Preprint version of
F. Brieler, M. Minas, A model-based recognition engine for sketched diagrams,
Journal of Visual Language and Computing (2010), doi:10.1016/j.jvlc.2009.12.002

A Model-Based Recognition Engine for Sketched Diagrams

Florian Brieler[a], Mark Minas[a,*]

[a] *Universität der Bundeswehr München, Department of Computer Science, 85577 Neubiberg, Germany*

Abstract

Many of today's recognition approaches for hand-drawn sketches are feature-based, which is conceptually similar to the recognition of hand-written text. While very suitable for the latter (and more tasks, e.g., for entering gestures as commands), such approaches do not easily allow for clustering and segmentation of strokes, which is crucial to their recognition. This results in applications which do not feel natural but impose artificial restrictions on the user regarding how sketches and single components (shapes) are to be drawn.

This paper proposes a concept and architecture for a generic geometry-based recognizer. It is designed for the mentioned issue of clustering and segmentation. All strokes are fed into independent preprocessors called transformers that process and abstract the strokes. The result of the transformers is stored in models. Each model is responsible for a certain type of primitive, e.g., a line or an arc. The advantage of models is that different interpretations of a stroke exist in parallel, and there is no need to rate or sort these interpretations. The recognition of a component in the drawing is then decomposed into the recognition of its primitives that can be directly queried for in the models. Finally, the identified primitives are assembled to the complete component. This process is directed by an automatically computed search plan, which exhibits shape characteristics in order to ensure an efficient recognition.

In several case studies the applicability and generality of the proposed approach is shown, as very different types of components can be recognized. Furthermore, the proposed approach is part of a complete system for sketch understanding. This system not only recognizes single components, but can also understand sketched diagrams as a whole, and can resolve ambiguities by syntactical and semantical analysis. A user study was conducted to obtain recognition rates and runtime data of our recognizer.

1. Introduction

Electronic devices supporting free-hand input have become more ubiquitous. PDAs with touch-sensitive displays, for example, are widespread nowadays, and are available in different flavors like the *Apple iPhone*. Another example are notebooks from the

*Corresponding author
Email addresses: florian.brieler@unibw.de (Florian Brieler), mark.minas@unibw.de (Mark Minas)

Protégé series by *Toshiba* that can be converted to tablet computers. *Wacom* produces and delivers displays and graphic tablets capable of pen input. Many more examples exist. Despite this technical evolution, software for such devices is very basic. Often, the touch input is only used as a bare replacement for a pointing device like a mouse, neglecting its special characteristics and unique capabilities. An exception to this rule is the recognition of hand-written text, e.g., by the hand-writing recognition module from *Microsoft Windows XP Tablet PC Edition*.

The obvious advantage of sketching, understood by many authors in the field, is that it means a very natural way of interacting with a computer [1–5]. Drawing sketches with a stylus on a screen is quite similar to drawing on paper with a pen. Even more, the computer expands the possibilities offered by paper by adding convenient features like save and load, cut, copy and paste, undo and redo, and so on. Because sketching as a research topic means more than just drawing, it is necessary that sketches are understood by the machine. The application of sketching is widespread and ranges from use of informal sketches like *Teddy* [6] or *ILoveSketch* [7] to more formal approaches like *SkG* (Sketch Grammars) [8] or the work by Zanibbi et al. [9]. In this context it is important to understand that, while sketches are informal by nature, there has to be introduced some kind of formality to enable an understanding by the machine at all.

In this paper we focus on hand-drawn diagrams, which is a subset of hand-drawn sketches. Diagrams are composed of *diagram components*; both the visual appearance of the components, and the syntactic and semantic rules for a complete diagram are known. This has an important implication on the recognizer for diagrams: it can be tailored especially to the components in question; there is no need to recognize any other components.

The issue of processing hand-drawn diagrams is two-fold. First, the diagram components drawn on the canvas must be recognized; then, analysis of the identified components takes place (cf. Section 2). In our case the latter is closely related to regular diagram processing (i.e., not hand-drawn). Many matured approaches exist for this purpose, and research is still going on. Examples of such approaches are *DiaGen* [10], *DiaMeta* [11], *AToM*^{3} [12], and *Fujaba* [13]. Despite a large body of research produced so far, the recognition process for hand-drawn diagrams is still not solved satisfactorily. It can be described by the question "Given a set of strokes drawn by the user on the canvas, which diagram components (and ultimately which diagram) are represented by these strokes?"

A challenge included in this question is that of *clustering* (or grouping) and *segmentation* (or fragmentation). The former means to decide which different strokes to combine in order to form a component. Segmentation means quite the opposite, regarding whether the same stroke contributes to more than one component and should be split accordingly. Clustering and segmentation are a central point for recognition, because their result must be known in order to decide about the actually drawn components, but it can only be known for sure after the recognition process identified all of those components. This is a typical chicken-and-egg problem. The capability to cluster and segment the input strokes is an important task for a sketching system. If one or the other is not possible, this results in a severe limitation in the usability of the system, as the user has to spent more attention on *how* he draws, instead of focusing on *what* he draws. Examples for clustering and segmentation can be seen in Section 6.2.

There are many approaches to recognition, which can be arranged in several main categories. There are *image-based* approaches, like the one given by Kara [14]. The general problem here is that segmentation and clustering cannot be done at all. Kara, for example, therefore combines his approach with some other technique (*marker symbols*) to overcome this drawback.

A very large group of recognizers (regarding references in literature) are *feature-based* approaches. The work of Rubine [15] has to be mentioned here, as it is well-known and served as basis or inspiration for several similar recognizers enhancing his original concept, e.g., [16–18]. Typically, feature-based approaches can be easily implemented, and there even exist frameworks providing recognizers ready for use [19, 20]. Sketching systems can be quicker implemented using these recognizers, getting rid of one of the pitfalls of the field. Segmentation and clustering are possible but very difficult for feature-based approaches, as it is (in general) not meaningful to compute features of a stroke if you have not decided about segmentation yet, and as the value of a feature can be very different depending on the actual segmentation intended by the user.

A different type of recognition that we follow in this paper is *geometry-based*. This means that components are described in terms of the primitives they are composed of, and constraints on these primitives. Examples will be given below. Recognition in a geometry-based approach comprises two steps. The first is low-level processing, where primitives likes lines, arcs, or curves are recognized from the strokes drawn by the user. The second step is high-level processing, where these primitives are used in order to compose the complete components. Usually, low-level processing is domain-independent, while high-level processing is not. An advantage of geometry-based approaches is the clear distinction between these two steps; another is that clustering is inherently supported, as the high-level processing usually does not consider from which strokes which primitives emerged.

