Jigsaw Language Toolkit

Oddvar Hungnes (oddvahu@stud.ntnu.no)
Hallvard Trætteberg (hal@idi.ntnu.no)
Dept. of Computer and Information Science, NTNU

Abstract

In recent years, graphical programming systems based on the jigsaw puzzle metaphor have gained popularity due to their ability to visualize syntactical constraints. The supported languages usually have very simple type systems and little to no static type checking. The jigsaw puzzle metaphor has the potential to visualise the constraints and possibilities in languages with more complex types.

In this paper, we describe the design and implementation of a reusable GUI toolkit based on the jigsaw puzzle metaphor. The toolkit supports concepts from object oriented design, such as type polymorphism, inheritance and generics.

1 Introduction

Visual languages are formal languages that use a graphical representation rather than a textual one. In visual modeling, people create graphical representations of domains of interest using such languages.

Graph-like visual languages [1], constitute a very common class of visual languages, e.g. UML [2] is a popular language in this genre which aims to unify the modeling practice with a general purpose visual software modeling language. However, it is unreasonable to assume that there exists a one-size-fits-all solution to the visual modeling problem (or textual for that matter); we sometimes need domain-specific languages(DSLs). [3] Furthermore, we believe that it is worthwhile to investigate alternative visual syntaxes which may be more suitable for specific domains.

In the field of visual programming languages, the jigsaw puzzle metaphor has gained popularity in recent years, in particular with novice programmers. A jigsaw puzzle is a puzzle where a picture has been cut into pieces, and the goal is to assemble the pieces back together, as shown left in Figure 1.

This paper was presented at the NIK-2015 conference; see http://www.nik.no/.

Figure 1: Left, an assembled jigsaw puzzle. Right, disassembled pieces, with the tab (red, left) and the slot (green, right) highlighted.

Visual programming languages based on the jigsaw metaphor use interlocking pieces, or ”blocks”, as the core element. [4] An important aspect is how the constraints of the language are explicitly visualised as the shapes of the tabs and slots on the blocks. This enables users to immediately understand what connections can and cannot be made.

Our aim is to make visual languages based on the jigsaw metaphor available to other disciplines; visual *modeling* in particular. In this paper, we describe the design and implementation of a reusable graphical user interface (GUI) toolkit, that can provide an alternative visual syntax for DSLs.

The structure of the paper is as follows. We first present related work related modeling languages and jigsaw-based visual programming. In subsequent sections we present requirements for the toolkit, describe the design (both user interaction and visualisation techniques) and implementation, with focus on conceptual issues and special challenges. We focus these sections on features that are of particular interest for supporting a DSL framework. Finally, we summarise our results.

2 Related Work

The Unified Modeling Language (UML) specifies a syntax and semantics for modeling object-oriented systems [5]. UML is a general purpose modeling language, intended to be used across a wide spectrum of domains. UML class diagrams or similar notations are often used to model the concepts in domains that are the target of DSLs and their relationships. Figure 2 shows the different types of relationships between classes of objects: [6]

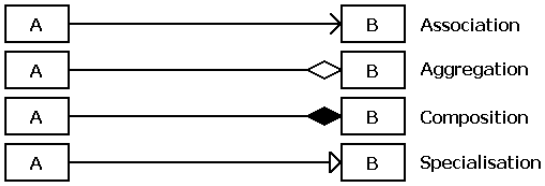

Figure 2: The different relationships between classes in UML.

- Association - Denotes that there can be a semantic relationship between instances of the associated classes.
- Aggregation and composition - These are binary association where an instance of a class may be part of another object of the same or other class. In case of composition, an aggregated object can only be a part of at most one composite object. This is the case in hierarchical Jigsaw-based languages.

- Specialisation - Denotes that the specialising class, the subtype, inherits the features of the the generalising class, the supertype, and may redefine or add features. This carries a notion of substitutability: An instance of the subtype can always be used where an instance of the supertype is expected, but not the other way around.

