Revista Romana de Interactiune Om-Calculator 9(3) 2016, 183- 216 © MatrixRom

Methods for Modelling Sketches in the Collaborative Prototyping of User Interfaces

Jorge Luis Pérez Medina

Université catholique de Louvain, Louvain School of Management Research Institute
Place des Doyens, 1 – B-1348 Louvain-la-Neuve (Belgium)
E-mail:jorge.perezmedina@uclouvain.be

Abstract. Cross-functional teams with different technical backgrounds working on cross-platform environments require the production of flexible modeling of user interfaces in early steps of a design process. We observe that model-driven engineering (MDE) is currently gaining acceptance in many domains. However, existing solutions have no support for collaborative prototyping of user interfaces by sketching recognition for multiple stakeholders (e.g., designers, developers, final end users) working with heterogeneous computing platforms (e.g., smartphones, tablets, laptops, desktop), on different, perhaps separate or shared, interaction surfaces (e.g., tables, whiteboards) in a co-located way or remotely. This requires flexibility to explore and reuse vague and uncertain ideas as model sketching. This paper presents UsiSketch, an MDE method for modeling sketches that offers the following novel features resulting from a requirement elicitation process: sketching recognition on different surfaces based on a new recognition algorithm that accommodates very large surfaces and model-based design of user interfaces with collaboration.

Keywords: Sketching, Collaborative Prototyping, Graphical User Interface, Design Tools and Techniques.

1. Introduction

Over the last years, Model-Driven Engineering (MDE) solutions and modeling platforms have been developed to simplify and automate many steps of MDE processes (Eisenstein et al., 2001). Kent (2003) argue that MDE promotes the reuse of productive modeling artifacts produced and consumed in the development process. We observe that MDE is currently gaining acceptance in many domains. However, several modeling platforms have focused on the last stages of the design process. The HCI domain shows a clear need for MDE methodology and tools to support User Interfaces Design (UID) (Pérez-Medina et al., 2007). Considering the UI as a model, MDE is able to answer the specific needs of the HCI community in terms of domain-specific meta-models and models. Nevertheless, the HCI community has to incorporate the proposed current standards used in MDE.

In the HCI domain, Buxton (2007) describes that the design process begins with ideation. Cross-functional teams with different technical backgrounds working on cross-platform environments require the production of several propositions of user interfaces at the latest stages of the design process. Those propositions are usually explored through sketches and prototypes considered as models (Demeure et al., 2011) which could then be submitted to static analysis for further exploration (Beirekdar et at 2002). Sketching is also particularly challenging when prototyping multi-platform user interfaces for multiple contexts of use (Florins et al., 2006). Coutaz (2010) found that MDE lacks support for early stages of design where the creativity and collaborative production of modeling sketches are crucial to elicit vague and uncertain ideas in projects requiring the design of advanced UIs.

In literature review, we found many software development frameworks like Gonzalez-Pérez (2010), interactive applications and academic research supporting sketching activities. However, the solutions have no support for distributed collaboration in very large surfaces, multi-level of prototyping and the execution of the user interface produced.

We present UsiSketch, a software-hardware environment to facilitate the tabletop collaborative prototyping of model-based UIs in early steps of the design process when multiple stakeholders have only a vague goal in mind of what should be produced. We present a method that recognizes UI sketches on very large interaction surfaces. UsiSketch is an Eclipse[1] application that supports multiple computing platforms and provide support for collaboration of stakeholders and final users. Our solution addresses the gap between HCI flexible practices and productive models required for the MDE community. The rest of this paper is structured as follows: Section 2 introduces our motivations and design challenges. Section 3 presents a review of sketch recognition algorithms for shape recognition. The Model Sketching method for very large interaction surfaces is later discussed using a case study in section 4. Finally, section 5 presents our conclusion and

[1] Eclipse is an open source community for individuals and organizations who wish to collaborate on comercially-friendly open source software based on Java (https://eclipse.org/).

some future avenue to this work.

2. Motivations and design challenges

2.1 General motivations

Sketching is largely recognized as an inexpensive way of producing low-fidelity prototypes, which helps framing design problems, therefore producing better design. However, before starting to discuss sketching in UI design, the main subject of the research presented in this paper, some definitions of classical sketching and UI design need to be presented.

Firstly, we refer to sketch as described in (Johnson et al., 2009): quickly made depictions facilitating visual thinking, which may include everything from abstract doodles to roughly drawn interface. The aforementioned work restricts neither the drawing medium nor the subject matter. Secondly, our work is related to both interaction and interface designs. Interaction design is the discipline “related to design interactive products to support people in their everyday and working lives” expressed by Sharp & Preece (2007). Interfaces of interactive systems are one example of such product.

2.2 Sketching in design

When designing, people draw things in different ways, which allows them to also perceive the problem in new ways. Schon & Wiggins (1992) found that designers engage in a sort of “conversation” with their sketches in a tight cycle of drawing, understanding, and interpreting. As the findings of Goel (1995) point out, the presence of ambiguity in early stages of design broads the spectrum of solutions that are considered and tends to deliver a design of higher quality. Van der Lugt (2002) conducted an experiment to analyze the functions of sketching in design in which participants produced individual sketches and then presented them to the group for discussion. From the experiment conducted by Vander Lugt, **three primary sketching functions** were identified:

F1: Sketching stimulates a re-interpretive cycle in the individual designer's idea generation process: design as a cyclic process of sketching, interpreting and taking the sketches further.

F2: Sketching encourages the designers to reinterpret each other’s ideas: when the sketches are also discussed (as opposed to sketch for self-interpretation), the designer invites others to interpret her drawings. The function of inviting re-interpretation described by van der Lugt (2002) is especially relevant for the idea generation process, as re-interpretation leads to novel directions for generating ideas.

F3: Sketching stimulates the use of earlier ideas by enhancing their accessibility. Since it is externalized, sketching also facilitates archiving and retrieval of design information.

2.3 Sketching in user interface design

In order to support sketching into UI design, we need to analyze the process in which UI design is included. Currently, the development life cycle of interactive applications consists of a sophisticated process that does not always proceed linearly in a predefined way. The tools available for UI development do not usually focus on UI **design** in which designers usually explore different alternatives, but in UI **modeling** as a final product, where designers must abide by formal standards and notations. Many tools are available for both modeling and design. However, practitioners are currently forced to choose formal and flexible tools. Whichever they choose, they lose the advantages of the other, with attendant loss of productivity and sometimes of traceability and quality.

(Johnson et al., 2009) claim that great care must be taken to support the designer's reflection when making design software that employs sketch recognition. If the system interprets drawings too aggressively or at the wrong time, it may prevent the human designer from seeing alternative meanings; recognize too little and the software is no better than paper.

The studies of (Cherubini et al., 2007) showed that designers desire an intelligent whiteboard because it does not require hard mental operations while sketching during meetings or design sessions. Calico proposed by (Mangano et al., 2010) is a good example of “vanishing tool” as it keeps itself out of the way between the developers and the models, and this can be useful especially during early design stages. However, it is not obvious to explain why software designers resist adopting them, despite of the ubiquity and low cost of pen-based and touch devices (Cherubini et al., 2007).

2.4 Design goals for collaborative sketching

We would define Collaborative Sketching (CS) as a mix of Collaborative Design and Design by Sketching. Although CS is already defined and supported by (Geyer et al., 2010; Bastéa-Forte & Yen, 2007; David & Hammond, 2010; Hailpern et al., 2007; and Haller et al., 2010). Our goal is also to define a specific domain of CS for User Interface design.

We have observed design sessions related to user interface development conducted in two companies. The people involved on those sessions were designers, project managers, programmers and frequently stakeholders. In overall, in these companies, design sessions are usually carried out around a central topic, about which people discuss in order to produce some artifact, usually a report with a list of requirements, wireframes and some session log of the decisions made around the interaction. It is important to note that this report is not produced on site but after the meeting, for what people usually take pictures to remember and register what was discussed. Nevertheless, the design sessions most often proceeded with three distinct phases:

1. *Mental model construction and concepts*: the mediator leads the task, asking the participants the essential elements of the tasks.
2. *Scenario construction*: the participants are usually divided into groups to focus on one scenario each. They usually do it using a big sheet of paper and use post-its. After each has agreed on its own scenario, the sheets are arranged as a storyboard on a wall for discussion.
3. *Interface prototyping*: the participants sketch the UI based on what was discussed and learned on the scenarios discussion.

3. Sketch recognition algorithms for shape recognition

3.1 Recognition algorithms

Generally, the purpose of the recognition algorithm is to enable a computer to identify the shape or element shown by a hand drawing. Starting from the idea that a drawing is always subject to interpretation, Beuvens &

Vanderdonckt (2012) found that these algorithms are accurate to the order of 80% when adapted to the user. Often, the algorithms are specific to a certain scope, for instance: recognition of symbols, geometric shape recognition, recognition of signatures, etc.

We focus on studying four generic algorithms for gesture recognition. The algorithms selected are Rubine by Rubine (1991), One Dollar proposed by (Wobbrock et al., 2007), Dollar P created by (Vatavu et al., 2012), Levenshtein by (Coyette et al., 2007) and Stochastic Levenshtein proposed by Ocina & Sebban (2006). The reasons for which we use these algorithms are that we know them well and they are part of the research performed in our research team. We invite the reader of this paper to review an exhaustive comparison of these algorithms performed in Usi Gesture: an Environment for Integrating Pen-based Interaction in User Interface Development proposed by Beuvens & Vanderdonckt (2012). All details and screenshots related to this procedure are accessible at https://goo.gl/0hPnih. They are not entirely described here since it is beyond the scope of this paper. However, we present an overview of Levenshtein's algorithm because there is the starting point of our new recognition algorithm.