Examples of geometry-based approaches to sketching will be discussed in Section 3. In this paper we want to improve on these approaches. Therefore, we propose a new concept and architecture for a geometry-based recognizer that is practically free of any limitations regarding the user's drawing style, that handles clustering and segmentation reliably, and that performs very efficiently. This concept represents a matured advancement of a much earlier workshop paper [21]. We feed the user's strokes into so-called *transformers* that process and abstract the strokes (low-level processing). The result of the transformers is stored in *models*. The advantage of models is that different interpretations of a stroke may coexist in parallel; there is no need to rate or sort these interpretations. As mentioned above, the recognition of a component is then decomposed into the recognition of its primitives, which can be directly searched for in the models. This process is guided by a *search plan*, which is automatically computed based on a domain-dependent specification of the components. Furthermore, it is possible to integrate already existing low-level recognizers and related approaches into our system. While useful for image- and feature-based approaches, training our system in any way is not considered.

The remainder of this paper is structured as follows. Section 2 briefly describes *DSketch*, the sketch understanding system our proposed recognizer is integrated in. Section 3 discusses related work in comparison with our approach. Section 4 explains

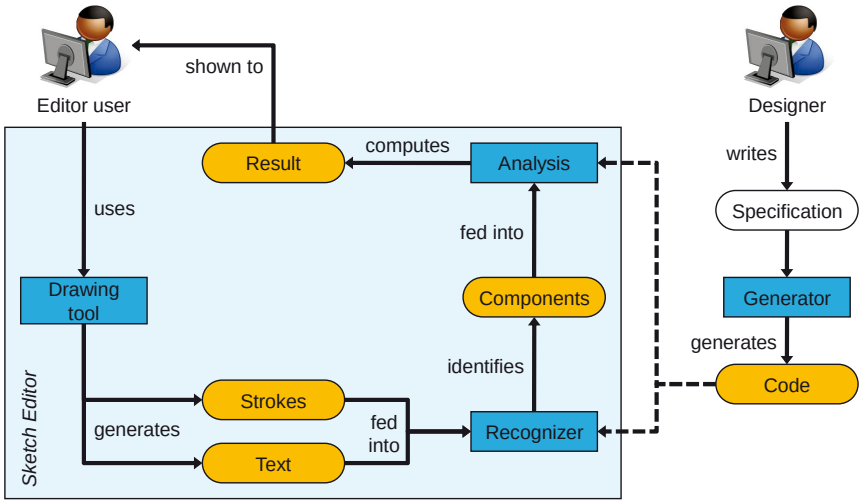

Figure 1: Architecture of our sketch understanding system *DSketch* [22].

the recognizer and how it works in detail. A comprehensive example is discussed in Section 5. Section 6 reports about an evaluation of the approach that has been conducted based on case studies and a user study. Finally, Section 7 concludes the paper and describes future work.

2. *DSketch* – a diagram sketch understanding system

In [22] we have described *DSketch*, a generic sketch understanding system that can be tailored to different types of diagram languages (although the name *DSketch* has not been used in this reference). The system comprises a framework and a generator. The generator creates source code from domain specifications, written by a language designer. This source code and the framework represent full applications (sketch editors). An overview of the architecture of our system is shown in Figure 1. Solid arrows denote control flow, rectangles denote processing units, and rounded boxes denote data. Dashed arrows denote where the generated source code is used in. The framework is not shown explicitly.

The user draws a diagram using the *drawing tool* (i.e., the graphical user interface) that generates a set of strokes by capturing the event stream generated by the stylus, and a set of text written on the canvas (text is not automatically distinguished from graphics, but the user is required to do this). The strokes and the text are then used by the *recognizer* to identify (or recognize) all represented components. This is the focus of the current paper. To solve the ambiguities, which naturally arise from the process of recognition, the set of all identified components is then *analyzed*. As the components are added to a complete diagram, syntax and semantics of this diagram can be checked. Details of this process are described in [23]. The analysis produces the final result of the processing, i.e., a syntactically and semantically correct diagram that

is finally shown to the user. Furthermore, the recognizer and the analysis are tailored by the afore-mentioned specification to account for the specific details of a diagram language. To satisfy the requirements of a usual application, the drawing tool is also equipped with editing capabilities, load and save functions and text input[1].

Having an analysis as we do has two major implications: first, ambiguities are resolved according to syntactical and semantical rules of a diagram language, and not by the use of inflexible and simple meta-rules like other approaches do. Second, the recognizer does not have to discard any recognized components; on the contrary, it is even better the more components are recognized, because false positives mean no harm, and will be dealt with by the analysis. This contrasts with other approaches where false positives must be avoided at all costs as there is no analysis.

3. Related work

Our focus on related work lies in approaches that are generic. However, there are also many domain-specific approaches, e.g., for user interface design [24], or for UML class diagrams [25].

The idea of geometry-based recognition is not new. Other approaches also follow this principle and recognize components by recognizing primitives first, and then combining these primitives to obtain the components. Among these approaches are *LADDER*, *Sketch Grammars*, *InkKit* and *SketchREAD*.

Hammond's approach, *LADDER* [26, 27], is focused on a language to describe shapes, editing behavior and related aspects. Components may be defined hierarchically, using abstract components. Regarding recognition, they use a rule-based approach. Strokes are classified as primitives by a low-level recognizer like the ones from Sezgin or Paulson (see below), and added as facts. Recognition of components is then represented as rules about the facts. As soon as a component is successfully recognized, the respective facts are removed from the rule system (as a consequence, only one of the three rectangles drawn in Figure 12 would be recognized, see Section 5). Ambiguity on the level of components is solved by meta-rules, an analysis as we provide with *DSketch* is not included. An advantage of *LADDER* and its recognizer is that very different domains can be specified and recognized, e.g., Japanese Kanji [28].

A drawback of the rule-based recognition of *LADDER* is its reported runtime performance that may exceed one hour for 30 lines that have not been recognized so far [29]. Accordingly, a new high-level recognizer has been proposed. The primitives identified by low-level processing are first indexed to speed up later access. Then the indexed primitives are assigned in an exhaustive manner to the primitives defined by the component specifications. Constraints are checked in a dynamic order. Recognition rates and runtime of this new approach are not reported. However, informal testing with the download version of *LADDER* has shown that runtime can generally be expected to be real-time.

The approach by Costagliola et al. [8, 30] is also closely related to our work. Domain-specific recognizers are generated from grammars written in the *SkG*-formal-

[1] *DSketch* can be downloaded at http://www.unibw.de/inf2/DiaGen/dsketch

ism (Sketch Grammars). Three levels of recognition are employed: at the lowest level, primitives like lines and arcs are constructed from the input strokes by the *SATIN* toolkit [20]. At the next level, the primitives are grouped into (partial) symbols, each having an *importance rate*. This is done in a non-deterministic manner, based on a grammar in the *SkG*-formalism. Having an importance rate and the context, at the highest level it is finally decided for an interpretation of the sketch, and conflicting partial symbols are pruned according to meta-rules. The full system works incrementally. The reported recognition rates typically exceed 90%, with some exceptions. As a comparison, recognition rates are also taken without disambiguation. In this case recognition rates are clearly reduced. Runtime is not reported.

