Université de Montréal

Extending Domain-Specific Modeling Editors with Multi-Touch Interactions

par

Md Rifat Hossain

Département d'informatique et de recherche opérationnelle (DIRO)
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

Orientation Génie logiciel

March 30, 2021

© Md Rifat Hossain, 2021

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Extending Domain-Specific Modeling Editors with Multi-Touch Interactions

présenté par

Md Rifat Hossain

a été évalué par un jury composé des personnes suivantes :

Abdelhakim Hafid
(président-rapporteur)

Eugène Syriani
(directeur de recherche)

Houari Sahraoui
(codirecteur de recherche)

Michalis Famelis
(membre du jury)

Résumé

L'ingénierie dirigée par les modèles (MDE) est une méthodologie d'ingénierie logiciel qui permet aux ingénieurs de définir des modèles conceptuels pour un domaine spécifique. La MDE est supportée par des outils de modélisation, qui sont des éditeurs pour créer et manipuler des modèles spécifiques au domaine. Cependant, l'état actuel de la pratique de ces éditeurs de modélisation offre des interactions utilisateur très limitées, souvent restreintes à glisser-déposer en utilisant les mouvements de souris et les touches du clavier. Récemment, un nouveau cadre propose de spécifier explicitement les interactions utilisateur des éditeurs de modélisation. Dans cette thèse, nous étendons ce cadre pour supporter les interactions multitouches lors de la modélisation. Nous proposons un catalogue initial de gestes multitouches pour offrir une variété de gestes tactiles utiles. Nous démontrons comment notre approche est applicable pour générer des éditeurs de modélisation. Notre approche permet des interactions plus naturelles pour l'utilisateur quand il effectue des tâches de modélisation types.

Mots-clés: Environnement de développement intégré, l'ingénierie dirigée par les modèles, multi-touches, modélisation spécifique au domaine

Abstract

Model-driven engineering (MDE) is a software engineering methodology that enables engineers to define conceptual models for a specific domain. Modeling is supported by modeling language workbenches, acting as editor to create and manipulate domain-specific models. However, the current state of practice of these modeling editors offers very limited user interactions, often restricted to drag-and-drop with mouse movement and keystrokes. Recently, a novel framework proposes to explicitly specify the user interactions of modeling editors. In this thesis, we extend this framework to support multi-touch interactions when modeling. We propose an initial set of multi-touch gesture catalog to offer a variety of useful touch gestures. We demonstrate how our approach is applicable for generating modeling editors. Our approach yields more natural user interactions to perform typical modeling tasks.

Keywords: Integrated development environment, model-driven engineering, mutil-touch, domain-specific modeling

Contents

Résumé 5

Abstract 7

List of figures 13

List of acronyms and abbreviations 15

Acknowledgements 17

Chapter 1. Introduction 19

1.1. Context 19

1.2. Problem statement and thesis proposal 19

1.3. Contributions 20

1.4. Outline 21

Chapter 2. Background and state of the art 23

2.1. Model driven engineering 23

2.1.1. Domain-specific modeling 24

2.1.1.1. **Abstract syntax** 24

2.1.1.2. **Concrete syntax** 25

2.1.1.3. **Semantics** 25

2.1.1.4. **Pragmatics** 25

2.1.1.5. **Metamodel** 25

2.1.2. User interaction with modeling editors 25

2.2. Modeling the interaction of DSM editors 27

2.2.1. Interface model 27

2.2.2. Interaction model 27

2.2.3. Event mapping model 27

2.2.4. Process of generating interactive DSM editors 28

2.3. Programming by touch 28

2.4. Interactive touch libraries 29

Chapter 3. Multi-touch domain-specific modeling editor 33

3.1. Multi-touch gestures for domain-specific modeling 33
3.1.1. List of gesture events and actions 34
3.1.2. Assigning semantics to gestures 35
3.1.3. Resolve conflicting mapping event 37

3.2. Domain-specific interactive modeling 37

Chapter 4. Implementation and examples 39

4.1. Defining multi-touch modeling editors 39
4.1.1. Implementation of multi-touch events 40
4.1.2. Interaction rules 41
4.1.3. Event mapping 43

4.2. Application examples 44
4.2.1. Game of Life simulator 45
4.2.1.1. **Metamodel** 45
4.2.1.2. **Interface model** 45
4.2.1.3. **Interaction model** 45
4.2.1.4. **Event mapping model** 47
4.2.2. Pac-Man Game configurator 47
4.2.2.1. **Metamodel** 47
4.2.2.2. **Interface model** 48
4.2.2.3. **Interaction model** 49
4.2.2.4. **Event mapping model** 51
4.2.3. Mind Map editor 53
4.2.3.1. **Metamodel** 53
4.2.3.2. **Interface model** 54
4.2.3.3. **Interaction model** 54
4.2.3.4. **Event mapping model** 56

4.3. Discussion 56

Chapter 5. Conclusion 59

5.1. Summary 59
5.2. Outlook 59
Bibliography 61

List of figures

2.1 MDE architecture and example [**4**] ... 24
2.2 Graphical framework examples ... 26
2.3 Multi-touch gesture behavior [**52**] ... 30
3.1 An example of defining UML diagram by touch gestures ... 38
4.1 Architecture of our approach ... 40
4.2 GoL interface model ... 45
4.3 GoL creation a deletion of cells ... 47
4.4 Interaction with the play button of GoL ... 48
4.5 The metamodel of the conceptual aspect of Pac-Man ... 48
4.6 Pac-Man interface model ... 49
4.7 An example of Pac-Man user interactions ... 51
4.8 An example of Pac-Man user interactions ... 52
4.9 An example of Pac-Man user interactions ... 52
4.10 An example of Pac-Man user interactions ... 52
4.11 The metamodel of Mind Map ... 53
4.12 An example of a Mind Map created by the touch editor ... 56

List of acronyms and abbreviations

API	Application Programming Interface
DSL	Domain-Specific Language
DSM	Domain-Specific Modeling
DSML	Domain-Specific Modeling Language
EGL	Epsilon Generation Language
EMF	Eclipse Modeling Framework
GMF	Graphical Modeling Framework
I/O	Input/Output
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering

MOF	Meta Object Facility
OCL	Object Constraint Language
UML	Unified Modeling Language

Acknowledgements

I want to express my sincere gratitude to my supervisor Prof. Eugène Syriani and co-supervisor Prof. Houari Sahraoui for their collaboration and mentoring in this research project. Their experience and insights were invaluable in developing my work.

I also want to acknowledge my colleagues at the GEODES lab for their help, advice and support. In particular, I would like to thank Vasco Sousa and Khady Fall who never hesitated to help me and share useful discussions and suggestions and provided their useful insights whenever required.

I dedicate this thesis to my parents, brothers and sister. I have been blessed with their wise advice, unwavering support and prayers throughout my life.

Chapter 1

Introduction

1.1. Context

Software engineering research aims at proposing novel solutions to improve software development quality and productivity. To achieve this goal, many software development techniques have been defined in last decades. Among them, a new paradigm gathers much attention by targeting domain-specific problems using Model-Driven Engineering (MDE) [**5, 31, 44**]. MDE helps, among others, to create conceptual models for a specific domain or a precise aspect of a system. Typically, the models are designed using a dedicated modeling language, in the form of domain-specific languages (DSL) [**21**]. DSLs are defined using a precise syntax, which can be visual or textual. They allow to express the solution to a problem at a level of abstraction closer to the concerned application domain [**53**].

The MDE community proposed language workbenches that allow us to define domain-specific languages. When the abstract syntax of a domain-specific language is specified, one can generate a domain-specific editor by explicating the concrete syntax. Editors allow domain experts design their models either textually or graphically. In this thesis, we focus solely on graphical editors. A graphical editor allows the designers to define models by creating a set of interrelated graphical elements shown in the diagrams. Many graphical modeling editors have been investigated [**1**]. In addition to defining the concrete syntax, those offer interaction features to create and manipulate model elements [**11**]. However, to the best of our knowledge, none of them integrates multi-touch interactions.

1.2. Problem statement and thesis proposal

Custom user interfaces and interactions contribute to increase efficiency and productivity in a modeling environment. The interactions can be traditional ones using keyboard and mouse devises or multi-touch supportive. Defining explicitly user interactions and their

semantics, rather than using default ones, helps to better address domain-specific modeling tasks. The idea behind our work is motivated by the following observations:

- Current language workbenches generate domain-specific modeling editors with no tactile support. Interactions are quite unnatural and may be repetitive, as most of the modeling task is done mainly with drag and drops, using mouse and keyboard.
- With the popularity of portable devices, i.e., phones, tablets, and laptops with touch screens, end-users and developers are now naturally comfortable with touch interactions. Touch interactions offer a variety of gesture possibilities that can be associated with specific activities and events. A touch interface allows more natural (e.g., moving an element independently in any coordinates on the canvas), fluid, and continuous movements. Additionally, the user can explore more dimensions with multi-touch while a mouse and keyboard have limited basic actions (point, move, and clicks). In conclusion, using multi-touch opens up more possibilities of interactions.

Multi-touch gesture interface refers to an interface where more than one contact point can be considered simultaneously in the interactions. There are several contributions that targeted the association of semantics to gestures. This is done in general for drawing editors such as in [**26**], or specifically for model editors to sketch models as in [**56**]. However, the above-mentioned research contributions only consider a single touch gesture at a time rather than handling multiple gestures together.

This thesis proposes an initial set of multi-touch gestures that help a domain expert or modeling engineers design their models through multi-touch interactions instead of using traditional mouse and keyboard. This thesis also provides a framework to assign to these gestures domain specific interaction semantics.

1.3. Contributions

This thesis aims to help domain experts define domain-specific gestures and their association with modeling tasks in order to produce more natural modeling environments. The specific contributions of this thesis are the following:

- A prototype of domain-specific editors enabled multi-touch interactions with an initial set of gestures for domain-specific modeling.
- An implementation of a library of gestures that can be plugged into modeling editors.
- A generation process of modeling editors enabled with domain-specific multi-touch interactions.

1.4. Outline

This thesis is organized as follows. Chapter 2 gives the necessary background on model-driven engineering and domain-specific modeling languages. Then it presents existing work on multi-touch modeling editors. In Chapter 3, we propose a set of multi-touch gestures and we detail their mapping process with modeling activities. In Chapter 4, we show an implementation of the proposed approach. Then, we present three case studies that illustrate the use of our framework. Finally, we conclude and reflect on future work in Chapter 5.

Chapter 2

Background and state of the art

In this chapter, we review different notions of modeling languages and concept of modeling editors. We also discuss related works on interactive modeling editor and multi-touch interactions.

2.1. Model driven engineering

MDE [**42**] is an emerging and modern software development approach that express domain-specific modeling concepts by defining abstractions. The abstraction of a domain encapsulates with domain-specific languages (DSLs) which represent the domain structure, constraints and behaviour. The concept of applying levels of abstraction on a complex domain helps in reducing complexity to find the concrete solution. Likewise, MDE aims to focus more on model-based conceptual evaluation that gives a concrete solution to a problem domain using models rather than code or programming. The practice of MDE is described in [**55**]. Consequently, MDE includes various model-driven approaches to the domain models to generate automatic transformation such as code generation [**18, 22, 48, 49**], editor generation and model transformation [**18, 30**].

In [**4**], the authors described the MDE architecture. Figure 2.1 illustrates different tiers of abstraction. Namely, ***M0*** exposes mostly on a domain system without doing any abstraction. Here, we consider an e-commerce system where no domain objects are initiated.

M1 level describes the domain model. Following an example, we define models representing a catalog of an e-commerce system, where we discover different categories, subcategories, values, etc. While ***M2*** specifies metamodel of a model which is defined in ***M1*** level. A metamodel is the language structure; it defines part of the syntax of the language with typing relation among the values, possibilities, and associations allowed between concepts. We have represented a metamodel where a metamodel explains the concept of category, subcategory, they have attributes and connections in Figure 2.1. The model ***M1*** is an instance of that metamodel.

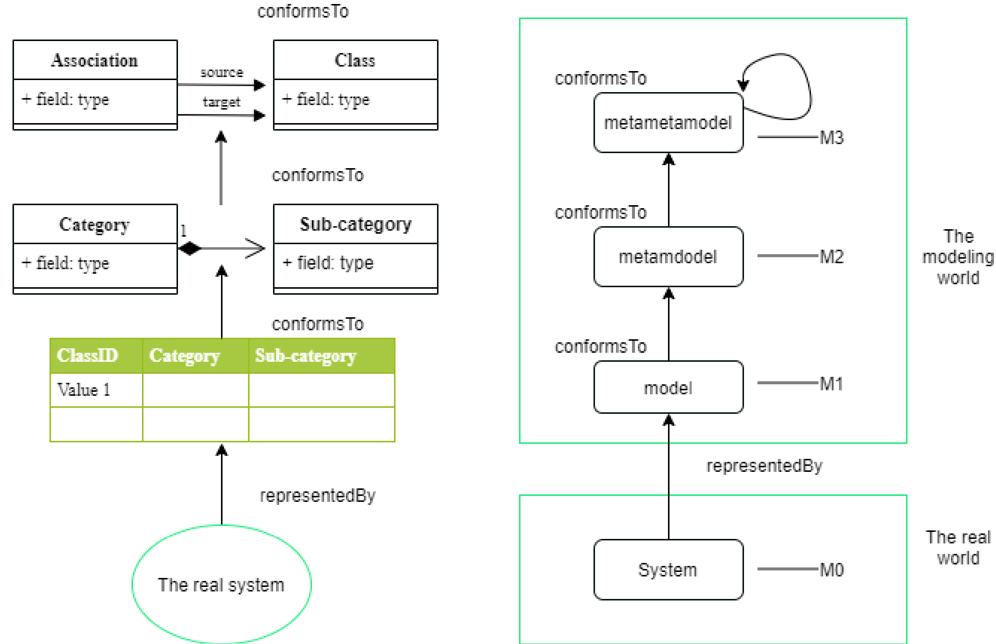


Fig. 2.1. MDE architecture and example [4]

The concept of ***M3*** mainly focuses on metametamodel. A metamodel is for one domain-specific language. However, when we have multiple domain-specific languages, they are additionally tided by a metametamodel. The metamodel of the metamodels is the metametamodel and is usually defined by the class diagram notations. Therefore, the metametamodel is the definition of a class diagram (e.g., MOF [**35**], Ecore [**50**]).