The Eclipse Modeling Framework (EMF) is a general-purpose modeling tool for the Eclipse platform. Its underlying modelling language, Ecore, supports a small subset of UML for defining the abstract syntax of DSLs, including inheritance, binary associations and composition/containment [7]. EMF also supports generics: Model classes can have type parameters, which serve as placeholders for a concrete type.

A Domain-Specific Language (DSL) is a formal language specialised for a particular domain. DSLs complement general langauges like UML, and allows for smaller languages that are easier to learn and use for both experts and non-experts, with more precise semantics and hence possibilities for executability. Such languages consist of an abstract syntax, defined by a metamodel, and one or more concrete syntaxes. [8] E.g. the Ecore language mentioned above has a UML-like graphical syntax, a Java-like textual syntax and XML- and Json-based syntaxes. In addition, these are often complemented by tree views, tables and forms.

Various EMF-based frameworks provide tools for building editors with specific concrete syntaxes, e.g. Xtext supports textual editors [10], while Sirius supports diagrams, trees and tables. [9]

Jigsaw-based languages

Scratch is a popular jigsaw-based visual programming environment designed for the development of simple graphical games, based on programmed ”sprites”. It was inspired by an older system called StarLogo. [12]

The programming tab in Scratch contains a palette and a working area, as seen in Figure 3.

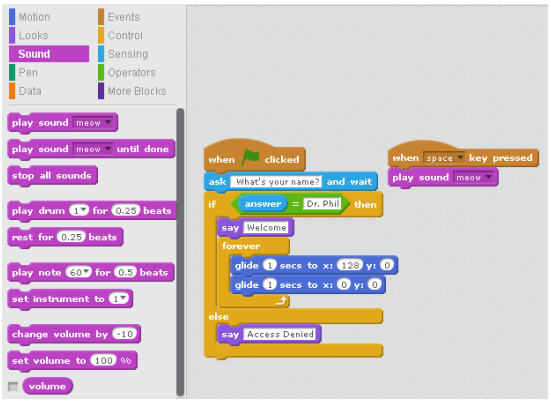

Figure 3: Screenshot of the programming tab in Scratch

The palette contains a single instance of each block type. These blocks can be copied into the working area, moved around and assembled, by means of drag and drop. Fundamentally, there are are two distinct types of blocks, corresponding to statements and expressions.

Statement blocks have a tab on the bottom and a slot at the top, are usually assembled into lists with special control structure blocks. Expression blocks do not have tabs or slots. Instead, they have a contour with a particular shape. The contour's shape reflects the blocks data types and corresponds to the shape of a hole which accepts that type as an input. Both expression blocks and statement blocks can have such holes.

Scratch' underlying domain model has two important characteristics:

- All associations between blocks represent composition in nature. This is important because there is no visual syntax for general associations or mechanism for attaching a block to more than one parent block.
- A very limited subset of the block types is the target of an association. This is important because each type with this property must have an unique slot and tab, or an unique contour shape in the visual syntax.

It is not difficult to imagine that the visual syntax used in Scratch could be used as a concrete syntax for other DSLs that share similar constraints. E.g. Google Blockly [13] is a project that allows other applications to use a similar concrete syntax, while Lego Mindstorms is a robotics construction kit with programming system for, primarily aimed at a young audience.

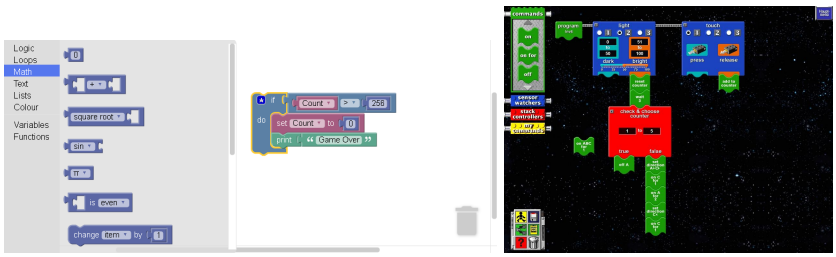

Figure 4: Screenshots of Google Blockly (left) and Lego Mindstorms [**?**] (right)

Many systems have been derived from Scratch, to make it suitable for other kinds of programming. Snap! [15] extends Scratch with constructs found in modern programming languages, including first class functions. However, it does not support parameterised types (generics) that is useful in a DSL, and since it is implemented in Javascript it cannot be used for implementing a DSL framework. OpenBlocks [14] is a Java library built for such a puropose, but it does not support the kind of type system we need, and uses Java Swing, which with Java 8 is being replaced by JavaFX.