A generic framework, *InkKit*, is contributed by Plimmer et al. [19, 31]. Its main goal is similar to that of *SATIN*, that is to build sketch editors with comparatively little programming effort. Recognition is done by a modified gesture recognizer based on Rubine [15] that also supports multi-stroke symbols. The framework is capable of automatic separation of drawing and text. Resolution of ambiguity is based on a meta-rule: each primitive is assigned a probability, and the most probable primitives are combined in order to obtain components. Research focus of *InkKit* often also goes in the direction of HCI, e.g., by investigating multi-fidelity [32].

In *SketchREAD* by Alvarado et al., Bayesian networks are employed to reason about diagrams [4, 33]. The approach first generates hypotheses for components, based on primitives gained from low-level processing the input strokes. Second, using the Bayesian network it is determined how well these hypotheses fit the input data. Because component hypotheses are also generated for partial components, primitives can be re-recognized as something else in a top-down approach, e.g., a line instead of a curve. Even more, it is possible to recognize components only drawn partially, and very simple rules can be specified to describe the context of a component, which helps to foster hypotheses. However, even for a small number of input strokes, runtime degrades as inference in a Bayesian network is costly.

Having reviewed high-level sketch recognition systems, we also take a look at low-level recognition. The low-level framework *CALI* is based on features [16]. Segmentation cannot be performed, clustering is done automatically if strokes are drawn quickly one after another. All calculated features depend on the convex hull, the ordering of the points (and strokes) is not regarded. The system detects a small set of primitives (straight lines, rectangles, circles, wavy lines, etc.) reliably and quickly using fuzzy logic. Training allows to add further primitives, but it is not clear for which visual appearance of primitives the applied features of *CALI* are suitable. Combining primitives into domain-dependent components must be done by applications utilizing *CALI*, such as [34], an application for drawing user interfaces.

Sezgin et al. use Hidden Markov Models [35] for low-level recognition. Their recognizer takes into account the specific drawing styles of individual users. Reported recognition rates and run times for the recognizer are very good.

Another low-level recognizer is *PaleoSketch* by Paulson et al. [36]. Based on a clever selection of features, this approach is capable of recognizing different graphical primitives like lines, arcs, curves, spiral, helixes and more. All primitives are assumed to be drawn in one stroke. A simple heuristics is applied to split a stroke in case that it contains more than one primitive. The reported recognition rates are excellent, but the

involved evaluation assumes that, with one exception, each stroke represents exactly one primitive. Given the user study we conducted, this assumption does not hold for practical sketches.

Derived from the overview of related work, given in this section, we want to achieve the following improvements over other approaches:

- We want to achieve a very efficient recognition. Therefore, we will employ a search plan that allows for searching primitives shared by different components. Also, the search plan will define a static order in which constraints are checked; a dynamic trial-and-error method that may check constraints in vain is not necessary this way.
- Our general architecture should easily allow for extensions in terms of primitives and constraints. The respective interface for these extensions will be very flexible and allows to compute data on the fly, which is then cached, as opposed to other approaches where the complete data must be precomputed and indexed in advance. An example is the clustering of two straight lines to form a third line that is longer. We can compute this longer line on demand, and do not have to compute it in advance.
- To cope with the complexity of geometry-based recognition, we will design the search plan in a way that the number of possibilities decreases as quick as possible. This will be achieved by assuring that the partial components obtained during recognition are always connected. This way, coincidence becomes a more important constraint than others.

In order to conclude this section, we draw a final comparison between the illustrated high-level approaches *LADDER*, *Sketch Grammars*, *InkKit*, *SketchREAD*, and *DSketch*. All of these approaches are generic, are capable of multi-stroke input (thus enabling clustering), and are geometry-based (except for *InkKit* that is feature-based). Table 1 briefly shows commonalities and differences.

4. The model-based recognizer

This section discusses the recognizer in detail. Its basic idea is to describe diagram components in terms of primitives. For this purpose, several types of primitives are predefined. For recognition, primitives are searched for in the models according to the description of components. If found, the component can be assembled and passed on to the analysis (cf. Section 2). Note that we currently support only a small set of primitives, other approaches like *PaleoSketch* certainly support a larger set. However, the focus of this paper is the overall architecture and concepts of our recognizer, and not a broad set of primitives. More kinds of primitives can be easily added to the system.

4.1. Specification

A component consists of primitives and constraints on these primitives, e.g., restricting the length of a line, or the angle between two lines. In Figure 2 the specification of an arrow with an open head and of a rectangle parallel to the axes is given. A

Approach/tool Recognition	Analysis	On a Glance
LADDER		
rule-based / complete enumeration	no	little restrictions for user, powerful specification language, can be applied to many different domains, incremental
Sketch Grammars		
grammar-based, driven by importance values	yes	same formalism for shape and syntax description, fully exploits context, incremental
InkKit		
enhanced multi-stroke Rubine	no	toolkit, can combine sketches from different domains, understands text in sketches, much focus on HCI aspects, e.g., multi-fidelity
SketchREAD		
Bayesian networks	no	top-down re-recognition of primitives, can recognize partial primitives, simple rules to describe context, incremental
DSketch		
models, search plan-driven	yes	no restrictions for user (segmentation), efficient, extensible, designed for diagram languages, fully exploits context

Table 1: Comparison of generic high-level approaches to sketch recognition. The table shows the approach to high-level recognition, presence of a syntactical and semantical analysis, and special characteristics for each approach.

graphical representation of the arrow and the rectangle, not showing the constraints, is shown for clarity.

The arrow consists of three primitives, two for its head, and one for its shaft. Each primitive and each junction point between primitives has a name (an identifier), as indicated by `from`, `to` and `id`. The `type` of the three lines is set to `arbitrary`, which means that they can be arbitrarily rotated and do not need to run parallel to the axes. This contrasts to the rectangle, where the four sides are specified to be either vertical (`up`, `down`) or horizontal (`left`, `right`).