3.2 Levenshtein's algorithm

This algorithm is based on the edit distance between two strings as a measure of their dissimilarity. The principle behind the distance is to transform one string “A” into another string “B” using the basic character wise operations delete, insert and replace. The minimal number obtained after the transformation is called the edit distance or Levenshtein's distance (Coyette et al., 2007). The minimal number of needed edit operations for the transformation from A to B is called the smaller. Its value represents the distance between these strings.

Figure 1 shows an example of the grid quantization of freehand shape. The features to be extracted from the raw data are based on the principle described by D. Llorens & Zamora (2008). The representation of the rectangle is superposed with a grid and the freehand drawing is quantized with respect to the grid nodes. Each grid node has 8 adjacent grid nodes and for each pair of adjacent nodes one out of 8 directions can be given (i.e: 1 for North, 2 for NorthEast, 3 for East, and so on). From the sequence of successive grid nodes, a sequence of directions can be derived. This sequence can be coded using an alphabet represented by numbers from 1 to 8. Each value represents one direction. Consequently, when comparing a sequence representing the gesture to recognize with the templates present in the training set, we look for the template with the smallest edit distance.

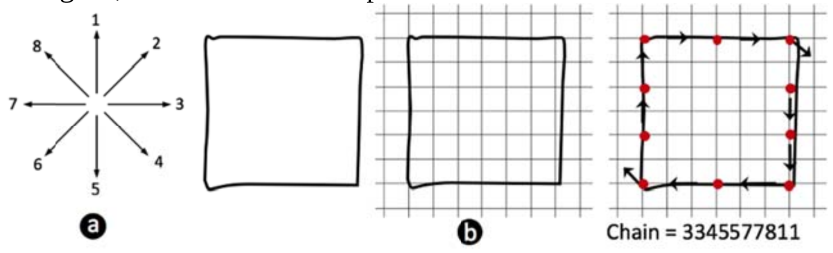


Figure 1. Example of a Square grid quantization of freehand shapes. The wind rose in (a) describe a gesture in the gesture in the form of a string. In (b), the gesture of the Rectangle. The corresponding string for this gesture in relation to the wind rose is “3345577811”.

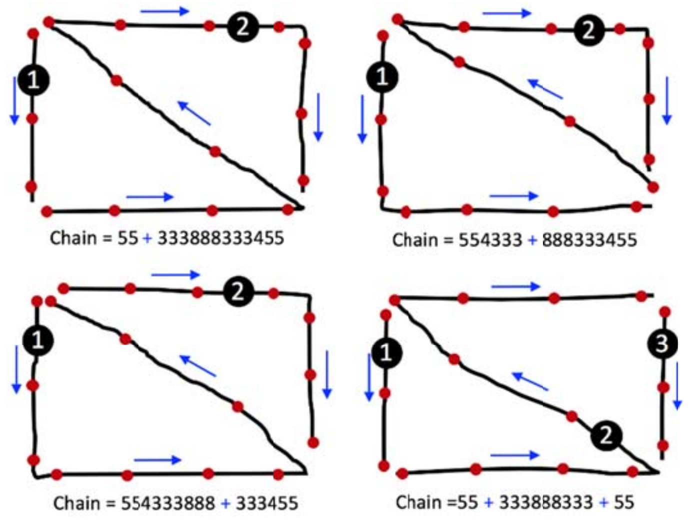
3.3 Stochastic Levenshtein's algorithm

Stochastic Levenshtein’s proposed by Ocina & Sebban (2006) is based on the number of suppressions, insertions and substitutions to transform one chain of characters into another. These operations are modeled with a fixed cost for each operation. An intuitive improvement consists in automatically learning these costs in such a way that the precision of the comparisons is increased. Levenshtein's algorithm is based on a probabilistic model to achieve its purpose. The procedure to recognize a gesture is thus similar to the classic version of the original Levenshtein's algorithm. The gestures are first transformed into sequences of characters and then compared with the gestures of the training set to find their corresponding class. The only difference is the use of a stochastic approach during the string comparison.

The main goal behind the selection of stochastic Levenshtein's algorithm is the evaluation of the performances of a statical-based algorithm. The advantage is its quite low time cost while most statical-based algorithms are very expensive. Furthermore it represents a natural improvement of the traditional Levenshtein's algorithm. The review of the recognition algorithms allowed us to understand the following principles.

3.4 Recognition of single and multiples strokes

We define a stroke gesture as the movement trajectories of user's finger or stylus contact points with a touch sensitive surface. A stroke gesture is usually encoded as a time-ordered sequence of two-dimensional (x,y) points. Optionally, stroke gestures can have time stamps as the third dimension. A single stroke corresponds to the sequence of points excluding the time stamps value (See gesture in Figure 1). As opposed to single stroke, Figure 2 shows equivalent representations of an imaginary representation in multi-strokes. Note that the imaginary gesture can be drawn using several strokes. Regarding the studied algorithms, only Levenshtein's algorithm, their variant and Dollar P support the use of multi-stroke gestures.



1
2
Chain = 55 + 333888333455
1
2
Chain = 554333 + 888333455
1
2
Chain = 554333888 + 333455
1
2
3
Chain =55 + 333888333 + 55

Figure 2. Examples of an imaginary representation in multi-stroke.

3.5 Direction invariant

Single and multi-strokes can vary in order and direction (See Figure 3). Working with multi-strokes requires manipulating a time stamp value to allow user's representations. In the case of the recognition of widgets from shapes we consider the use of a threshold value configured by the user.

The chain value for this kind of gestures is then calculated by the addition of the chain representations of each stroke and the time stamp value consumed by the user when finishing and starting a new stroke (See figure 3). The use of gestures based in multi-stroke needs to generate all possible permutations of a given representation, which generally causes an explosion in both memory and execution time. Figure 2 shows examples of the chain conversion for multi-stroke gestures.

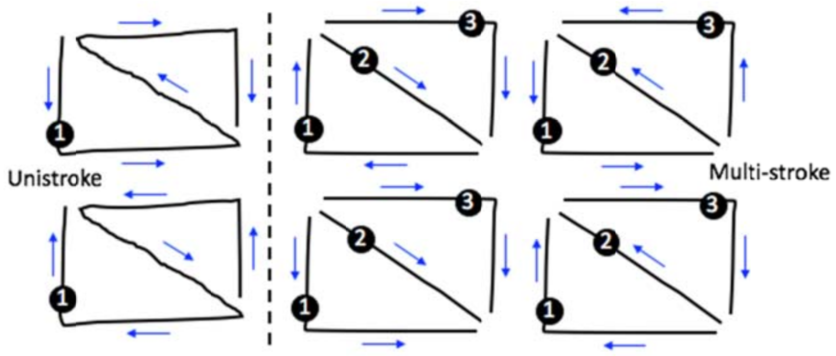


Figure 3. Illustration of the imaginary gesture by single and multi-stroke.

3.6 Scale invariant

All studied recognition algorithms are scale invariant. Meserve (1983) expresses that scale invariant is founded on a homothetic (or homothety) transformation. The basic principle consists in scaling a gesture to a reference square determined by a point "S" called its center and a nonzero number "r" called its ratio. We found that all the algorithms reviewed support this principle. Figure 4 shows an example of a gesture in different scales. Note that depending on the size of the gesture, the chain representation might be different. This difference could make it difficult to recognize gestures on very large surfaces. In the next sections we will propose a strategy that artificially enlarges one chain while preserving the general appearance of the gesture.

3.7 Rotation invariant

The principle of rotation requires that the set of points of an angle could be rotated to best align with the other. The basic principle is to find over the space of possible angles for the best alignment between two points paths. The process of rotation iteratively searches the best candidate gesture from +1° to 360° (See Figure 5). We found that One Dollar requires much more training examples to recognize the rotation gestures that are too dependent on the rotation. Dollar P does not support rotation gestures.

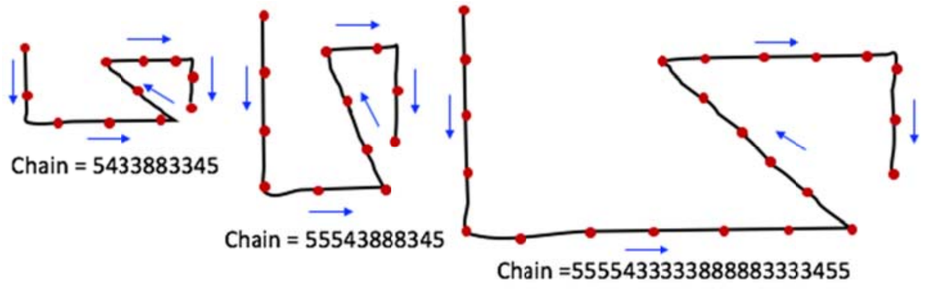


Figure 4. Illustration of similar gestures in different scales.

Table 1 summarises the comparisons of generic algorithms for gesture recognition in function of the presented principles. Existing software incorporating an object recognition typically only supports one single representation per object, frequently through a mono-directional single-strike gesture.

We are interested in recognizing several representations for a single object, without significantly affecting the performance. In addition, each representation could be sketched in a multi-stroke way that is independent of the direction of gesture. In this way, left-handed or right-handed persons are equally supported.

