A New Approach to Flexible, Trainingless Sketching

Florian Brieler, Mark Minas
Universität der Bundeswehr München, Germany
Department of Computer Science
{Florian.Brieler|Mark.Minas}@unibw.de

Abstract

Traditional diagram editors require the user to select the different available shapes from a list before placing them on the drawing area. In contrast, sketching allows the user to really draw diagrams like with pen and paper. The actual shapes are later recognized by software. This means an increased flexibility and simplicity in graphical user interfaces.

We present a new approach towards sketching. Goal is to have a powerful recognition engine that does not imply any restrictions on the user, and which is not bound to a specific drawing style or training. Although we design our approach to be quite flexible, our special interest lies in visual editors with diagram analysis and interpretation. As these are able to check the syntax and provide semantics of a diagram, the recognized diagram can be compared against these to see if the recognition is correct and meaningful.

1. Introduction

Consider a meeting of several designers that cooperatively create, edit and, to some extent, execute diagrams, e.g., when modeling during the early stages of the design process. There exist powerful applications and environments that greatly support the designers in drawing the diagrams, checking for correct syntax and semantics, and managing projects containing different diagrams. However, the user interfaces of such programs mostly impose an artificial usage. Drawing a simple arrow, for example, when modeling the flow of data in a diagram, usually requires the user to select the arrow shape from a list of available shapes, and place it on the drawing area by pointing at its start point and its end point. While this procedure has been adopted by most users, it is actually not very intuitive. Instead, simply drawing an arrow directly on the canvas, using a pointing device like a mouse or a stylus, and having the computer find out that it is an arrow, is a much more natural way. This way, the designers are freed from handling an application with a complicated interface, and can focus on the actual design. Unfortunately, these sketch-based approaches often come with some drawbacks, as the user has to train the system prior to using it, or requires a special drawing style. On the contrary, the goal of sketch-based approaches is to support the designers in their creative work; restrictions are just hindering. Our approach tackles this problem by providing a fully-fledged drawing application which supports sketching, but does not impose any restrictions on the user.

The benefits of sketching go beyond creating a diagram. It is even more helpful when it comes to editing. For example, instead of removing a whole arrow and replacing it by another one with a different representation and meaning, e.g., an aggregation in UML class diagrams, sketching allows erasing only the arrowhead and simply drawing a new one with the respective meaning.

Although other fields of application are conceivable, the driving idea behind our approach is to use the recognized information as input for *DiaGen*, a system for developing diagram editors [11]. By the use of our application, these editors are no longer restricted to traditional computer drawing as described above, but can rely on the sketching approach presented here. Additionally, DiaGen allows its editors to perform diagram analysis; the information gathered during this step may be used to improve the recognition process, as incorrectly recognized components can be identified.

The results being presented in this paper are still preliminary. Currently we assume a monochrome drawing; the background is white, drawn lines and points that form the components are black. Curved arcs, text, and filled or hatched regions are not considered by now, as the focus barely lies on recognition of sketches. The full recognition process looks as follows (its details are explained in the following sections):

1. The visual language designer defines all components of a diagram type. This has to be done only one time and in advance.
2. The user draws a new diagram or edits an existing one.
3. The bitmap image is parsed to create a graph as an internal representation of the diagram.

4. The internal representation is reworked to compensate for the impreciseness introduced by hand drawing.
5. The actual recognition of the components takes place, based on the graph representation of the sketch.
6. The recognized components are passed to DiaGen, which performs the diagram analysis.

Section 2 describes the design goals we have in mind for our approach, apart from the field of application outlined in this section. Section 3 classifies the above mentioned impreciseness of hand drawn sketches. The classification is necessary to identify and repair these deficiencies in the internal representation. Section 4 explains how the input bitmap is transformed into this representation (cf. step 3). Compensation for impreciseness is described in section 5 (cf. step 4). Finally, definition and recognition of the components in the internal model is focus of section 6 (cf. steps 1 and 5). Passing the found components to DiaGen has not yet been considered (cf. step 6). Related work is presented in section 7, along with a comparison to our approach. Section 8 gives prospect to further research we will do, and a conclusion of this paper.