3 Requirements

Our goal is to design and implement a reusable toolkit for making DSLs based on the jigsaw metaphor. In this section, we specify the requirements for our toolkit, for the following two stakeholders:

- The application developer, who uses our toolkit directly to create applications based on the jigsaw puzzle metaphor.
- The end user, who uses the toolkit indirectly through the application.

Here, we focus on requirements that enable the toolkit to support a DSL framework.

In addition to the hierarchical block structure, the toolkit should be able to represent non-compositional associations. The end user should be able to create multiple references to the same block.

The application developer must be able to create a type hierarchy and assign types to tabs and slots, that determine their shape. Types must support type parameters, and it should be possible for the end user to specify type arguments for those type parameters.

The application developer should be able to customize the shapes of slots and tabs without writing any source code. The toolkit should be able to automatically generate the shapes of slots and tabs.

To allow editing addition block properties, blocks should be capable of holding conventional user interface widgets that are displayed to the end user.

The application developer should be able to integrate the panels that the toolkit provide in any JavaFX application, using FXML [16]. The toolkit should provide templates for common workspace layouts, e.g. a block palette on the left side and a working area on the right side should be provided.

4 Design

The Block

A block in our toolkit has a type, represented by its tab(s). A block can have any number of slots, which act as receptacles for child blocks. The application developer may install arbitrary user interface widgets into a block, such as a text input field. A block's size expands or contracts according to the sizes of its tabs, slots, widgets and child blocks. We allow a block to have both a horizontal and a vertical tab as long as it is not connected, and then remove the unused tab once it has been connected to a slot, as shown left in in Figure 5.

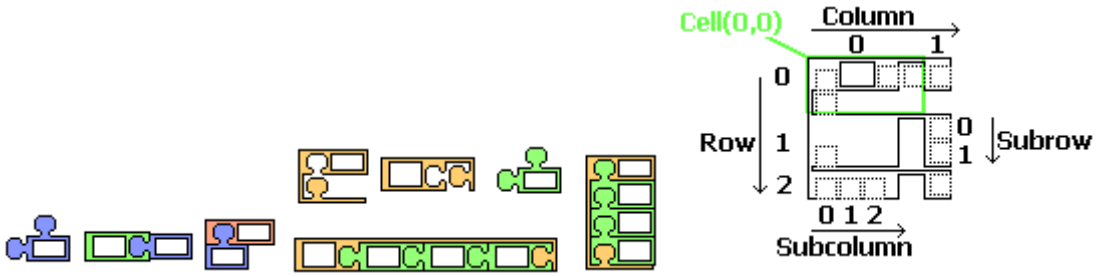

Figure 5: Examples of blocks and block structures.

Our support of list slots, where multiple child blocks may be added, generalise Scratch' "C-blocks". Firstly, we will allow C-blocks that are rotated so that blocks are laid out horizontally, as shown in the middle of Figure 5.

In Scratch, a C-block can have multiple rows, exemplified by the if-else block. Combining this with our rotated C-blocks leads us to a table-like structure. An example of this is

shown right in Figure 5. This is reasonably simple to work with if each row and each column has a uniform size.

References

As mentioned above, blocks form typically a strict tree, while in box-and-arrow-based modeling systems, a box can typically have any number of arrows pointing into it. In our system, we combine these two approaches. To create additional references to a block, a ”pointer block” is created, that refers to the original block (see Figure 12). The pointer block can be substituted for the original block in any slot where it would be accepted. An arrow is be rendered from the pointer block to the original block to indicate the connection. This allows us to reuse all the qualities of blocks without any major alterations to the mechanics of user interaction.