2.1.1. Domain-specific modeling

Domain-specific modeling (DSM) [**22**] is a model-driven software development approach that helps defining domain-specific models. To define domain-specific models, we need a domain-specific language (DSL). Why do we use domain-specific models instead of generic models? It is because they are tailored to a specific domain using notations from their domain, habits, concepts, and terminology from that domain. However, to define these concepts, we need domain experts. A domain expert allows to design domain-specific models and a domain-specific language is inevitable to sketch the models. A DSL is composed of abstract syntax, concrete syntax, semantic, and pragmatics. Moreover, it helps to migrate the ambiguity of a problem domain by defining the higher level of abstraction and formal semantics which assists to generate coding interface [**53**].

2.1.1.1. **Abstract syntax**. In [**36**], the authors considered the abstract syntax expresses models often in UML class diagram notations. An abstract syntax defines the structure of how the models are going to be represented in memory. There are two parts in the abstract syntax; it has a metamodel of the language and constraints such as static semantics imposed

on the metamodel. The static semantics is usually defined using constraints (e.g., expressed in OCL). Furthermore, constraints also may specify class name and attributes must be unique across all its instances.

2.1.1.2. **Concrete syntax**. A concrete syntax of a language is the representation of the abstract syntax of the language. The concrete syntax consists of a set of rule that defines how to represent each concept of the metamodels (e.g., abstract syntax) to the user. Typically, the concrete syntax of DSL considers graphical or textual notations described in [**17**]. In this thesis, we concentrate on graphical syntax only. What is the concrete syntax for graphical notations? We can assign geometric shapes and icons to each concept or class defined in the metamodel. We can also define splines or topological relations for the associations.

2.1.1.3. **Semantics**. The semantics of a language [**32**] explicitly utilizes the semantic grammar to map a DSL abstract syntax to a semantic domain. The semantics of a modeling language gives a meaning to its models, where all model elements have a unique meaning. In particular, DSL semantics is usually defined by mapping the abstract syntax to a semantic domain.

2.1.1.4. **Pragmatics**. Pragmatics aims to understand the language meaning of its design which is dependent on the context. Therefore, it assists to deals with the relations of signs with aspects of behind the text.

2.1.1.5. **Metamodel**. A metamodeling is the modeling of models [**47**]. Metamodeling defines the structure of the abstract syntax. The concept is derived from UML class diagrams notation where classes represent concepts, associations state relations between concepts and attributes are the properties of concepts. Static semantics is often described in the object constraint language (OCL) [**25**]. Static semantics form logical rules such as business rules; domain rules define how conditions work together—metamodels established from the domain models typically represent classes, attributes, and objectives. The Object Constraint Language (OCL) is an expression language that describes constraints on semantics. In the higher level of abstraction, metamodels shield to generate abstract syntax and static semantics. The concrete syntax represents the attribute and concepts graphically to the user that can be mapped into metamodels [**54**].

2.1.2. User interaction with modeling editors

Interactions are invariably predefined based on the language workbench framework [**10**]. A language workbench is a modeling framework (e.g., EMF [**6**], AToMPM [**51**]) that imposes a certain association among tools. Tools such as AToMPM, the Graphical Modeling Framework (GMF) [**13**], EuGENia [**24**], Visual Paradigm (VP)[1], or Sirius [**45**] mostly offer

[1]Visual Paradigm: `https://www.visual-paradigm.com/`

the graphical interface, such as drag-and-drop the components to build models using the mouse and keyboard. However, such tools default behavior only assists with a traditional input device (e.g., *mouse* and *keyboard*). As a result, we are restricted with a limited number of gesture actions. Whereas, a touch-based graphical framework may assist diverse gesture actions to perform tasks.

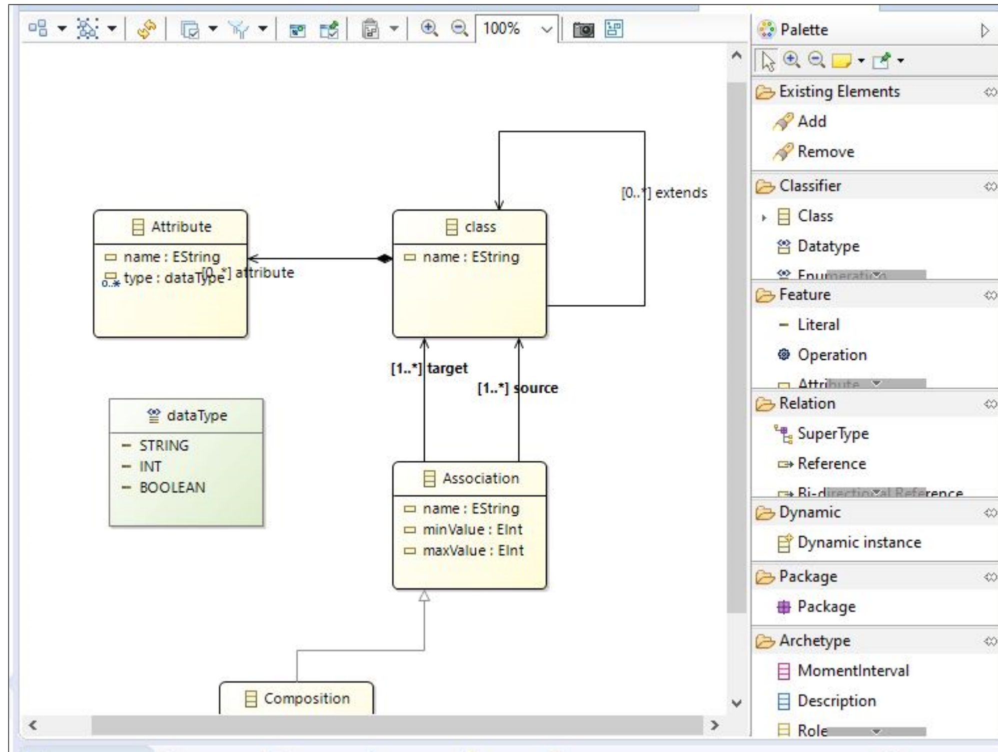


Fig. 2.2. Graphical framework examples

The authors of [**16**] addressed the reactive nature of the statechart to identify states, transitions, events, and variables that are easy to describe the modeling editors. Each state represents an object state at a particular given point in the modeling editor. Next, an arrow represents the transition from one state to another state of objects [**9**]. It also allows to activate an event based upon responsive user input (e.g., internal or external events). The internal event helps to terminate dialog windows in [**33**]. On the contrary, an external event performs user actions while interacting with a modeling environment. A flowchart [**29**] is simply a graphical representation of steps to process tasks in sequential order. It helps when additional interactions are important to adapt for various purposes and can be used to describe various states.

Above all the frameworks (see above) reuse a DSML to conduct the user interface and interactions to the modeling environment to create models. Once models are considered, code can be generated automatically based on its metamodel and concrete syntax. Even frameworks offer to manipulate code manually to define your own user interaction. All

editor frameworks often have the same graphical environment. Like, a canvas area and container which is engaged with different component models (Figure 2.2). Importantly, the component helps to create domain-specific models with the notations of the abstract syntax and graphical concrete syntax.

2.2. Modeling the interaction of DSM editors

A default DSM editor is generated according to the abstract syntax and concrete syntax of a DSL, where the higher level of user interactions are restricted. However, to get independent user interactions on a DSM editor, different interactions are confirmed with user experts that integrated into domain editors. In [**46**], the authors considered a new version of DSM editor that improved user interactions by combining different design model viewpoints of interface model, interaction models, and event mapping models. The goal is to enable to customize user interactions to the editor. To some extent, the generated DSM editors with custom interactions are more appropriate context-free, offering more interactive mockups than those generated in language workbenches.

2.2.1. Interface model

An interface model focuses on the design interface to improve the user experience and usability, representing how the end-users interact with interface properties and how the elements respond. For example, a user can define specific elements, their representations, and distributions. The goal of using interface models is to explain interaction objects, assignments, and lower-level dialogs. A user interface can help the better understanding of useful user elements, avoid premature commitment to specific layouts and widgets, and make the relationships between an interface's different parts and their roles explicit.

2.2.2. Interaction model

The UX designer or language engineer defines the interaction models. The interaction models the semantics of the interface elements, user interactions, interactive actions, and a set of interaction rules. The purpose of defining the interaction models is to tailor interactions for the end-users. For example, user experts can define functions, actions, behavior, and effects that are mapped with an event in the event mapping model.

2.2.3. Event mapping model

Event mapping models propose the relationship between events and processes. At the same time, an interaction rule relies on domain-specific events, called essential actions. An action is something done to accomplish a purpose, while an event is an occurrence. There are few events such as input events, operation events, and internal events to complete the

DSM editor's task. Input events express how user interactions occur in the editor (e.g., Tap a button). Operation events represent an operation performed to manipulate an interface element (e.g., the action indicating whether the element can be used as the target of a drag-and-drop operation). In contrast, internal events represent the system's reaction to an operation event for a specific function; either the process continues or terminates.

2.2.4. Process of generating interactive DSM editors

DSM editors are being developed according to the MDE automation techniques. The code generation technique applies the transformation steps in between an input and output models to produce GUI of the editor. The input models encode the code generation rules and process, and the output models show the generated code. The abstract syntax and concrete syntax of the DSLs constructed with interface models, interaction models, and event mapping models.

2.3. Programming by touch

The touch-enabled user interface (UI) conducts more natural or intuitive gestures events and actions than a typical traditional input device (e.g., mouse, keyboard). End-users can use their fingers to execute tasks mentioned in [**20**]. Additionally, it refers multi-touch interactions in the same interface. A multi-touch interface offers more dimensions of touches at a time rather than single touch as an input. All the mechanisms occur programmatically with the event-action framework (*e.g., touchstart, touchmove, touchend*) or a higher level of touch event abstraction in [**37**].

In [**37**], the authors developed a group touch and cross-events primitives that offer multiple touches input together (e.g., two fingers or more touch input). Whereas default language workbenches offer limited user interactions to restricted diverse event actions.

The authors of [**26**] developed gesture-based interactive tools to create user-defined gesture recognizers and rendering structure movement from an assigned script to investigate accurate recognizer. Developers define gesture examples, write the script for a specific gesture that can render the output on the canvas, and define gesture actions to map the accurate recognizer. Additionally, they can declare a predefined gesture recognizer to render unique abstraction. For example, to draw geometric shapes (*e.g., rectangle, triangle, sector, arc. polygon, etc.*) on a canvas.

Lü et al. [**27**] proposed a tool for generating the gesture code (e.g., function) using a multi-touch gesture. They set up the default recognizers for all possible gestures. Therefore, the developer performs a gesture instance on the canvas, it automatically generates code against each gesture instance. Furthermore, the generated code can be explicitly modified

or invoked with the corresponding needs, and gesture code can be refined by adding more examples.

A multi-touch framework called Proton was proposed in [**23**] to resolve gesture conflicts. If any conflict has been detected, Proton can automatically resolve errors and creates an unique gesture sets for the whole system.

In [**56**], the authors proposed a prototype that allows sketching a formal or informal software engineering model, refining the model, assigning syntax, and semantic. However, the reseach contribution only considers single interaction at a time (*drag and drop only*).

In [**43**], the authors developed a tool designed to simplify the development of rules-based approach. The tool helps to differentiate multi-touch actions and gesture behavior, which improves gesture performance. The approach is executed from a rule-based logical language, logical rules defined explicitly for each gesture's gesture of how gesture will behave or trigger the task.

Multi-touch gestures can be very difficult to program correctly because they require that developers build high-level abstractions from low-level touch events. Even a coding-based initial set of gestures is not working independently [**39**]. Therefore, a multi-touch modeling editor must be deployed where domain experts or stakeholders control gesture features by their own needs. We focus on initial sets of gesture-based interactions that enable us to work with domain-specific multi-touch modeling editors.

2.4. Interactive touch libraries

Touch libraries have been performing an interactive and collaborative activity with gesture tools to move toward a more natural user interface. In 1995, IBM developed graphical user interfaces (GUIs) and mice that support interactive work toward more instinctive ways to interface with the machines and humans [**2**]. Since then, a new way of interaction upgrading to complete interactive work. There are many touch libraries that support the technological perspective —such as HammerJs [**28**], EventJs [**34**], InteractJs [**40**], ZingTouch.js [**7**].