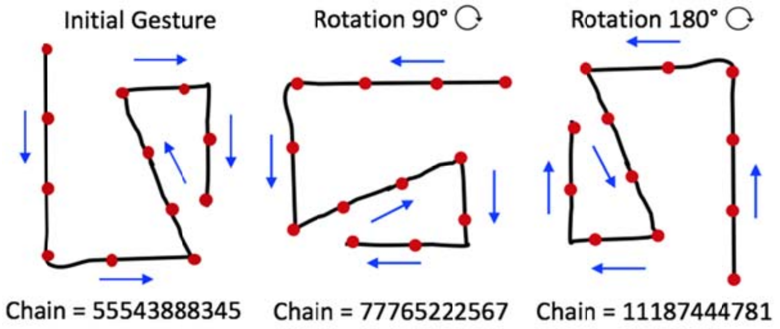


Figure 5. Rotation in 90° and 180° of an imaginary gesture.

Table 1. Classification of algorithms for gesture recognition

Principle	Levenshtein	Stoc. Levenshtein	Rubine	$1	$P
Single stroke	√	√	√	√	√
Multiple stroke	√	√	χ	χ	√
Direction invariant	√	√	√	√	√
Scale invariant	χ	χ	√	√	√
Rotation invariant	√	√	√	√	χ

4. The method for modeling sketches on very large interaction surfaces

This section shows the method for model sketching for very large surfaces. The state-of-art for gesture recognition, such as Rubine proposed by Rubine (1991), One Dollar created by (Wobbrock et al., 2007) and Dollar P by (Vatavu et al., 2012) propose low-cost solutions, easy to understand, implement, and offer high performances. However, these approaches have limitations. For instance:

1. One Dollar only handles single stroke gestures and the algorithm provides tolerance to gesture variation. It means that the algorithm cannot distinguish gestures whose identities depend on specific orientations, aspect, ratios or locations. For example, separating circles from ovals, up-arrows from down arrows is not possible.
2. Dollar P is invariant to direction due to its point-cloud representation. It means that clockwise and counterclockwise circles cannot be identified.

We are interested in recognizing gestures without considering their direction and their size. A viable alternative to improve the recognition accuracy is to apply a pre-processing on each gesture before calling the Levenshtein's algorithm.

Before starting the description of the method for very large surfaces, a case study is presented. The aim is to take advantage of the proposed method. The case study consists in a set of sequential tasks accomplished by an executive coordinator in order to manage the members of a group for a course in the context of a Learning Management System. The interaction scenario is defined in a task tree model depicted by Figure 6. The task model shows the activities that must be carried out in the application. The first task that user needs to perform is select a group. Next, the application shows automatically the potential members. Then, the user must select a member and confirm the selection.

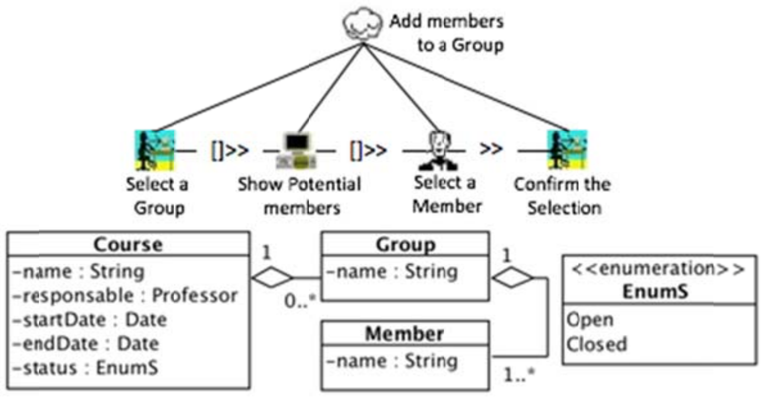


Figure 6. Task model and Class diagram of the case study.

Figure 6 shows an UML class diagram of the case study. A course can have groups. The basic information related to a training is: name, start (and end) date. Status (open/closed) and manager. The groups consist of an aggregation of members. One member can appear only in one group.

From the literature review, and especially in (Coyette et al., 2004; West et al., 2015a; West et al., 2015b) we formulated a list of 42 main requirements (30 functional requirements and 12 non-functional requirements) identified for our Modeling Sketches method. The beneficiaries of these requirements are : Designers, Testers and Developers. These requirements were classified into: 33 requirements for designers, 16 requirements for testers and 14 requirements for developers.

These requirements were then pre-validated from discussions together with two Belgian companies interested in our tool. We also grouped the requirements of several subsections. The defined categories, with the total of requirements, are: Recognition (7), Drawing-Rendering (9), Prototyping (4), Data (4), Simulation (4), Ergonomics/Usability (6), and Architecture (7).

Each requirement has a priority based on the importance level of the studied functionality. A high priority (with 18 requirements) is a vital function for the tool, a medium priority (with 19 requirements) is a useful functionality but not indispensable, finally a low priority (with 5 requirements) is a functionality regarded as an accessory.

Most of the requirements for the proper functioning of UsiSketch are undoubtedly those in categories Recognition and Drawing-Rendering. Without them, the tool could not achieve its main objective. The requirements of categories Ergonomics/usability and Architecture are essentially non-functional. For reasons of space, the presentation of all the requirements is beyond the scope of this document, but one example is shown in table 2.

Table 2. An example of one requeriment for recognition

Number: 5	Priority: Medium
Type: Non-Functional	Responsible: ZZ
Description: Composition rules (or grammar) must be specified outside the software code	
Motivation: Set new rules without touching the source code; allows great flexibility of the tool in the definition of the compositions; prevents the designer from having to adapt to the rules of composition which do not suit it	
Scenario: New widget to be defined; changing a composition rule (or grammar) clearer for the designer	
Beneficiary: Designer	
Prerequisites: Requirement 2 (the tool must be able to combine simple forms in a more complex form or widget, according to pre-established rules. Each time a new form is added to a window, the software must check whether it is possible to combine it with other forms)	

From these objectives, we describe our method for model sketching for very large interaction surfaces in the next section. The proposed method is composed of three main phases, that are: pre-production, production, and execution phases. The pre-production phase aims at defining the underlying grammar of the gestures to be made during the design session and at training the algorithms for enhancing their recognition capabilities; the production phase regard the recognition of performed gestures on a large surface and the creation of an XML file as output; the execution phase aims at executing a simulation of the designed UI.

4.1 The pre-production phase

Figure 7 shows a UML activity diagram of the pre-production phase. The block of pre-production allows defining the grammar and some classification of sketches. The "Pre-production phase" is required to configure the application for working with a specific set of widgets. Pre-production also requires a training step.

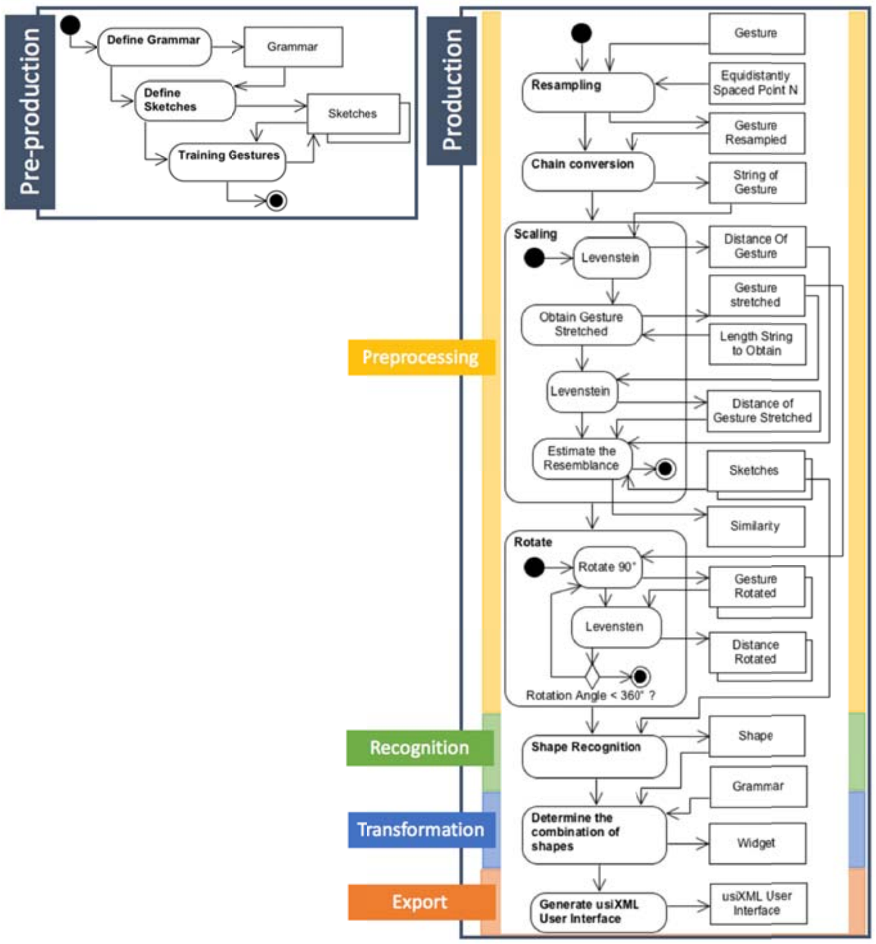


Figure 7. Pre-production and production phases for Modeling Sketches.

4.1.1 Define grammar

The contextual grammars are used to define forms as a composition of simpler shapes. These forms may be pre-drawn or drawn and recognized by a pattern recognition technique. Each complex form is accompanied by a grammar, itself composed of forms and constraints (the most typical being "<form1> in <form2>"). If all the constraints of grammar are respected, the complex form is recognized.

