only contains a single root object (Figure 26). This means that with the aforementioned approach, the work area would often only contain a single block cluster. This is a problem because it strips the user of all freedom in terms of layout; the layout of a block cluster is completely automatic. It is also likely to make poor use of the available screen space.

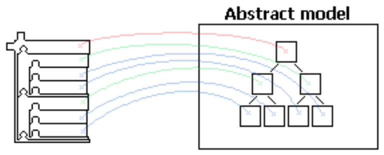


Fig. 26: How an EMF model with a single root maps to a single block cluster.

To get around this, we propose to allow root objects to optionally be represented as work areas (Figure 27). This means that a work area may either be a resource or an object. This presents a few challenges.

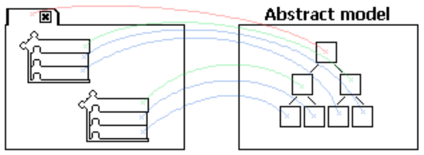


Fig. 27: Allowing the root object to be represented by a work area provides more layout freedom.

Since there can be more than one work area, and they can be assumed to take up a large amount of screen space, organizing them into tabs seems like a good idea. There is normally no need to look at more than one at the time, but there may be a need to move blocks between them. One way to facilitate this is to allow the user to bring up two work areas side by side. Furthermore, the user should be able to create and destroy work areas. Destroying them can be done by clicking on a button located on the tab. Since it is completely valid to have several classes of objects which may be represented as work areas, the user may need to be able to select which type of work area they want to create. This can be achieved by opening a drop-down menu when the user clicks on a button. The proposed user interface is illustrated in Figure 28.

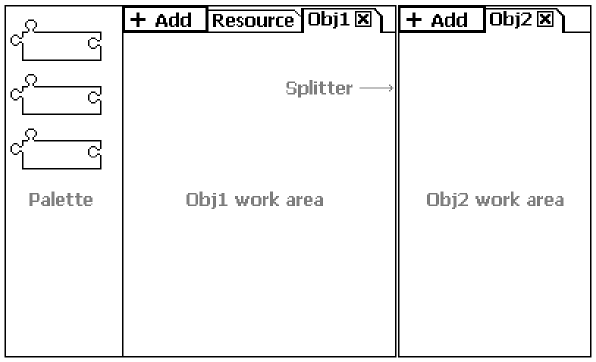


Fig. 28: A mockup of the proposed user interface used to manage work areas. The work areas are organized in tabs. The user is able to drag the tabs around, or bring them up side by side and move blocks between them. They can be created and destroyed using dedicated buttons.

If an object represented as a work area has attributes, then they may be represented in a property sheet next to the work area. However, a work area is not a block, and as such it does not have all the facilities that a block provides. In particular, it does not have a tab connector and it does not have slots. Because of this, there are certain limitations on objects represented as work areas:

- It is not possible to create a pointer block to a work area, so references to such objects cannot be represented.
- A work area only supports containment references.
- If an object represented by a work area has multiple containment references, they should not have overlapping types. If the types are separate, then it is possible to determine which reference a block should be assigned to by inspecting its type. This limitation also exists for objects in the EMF tree editor.

Also, unlike the resource, objects may not be able to contain every type of object. This means that some blocks might not be possible to place directly into an object work area. It is possible to work around all of these limitations if it becomes necessary, but it would lead to relatively complicated solutions.

Objects that are not represented as work areas, are represented as blocks.

B. Attributes and references

1) Property sheets: Object attributes may be represented as conventional property sheets. In the EMF tree editor, the property sheet appears in a separate pane next to the tree view. This is an option, but we believe that it is better to let the attributes appear on the block itself, much like the editable properties on Scratch blocks. However, in many cases this may lead to very large and unwieldy blocks. This is particularly evident in the Ecore metamodel, which we talk about in Section VII-B. To alleviate this, we propose to allow the property sheets to be expanded and collapsed by the user, freeing up screen space as necessary (Figure 29).

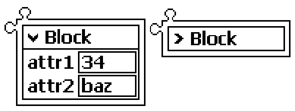


Fig. 29: Property sheets on blocks can be collapsed to save space.

2) Containment references and cross references: The test application written for the Jigsaw Toolkit does not differentiate between containment references and cross references: Pointer blocks are mutually substitutable for their referenced blocks. In EMF models, however, there is a sharp distinction between the two types of references.

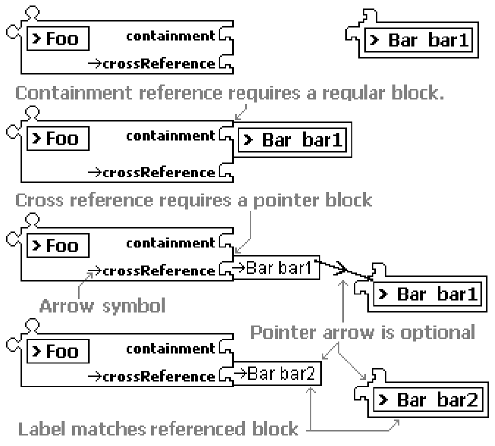
We imagine this discrepancy can be mitigated in two different ways.

The obvious solution is to set them apart using some visual symbol. Slots that require a reference block would need to appear different from slots that require an ordinary block. This effectively mandates that the jigsaw block structure must match the actual containment structure of the model.

Another possible solution would be to allow the block containment structure to differ from the model containment structure, and let the editor infer which references are actual containment and which are not. This would provide some additional freedom for the user, but it also creates some complications. Firstly, it would be somewhat difficult to implement, but more importantly, it would quickly lead to confusing situations. For example, it is illegal for an EMF object to be contained in two parents, but if the actual containment structure is obscured, then this is not immediately obvious. It would also be easily possible to create blocks that are referenced but not contained anywhere, which is an error.

For these reasons, we decide to maintain the separation between containment references and cross references. Slots that represent cross references are marked with an arrow symbol, and require a pointer block. Slots that represent containment references do not have the arrow symbol, and require a regular block (Figure 30).

In models with many cross references, the pointer arrows associated with pointer blocks can become obtrusive and make the model more difficult to read. The Jigsaw Toolkit allows the arrows to be hidden, but this leaves us with the problem of identifying which block the pointer block is referencing. As a possible solution, we propose two measures. Firstly, it should be possible to let blocks display a label that distinguishes them from other blocks of the same class, for instance by displaying a (unique) name or title. Many models include object names which are suitable for this purpose. Secondly, the pointer blocks should display the same label text as their referenced blocks. This allows user to identify what is the referenced block, even if the pointer arrow is hidden(Figure 30).



Foo
containment
→crossReference
Bar bar1
Containment reference requires a regular block.
Cross reference requires a pointer block
→Bar bar1
Arrow symbol
Pointer arrow is optional
→Bar bar2
Bar bar2
Label matches referenced block

Fig. 30: The labels for slots that represent cross references are marked with a special symbol. We initially used a transparent diamond, similar to aggregation in UML, but we later found that users preferred the arrow symbol. Pointer blocks are marked with the same label text as their referenced blocks. If the referenced block has an unique name, this allows users to identify the relationship, even without the pointer arrow.

3) Bidirectional References: In the Jigsaw Toolkit, there are no nondirectional or bidirectional associations. Both ordinary block containment and references via pointer blocks are inherently unidirectional. To represent a bidirectional references, we could create one pointer block in each direction, but this would often be rather messy and inconvenient. As an alternative to this, we can choose to represent it as an unidirectional reference, and let the other direction be implicit. This will not always work, however. For example, if both sides of the reference are parameterized types, then they both need to be represented as slots.

V. Modeling Related Design

In this section, we describe the solutions to modeling problems related to the design of the editor, seen from a modeling perspective.

The editor must be able to work with Ecore models and the associated model instances. We employ the EMF libraries to create and modify models. Thus, the editor is backed by a set of EMF models when in operation. Each object in the model instance is represented using a jigsaw object, as described in Section IV.

The EMF libraries also provide support for saving and loading XMI models, as well as undo and redo support through the EMF.Edit framework. To benefit as much as possible from this, substantial parts of the editor are modeled using EMF.

A. Layout model

The jigsaw objects contain some layout information which is not represented in the abstract model, such as the locations of blocks and the order of the work area tabs.

In order to make layout changes undoable through EMF.Edit, the layout information needs to be stored in an EMF model. We would like to maintain the separation between the abstract model and the concrete model. This is achieved this by designing a separate layout model, which is analogous to the diagram model in Sirius.

The layout model is essentially a mapping from EMF objects to *representation* objects which hold the layout information for each jigsaw object, as well as a reference to the underlying EMF object in the model instance. A diagram of the layout model can be seen in Appendix A. When saving the model instance, a layout file is saved separately, so that the layout can be recovered when loading the model later.

When loading a model instance, the layout file may be nonexistent or incomplete. If this is the case, then the editor should fill in default values for the missing layout information. This way, it ensures that it can still work with models that have been modified by other editors, although the layout may need to be manually adjusted.

B. Mapping Model

The mapping model is split into three parts:

- The slot and tab shapes
- The palette
- The jigsaw mapping model

1) Slot and tab shapes: The slot and tab shapes are managed by the Jigsaw Toolkit. The toolkit is able to generate default cuts, but these are often not very helpful. Editing them is an important step in configuring the editor for a target domain. To facilitate this, the Jigsaw Toolkit provides a dedicated image editor for slots and tabs, called the *Cut Editor*. The shapes can also be edited through almost any other image editor, since they are stored as PNG [17] image files.

The types defined in the Ecore metamodel are likely to be reused across many domains. Because of this, we have elected to create a set of default shapes for these types, which are imported as defaults when generating a new mapping model.

Also, some classes usually do not need a unique shape, so cut generation can be skipped for some types. Specifically, we assume that if, for a given type, there are no slots with that type, then the type does not need a unique shape. Since slots represent references, this is equivalent to checking if there are any EMF references in the domain which have that type. If generics are used, then it is possible to construct a scenario where this assumption does not hold. However, that is a low price to pay, since the shapes are very easy to customize.

To understand why this is normally a sensible assumption, consider a domain that contains power supplies that can be used to charge a phone. Think of the charger plug as a tab, and the charging port as a slot. The class *power supply* is an abstract class which has multiple subclasses, e.g. linear power supplies and switched mode power supplies. [22] While the implementations are certainly different, there is no case where one is *required* over the other. Thus, it would be redundant to mark the plugs differently(Figure 31).

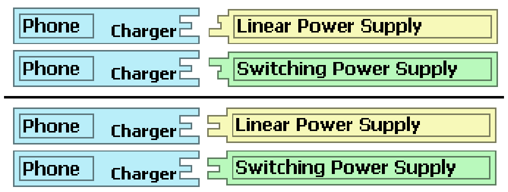


Fig. 31: We could give the two types of power supplies unique shapes, like at the top. However, there are no slots that discriminate between the two: The slot takes any power supply, so the two types do not need unique shapes. They can share the generic power supply shape.

The Scratch programming system also exploits this: In reality, it has a large number of block types, but the number of unique block shapes is very limited because there is no need to discriminate between them. Reducing the number of unique shapes is important, because as the number of unique shapes grows, their complexity increases, making them much more difficult to read.

2) Palette model: The palette model defines which blocks should appear in the palette.

At the top level, the palette model consists of one or more tabs. This allows items to be grouped together in a meaningful way, similar to how it is done in Scratch and its relatives. Each tab consists of palette items. Each palette item is used to generate a block in the palette.

There are two different types of palette items, which serve different purposes:

- *Container palette items* - contains a prototype EMF object of a particular class. This object is used to create a block representation of that object in the palette. Since the item contains a prototype object, it is possible to have multiple items for each class. Furthermore, the prototype objects may have customized attributes or even contain other objects. This makes it possible to add preconfigured objects to the palette.
- *Reference palette items* - holds a reference to an EMF object that exists in a different model. This is used to create a special type of pointer block in the palette, enabling the creation of references to objects from other models, which can be very important. For example, these items can be used to refer to the data types in the Ecore metamodel.

By default, the editor adds references to the Ecore literals to the palette. Also, when the editor loads an Ecore model and finds an instantiable class which doesn't have a representative in the palette, it will create a palette item for that class. The editor fills in missing elements wherever it finds them. However, this means that even if a modeler deletes a palette item from the generated palette model, it might reappear later. To avoid this, each palette item can be disabled, effectively making it disappear and preventing the editor from making it reappear.

Since the palette model is an EMF model, a modeler can edit it in any sufficiently powerful EMF-based editor. A diagram of the EMF model used for the palette can be found in Appendix A.

C. Generics

The Jigsaw Toolkit supports both rendering generic types and assigning type arguments. The editor should leverage this functionality. A problem with this is that type arguments introduced in the EMF model instance are non-reifiable. This means that if they are to be used, they need to be included in the concrete syntax model along with the layout information.

Thus, the layout model also contains the values of type arguments assigned to each object.

The generics support also has a few other implications. It is entirely legal to use a data type as a type argument in EMF. This can affect the type of an attribute, meaning the associated property sheet element may need to be swapped out by the editor.

But that is not all. When a type argument is assigned to a block or a slot in the Jigsaw toolkit, the shape of the type argument is rendered extruded from the primary type, as we illustrated in Figure 19. This means that data types need to have slot and tab shapes, even though there can never be a block or a slot whose type is a data type.

Furthermore, type arguments in the Jigsaw Toolkit are assigned by dragging type chips onto concrete type boxes, as we illustrated in Figure 20. Type chips are normally obtained from a block with the desired type, but there are still no blocks whose type is a data type. However, it is possible to create a pointer block to the meta-object which defines the data type. Recall that the palette can contain references to objects in other models, including the meta-model. If a pointer block references a meta-object which defines a data type in the current model, then we can expose the data type chip on that block, as shown in Figure 32.

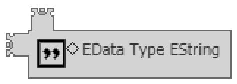


Fig. 32: A pointer block to the meta-object which defines the string data type. It exposes a type chip for the string type. Note that the type of the block is not the *string* type: The type of the block is simply the *data type* type. The *string* type is the object itself.

1) Jigsaw mapping model: The jigsaw mapping model is based on the idea that each structural concept in the Ecore model should be represented by a jigsaw object. At the top level, the mapping model contains a mapping from EMF classes work area mappings or block mappings.