2. Design Goals

We state ease of use as one goal of our approach, which is, to a certain degree, intrinsic to the application domain. Considering a stylus, or even a mouse (although the former is much better suited for drawing diagrams), there is no need for a complex usage. This also includes the personal drawing style of a user. If the user needs one or ten strokes to draw a rectangle, starts at the upper-left corner, or at the lower right, intersects the sides of the rectangle, draws rounded corners, or draws the different strokes in any order; all of these examples are just a matter of personal style and must not influence the recognized result. The only common knowledge the system shares with the user is the type of diagram, which implies the available diagram components (rectangles, arrows, etc.). As mentioned in section 1, the intention is to have the user simply draw on the canvas. The actual user interface has only to go as far as to offer activities that cannot be done with traditional pen and paper, e.g., saving and loading diagrams, or starting the analysis.

According to the intended setting of collaborative design, we abdicate training, because it is quite hindering. Training means that the user has to draw each available component several times prior to using the system for the actual task of drawing diagrams. This is not only time-consuming, but sometimes comes with another severe drawback; components have to be drawn the same way each time. For example, if a user drew a rectangle during the training phase always in a single stroke, starting at the upper left corner, he has do it exactly the same way each time when he later wants to draw a rectangle. Consider a handful of designers that sit around a display that supports input via a stylus. If the system has to be trained prior to using it, who should train it? If all of the designers had to do it, this would certainly contradict the idea of training, which is to recognize components more precisely as the system get accustomed to a single drawer; besides, it would mean a lot of overhead. On the other hand, if only one of the designers trains the system, he is the only one who can possibly use it, because the others probably draw a little bit different. Either their drawings are not correctly recognized, or they are recognized, which abandons the need for training. Furthermore, we are convinced that it is possible to build a system that exhibits high recognition rates even without training.

Additionally, as DiaGen is not limited to a special kind of diagram[1], generality in terms of which diagram components can be recognized is another goal. Although systems designed for a special field of application may lead to better results, as they can assume more properties, we do not want to limit our approach that way

Inbound to hand drawing is impreciseness. Of course, the recognition process must be reliable, and lead to meaningful results even in case of sloppy drawn diagrams. Nevertheless, resolution of this impreciseness is inherent to the recognition of hand drawn diagrams; therefore, we do not have to explicitly state it as a goal. Instead, we want to put emphasis on the resolution process itself. According to the generality of design, it must be capable of reliably recognizing different components from different types of diagrams.

3. Classification of Drawing Deficiencies

Typical for hand drawn images are deficiencies. Lines are rarely straight, corners are rarely exact, and so on. Dealing with these deficiencies is one of the main tasks of a sketching system. For example, although the sketch in Figure 1 is quite imprecise, the user certainly expects the rectangle to be correctly recognized. To ease addressing this issue, a classification of the possible deficiencies of hand drawings has to be applied.

Deficiencies are classified into *minor* and *major* deficiencies. The difference is how a deficiency is dealt with. Minor ones are corrected by appropriate algorithms. The actual deficiency cannot be seen in the result any more. On the contrary, major deficiencies are not removed, but an improvement to the deficiency is constructed and kept in parallel to the original. Background of this distinction is that a major deficiency is either a real impreciseness in the drawing, or intended by the user. The sketching program can decide this question not until the actual recognition process

1 DiaGen builds editors based on diagram language specifications

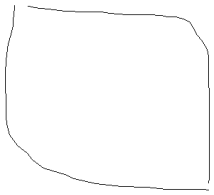


Figure 1. A quickly drawn rectangle, which shows some typical deficiencies.

(cf. section 6), as it depends on the components that are available in the respective type of diagram. A minor deficiency is always regarded as an impreciseness. Therefore, removing it is considered harmless.

3.1. Minor Deficiencies

A minor deficiency is a line that is not perfectly straight. Hand drawn lines are imprecise, and most of them are actually curved or corrugated. Another impreciseness is introduced by two or more lines that are actually meant as a single straight line, but which are not drawn in a single stroke, but in several ones. At the splices, the resulting line is not exact and can even have small corners or steps.