Figure 8(a) shows an excerpt of the listBox representation. In our case study, a component of type Listbox is used as a catalog of existing groups, as well as a catalog of all potential members to be included/deleted from a selected group. This widget is generally represented by a rectangle with two triangles (the first at the top right, the second at the bottom right), and a horizontal line inside the rectangle representing their elements. The selection list is then translated as a contextual grammar described by (Caetano et al., 2002) and is presented in Table 3.

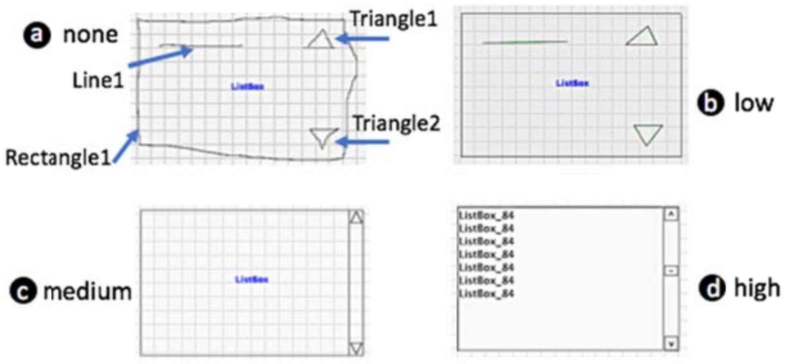
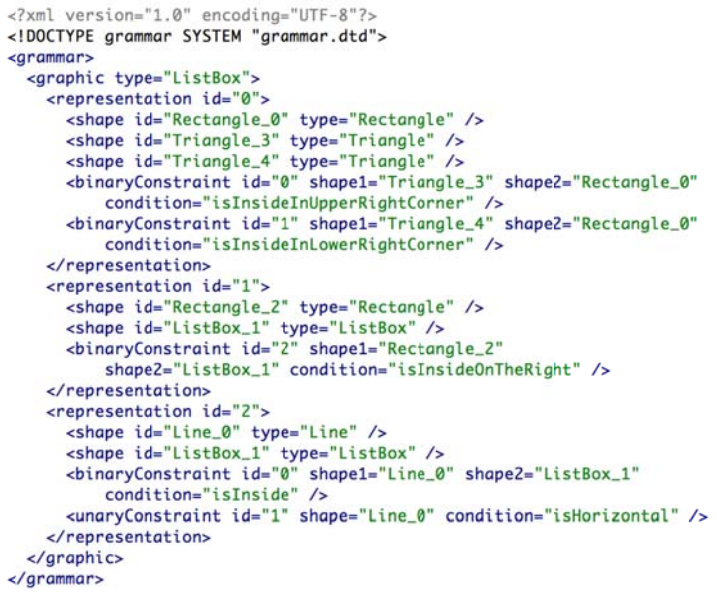


Figure 8. Representation of the ListBox Component.

Table 3. Contextual grammar for the ListBox representation

#	Shape	Description of Constraint
1	Rectangle1	-
2	Line1	Line1 is horizontal
3	Line1	Line1 in the Rectangle1
4	Triangle1	Triangle1 in the Rectangle1 at the top right
5	Triangle1	Triangle2 in the Rectangle1 at the bottom right

Contextual grammar will be used in the transformation phase to combine multiple recognizer shapes in a more complex representation. We define three types of constraints: unitary, binary and global. Unitary constraint tests a specific attribute on one form at a time. For example, the constraint *IsHorizontalConstraint(f)* tests whether a line denoted as f (and only one line) is considered horizontal. Binary constraint tests if a specific relation between two forms is verified. For example, the constraint *isInsideContraint(f1, f2)* tests if the form f1 is inside the form f2. Global constraint tests a more complex relationship, involving an indefinite number of forms. For example, the constraint *ClosedLoopConstraint(l1, .. ,ln)* tests whether a group of lines forms a closed loop. Figure 9 shows the grammar representation for the *ListBox* in an XML format.



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE grammar SYSTEM "grammar.dtd">
<grammar>
  <graphic type="ListBox">
    <representation id="0">
      <shape id="Rectangle_0" type="Rectangle" />
      <shape id="Triangle_3" type="Triangle" />
      <shape id="Triangle_4" type="Triangle" />
      <binaryConstraint id="0" shape1="Triangle_3" shape2="Rectangle_0"
          condition="isInsideInUpperRightCorner" />
      <binaryConstraint id="1" shape1="Triangle_4" shape2="Rectangle_0"
          condition="isInsideInLowerRightCorner" />
    </representation>
    <representation id="1">
      <shape id="Rectangle_2" type="Rectangle" />
      <shape id="ListBox_1" type="ListBox" />
      <binaryConstraint id="2" shape1="Rectangle_2"
           shape2="ListBox_1" condition="isInsideOnTheRight" />
    </representation>
    <representation id="2">
      <shape id="Line_0" type="Line" />
      <shape id="ListBox_1" type="ListBox" />
      <binaryConstraint id="0" shape1="Line_0" shape2="ListBox_1"
          condition="isInside" />
      <unaryConstraint id="1" shape="Line_0" condition="isHorizontal" />
    </representation>
  </graphic>
</grammar>
```

Figure 9. The XML grammar representation for the ListBox.

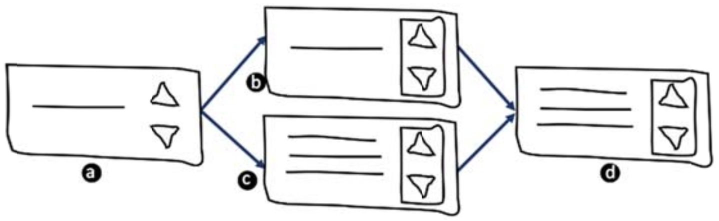


Figure 10. A hierarchical representation for the ListBox component.

Our grammar specification supports multiple representations of a specific widget. Figure 10 shows a hierarchical representation of the ListBox component. On the hierarchical representation many representations can be combined to define new representations. At the top (Part a of figure) of the hierarchical representation the *ListBox* component is represented by a rectangle, two triangles and a line. Next, two additional representations (Parts b and c of the figure) have been created to specialize the original representation of *ListBox*. Finally, these representations can be combined into a more complex representation of *ListBox*.

4.1.2 Define sketches

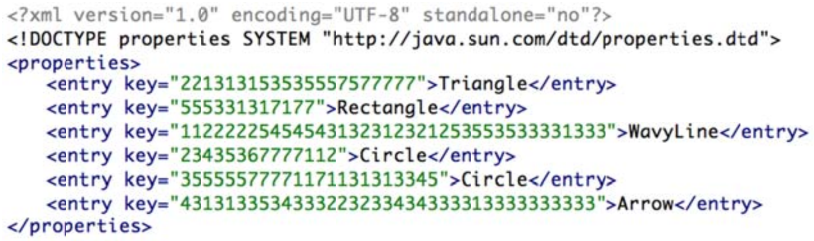
UsiSketch requires a training phase in which every gesture can be classified according to a basic geometric shape. To facilitate this work, the define sketches activity aims to provide an initial list of basic geometric shapes. Figure 11 shows a set of initial shapes. The list will grow as the user works with the tool.

4.2 The production phase

The “Production phase” (see Figure 7) started in the capture of the gestures performed by the user to recognize and translate them in widgets. The activities in the “Production phase” are classified into four groups: pre-processing, recognition, transformation and execution. Each of these activities is explained in more details in the next sections.

4.2.1 The pre-processing block

A viable alternative to improve the recognition accuracy is to apply a pre-processing on each gesture. Figure 7 shows the activities of the pre-processing block. These activities are: Resampling, Chain conversion, Scaling, and Rotate.



