Model-Based Multi-touch Gesture Interaction for Diagram Editors

Florian Niebling, Daniel Schropp, Romina Kühn, and Thomas Schlegel

Institute of Software- and Multimedia-Technology, Technische Universität Dresden, Dresden, D-01062, Germany
{florian.niebling,thomas.schlegel,romina.kuehn}@tu-dresden.de, d.schropp@gmx.de

Abstract. Many of todays software development processes include model-driven engineering techniques. They employ domain models, i.e. formal representations of knowledge about an application domain, to enable the automatic generation of parts of a software system. Tools supporting model-driven engineering for software development today are often desktop-based single user systems. In practice though, the design of components or larger systems often still is conducted on whiteboards or flip charts. Our work focuses on interaction techniques allowing for the development of gesture-based diagram editors that support teams in establishing domain models from a given meta-model during the development process. Users or groups of users are enabled to instantiate meta-models by free-hand or pen-based sketching of components on large multi-touch screens. In contrast to previous work, the description of multi-touch gestures is derived directly from the graphical model representing the data.

Keywords: Multi-touch gestures, model-based development.

1 Graphical Model-Driven Development

To allow for the graphical modeling of artifacts according to a given data model, graphical models can be used to represent features of the data model. These models contain shapes and containers providing a graphical description of data models and supporting the development of graphical diagram editors. One example of graphical modeling within the Eclipse framework is the Graphical Editing Framework (GEF) [13], which provides methods for the creation of graphical editors for the Eclipse Modeling Framework (EMF). The Graphiti Toolkit [7] based on GEF provides a graphical model for the representation of model instances, the Graphiti pictogram model. In our prototypical diagram editor, instances of a data-model can be created and manipulated by interacting with graphical representations specified using the Graphiti pictogram model, which are linked to the appropriate elements of the data model (see Figure 1).

M. Kurosu (Ed.): Human-Computer Interaction, Part II, HCII 2014, LNCS 8511, pp. 121–130, 2014.

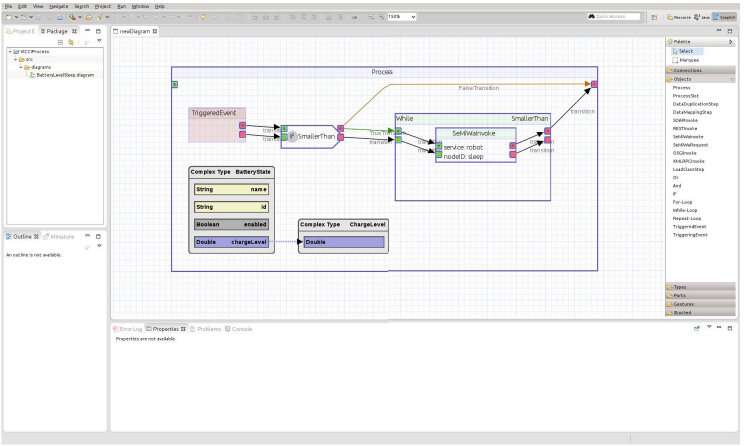

Fig. 1. Model-driven diagram editor based on Eclipse

1.1 Formal Representation of Gestures

Explicit methods for gesture recognition are based on patterns of strokes that are compared to the user input and evaluated regarding their similarity. To represent these strokes, previous developments use domain-specific languages to define multi-touch interaction such as the Gesture Description Language (GDL) [9] or the Gesture Markup Language (GML). In contrast, we propose to use the graphical representation of artifacts that is already present in the graphical model to derive multi-touch gestures. We identified three modes of gesture interaction that have been employed by users to sketch the various graphical items specified using the pictogram model: Single-touch / single-stroke, single-touch / multi-stroke, and multi-touch / multi-stroke (see Figure 2).

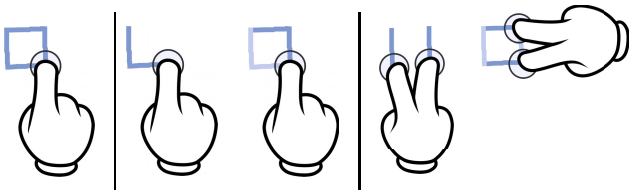

Fig. 2. (left) single-touch / single-stroke. (middle) single-touch / multi-stroke. (right) multi-touch / multi-stroke.