A work area mapping dictates that the class should be represented as a work area. A key feature of work areas is that they need to be able to infer which reference an object should be assigned to based on the type of the object. To facilitate this, the work area mapping maps classes to references. The work area mapping also maps attribute names to form element mappings, which are used to organize the attributes into a property sheet. The property sheet is displayed above the work area, in a rectangular region which uses a grid layout. A work area form element mapping indicates a location and a relative size in this grid. In addition, it specifies the name of a factory class, which is responsible for creating the GUI for the attribute.

The default factory creates a label with the attribute name, and an appropriate GUI widget depending on the type of the attribute:

- A checkbox is created if the attribute type is a boolean type.
- A combobox with a drop-down menu is created if the attribute type is an enum type.
- A non-editable label is created if the attribute type is not a serializable data type.
- A text field is created in all other cases. The text field uses the built-in EMF serialization functionality to display and parse the value.

The default factory also takes care to swap out the GUI widget if the type of the attribute changes, i.e. because a type argument which affects the attribute is changed.

A block mapping dictates that the class should be represented as a block. Like the work area mapping, it maps attributes to form element mappings, which determine how and where the attributes are displayed on the block.

When loading an Ecore model, the editor fills in block mappings for each class that does not have a mapping. Furthermore, mappings for attributes and references are automatically filled in. Again, this means that mappings deleted by the modeler can be recreated. To avoid this, every mapping can be disabled, making it disappear from the editor. A special mapping object which is always disabled can also be created.

A modeler should be able to customize the labels for a class. The EMF.Edit label providers can be customized by specifying a *label feature*, which is an attribute whose value is inserted into the label text. In Sirius, labels can be written using a query language.

We have elected to use a variant of the format string, [18] which is stored on the work area mappings and block mappings. This allows for a fair amount of flexibility while being very simple to implement. The format string consists of two parts. The first part is a string surrounded by double quotes, which can contain an arbitrary number of *%s* tokens. The second is a comma-separated list of attribute names. The number of attribute names must be equal to the number of *%s* tokens. Each *%s* token will be replaced by the value of the attribute which is named at its respective position in the attribute name.

For example, consider a block mapping for the class Author, which has an attribute *name*. The format string is as follows:

"%s: %s", eClass, name

If the user creates an Author block and fills in *Bob* for the *name*. The label text for the block will then be *Author: Bob*. The default label text pattern is similar to what is given above.

Again, the jigsaw mapping model is an EMF model, meaning a modeler can edit it in EMF-based editors. A diagram of the EMF model used for the jigsaw mapping model can be found in Appendix A.

VI. Implementation

In this section, we describe important aspects of the implementation of the editor.

A. Overview

The editor is intended to be able to integrate with the Eclipse platform as a plugin. The Eclipse plugin APIs are well-documented, but Eclipse is implemented using the SWT GUI library, while the Jigsaw Toolkit is implemented using the JavaFX GUI library. Fortunately, interoperability between the two libraries is very simple. [39]

Despite this, we decided to implement the editor in a way that allows it to be run as a stand-alone application. This has two benefits. Firstly, it ensures that the editor is readily deployable outside of the Eclipse platform, which may be beneficial in the future. Secondly, it reduces the overhead for testing the editor during development: Since there is no need to load the plugin in a new Eclipse workbench for each test run, the time wasted on loading times is significantly reduced.

Eclipse plugins are able to benefit from the user interface of the Eclipse workbench, such as menu bars and hotkeys. The stand-alone version of the editor needs to implement its own window, menu bar and layout in order to function. Since these UI elements will be obsolete in the plugin version of the editor, they are kept logically separate from the core editor UI in the implementation. This prevents them from impinging on the development of the plugin. The separation is illustrated in Figure 33.

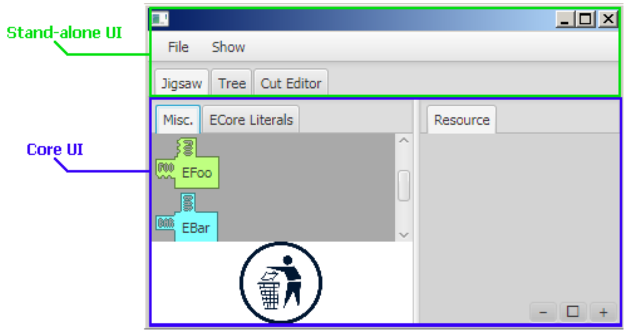


Fig. 33: A screenshot of the stand-alone editor. The stand-alone editor UI, including the window, is logically separate from the core editor UI.

The stand-alone editor provides 3 different views. The first view is the jigsaw editor itself, which contains the palettes and work areas. The two remaining views contain a simple tree editor and the Cut Editor.

The Cut Editor is reused from the Jigsaw Toolkit (Figure 34). It facilitates efficient editing of the shapes of the slots and tabs in the model domain.

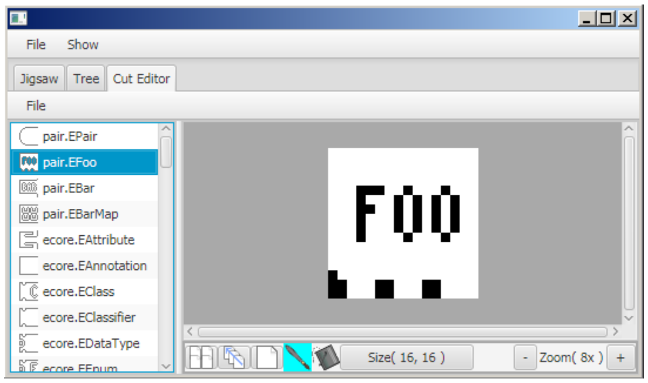
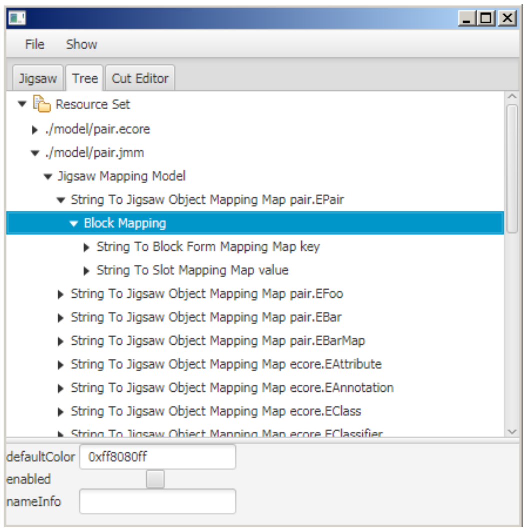


Fig. 34: The Cut Editor is an integrated editor for slot and tab shapes.

The tree editor (Figure 35) employs a JavaFX implementation of the user interface adapters from the EMF.Edit framework [41] provided by the e(fx)clipse project. [40] It provides access to editing and inspection of all EMF resources used by the editor. This is useful for debugging purposes and makes it easier to edit the mapping model while the editor is running.



Resource Set
./model/pair.ecore
./model/pair.jmm
Jigsaw Mapping Model
String To Jigsaw Object Mapping Map pair.EPair
Block Mapping
String To Block Form Mapping Map key
String To Slot Mapping Map value
String To Jigsaw Object Mapping Map pair.EFoo
String To Jigsaw Object Mapping Map pair.EBar
String To Jigsaw Object Mapping Map pair.EBarMap
String To Jigsaw Object Mapping Map ecore.EAttribute
String To Jigsaw Object Mapping Map ecore.EAnnotation
String To Jigsaw Object Mapping Map ecore.EClass
String To Jigsaw Object Mapping Map ecore.EClassifier

defaultColor	0xff8080ff
enabled	
nameInfo	

Fig. 35: The JavaFX tree editor exposes the objects in each EMF resource used by the editor.

The tree editor and the Cut Editor can be used to modify the mapping model directly. The stand-alone editor performs a full rebuild of the user interface when switching back to the jigsaw editor tab. Thus, the jigsaw objects are recreated with the updated mapping model.

B. EMF integration

Every work area and block represent some object in the EMF model. There are multiple types of work areas and blocks used for different purposes:

- ResourceWorkArea - Represents an EMF resource, which is a special facility for holding root objects. The ResourceWorkArea contains all root objects that have block representations.
- EObjectWorkArea - Represents a root object as a work area, and contains block representations of its children.
- EObjectBlock - Is the canonical block representation of an object, i.e. the one that participates in containment relationships. Since an object can only have one container, there should never be more than one EObjectBlock representing the object. This type of block is created from *container palette items*, mentioned in Section V-B2.
- EObjectPointerBlock - Is a pointer block which holds a reference to an EObjectBlock. EObjectPointerBlocks are created from EObjectBlocks.
- EProxyPointerBlock - Is a pointer block which holds a reference to an object that is contained in a another model. This type of block is created from *reference palette items*, also mentioned in Section V-B2.

Every slot represents an EMF reference, and every property sheet element represents an EMF attribute.

The editor exclusively uses the reflective API to communicate with the models. This allows it to work with both dynamic model instances and model instances with generated code.

The jigsaw objects propagate changes to their underlying EMF objects. For example, when moving a block onto a slot, the EMF object represented by the block gets assigned to the reference represented by the slot. The command stack mechanism from EMF.Edit is used to implement undo and redo. When performing changes to the underlying EMF objects, the editor uses command objects to record the changes in the command stack. The jigsaw objects have adapters registered on their respective EMF objects in order to listen to changes. This allows the jigsaw objects to respond to any changes that happen to the EMF objects when the user triggers an undo or a redo.

An EObjectPointerBlock requires a reference to an EObjectBlock before it can be properly initialized. However, when an EObjectPointerBlock is created, is not always obvious which EObjectBlock it is supposed to point to, even if the reference to the underlying EMF object is known. For example, a pointer block might be created because the value of a cross reference changes as a result of an undo event. In some cases, the order of operations may even lead to the pointer block being created before the block it is referencing. To get around this, we implemented a reference resolver mechanism, which maps EMF objects to EObjectBlock representations. When attempting to resolve a block that doesn't exist yet, a placeholder object is returned, which is updated as soon as the block becomes available.

VII. Results

In this section, we present our results from the model evaluation and survey.

During the course of this project, we have spent the greatest amount time on design and implementation in order to produce a prototype that fulfills all of the requirements. For evaluation purposes, we have made our best effort to select strategies that provide the most informative results, given the time constraints of the project.

Existing jigsaw-based visual languages are based on containment, so it seems reasonable to assume that the editor will function best for models that exclusively use containment references. However, it is not known whether a jigsaw-based language will function well for languages that predominantly use cross references. Thus, the models selected for evaluation should provide coverage in terms of the ratio of containment references to cross references.

Furthermore, the models should be relevant to real modeling problems.

The following test models are used:

- A library model, with books and authors. [7] This model uses containment references for the root object and cross references elsewhere. It is used in many tutorials, both for EMF and other DSL frameworks.
- The Ecore metamodel. This model is used to create other EMF models. The use of containment and cross references is mixed. The Ecore metamodel is of particular importance in EMF, and virtually all EMF users are familiar with it.
- A family model, with parent-children relations. [35] This model is structurally similar to the Library model, but with a greater amount of cross references. It is used in the official Sirius starter tutorial. [35]
- A model of a quiz domain. This model predominantly uses containment references. It is used in a course on model-based development at our department.

When evaluating the models, we start by opening the Ecore model for the relevant domain, e.g. *library.ecore*, in the editor. The editor will then generate a blank model instance and a default mapping model. Next, we inspect the generated blocks as found in the palette and compile a list of obvious problems that can be fixed in the mapping model. Finally, we edit the mapping model and present the result. The problems that we face underway are used as feedback into the design of the editor (Figure 36).

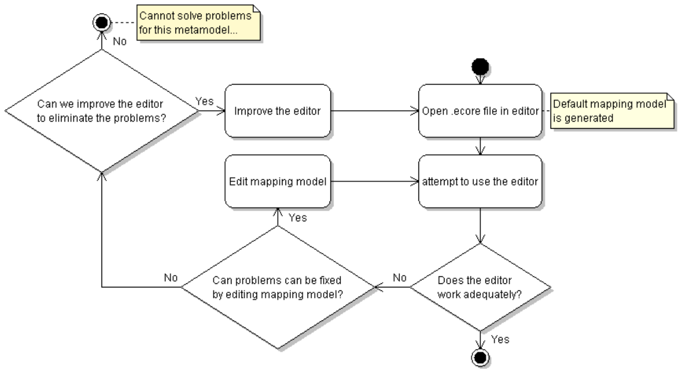

Cannot solve problems for this metamodel...
No
Can we improve the editor to eliminate the problems?
Yes
Improve the editor
Open .ecore file in editor
Default mapping model is generated
Edit mapping model
attempt to use the editor
Yes
No
Can problems can be fixed by editing mapping model?
No
Does the editor work adequately?
Yes

Fig. 36: Some problems encountered during model evaluation cannot be solved by editing the mapping model. This is used as feedback into the design and development of the editor.

A. Library model evaluation

The Library model is a very simple model which defines a library object that contains books and authors. The books and authors can be associated using a bidirectional cross reference. Thus, only the root object has containment references. The complete model is shown in Figure 37.

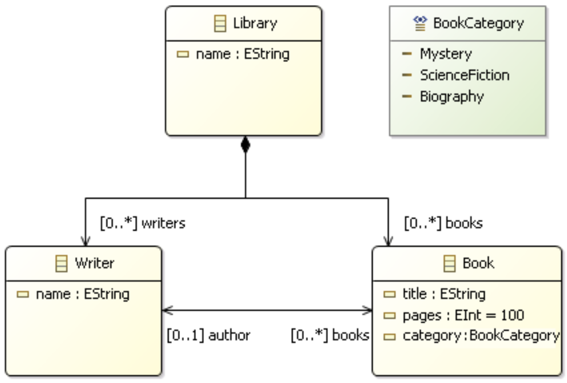


Fig. 37: UML diagram of the library model displayed using the EcoreTools [5] graphical editor.

Figure 38 shows the prototype blocks which are automatically generated into the palette for the Library model.