```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <entry key="2213131535355557577777">Triangle</entry>
    <entry key="555331317177">Rectangle</entry>
    <entry key="112222254545431323123212535553533331333">WavyLine</entry>
    <entry key="23435367777112">Circle</entry>
    <entry key="35555577771171131313345">Circle</entry>
    <entry key="4313133534333223233434333313333333333">Arrow</entry>
</properties>
```

Figure 11. Sketches representations made with the tool.

The Resampling activity is performed according to the studies proposed by (Wobbrock et al., 2007; Beuvens & Vanderdonckt (2012). But comparing two gestures point by point in their initial form is not relevant. For two gestures which are similar, the speed of the user mixed to the hardware sensitivity as well as software can ensure that the set of points (M) of appearance are frequently different. Figure 12 illustrates an example of this situation.

We realize that the first two sets of points define the same gesture despite a different frequency of point occurrence. A resampling of gesture solves this problem. In contrast to (Vatavu et al., 2012) we do not define in advance the number of points to get, but the distance between each of them. This distance is defined by N equidistantly spaced points (see Figure 13) as proposed (Wobbrock et al., 2007). Indeed, this keeps a resemblance between the re-sampled shape and the initial, whatever the length of the gesture. Figure 13 illustrates the process.

To resample a gesture, we need to obtain the length increment (I) between the set of M points. For this operation, we calculate the total of M points of the gesture dividing this length by (N-1). Iterating again over the set of points, we get the new gesture resampled.

The Chain conversion activity. Figure 14 illustrates the Chain conversion activity. Levenshtein's algorithm runs based on the relative direction between two points. Therefore, the multi-features are natively manageable: it only requires considering a gesture as a temporal sequence of points.

The last point of the trait and the first of the following are placed end-to-end and the chain conversion can operate. However, this chain does not properly represent the real gesture. As shown in Figure 16, if we consider only a gesture as a set of resampling points the chain produced varies very little. In the example of this figure, the first gesture is a rectangle performed in two lines. The second is a single broken line. The returned chains are very similar: only the “8” in red is added to the multi-gesture features. The Levenshtein's distance between two chains is 1: simply add/remove “8” to switch from one chain to another. However, these forms are not alike.

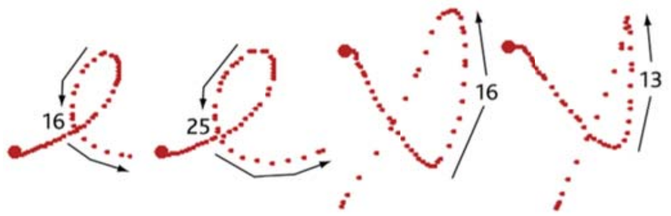


Figure 12. Example of disparities between similar gestures (Wobbrock et al., 2007).

To remedy this problem, we adopt a different approach: instead of just putting two traits end-to-end, an imaginary line is drawn between the last point of a trait and the first point of the next trait. This imaginary line is the shortest distance at which the pointer must be to reach the position of the next trait. In this way, the chain conversion keeps a trace of displacement performed without the stylus touching the drawing surface. Figure 15 illustrates this kind of conversion. As we can see, the “8” is replaced by five “8” consecutively. The Levenshtein's distance then returns 5. Chains obtained in multi-lines are more realistic and easier to distinguish with this method. Figure 16 shows an example for the case study. Note that a chain is calculated for each gesture.

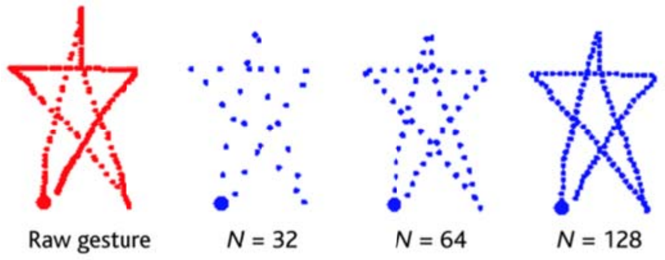


Figure 13. Example of resampling (Wobbrock et al., 2007).

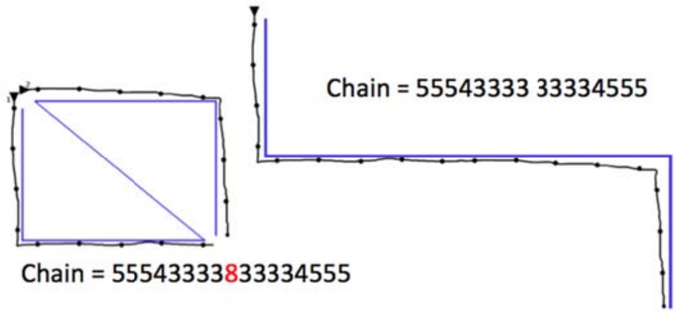


Figure 14. Illustration of ambiguity in multi-trait.

The Scaling activity. Depending on the length of the gesture, the sampling activity returns a variable number of points. Therefore, two identical shapes on a different scale will have a different number of resampled points and therefore chains with different length. In order to support the use of very large interaction surfaces we conceived a strategy based on Levenshtein's properties.

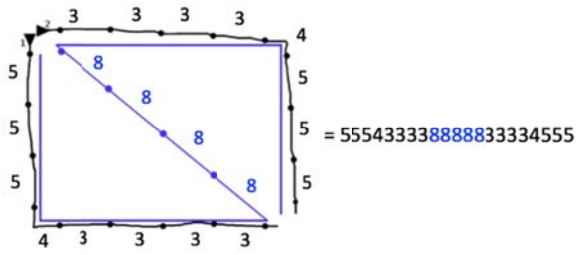


Figure 15. Illustration of the imaginary gesture in multi-stroke.

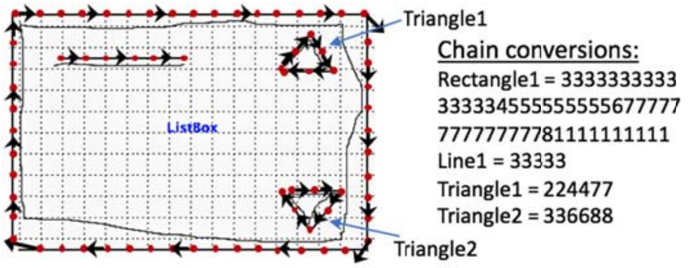


Figure 16. The chain distance for the gestures representing a ListBox component.

The Levenshtein's distance requires the lower bound on the distance between two sequences to be equal to the difference in length between these two strings:

Indeed, the shortest modification between two strings is the elimination of as many characters as the difference in their length. If after these suppressions, the strings are identical, the Levenshtein's distance is the difference.

Based on this property, when two gestures of very different length are compared, the Levenshtein's distance will always be high, regardless of the similarity of the shapes at the level near. Figure 17 illustrates this problem: both shapes are very similar, but the minimal Levenshtein's distance is (72 - 12) = 60.

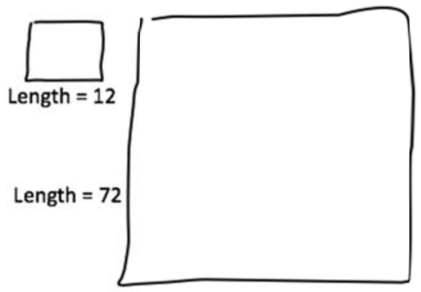


Figure 17. Illustration of the problem of scale.

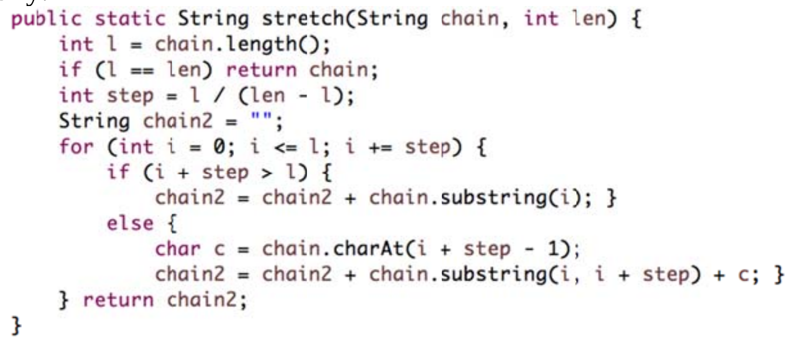
To remedy this problem, we use an approach that artificially enlarges one chain while preserving the general appearance of the gesture. Before comparing two chains, the shorter of the two is extended to the length of the other. In this way:

This means that all gestures of the training set have the same probability of being selected. To extend a chain of length l in a chain of length l', we use the algorithm described in Figure 18. In essence, it duplicates a character at all step characters in the string, until the new string has the desired length. For example, the string 111222333444, with a length of 12, extended to a length of 20 becomes 11111122222233333444444. Since this character duplication is always performed at the same frequency, the general shape of a gesture is retained. When a user to enlarge the string to 15 or 18 characters, the new characters are 111222233334444 and 111122222333344444 respectively. In these cases the shapes are not the same, in consequence the algorithm takes care of those cases and the gestures are not retained.

The chain stretching allows normalizing the similarity value between two forms. Indeed, another property of the Levenshtein's distance is the longest distance possible between two chains:

This property allows estimating the similarity between two strings of the same length l as the relation:

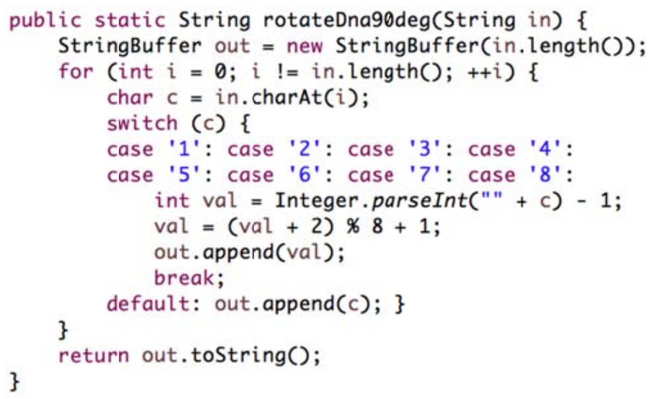
This value is used to determine the closest training gesture of the gesture to classify.