2 Related Work

Interaction with diagram editors was been simplified by enabling finger- or pen-based gestural sketching input. Plimmer et al. [12] give some overview about sketching tools developed for multiple application domains, such as UML modelling and UI generation, as well as on the gesture recognition algorithms that

have been employed in the various works. Rubine's algorithm [14], a single-stroke pattern-matching algorithm, is one of the recognition techniques employed e.g. in InkKit [12], SUMLOW[5] and the Knight UML Designer [6]. Rubine's algorithm performs a comparison between features extracted from registered patterns and features extracted from user input. In the implementation in InkKit, the recognition process is started manually, while in SUMLOW, a timer is responsible for starting the recognition.

This is regarded as a drawback by Alvardo et al., who provide continuous recognition in their SketchREAD engine [1], using a gesture recognition algorithm based on bayesian networks.

In their SKETCH framework [15], Sangiorgi et al. employ a recognition algorithm based on the Levenshtein Distance [10], using string-based descriptions of gestures describing cardinal directions. The development effort on SKETCH seems to have ceased since 2010.

Scribble [16] is a GEF-based framework which allows for a seamless extension of GEF editors with gesture input. Since no pre-generated patterns are used, users are enabled to choose their own gestures, making the framework usable in many different application domains. The GEF editors that are augmented by Scribble have to be trained by the user to support their respective gestures of choice.

The related work shows the relevance of gesture-based input for diagram editors in multiple application domains. Multiple methods towards gesture recognition have been evaluated, with descriptions of gestures being either programmatic, pattern-based or feature-based. In contrast to the existing work, we propose methods for generating gesture description from the graphical models used to represent entities of the application domain.

3 Specification of Graphical Models

A *Graphical Model* contains graphical representations of the elements contained in a *Data Model* that represents concepts of the underlying application domain. In the context of workflow editing, a model-based graphical workflow editor contains a model of the workflow items (i.e. *activity*, *event*, *loop*, *connection*, etc.), and a graphical model containing graphical representations of these items (i.e. *rectangle*, *diamond shape*, *line*, etc.). By selecting graphical shapes in the editor, the user is enabled to instantiate concepts of the underlying data model.

In our prototypical application, we extended the graphical modeling framework Graphiti to allow for sketching of instances of the underlying *pictogram* model used by Graphiti. As can be seen in Figure 3, we extended Graphiti's *Diagram Editor* to make use of a *Gesture Recognizer*, that is able to detect shapes contained in the graphical model of the application. On detection of sketched fragments of the graphical model, instances of the underlying data model are created and the associated feature of the graphical model is added to the editor's scenegraph.

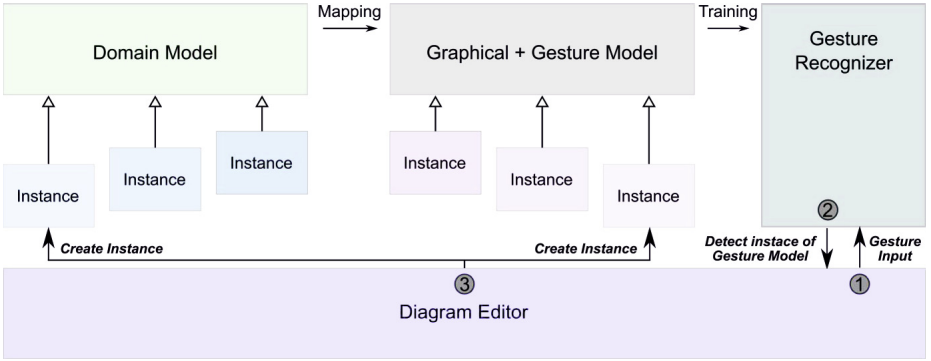

Fig. 3. Architecture of the prototypical diagram editing framework. Instances of the gesture model are recognized by a $N gesture recognizer, features of the data model and the graphical model are instantiated for display by the system.