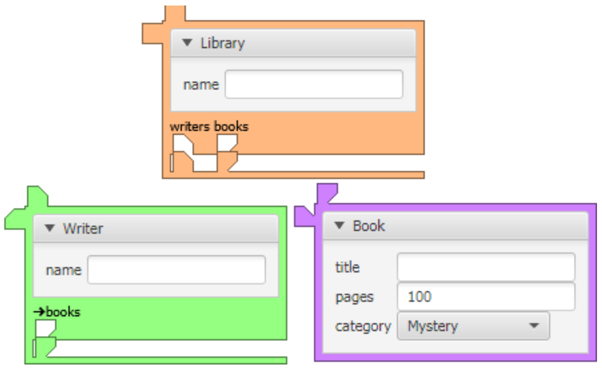


Fig. 38: The blocks from the generated Library palette. The blocks are expanded, displaying the property sheets.

Because the number of unique tab shapes is very low, the automatically generated shapes are easily distinguishable. There is little to gain by editing them.

The bidirectional writer-books association has been automatically collapsed to a single slot on the Writer object. The automatically chosen direction is based on arbitrary data, but in this case it is an acceptable default.

The model is rooted in a single object (Figure 39), making the entire model clumped up in a single block cluster. This is undesirable. In addition, the default layout of the slots may lead to blocks being stretched in an unflattering way. When we first evaluated the editor with this model, representing an object as a work area was not possible. However, the problems made it apparent that this would be an important feature.

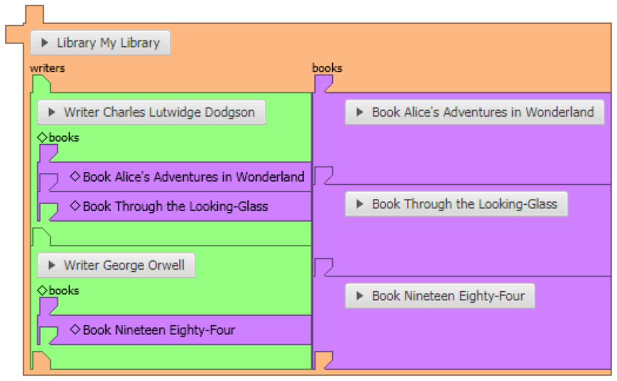
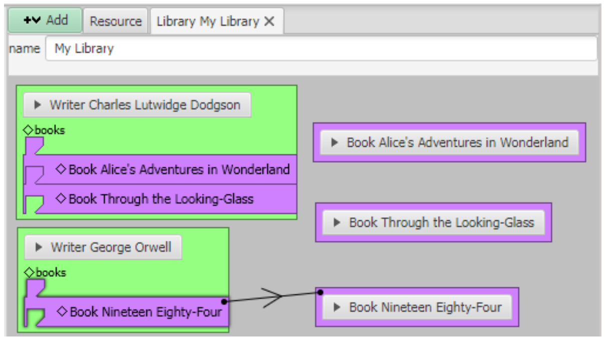

Library My Library
writers
books
Writer Charles Lutwidge Dodgson
books
Book Alice's Adventures in Wonderland
Book Through the Looking-Glass
Writer George Orwell
books
Book Nineteen Eighty-Four
Book Alice's Adventures in Wonderland
Book Through the Looking-Glass
Book Nineteen Eighty-Four

Fig. 39: An example Library model instance. The model is rooted in a single Library block. The Book blocks are stretched to fit the height of the left column, making them larger than necessary.

Figure 40 shows the result of replacing the Library block mapping with a work area mapping. This fixes the aforementioned problems.



Add
Resource
Library My Library
name
My Library
Writer Charles Lutwidge Dodgson
books
Book Alice's Adventures in Wonderland
Book Through the Looking-Glass
Book Alice's Adventures in Wonderland
Book Through the Looking-Glass
Writer George Orwell
books
Book Nineteen Eighty-Four
Book Nineteen Eighty-Four

Fig. 40: The same example Library model instance as in Figure 39, with the customized mapping model. The pointer arrow is shown for the focused pointer block only.

B. Ecore metamodel evaluation

The Ecore metamodel is the model used to create Ecore models. The use of cross references and containment references is mixed, with a slight majority of containment. Figure 41 shows the prototype blocks which are generated into the palette when using the Ecore metamodel without any manual configuration.

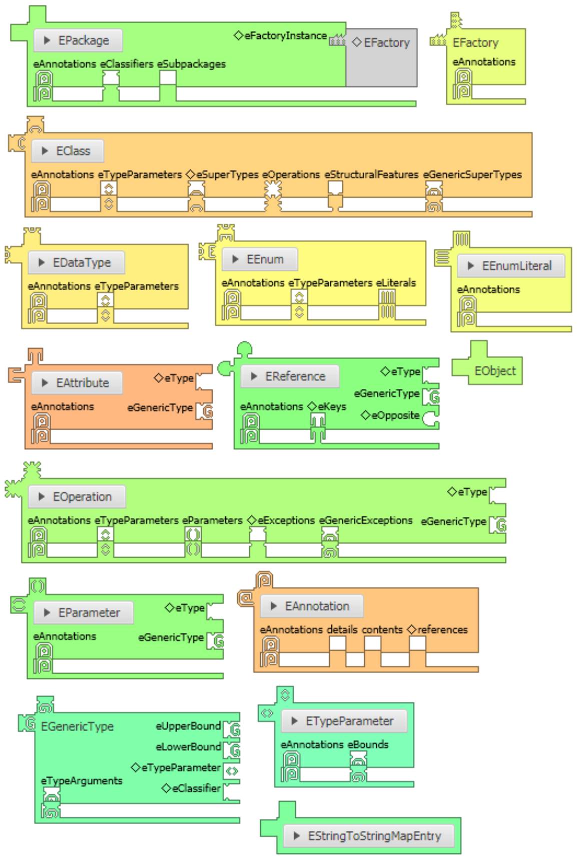


Fig. 41: The automatically generated Ecore blocks.

Modeling using Ecore depends on access to the data type literals from the Ecore package, since they are required to assign standard types to EAttributes. This showed that it is necessary to be able to create pointers to objects in other models, leading to the creation of the EProxyPointerBlock. Figure 42 shows the list of pointer blocks into the Ecore package. This list is automatically added into the palette, and includes all of the built-in EClasses and EDataTypes.

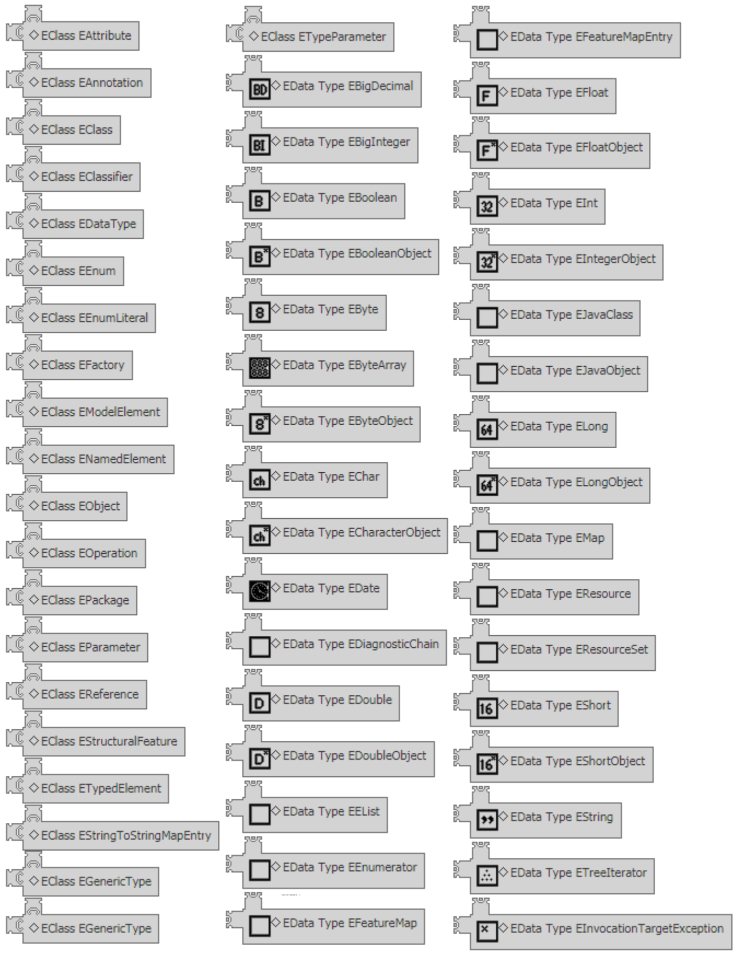
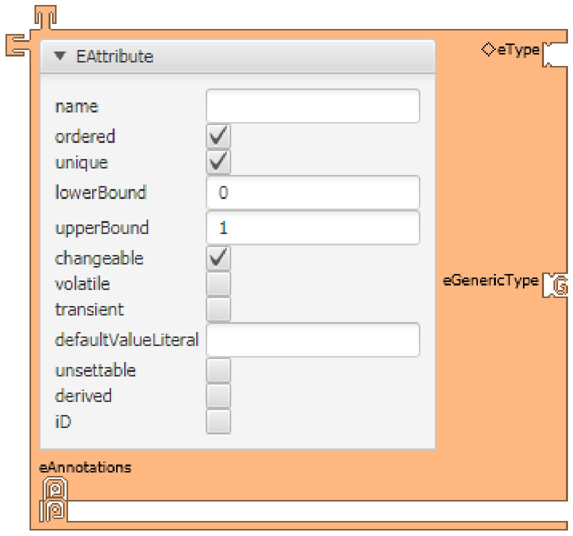


Fig. 42: All of the pointer blocks to the Ecore literals.

When first testing the editor with Ecore, we found that the generated slot and tab shapes were unsatisfactory, so we edited them to be more easily identifiable. Since these shapes would be routinely reused in other models, we decided to implement special support for importing them automatically. Thus, the editor uses these shapes by default, and there is no need to edit them for this domain.

Figure 43 shows the size of an expanded EAttribute block. In the early prototypes, it was not possible to collapse the property sheet. The Ecore metamodel showed that it is an essential feature due to the large size in expanded form.


EAttribute

name	
ordered	✓
unique	✓
lowerBound	0
upperBound	1
changeable	✓
volatile	
transient	
defaultValueLiteral	
unsettable	
derived	
iD	

Fig. 43: When expanded, many of the blocks become very tall due to the number of attributes.

Yet, using the editor with the generated mapping model for this domain is problematic for several reasons. Because Ecore models are rooted in a single EPackage, the model becomes a single block cluster, which is undesirable. Also, the default layout of the slots makes several of the blocks very wide. These symptoms can be seen in Figure 44.

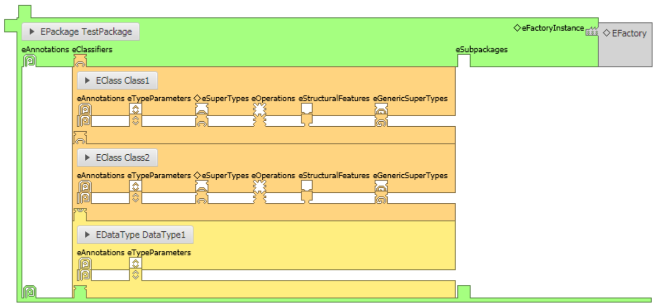

EPackage TestPackage
◇eFactoryInstance
◇EFactory
eAnnotations eClassifiers
eSubpackages
EClass Class1
eAnnotations eTypeParameters ◇eSuperTypes eOperations eStructuralFeatures eGenericSuperTypes
EClass Class2
eAnnotations eTypeParameters ◇eSuperTypes eOperations eStructuralFeatures eGenericSuperTypes
EDataType DataType1
eAnnotations eTypeParameters

Fig. 44: Using the default mapping model, the blocks become very wide and clumped up in a single EPackage block.

Because the default mapping model displays all information about the model, it appears very verbose. An example of this can be seen in Figure 45.

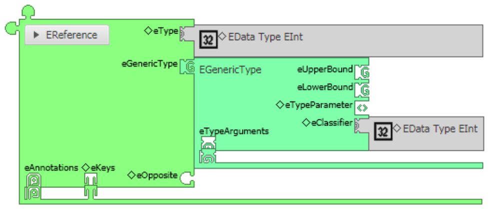


Fig. 45: When setting the eType of an EReference, the eGenericType gets automatically assigned, and vica versa. This is necessary to create generic models, but in most cases it is a waste of space: It adds to the perceived complexity without adding any value.

To make the editor more usable, some modifications are necessary. We change the EPackage representation to a work area, which is done by replacing its mapping model element with a work area mapping. Next, we make some simplifications, based on general assumptions about how often different functionality is used. We hide the blocks and slots related to generic types, factories and annotations. We also remove the EStringToStringMapEntry block, the EObject block and the eKeys slot.

In the initial prototypes, mapping model elements would reappear if we removed them, because the editor is designed to fill in missing elements in the mapping model. This proved that the ability to mark elements as hidden was necessary, as this allows the editor to identify elements that the modeler has explicitly hidden.

Finally, we change the orientations and locations of some of the slots in order to improve the block layouts. Figure 46 shows the Library model after the mapping model customizations.

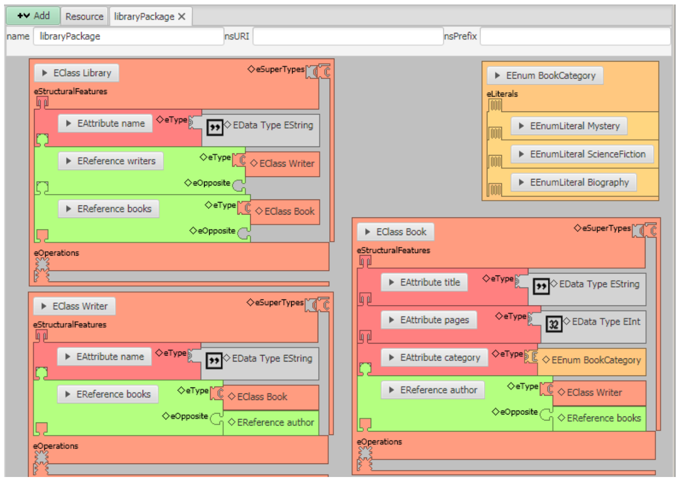


Fig. 46: The Library metamodel, represented using the customized mapping model for Ecore.

Although we have hidden some information, the customizations make the editor use the screen space more efficiently, as well as making the models easier to understand.

C. Family model evaluation

The Family model is a simplified model of a family tree. As in the Library model, only the root object uses containment, but this model has more cross references. A diagram of the model is shown in Figure 47.