```
public static String stretch(String chain, int len) {
    int l = chain.length();
    if (l == len) return chain;
    int step = l / (len - l);
    String chain2 = "";
    for (int i = 0; i <= l; i += step) {
        if (i + step > l) {
            chain2 = chain2 + chain.substring(i); }
        else {
            char c = chain.charAt(i + step - 1);
            chain2 = chain2 + chain.substring(i, i + step) + c; }
    } return chain2;
}
```

Figure 18. Algorithm for the gesture stretching activity.

The Rotation activity. A rotation of 90°, 180° and 270° is performed on the gesture to classify and then the Levenshtein's algorithm is called for each rotation for all training gestures. This operation is done directly on the chain of character corresponding to a gesture. It reduces the number of training gestures necessary to achieve an adequate accuracy, and avoids having to remake every gesture in all directions to make it effectively recognized. Figure 19 shows the function used for the case study and Figure 20 shows the result of running the algorithm for the rotation of gestures of the ListBox component.

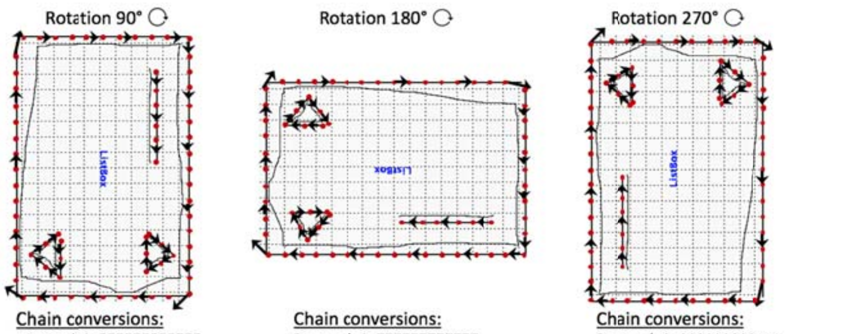


```
public static String rotateDna90deg(String in) {
    StringBuffer out = new StringBuffer(in.length());
    for (int i = 0; i != in.length(); ++i) {
        char c = in.charAt(i);
        switch (c) {
        case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8':
            int val = Integer.parseInt("" + c) - 1;
            val = (val + 2) % 8 + 1;
            out.append(val);
            break;
        default: out.append(c); }
    }
    return out.toString();
}
```

Figure 19. Algorithm for the rotation activity.

4.2.2 The recognition block

Once made the pre-processing block of each every gesture, the next activity is to recognize every gesture in a geometric shape. This section describes two categories of shape recognition used by the Modeling Sketches Method.



Chain conversions:
Rectangle1: 555555555555 5556777777777781111111111 111123333333333 Line1: 55555 Triangle1: 446611 Triangle2: 558822

Chain conversions:
Rectangle1: 777777777777 7778111111111123333333333 333345555555555 Line1: 77777 Triangle1: 668833 Triangle2: 772244

Chain conversions:
Rectangle1: 111111111111 1112333333333345555555555 555567777777777 Line1: 11111 Triangle1: 882255 Triangle2: 114466

Figure 20. Rotation of gestures of the ListBox component.

The *Shape Recognition activity*. UsiSketch has the ability to recognize and translate manuscript gestures towards geometric shapes. Next, these geometric shapes are translated into widgets through a combination of shapes.

Beuvens & Vanderdonckt (2012) define a manuscript gesture as a suite of one or several traits temporalized. It allows defining gestures as movements. In the category of manuscript gestures, it is possible to make distinctions between different types of symbols, for instance, letters and numbers: alphabet letters to uppercase or lowercase, with or without accents. We consider two categories of shape recognition:

1. An online recognition, direct and intuitive, which takes place during the construction of the image. The online recognition requires one coherent result at any time. In addition, the algorithm must be fast enough to ensure user satisfaction during the design, and therefore be sensitive from a computational point of view.
2. The offline recognition which is done only when the user requests it explicitly, usually takes place at the end of the construction of the image. The online recognition is less restricted from a computational point of view. However, this recognition is inherently non-deterministic, the result may be wrong, but users will be able to correct their result. It is nevertheless useful when designers need to import a low fidelity prototype image.

Figure 21 part (a) shows all gestures accomplished to design the user interface for the case study. Note that in part (b) the recognition of all gestures in geometric shapes.

4.2.3 The transformation block

UsiSketh must be able to combine multiple simple shapes in a more complex form or widget, according to pre-established rules. This section describes how the transformation activity is implemented.

The combination of shapes. Once the shapes are recognized, we must also define a way to bring them to widgets. The complexity of defining a widget lies in the fact that a shape can represent different types of widgets. For example, a rectangle there is a button, a text field, container, etc.? In addition, certain shapes (in particular a triangle, a rectangle, cross) can be seen as a combination of lines allowing this type of combination an interesting feature to implement.

To resolve this ambiguity, we have used the concept of contextual grammars proposed by (Caetano et al., 2002) and described in previous sections. For each widget, the grammars are tested on all possible combinations of shapes. An implementation of this technique leads us to solve an NP-hard problem.

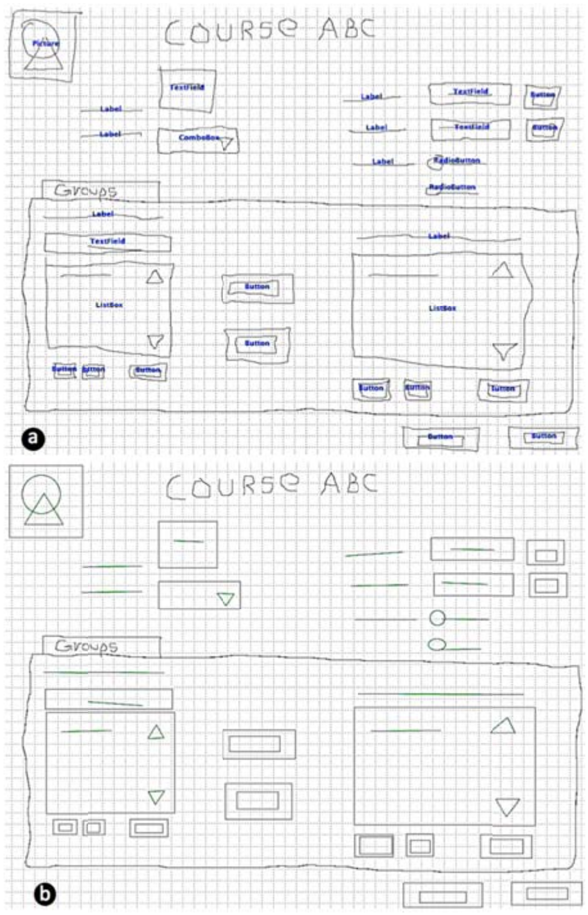


Figure 21. All gestures accomplished to design the user interface for the case study.

The number of combinations to be tested for a grammar consisting only in forms of a same type is:

The element “n” is the number of identical shapes of a certain type in a window, and “k” is the number of forms of this type specified in the grammar. If several types are present in a grammar, the number of combinations is the product of the combinations for each type. For example, the grammar of the ListBox of the Figure 21 part (b) is composed of 1 rectangle, 2 triangles and 1 line. On a windows composed of 32 rectangles, 6 triangles and 15 lines the number of cases to be tested is: 7.200 combinations.

$$\frac{32!}{1!\,(32-1)!}\,x\,\frac{36!}{2!\,(6-2)!}\,x\,\frac{15!}{1!\,(15-1)!} = 7.200 \qquad (6)$$

To make this technique more efficient, we use a constraint programming engine proposed by Haralick & Elliot (1979). Each grammar is defined as follows: each form used is a variable whose initial domain is the set of the forms contained in the windows being analyzed. These areas are reduced progressively until they contain only a member; the variable is then considered to be linked. Once all the variables (i.e. all the forms involved in the grammar) are linked, the grammar is tested. If the grammar is valid, the solution is stored in a list of acceptable solutions. The engine then continues searching in linking variables, otherwise, until all possible solutions have been tested.

The advantage of using a constraint programming engine is its ability to prune the search tree by constraint propagation: each constraint is treated independently, and if certain forms do not respect a constraint they may be removed.

Types of Constraints. In the previous sections, we described three types of constraints: unitary, binary and global. Table 4 summarizes the constraints available for the combination of shapes. In addition to these descriptions, some constraints are perfectly Boolean, others are not. These constraints return a number between 0 and 1, where 1 means “perfect match” and 0 “no match”. We call them fuzzy constraints. For example, the constraint “<line> is vertical" is blurred because the system cannot oblige the designer to sketch a perfectly vertical line. Therefore, the constraint allows a straight line to be “near vertical”. A threshold value is defined for each constraint to enable the validation of these constraints. So, the constraint is considered valid if the returned value is greater than the threshold value.

Table 4. Constraints for the combination of shapes

Unitary constraints	
isHorizontal	Exclusively for lines (in wavelets or not) and arrows. It tests wheter a line (and only one line) is considered horizontal
isVertical	Exclusively for lines (in wavelets or not) and arrows. It tests wheter a line (and only one line) is considered vertical
Binary constraints	
Intersects	Tests if two forms overlap
IsInside(f1, f2)	Test if f1 is contained in f2
IsInsideInLower Left- Corner(f1, f2)	Test if f1 is contained in f2, in the lower left corner
IsInsideInLower Right- Corner(f1, f2)	idem, in the lower right corner
IsInsideInUpper Left- Corner(f1, f2)	idem, in the upper left corner
sInsideInUpper Right- Corner(f1, f2)	idem, in the upper right corner
IsInsideOnTheBottom(f1, f2)	Test if f1 is contained in the lower part of f2
IsInsideOnTheTop(f1, f2)	idem, in the upper part of f2
IsInsideOnTheLeft(f1, f2)	idem, in the left side of f2
IsInsideOnTheRight(f1, f2)	idem, the right side of f2
IsOnTheRightOf(f1, f2)	Test if f1 is on the right of f2
Global constraints	
AllDifferent(f1, ..., fn)	Test if all shapes from f1 to fn are different. It is used to ensure that a shape is not selected more than once in a composition
ClosedLoopConstraint(l1, ..., ln)	Test if a group of lines from l1 to ln make a closed loop

Mutually exclusive grammars. Sometimes a grammar results to be valid on a group of forms, but there exist another combination of better quality (i.e. the value returned when testing is closer to 1). The value of these fuzzy constraints is used as a *tie-breaker* if a grammar is correct for several mutually exclusive combinations. Figure 22 illustrates this feature. At the left, eight lines handmade to represent two rectangles. Picture (b) represents one retained combination, but very blurred. Picture (c) represents a possible combination of better quality. The last picture is to be chosen.

Linear constraints are used to validate if two mutually exclusive combinations are possible based on a list of forms. In this case, the most constrainted combination should be selected. When two grammars are mutually exclusive, the one with more forms is chosen. If the number of forms is the same, the number of constraints plays the role of "*tie-breaker*".

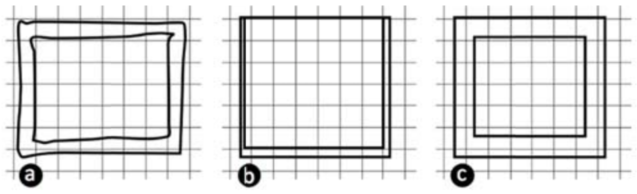
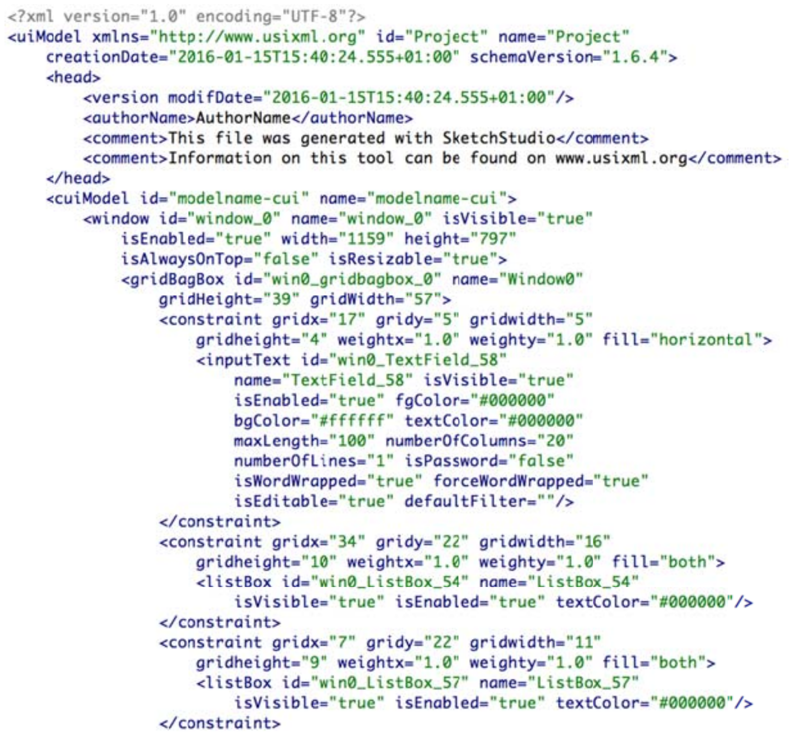


Figure 22. All gestures accomplished to design the user interface for the case study.

Let us take the representation of the *ListBox* and the set of widgets of figure 21 part (b). Initially the number of cases to be tested is 7,200 combinations, but, with the use of constraints, many shapes are removed from the initial number of combinations. We present how theses combinations are reduced:

1. The constraint (3) of the table 3 removes 9 lines (horizontal which are not in a rectangle) of domain of the line1; there remain 6 lines in these domains.
2. The constraint (4) removes 4 triangles (which are not in a rectangle at the upper right) of the domain triangle1 and removes 30 rectangles (which does not contain triangles at the top right) of the domain rectangle1; 2 triangles remain in the domain of triangle1, and 2 rectangles in the domain of rectangle1.
3. The constraint (5) removes 3 triangles (which are not in a rectangle at the bottom right) of the domain triangle2 and removes 29 rectangles (which does not contain triangles at the bottom right) of the domain rectangle1; 3 triangles remain in the domain of triangle2, and 3 rectangles in the domain of rectangle1.

After applying the restrictions, the number of shapes has been reduced to 6 lines for the domain of line1, 2 triangles for the domain of triangle1, 3 triangles for the domain of triangle2 and 3 rectangles for the domain of rectangle1. The number of combinations are therefore reduced to: 6x2x3x3 = 108 combinations. It represents only 1.5% combinations of the initial number of test combinations.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<uiModel xmlns="http://www.usixml.org" id="Project" name="Project"
    creationDate="2016-01-15T15:40:24.555+01:00" schemaVersion="1.6.4">
    <head>
        <version modifDate="2016-01-15T15:40:24.555+01:00"/>
        <authorName>AuthorName</authorName>
        <comment>This file was generated with SketchStudio</comment>
        <comment>Information on this tool can be found on www.usixml.org</comment>
    </head>
    <cuiModel id="modelname-cui" name="modelname-cui">
        <window id="window_0" name="window_0" isVisible="true"
            isEnabled="true" width="1159" height="797"
            isAlwaysOnTop="false" isResizable="true">
            <gridBagBox id="win0_gridbagbox_0" name="Window0"
                gridHeight="39" gridWidth="57">
                <constraint gridx="17" gridy="5" gridwidth="5"
                    gridheight="4" weightx="1.0" weighty="1.0" fill="horizontal">
                    <inputText id="win0_TextField_58"
                        name="TextField_58" isVisible="true"
                        isEnabled="true" fgColor="#000000"
                        bgColor="#ffffff" textColor="#000000"
                        maxLength="100" numberOfColumns="20"
                        numberOfLines="1" isPassword="false"
                        isWordWrapped="true" forceWordWrapped="true"
                        isEditable="true" defaultFilter=""/>
                </constraint>
                <constraint gridx="34" gridy="22" gridwidth="16"
                    gridheight="10" weightx="1.0" weighty="1.0" fill="both">
                    <listBox id="win0_ListBox_54" name="ListBox_54"
                        isVisible="true" isEnabled="true" textColor="#000000"/>
                </constraint>
                <constraint gridx="7" gridy="22" gridwidth="11"
                    gridheight="9" weightx="1.0" weighty="1.0" fill="both">
                    <listBox id="win0_ListBox_57" name="ListBox_57"
                        isVisible="true" isEnabled="true" textColor="#000000"/>
                </constraint>
```