3.1 Recognition of Sketched Graphical Models

To be able to recognize sketched user input, fragments of the graphical model, such as can be seen in Figure 5, have to be able to be detected by the *Gesture Recognizer*. We would like to give a short introduction about the methods that are employed in the *Gesture Recognizer* for sketch recognition, and for transformation of the *pictogram* model fragments to reference templates for the recognizer component.

The $1-Recognizer, a pattern-matching algorithm for single-stroke gestures, was introduced by Wobbrock et. al. [17]. It uses simple lists of coordinates as patterns for gesture recognition, which are compared to user input. The algorithm is implemented in four steps:

- Resampling. Because of different speed of user input, gestures contain different numbers of input points. In this step, the point path is resampled to contain a certain number of equidistant points, Wobbrock et. al. propose to use 64 points per point path.
- Rotation. Point paths are rotated in negative direction such that the *indicative angle*, the angle formed between the centroid of the gesture and the gestures first point, is 0.
- Scale and translation. After scaling the point path to a reference square, the centroid of the point path is translated to (0,0).
- Recognition. The point path is continuously rotated to find the minimum path-distance between the point path and supplied reference patterns.

The $1 recognizer's main benefits are simple implementation, high speed and that extensive training is unnecessary. This simplicity comes with several drawbacks. The algorithm is not able to distinguish input according to its orientation, aspect ratio or position, making it impossible to differentiate between e.g. squares and non-square rectangles. Also, $1 is not usable for multi-stroke gestures.

Listing 1.1. or.pictograms

```
<?xml version="1.0" encoding="UTF-8"?>
<pi:Diagram xmi:version="2.0"
 xmlns:xmi="http://www.omg.org/XMI"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:al="http://eclipse.org/graphiti/mm/algorithms"
      xmlns:pi="http://eclipse.org/graphiti/mm/pictograms"
      visible="true" active="true" name="">
  <children xsi:type="pi:ContainerShape" visible="true" active="true">
    <graphicsAlgorithm xsi:type="al:Polyline" foreground="//@colors.0"
        lineWidth="2" width="40" height="40">
      <points x="0"  y="20"/>
      <points x="20" y="0"/>
      <points x="40" y="20"/>
      <points x="20" y="40"/>
      <points x="0"  y="20"/>
    </graphicsAlgorithm>
    <children visible="true">
      <properties key="gesture" value="1"/>
      <graphicsAlgorithm xsi:type="al:Ellipse"
          foreground="//@colors.0" lineWidth="3" filled="false"
          width="18" height="18" x="11" y="11"/>
    </children>
    <children>
      <properties key="gesture" value="2"/>
      <graphicsAlgorithm xsi:type="al:Polyline"
          foreground="//@colors.0">
        <points x="0"   y="0"/>
        <points x="50"  y="100"/>
        <points x="100" y="0"/>
      </graphicsAlgorithm>
    </children>
  </children>
  <colors red="102" green="102" blue="255"/>
</pi:Diagram>
```

Fig. 4. Instance of Graphiti *pictograms* model representing the *or* data model instance of our prototypical workflow editor

Protractor [11] was developed to address some shortcomings of the $1 recognizer, the key difference being sensitivity to orientation, making it possible to distinguish eight base orientations. The $N recognizer [2] improves on the $1 algorithm, allowing for the representation of multi-stroke gestures as single-stroke gestures, combining the last point in a stroke with the first point of the following stroke. This allows for the recognition of a mixture of single- and multi-strokes. The $N Protractor [3] is a combination of the $N recognizer and the aforementioned Protractor algorithm.

The main reason for selecting the $N family of algorithms for our application framework is the simplicity of transforming fragments of the employed graphical model into patterns for the recognizer. The coordinate lists contained in the *al:Polyline* elements of the graphical model, as well as *al:Rectangle* and *al:Polygon* elements, as can be seen in Figure 5, can be easily converted into required coordinate lists for the $N recognizer. For shapes such as *al:Ellipse* and *al:RoundedRectangle*, we sample the shape to provide the appropriate coordinate lists.