Additionally, *small* gaps in lines and at corners are considered as minor deficiencies, although one can argue that small is a relative term. Consequently, we define a threshold value t_{nei}. Gaps smaller than the threshold are regarded as small, while gaps wider are not.

3.2. Major Deficiencies

Because a gap can be an impreciseness no matter how wide it is, gaps that are not small (see above) are considered as a major deficiency. Of course, there is also an upper limit for non-small gaps to still be a deficiency; it is the threshold t_{spa}. Gaps wider than this value are never thought of as an impreciseness, but as the intention of the user.

The next major deficiency are "chopped corners", which can be seen at the upper right and the lower left corners in Figure 1. When quickly drawing a rectangle, for instance, many users tend to make rounded corners that look a little bit like somebody chopped off the corner.

4. From the Bitmap to an Internal Representation

Before starting to recognize the diagram components, the drawing has to be transformed into an internal representation that is suitable for the recognition process. For this purpose *attributed undirected graphs* are employed. Graphs

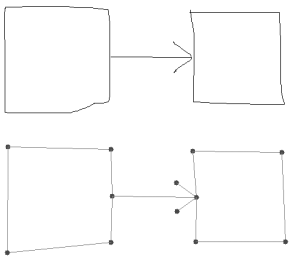


Figure 2. A simple diagram and its final representation as a graph.

are data structures that consist of a set of vertices, and a set of edges connecting these vertices. See [14] for a formal introduction to graphs.

We do not use edges that connect a vertex to itself. Additionally, we allow at most one edge between each two vertices. *Attributed* means that each vertex and edge can have several attributes assigned. The two most important attributes are coordinates x and y, which are assigned to vertices. The idea of transforming the raw image information into the graph now is to represent straight lines in the image by edges in the graph. Hence, vertices mark the end points of lines. See Figure 2 for a sketch (top) and the graph (bottom) which is to be returned by the transformation process. In figures, vertices are depicted as filled circles, while edges are the lines connecting these circles. Although the term "edge" refers to the graph, and "line" refers to the sketch, we use both terms interchangeably. The same is true for "vertex" and "point".

In the following we assume some terms and definitions:

- Vertices are often denoted by v or v_i, edges by e or e_j; since its vertices unequivocally identify an edge, often edges are denoted by their vertices, e.g., (v_1, v_2). If not stated otherwise, different identifiers always mean different vertices and edges.
- If a vertex v lies on an edge $e = (v_1, v_2)$ (which can be determined by looking at the coordinates of the vertices), this is denoted as $v \in e$.
- The *neighborhood* of a vertex v, $neighbours(v)$, is the set of all vertices v that are directly connected to v.
- The *direction* of an edge e, $dir(e)$, is defined as the angle between the edge and a horizontal line. $\forall e : 0 \leq dir(e) < 180$ holds true in every case. The direction of an edge is to used in comparison with the direction of other edges, e.g., if they are parallel, or orthogonal.
- The *distance* between two vertices v_1 and v_2, $dist(v_1, v_2)$, is defined as the Euclidean distance be-

tween v_1 and v_2; it is always equal to the *length* of an edge $e = (v_1, v_2)$, $len(e)$.

- The spatial neighborhood of a vertex v, $spatial(v)$, is the set of all vertices w where $w \in spatial(v) \Leftrightarrow (dist(v, w) \leq t_{spa} \wedge w \notin neighbours(v))$. t_{spa} is a threshold value.

We consider two possible alternatives to obtain the graph from a drawing. We could use the events generated by the input device, i.e., a list of coordinates where a button was pressed, or when the stylus touched the surface of the display. From these coordinates the edges could be directly created. However, there are some difficulties in this approach which we have not solved yet. One is the question when two edges can be connected. Another is what happens when the user moves the input device quite slow, thus generating a large set of coordinates that lie very close to each other. Therefore, we do not use the events to obtain the graph.

Instead, we rely on the bitmap representation of the drawing, which is scanned for drawn pixels (this process is also called *vectorization*). The idea is to look for drawn pixels. If such a pixel is found, a small area around the pixel is inspected for other drawn pixels. If there are some, the pixels are taken as vertices and are connected by an edge. In the end, this leads to a relatively large number of very short edges. In later steps, these short edges are merged and edited in different ways to obtain the desired representation of the graph (cf. Figure 2).