The Palette and Working Area

The toolkit provides two main panels, the Palette and Working Area. Blocks are snapped together in one or more hierarchical block clusters, by dragging block prototypes from a palette and dropping them into the Working Area. Child blocks are attached to slots on the parent block via their tabs.

We allow the application developer to be able to configure the workspace the way they want, for example by dividing the palette into tabs or bringing up two different working areas side by side. An example of how the palette and working area might be organised is shown in Figure 6.

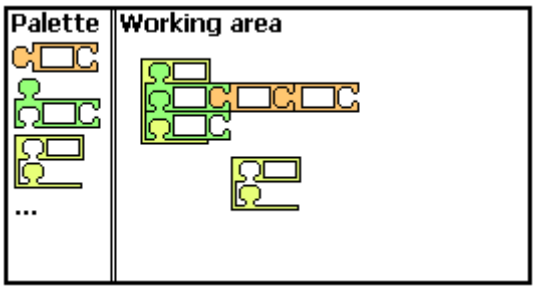

Figure 6: An example of a workspace with a palette to the left and a working area to the right.

Polymorphism

We support setting the types of tabs and slots, and this constrains which tabs and slots fit together. We want the tab shape associated with a type to be formed in such a way that it fits into the slot shape of its super-types, but not the other way around. A general, compact and intuitive way of achieving this is to let a type’s contour shape resemble the contour shape of its super-type, but with a small piece cut away, as illustrated in Figure 7. The piece that is cut away must be carefully chosen.

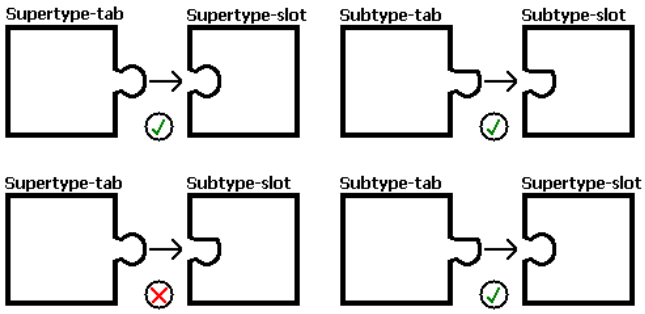

Figure 7: The contour shape of the subtype resembles that of the super-type, but with a piece cut away. Notice how the super-type tab would intersect with the subtype slot if it was inserted.

With multiple inheritance, the type hierarchy is a rooted directed acyclic graph, and not necessarily a tree. Assume we have types S1 and S2, of which neither is an ancestor of the other, and a type C1, which inherits from both types. C1's tab must fit into the slots of both S1 and S2, but it must not be possible to connect S1 and S2 in any way. For this to work, C1 needs to inherit the cuts from both parents, as well as possibly defining its own cut (see Figure 8).

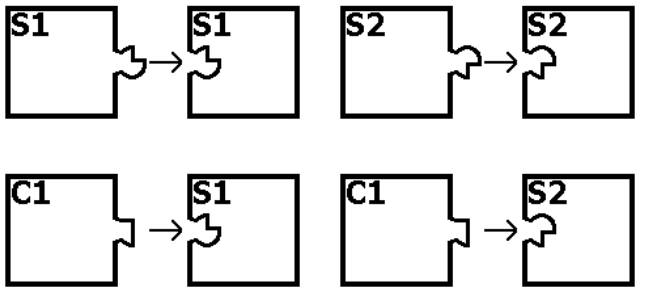

Figure 8: C1 inherits from both S1 and S2, but S1 and S2 do not inherit from each other. C1 inherits both of their cuts, as well as defining its own cut. This ensures that it fits into both slots.