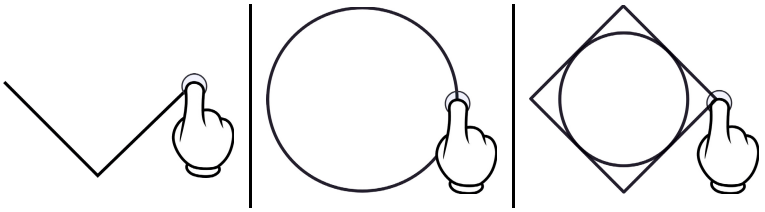

Fig. 5. Input possibilities for the *or* element: (left) mathematical symbol. (middle) circular gesture. (right) circular + diamond gesture.

3.2 Functionality of the Workflow Editor

We selected an existing EMF-based workflow editor to evaluate our prototypical gesture based sketching framework. The editor uses traditional mouse based interaction according to the WIMP concept. In addition to interactions performed using existing Eclipse interfaces, possible interactions with the editor are separated into two categories. As can be seen in Figure 1, the right side of the editor contains a list of workflow objects that can be dragged to the main diagram in the middle of the workspace, instantiating entities of the graphical model and placing them inside the diagram. Inside the diagram, existing workflow objects can be moved, deleted or connected using transitions between ports contained in workflow objects.

3.3 Integration of Gesture Recognition

The extended architecture of the workflow editor can be seen in Figure 6. Input, processing and detection of gesture based sketching are enabled inside Graphiti's *Diagram Editor* and our additional *Gesture Recognizer*. The link between Graphiti and gesture recognition is Graphiti's *Interaction Component*, where touch input is received and forwarded to the newly introduced gesture recognizer. Upon detection of one of the entities provided in the graphical model based on Graphiti's *pictogram model*, the *Diagram Type Provider* is used to instantiate the particular entities of the data model and the graphical model respectively.

4 Evaluation

The evaluation of our prototype was performed on a Dell XPS One 27 featuring a capacitive multi-touch display. The gestures that were automatically generated from the graphical model of our workflow editor were evaluated in a user study involving 15 participants. Furthermore, a comparison between the existing, traditional mouse-based interaction and gesture based usage was performed. This was done to gather evidence towards if expected advantages of gesture based sketching, such as higher intuitiveness, or expected disadvantages such as fat-finger problem or user fatigue, dominate the user experience. The evaluation

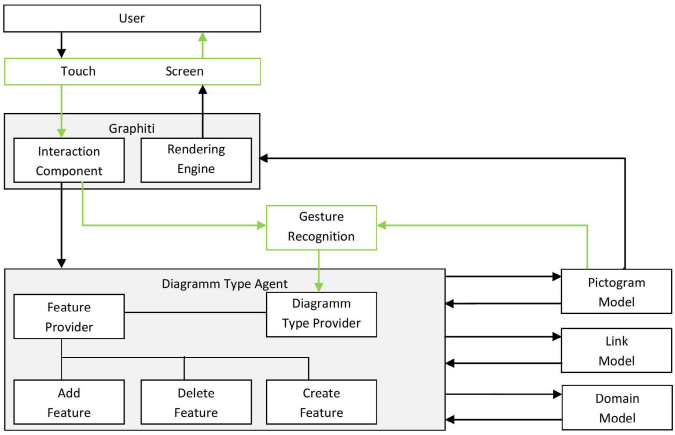

Fig. 6. Integration of gesture based sketching into the Graphiti architecture (adapted from Brand et al. [4])

was performed using two different quantitative methods, a formative user survey, and user transparent observation of behaviour. Participants in the study have not been involved in the development of the gesture recognition, although most of them belonged to the same faculty with similar background in software engineering, the application domain of the tasks in the user study.

To achieve the above mentioned goals, the following scenario was prepared. The participants were to sketch the graphical workflow elements *Process*, *If*, *Or*, *And*, *Loop* and *Ports*. *Ports* belonging to some of the elements were to be connected using *Transitions*. The workflow editor was to detect sketched workflow elements and position them at the respective position in the diagram.

Indicators that were rated were based on the NASA TLX evaluation [8] to assess cognitive and physical demands, overall effort, mental effort, physical effort, temporal effort, time pressure, performance and frustration levels. The observer that was monitoring the participants was mainly passive. Although the sequence of user tasks was arranged through the use of a survey sheet, the approach towards the solution of each task was presented to be open to the preferences of the user. The advances of the users were logged in the background and analyzed afterwards.