Table 2.1 illustrates a different gesture operations and dependencies of each touch library. Distinctive touch library performs a distinct gesture instance. For example, interact.js [**19**] offers pan and press gesture instance to drag and drop elements on canvas. So a user can move an element on the canvas. Same as ZingTouch [**57**] and EventJS [**34**] render only six conventional gestures.

We sought to create our own gesture instance to anticipate a user-defined interactive gesture event. Hence, a gesture can be supported on any embedded system independently, either on a web-based platform or desktop-based application. HammerJs [**15**], by default, alone has all basic interactive gestures like tap, double-tap, press, horizontal pan, swipe,

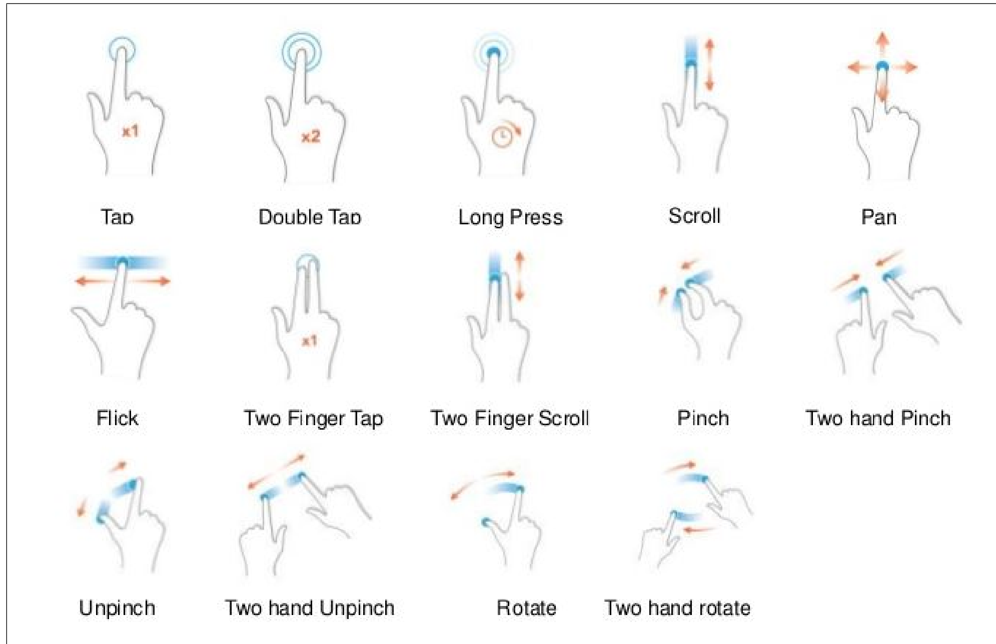


Fig. 2.3. Multi-touch gesture behavior [**52**]

Table 2.1. Characteristic of gesture libraries

Interactive touch library and gestures				
List of gesture	HammerJs [**15**]	EventJs [**34**]	InteractJs [**40**]	ZingTouch.js [**7**]
Dependencies	No	Yes	Yes	Yes
Pan	Yes	Yes	Yes	Yes
Pinch	Yes	No	No	No
Press	Yes	Yes	No	Yes
Swipe	Yes	Yes	No	No
Rotate	Yes	No	Yes	No
Multiple-Tap	Yes	No	No	No
LongPress	Yes	No	No	No
MultiPointer	Yes	No	No	No

and solely with multi-touch pointers to pinch and rotate recognizers (Figure 2.3[2]). It has no dependencies while integrating with an embedded system. HammerJs offers custom code and reusable gesture recognizer. The language engineer defines gesture recognizer with the combination of multiple gesture events.

```
1 var myElement = document.getElementById('myElement');
2 // create a simple instance
3 // by default, it only adds horizontal recognizers
4 var mc = new Hammer(myElement);
5 // let the press gesture support all directions.
```

[2]Touch gestures: `https://hammerjs.github.io/examples/`

```
6 // this will block the vertical scrolling on a touch-device while on the element
7 mc.get('press').set({ direction: Hammer.DIRECTION_ALL });
8 // listen to events...
9 mc.on("tap press", function(ev) {myElement.textContent = ev.type +" gesture detected.";
10 });
```

Listing 2.1. Hammer Pan gesture example

Listing 2.1 shows the concrete example of how the default gesture (e.g., *p*an) create its instance. It allows the interpretation without changing anything. However, all possible interactions will be returned with a single gesture. Therefore, the user can drive the element on the canvas individually.

```
1 // Get a reference to an element.
2 var square = document.querySelector('.square');
3 // Create a manager to manage the element.
4 var manager = new Hammer.Manager(square);
5 // Create a recognizer.
6 var TripleTap = new Hammer.Tap({
7   event: 'tripletap',
8   taps: 3
9 });
10 // Add the recognizer to the manager.
11 manager.add(TripleTap);
12 // Subscribe to the event.
13 manager.on('tripletap', function(e) {
14   e.target.classList.toggle('expand');
15   console.log("You're triple tapping me!");
16   console.log(e);
17 });
```

Listing 2.2. Hammer manager example

Hammer manager yields extended control on gesture recognition so that the end-user can run various recognizers simultaneously in Listing 2.2. If a user wants to recognize his own gestures, such as *tripletap*, then a *manager* container is important to get such a recognizer. The *manager* is considered as a container that accumulates all the recognizer instances. It sets up the input event listeners and sets the touch-action property to trigger such events.

```
1 var mc = new Hammer.Manager(myElement, myOptions);
2 mc.add( new Hammer.Pan({ direction: Hammer.DIRECTION_ALL, threshold: 0 }) );
3 mc.get('pan').set({ event: 'pan', pointers: 4 });
4 mc.on("pan", function(ev) {myElement.textContent = ev.type +" gesture detected.";
5 });
```

Listing 2.3. Hammer multi-pointers example

Listing 2.2 and Listing 2.3 represent the same algorithm to configure unique events precisely to the multiple pointers. A user can specify multiple pointers at a time while targeting a new event to trigger a function. For instance, if a user wants to create a gesture recognizer by pointing his finger on the canvas, it should be possible.

Chapter 3

Multi-touch domain-specific modeling editor

In this chapter, we will present the proposed gestures that are useful for domain-specific modeling. We also discuss how gestures are integrated into DSM editors.

A touch gesture is used to manipulate content or element versatility under the finger or perform user satisfied touch actions to conduct an assignment. For example, a touch action could be touch and hold on the screen, release gesture until the actions or operations occurs. Some common touch gestures for scrolling, pinching, and tapping to operate an interface; however, these gestures could produce distinct functions in the DSM editor by assigning the domain-specific tasks.

3.1. Multi-touch gestures for domain-specific modeling

Modeling language workbenches produce domain-specific modeling editors followed by the abstract and concrete syntax languages where their default operation interacts with the interface. Therefore, an editor's user interaction is restricted upon their generic interactions and cannot recognize the modified gesture events. In contrast, a touch supportive graphical editor or application software to process additional user interactions by integrating touch library. To extend user interactions and unique user interfaces on a modeling editor, different interaction viewpoints are required to offer more dimensions of interactions to complete the task without the redundancy and confliction effects. Ample UX interactions urge to set up new gesture events to revolve tasks in various ways such as model integration, tool integration, and integration between process and tool support [**3**]. For example, multi-touch user interactions actively promise to modify and create a task at any dimension on the interface (e.g., canvas) and the link between the diagrams or elements [**12**]. Particularly, designing a model or sketch a diagram in the graphical editor is very useful for a multi-touch gesture rather than considering click input. In [**14, 8**], the authors explained the disadvantages of using structural editors toward editing or modeling diagrams with formal

Table 3.1. Generic gesture events and their actions for graphical editor

Catalog of useful gestures for a graphical editor	
Gesture event	**Action**
Tap	Tap the screen to perform an action. Some gestures may require you to tap the screen with 1, 2 or even 3 fingers multiple times.
Pinch	Pressing two fingers apart on the screen, move them towards each other as if pinching them together.
Press	Press a finger on the screen or a pen as an input.
Press and Hold	Press a finger on the screen and continue to hold the finger against the screen.
Swipe	A swipe gesture occurs when the user moves one or more fingers across the screen in a specific horizontal or vertical direction.
Rotate	Pressing two fingers together on the screen, move them away from each other as if stretching them apart.
Long-press	Long-press gestures detect one or more fingers (or a stylus) touching the screen for an extended period of time.
Pan	A longer, slower movement of a fingertip across the screen, usually in a vertical or horizontal direction away from an edge of the screen.
Long-press and Drag	the operation helps to create complex task by the combination of multiple gesture and input pointers

notations that are finite and fixed. Therefore, multi-touch gesture events are particularly notable for leveraging challenging tasks.

3.1.1. List of gesture events and actions

Table 3.1 shows the possible and useful gesture events and actions; these are mostly used to draw diagrams on the graphical editor to draw shapes, icons, widgets, and customized dimensions on the editor's canvas or interface. The author in [**39**] considered the valuable gesture to sketches based on the touch-sensitive surface to perform the task. A sketch is completed by fingers as an input to draw a single-stroke or multiple-stroke diagram. In particular, they used touch gestures to achieved the task. A study in [**12**] developed all basic gesture events to edit, create the diagram on a tabletop or touch screen to initiate event operations to manipulate tasks such as move elements, add nodes, and connect edges between nodes. However, most of the tasks are defined by the basic operations of the events.

The collaboration between two or multiple gestures to trigger a single operation is possible in the touch interface, combining the gesture like long-press and touch drag into a single

operation. For example, if the user wants to remove or erase a line or multiple objects simultaneously, complex and combined gesture event compilation is important. It could be executed by one or multiple user pointers.

In [**38**], the authors proposed an extended gestUI gestures that can allow individual users to define their own custom gesture catalog and redefine some custom gestures in case of difficulty using. The gesture catalog (Table 3.1) shows the different meanings; however, we can customize user-definition gestures. The specific domain does not define the semantics of the gesture. The catalog shows the spectrum of the possible gestures that a user needs to define the semantics for each gesture. Gesture meaning could be domain-specific.

3.1.2. Assigning semantics to gestures

On the one hand, we have the usual operations when designing DSMs:

- **Instantiate**: Instantiation means creating a new class instance.
- **Connect**: A connection interface uses to connect two or multiple instances by a link.
- **View properties**: A view is a container that supports layout with flexbox, style of each instance.
- **Edit properties**: Customize the property value of each instance and their accessibility controls.
- **Remove the object**: A remove operation usually performs to delete node or object from the canvas (e.g., remove the link between two interfaces).
- **Move**: A move operation means to move an object or element on the interface flexibly.
- **Insert** (containment relation): Containment relationship between a parent and child is unambiguously represented in a graphical view of the object (e.g., reconnect the link between objects).
- **Scale**: A scale operation adds a scaling transformation to the canvas to extend object size horizontally and/or vertically (e.g., a scaling factor of 5px changes the object size to 10px).
- **Rotate**: It means to rotate an element horizontally and/or vertically, invoke special functions (e.g., validate, refactor, transform, etc.).

However, end-users can redefine his/her own custom gesture catalog to obtain one gesture catalog per user. User experts can encode each gesture as an event. Additionally, they can redefine a characterized gesture when conflict has arisen; redefinition can be done during the execution stage or before a new iteration has been made on the software system's mapping model.

To assign gesture events to DSM operations, an event mapping model defines how gesture events have been recognized by actions and map the operations to achieve the task [**46**]. In

Table 3.2. An example of assigning gesture events and operations

Catalog of gestures and regarding operations		
Gesture Events	**Operations**	**Semantics**
Tap	add	Add object, instantiate object, mark element on the canvas.
Swipe	add	Visible and hide properties window.
Long-press and Drag	add	Create new object instances.
Double-tap	delete	Delete the target object.
Pinch	zoom	Scale the object, resize the canvas or interface.
Press	mark	Mark an element or select an object.
Pan	link	Connect a link between object, reconnect or re-link objects.
Rotate	rotation	Rotate an object in any dimension on the canvas.

particular, all gesture action requires a mapping individually to target an operation. In the specification, I (input) or O (output) events define mapping type on the DSM editor. Input mapping represent the gesture event in the mapping model; alternatively, an operation is used for Output mappings.

A mapping between user actions and system events is provided, mapping pre-made action names to different system events such as press, tap, and other gesture events. Listing 3.1 and Listing 3.2 shows the events and operations mapping.

```
1   // setup an event mapping by "I" tag
2   I # tap # tap a finger on the touch surface or canvas
```

Listing 3.1. Event mapping example

```
1   // setup an operation by "O" tag
2   O # add # adds a object, instantiate a icon, mark an element
```

Listing 3.2. Operation mapping example

Table 3.2 shows exemplary gesture events and their respective operations to complete or achieve a task. For instance, to use one finger to touch the screen in one spot quickly. Tap to select objects on the canvas, open or close menus and panels, select tools from the Toolbar, apply commands and effects. However, end-users can define their own gesture event and operation depending on their comfortable usage of unique events. Each DSL can have its own mapping. Nevertheless, give guidelines that this gesture is typically better suited for this set of operations.