In detail, if a drawn pixel v is found, its neighborhood is inspected for other drawn pixels v_i. If such pixels are found, new edges $(v, v_i) \forall i$ are added to the graph. In contrast to the neighborhood of a vertex, the neighborhood of a pixel p consists of all other pixels whose Euclidean distance to p is not greater than t_{nei}. Obviously, the smaller t_{nei} is, the more edges are created, and the "finer" the graph will be. After a pixel is dealt with that way, the next pixel is inspected.

While this approach works fine, there have to be made several additions to obtain the desired result:

- If a new edge (v_1, v_2) is to be added to the graph, it is checked whether there is already another vertex o in the graph where $dist(v_1, o) < t_{nei}$. If so, v_1 is replaced by o. For v_2, the same rules apply accordingly. This leads to two different improvements. On the one hand, it reduces the number of vertices and edges in the graph, which accelerates later steps of processing. One the other hand, it impedes edges whose length is less than t_{nei}.
- The size of the graph can be further reduced. Consider a horizontal line in the image whose length is $6 \cdot t_{nei}$, for example. The algorithm described above leads to six edges that make up the line, each edge having a length of t_{nei} (see Figure 3). Hence, a mechanism is required that removes these six short edges and replaces them by one long edge. Let e_1 and e_2 be two connected edges, $e_1 = (a, b)$ and $e_2 = (b, c)$. If $|dir(e_1) - dir(e_2)| < t_{dir}$ holds true, e_1 and e_2 are replaced by a new edge (a, c). This process is repeated for all edges in the graph, including newly created edges like (a, c) from the example. In the end, only the desired graph remains.

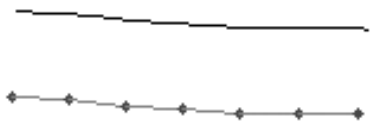


Figure 3. A line and its interim representation as a graph.

In fact, the threshold t_{dir} is measured in degree. A value of 20 turned out to lead to good results. The reason for having a threshold at all is that hand drawn lines are never perfectly straight, but always a little bit corrugated and imprecise. As mentioned above, edges whose length is below t_{nei} are never added to the graph. Now the reason for doing so becomes obvious. The shorter an edge is, the more coarse-grained its direction is, due to the fact that discrete pixels are used. For example, an extreme case is an edge whose length is 2. Its only possible directions are (in degree) 0, 45, 90, and 135. If t_{dir} is 20, the only possible case for two edges to have similar directions according to t_{dir} is for both edges to have exactly the same direction. This renders t_{dir} useless and, much worse, requires the user to draw perfect lines. A value of 5 turned out to be suitable for t_{nei}, which prevents the stated problem.

There is another problem not tackled by now. An often seen result from the algorithm described above is that corners are not real corners, but are chopped off in the graph (see Figure 4). The actual graph should appear similar to Figure 2, where the corners are clearly more precise.

Let e_1, e_2 and e_3 be three connected edges[2], $e_1 = (a, b)$, $e_2 = (b, c)$, and $e_3 = (c, d)$. e_1 and e_3 are intersected, which leads to a new vertex i (the intersection is possible if the two edges are taken as lines). Then, the three edges e_1, e_2, e_3 are removed, and (a, i) and (i, d) are added to the graph. In case that i cannot be calculated, e.g., because e_1 and e_3 are parallel, nothing happens. Additionally, e_2 has to be a short edge, whose length is below a threshold t_c, and e_1 and e_3 have to be longer than t_c. This assures that inci-

2 The current algorithm detects exactly three edges in most cases as depicted in Figure 4. More than three edges are detected extremely rarely.

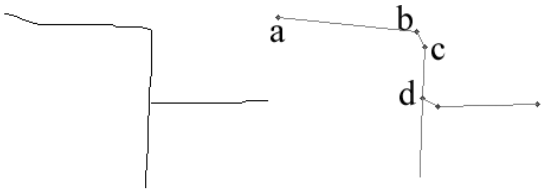


Figure 4. Some lines and their interim representation as a graph; the graphs exhibits imprecise corners.