Generics and type parameters

A type can be generic by having type parameters, and a generic type can be parameterised by providing type arguments for the type parameters. In order to render slots or tabs with parameterized types, we may consider a few different approaches. The type contours of the generic type and the type arguments could be blended *on top of* each other. However, since the contour shapes of the different types can take any appearance, the result would be unpredictable and in most cases difficult to interpret. The type contours could be rendered *side by side*. Unfortunately, this would likely lead to confusion since ordinary slots can also be laid out side by side. The type contours could be composed in a *hierarchical* manner, so that type contours on a deeper level are extruded from the type contours on the more shallow level.

The hierarchical approach makes sense since it matches how parameterised types are declared. To indicate that the type arguments are additional information and not part of

the generic type itself, they can be rendered using partial transparency. This is illustrated in Figure 9.

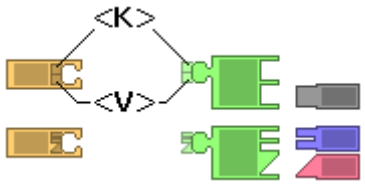

Figure 9: An example of how parameterized slots and tabs might look. The tabs on the green blocks and the slots on the orange blocks are parameterized over <K> and <V>. On the top, the type of the gray block is used for both type arguments. On the bottom, the type of the blue block is used as the argument for <K> and the type of the pink block is used as the argument for <V>. Note how the type argument shapes are extruded from the generic type shape.

We use separate drag-and-drop mechanism for assigning concrete types, using type sockets and type chips, as illustrated in Figure 10.

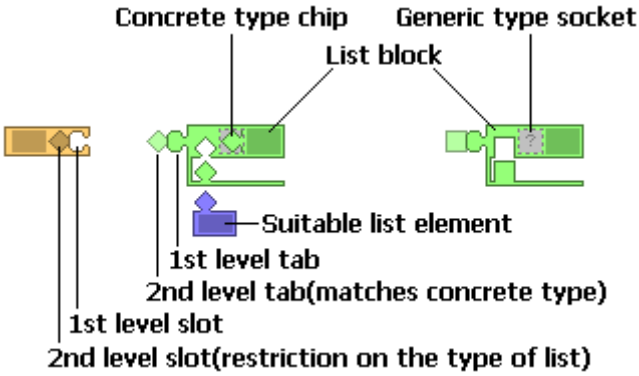

Figure 10: A list block (green) can be parameterized by dropping a type chip onto the type socket on the block. The type of the blue block has been used as the type argument to the leftmost list block, altering the type of the list slot. The type argument is reflected in the 2nd level tab. The rightmost list block has no concrete type chip assigned to it, so the type argument defaults to the most general type. Only the leftmost list block can fit into the orange block's slot due to the slot's parameterization.

A type parameter can be bounded on a particular type, placing restrictions on which type arguments are legal. Upper bound means the type argument must be equal to, or a sub-type of the bounding type. Lower bound means the type argument must be equal to, or a super-type of the bounding type.

If the visualisation technique from Figure 10 and Figure 9 is used, then it is natural to render the slot shape of the upper bounding type in the type socket. Since a type chip is rendered as the tab shape of the concrete type, then the chip will fit if and only if it is equal to, or a subtype of the bounding type(section 4). Furthermore, the upper bounding mode is able to cover most cases.

The same reasoning can be used for type parameters of slots. It is the upper bounded mode that is intuitively represented using our visualization techniques, because this is

how our tabs and slots are designed. If a different bounding mode is required, then the slot can be marked with a symbol indicating the special behaviour.

It is also possible for a tab or a slot to use a generic type directly, without specifying type arguments. When a generic type is used in this manner, it is referred to as a raw type. If a slot has a raw type, it accepts tabs with that raw type, or any parameterization of it. If a tab has a raw type, it is only accepted by a slot with that raw type.

5 Implementation

Architecture

Our toolkit is implemented using version 1.8 of the Java programming language and the JavaFX GUI library. The code is available from `https://bitbucket.org/Oddwarg/javafx-jigsaw/`, as well as an example application that waS used to validate the requirements.