4.1 Evaluation Results

Following a short introduction of the evaluated categories are an evaluation of the most interesting results of the user study. Mental effort of gesture based sketching was perceived to be lower as traditional mouse interaction throughout the study. Further, a significant reduction in mental effort between the first and the following tasks leads to the impression that the method of interaction is

learned after a very short period of familiarization, and can thus be characterized as intuitive.

Physical effort was perceived to be higher using gestures than using mouse interaction, a result that was to be expected since gesture interaction requires free movement of a stretched arm in mid-air in front of the display. Physical effort seems to be a fundamental weakness of gesture interaction, which is also seconded by results in the overall effort category. Temporal effort was also perceived to be higher for gesture interaction, even on tasks where measurements of the time requirements for mouse based and gesture based interaction where similar. Overall values for frustration where quite high when recognition of sketched objects failed. This happened mainly in the sketching of ports, with a recognition rate as low as 67.5%, where recognition rates for the other workflow elements reliably achieved between 90% and 100%. Further evaluation has since shown that the low rate of recognition of ports was due to problems with the positioning of the performed sketching. Multiple users have tried to sketch ports slightly on the outside of existing workflow objects, when ports were actually only added to workflow objects when the sketching was performed on the inside of an object, due to programming errors in the prototype.

Although multiple participants of the study voiced their discomfort with longer periods of gesture interaction on a desktop computer due to physical effort, overall evaluation has shown that users accepted the process of gesture based sketching of graphical representations as equal to mouse based interaction.

The decision to allow for multi-stroke gestures has to be reconsidered, as only two of the 15 participants made active usage of multi-stroke sketching for the *And* element, even after being explicitly advised towards the possibility of multi-stroke sketching.

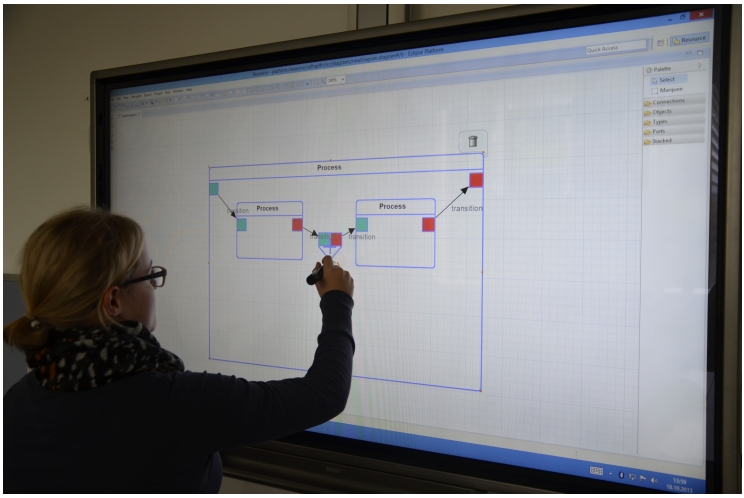

Fig. 7. Pen-input on interactive whiteboard

Multiple users intuitively reduced complex geometries to simpler gestures that represented subsets of the graphical representation of objects. E.g., the surrounding diamond of the graphical representations of the *And* and *Or* elements (see Figure 5) have been disregarded by most users, leaving just a simple *circle* gesture for the *Or* element and a *plus* gesture for the *And* element.

All participants but one have sketched transitions in a straight line between workflow objects, even when the final graphical representation of a transition was not a straight line to avoid cutting existing workflow elements.

5 Summary and Conclusion

We have evaluated a method for diagram sketching where gestures were automatically derived from the underlying graphical model of the application. A prototypical workflow editor based on Eclipse and Graphiti was augmented to support the generation of templates for a gesture recognizer from the Graphiti *pictogram* model. A formative user study was performed to evaluate user interaction with the modified editor.

As a fundamental difference towards previous work, the presented concept and prototypical implementation allows for collaborative multi-user interaction using multi-touch multi-stroke gestures. Evaluation with single user interaction on a desktop PC have shown that the sketching of workflows was accepted to be largely equivalent to mouse-based interaction concerning the preparation of workflow diagrams. Follow-up testing has shown tendencies that collaborative scenarios featuring digital whiteboards are promising targets for further user studies. Independent of the testing environment, our evaluations have shown that gestures derived from graphical models are accepted as input methods by users. The intuitive reduction of graphical representations by users towards simpler geometric subsets suggests further areas of research towards automatic generation of intuitive gestures from graphical models.