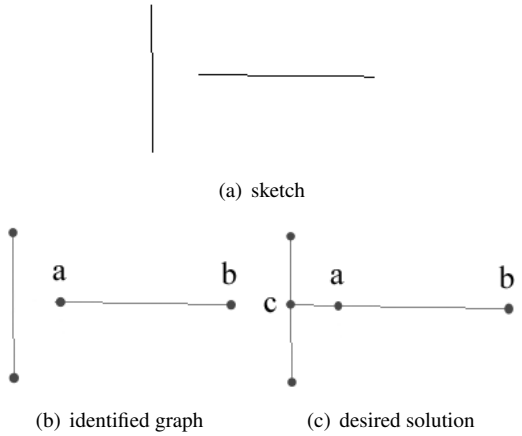
dentally chopped drawn corners are not removed, but only those ones that are incorrectly identified by the algorithm. t_c is set to $1.5 \cdot t_{nei}$ for this purpose. This mechanism also helps in case of a line starting in the middle of another line (see Figure 4).

The sum of all the steps presented above leads to the desired graph, which now is a good approximation of the actual drawing. The use of thresholds allows setting up the system to obtain best results. Minor deficiencies, as identified in section 3.1, are eliminated.

5. Dealing with Major Deficiencies

The mechanisms established in the previous section allow recognizing straight lines from hand drawn lines, and assure a proper graph based on the drawing. But this is not sufficient. Before turning to the actual recognition process, the major deficiencies of a hand drawing have to be compensated for. This requires to close gaps and deal with rounded edges, respectively edges that are not closed (cf. section 3.2). The former problem can be quite easily solved. Let (a, b) and (c, d) be two edges, $|dir(a, b) - dir(c, d)| \leq t_{dir}$ and $c \in spatial(b)$. If the missing edge (b, c) satisfies $|dir(a, b) - dir(b, c)| \leq t_{dir}$, (b, c) is added to the graph. However, the decision whether the gap is deliberate, or a drawing deficiency has to be deferred until component recognition. Hence, the new edge is flagged by setting its *original* attribute to false while it is set to true for all other edges. The recognition process, therefore, can distinguish those newly created edges from the original ones.

The other type of major deficiencies, the problem of corners (cf. section 3.2), can also be easily solved, as the spatial neighborhood can be employed again. The mechanism applied is similar to the mechanism presented in section 4, which deals with incorrect edges at corners. Again, let $e_1 = (a, b)$ and $e_2 = (c, d)$ be two edges where $c \in spatial(b)$. Let i be the vertex that is created by intersecting e_1 and e_2. Unlike before, some special cases have to be investigated. If $i \in e_1$ holds true, e_1 is divided into (a, i) and (i, b), and furthermore i is connected to the vertex of e_2 that is closer to i. On the other hand, if $i \in e_2$, the same rules apply accordingly. If $i \notin e_1 \cup e_2$, i is connected to the particular vertices of e_1 and e_2 which are closer to i.



(a) sketch

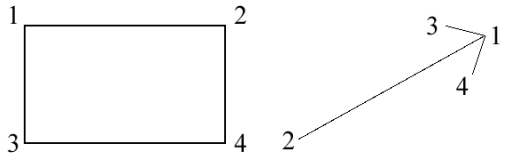
(b) identified graph (c) desired solution

Figure 5. Vertex a does not know of the vertical edge; this exacerbates finding vertex c, which is added later.

Nevertheless, there is a more severe problem with intersections, as the spatial neighborhood is not sufficient to detect all occurrences of this class of drawing deficiencies. Figure 5 gives an example of a problematic case (5(a)).

The vertical edge has no vertex in the spatial neighborhood of vertex a, as c is not known by now (5(b)). Anyway, a new edge (a, c) has to be added to the graph, which extends (a, b), satisfies $dir(a, b) = dir(a, c)$ and has c lying on the vertical edge, thus intersecting the edge (5(c)). As before, the new edge (a, c) has its 'original' attribute set to false.