Usually, we can define one gesture event to fire a single operation (Table 3.2). To define a single gesture event to trigger multiple operations to complete a task is quilt unnatural and error-prone. In contrast, operating multiple events to map a single operation is quite an easy process. For example, the end-user can use one finger to tap the screen in one spot and keep it pressed. This gesture can be used to open the properties window when working on the canvas. Even the same work has been done by swipe-right event.

3.1.3. Resolve conflicting mapping event

Gesture conflict mostly occurs on the graphical interface when it performs similar operations behavior. We found gesture event confliction on some actions such as *swipe* and *pan*. For example, we injected a *pan* gesture for moving objects on the canvas and defined a *swipe* gesture for the invisible properties window. However, when we move the element randomly on the touch interface, it automatically triggers the *swipe* event since we could not specify directions. Therefore, it's important to define the directions (e.g., *get('swipe').set(direction: Hammer.swipe-left or swipe-right)*) that resolved the redundancy.

A gesture event can trigger one operation only at a time. For instance, if an event mapped to triggers to multiple operations, it creates the enormous output.

3.2. Domain-specific interactive modeling

The proposed example is cited from Lucidchart[1] and Flowchart maker[2] as shown in Figure 3.1, the domain expert or modeling engineer could design their models through the multi-touch enabled interface. The collaboration of using multi-touch supportive editors could support the highly interactive tasks. As shown in Figure 3.1(a), a domain expert could design models in the editor canvas then he can initiate a class in the interface by using a finger gesture (Figure 3.1(b)). Gradually, add class attributes, create links between classes as seen in Figure 3.1(c) and 3.1(d). The objects of using touch interface to create the respective elements of the diagram, to add different attributes more efficiently. Moreover, domain experts can get more dimensions of interaction rather than using traditional interactions (mouse and keyboard).

To sum up, we discussed in general multi-touch enabled gestures catalog and their usage in graphical diagramming context. The conflicts between events and their actions. Finally, we explained an example of how domain experts or modeling engineers could model collaboratively in modeling editors. In the next chapter, we will take more about the implementations.

[1]Virtual graphical editor: `https://www.lucidchart.com/`
[2]Virtual graphical editor: `https://app.diagrams.net/`

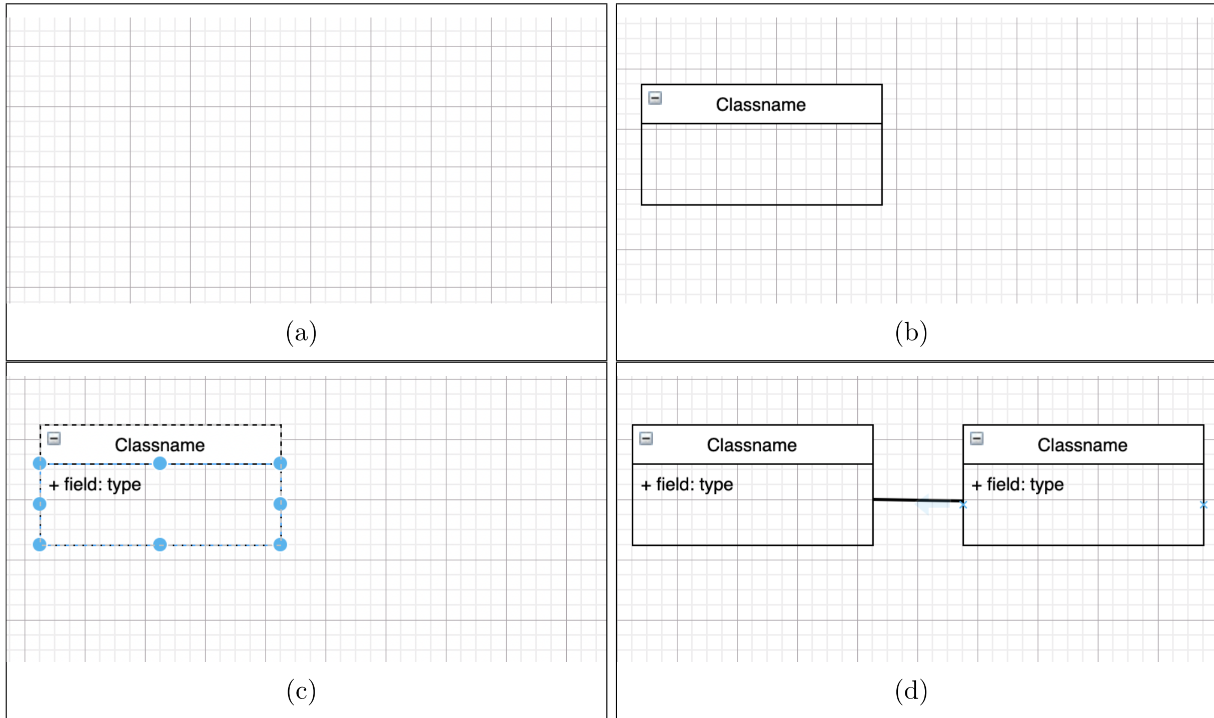


Fig. 3.1. An example of defining UML diagram by touch gestures

Chapter 4

Implementation and examples

In this chapter, we detail the implementation and apply our approach on three DSM editors. Our approach examines these DSM editors that rely on external interactive library support. We also discuss the limitations of our approach.

4.1. Defining multi-touch modeling editors

In our approach, there are three actors involved to design the prototype. At first, the creator defines the specification of the external library to reform all the possible gesture actions and operations to the Library that can extend from HammerJs. Secondly, the language engineer creates DSLs and generates editors. Finally, a domain expert uses the editor.

The creator could be the Language engineer, or just a software engineer. At first, he investigates the external library (e.g., HammerJs) to expand relevant gesture operations to make a library called `Gesture catalog`. The gesture catalog deals all the relevant user-defined gesture actions and operations. The custom gesture catalog offers additional touch-enabled gesture events to the editor.

As shown in Figure 4.1, the language engineer creates DSLs and generates editors. To do so, he defines the three models of the editor (interface, interaction, and event mapping models) on top of the metamodel and concrete syntax. Typically, he starts by designing the interface model to create GUI elements and their layout. Then, he defines the interaction model to specify the interaction rules. These rules reference elements of the interface model and of the metamodel. Each rule is triggered by an event. The event mapping model defines each event with the gesture in the Gesture catalog stage. The event mapping model can also define operations that can be invoked from an interaction rule to run a procedure in the backend.

From these models, we automatically synthesize a modeling editor tailored to the DSL and augmented with multi-touch interactions. A domain expert can then interact with the editor with the touch of his fingers to create and manipulate domain-specific models.

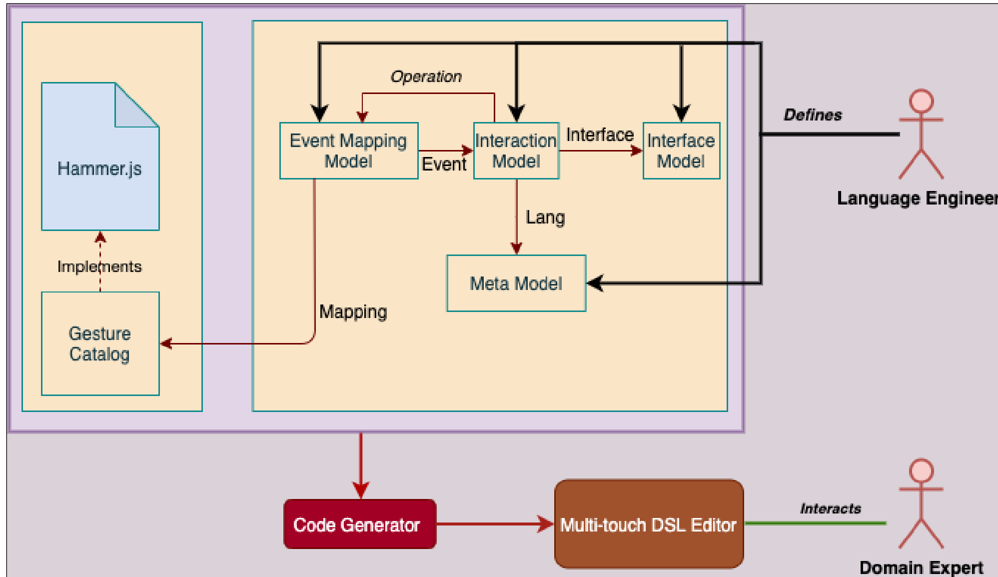


Fig. 4.1. Architecture of our approach

4.1.1. Implementation of multi-touch events

We added a gesture catalog (referred as Gesture Catalog in Figure 4.1) to the proposed work to explore how touch gesture works in a graphical modeling interface. We implemented a gesture catalog that should fit with most DSLs. Table 4.1 reports all the possible gesture events we implemented along with a brief outline of operations that are useful for a touch-enabled editor. For instance, if the language engineer designs interaction rules to move an element on the canvas, our catalog shows that pan gestures could be used.

The gesture catalog is a wrapper around HammerJS to facilitate the definition of gesture events. The catalog offers custom events provided by HammerJS. A HammerJS event can be customized by the number of pointers required for the interaction (e.g., pan with two or more fingers). It can also be customized by the number of taps to recognize for this event (e.g., double-tap).

Listing 4.1 depicts how to add a user-defined gesture event to the gesture catalog. As seen on line 3 and line 4, *pointers* keyword represents how many pointers should consider to interact with, *taps* keywords refer to inject number of tap recognizer. These all user define action and events configures to the Hammer manager containers as seen on line 2. Once all above gesture recognize has mapped, a *get* method applied to access these gesture events as seen on line 5. Therefore, a user can choose the best gesture events and actions from their implemented gesture catalog.

```
1   var myElement = document.getElementById('<ElementName>');
2   var canvas = new Hammer.Manager('myElement');
3       Canvas.add( new Hammer.<EventName>({ event: '<EventName>', pointers: <Required pointers>,
            taps: <Required taps> }) );
```

```
4         canvas.add( new Hammer.Tap({ event: '<EventName>' }) );
5         canvas.get('<EventName>').recognizeWith('<EventName>').requireFailure('<EventName>');
```

Listing 4.1. Domain-specific gesture rules implementation format on Library

Table 4.1. Implemented gesture list

Gesture Name	Useful Operations
Press	Press an element on the canvas to mark or select, useful to active a button.
LongPress	LongPress to create an element, element to delete, mark or select, and popup score button.
ShortPress	ShortPress an element to delete, mark or select and setup time frame.
Tap	Tap to popup score button, create element, delete element, mark or select object.
DoubleTap	DoubleTap to create or delete an object.
TripleTap or more	Multipletap to delete object.
2 pointers Tap or more	Create an instance or delete object
SwipeLeft	Useful to open or remove a window.
SwipeRight	Useful to visible or invisible a window.
SwipeUp	Useful to visible or invisible a window.
SwipeDown	Useful to visible or invisible a window.
Swipe with 2 or more pointers	Useful to visible or invisible a window.
PanStart	Mark or select source element to create link.
PandEnd	Mark or select target element to create link.
PanLeft	Move element only in left direction.
PanRight	Move object only in right direction.
PanUp	Move object only in upside direction.
PanDown	Move object only in downside direction.
Pan with 2 or more pointers	Move element at any directions on the interface.
Rotate with 2 or more pointers	Rotate object or canvas clockwise or counterclockwise.
Pinch with 2 or more pointers	Zoom object or canvas.
Long-press and Pan	Best choose to link two or multiple objects.

4.1.2. Interaction rules

Listing 4.2 shows the format of interaction rules in the interaction model as defined in [**46**]. A rule has a condition that describes the required configuration of the editor for the interaction to occur. The condition can comprise elements from the interface model (Interface), from the metamodel (Lang), or the canvas. Variables (Var) can be used to add

further control to the rules. When a variable is used in the condition, its value must equal the value specified to trigger the rule. The focus keyword indicates which element should be pointed at, acting as the context of the rule. For example, if the focus is on a Lang element, then it is on this element that the touch gesture (like, tap or press) must occur to mark or select it. It is possible to negate parts of the condition using the not keyword.

The event name is a domain-specific action which triggers the rule. Thus, to trigger a rule the condition must be satisfied, and the event must be received. Event names are usually agnostic of the platform used. For instance, we can have an event move to signify that a rule moves the element (in focus) on the canvas.