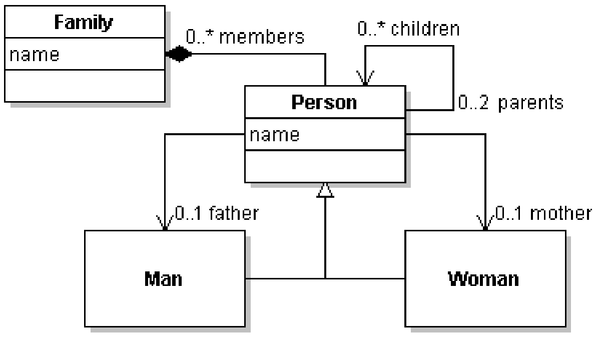


Fig. 47: An UML diagram of the Family model. Note that the *father* and *mother* relations can be derived from the *parents* relation, since they repeat the same information.

Figure 48) shows the prototype blocks which are automatically generated for the Family model.

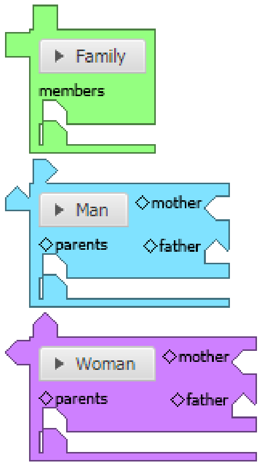


Fig. 48: The blocks from the generated Family palette.

The tab shapes are easily distinguishable. The Family object is a good candidate for being a work area. The bidirectional parents-children association has been collapsed to a single slot on the Man and Woman blocks. However, in this case, it seems more reasonable to flip the direction. We do this by setting the *children* reference to dominant in the mapping model. Figure 49 illustrates the result.

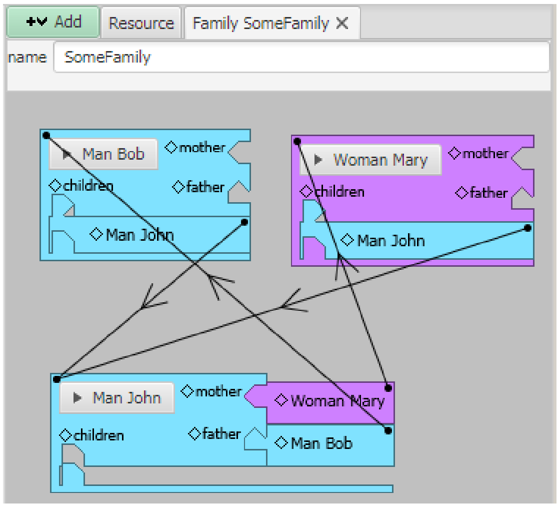


Fig. 49: An example Family model instance. Pointer arrows are enabled, showing the high amount of cross referencing.

D. Quiz model evaluation

The Quiz model defines different types of questions and answers. Most of the relations are containment references, which leads us to believe it is a good candidate. A simplified diagram of the model is shown in Figure 50.

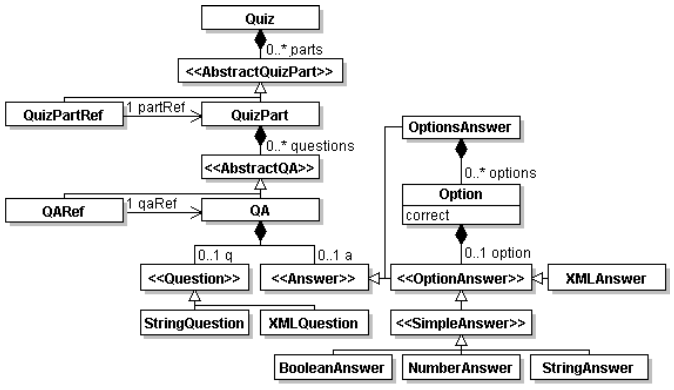


Fig. 50: A simplified UML diagram of the quiz model. The XML data structure is not shown.

Figure 51) shows the prototype blocks which are automatically generated for the Quiz model.

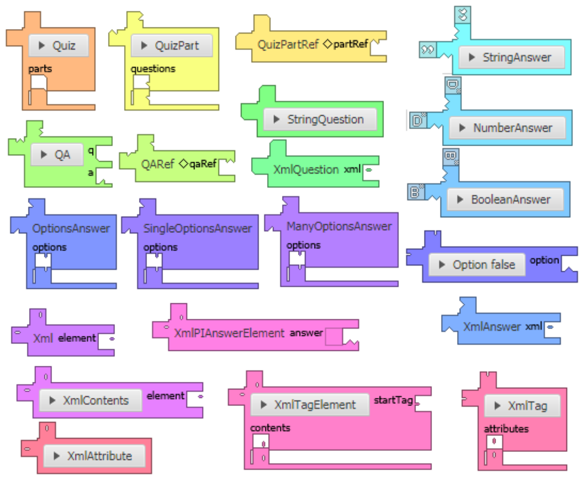


Fig. 51: The blocks from the generated Quiz palette.

This model was originally created for a textual DSL. There are a few things to explain about it.

- The Quiz object contains QuizParts, but sometimes it can be useful to cross reference QuizParts from other models. The QuizPartRef object facilitates this: It holds a cross reference to another QuizPart, and is able to serve as a placeholder for that object in a containment relationship. This is directly analogous to how pointer blocks are implemented in the Jigsaw Toolkit! The technique is useful in cases where the DSL provides incentives to favor containment over cross referencing. This is obviously the case in jigsaw-based DSLs, but also in textual DSLs: In textual DSLs, cross references must be created using a textual identifier, while the actual object is defined elsewhere. This kind of fragmentation constitutes a loss of *source code locality*, which can have detrimental effects on productivity. [23] The QARef object fulfills an identical role.
- The abstract SimpleAnswer class is parameterized, and its implementations StringAnswer, NumberAnswer and BooleanAnswer introduce the type arguments String, Double and Boolean, respectively. The editor renders type arguments where it finds them, but in reality, the type arguments for this domain are only an implementation detail. There are no references that expect a particular type argument, and nor is it possible to reassign any type arguments in the model instance.
- The XMLQuestion and XMLAnswer Blocks can be used to embed XHTML [14] fragments, for example to include images. XHTML is a separate, XML-based DSL. In a textual DSL, it makes sense to embed XML support directly in the metamodel instead of storing the XML text as a string. However, in a graphical editor such as ours, the XML objects are not useful. Attempting to construct an XHTML document using raw XML blocks would only be extremely cumbersome.

The Quiz object seems to be a good candidate for being a work area, so we change this in the mapping model. Next, we hide the XML-related objects from the palette. Finally, the Quiz domain has many unique tab shapes, which leads to the generated shapes being difficult to read. We improve them by altering them in the Cut Editor.

The customization of the shapes of slots and tabs is a time consuming part of the mapping model customization, but it is a very important step. For models with more than four unique shapes, the generated shapes begin to become somewhat difficult to read.

To design good shapes as a modeler, there are two needs to consider. Firstly, and most importantly, the shapes should be easily distinguishable when they need to be. If two types can never be substituted for one another, then they should appear completely different so that there is no doubt that they will never fit. Secondly, specialization should be easy to identify. This is more tricky, and is best illustrated with an example. Consider the QuizPart and QA classes, illustrated in Figure 52.

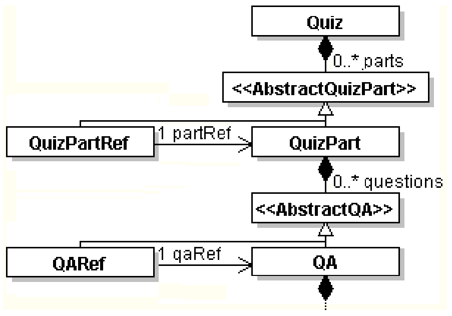


Fig. 52: The partition of the quiz domain that concerns QuizParts and QAs.

There is no reference that accepts both QuizPart and QA, so their shapes may be completely different. However, the sibling classes QuizPartRef and QARef also need to be considered.

The important thing to note is that QARef can only reference a QA, not an AbstractQA. This means that QA is a specialization that requires an unique shape. QARef, however, does not require an unique shape, because it can always be substituted for an AbstractQA. The same argument holds true for QuizPartRef.

Thus, the QA shape needs to be different from AbstractQA: It must be possible to tell that only a QA reference can be referenced by the QARef object. In the other hand, it must also be possible to tell that QA can still be contained in the QuizPart object.

We start by drawing a unique shape for AbstractQA in the Cut Editor. Since this type has a subtype which needs a unique shape, we take special care to reserve enough space to accommodate the cuts for the subtype. In general, it is important to consider how many times the shape needs to be modified further down the inheritance graph.

Then, we go to the QA class and add a couple of cuts that help distinguish it. These cuts should not distort the cuts inherited from AbstractQA: The goal is to help the user identify the subtype as a modified version of the parent class, not a completely distinct type (Figure 53).

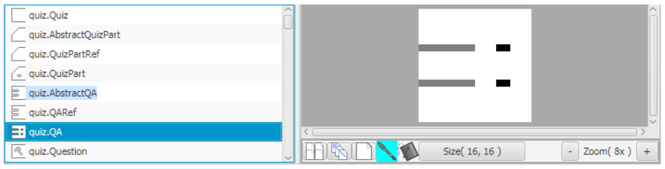


Fig. 53: The QA shape is differentiated from the AbstractQA shape by adding the black internal cuts. The gray cuts are automatically inherited from AbstractQA; this is handled automatically by the Jigsaw Toolkit.

The result is a set of shapes that hopefully allow the user to identify what connections can and cannot be made (Figure 54), even in a static image of a model. In practice, reading the subtype relationships likely requires some training. For this reason, the Jigsaw Toolkit also highlights valid drop targets when the user drags a block.

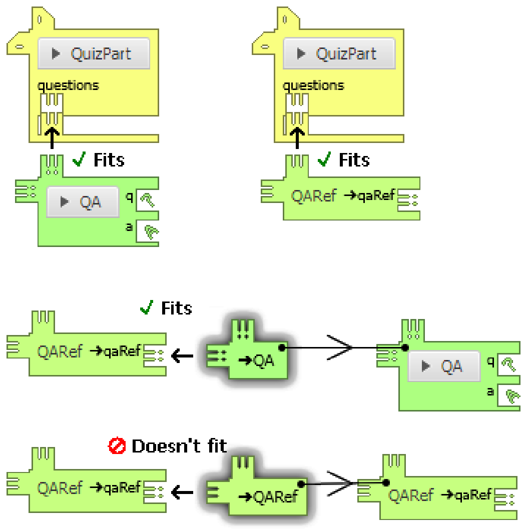


Fig. 54: The QA-related shapes are designed to communicate what connections can and cannot be made.

In some cases, we have opted to embed little symbols into the shapes. This is not necessary, but the use of familiar symbols may be helpful in producing shapes that are easy to distinguish.

Figure 55 illustrates the result.

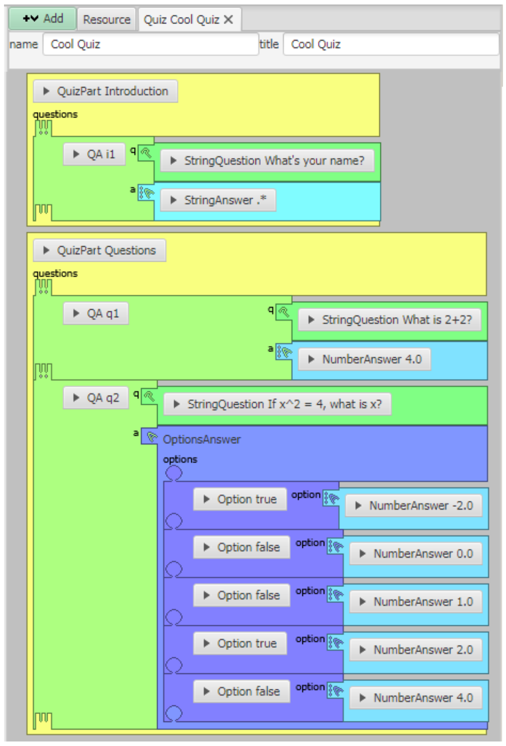


Fig. 55: An example Quiz model instance.

E. Survey results

The survey is intended to cover multiple aspects of the project.

Firstly, the survey acts as a usability test. The respondents are asked to run a prototype of the editor and complete a sequence of tasks. The prototype is configured to run as an Ecore editor. Since virtually all EMF users are familiar with Ecore, this allows the respondents to complete the tasks without the need for understanding an unfamiliar domain. After completing the tasks, they are asked to provide feedback on usability problems. The survey includes a download link for the prototype and a walkthrough video in case some users cannot figure out how to complete some of the tasks.

Secondly, the survey provides guidance on how to improve the editor. The respondents are asked for their opinion on various features that we are considering to implement, as well as their expectations for an editor such as ours.

Thirdly, the survey provides information on whether the editor's features are useful. In particular, we are interested in what users think of the features that are unique to the jigsaw-based concrete syntax. The users are asked for their opinion on these features, and whether they would be likely to use our editor in the future.

It would be possible to let the usability test be executed separately from the rest of the survey. However, we chose to combine them. The rationale for this was that by running a prototype of the editor first, the respondents would get a stronger impression of what the project is about, and thus provide more accurate responses.

The survey was created using Google Forms [9]. It was open to respondents during the last two months of the project. Before making the survey public, we had one person evaluate it and provide suggestions for improvements to the survey itself. It was then published on the Eclipse community forums, [8] where we hoped it would gain some attention. This was unsuccessful, so we started recruiting respondents more actively, by contacting colleagues and other acquaintances. A reward was also offered to students who had past experience with modeling.

A total of 5 people delivered a survey response. This is a low number, but it is enough to gain a general impression of where to focus development and what users think of the editor's features. In particular, it helps uncover many usability problems. The respondents had mixed levels of experience with EMF and other jigsaw-based languages.

The full survey is included in Appendix B, and the complete survey response data set is included in Appendix C.

1) Testing the editor: In this part of the survey, the respondents were asked to run a prototype of the editor and complete a set of tasks.

The majority of the respondents were able to complete most of the tasks without problems. The reported problems were concentrated around a few features.