The problem is to detect such an occurrence and find an appropriate vertex c. Edges are only defined by their vertices and not by any points in between. Hence, the system cannot simply recognize vertex a being near the edge (near means that $len(a, c) \leq t_{spa}$). A naive way to detect the proximity of the vertex and the edge would be to intersect *all* edges with (a, b); obviously, this is not bearable in terms of computational expenses, as it exhibits a complexity of $O(n^2)$ in time for the whole graph, where n is the number of edges in the graph. To solve this problem, the number of edges that are intersected with (a, b) must be reduced. [10] gives an algorithm based on the sweep line paradigm that solves this problem in $O((n + s) \cdot \log n)$ in time, where s is the number of actual intersections.



(a) rectangle (b) arrow

Figure 6. Two diagram components and their abstract vertex identifiers.

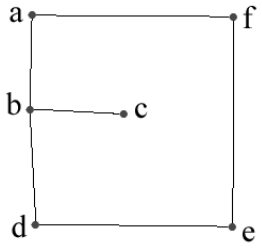


Figure 7. The graph of a user's sketch.

6. Component Recognition

As the previous two sections have shown how the raw bitmap image is converted into the internal data structure, and all drawing deficiencies are cleared, the actual recognition of the components of the diagram can be performed now. This requires defining information about the components in a suitable way, which is described in the next section.

6.1. Defining Components

A *component* is a graph (to distinguish between vertices in the graph of the sketch and components, we denote the latter by numbers). *Constraints* can be used to further specify the visual appearance of a component. So far, only constraints for angles and lengths are provided.

When specifying an edge, the user can additionally define the relation of the points to each other. Possible values are the eight 45 degree directions (up, up left, left, ...). This could be also specified by constraints, but searching can be performed more quickly if not using constraints for this purpose (see section 6.2).

For the recognition process, each component needs a designated start pointing. Furthermore, a linear ordering is imposed on the vertices, which implies a searching plan.

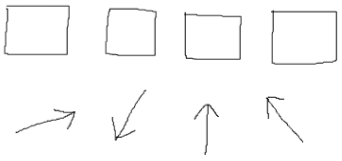
A simple example of a component is a rectangle, as it does not need any constraints. As a rectangle has four sides, we have to define four lines for the component (Figure 6(a)): $line(1, 2, right)$, $line(1, 3, down)$, $line(3, 4, right)$, and $line(2, 4, down)$. As starting point we select 1. Another example is an arrow, as depicted in Figure 6(b). The arrow has three lines; as we want the arrow to lie arbitrarily rotated on the canvas, we define the lines without giving a direction: $line(1, 2)$, $line(1, 3)$, and $line(1, 4)$. However, this is not sufficient for the arrow; we need some constraints as well. The angles between lines $(3, 1)$ and $(1, 2)$, and $(2, 1)$ and $(1, 4)$ both should be, say, 40 degrees: $angle(3, 1, 2) = 40$, and $angle(2, 1, 4) = 40$. Additionally, we want the lines $(3, 1)$ and $(1, 4)$ to be of a length of 20 pixels, and the shaft of the arrow $(2, 1)$ of a length greater than 50: $length(3, 1) = 20$, $length(1, 4) = 20$, and $length(2, 1) \geq 50$. The starting point again is 1.

6.2. Recognizing Components

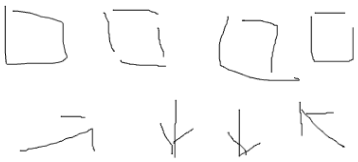
Due to the preparatory work done so far, recognizing components becomes a matter of generalized graph matching. Occurrences of graph expansions of the component definition graphs are searched in the graph that results from parsing the sketch, because a single edge in the component may be represented by several edges in the graph, e.g., when different components are touching each other. Goal of the algorithm is to find a mapping between the abstract identifiers of a component and actual vertices in the graph.