The effect of a rule has a similar syntax to the condition. When a rule is triggered the effect is applied to all the elements it comprises, thus updating their values. Lang elements can be specifically added to, removed from, or moved on the canvas. Var elements can be assigned a value. Additionally, Operation elements can be added to the effect of a rule. They are surrogates to invoking a procedure defined in the event mapping model.

```
1  InteractionRule <ruleName>
2     Condition {
3      ( (focus|not)
4          ( Canvas {},
5             Interface <elementName> {
6             (value = "<valueString>")?
7              },
8              Lang <elementType> {
9              (value = ("<valueString>"|<varName>))?
10             },
11             Var <varName> {
12             (value = ("<valueString>"|<varName>))?
13             } )
14        )*
15      }
16  ( --- <eventName> -->
17  Effect {
18      ( Interface <elementName> {
19          (value = "<valueString>")?
20          },
21          Lang <elementType> {
22          op = (add|rem|move) (, value = [<varName>(,< varName>)*])?
23          },
24          Var <varName> {
25          (value = ("<valueString>" | <varName>) )?
26          },
27      Operation <operationName> { }
28      )+
```

```
29 } )*
```

Listing 4.2. Interaction rule format

4.1.3. Event mapping

The mapping model associates the interaction rules on how and when an event can trigger the specific rules. The format of a mapping model (Listing 4.3) is described in three different sections. At first, the *MappingType* attribute can either be identified as I or O for mapping Input and Output events respectively. Second, *Event* attribute represents the event name as used in the Interaction rules. Alternatively, *Operation* attribute can be used for Output mappings to call available functions. Third, *Expression* attribute is to add a platform-specific code which the event is translated to (ref. Listing 4.4).

```
1    <MappingType> # <Event|Operation> # <Expression>
```

Listing 4.3. Event mapping models format

The events triggering an interaction rule represent actions specific to the domain, such as moving an element or adding a language element to the model displayed. They are called essential actions. The event mapping model allows one to correspond these actions to platform-specific events (called system events), such as pan or tap. The correspondence between essential actions and system events available on a specific platform allows us to reuse the interaction model on different platforms.

This mapping can be one-to-many, meaning that the editor offers multiple ways of achieving the same action. For example, creating an element on the canvas can be done by tap or press (the former is triggered before releasing, while the latter is triggered after releasing). Furthermore, we can associate the same system event to multiple essential actions. This enables the interactions to be context sensitive. Section 3.1.3 discusses how conflicts are resolved.

We have extended the event mapping model from [**46**] to take into account touch events from HammerJS. Listing 4.4 shows the format of the mapping model. The *<Element Name>* tag can hold the canvas "id" or interface element "id", the *<System Event>* tag holds a gesture event from Library, the *<Rule Event>* tag represents the essential action, and *<Project Name>* is the name of the current project to commit it with the generated editor.

```
1 #<EventName>Hammer(document.getElementById(<Element Name>)).on("<System Event Name>", function()
      {<Project Name>.Interaction.prototype.triggerEvent( event, <Element Name>,<Rule Event Name>
     );}, false);}
```

Listing 4.4. Expression format of a mapping models

Listing 4.5 shows the user specific event mapping format that should connect with the implemented domain-specific gesture rules on the Library. The event mapping is almost similar in Listing 4.4. The only difference, we added an instance variable (canvas) rather than default instance Hammer(*document.getElementById(myElement)*) (ref. Listing 4.1).

```
1 #<EventName>canvas.on("<System Event Name>", function() {<Project Name>.Interaction.prototype.
      triggerEvent( event, <Element Name>,<Rule Event Name>);}, false);}
```

Listing 4.5. Expression format of a advanced mapping models

4.2. Application examples

In previous work [**46**], Sousa et al. demonstrated the feasibility of their approach on three DSL editors: Game of Life simulator, Pac-Man game configurator, and Mind Map editor. The generated editors run on a chrome browser supporting HTML and JavaScript. They generate the respective modeling editors with customized interactions. However, these interactions were introduced with generic keystrokes or mouse movements and clicks. Therefore, we extended the same DSLs and their modeling editors with a touch interactive domain-specific editor. We experimented with the editors deployed on a mobile device and a multi-touch table.

We have been using the Eclipse Integrated Development Environment (IDE), version 2019-09 R (4.13.0)[1], to write and edit code. We integrated and implemented an API to the project called HammerJs[2]. A chrome browser (any version) is being used to examine the experiment[3]. The experiment could run in any version of Android phone[4]. We deployed our editor in a multi-touch table which is running with Windows 10. The custom table (multi-touch table) specifications are about the Screen: 55" BDL5530QI, Glass: Polycarbonate with touch foil, Computer: HP - Z230 Small Form Factor WS I5 / 3.3Ghz / 8GB / 1TB / W7P-W8, and Graphics card: PNY - NVIDIA Quadro K420 - PCI Express 2.0 x16 - 1GB GDDR3.

We choose these three DSLs to experiment with different aspects of touch interactive modeling editors. Game of life is the simplest editor constraining the user to a minimal set of interactions to instantiate cells on the canvas and run a simulation. Pac-Man is a complete DSL focusing on the creation of containment relations between objects and editing their attributes. Mind Map explores the creation of links and implicitly creating object.

[1]Eclipse editor - v2019-09 R: `https://www.eclipse.org/downloads/packages/release/2020-09/r`
[2]Hammer.JS - v2.0.8: `http://hammerjs.github.io/`
[3]Chrome Browser: `https://www.google.ca/chrome/`
[4]Android: `https://www.android.com/`

4.2.1. Game of Life simulator

We first report on the simplest modeling editor. In the following, we briefly describe the input models required to generate the editor.

4.2.1.1. **Metamodel**. Game of Life (GoL) consists of a simple DSL with one element in the metamodel. The metamodel consists of a single class *Cell* representing a grid cell. It can be either alive or dead with a *live* Boolean attribute set by default to true. A black square grid represents an active (alive) cell, and a deactivated (dead) cell is portrayed by a white square. A cell can have up to eight adjacent neighbors on the grid. The cells are subject to a series of rules that change their life status. If a cell is alive and has more than three neighbors, or fewer than two, it dies. If a dead cell has exactly three neighbors, it comes back to life. The editor we create is to define the initial configuration of live and dead cells and to launch a pre-built simulator.

4.2.1.2. **Interface model**. The GoL interface model is depicted in Figure 4.2. It has one interaction stream since we interact with the screen only. There are two layers of interaction. One holds the default layer for the canvas (the canvas layer always exists in this framework). The second stands for a toolbar with a play button. The play button will be used to launch the simulation.

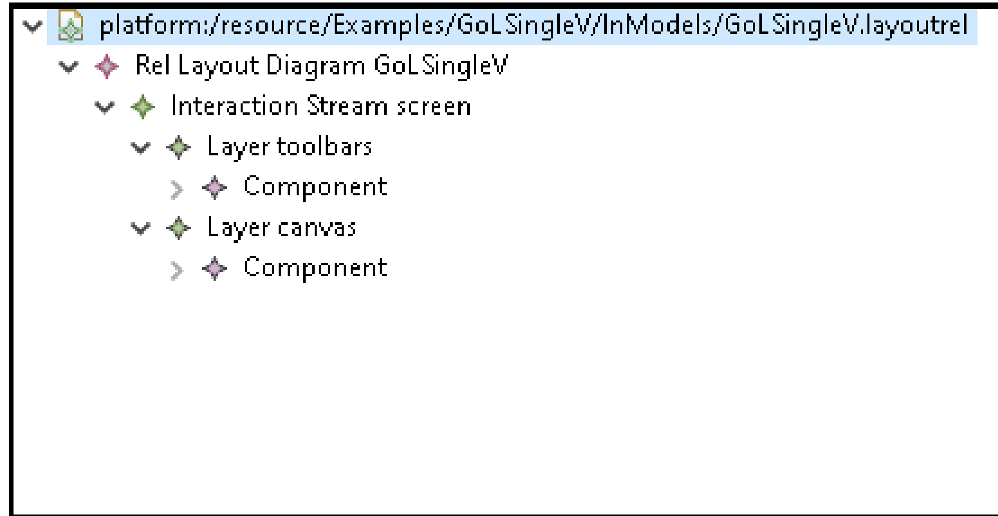


Fig. 4.2. GoL interface model

4.2.1.3. **Interaction model**. Listing 4.6 shows the interaction rule to create a cell on the canvas. Initially, a default live cell can be created on the canvas. This rule is triggered when the add action occurs on the canvas. Listing 4.7 depicts the rule to remove a cell when the dead action occurs on a cell. Finally, the simulation rules in Listing 4.8 run the simulation when the play button is pressed by invoking the pre-defined runGoL procedure available

in the event mapping model. This disables the button until the internal event _finish is received, announcing that the operation has terminated.

```
1 InteractionRule AddCell
2    Condition {
3       focus Canvas {}
4    }
5    --- add -->
6    Effect {
7       Lang Cell {op = add}
8    }
```

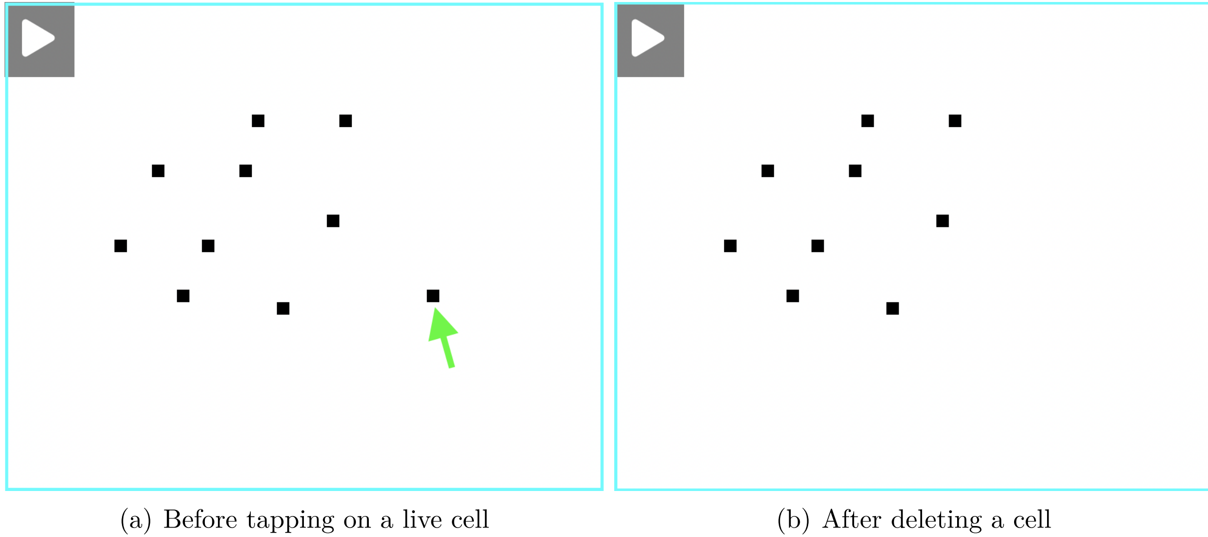
Listing 4.6. Interaction rule to add a cell

```
1 InteractionRule DeleteCell
2    Condition {
3       focus Lang Cell{}
4    }
5    --- dead -->
6    Effect {
7       Lang Cell {op = rem}
8    }
```

Listing 4.7. Interaction rule to remove a cell

```
1 InteractionRule RunTransfromation
2   Condition {
3     focus Interface playModelButton{}
4   }
5   --- press -->
6   Effect {
7     Interface playModelButton {value = "active"}
8     Operation runGoL {}
9   }
10
11 InteractionRule EndTransformation
12   Condition {
13     Interface playModelButton {value = "active"}
14   }
15   --- _finish -->
16   Effect {
17     Interface playModelButton {value = "default"}
18   }
```

Listing 4.8. Interaction rule to run simulation



(a) Before tapping on a live cell (b) After deleting a cell

Fig. 4.3. GoL creation a deletion of cells

4.2.1.4. **Event mapping model.** In the event mapping model, we assign the essential actions to specific touch gestures from the catalog. Listing 4.9 shows the example of mapping the tap event to create a cell on the canvas on line 1 (see Figure 4.3). On line 2, if we tap on the cell, it will be removed. We mapped same event (e.g., tap) to execute both operations. On line 3, tapping on the play button will run the simulation (see Figure 4.4).

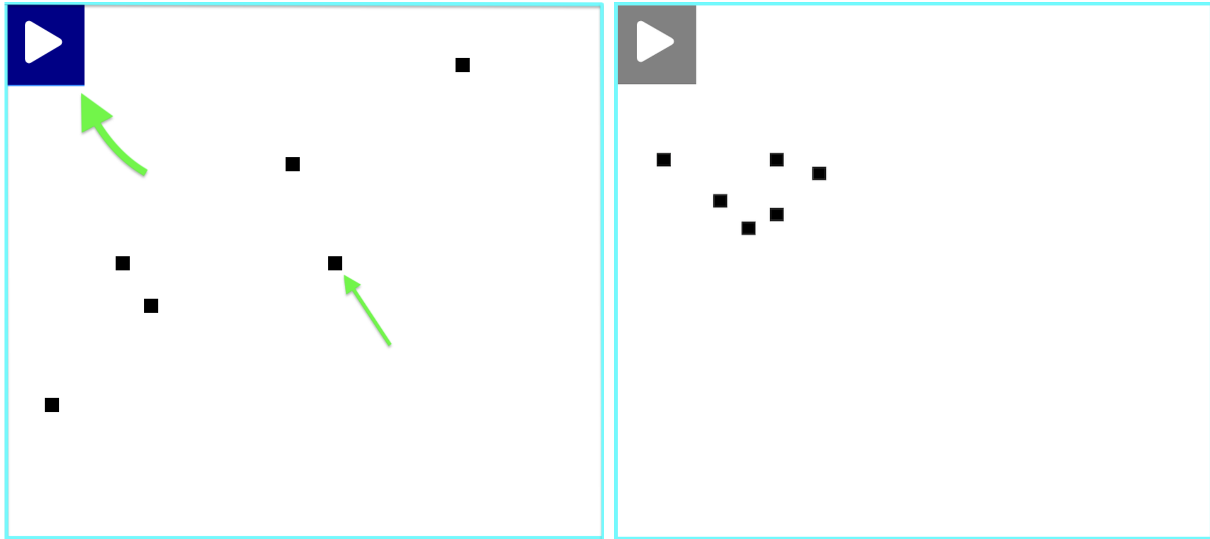
```
1 I#add#Hammer(document.getElementById("23")).on("tap",function(){GoL06.Interaction.prototype.
      triggerEvent(event,event.target.id,'add');}, false);
2 I#dead#Hammer(document.getElementById("23")).on("tap",function(){GoL06.Interaction.prototype.
      triggerEvent(event,event.target.id,'dead');}, false);
3 I#press#Hammer(document.getElementById("playModelButton")).on("tap", function() {GoLSingle.
      Interaction.prototype.triggerEvent(event,"playModelButton",'press');}, false);
```

Listing 4.9. Mapping the add event to a tap touch interaction

4.2.2. Pac-Man Game configurator

We report on a more comprehensive modeling editor, but mainly focusing on the instantiation of different types of language elements. In the following, we briefly describe the input models required to generate the editor.