The survey requires users to create an EPackage as a new work area. This is done using a dedicated *Add* button. Three respondents reported not discovering how to create the work area until they had already created the blocks. One respondent reported that they attempted to move the blocks into the new work area after creating it, but that their attempt was unsuccessful. It is possible to open the two work areas next to each other by double-clicking on one of the tabs (Figure 28), thus allowing the user to move blocks between them. The respondent did not discover this feature.

Several tasks require the use of Ecore data type literals. These blocks are in a palette tab that is separate from the ordinary blocks. Three respondents reported problems finding these blocks.

Multiple tasks require the use of pointer blocks. This can be done using either a keyboard shortcut or through a context menu which is accessed by right-clicking on a block. Four respondents reported problems with discovering how to create pointer blocks. One of them did not appear to discover how to use them at all.

One respondent reported initially having problems with understanding how to assemble blocks. They appear to have attempted to place a slot on top of a tab in order to connect them. This is currently not possible, and does nothing. The slot also required a pointer block, while the tab belonged to an ordinary block, so this would not work even when done the other way around. However, they did figure it out eventually.

One respondent reported that they did not like the colors of the blocks, in particular the red ones. They noted that red is often associated with errors or warnings. In response to this, we changed the editor to never generate red hues as the block color.

2) Current functionality of the editor: In this part of the survey, the respondents were asked to rank the usefulness of various features that are implemented in the editor.

Three of the respondents were positive on all points. One was neutral on several points, and another was negative to how the type system is visualized using tabs and slots.

3) Potential functionality of the editor: In this part of the survey, the respondents were asked to rank the importance of various features that we were considering to implement.

Support for model validation was ranked the highest by a wide margin. Response towards the other proposed features was more mixed.

Note that we opted to implemented support for generics shortly after publishing the survey, before we received more than one response.

4) Qualitative expectations for the editor: In this part of the survey, the respondents were asked to rank the importance of various qualitative propositions regarding the editor.

The respondents expressed that the stability and performance of the editor are very important.

When asked if it is important that the editor supports all EMF functionality, or that the mapping model is highly flexible, the response was mixed.

The amount of flexibility the editor allows for in terms of deployment was met with overall positive responses. Some users expressed more interest in the Eclipse integration, while others expressed more interest in use independently from the Eclipse platform.

When asked whether they would be likely to use our editor instead of the EMF tree editor, or as a replacement for an established DSL framework such as Sirius, the response was mixed.

VIII. Discussion

In the previous section, we evaluated the editor against a number of test domains, and presented the survey results. In this section, we discuss the results.

A. Usability of the editor

We discovered several usability problems through both the model evaluation and the survey. However, we believe that the vast majority of the usability problems can be eliminated by making the correct adjustments.

As we saw in Section VII-E, the majority of the respondents were able to understand and complete most of the tasks on their own. Most of the problems were centered around work areas, pointer blocks, and the use of multiple palette tabs.

Several respondents had a hard time discovering the Ecore data types (Figure 42), because they were in a separate palette tab. While we could fix this by simply moving all blocks into a single palette tab, this is not a good solution. It inevitably results in a highly cluttered palette which is inefficient for more experienced users. Also, the use of multiple palette tabs is not something that is new in our editor. Most or all existing jigsaw-based languages use multiple palette tabs. In Scratch, the palette tabs are color coded, and a significant amount of screen space is dedicated to them (Figure 14). By copying this design, users may be able to discover them more easily in our editor.

The Ecore data type blocks also share the same color, possibly making them difficult to distinguish. Other blocks are colored based the color attribute of the associated block mapping in the jigsaw mapping model. While we could certainly do something similar for EProxyPointerBlocks, this might be insufficient because many of them share the same class. An alternative would be to augment the palette and layout models with color information, effectively allowing different instances to have different colors. However, it is somewhat uncertain if this will improve the readability of models or simply make them more confusing.

The way work areas are created is inconsistent with how blocks are created, since work areas are not in the palette. As a result, the respondents did not immediately discover how to create them. One possible solution to this is to create a new type of block, the *work area block*, which serves as a secondary representation of a work area (Figure 56). This is an interesting idea for several reasons.

Fig. 56: A special block could be created which serves as a secondary representation of a work area. The user would be able to click on the block to open the work area.

Firstly, by including these blocks in the palette, we make them more easily discoverable, and creating them becomes consistent with how other blocks are created.

Secondly, this would allow any object to be opened as a work area, not just root objects. This effectively enables models to have multiple abstraction layers, similar to the concept of submodels in GEM or *subpatches* in the Pure Data visual programming language. [21] This would be useful in, for example, the Quiz model:

The containment structure in the QuizPart objects can be quite deep. As a result, QuizPart exhibits some of the same problems as a model with a single block cluster. Some of the blocks become unduly stretched (Figure 55), and it might be better to use the editor if it was possible to position the QA blocks freely. Thus, it is possible that it would be better if the QuizPart objects were represented as work areas instead of the Quiz object. This is currently not possible because QuizPart is not a root object. However, the use of work area blocks would make this possible.

The survey respondents did not discover how to open two work areas next to each other. By extension, they were unable to move blocks between the two work areas. One user reported trying to drag the blocks onto the work area tab near the top of the editor. This is not implemented, but it is a good idea. This would solve many of the same problems as opening work areas next to each other. In addition, we would like to make it easier to discover how to use the existing functionality. Adding context menus on the work area tabs would be a step in the right direction. Support for dragging and dropping the work area tabs in order to put them next to each other would also be an improvement.

Most of the respondents reported issues with discovering how to create pointer blocks. One respondent indicated that better error messages would help alleviate this. When the user attempts to use an ordinary block as a pointer block, the editor currently displays an error message. Adding instructions for how to create a pointer block to this message might help. In addition, we propose to add an alternative method of creating pointer blocks, wherein a pointer block can be created by dragging a distinctively marked, dedicated area of an ordinary block (Figure 57).

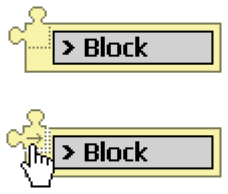


Fig. 57: A pointer block could be created by dragging the corner of a block. The corner would be marked and provide feedback when the user hovers the cursor over it.

The explicit management of work areas and the use of pointer blocks are novel techniques that do not exist in established jigsaw-based languages. As such, it is not surprising that they are somewhat problematic. The respondents appear to have had less problems with the other novel features, such as the use of collapsible property sheets and the more flexible use of tabs and slots compared to established jigsaw-based languages.

B. Suitability of the jigsaw-based syntax

We do not expect the jigsaw-based concrete syntax to be suitable for all models. In particular, established jigsaw-based languages only support containment references. Our editor uses pointer blocks to get around this limitation, but we do expect that overuse of pointer blocks may be problematic.

Both the library model and the family model are dominated by cross references; all the containment references are on the root objects.

For the library model, the jigsaw-based concrete syntax performs no worse than a graph-based syntax. Indeed, it is possible to merely treat the editor as a somewhat strange graph-based editor, where the first anchor-point of an edge happens to be a pointer block. One interesting thing to note is that the use of pointer blocks allows users to impose an explicit order in reference lists, something that is normally not possible in graph-based editors. In graph-based editors, defining a clear order in a many-element relation is a more difficult problem.

Since each book object can only have one author, an obvious way to reorganize this model would be to let the writer object contain the books. This would allow the user to drop the book objects directly on the writer objects, with no need for pointer blocks at all. With Sirius, this type of structural modification could probably be implemented in the mapping model, but it is not possible with our editor.

For the family model, the number of cross references grows much faster. Furthermore, the children-parents association is many-to-two, meaning there would be no way to model it using a containment tree.

The large number of pointer arrows quickly make the family model instances unreadable. This could be alleviated to some extent by improving the edge routing. Currently, only straight lines are supported. Allowing the user to add bend points to the arrows would allow for improvements. However, unlike in a graph-based editor, the pointer arrows cannot emerge from any edge of the block; they must be attached to a pointer block, which must be attached to the relevant slot. Thus, it cannot perform as well as a graph-based editor, because the pointer arrows are inevitably more difficult to trace.

The pointer arrows can be turned off, but this is not a very good idea either. A family model where the familial relations have to be read through textual labels is just inconvenient.

The Ecore metamodel uses an even combination of cross references and containment references. Like in the family model, the pointer arrows can lead to a significant amount of clutter. However, hiding them is a much better idea for this model. Most of the pointer blocks reference data types or classes, and we expect that users should be able to memorize their names and significance without much trouble.

In the other hand, the Quiz model uses containment references almost exclusively. As such, we would expect it to be more suitable for our editor. Indeed, it seems very easy to understand how the blocks fit together, and reading a model is very straightforward.

As expected, the editor performs well for models that predominantly use containment references, and less well for models where cross references are more important. Although the editor can be treated like a graph-based editor when dealing with cross references, the way the pointer arrows are organized is a problem. Although better edge routing would be an improvement, it is not a definitive solution.

The editor allows the user to toggle the pointer arrows off to decrease clutter. This is helpful if the number of pointer blocks is relatively low, and it is easy to identify what they are referencing based on the label. However, in cases where it is very important to be able to trace the links visually, such as in the Family model, turning them off is not very helpful.

Thus, the editor is less suited to models that are inherently graph-like, with heavy cross referencing.

Our editor provides incentives to use containment references rather than cross references. An interesting thing to note is that this is also true for textual editors, as we explained in Section VII-D. Thus, the editor might serve as a more user friendly alternative to textual editors, and not so much as an alternative to graph-based editors. To help back up this claim, we might add that the jigsaw-based visual syntax is traditionally used as a user friendly alternative to textual programming languages.

Several classes in the Ecore metamodel, including EReference and EAttribute, inherit from the abstract ETypedElement class. This class defines a changeable reference named *type*, which has the type EClassifier. This suggests to the user that they can use any EClassifier with any typed element. However, this is wrong! The *type* slot on the EReference block should not accept an EDataType block, and similarly the *type* slot on the EAttribute block should not accept an EClass block. This is a problem because it suggests to the user that it is valid to connect the blocks in a way that is, in reality, never valid.

In general, the editor works best with metamodels where the types of references are precisely defined. If the types are imprecise, as in Ecore, then this communicates the wrong ideas to the user. We can obviously not patch this in the Ecore metamodel itself, but we could augment the jigsaw mapping model to allow the type of a slot to be different from the actual type defined in the abstract model.

Models displayed in our editor become rather huge. In its current state, our editor is not well suited to handle models with much more than 100 objects in a work area. This is a key problem which limits the applicability of our editor.

We do our best to manage space efficiently, for example by allowing the user to collapse property sheets and adjust the zoom in a work area, but there does not appear to be any definitive solution. While some space is certainly wasted due to the way blocks are laid out, much of the space is simply required to visualize all the information that our editor provides.

Indeed, one of the biggest strengths of jigsaw-based languages is that they show a lot of information to the user, in a way that is simple to understand. For example, the shapes of the tabs and slots allow the user to see the types of both objects and references at all times, as well as the relationships between them. This is an unique feature when compared to other EMF editors. Our editor also offers better locality than many other graphical editors, by placing the property sheets inside the blocks rather than in a separate panel. Furthermore, the available blocks are very easy to discover and use through the palette.

However, this necessarily means that the jigsaw-based concrete syntax is generally less compact than alternative syntaxes. The Google Blockly framework attempts to get around this by allowing blocks to be completely collapsed, in a way that also hides the slots. This hides a lot of information, but it might be a good idea. For our editor, we could even imagine that a block could be collapsed into a work area block, which we described in Section VIII-A. This would allow the user to manage how to divide the model into work areas in order to better manage space.

In the Ecore metamodel, we chose to hide some functionality in order to make the editor more convenient to use. In particular, everything related to generic types and annotations was completely hidden. If it becomes necessary to show this functionality on the jigsaw objects, then the mapping model needs to be edited. This is a poor solution. An alternative solution is to provide users with a more convenient way to show or hide slots, perhaps similar to how property sheets can be collapsed.

However, another way to handle this is to use a secondary view of the model, like a tree editor, to manage rarely-used functionality. Thus, the associated model elements may remain hidden in the jigsaw editor. We believe that in most cases, this is an acceptable solution.

In the Ecore model, the EPackage object is represented as a work area. A limitation of this is that it is impossible to create nested EPackages, because a work area cannot be contained in a work area. Although nested packages are rarely used in EMF, this limitation matters on a general basis.

The use of work area blocks, as described in Section VIII-A, would solve this problem, since the work area block could be contained in a work area with no problems. Furthermore, it would allow users to create pointer blocks to work areas. Thus, it would be an important feature in that it would add to the the expressiveness of the syntax.

In the Quiz model, we configured the editor to represent the Quiz object as a work area. There is a problem with this; the work area has no sense of order, so the order of the QuizParts in the abstract model is rather arbitrary. Order matters in many models, so this is a noteworthy problem. It would be vastly preferable if it could be fixed without altering the abstract model. As a solution, we propose that it should be possible to allow a work area to be sorted, for example top-to-bottom. A general sorting mechanism could be implemented by allowing the modeler to specify a comparator function in the work area mapping.

The Quiz model also exposes a set of XML-related objects. For the reasons we described in Section VII-D, these make sense when the model is used with a textual syntax, but they would be very inconvenient to use and difficult to understand with our syntax. In order to make these objects useful in our editor, the best solution would be to embed a textual editor such as XText into a block. This would be a somewhat ambitious project, however. Changing the Quiz model to hold the XML as a string would obviously be much easier in the short term.

C. Limitations of our evaluation

We have done our best to select evaluation models according to reasonable criteria, as explained in Section VII. However, we are far from a complete understanding of when our syntax is applicable. In the future, we would like to use the editor with more models in order to gain a better understanding.

The number of respondents was lower than we had hoped to achieve. In hindsight, it would have been wiser to split the survey into a usability test and a conceptual survey. The conceptual survey could have been published earlier in the project, and respondents might have been less put off by its complexity.

The survey does function as a usability test, and it does provide pointers to how to improve the editor. However, we cannot form a clear conclusion on whether a jigsaw-based modeling editor will be likely to attract users.

It is certain that our editor does require improvements before it can be put to serious use, but we hope that it will be seen as a serious alternative in the future.