For the arrow some further constraints are necessary in order to prohibit any three lines meeting in one junction point to be an arrow. Thanks to the names these constraints can be easily specified. The first constraint requires the shaft to be longer than 80 [pixel]. The next two constraints require the two lines forming the arrow head to be shorter than the shaft. These two constraints are also `required`, which means that they must be satisfied for a valid component. If not specified, this attribute is regarded as `false`, meaning that the constraint should be satisfied (but does not have to be) in order to identify a valid component. This is equal to hard and soft constraints as they are used in *LADDER*. If a non-required constraint is not satisfied, the rating of the component is decreased (cf. Section 4.4), but the component is not discarded. Finally, the

```
<component name="rectangle">
  <primitives>
    <line from="ul" to="ur" type="right" id="top" />
    <line from="ll" to="ul" type="up"    id="left" />
    <line from="lr" to="ll" type="left"  id="bottom" />
    <line from="ur" to="lr" type="down"  id="right" />
  </primitives>
</component>

<component name="arrow">
  <primitives>
    <line from="head" to="tail"  type="arbitrary" id="shaft" />
    <line from="head" to="sideA" type="arbitrary" id="lineA" />
    <line from="head" to="sideB" type="arbitrary" id="lineB" />
  </primitives>

  <constraints>
    <fixedlength compare="gt" ids="head,tail" length="80" />
    <relativelength required="true"
        compare="lt" ids="head,sideB,head,tail" />
    <relativelength required="true"
        compare="lt" ids="head,sideA,head,tail" />
    <fixedangle" required="true"
        compare="gt" ids="sideA,head,tail" angle="20" />
    <fixedangle required="true"
        compare="lt" ids="sideA,head,tail" angle="70" />
    <fixedangle required="true"
        compare="gt" ids="tail,head,sideB" angle="20" />
    <fixedangle required="true"
        compare="lt" ids="tail,head,sideB" angle="70" />
  </constraints>
</component>
```

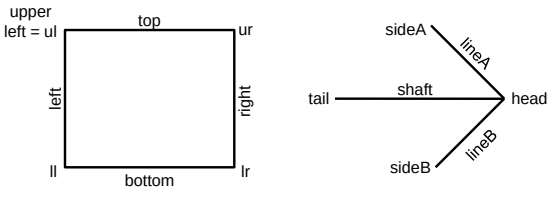

Figure 2: XML specification of an axially parallel rectangle and an arrow, and graphical representations.

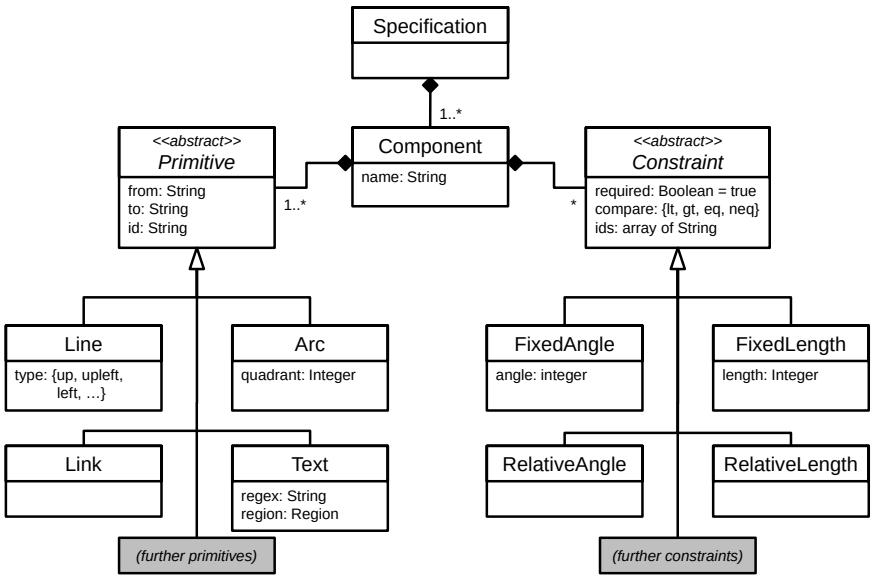

Figure 3: Class diagram showing the range of primitives and constraints that can be specified for a component.

last four constraints require the angles between `sideA` (`sideB`, respectively) and the shaft to be less than 70° and greater than 20°.

Although these two examples use only lines, four types of primitives are currently supported: *straight lines* (as in the examples), *arcs*, *links*, and *text*. Arcs (always assumed as a quarter of an ellipse lying between the axes of the ellipse, i.e., fully inside a quadrant of the coordinate system with the center of the ellipse as origin) are quite similar to lines, but do not have a `type`. Instead, they are characterized by an attribute identifying the quadrant (1 to 4) where the arc is placed in, and an attribute telling about the direction of rotation. Links, on the other hand, simply connect two junction points with an arbitrarily shaped connection in between. For example, for the shaft of an arrow this is a useful alternative to straight lines. Using a link, the shaft may have bends and can be curved. Using a line, as in the example in Figure 2, requires the shaft to be straight. Finally, the text primitive is described in terms of regions where text can be added to components. Regions are constructed as polygons of junction points, defined in the specification. Unlike other primitives, but like constraints, text can be specified as required or not. In case that some text is required, but not present, the component is discarded. Additionally, text can be checked against regular expressions to allow for some basic filtering. This allows, for example, for restricting text to numbers.

We found these four types of primitives sufficient for most domains (cf. Section 6). Further primitives could be added to the system. Figure 3 shows a class diagram comprising the full range of what can currently be specified. Note the two gray boxes which indicate where further primitives and constraints may be added in the future. Also note

```
define shape OpenArrow
  description "An arrow with an open head"
  components
    Line shaft
    Line lineA
    Line lineB
  constraints
    coincident shaft.p1 lineA.p1
    coincident shaft.p1 lineB.p1
    coincident lineA.p1 lineB.p1
    equalLength lineA lineB
    acuteMeet lineA shaft
    acuteMeet shaft lineB
  aliases
    Point head shaft.p1
    Point tail shaft.p2
display original-strokes
```

Figure 4: Specification of an arrow with an open head in *LADDER*. Editing operations are not shown.

$$\begin{aligned}\text{Arrow} \rightarrow\ & \text{LINE}_1(60)\ \langle \text{joint}_{2\_1}(t_1),\ \text{rotate}(45,t_2)\rangle\ \text{LINE}_2(20) \\ & \langle \text{joint}^1_{2\_1}(t_3),\ \text{rotate}^1(\text{-}45,t_4)\rangle\ \text{LINE}_3(20), \\ \{\ & \text{Arrow.attach}(1) = \text{LINE}_1.\text{attach}(1); \\ & \text{Arrow.attach}(2) = \text{LINE}_1.\text{attach}(2) \cup \text{LINE}_2.\text{attach}(1) \cup \text{LINE}_3.\text{attach}(1);\ \}\end{aligned}$$

Figure 5: Specification of an arrow with an open head in *SkG*.

that, for the syntactical analysis in *DSketch*, further properties of components have to be specified as well; however, these are of no interest in the context of this paper.