4.2.2.1. **Metamodel.** In the Pac-Man Game, Pac-Man navigates through grid nodes searching for food to eat, while ghosts try to kill him [**41**]. To play this game, a player controls Pac-Man, who must eat all the food inside an enclosed maze while avoiding ghosts. In [**46**], the authors developed a simple DSL for Pac-Man game configurations, from which we generate the web-based editor. The editor consists of a canvas where grid nodes, Pac-Man, ghosts, and food can be created, removed, and moved around (see Figure 4.5). However,



(a) An example of running simulation (b) Button is disabled while running a simulation

Fig. 4.4. Interaction with the play button of GoL

instead of creating a Pac-Man model with a mouse and keyboard, we built a custom touch to enable events and actions to interact more naturally with the editor, tabletop, or touch screen. Note that, in the metamodel, *PositionableEntities* (i.e., Pac-Man, food or ghost) must always be placed on grid nodes, even at instantiation-time.

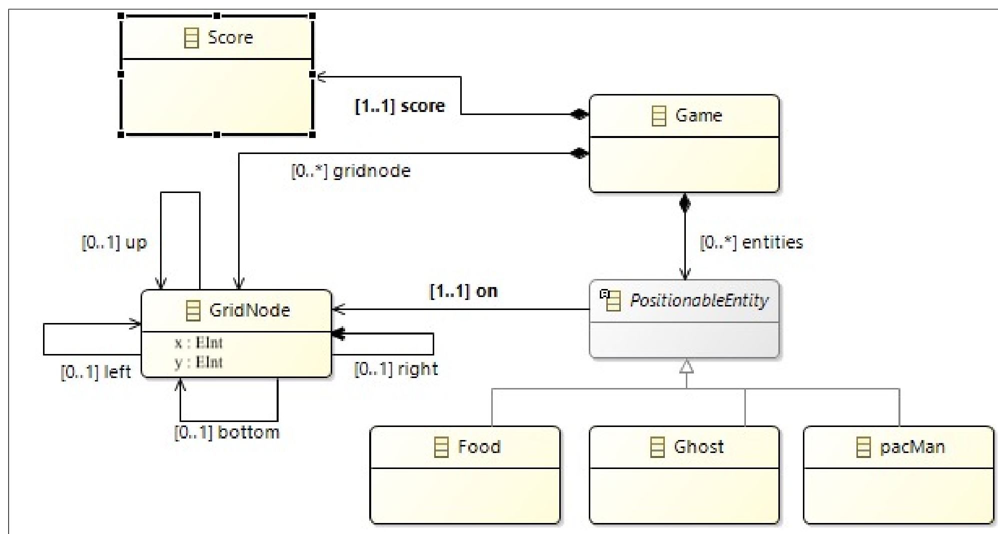


Fig. 4.5. The metamodel of the conceptual aspect of Pac-Man

4.2.2.2. **Interface model**. The interface model of a Pac-Man game editor shows different levels of the hierarchical structure of its element. As depicted in Figure 4.6, there are three different layers. Like in GoL, the first one is always the canvas and the second is for the

toolbar to create a score element. A third layer is dedicated to showing the properties (attribute values) of the language elements, such as the *x,y* coordinates of a grid node.

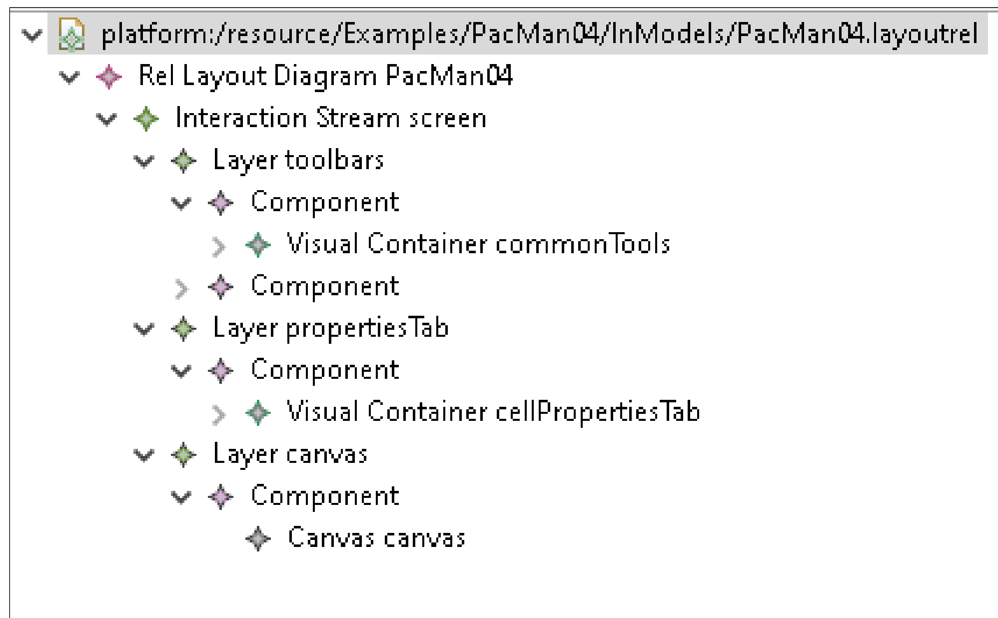


Fig. 4.6. Pac-Man interface model

4.2.2.3. **Interaction model**. Unlike in [**46**], our rules are designed to not require a button for each language element to instantiate. Instead, we will use contextual information to create element on the canvas. For example, Listing 4.10 shows how to add a grid node to the canvas just like Listing 4.5 to create cells in GoL. We then have one interaction rule per positionable entity to add (Listings 4.11 and 4.12 show Pac-Man and ghost). Note how we use different essential actions to trigger them from the same focus.

Listing 4.13 specifies the interaction to move a grid node. In this framework, the coordinates of the pointer dictate the target position of the element. Listings 4.14 and 4.15 specify how to show and hide the properties side panel.

Since the rules are event specified, we have to trigger different events. In Table 4.2, events are described with their operations and actions. We added three four different events for girdNode, pacMan instance, ghost instance and food instance.

```
1 InteractionRule AddGridNode
2  Condition {
3    focus Canvas {}
4  }
5  --- add -->
6  Effect {
7    Lang GridInstance {op = add}
```

```
8   }
```

Listing 4.10. Interaction rule to create a gridnode

```
1 InteractionRule AddPacMan
2   Condition {
3     focus Lang GridInstance {}
4   }
5   --- pac_add -->
6   Effect {
7     Lang PacManInstance {op = add}
8   }
```

Listing 4.11. Interaction rule to add a Pac-Man on a grid node

The example of Listing 4.14 and 4.13 show the implications of move and delete rules that is injected to move or delete events on the mapping models. These rules are only applied when a similar event is mapped on the mapping model with an event.

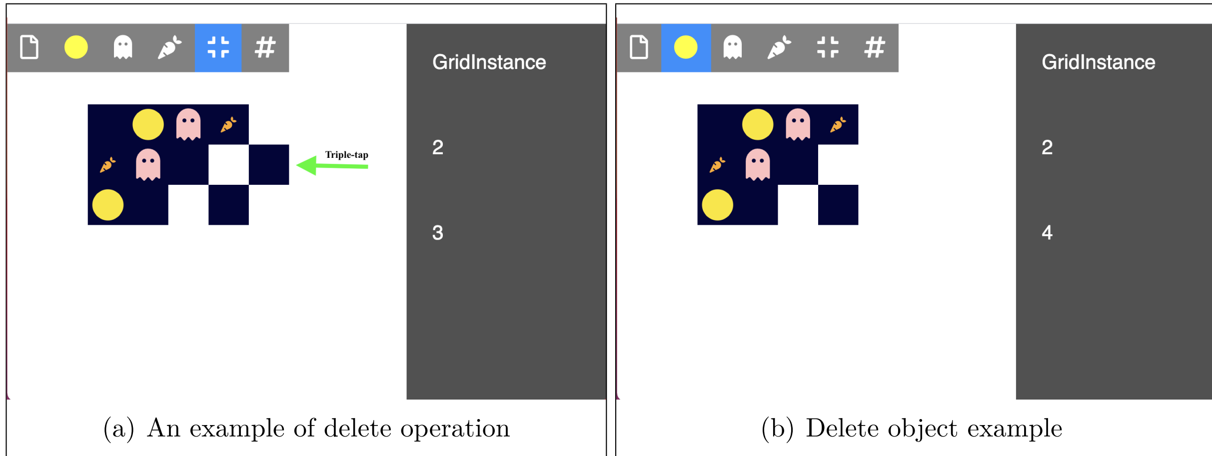
```
1 InteractionRule AddGhost
2   Condition {
3     focus Lang GridInstance{}
4   }
5   --- ghost_add -->
6   Effect {
7     Lang GhostInstance {op = add}
8   }
```

Listing 4.12. Interaction rule to add a ghost on a grid node

```
1 InteractionRule MoveGridInstance
2   Condition {
3     Lang GridInstance {value = marked}
4     Var moving {value = "true"}
5   }
6   --- move -->
7   Effect {
8     Lang GridInstance {op = move}
9     Var moving {value = "false"}
10  }
```

Listing 4.13. Interaction rule to move a grid node

```
1 InteractionRule ShowProperties
2   Condition {
3     focus Canvas {}
4   }
```



(a) An example of delete operation

(b) Delete object example

Fig. 4.7. An example of Pac-Man user interactions

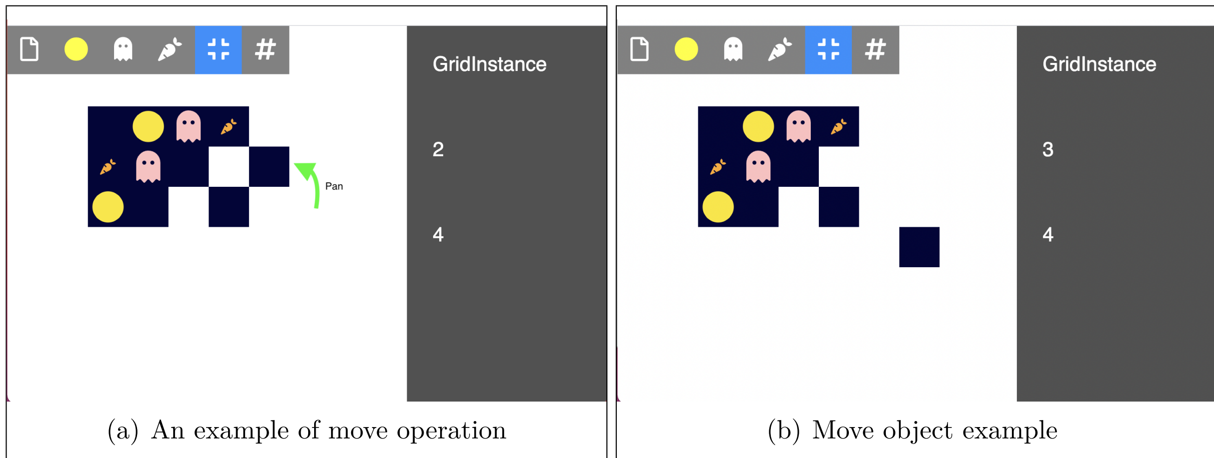
```
  --- show -->
  Effect {
    Interface cellPropertiesTab {value = "visible"}
    Interface namePropertie {value = "update"}
    Interface xPos {value = "update"}
    Interface yPos {value = "update"}
  }
```

Listing 4.14. Interaction rule to show the properties

```
InteractionRule HideProperties
  Condition {
    focus Canvas {}
  }
  --- hide -->
  Effect {
    Interface cellPropertiesTab {value = "hidden"}
    Interface namePropertie {value = "update"}
    Interface xPos {value = "update"}
    Interface yPos {value = "update"}
  }
```

Listing 4.15. Interaction rule to hide the properties

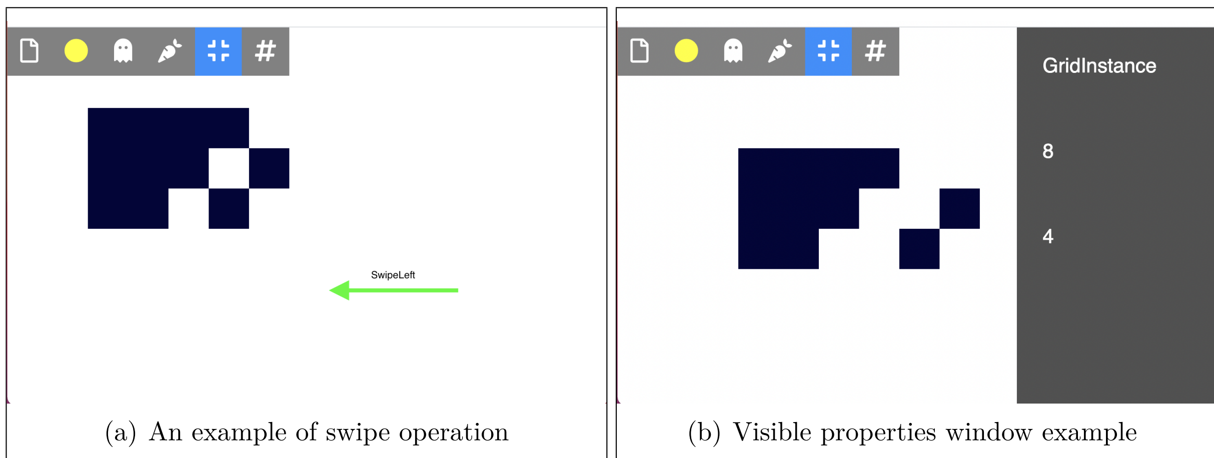
4.2.2.4. **Event mapping model**. The event mapping model associates the contextual essential actions to different touch gestures from the catalog. Table 4.2 shows this mapping. The editor in [**46**] required multiple user actions to perform each operation: e.g., click on the grid node toolbar button then click on the canvas to instantiate it. In our touch editor, the user can perform the same operation with a single action at the desired location with the touch of his fingers. As shown in Figure 4.7, triple-tapping any element will remove it.