To demonstrate the mechanism, we give a simple example. Figure 7 gives the graph of a user's sketch. The goal is to find the rectangle defined in the previous section. We start by mapping vertex a to identifier 1. The first line defined for the rectangle says $line(1, 2, right)$. Luckily, there is vertex f which lies to the right of a, so f is mapped to identifier 2. The next line says $line(1, 3, down)$. Below vertex a, there are two candidates, b and d. We try b first and map it to identifier 3. The third line of the rectangle definition says $line(3, 4, right)$, and since vertex c lies to the right of b, it is mapped to identifier 4. However, the last line, $line(2, 4, down)$, cannot be found as 2 has been mapped to f and 4 to c, but neither lies f above c nor are the two of them connected. In the course of backtracking, we discard the mapping of 4 and, as there are no alternatives to vertex c, the mapping of 3. Instead, vertex d is mapped to 3 and e to 4. $line(2, 4, down)$ can be found now, because e and f are connected and f lies above e. The mapping is complete, all lines are found, and there are no constraints to check. The rectangle has been recognized.

Now the same procedure would be applied for all other vertices in the graph mapped to the starting point of the rectangle component, but, obviously, without success. In case of constraints, they can be checked after the component is recognized. Since all identifiers are mapped to concrete vertices that have coordinates, angles and lengths can be easily checked then. If all constraints are fulfilled, fine. If not



(a) clearly drawn



(b) ugly drawn

Figure 8. Examples of sketches.

so, the component is not returned as recognized, but simply discarded.

6.3. Assessing an Instance

As stated in the introduction, the recognition is done with respect to visual editors. For example, if a UML class symbol is recognized, but the diagram analysis reveals that it is syntactically wrong, the recognition process must be able to recognize a different component as well, if possible, e.g., a UML package symbol. This is done automatically by the algorithm presented in the previous section. However, recognizing all possible components raises the question of which to try first, if some if the components overlap. While we have not solved this issue yet, as a first step we introduce a measure q for the quality of a recognized component. q is calculated after all constraints have been checked. q is initialized with 1, which corresponds to a component that is perfectly drawn, and is multiplied by a factor p where $0 < p < 1$ each time the sketch is imprecise, e.g., when vertices are traversed that have their 'original' attribute set to false.

The idea is to try components with higher q value first as drawings with better quality are less likely misinterpreted by the recognition process.

6.4. Implementation

A prototype implementation of our approach is able to recognize most components correctly, when the user is anxious to draw clearly. Figure 8(a) shows some components that are identified correctly.

However, as stated as one of the goals of this project, the user should not be bound to a peculiar clear drawing style. It is rather the task of the recognition process to identify the components correctly, even if these are drawn sloppy. The ideas stated in section 5 are not fully implemented by now, so their impact cannot be analyzed yet. Without these, sleazy drawn components like in Figure 8(b) are not recognized.

7. Related Work

This section shows some fields of application for sketching and some approaches similar to the one presented in this paper.

Most other approaches rely on strokes, which we do not. *Satin* [6] is a representative of stroke-based systems, developed at UC, Berkeley. It is an extensive, high-level framework for creating pen-based applications. From the strokes drawn, *Satin* is able to recognize either graphical objects or gestures, depending on the pressed button on the pointing device. An interesting application built on top of *Satin* is *Denim* [8], an editor that supports designers in the early stages of web site creation. It fully supports the pen-based paradigm, and exposes a novel approach in user interfaces, where sketching is only one of several aspects. Costagliola et al. [5] also present an approach towards stroke based sketch recognition, which is also based on *Satin*. It is additionally capable of diagram analysis. Quite similar to what we plan to do, their application, called *SketchBench*, allows the user to design a full visual language, including the graphical symbols, and syntax and semantics checkers whereas they employ a different recognition technique which is based on grammars.

Some other approaches are restricted to certain application domains. Stahovich et al. developed a program called *SketchIT* [13]. It allows for abstraction of the qualitative behavior of a mechanical device, which is delivered by the user as a sketch. Quite similar to *SketchIt* as for analyzing mechanical diagrams, Kurtoglu and Stahovich presented a program able to translate hand-drawn sketches into respective diagrams [7]. The user draws symbol for symbol, one after another. After finishing drawing a symbol, the user presses a button to initiate recognizing that symbol. After all symbols are drawn, the program creates the diagram, and resolves all ambiguities that arose from symbols whose meaning is not self-contained. For actual recognition, the program depends on a generic approach that uses several techniques to recognize symbols from strokes, too [2]. A sketch based electronic whiteboard is described by Chen et al. [3], but it is restricted to UML diagrams, which are recognized, and then passed over to a conventional CASE tool. Text is supported by using Rubine's algorithm for recognition of gestures [12].