Figure 23. An excerpt of the exported file for the case study.

4.2.4 The export block

This section details the export phase. UsiSketch must be able to export a project in User Interface eXtensible Markup Language (UsiXML) (UsiXML Consortium, 2007). Thanks to UsiXML, non-developers can shape the UI of any new interactive application by specifying, describing it in UsiXML, without requiring programming skills usually found in markup languages (e.g., HTML) and programming languages (e.g., Java or C++).

The designed interface must be exported under a standardized format to allow you to edit the result with a higher fidelity editor, chosen by the designer. Figure 23 shows an excerpt of the exported file in UsiXML format. Designers can use GrafiXML proposed by Michotte & Vanderdonckt (2008), a multi-target user interface builder based on

UsiXML for future modifications.

4.3 The Execution Phase

This section details the execution phase. An interface defined in UsiSketch must be simulated summarily. The activity of the design of the user interface is not isolated. During the design of the interface, the designer has the flexibility to create and modify the UI. It can also collaborate with final users by producing some sketches depicting a UI in the form of screens, roughly one for each "state" of the interface. Here the collaboration could enable the assimilation and the integration of different points of views to improve the design. The designers can also take advantage to simulate the interface without having to export it on UsiXML every time and eventually include final users for providing feedback.

UsiSketch should enable a more accurate simulation than just statically window display. Typical actions on a widget or window (show, hide, minimize, maximize, setText, etc.) and typical event for a widget (onClick, onChange, …) must be defined. Figure 24 shows a UML activity diagram of the execution phase.

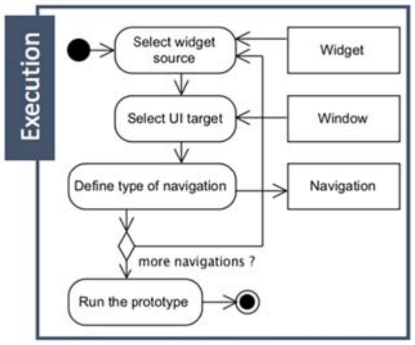


Figure 24. Execution phase for Model Sketching.

The action "*Select widget source*" allows defining the actions for a widget. The navigation is then linked to another windows. The user has the possibility to configure the navigation link. The type of configurations supported by the tool are: Open the target windows, Open the target windows and close the source windows, Open the target windows and maximize or minimize the source windows, Open the target windows in front of (or in back of) the source windows. Finally, the user can execute the prototype.

5. Conclusion

We have presented a sketching recognition method used to develop UsiSketch. Its technology runs on cross-platform and supports sketching recognition on different surfaces based.

The presented method describes a new recognition algorithm that accommodates very large surfaces. It has the ability to recognize shapes and combinations of shapes. Actually, UsiSketch recognizes and interprets 8 basic predefined shapes (i.e., triangle, rectangle, line, cross, wavy line, arrow, ellipse, and circle); 32 different types of widgets (ranging from check boxes, listboxs, textfields, buttons, video multimedia, etc.), and 6 basic commands (i.e., undo, redo, copy, paste, cut, new window).

Users of different domains can combine multiple simple shapes in a more complex combination or widget, according to pre-established rules. The recognition is done at the time of drawing, and not at the end thereof. The high fidelity elements are widgets. These are very numerous, and new ones appear frequently. Therefore, and since the recognition algorithms are based on a supervised learning, wanting to recognize these individual widgets requires a training phase of the tool for each of them. Such a process would be tedious, and will never be completed: for every new widget, we should repeat the process.

We have addressed the solution using the representation of a low fidelity widget as a composition of simple geometric shapes. For this reason, we decided to implement pattern recognition only on these geometric forms, greatly reducing the number of shapes to recognize. This reduces the time needed to train the algorithm. These forms are then combined in predefined grammars.