(a) An example of move operation

(b) Move object example

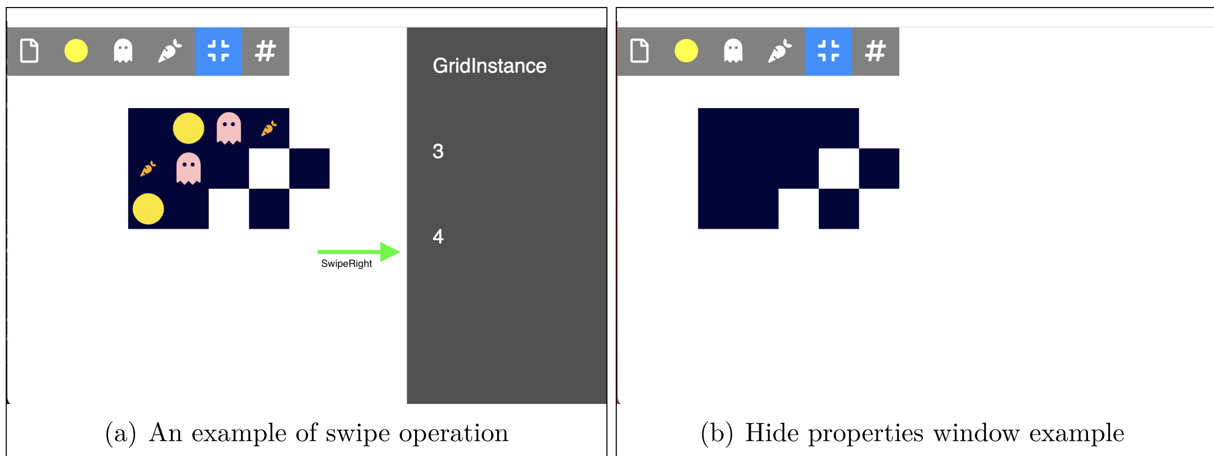
Fig. 4.8. An example of Pac-Man user interactions



(a) An example of swipe operation

(b) Visible properties window example

Fig. 4.9. An example of Pac-Man user interactions



(a) An example of swipe operation

(b) Hide properties window example

Fig. 4.10. An example of Pac-Man user interactions

Table 4.2. Pac-Man touch event and operations

Touch event	Operation
Tap on canvas	Create grid node instance
LongPress on grid node	Create Pac-Man instance
Double Tap on grid node	Create food instance
2 pointers Tap on grid node	Create ghost instance
Tap on element	Mark or select an element
Swipeleft with 2 pointers	Visible score widow
Swiperight with 2 pointers	Hide score window
Pan	Move an element on the canvas
Pinch in and out with 3 pointers	Zoom in and out the canvas
Triple tap on targeted element	Delete an element

If we wanted to move an object around the canvas, a pan gesture can execute the task (see Figure 4.8). The final example in Figure 4.9 and Figure 4.10 depicts that a swipe gesture on the canvas can make the properties panel appear or disappear.

4.2.3. Mind Map editor

In the final modeling editor example, we focus on manipulating links between elements of a model. In the following, we briefly describe the input models required to generate the editor.

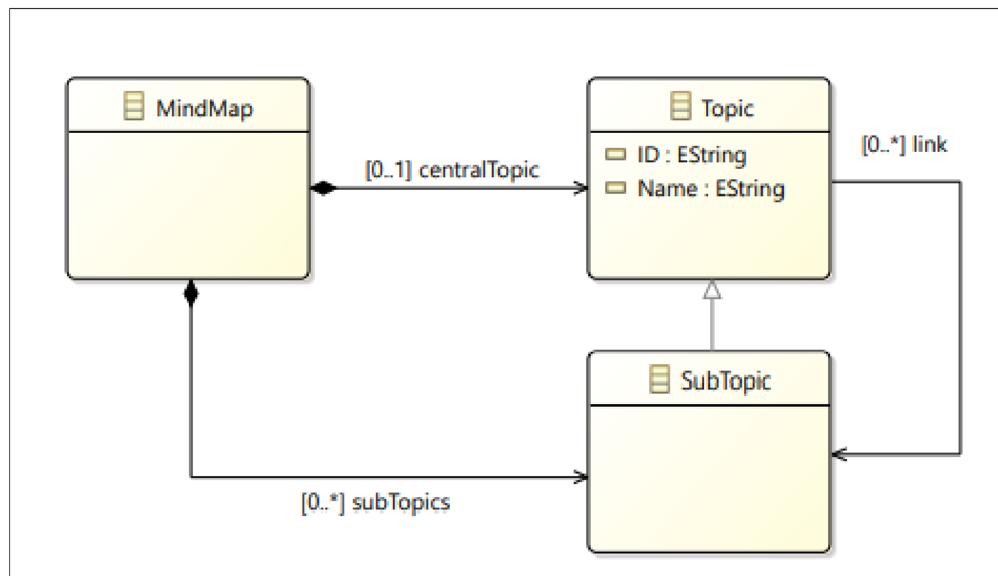


Fig. 4.11. The metamodel of Mind Map

4.2.3.1. **Metamodel**. The metamodel of the Mind Map DSL (Figure 4.11) focuses on only two elements: a topic and a sub-topic. A Mind Map has one central topic. It can be

linked to sub-topics which, in turn, can be linked to sub-topics, and so on. Topics have an id and a name.

4.2.3.2. **Interface model**. The interface model of the Mind Map editor has no element other than the canvas. The user must rely on touch gestures to create topics, subtopics and link them.

4.2.3.3. **Interaction model**. Since a Mind Map must always have at most one central topic we use very similar rules to create the topic and sub-topics. Listing 4.16 shows the interaction rule for creating the central topic on the canvas. It relies on the Var element has_topic to prevent it from triggering more than once. Listing 4.17 specifies the rule to create a sub-topic, but it is only applicable once a topic has been created first.

```
1 InteractionRule AddCentralTopic
2  Condition {
3    focus Canvas {}
4    not Var has_topic {value="true"}
5  }
6  --- add -->
7  Effect {
8    Lang Topic {op = add}
9    Var has_topic {value="true"}
10  }
```

Listing 4.16. Interaction rule to create a topic

```
1 InteractionRule AddSubTopic
2  Condition {
3    focus Canvas {}
4    Var has_topic {value="true"}
5  }
6  --- add -->
7  Effect {
8    Lang SubTopic {op = add}
9    Var adding_sub {value="true"}
10  }
```

Listing 4.17. Interaction rule to create a sub-topic

Listing 4.18 shows the interaction rules to create a link between two sub-topics. The link requires a source and a target elements. The rules rely on the linking Var element to indicate that is in link creation mode and not to interfere with other rules.

```
1 InteractionRule StartLink
2  Condition {
3    focus Lang Topic {}
```

```
4      not Var linking {value = "true"}
5    }
6    --- source -->
7    Effect {
8      Var linking {value = "true"}
9      Var source {op=copy, from=focusElement}
10   }
11
12 InteractionRule StartSubLink
13   Condition {
14     focus Lang SubTopic {}
15     not Var linking {value = "true"}
16   }
17   --- source -->
18   Effect {
19     Var linking {value = "true"}
20     Var source {value=focusElement}
21   }
22
23 InteractionRule EndLinkOnSubTopic
24   Condition {
25     focus Lang SubTopic {}
26     Var linking {value = "true"}
27   }
28   --- target -->
29   Effect {
30     Var linking {value = "false"}
31     Lang Link {op = add, value=[source,focusElement]}
32   }
33
34 InteractionRule EndLinkOnCanvas
35   Condition {
36     focus Canvas {}
37     Var linking {value = "true"}
38   }
39   --- target -->
40   Effect {
41     Var linking {value = "false"}
42   }
43
44 InteractionRule EndLinkOnTopic
45   Condition {
46     focus Lang Topic {}
47     Var linking {value = "true"}
48   }
```

```
49  --- target -->
50  Effect {
51    Var linking {value = "false"}
52  }
```

Listing 4.18. Rules of creating link between two elements

4.2.3.4. **Event mapping model.** In the mapping model in Listing 4.19, we show that topics and sub-topics are created by double-tapping on the canvas. On lines 2 and 3, creating links is performed by a pan gesture, starting from the source topic and ending on the target sub-topic.

```
1 I#add#canvas.on("doubletap", function(ev) {MindMap.Interaction.prototype.triggerEvent(event,
     event.path[0].getAttribute("data-type"),'add')});
2 I#source#Hammer(document).on("panstart", function(ev) {MindMap.Interaction.prototype.
     triggerEvent(event,event.path[0].getAttribute("data-type"),'source')});
3 I#target#Hammer(document).on("panend", function(ev) {MindMap.Interaction.prototype.triggerEvent(
     event,event.path[0].getAttribute("data-type"),'target')});
```

Listing 4.19. Mapping the events to a doubletap and pan gestures interaction

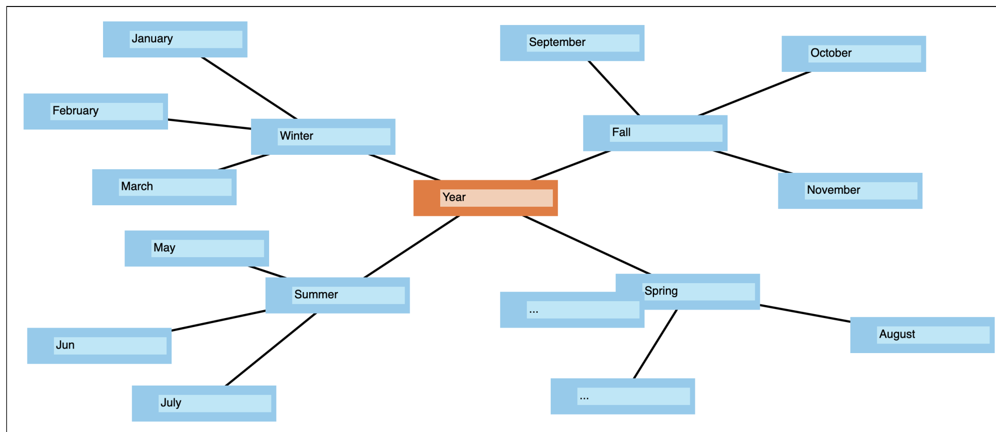


Fig. 4.12. An example of a Mind Map created by the touch editor

Figure 4.12 shows a topic connected with to sub-topics; and inner links between sub-topics created as a result of the touch interactions with the editor.

4.3. Discussion

We wanted to show different cases that are possible to interact with the DSL editors. The three examples have shown the feasibility of our approach. Our initial experiments with the prototype show that it is quite natural and requires less user interactions to execute a task than non-touch environments. In [**46**], the authors experimented the tasks with traditional mouse and keyboard interactions. For example, if Alice wants to create a grid node instance

on the canvas then create a Pac-Man instance inside the grid node, she will need to perform several clicks and mouse interaction. First, Alice selects the grid button in the menu with a mouse click and clicks on the canvas to create the grid node instance. Second, she chooses the other instance button (i.e., Pac-Man, Food or Ghost instance) from the menu bar, then selects the grid node, and clicks inside the grid node to add the selected instance. In total, the process requires over five clicks and mouse interactions. If Alice performs the same task with our prototype, it would require only two gesture interactions (e.g., Tap and LongPress) to complete the process (see Table 4.2). Therefore, without having mouse clicks, the user can go directly on the element itself. In this particular case, our approach reduces the number of interaction points but also the physical movements required thanks to domain-specific touch events.