As mentioned before, both *LADDER* and *SkG* are similar in that they allow for specification of the visual appearance of components. *LADDER* uses different terms: a component is called a *shape* here, and so is a primitive called a *component*. Figure 4 shows the specification of the arrow, taken from [37]. Unlike our approach, *LADDER* also allows for defining editing operations. However, this part of the specification is omitted in the figure. Figure 5 shows the specification of the respective arrow in *SkG* [30]. Not shown in the figure, but very appealing, is that the grammar of a diagram language can be described in *SkG* as well, and by the same formalism. Without going into details, the similarities of our description language and the other two are obvious. In general, *LADDER* supports many different constraints, and has a syntax that is easily understandable and writable. The syntax of *SkG* is more cumbersome, as much information is contained in indexes. We decided for a specification in XML, as this simplifies reading and writing the specification using respective XML libraries that are available for many programming languages nowadays.

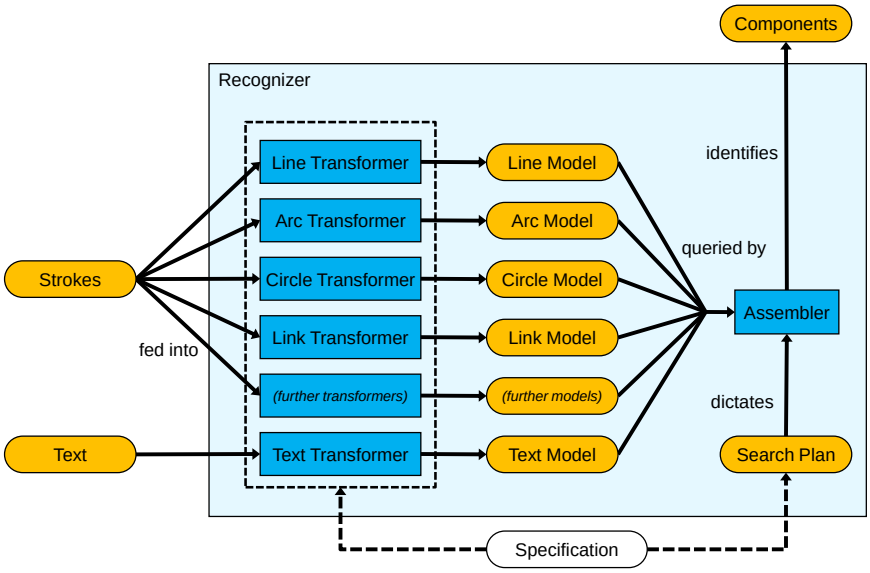

Figure 6: Conceptual view of the recognizer shown in Figure 1.

4.2. Models

When the user draws strokes using the stylus on the canvas of the drawing tool, the system records these strokes for the recognition process. Like in almost any other approach, strokes are regarded as sequences of tuples (x, y, t), where x and y mark a position on the canvas, and t is a timing information (the elapsed time in milliseconds since the first tuple in the sequence). Currently, timing information is not used.

Figure 6 shows a conceptual view of the recognizer that will be explained in this section. While the user draws, the system does neither know about clustering and segmentation, nor does it know what strokes (or substrokes) are meant to represent which primitives. To account for this uncertainty, each (sub)stroke must be interpretable as any available primitive, i.e., as a straight line, an arc, or a link. To do so, we rely on different *models*. Each model represents a certain view on the strokes and interprets them only regarding its view. The *assembler* can then query the models for the different primitives. If a model identifies primitives suitable for the query, it returns them.

Models are not supposed to store strokes directly, but only preprocessed information required to satisfy the queries from the assembler. Therefore, each model has associated a *transformer* that performs the preprocessing for exactly this model, i.e., low-level processing of the strokes. The preprocessing serves two purposes. First, it stores information in a format suitable for the model, e.g., in the circle model circles are stored by their center and radius, and in the line model, straight lines are stored by their end points. In this respect, our transformers work like other low-level recognizers, e.g., *PaleoSketch* [36]. Second, as the transformation takes place, the information is inevitably abstracted, thus enabling faster replies to the queries of the assembler.

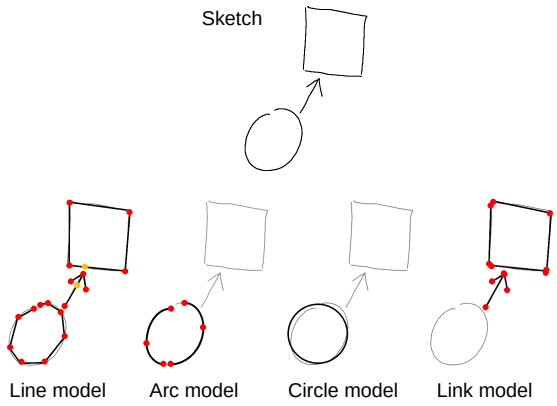

Figure 7: A simple exemplary sketch and graphical representations of the four models. The text model is empty and therefore not shown.

Additionally, the transformers also access the specification and decide about which strokes to preprocess and which to discard. This way, the contents of the model can be adapted to the specified diagram language, and useless information can be discarded as early as possible, thus speeding subsequent processing. As an example, the circle transformer discards each stroke that is a straight line. Furthermore, if a certain kind of primitive is not used in the specification, the transformers may disable themselves completely, e.g., in Nassi-Shneiderman diagrams only straight lines are used, so all transformer preprocessing other primitives are disabled.

Until now five different models have been implemented. One for each primitive (line, arc, link, text), and one especially designed for identifying circles. Further models are conceivable and may be added to the system, either to recognize new primitives, or to improve reliability for the mentioned primitives. In the following each model is discussed in detail, then the actual recognition process is explained.

As mentioned before, the transformers preprocess (filter, transform, and abstract) the information represented by the strokes. Figure 7 shows an example drawing and graphical representations of its different models. The drawing consists of six strokes forming a rectangle, a circle and an arrow. Results of preprocessing can clearly be seen by the bold, black lines that indicate the data contained in the respective models. Also part of the preprocessing by the transformers is to decide about clustering and segmentation. Depending on the view of their models, the transformers must decide which strokes to combine, and which to split.

The **line transformer** tries to interpret the whole drawing as if it consists of straight lines only. Basic vectorization algorithms can be applied for this purpose. We decided for a very simple one that proves to work very well and very fast for practical testing. The transformer applies the following steps on each stroke independently:

1. to initially smooth the user's strokes, from a stroke each tuple is discarded that has a distance less than 5 [pixel] from its predecessor. This value is a threshold and can be changed by the user. Like all other thresholds, its value has been determined empirically by testing.