Currently, we are working on developing new features. We will be able to easily evaluate the feasibility of our tool by conducting user experiments. The results will be used to evaluate the performance of the tool, and obtain new research perspectives.

Acknowledgement

The authors would like to thank Olivier Bourdoux for the big contribution to the UsiSketch tool.

References

Bastéa-Forte, M., & Yen, C. (2007). Encouraging contribution to shared sketches in brainstorming meetings. In *CHI '07 Extended Abstracts on Human Factors in Computing Systems* CHI EA'07 (pp. 2267–2272). New York, NY, USA: ACM.

Beirekdar, A., Vanderdonckt, J., & Noirhomme-Fraiture, M. (2002). A Framework and a Language for Usability Automatic Evaluation of Web Sites by Static Analysis of HTML Source Code. In *Proc. of CADUI'2002*, pp. 337-348, Kluwer Academics Pub.

Beuvens, F., & Vanderdonckt, J. (2012). UsiGesture: An environment for integrating pen-based interaction in user interface development. In *Proceedings of Sixth International Conference on Research Challenges in Information Science* RCIS'2012 (pp. 1–12).

Buxton, B. (2007). Sketching User Experiences: Getting the Design Right and the Right Design. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Caetano, A., Goulart, N., Fonseca, M., Jorge, J. (2002) JavaSketchIt: Issues in Sketching the Look of User Interfaces. In *Proceedings of the 2002 AAAI Spring Symposium - Sketch Understanding* (Palo Alto, March 2002). AAAI Press (2002) 9–14.

Cherubini, M., Venolia, G., DeLine, R., & Ko, A. J. (2007). Let's go to the whiteboard: How and why software developers use drawings. In *Proc. SIGCHI Conference on Human Factors in Computing Systems* CHI'2007 (pp. 557–566), ACM.

Coutaz, J. (2010). User interface plasticity: Model driven engineering to the limit! In *Proc. of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems* EICS'2010 (pp. 1–8). New York,: ACM. doi:10.1145/1822018.1822019.

Coyette, A., Faulkner, S., Kolp, M., Limbourg, Q., & Vanderdonckt, J. (2004). Sketchixml: Towards a multi-agent design tool for sketching user interfaces based on usixml. In *Proceedings of* TAMODIA'2004 (pp. 75–82), ACM.

Coyette, A., Schimke, S., Vanderdonckt, J., & Vielhauer, C. (2007). Trainable sketch recognizer for graphical user interface design. In C. Baranauskas, P. Palanque, J. Abascal, & S. Barbosa (Eds.), *Proc. of* INTERACT 2007 (pp. 124–135). Springer.

D. Llorens, et al. (2008). The UjiPenchars database: a pen-based database of isolated handwritten characters. In *Proceedings of the Sixth International Conference on Language Resources and Evaluation* LREC'2008. Marrakech, Morocco. Http://www.lrec-conf.org/proceedings/lrec2008/.

David, J., Eoff, B., and Hammond, T. (2010). CoSke-An Exploration in Collaborative Sketching. In *Proceedings of the ACM conference on Computer supported cooperative work* CSCW'2010, (pp. 471–472). New York, NY, USA: ACM.

Demeure, A., Masson, D., & Calvary, G. (2011). Graphs of Models for Exploring Design Spaces in the engineering of Human Computer Interaction. In *Proc. of ACM* IUI'2011 (p. 5p.). Palo Alto, CA, United States.

Eisenstein, J., Vanderdonckt, J., & Puerta, A. (2001). Model-Based User-Interface Development Techniques for Mobile Computing. In *Proc. of 5th ACM Int. Conf. on Intelligent User Interfaces* IUI'2001 (pp. 69–76) , ACM.

Florins, M., Montero, F., Vanderdonckt, J., & Michotte, B. (2006). Splitting Rules for Graceful Degradation of User Interfaces. In *Proc. of 8th Int. Working Conference on Advanced Visual Interfaces* AVI'2006 (Venezia, 23-26 May 2006) (pp. 59–66). New York, NY, USA: ACM Press.

Geyer, F., Jetter, H.-C., Pfeil, U., & Reiterer, H. (2010). Collaborative sketching with distributed displays and multimodal interfaces. In *Proceedings of ACM International Conference on Interactive Tabletops and Surfaces* ITS'2010 (pp. 259– 260). New York, NY, USA: ACM. doi:10.1145/1936652.1936705.

Goel, V. (1995). *Sketches of Thought*. Cambridge, MA: MIT Press.

Gonzalez-Pérez, C. (2010). Filling the voids - from requirements to deployment with open/metis. In ICSOFT 2010 - *Proceedings of the Fifth International Conference on Software and Data Technologies*, Volume 1, Athens, Greece, July 22-24, 2010 (p. 19).

Hailpern, J., Hinterbichler, E., Leppert, C., Cook, D., & Bailey, B. P. (2007). Team storm: Demonstrating an interaction model for working with mul- tiple ideas during creative group work. In *Proceedings of the 6th ACM SIGCHI Conference on Creativity &Amp; Cognition* C&C'2007 (pp. 193–202). New York, NY, USA: ACM.

Haller, M., Leitner, J., Seifried, T., Wallace, J. R., Scott, S. D., Richter, C., Brandl, P., Gokcezade, A., & Hunter, S. (2010). The nice discussion room: Integrating paper and digital media to support co-located group meetings. In *Proceedings of the ACM Conference on Human Factors in Computing Systems* CHI'2010 (pp. 609–618), ACM.

Haralick, R. M., & Elliott, G. L. (1979). Increasing tree search efficiency for constraint satisfaction problems. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence - Volume 1* IJCAI'1979 (pp. 356–364). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Johnson, G., Gross, M. D., Hong, J., & Yi-Luen Do, E. (2009). Computational support for sketching in design: a review. *Foundations and Trends in Human-Computer Interaction*, 2, 1–93.

Kent, S. (2002). Model driven engineering. In M. Butler, L. Petre, & K. Sere (Eds.), *Integrated Formal Methods* (pp. 286–298). Springer Berlin Heidelberg volume 2335 of *Lecture Notes in Computer Science*. doi:10.1007/3-540-47884-1_16.

van der Lugt, R. (2002). Functions of sketching in design idea generation meetings. In *Proceedings of the 4th Conference on Creativity & Cognition* C&C'2002 (pp. 72–79). New York, NY, USA: ACM.

Mangano, N., Baker, A., Dempsey, M., Navarro, E., & van der Hoek, A. (2010). Software design sketching with calico. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* ASE’2010 (pp. 23–32). New York, NY, USA: ACM.

Meserve, B. E. (1983). *Fundamental concepts of geometry*. Courier Corporation.

Michotte, B., & Vanderdonckt, J. (2008). GrafiXML, a multi-target user interface builder based on usixml. In *Proceedings of Fourth International Conference on Autonomic and Autonomous Systems* ICAS’2008 (pp. 15–22). Piscataway, USA: IEEE Press. doi:10.1109/ICAS. 2008.29.

Oncina, J., & Sebban, M. (2006). Learning stochastic edit distance: Application in handwritten character recognition. *Pattern Recognition*, 39 , 1575–1587. doi:http://dx.doi.org/ 10.1016/j.patcog.2006.03.011.

Pérez-Medina, J.-L., Dupuy-Chessa, S., & Front, A. (2007). A survey of model driven engineering tools for user interface design. In M. Winckler, H. Johnson, & P. Palanque (Eds.), *Task Models and Diagrams for User Interface Design* (pp. 84–97). Springer Berlin Heidelberg volume 4849 of *Lecture Notes in Computer Science*. doi:10.1007/978-3-540-77222-4_8.

Rubine, D. (1991). Specifying gestures by example. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques* SIGGRAPH ’91 (pp. 329–337). New York, NY, USA: ACM..

Schon, D. A., & Wiggins, G. (1992). Kinds of seeing and their functions in designing. *Design Studies*, 13 , 135 – 156. doi:http: //dx.doi.org/10.1016/0142-694X(92)90 268-F.

Sharp, Y., H. Rogers, & Preece, J. (2007). *Interaction Design: Beyond HCI*. New York, USA: John Wiley and Sons.

UsiXML Consortium. UsiXML, a General Purpose XML Compliant user Interface Description Language, *UsiXML V1.8*, 23 February 2007. Available on line: http://www.usixml.org.

Vatavu, R.-D., Anthony, L., & Wobbrock, J. O. (2012). Gestures as point clouds: A $p recognizer for user interface prototypes. In *Proceedings of the 14th ACM International Conference on Multimodal Interaction* ICMI ’12 (pp. 273–280), ACM.

Wobbrock, J. O., Wilson, A. D., & Li, Y. (2007). Gestures without libraries, toolkits or training: A $1 recognizer for user interface proto- types. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology* UIST’2007 (pp. 159–168). New York, NY, USA: ACM. doi:10.1145/1294211.1294238.

Wuest, D., Seyff, N., & Glinz, M. (2015a). Flexisketch team: Collaborative sketching and notation creation on the fly. In *IEEE/ACM 37th IEEE International Conference on Software Engineering* ICS’2015 (pp. 685–688). volume 2.

Wuest, D., Seyff, N., & Glinz, M. (2015b). Sketching and notation creation with flexisketch team: Evaluating a new means for collaborative requirements elicitation. In *Proceedings of IEEE 23rd International Conference on Requirements Engineering* RE’2015 (pp. 186–195).