6 Future Work

Several different methods that support graphically similar objects need to be evaluated in future work. First, using context to allow the system to choose the item that is perceived to be of higher probability. Second, the support of similar objects using the same gestures, with additional pop-up menus allowing the user to choose one of the different objects. Third, further evaluation which parts of the graphical model are perceived by users to carry the most significance or relevance. Identifying parts that are perceived to be meaningful by users given a graphical representation is also necessary the more complex graphical models become.

Further work is also needed in the evaluation of introducing mobile multi-touch devices such as tablets into the software modeling process, expanding the collaborative user environment from single devices such as whiteboards to multiple devices.

References

1. Alvarado, C., Davis, R.: Sketchread: A multi-domain sketch recognition engine. In: Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology, UIST 2004, pp. 23–32. ACM, New York (2004)
2. Anthony, L., Wobbrock, J.O.: A lightweight multistroke recognizer for user interface prototypes. In: Proceedings of Graphics Interface 2010, GI 2010, pp. 245–252. Canadian Information Processing Society, Toronto (2010)
3. Anthony, L., Wobbrock, J.O.: $N-protractor: A fast and accurate multistroke recognizer. In: Proceedings of Graphics Interface 2012, GI 2012, pp. 117–120. Canadian Information Processing Society, Toronto (2012)
4. Brand, C., Gorning, M., Kaiser, T., Pasch, J., Wenz, M.: Development of High-Quality Graphical Model Editors. Eclipse Magazine (2011)
5. Chen, Q., Grundy, J., Hosking, J.: An e-whiteboard application to support early design-stage sketching of uml diagrams. In: Proceedings of the 2003 IEEE Conference on Human-Centric Computing, pp. 219–226. IEEE CS Press (2003)
6. Damm, C.H., Hansen, K.M., Thomsen, M.: Tool support for cooperative object-oriented design: Gesture based modelling on an electronic whiteboard. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI 2000, pp. 518–525. ACM, New York (2000)
7. Fuhrmann, H.A.L.: On the Pragmatics of Graphical Modeling. Kiel Computer Science series. Books on Demand (2011)
8. Hart, S.G., Staveland, L.E.: Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. Human Mental Workload 1(3), 139–183 (1988)
9. Khandkar, S.H., Maurer, F.: A language to define multi-touch interactions. In: ACM International Conference on Interactive Tabletops and Surfaces, ITS 2010, pp. 269–270. ACM, New York (2010)
10. Levenshtein, V.: Binary Codes Capable of Correcting Deletions, Insertions and Reversals. Soviet Physics Doklady 10, 707 (1966)
11. Li, Y.: Protractor: A fast and accurate gesture recognizer. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI 2010, pp. 2169–2172. ACM, New York (2010)
12. Plimmer, B., Freeman, I.: A toolkit approach to sketched diagram recognition. In: Proceedings of the 21st British HCI Group Annual Conference on People and Computers: HCI...But Not As We Know It - Volume 1, BCS-HCI 2007, pp. 205–213. British Computer Society, Swinton (2007)
13. Rubel, D., Wren, J., Clayberg, E.: The Eclipse Graphical Editing Framework (GEF). Eclipse, Addison-Wesley (2011)
14. Rubine, D.: Specifying gestures by example. SIGGRAPH Comput. Graph. 25(4), 329–337 (1991)
15. Sangiorgi, U.B., Barbosa, S.D.J.: Sketch: Modeling using freehand drawing in eclipse graphical editors. In: Proceedings of the FlexiTools Workshop (May 2010)
16. Scharf, A.: Scribble - a framework for integrating intelligent input methods into graphical diagram editors. In: Software Engineering 2013 Workshopband (inkl. Doktorandensymposium), pp. 591–596 (February 2013)
17. Wobbrock, J.O., Wilson, A.D., Li, Y.: Gestures without libraries, toolkits or training: A $1 recognizer for user interface prototypes. In: Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology, UIST 2007, pp. 159–168. ACM, New York (2007)