2. from three remaining consecutive tuples p_1, p_2, p_3 the transformer discards p_2 if the angle between the three tuples lies within the interval $(160°, 200°)$ that represents another threshold. Without the first step, this second step fails, as most hardware has a very high sampling rate and consecutive samples are usually next to each other (distance 1 or $\sqrt{2}$), resulting in angles of 0°, 45°, 90°, 135°, etc.
3. For each remaining two consecutive tuples, a line is added to the model from the one point to the other. Furthermore, each of those lines is attached an attribute regarding its direction. It can have one of the four values horizontal, vertical, diagonal ascending to the left, or diagonal ascending to the right. This attribute is later used for answering queries by the assembler.

Finally, all lines are split at points of mutual intersection or proximity, and information is collected about lines close to each other. The results of the line transformer can be seen in the line model in Figure 7. The circle is approximated by straight lines as well, as the line transformer, like all other transformers, works on a best-effort basis to produce any reasonable result that is possible.

The **arc transformer** works very similar to the line transformer. The idea here is to approximate each stroke by quarters of ellipses lying inside quadrants of the coordinate system. Straight lines are dismissed (cf. Figure 7). In the arc model, arcs are stored by their two end points, direction of rotation, and quadrant (1 to 4). The following steps are applied to each stroke independently:

1. same as for the line model, step 1.
2. the list of remaining tuples is split at points where the direction of rotation changes. Consequently, the samples in the resulting sub-lists always describe a turn to the left or to the right.
3. based on the direction of the connection between two consecutive points, for each sub-list the arcs (quarters of ellipses) can be obtained.

Unlike the line transformer and the arc transformer, the link transformer and the circle transformer are more driven by heuristics. For the **circle transformer** we implemented a feature-based recognizer that is able to identify closed circles drawn in one stroke. This is justified by the observation that most users draw circles in one stroke indeed, even if not explicitly told to do so. All strokes that pass this series of tests are then stored as circles in the circle model, saving center and radius. The transformer works in the following way:

1. if the bounding box of the stroke is more than twice as high as wide, or more than twice as wide as high, the stroke is dismissed, i.e., it is not regarded as a circle.
2. using a least-square-error analysis, center, radius, and total rotation (like the *total rotation* in [18], or the *total angle traversed* in [15]) are calculated.
3. if the total rotation is less than 300°, the stroke is dismissed.
4. if the actual length of the stroke compared to the calculated length for a perfect circle (regarding the total rotation) differs more than 10%, the stroke is dismissed.

5. if three consecutive tuples from the stroke describe an angle that is too acute (very similar to the line model), the stroke is dismissed. For this test the first and last 20% of a stroke are excluded, as we found out that users frequently draw hooks here that in fact describe acute angles.

Actually, the arc model contains all information necessary to identify a circle, as a circle is composed of four arcs. However, the arc transformer sometimes misses an arc, so the overall reliability of our approach has been improved by including the circle transformer and circle model. Including additional such pairs of transformers and models is supported by the architecture of our approach: new pairs can be easily added in order to augment the overall robustness and reliability.

The **link transformer** regards every stroke as a link, unless its end points are very close to each other, thus the stroke forms some closed shape. For instance, the stroke forming a circle in Figure 7 cannot be regarded as a link since its end points are too close. The other strokes, however, that make up the square, as well as the two strokes that make up the arrow shaft and head are recognized and stored as links as seen.

Additionally, links may be split at points with a very acute angle. This mechanism had to be added as we observed that, for example, when drawing an open-headed arrow some users tend to draw the shaft (a link) and one of the two lines for the head in one stroke. To account for this behavior, links must be split at acute angles. The arrow head in Figure 7 is split, for example, because it is drawn in one stroke and shows an acute angle at the arrow's tip. Links are stored in the link model by the two points they connect.

As shown in Figure 6, there is also a **text model** which can be queried by the assembler. To distinguish text from graphics, no *divider* is used (a piece of software that can perform this distinction), which is a difficult issue and far from solved [19, 38]. Instead, the drawing tool of *DSketch* is mode-based and requires the user to explicitly indicate input of text. The input is then directly transformed into a string representation that is fed into the text transformer. This transformer stores the bounding box of the text and the text itself in the text model. After all primitives of a component have been identified, the assembler consults the specification and constructs the regions where text may be added. The assembler then queries the text model and adds those texts from the text model to the components where the text's bounding box and the specified regions overlap.

As this section shows, low-level recognition of strokes is done by the transformers. Already existing and newly created approaches may be combined with our proposal, by adding them as new transformer-model pairs just like the circle transformer and model. This means that very much of the research done so far is orthogonal to our approach, and can be seamlessly integrated.

4.3. Recognition

As mentioned before, the assembler searches for primitives of a component one after another, until all of them are identified. Then, the component can be assembled. To speed up processing by avoiding double effort, primitives common to different components are searched for jointly. For this purpose a *search plan* is used that is precomputed based on the specification. As an example consider *Nassi-Shneiderman*

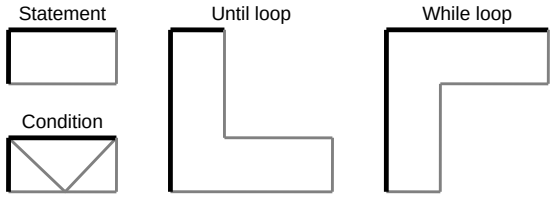

Figure 8: Diagram components of NSD. One possibility for common primitives is indicated by bold strokes.

diagrams (NSD). They have a simple syntax with only four different diagram components, as shown in Figure 8. Primitives common to all components are, for example, a vertical line that is, at its upper end, connected to a horizontal line (indicated by the bold strokes in the drawing). There are other combinations possible as well.

These common primitives are searched for jointly as long as possible. Then the search process branches off for the primitives of the individual components. The decision which primitives to search for jointly, and when to branch off individual components is determined by the search plan. Because finding an optimal search plan is not trivial, a heuristic is applied. A greedy algorithm always selects that alternative for the next step that preserves most of the components for joint search. The search plan is always a tree. In general, given a specification, there are different possibilities for the search plan. Figure 9 shows *one possible* search plan for NSD. For each node (i.e., step in the search process) the highlighted bold lines indicate that primitive that has been added last. Components are fully identified at leaves of the search plan (indicated by *completed*), after the last primitive was found. Furthermore, although not the case in this example, components may even be fully identified at an inner node. The information which components are still possible at a given node is actually not required by the search process, but serves as clarification here. Note that nodes `D` and `I` are different, because `D` connects both horizontal lines, while `I` does not.

Following the search plan shown in the figure, the assembler starts by querying a horizontal straight line in the line model. In the second step (node `B`), the assembler queries a vertical straight line whose upper end point is equal (or close to) the left end point of the horizontal line obtained in the first step. For each result, the next horizontal line is queried in the third step. This process continues for each result until either the leaves of the search plan are reached, or if no model returns a primitive to a query. An example for an actual search process is given in Section 5.