Touch gestures promote quite natural and direct interactions to operate with elements on the interface. Our intuition is that it is more natural for some users. Some users still like to use mouse and keyboard, but for other users it is more natural. The interaction we make with traditional devices, like a keyboard and a mouse, does not reflect naturally what is happening on the screen. With our prototype, we bring the experience closer to what is happening on the screen. The user can directly touch the screen and interact with the screen so that the movement he makes, make more sense and fit more naturally inside the editor.

An obstacle occurred while a user uses a double-tap gesture to execute a specific task. As we know, a double-tap is a simple extension of the single touch where the second touch must occur within a short delay after the first touch, and the second touch must also follow the same timing and positional requirements as the first touch. We defined that double-tap only be triggered while tap occurs on an element. However, the user's first tap was on the element and the second tap on the canvas. Therefore, we should keep in mind that if an engineer is implementing a double-tap gesture, additional guidelines must be provided to the user to ensure that he is not executing two separate taps on the surface.

The obstacle occurred while a user uses a double-tap gesture to execute a specific task. As we know, a double-tap is a simple extension of the single touch where the second touch must occur within a certain amount of time after the first touch, and the second touch must also follow the same timing and positional requirements as the first touch. We defined that double-tap only be triggered while tap occurs on an element. However, the user's first tap was on the element and the second tap on the canvas. Therefore, keep in mind that if you are implementing a double-tap gesture, explain the additional guideline to ensure that the user is not executing a double touch on our surface.

While dragging a gesture to link between two or multiple elements, it is often needed at the gesture-recognition level from source and target point. Since the touch controller only reports coordinates when a finger touches the bounding box of an element, the application can treat those coordinate reports as a drag. Otherwise,the touch reports can be ignored or

continually analyzed for other events. We are limited to what the touch library provides, as we use HammerJS and EventJS libraries, we cannot do more interactions that are not inside the library.

The proposed examples represent a distinctive set of user interactions to examine and provide editors that create the modeling environment for DSMLs. The editor mostly designs and depends on the rules of a prototypical approach to the domain. Our prototype examines the graphical user interaction in a modeling environment. The DSL editor offers a limited interaction that we implemented and mapped; it is not a complete modeling editor to apply every interaction. However, implementing different examples with more graphical interaction could be the considerably new generation of modeling editors that could support higher numbers of interaction rules and deliver a complete package of multi-touch gesture facilities.

We proposed three different examples with distinct metamodels and interaction rules to examine domain-specific user interactions in a multi-touch environment. The example of GoL shows how to instantiate elements more naturally on the canvas by touch gestures. The Pac-Man example designs with different metamodel types and containment relationships. The Mind Map example focuses on links among different elements. Therefore, we are confident that we can cover most interactions for domain-specific modeling editors.

Chapter 5

Conclusion

We conclude by summarizing the contributions of this thesis and outlining future work.

5.1. Summary

In this thesis, we presented an initial catalog of possible gestures that are useful for touch-enabled graphical modeling editors. The catalog describes the events and their typical usage. This will help language engineers to define the interaction models. Additionally, they can change the interactions based on the demands of domain experts by customizing the basic touch events.

To examine the domain-specific operations, we implemented three examples. We developed domain-specific interaction rules and event library, injected the library into the framework to respond to specific gestures. The set of examples we have shown covers most operations required by a modeling editor.

5.2. Outlook

Our approach deals with the multi-touch gestures supported by external available libraries. Though we extended the gesture catalog from the existing touch libraries, we are still limited in the extensions. Particularly, we are restricted with their offered events and actions. Therefore, we can not implement new gesture events. However, a potential new version of the touch library may offer more interaction on DSM editors. Investigating further supported gestures may expand the possibilities of interactions with the editor. For example, we could take into consideration the accelerometer of a mobile device to interact with the editor.

Currently, the interaction rules rely only on one focus point. In the future, we will explore how to have multiple focus points to really enable multi-touch with multiple contexts. For example, if we select two different objects and swipe them together, it may merge them

together. Also, if we select two elements and swipe between them, it may create a link between them.

This could be extended for collaborative modeling. For that, the framework should enable recognizing multiple people on the same touch device. This adds on multiple focus points, but from different people. This can enable triggering multiple interaction rules concurrently.

Bibliography

[1] https://modeling-languages.com/comparing-tools-build-graphical-modeling-editors/.

[2] https://union.co/articles/gesture-based-technology.

[3] Abrahão, S., Bourdeleau, F., Cheng, B., Kokaly, S., Paige, R., Stöerrle, H., and Whittle, J. User experience for model-driven engineering: Challenges and future directions. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)* (2017), IEEE, pp. 229–236.

[4] Bézivin, J. In search of a basic principle for model driven engineering. *Novatica Journal, Special Issue 5*, 2 (2004), 21–24.

[5] Brown, A. W. Model driven architecture: Principles and practice. *Software and Systems Modeling 3*, 4 (2004), 314–327.

[6] Budinsky, F., Ellersick, R., Steinberg, D., Grose, T. J., and Merks, E. *Eclipse modeling framework: a developer's guide*. Addison-Wesley Professional, 2004.

[7] Butz, A., Beyer, G., Hang, A., Hausen, D., Hennecke, F., Lauber, F., Loehmann, S., Palleis, H., Rümelin, S., Slawik, B., et al. Out of shape, out of style, out of focus. *Informatik-Spektrum 37*, 5 (2014), 390–396.

[8] Damm, Christian Heide, K. M. H., and Thomsen, M. Tool support for cooperative object-oriented design: gesture based modelling on an electronic whiteboard. In *In Proceedings of the SIGCHI conference on Human Factors in Computing Systems* (2000), pp. 518–525.

[9] Dellas, C. M., and Hogan, K. M. Statechart development environment with embedded graphical data flow code editor, Feb. 26 2013. US Patent 8,387,002.

[10] Erdweg, S., Van Der Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W. R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., et al. The state of the art in language workbenches. In *International Conference on Software Language Engineering* (2013), Springer, pp. 197–217.

[11] Fondement, F. Graphical concrete syntax rendering with svg. In *European Conference on Model Driven Architecture-Foundations and Applications* (2008), Springer, pp. 200–214.

[12] Frisch, Mathias, J. H., and Dachselt, R. Investigating multitouch and pen gestures for diagram editing on interactive surfaces. In *In Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces* (2009), pp. 149–156.

[13] (GMF), G. M. F. http://www.eclipse.org/gmf-tooling/.

[14] Grundy, J., and Hosking, J. Supporting generic sketching-based input of diagrams in a domain-specific visual language meta-tool. IEEE, pp. 282–291.

[15] Hammer. https://hammerjs.github.io/getting-started/.

[16] Harel, D., and Naamad, A. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM) 5*, 4 (1996), 293–333.

[17] Harel, D., and Rumpe, B. Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff. Report, Weizmann Institute Of Sience, 2000.

[18] Hemel, Z., Kats, L. C., Groenewegen, D. M., and Visser, E. Code generation by model transformation: a case study in transformation modularity. *Software & Systems Modeling 9*, 3 (2010), 375–402.

[19] InterectJs. https://interactjs.io/.

[20] Kammer, D., Wojdziak, J., Keck, M., Groh, R., and Taranko, S. Towards a formalization of multi-touch gestures. In *ACM International Conference on Interactive Tabletops and Surfaces* (2010), pp. 49–58.

[21] Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., and Völkel, S. Design guidelines for domain specific languages. *arXiv preprint arXiv:1409.2378* (2014).

[22] Kelly, S., and Tolvanen, J.-P. *Domain-specific modeling: enabling full code generation.* John Wiley & Sons, 2008.

[23] Kin, K., Hartmann, B., DeRose, T., and Agrawala, M. Proton: multitouch gestures as regular expressions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2012), pp. 2885–2894.

[24] Kolovos, D. S., García-Domínguez, A., Rose, L. M., and Paige, R. F. Eugenia: towards disciplined and automated development of gmf-based graphical model editors. *Software & Systems Modeling 16*, 1 (2017), 229–255.

[25] Language, O. C. https://www.omg.org/spec/ocl/2.4/.

[26] Lü, H., Fogarty, J. A., and Li, Y. Gesture script: Recognizing gestures and their structure using rendering scripts and interactively trained parts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2014), pp. 1685–1694.

[27] Lü, H., and Li, Y. Gesture coder: a tool for programming multi-touch gestures by demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2012), pp. 2875–2884.

[28] Ma, Z., Yeh, C.-Y., He, H., and Chen, H. A web based uml modeling tool with touch screens. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering* (2014), pp. 835–838.

[29] Masiero, P. C., Maldonado, J. C., and Boaventura, I. A reachability tree for statecharts and analysis of some properties. *Information and Software Technology 36*, 10 (1994), 615–624.

[30] Mens, T., and Van Gorp, P. A taxonomy of model transformation. *Electronic notes in theoretical computer science 152* (2006), 125–142.

[31] Mohagheghi, P., Gilani, W., Stefanescu, A., Fernandez, M. A., Nordmoen, B., and Fritzsche, M. Where does model-driven engineering help? experiences from three industrial cases. *Software & Systems Modeling 12*, 3 (2013), 619–639.

[32] Mosses, P. D. Formal semantics of programming languages:—an overview—. *Electronic Notes in Theoretical Computer Science 148*, 1 (2006), 41–73.

[33] Neumann, N. G., and Hartadinata, T. Debugging a statechart using a graphical program, Jan. 28 2014. US Patent 8,640,100.

[34] Newton, A. *MooTools Essentials: The Official MooTools Reference for JavaScript and Ajax Development.* Apress, 2008.

[35] object facility, M. https://www.omg.org/spec/qvt/.

[36] OMG. Unified modeling language. `https://www.omg.org/spec/UML/2.5.1/`, dec 2017. formal/2017-12-05.

[37] Oney, S., Krosnick, R., Brandt, J., and Myers, B. Implementing multi-touch gestures with touch groups and cross events. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (2019), pp. 1–12.

[38] Parra, Otto, S. E., and Panach, J. I. Extending and validating gestui using technical action research. In *In 2017 11th International Conference on Research Challenges in Information Science (RCIS)* (2017), pp. 341–352.

[39] Parra, Otto, S. E., and Pastor, O. Gestui: a model-driven method and tool for including gesture-based interaction in user interfaces. *Complex Systems Informatics and Modeling Quarterly* (2016), 73–92.

[40] Phan, C. T. Developing a hybrid mobile application with ionic: Case: Kunkku application and super-app oy.

[41] Rohlfshagen, Philipp, J. L. D. P.-L., and Lucas, S. M. Pac-man conquers academia: Two decades of research using a classic arcade game. *IEEE Transactions on Games 3*, 10 (2017), 233–256.

[42] Schmidt, D. C. Model-driven engineering. *Computer-IEEE Computer Society- 39*, 2 (2006), 25.

[43] Scholliers, C., Hoste, L., Signer, B., and De Meuter, W. Midas: a declarative multi-touch interaction framework. In *Proceedings of the fifth international conference on Tangible, embedded, and embodied interaction* (2010), pp. 49–56.

[44] Selic, B. The pragmatics of model-driven development. *IEEE software 20*, 5 (2003), 19–25.

[45] Sirius. http://www.eclipse.org/sirius/.

[46] Sousa, V., Syriani, E., and Fall, K. Operationalizing the integration of user interaction specifications in the synthesis of modeling editors. In *Software Language Engineering* (Athens, oct 2019), ACM, pp. 42–54.

[47] Sprinkle, J., Rumpe, B., Vangheluwe, H., and Karsai, G. Metamodelling: state of the art and research challenges. In *Proceedings of the 2007 International Dagstuhl conference on Model-based engineering of embedded real-time systems* (2007), pp. 57–76.

[48] Stürmer, I., Conrad, M., Doerr, H., and Pepper, P. Systematic testing of model-based code generators. *IEEE Transactions on Software Engineering 33*, 9 (2007), 622–634.

[49] Syriani, E., Luhunu, L., and Sahraoui, H. Systematic mapping study of template-based code generation. *Computer Languages, Systems Structures 52*, 1 (2018), 43–62.

[50] Syriani, E., Luhunu, L., and Sahraoui, H. Systematic mapping study of template-based code generation. *Computer Languages, Systems & Structures 52* (2018), 43–62.

[51] Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., and Ergin, H. AToMPM: A Web-based Modeling Environment. In *Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition* (2013), vol. 1115 of *MODELS'13*, CEUR-WS.org, pp. 21–25.

[52] Technology, T. https://www.slideshare.net/iaminyoung/smart-phone-touch-technology-20111104/.

[53] Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C., Visser, E., Wachsmuth, G., et al. *DSL engineering: Designing, implementing and using domain-specific languages*. dslbook. org, 2013.

[54] Warmer, J. B., and Kleppe, A. G. *The object constraint language: getting your models ready for MDA*. Addison-Wesley Professional, 2003.

[55] Whittle, J., Hutchinson, J., and Rouncefield, M. The state of practice in model-driven engineering. *IEEE software 31*, 3 (2013), 79–85.

[56] Wüest, D., Seyff, N., and Glinz, M. Flexisketch: A mobile sketching tool for software modeling. In *International Conference on Mobile Computing, Applications, and Services* (2012), Springer, pp. 225–244.

[57] ZingTouch. https://zingchart.github.io/zingtouch/.