A class diagram of the most important classes in our system is shown in Figure 11. The type hierarchy is separated from the hierarchical block structure, to make the former easily configurable.

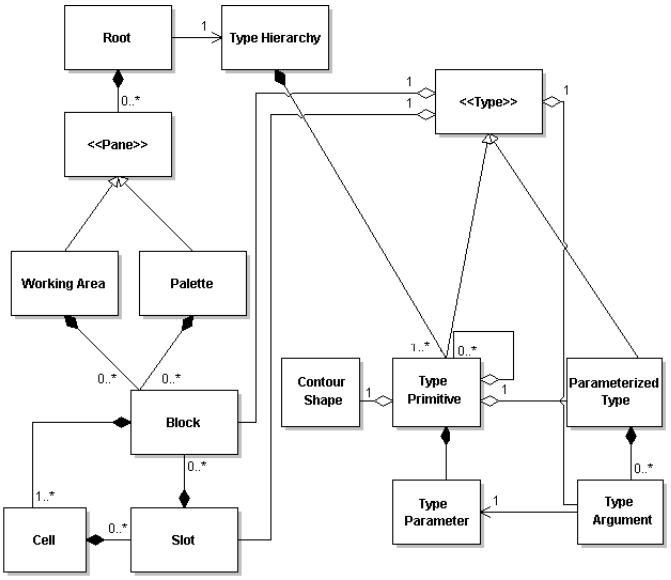

Figure 11: An overview of the most important model classes in our system.

The block structure supports both the standard hierarchy and cross-references, as shown in Figure 12.

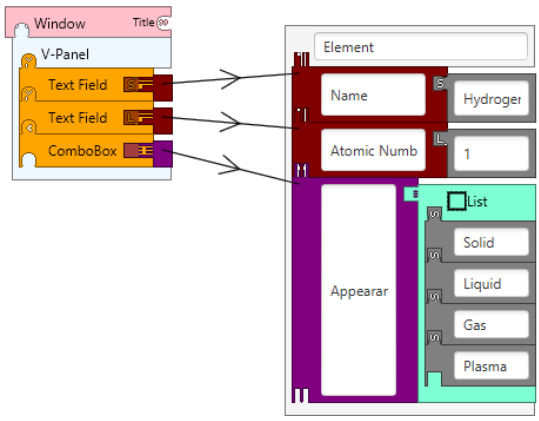

Figure 12: Multiple references to field blocks, using pointer blocks.

The Palette and Working Area are typically embedded in GUI panels as illustrated in Figure 13. However, the Working Area and Palette may be created with FXML, as parts of a larger application structure file. E.g. there may be a single Palette object or tabbed pane which a Palette in each tab. This gives the application developer a large degree of freedom when it comes to determining the user interface structure.

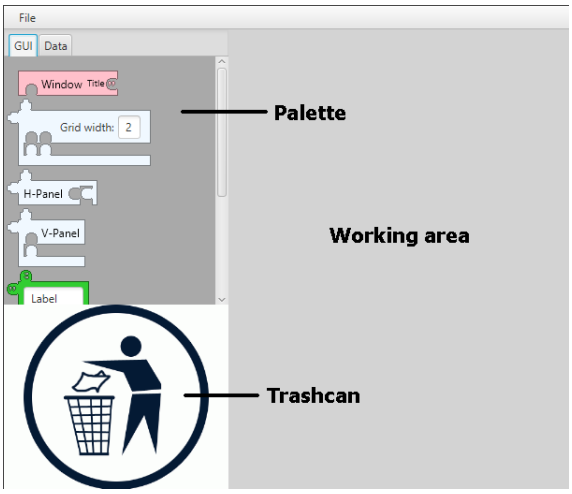

Figure 13: A screenshot of the window builder test application.