Constraints are defined on the junction points between primitives, as it can be seen in the example in Figure 2. By following the search plan, the assembler identifies these junction points one after another, by adding further primitives to a partially identified component. As soon as all junction points are identified to check a required constraint, this is immediately done. If the constraint is not satisfied, the search process for that partially identified component is pruned at once.

A noteworthy property of the recognition process is that, at each node, all identified primitives are always connected to each other. This property is assured when the search plan is computed. The first primitive of a component that is searched for (the horizontal line in the example) is not restricted regarding its position on the canvas. The next

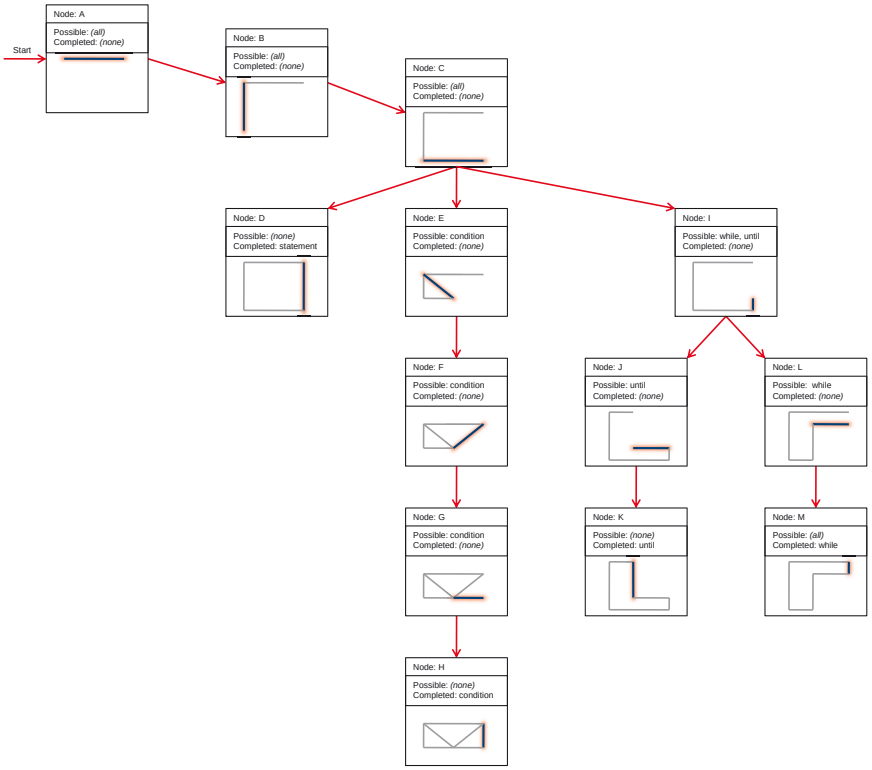

Figure 9: One possible search plan for NSD.

primitive must then be connected to this first one, i.e., must have a common junction point. The third must be connected to one of the two previous ones, and so on. This way, the set of already identified primitives is always connected. The benefit of this method becomes clear during the recognition process. As one or more of the junction points of a primitive are already known, there are less alternatives for this primitive that need to be considered. Accordingly, the recognition process is speeded.

It lies in the nature of hand-drawing that precision is lacking, and that there may be some gaps in drawings of a component that is actually solid. Examples can be seen in Figure 7; neither the circle nor the rectangle are closed, the lines of the rectangle and of the arrow are not perfectly straight, the circle is not evenly rounded. For connecting such primitives, of course some threshold is considered, in order to also recognize imprecisely drawn components. Furthermore, for components which are not completely connected, e.g., a final state in a state chart, all connected sub-components are recognized first, and only then these sub-components are related to each other, restricted by constraints.

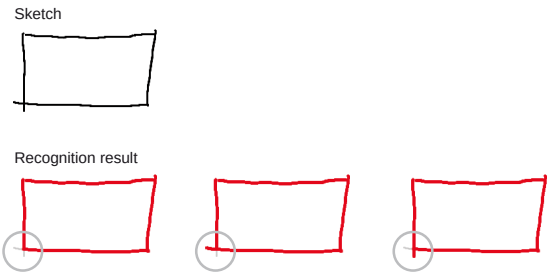

Figure 10: One sketched rectangle, and the three rectangles recognized from this sketch. Two of the three are duplicates; the differ in the lower left corner, as marked by the circles.

4.4. Elimination of duplicates

A typical behavior of the recognizer is that the same component is often identified more than once, each time with slightly different junction points between the primitives. An example is shown in Figure 10. The lower left corners of the three recognized rectangles are different. Such duplicates happen due to the thresholds applied during recognition; in each of the three recognized rectangles in the figure, the left vertical line and the lower horizontal line are considered to be coincident at the corner. However, as the figure shows, different lines satisfy this requirement and are coincident. The assembler considers all of these different lines, which results in the duplicates.

The subsequent analysis in *DSketch* (cf. Section 2) can handle these duplicates: all rectangles evolve from the same strokes, and thus only one is considered to be correct. However, this analysis consumes more time the more components have been recognized. Accordingly, three heuristics have been implemented to suppress those duplicates. One of two completely identified components of the *same type* (depending on the specification, e.g., statement, condition, while loop, or until loop) is discarded

- if the junction points with the same identifier from the two different components are very close to each other (within a range of some pixels), or
- if exactly the same substrokes are used for both components, not regarding how these are assigned to different primitives, or
- if the *fully connected primitives* of both components use the same substrokes. Fully connected primitives are those primitives where *all* junction points are connected to other primitives. For example, an arrow has no fully connected primitives, and a rectangle has only fully connected primitives.

Each identified component is equipped with a rating that depends on how many primitives and constraints are defined for that component, on how precisely each primitive is drawn, and on how close the connections at the junction points are. Whenever one of two components has to be discarded, that one is chosen that has the lower rating. In the example from Figure 10, the leftmost rectangle gets the highest rating, so the other two are considered to be duplicates. Note that, as briefly mentioned before, the rating is decreased for non-required constraints that are not satisfied.

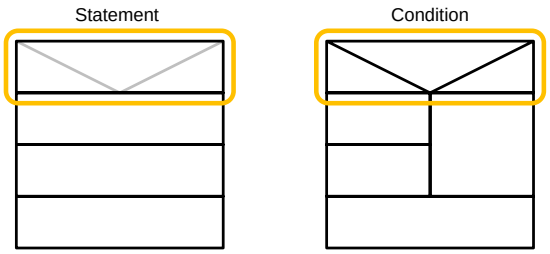

Figure 11: An example of ambiguity resolution by use of context information. The light gray lines in the marked statement on the left hand side are regarded as wrong.