The use of Ecore as the domain model in the prototype allowed the respondents to complete the tasks without the need for understanding an unfamiliar domain. The idea behind this was to let the usability test be more about understanding the editor and less about understanding the domain. However, some of the respondents appear to have carried over incompatible expectations from other Ecore editors. It would also be interesting to see if the editor aids users in understanding unfamiliar domains. Indeed, it is possible that this would be one of its greatest strengths. In the future, we hope to learn more about this.

In addition, we have not had anyone test the editor from the modeler's perspective. We can configure the editor for a new domain in minutes, but we do not yet know if learning to work with the mapping model is an easy task for other modelers.

IX. Conclusion

We have developed a new framework for creating jigsaw-based DSLs, and provided some initial research into when the jigsaw-based concrete syntax is appropriate.

Several Jigsaw-based DSLs already exist, most of which are used to create computer programs. However, because our editor uses the Jigsaw Toolkit, it offers more expressiveness than existing jigsaw-based DSLs. In particular it is able to visualize cross references between blocks and much more complex type systems.

Furthermore, unlike the existing frameworks for creating jigsaw-based GUIs, our editor can be used to create DSLs using structured modeling techniques, without the need to write any code.

We evaluated a number of existing metamodels against our editor, and found several interesting strengths and weaknesses. Most importantly, our editor is not a drop-in replacement for graph-based editors. It is more likely to perform well as a user-friendly editor in domains where graph-based editors do not perform so well; in particular where textual editors are traditionally preferred.

We also collected data through a survey. This revealed several usability problems, but also suggested that the editor has the potential to be useful. Although the respondents were not assured that our editor is a good replacement for established editors, they were generally positive to its concept and unique features.

There is still much room for improvement in our editor. At least some further work is necessary before the editor can be considered a finished product. In Section VIII, we laid out a number of suggestions for possible future improvements based on the results from the model evaluations and the feedback from the survey. In particular, we need to work on resolving the usability problems.

The survey respondents expressed interest in both Eclipse integration and use separately from Eclipse. As such, Eclipse integration should be a priority, but we should avoid making changes that irrevocably bind the editor to the Eclipse platform.

There was also a great deal of interest in model validation, which we have not implemented. EMF includes support for model validation, and it generates a set of *markers*. These markers can be displayed on the blocks as error or warning symbols, which provide additional information when the user hovers over them with the cursor.

Stability and performance were also ranked with overall high importance. This is not surprising, but it is worth taking into account. Currently, the performance is limited. Filling the screen with blocks can lead to the editor lagging. In the future, we should look into how to improve the rendering performance of the editor.

The latest version of the Jigsaw EMF Editor can be found at the Bitbucket repository. [2] At least Java version 8 update 40 is required to run the editor.

References

[1] Hungnes, Oddvar, and Hallvard Trætteberg. ”Jigsaw Language Toolkit.” Norsk Informatikkonferanse (NIK) (2015).

[2] Hungnes, Oddvar. Retrieved 04. February 2016. ”Jigsaw EMF Editor - Overview.” Available: https://bitbucket.org/Oddwarg/jigsaw-emf-editor/overview

[3] Preece, J., Rogers, Y., Sharp, H. ”Interaction Design: Beyond Human-Computer Interaction”, New York: Wiley, (2002) p.21

[4] Steinberg, Dave, et al. ”EMF: eclipse modeling framework.” Pearson Education, 2008.

[5] The Eclipse Foundation. Retrieved 05. January 2016. ”Ecore-Tools - Graphical Modeling for Ecore.” Available: www.eclipse.org/ecoretools/

[6] Catherine Griffin, IBM. Retrieved 04. December 2016. ”Using EMF.” Available: https://eclipse.org/articles/Article-UsingEMF/using-emf.html

[7] The Eclipse Foundation. Retrieved 07. January 2016. ”Generating an EMF Model” Available: https://www.eclipse.org/modeling/emf/docs/2.x/tutorials/clibmod/clibmod_emf2.0.html

[8] The Eclipse Foundation. Retrieved 01. February 2016. ”Eclipse Community Forums.” Available: https://www.eclipse.org/forums/index.php?t=thread&frm_id=108

[9] Google. Retrieved 01. February 2016. ”Google Forms.” Available: https://www.google.com/forms/about/

[10] The Eclipse Foundation. Retrieved 15. December 2015. ”Eclipse Modeling Project.” Available: https://eclipse.org/modeling/

[11] IBM. Retrieved 15. December 2015. ”Build metamodels with dynamic EMF” Available: http://www.ibm.com/developerworks/library/os-eclipse-dynamicemf/

[12] Object Management Group. Retrieved 15. December 2015. ”OMG Meta-Object Facility (MOF) Core Specification, Version 2.4.1.” Available: http://www.omg.org/spec/MOF/2.4.1/PDF

[13] Object Management Group. Retrieved 15. December 2015. ”XML Metadata Interchange (XMI)” Available: http://www.omg.org/spec/XMI/

[14] W3C. Retrieved 19. January 2016. ”XHTML^{TM} 1.0 The Extensible HyperText Markup Language (Second Edition)” Available: https://www.w3.org/TR/xhtml1/

[15] Object Management Group. Retrieved 09. January 2016. ”Object Control Language (OCL)” Available: http://www.omg.org/spec/OCL/

[16] Microsoft. Retrieved 09. January 2016. ”SQL Server 2014.” Available: https://www.microsoft.com/en-us/server-cloud/products/sql-server/

[17] Boutell, Thomas. ”PNG (Portable Network Graphics) Specification Version 1.0.” (1997).

[18] Wikipedia. Retrieved 15. January 2016. ”printf format string.” Available: https://en.wikipedia.org/wiki/Printf_format_string

[19] Wikipedia. Retrieved 25. January 2016. ”Inheritance (object-oriented programming).” Available: https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)

[20] JetBrains s.r.o. Retrieved 25. January 2016. ”IntelliJ IDEA the Java IDE.” Available: https://www.jetbrains.com/idea/

[21] Pd-community. ”Pure Data Documentation chapter 2: theory of operation” Available: https://puredata.info/docs/manuals/pd/x2.htm

[22] Wikipedia. Retrieved 18. January 2016. ”Switched-mode power supply.” Available: https://en.wikipedia.org/wiki/Switched-mode_power_supply

[23] Sillito, Jonathan. ”Improving Source Code Locality.” (2004).

[24] The Eclipse Foundation. Retrieved 22. Dec 2015. ”Dynamic Browsing and Instantiation Capabilities in EMF.” Available: http://wiki.eclipse.org/Dynamic_Browsing_and_Instantiation_Capabilites_in_EMF

[25] Weilkiens, Time et al. Model-Based System Architecture. John Wiley & Sons, 2015.

[26] Heidenreich, Florian, et al. ”Model-Based Language Engineering with EMFText.” Generative and Transformational Techniques in Software Engineering IV. Springer Berlin Heidelberg, 2013. 322-345.

[27] EMFText Core Development Team. Retrieved 17. January 2016. ”EMFText Concrete Syntax Zoo.” Available: http://www.emftext.org/index.php/EMFText_Concrete_Syntax_Zoo

[28] EMFText Core Development Team. Retrieved 04. January 2016. ”EMFText Getting Started Screencast.” Available: http://www.emftext.org/index.php/EMFText_Getting_Started_Screencast

[29] Scowen, Roger S. Extended BNF-a generic base standard. Technical report, ISO/IEC 14977. http://www. cl. cam. ac. uk/mgk25/iso-14977. pdf, 1998.

[30] Efftinge, Sven, and Markus Völter. ”oAW xText: A framework for textual DSLs.” Workshop on Modeling Symposium at Eclipse Summit. Vol. 32. 2006.

[31] The Eclipse Foundation. Retrieved 03. January 2016. ”Graphical Modeling Project(GMP).” Available: http://www.eclipse.org/modeling/gmp/

[32] The Eclipse Foundation. Retrieved 03. January 2016. ”GEF (Graphical Editing Framework).” Available: https://eclipse.org/gef/

[33] Van Wyk, Eric, and Mats Per Erik Heimdahl. ”Flexibility in modeling languages and tools: A call to arms.” International journal on software tools for technology transfer 11, no. 3 (2009): 203-215.

[34] Vujović, Vladimir, Mirjana Maksimović, and Branko Perišić. ”Comparative analysis of DSM Graphical Editor frameworks: Graphiti vs. Sirius.” 23nd International Electrotechnical and Computer Science Conference. 2014.

[35] The Eclipse Foundation. Retrieved 03. January 2016. ”Sirius/-Tutorials/StarterTutorial).” Available: https://wiki.eclipse.org/Sirius/Tutorials/StarterTutorial

[36] Musset, Jonathan, et al. ”Acceleo user guide.” (2006).

[37] Oracle America, Inc. Retrieved 12. Dec 2015, ”The Java Language Specification, Java SE 8 Edition.” Available: https://docs.oracle.com/javase/specs/jls/se8/html/

[38] Oracle America, Inc. Retrieved 14. January 2016, ”Non-Reifiable Types.” Available: https://docs.oracle.com/javase/tutorial/java/generics/nonReifiableVarargsType.html

[39] Oracle America, Inc. Retrieved 13. January 2016, ”JavaFX Interoperability with SWT.” Available: https://docs.oracle.com/javafx/2/swt_interoperability/jfxpub-swt_interoperability.htm

[40] BestSolution.at. Retrieved 13. January 2016, ”e(fx)clipse - JavaFX Tooling and Runtime for Eclipse and OSGi.” Available: https://www.eclipse.org/efxclipse/index.html

[41] Tom Schindl. ”EMF-Edit-Support is coming to JavaFX via e(fx)clipse.” Available: http://tomsondev.bestsolution.at/2012/12/13/emf-edit-support-is-coming-to-javafx-via-efxclipse/

[42] Bank, Joseph A., Andrew C. Myers, and Barbara Liskov. ”Parameterized types for Java.” Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1997.

[43] Lifelong Kindergarten Group, MIT Media Lab. Retrieved 02. December 2015. ”Scratch.” Available https://scratch.mit.edu/

[44] Malan, David J., and Henry H. Leitner. ”Scratch for budding computer scientists.” ACM SIGCSE Bulletin. Vol. 39. No. 1. ACM, 2007.

[45] MIT STEP. Retrieved 04. Jun 2015. ”StarLogo TNG.” Available: http://education.mit.edu/projects/starlogo-tng

[46] Lifelong Kindergarten Group, MIT Media Lab. Retrieved 08. Dec 2015. ”List of Scratch Modifications - Scratch Wiki.” Available: http://wiki.scratch.mit.edu/wiki/List_of_Scratch_Modifications

[47] Lifelong Kindergarten Group, MIT Media Lab. Retrieved 08. Dec 2015. ”Blocks - Scratch Wiki.” Available: http://wiki.scratch.mit.edu/wiki/Blocks#Block_Shapes

[48] Lifelong Kindergarten Group, MIT Media Lab. Retrieved 08. Dec 2015. ”Web Blox (Scratch Modification) - Scratch Wiki.” Available: http://wiki.scratch.mit.edu/wiki/Web_Blox_(Scratch_Modification)

[49] Massachusetts Institute of Technology. Retrieved 07. January 2016. ”MIT App Inventor.” Available: http://appinventor.mit.edu/

[50] Einstein’s Workshop. Retrieved 10. May 2015. ”BlocksCAD.” Available: http://www.einsteinsworkshop.com/blockscad

[51] Booch, Grady. ”The unified modeling language.” Unix Review 14.13 (1996): 5.

[52] Bray, Tim, et al. ”Extensible markup language (XML).” World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210 16 (1998).

[53] Fowler, Martin. ”Domain-specific languages.” Pearson Education, 2010.

[54] Fondement, Frédéric. ”Concrete syntax definition for modeling languages.” Diss. ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2007.

[55] Ranta, Aarne. ”Implementing Programming Languages.” College Publications, London, 2012.

[56] Ehrig, Karsten, et al. ”Generation of visual editors as eclipse plug-ins.” Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. ACM, 2005.

[57] Guzak, Christopher J., et al. ”Tree view control.” U.S. Patent No. 5,977,971. 2 Nov. 1999.

[58] Rothermel, Gregg, et al. ”What you see is what you test: A methodology for testing form-based visual programs.” Proceedings of the 20th international conference on Software engineering. IEEE Computer Society, 1998.

[59] Ráth, István, András Ökrös, and Dániel Varró. ”Synchronization of abstract and concrete syntax in domain-specific modeling languages.” Software & Systems Modeling 9.4 (2010): 453-471.

[60] Aho, Alfred V. Compilers: Principles, Techniques and Tools (for Anna University), 2/e. Pearson Education India, 2003.

[61] Smolander, Kari, et al. ”MetaEdit—a flexible graphical environment for methodology modelling.” Advanced Information Systems Engineering. Springer Berlin Heidelberg, 1991.

[62] Ledeczi, Akos, et al. ”The generic modeling environment.” Workshop on Intelligent Signal Processing, Budapest, Hungary. Vol. 17. 2001.

[63] Tolvanen, Juha-Pekka, and Steven Kelly. ”MetaEdit+: defining and using integrated domain-specific modeling languages.” Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications. ACM, 2009.

[64] Wachsmuth, Guido. ”Metamodel adaptation and model co-adaptation.” ECOOP 2007–Object-Oriented Programming. Springer Berlin Heidelberg, 2007. 600-624.

[65] Heiko Kern, Service and Integration Technology. Retrieved 15. December 2015. ”Processing of MetaEdit+ Models with oAW” Available: http://www.integration-engineering.de/2009/08/processing-of-metaedit-models-with-oaw

[66] De Smedt, Philip. ”Comparing three graphical DSL editors: AToM3, MetaEdit+ and Poseidon for DSLs.” University of Antwerp (2011).

[67] Box, Don, et al. ”Simple object access protocol (SOAP) 1.1.” (2000).