Types, Tabs and Slot Shapes

The Type Hierarchy is assembled from Type Primitive that know their super-types. As a convenience for the application developer, Java Class objects can be wrapped into an appropriate Type Primitive implementation. Tab and slot shapes are generated from basic shapes and cuts, as illustrated in Figure 8, based on the type hierarchy. The toolkit provides automatic generation of cuts, as well as a cut editor (see Figure 14), in order to produce tabs and slots that are more appropriate for a given domain or more easily distinguishable from one another. We considered both vector-based and bitmaps and ended up using the latter, as they are easier to compose and edit.

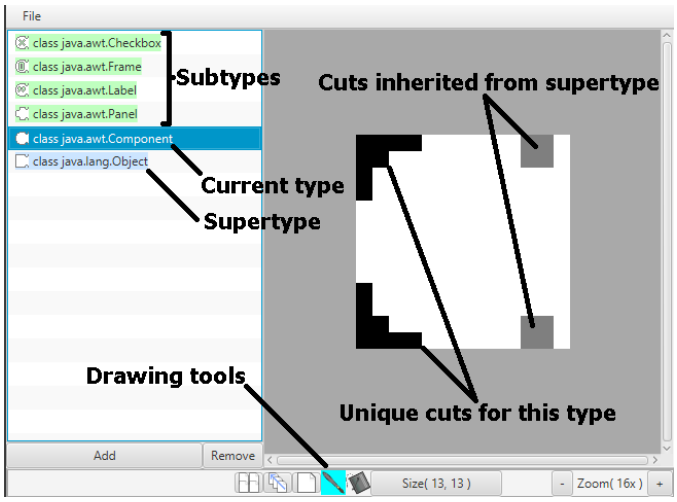

Figure 14: A screenshot of the cut editor for designing tabs and slots in our toolkit.

6 Conclusion

We have developed a new graphical user interface toolkit for developing applications based on the jigsaw puzzle metaphor. Our toolkit can be used to implement a new kind of visual concrete syntax for domain-specific languages.

Our visual syntax expands upon techniques found in existing jigsaw-based visual programming systems. We have developed new techniques for representing non-containment references between blocks, as well as static type checking with polymorphism and generics. In particular, our new syntax allows users to determine, upon inspection of tabs and slots, what connections can and cannot be made between objects.

We implemented a test application to evaluate our toolkit, and found that it satisfied the stated requirements. In the future, we would like to implement tools for creating mappings between existing metamodels and our concrete syntax. These tools should be tested against several domains, and the resulting editors should be tested by users who are familiar with existing modeling languages.

References

[1] 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.

[2] Object Management Group, Inc. Retrieved 03. Feb 2015, Graphite Website, Available: `http://www.uml.org`

[3] 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.

[4] Maloney, John, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. ”The scratch programming language and environment.” ACM Transactions on Computing Education (TOCE) 10, no. 4 (2010): 16

[5] Quatrani, Terry. ”Visual modeling with rational rose 2002 and UML.” Addison-Wesley Longman Publishing Co., Inc., 2002.

[6] Object Management Group, Inc. ”OMG Unified Modeling Language (OMG UML), Infrastructure.” Version 2.4.1, 2011-08-05

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

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

[9] The Eclipse Foundation. Retrieved 04. May 2015, ”Sirius Overview.” Available: http://eclipse.org/sirius/overview.html

[10] The Eclipse Foundation. Retrieved 04. May 2015, ”Xtext.” Available: http://eclipse.org/Xtext/

[11] MIT. Retrieved 31. July 2015, ”Scratch - Imagine, Program, Share.” Available: https://scratch.mit.edu

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

[13] Google. Retrieved 19. Feb 2015. ”Google Blockly,” Available: https://developers.google.com/blockly/

[14] Roque, Ricarose Vallarta, ”OpenBlocks : an extendable framework for graphical block programming systems”. Master thesis, MIT, 2007. Available: http://hdl.handle.net/1721.1/41550

[15] University of California, Berkeley. Retrieved 31. August 2015, ”Berkeley Snap! (Build Your Own Blocks) 4.0.” Available: snap.berkeley.edu/

[16] Sharan, Kishori. ”Understanding FXML.” Learn JavaFX 8. Apress, 2015. 1120-1156.