Penguins [4], developed by Sitt Sen Chok at Monash University, also aims at the same direction as we do, especially with respect to diagram analysis. However, the pro-

posed editor also supports traditional drawing mechanisms which we want to avoid. Free hand sketch recognition is also based on strokes.

Alvarado and Davis show another interesting stroke based approach using Bayesian networks to represent hypotheses about the user's sketch, thus finally solving ambiguities in the recognition process [1]. Their approach is not restricted to specific domains. It allows recognizing symbols as these are drawn, and compensates for drawing deficiencies.

Akin to our proposal, Mahoney and Fromherz present an approach that relies on graphs which are created from the drawing [9]. Before starting the recognition process, several mechanisms are applied to compensate for the impreciseness of a hand drawn sketch. Specifying the visual components by recognizing sketches is also considered; this would free the user from the burden of specifying the components textually, but introduces difficulties in terms of ambiguity.

8. Conclusions and Future Work

In this paper we have presented our first ideas of a flexible and general approach towards recognition of hand drawn sketches. While the results seen so far are promising, a lot of work is still to be done. First, efficiency will be improved by processing sketches as sets of line segments that are induced by the event sequence from mouse or stylus instead of processing a bitmap image. As stated in section 6.4, the recognition process has to be improved for imprecisely drawn components. We have to investigate the impact of the mechanisms from section 5. Furthermore, we are investigating how our approach can be extended by arc and text recognition. Finally, the linking with DiaGen is missing. Currently, the components recognized are simply printed on the screen, instead of being processed any further.

References

[1] C. Alvarado and R. Davis. Dynamically constructed bayes nets for multi-domain sketch understanding. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 2005.

[2] C. Calhoun, T. Stahovich, T. Kurtoglu, and L. Kara. Recognizing multi-stroke symbols. In *AAAI Technical Report SS-02-08*, pages 78–85, 2002.

[3] Q. Chen, J. Grundy, and J. Hosking. An e-whiteborad application to support early design-stage sketching of uml diagrams. In *Symposium on Human-Centric Computing Languages and Environments, Auckland, New Zealand*, October 2003.

[4] S. S. Chok. *Automatic Construction of User Interfaces for Pen-based Computers.* PhD thesis, Monash University, Victoria, Australia, 1998.

[5] G. Costagliola, V. Deufemia, G. Polese, and M. Risi. A parsing technique for sketch recognition systems. In *Symposium on Visual Languages/Human-Centric Computing, Rome, Italy*, September 2004.

[6] J. Hong and J. Landay. Satin: A toolkit for informal ink-based applications. In *The ACM Symposium on User Interfaces and Software Technology, CHI Letters, 2 (2)*, pages 63–72, 2000.

[7] T. Kurtoglu and T. Stahovich. Interpreting schematic sketches using physical reasoning. In *AAAI Spring Symposium on Sketch Understanding, AAAI Technical Report SS-02-08*, pages 78–85, 2002.

[8] J. Lin, M. Newman, J. Hong, and J. Landay. Denim: An informal tool for early stage web site design. In *Extended Abstracts of Human Factors in Computing Systems: CHI 2001*, 2001.

[9] J. V. Mahoney and M. P. Fromherz. Three main concerns in sketch recognition and an approach to addressing them. In *AAAI Spring Symposium on Sketch Understanding*, Stanford, CA, March 2002.

[10] K. Mehlhorn. *Multi-dimensional Searching and Computational Geometry*. Springer-Verlag, Berlin, 1984.

[11] M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 44(2):157–180, 2002.

[12] D. H. Rubine. *The automatic recognition of gestures*. PhD thesis, Carnegie Mellon University, 1991.

[13] T. F. Stahovich, R. Davis, and H. E. Shrobe. Generating multiple new designs from a sketch. In *AAAI-96, Vol. 2*, pages 1022–1029, 1996.

[14] M. N. S. Swamy and K. Thulasiraman. *Graphs, Networks, and Algorithms*. John Wiley & Sons, New York, 1981.