Appendix A
EMF models used by the editor

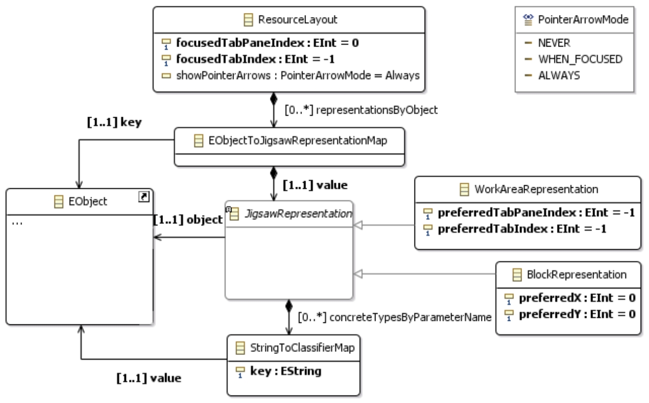

ResourceLayout
focusedTabPaneIndex : EInt = 0
focusedTabIndex : EInt = -1
showPointerArrows : PointerArrowMode = Always
PointerArrowMode
NEVER
WHEN_FOCUSED
ALWAYS
[0..*] representationsByObject
[1..1] key
EObjectToJigsawRepresentationMap
[1..1] value
WorkAreaRepresentation
preferredTabPaneIndex : EInt = -1
preferredTabIndex : EInt = -1
EObject
[1..1] object
JigsawRepresentation
BlockRepresentation
preferredX : EInt = 0
preferredY : EInt = 0
[0..*] concreteTypesByParameterName
StringToClassifierMap
key : EString
[1..1] value

Fig. 58: The EMF model that the editor uses to store layout information.

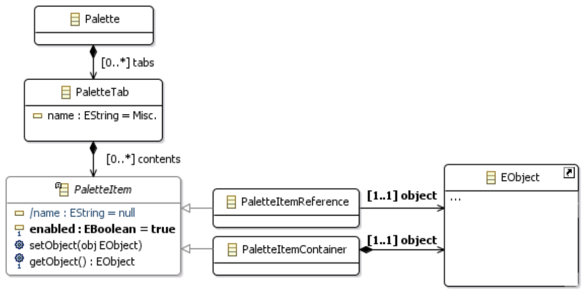


Fig. 60: The EMF model which is used to represent palettes in the editor.

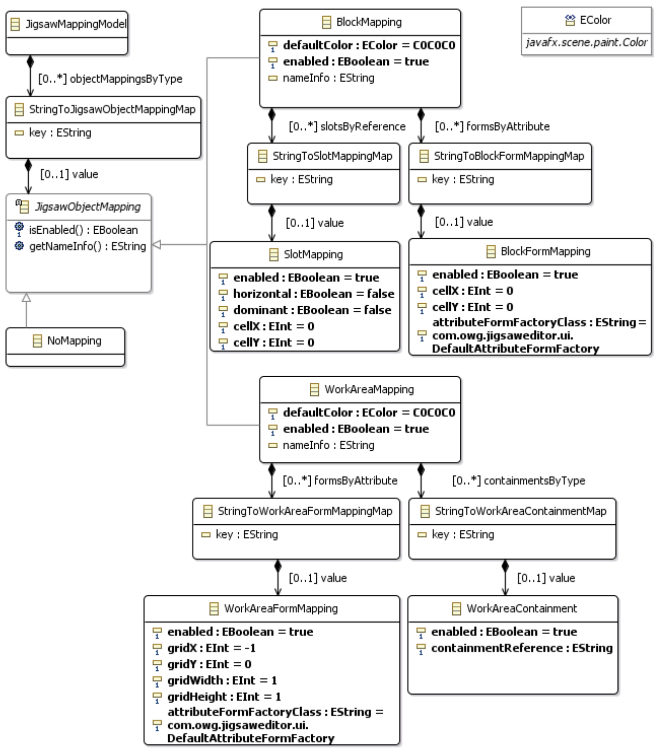

JigsawMappingModel
[0..*] objectMappingsByType
StringToJigsawObjectMappingMap
key : EString
[0..1] value
JigsawObjectMapping
isEnabled() : EBoolean
getNameInfo() : EString
NoMapping
BlockMapping
defaultColor : EColor = C0C0C0
enabled : EBoolean = true
nameInfo : EString
EColor
javafx.scene.paint.Color
[0..*] slotsByReference
[0..*] formsByAttribute
StringToSlotMappingMap
key : EString
StringToBlockFormMappingMap
key : EString
[0..1] value
[0..1] value
SlotMapping
enabled : EBoolean = true
horizontal : EBoolean = false
dominant : EBoolean = false
cellX : EInt = 0
cellY : EInt = 0
BlockFormMapping
enabled : EBoolean = true
cellX : EInt = 0
cellY : EInt = 0
attributeFormFactoryClass : EString = com.owg.jigsaweditor.ui.DefaultAttributeFormFactory
WorkAreaMapping
defaultColor : EColor = C0C0C0
enabled : EBoolean = true
nameInfo : EString
[0..*] formsByAttribute
[0..*] containmentsByType
StringToWorkAreaFormMappingMap
key : EString
StringToWorkAreaContainmentMap
key : EString
[0..1] value
[0..1] value
WorkAreaFormMapping
enabled : EBoolean = true
gridX : EInt = -1
gridY : EInt = 0
gridWidth : EInt = 1
gridHeight : EInt = 1
attributeFormFactoryClass : EString = com.owg.jigsaweditor.ui.DefaultAttributeFormFactory
WorkAreaContainment
enabled : EBoolean = true
containmentReference : EString

Fig. 59: The EMF model which is used as the editor's mapping model

Appendix B
Jigsaw EMF Editor Survey

,

Jigsaw EMF Editor

We are designing an editor for the Eclipse Modeling Framework (EMF) with a visual syntax based on the jigsaw puzzle metaphor, similar to Scratch (https://scratch.mit.edu/). To aid in our research, we would like you to test the editor and provide feedback.

The survey consists of 8 pages and is estimated to take about 30 minutes, maybe a little more depending on your speed.

A test build can be downloaded here:
http://folk.ntnu.no/oddvahu/jigsawemfeditor.zip
The test build requires at least Java 8 update 40! If it does not work, please update your Java version.

First, we would like to know a little bit about your previous experience.

* Required

1. **How familiar are you with the Eclipse Modeling Framework (EMF)?** *
Mark only one oval.
   - Not at all familiar
   - Slightly familiar
   - Somewhat familiar
   - Familiar
   - Very familiar

2. **How familiar are you with Ecore?** *
Ecore is the metamodel employed by EMF for describing domains.
Mark only one oval.
   - Not at all familiar
   - Slightly familiar
   - Somewhat familiar
   - Familiar
   - Very familiar

3. **How often do you use text-based model editors?** *
These editors represent the model using a textual language. Example: The Xcore notation for Ecore (https://wiki.eclipse.org/Xcore)
Mark only one oval.
   - Never
   - Almost never
   - Sometimes
   - Fairly often
   - Very often

4. **How often do you use graph-based model editors?** *
These editors represent the model using nodes and edges. Example: UML (http://tinyurl.com/o2l5v9t)
Mark only one oval.
   - Never
   - Almost never
   - Sometimes
   - Fairly often
   - Very often

5. **How often do you use form-based model editors?** *
These editors represent the model using trees, lists and/or tables. Example: Ecore tree editor (http://tinyurl.com/pndnkpo)
Mark only one oval.
   - Never
   - Almost never
   - Sometimes
   - Fairly often
   - Very often

6. **Have you ever used a visual language based on the jigsaw puzzle metaphor?** *
These languages have blocks that snap together. Example: Scratch (https://scratch.mit.edu/)
Mark only one oval.
   - No
   - Yes

Testing the editor (Part 1)

On this page, we would like to investigate how easy it is to learn to use the editor. If you haven't already, please download the test build from http://folk.ntnu.no/oddvahu/jigsawemfeditor.zip. The test build is preconfigured as an Ecore editor.

The test build requires at least Java 8 update 40! If it does not work, please update your Java version.

The following UML diagram describes the library model: http://help.eclipse.org/mars/topic/org.eclipse.emf.doc/tutorials/clibmod/images/model.gif

The library model is a simplified model of a library. We would like you to reproduce this model using our editor.

The classes that you define should be in a package called "library".

The library model comprises the following classes:
 - Library:
A library has a name (a string), and it contains writers and books.
 - Writer:
A writer has a name (a string), and a list of the books they have written.
 - Book:
A book has a name (a string), a page count (an integer), a category, and one author(a writer).

The category of a book should use an enum type with Mystery, Science Fiction and Biography as possible values.

The books of a writer and the author of a book should be "opposites", constituting a bidirectional association: If a book has a given author, then this is the same as stating that that writer has written the book.

Please try to complete these modelling tasks, before proceeding to the next page to answer questions about the tasks.

Testing the editor (part 2)

For reference, the completed model should look something like this:
https://drive.google.com/file/d/0B1WFkELkcR6XOThHcTh6QW1HcW8/view?usp=sharing

A walkthrough video is posted here: https://www.youtube.com/watch?v=-n9bKVM7BSo
If you needed to look at the walkthrough to complete a task, please write down why you couldn't do it on your own.

7. **I was able to create a package for my classes.**
Your model should contain a tab for the library package.
Mark only one oval.
   - Yes
   - Partially
   - No
   - I don't know

8. **If you had any problems with the above step, please describe them.**

9. **I was able to create classes.**
Your model should contain EClass blocks.
Mark only one oval.
   - Yes
   - Partially
   - No
   - I don't know

10. **If you had any problems with the above step, please describe them.**

11. **I was able to assign names to my classes.**
Your EClass blocks should be named Library, Book, and Writer.
Mark only one oval.
   - Yes
   - Partially
   - No
   - I don't know

12. **If you had any problems with the above step, please describe them.**

13. **I was able to add attributes to my classes.**
Your EClass blocks should contain EAttribute blocks.
Mark only one oval.
   - Yes
   - Partially
   - No
   - I don't know

14. **If you had any problems with the above step, please describe them.**

15. **I was able to assign standard types to my attributes.**
Your 'name' attribute blocks should have an EString block attached. The 'pages' attribute block should have an EInt or EIntegerObject attached.
Mark only one oval.
   - Yes
   - Partially
   - No
   - I don't know

16. **If you had any problems with the above step, please describe them.**

17. **I was able to create the BookCategory enum.**
Your model should contain an EEnum block containing three EEnumLiteral blocks.
Mark only one oval.
- Yes
- Partially
- No
- I don't know

18. **If you had any problems with the above step, please describe them.**

19. **I was able to assign the 'BookCategory' type to the 'category' attribute**
A pointer block to the 'BookCategory' enum block should be attached to your 'category' attribute block.
Mark only one oval.
- Yes
- Partially
- No
- I don't know

20. **If you had any problems with the above step, please describe them.**

21. **I was able to add references to my classes.**
Your EClass blocks should contain EReference blocks.
Mark only one oval.
- Yes
- Partially
- No
- I don't know

22. **If you had any problems with the above step, please describe them.**

23. **I was able to assign types to my references.**
Pointer blocks to your EClass blocks should be attached to your EReference blocks.
Mark only one oval.
- Yes
- Partially
- No
- I don't know

24. **If you had any problems with the above step, please describe them.**

25. **I was able to assign opposites to my references.**
Your 'books' reference block should have a pointer to the 'writer' reference block and vica versa.
Mark only one oval.
- Yes
- Partially
- No
- I don't know

26. **If you had any problems with the above step, please describe them.**

27. **Comments**
If you have any further comments on the test case, please enter them here.

Current functionality of the editor

On this page, we would like to know your opinion of the functionality that is already implemented in the editor.

28. **It is useful to visualize the types of objects and references using the shapes of tabs and slots.**
The shape of the tabs in the upper-left corner of a block represent the type of its object. The shape of a slot represent the type of the reference. Example: https://drive.google.com/open?id=0B1WFkELkcR6XY0pPajAwMlAyZVk
Mark only one oval.
- Strongly disagree
- Disagree
- Neutral
- Agree
- Strongly agree
- I don't know

29. **It is useful to visualize the relationships between types using the shapes of tabs and slots.**
Example: https://drive.google.com/file/d/0B1WFkELkcR6XNUxKdFBjczVNelk/view?usp=sharing - Both EAttribute and EReference 'fit' into the eStructuralFeatures slot without colliding with it, implying they are both subtypes thereof. However, only an EReference pointer can fit into the eOpposite slot. Visually, it can't accommodate an EAttribute, implying the type is incompatible.
Mark only one oval.
- Strongly disagree
- Disagree
- Neutral
- Agree
- Strongly agree
- I don't know

30. **Visualizing the possible drop targets when dragging a block is useful.**
When dragging a block, the editor highlights the slots where it can be dropped.
Mark only one oval.
- Strongly disagree
- Disagree
- Neutral
- Agree
- Strongly agree
- I don't know

31. **It is useful to be able to collapse and expand the block contents.**
The blocks contain forms for editing the attributes of the underlying objects. These forms can be collapsed to save space and expanded when necessary.
Mark only one oval.
- Strongly disagree
- Disagree
- Neutral
- Agree
- Strongly agree
- I don't know

32. **It is useful to display the domain objects in a palette.**
By default, the editor contains a palette with one block for of each type of object that the user can create.
Mark only one oval.
- Strongly disagree
- Disagree
- Neutral
- Agree
- Strongly agree
- I don't know

33. **It is useful to be able to have predefined prototype objects in the palette.**
The palette can contain objects configured so that they function as presets. For example, the Ecore editor palette contains two EReference blocks, one of which is preconfigured as a containment reference: https://drive.google.com/file/d/0B1WFkELkcR6XRmtuazZQaVNUNTQ/view?usp=sharing
Mark only one oval.
- Strongly disagree
- Disagree
- Neutral
- Agree
- Strongly agree
- I don't know

Functionality as a general purpose editor for domain specific languages

The editor you have used is configured as an Ecore editor, meaning it uses Ecore.ecore as the domain model.

However, it is designed to be easily adapted to other domains, by loading a different Ecore model. For example, by loading the library metamodel from earlier, the following editor is automatically generated: https://drive.google.com/file/d/0B1WFkELkcR6XWldoVlRsMFhQaXM/view?usp=sharing

On this page, we would like you to evaluate some of the editor's functionality as a general purpose editor for a domain specific language (DSL).

34. **It is useful to be able to create a domain specific editor by loading an Ecore model directly.**
The editor is designed to be able to work with models created using dynamic EMF as well as models with generated Java code. For example, it is possible to add a feature to a class in the Ecore file, then reload it in the editor. This is as opposed to for example the Sirius framework, where it is required that the edited Ecore model is first loaded into a new Eclipse workbench instance.
Mark only one oval.

- Strongly disagree
- Disagree
- Neutral
- Agree
- Strongly agree
- I don't know

35. **It is useful to be able to reload the mapping model in a running editor.**
The editor is designed to be able to reload the mapping model on the fly. For example, it is possible to change where a slot should appear on a block, or what a tab should look like, without requiring a restart of the editor. This is as opposed to for example the Graphical Modeling Framework (GMF), where it is required that Java code for the editor is first generated, then loaded into a new Eclipse workbench instance.
Mark only one oval.

- Strongly disagree
- Disagree
- Neutral
- Agree
- Strongly agree
- I don't know

36. **It is useful that the editor generates a default mapping model.**
When loading a new domain specification, the editor attempts to generate a default mapping model, which can then be edited by the developer. This is as opposed to for example the Sirius framework, where the developer is expected to build the mapping model from scratch.
Mark only one oval.

- Strongly disagree
- Disagree
- Neutral
- Agree
- Strongly agree
- I don't know

37. **Comments**
If you have any other comments on the current features of the editor, please enter them here.

Potential functionality of the editor

On this page, we would like you to evaluate functionality that we are currently considering to implement.

38. **The editor should be able provide additional feedback about which blocks can be placed on a particular slot.**
When interacting with a slot, the blocks that can be dropped on it should be highlighted in the palette.
Mark only one oval.

- Strongly disagree
- Disagree
- Neutral
- Agree
- Strongly agree
- I don't know

39. **Support for generics is important.**
The types of slots and tabs should be able to represent parameterized types. This would be implemented as described in the Jigsaw Language Toolkit paper: http://ojs.bibsys.no/index.php/NIK/article/download/254/217
Mark only one oval.

◯ Strongly disagree
◯ Disagree
◯ Neutral
◯ Agree
◯ Strongly agree
◯ I don't know

40. **Support for model validation is important.**
The editor should be able to display validation errors and warnings on the blocks.
Mark only one oval.

◯ Strongly disagree
◯ Disagree
◯ Neutral
◯ Agree
◯ Strongly agree
◯ I don't know

41. **Being able to load multiple model resources into the same editor is important.**
The editor should support loading multiple model instances, so that it is possible to create cross references between the models.
Mark only one oval.

◯ Strongly disagree
◯ Disagree
◯ Neutral
◯ Agree
◯ Strongly agree
◯ I don't know

42. **Comments**
If you have any comments on the features described on this page, or requests for features that have not been mentioned, please enter them here.

Qualitative expectations for the editor

On this page, we would like to know what your expectations are for the qualitative aspects of an editor like ours.

43. **It is important that the editor is stable and bug-free.**
The editor must never crash, behave erratically or otherwise produce incorrect or unintentional results.
Mark only one oval.

◯ Strongly disagree
◯ Disagree
◯ Neutral
◯ Agree
◯ Strongly agree
◯ I don't know

44. **It is important that the editor is optimized for demanding conditions.**
The editor must not suffer from poor performance due to slower hardware or big data sets.
Mark only one oval.

◯ Strongly disagree
◯ Disagree
◯ Neutral
◯ Agree
◯ Strongly agree
◯ I don't know

45. **It is important that the editor maintains its usability with large or complex models.**
The editor's user interface must not become unwieldy or otherwise unusable when the scale of the model increases.
Mark only one oval.

◯ Strongly disagree
◯ Disagree
◯ Neutral
◯ Agree
◯ Strongly agree
◯ I don't know

46. **On average, approximately how many classes do you have in your Ecore models?**
Please provide a rough estimate of how many classes there might be in a domain that you normally work with using EMF.
Mark only one oval.

◯ 1-10 classes
◯ 11-50 classes
◯ 51-200 classes
◯ 201-1000 classes
◯ More than 1000 classes
◯ I don't know

47. **On average, approximately how many features do the classes have in your Ecore models?**
How many attributes, references and operations do you typically have in each of your EClasses?
Mark only one oval.

- 1-5 features
- 6-10 features
- 10-20 features
- 21-50 features
- More than 50 features
- I don't know

48. **On average, what is the average number of top-level objects (number of root objects or number of children if there is only one root object) in your domain-specific model instances?**
Mark only one oval.

- 1-10 objects
- 11-50 objects
- 51-200 objects
- 201-1000 objects
- More than 1000 objects
- I don't know

49. **On average, what is the average depth of the containment tree (the length of the longest parent chain) in your domain-specific model instances?**
Mark only one oval.

- 1-10 objects
- 11-50 objects
- 51-200 objects
- 201-1000 objects
- More than 1000 objects
- I don't know

50. **It is important that the editor is suitable for most kinds of models.**
The editor's visual syntax must be able to serve as a good representation of as many kinds of models as possible.
Mark only one oval.

- Strongly disagree
- Disagree
- Neutral
- Agree
- Strongly agree
- I don't know

51. **It is important that the editor supports all EMF functionality.**
The editor must have complete coverage, including generics, unsettability, proxies, etc.
Mark only one oval.

- Strongly disagree
- Disagree
- Neutral
- Agree
- Strongly agree
- I don't know

52. **It is important that the editor's mapping model is highly flexible.**
The editor's mapping model must allow the visual representation to significantly deviate from the actual structure of the model.
Mark only one oval.

- Strongly disagree
- Disagree
- Neutral
- Agree
- Strongly agree
- I don't know

53. **It is important that the editor integrates tightly with the Eclipse platform.**
The editor must feel like an extension of Eclipse, reusing the existing toolbars, menu items, etc.
Mark only one oval.

- Strongly disagree
- Disagree
- Neutral
- Agree
- Strongly agree
- I don't know

54. **It is important that the editor can be used independently of the Eclipse platform.**
The editor must be able to run in a different context than Eclipse.
Mark only one oval.

- Strongly disagree
- Disagree
- Neutral
- Agree
- Strongly agree
- I don't know

55. **Comments**
If you have any other comments on the qualities that you expect from the editor, please enter them here.

In conclusion

56. **Would you be likely to use our editor instead of the sample reflective model editor in Eclipse?**
Mark only one oval.

- ○ Not probable
- ○ Somewhat improbable
- ○ Neutral
- ○ Somewhat probable
- ○ Very probable
- ○ I don't know

57. **Would you be likely to use our editor instead of a framework such as Sirius?**
Mark only one oval.

- ○ Not probable
- ○ Somewhat improbable
- ○ Neutral
- ○ Somewhat probable
- ○ Very probable
- ○ I don't know

58. **Comments**
If you have any comments about the suitability of jigsaw-based languages for modeling, please enter them here.

Powered by Google Forms

Appendix C
Jigsaw EMF Editor Survey Responses

,

Timestamp	12/5/2015 9:17:24	1/21/2016 16:22:09	1/24/2016 20:04:33	1/26/2016 16:42:33	2/2/2016 9:56:20
How familiar are you with the Eclipse Modeling Framework (EMF)?	Moderately familiar	Somewhat familiar	Somewhat familiar	Familiar	Familiar
How familiar are you with Ecore?	Moderately familiar	Somewhat familiar	Somewhat familiar	Familiar	Familiar
How often do you use text-based model editors?	Fairly often	Never	Never	Fairly often	Never
How often do you use graph-based model editors?	Very often	Sometimes	Never	Never	Sometimes
How often do you use form-based model editors?	Fairly often	Sometimes	Never	Almost never	Very often
Have you ever used a visual language based on the jigsaw puzzle metaphor?	Yes	No	Yes	No	No
I was able to create a package for my classes.	Yes	Yes	I don't know	Yes	No
If you had any problems with the above step, please describe them.		I was able to create the package, but I didn't know how to move pre-existing blocks into the package (I tried click-dragging them onto the package tab).	I placed the entirety of my objects in the Resource space.		I forgot to think about packages. Created Classes directly.
I was able to create classes.	Yes	Yes	Yes	Yes	Yes
If you had any problems with the above step, please describe them.					
I was able to assign names to my classes.	Yes	Yes	Yes	Yes	Yes
If you had any problems with the above step, please describe them.					
I was able to add attributes to my classes.	Yes	Yes	Yes	Yes	Yes
If you had any problems with the above step, please describe them.					
I was able to assign standard types to my attributes.	Partially	Yes	Yes	Yes	No
If you had any problems with the above step, please describe them.	It took a while before I found them. I was looking in the classes' properties for a "button". Then I found it in the palette. It also took me some time to assign them -> drop the block behind the attribute. I would have been more "scratchy" to have a placeholder "inside" the attribute block.			Took a while for me to find the tab "Other types". Instead I tried to make EDataType fit the eType hook.	Could not find a way of specifying type
I was able to create the BookCategory enum.	Yes	Yes	Yes	Yes	Yes
If you had any problems with the above step, please describe them.					
I was able to assign the 'BookCategory' type to the 'category' attribute	Yes	Yes	Yes	Yes	No
If you had any problems with the above step, please describe them.				Took me a while to figure out how to do this (aprox. 5 min). Right-clicked the enum and clicked "Create pointer block".	Could not find an intiutive way of adding type. I have added the attribute, but struggle with adding the type.
I was able to add references to my classes.	Partially	Yes	Yes	Yes	Partially
If you had any problems with the above step, please describe them.	It was ok, but I was not sure if the reference block should be standalone or part of the class block. The shape of the class (upper left corner) indicates that you can create some kind of puzzle with the blocks. This is not possible as they should be "inside" the block.			Same problem as the previous step.	
I was able to assign types to my references.	Partially	Yes	Yes	Yes	No
If you had any problems with the above step, please describe them.	Right-Clicking was not obvious. Drag-drop was the concept all the way and I had to try several gestures to find the pointer block. Highlighting of connector points should be even more visible.	I was unsure at first how to assign pointers for the types (the error message was cut off) but eventually figured it out.		By now I realized how this worked.	
I was able to assign opposites to my references.	Yes	Yes	Yes	Yes	No
If you had any problems with the above step, please describe them.	When I found the pointer blocks it as ok				I wanted to create the same references as in the uml diagram. I could add references to the classes but I could not find to specify the references such that they map both a source and a target. For instance, I wanted to create a ref between book and writer. I could add a reference to either book or writer, but I could not fint our how to map it to the other endpoint.

Comments	Interesting concept. As a modeler with experience from RSM, RSA, TGA, Papyrus, EA etc, it is hard to change the way of thinking wrt references. Color scheme should be changed. Red indicates errors/warnings. Look at color.adobe.com				
It is useful to visualize the types of objects and references using the shapes of tabs and slots.	Neutral	Agree	Agree	Agree	Agree
It is useful to visualize the relationships between types using the shapes of tabs and slots.	Disagree	Agree	Agree	Agree	Agree
Visualizing the possible drop targets when dragging a block is useful.	Strongly agree	Strongly agree	Strongly agree	Strongly agree	Neutral
It is useful to be able to collapse and expand the block contents.	Strongly agree	Agree	Strongly agree	Agree	Neutral
It is useful to display the domain objects in a palette.	Strongly agree	Strongly agree	Strongly agree	Agree	Agree
It is useful to be able to have predefined prototype objects in the palette.	Agree	Agree	Agree	Agree	Neutral
It is useful to be able to create a domain specific editor by loading an Ecore model directly.	Agree	Agree	Agree	Agree	Neutral
It is useful to be able to reload the mapping model in a running editor.	Strongly agree	Strongly agree	Agree	Agree	Strongly agree
It is useful that the editor generates a default mapping model.	Strongly agree	Agree	Strongly agree	Agree	Neutral
Comments					
The editor should be able provide additional feedback about which blocks can be placed on a particular slot.	Strongly agree	Strongly agree	Agree	Neutral	Strongly agree
Support for generics is important.	Agree	Strongly agree	I don't know	I don't know	Agree
Support for model validation is important.	Strongly agree	Strongly agree	Agree	Agree	Agree
Being able to load multiple model resources into the same editor is important.	Strongly agree	Agree	I don't know	I don't know	Agree
Comments	In real life situations, validation is extremely important. When the users invest lots of energy and time in creating a formal and detailed model the would like to have feedback (RoI) before code generation. Early validation can be the key selling point for DSML .				
It is important that the editor is stable and bug-free.	Strongly agree	Strongly agree	Strongly agree	Agree	Strongly agree
It is important that the editor is optimized for demanding conditions.	Strongly agree	Strongly agree	Strongly agree	Agree	Agree
It is important that the editor maintains its usability with large or complex models.	Strongly agree	Strongly agree	Strongly agree	Agree	Agree
It is important that the editor is suitable for most kinds of models.	Agree	Strongly agree	Strongly agree	Disagree	Disagree
It is important that the editor supports all EMF functionality.	Neutral	Strongly agree	I don't know	Neutral	Neutral
It is important that the editor's mapping model is highly flexible.	Neutral	Agree	I don't know	Disagree	Neutral
It is important that the editor integrates tightly with the Eclipse platform.	Agree	Strongly agree	Disagree	Agree	Disagree
It is important that the editor can be used independently of the Eclipse platform.	Neutral	Strongly agree	Strongly agree	Disagree	Agree
Comments	For industri, stability is of utmost importance. As modeling deals with complexity, most real-life models are complex - and large. So, scalability is important. Sirius uses a benchmark with 1 million elements in a model.				
Would you be likely to use our editor instead of the sample reflective model editor in Eclipse?	Somewhat probable	Somewhat probable	Somewhat improbable	Somewhat improbable	I don't know

Would you be likely to use our editor instead of a framework such as Sirius?	Somewhat improbable	Neutral	Somewhat probable	Somewhat probable	I don't know
Comments	It is important to investigate these kinds of concepts. From the top of my head I think it would be even more important to investigate how an editor could better support a hierarchical design - allowing the designer to zoom in and out of details - not only through expand-collapse and layering (sirius) - but a more streamlined way of looking into packages, classes, interfaces, attributes. Many users would like to double click to see the details "inside" an element. This must be explicitly design in sirius and most other uml tools (composite).			emf4life	
On average, approximately how many classes do you have in your Ecore models?		11-50 classes	I don't know	1-10 classes	I don't know
On average, approximately how many features do the classes have in your Ecore models?		10-20 features	I don't know	1-5 features	I don't know
On average, what is the average number of top-level objects (number of root objects or number of children if there is only one root object) in your domain-specific model instances?		I don't know	I don't know	1-10 objects	1-10 objects
On average, what is the average depth of the containment tree (the length of the longest parent chain) in your domain-specific model instances?		1-10 objects	I don't know	1-10 objects	1-10 objects