Empirical testing has shown that there are duplicates that are matched by only one of the three rules. Accordingly, it is useful to apply all rules, and not just one. Testing has also shown that these rules only remove duplicates, and no other components.

4.5. Analysis

Subsequent to the recognizer, *DSketch* applies the analysis. Of course, the results of recognizers are almost always ambiguous. In this subsection a source of ambiguity specific to our approach is discussed. The search plan for NSD (Figure 9) clearly shows that the assembler always identifies a statement whenever a condition is identified. The system recognizes this ambiguity, but cannot decide for either the statement or the condition yet. A conceivable meta-rule could suppress the statement, for example, because the condition has more primitives, thus gaining a higher rating. However, this rule does not include the context of the component in question that usually points out what choice is the right one. An example is given in Figure 11. Although the framed component looks like a condition, on the left hand side it makes only sense to regard it as a statement. The two diagonal lines can be regarded as wrong in this case. On the contrary, on the right hand side, the component must be clearly a condition. No meta-rule could ever distinguish these two cases. Accordingly, their resolution is done by the analysis that happens subsequent to the recognition of components. Apart from this example, analysis is not discussed in this paper, but in [23].

5. Example

This section discusses a comprehensive example for NSD. We assume the drawing in Figure 12 and the search plan shown before in Figure 9. Only straight lines are used for NSD, so the former figure only shows the line model. In the model each line is given a unique name that is used below to describe the search process. The line model does not contain any information about order, time, or direction of the strokes, as the assembler is independent of this information. Lines `g` and `h` are actually only one line, but were split by the line transformer because the right end point of `d` is closer than a threshold to that one line. We see in the following why this is necessary.

The complete run of the assembler is shown in Table 2. The left-most column refers to the different nodes of the search plan (cf. Figure 9). Each cell in the body of

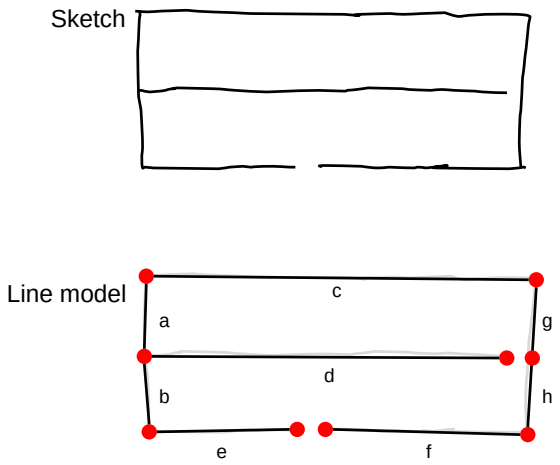

Figure 12: An exemplary drawing of a NSD, consisting of two statements. Text is not regarded.

the table represents a state of the search process, i.e., a partially or completely identified component. Bold crosses indicate that no further primitive can be added to a state, and that the search process does not continue here. Checkmarks indicate that a component has been completely recognized. The letters inside the cells refer to the lines in the line model, and are always given in the order in which they have been added. The bold dashed lines indicate which state emerged from which other state by adding a new primitive, where the predecessor is always above its successor. In the following, a cell (a state) is referred to by listing the lines inside the cell.

Consulting the search plan, the assembler first queries for horizontal lines (node `A`). The line model has four results to this query: `c`, `d`, `e`, and `f`. The only successor of node `A` in the search plan is node `B` that requires the assembler to query for a vertical line connected to the first horizontal line. Both for `e` and `f` the line model returns empty results, so the search process ends immediately for these two states. For `d`, line `b` is returned. For `c`, there are two possibilities, `a` and `ab`. At this point, state `c` is forked, both resulting states `ca` and `cab` are then processed independently. The answer `ab` is computed by the line model on the fly, and cached for possible later access. It results from the fact that `a` and `b` can be joined in order to make a longer vertical straight line.

Up to this point the search process is described in breadth-first order. However, the actual order in that the assembler processes a state by querying for its next primitives does not matter. An arbitrary mix of breadth-first and depth-first is possible and does not alter the final result, i.e., the components that are recognized.

The search process continues as it can be seen in the table. It needs to be forked two more times. In general, an independent branch is forked for every result returned for a query. For the search plan in this example, each state in rows `D`, `F`, `I`, and `K` represents a completely identified component. It can be seen that the search process identifies three statements, although only two are actually drawn. The reason is that the third statement `cabefhd` is given by all lines, except `d`. Note that `cadg` could not have been identified

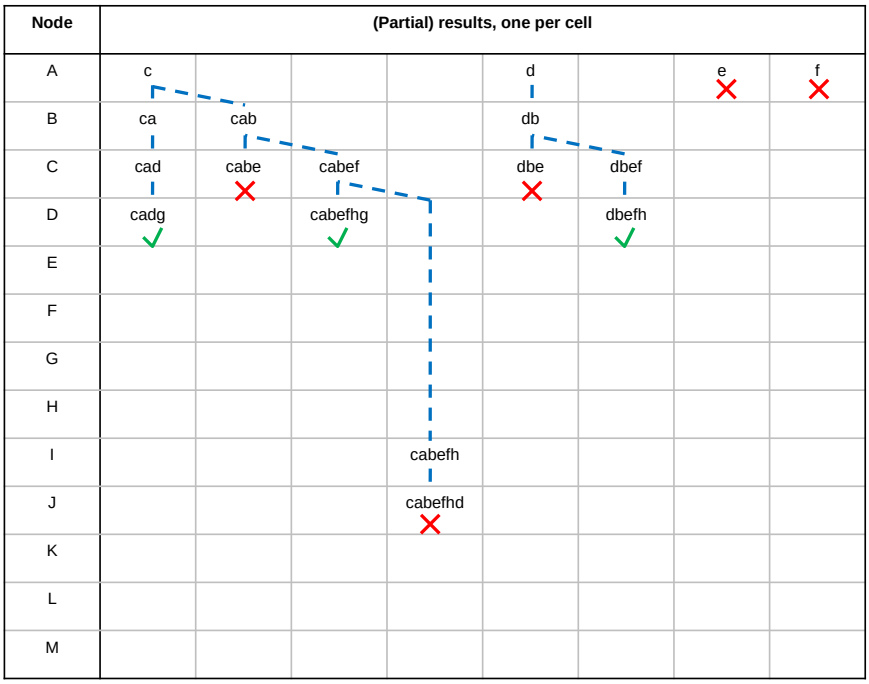

Table 2: Table showing all partial and completed results obtained by the assembler during the recognition process.

if `g` and `h` would not have been split, as in this case the line model would not have been able to return a result for `cad` at node `D`.

In this example we have seen that lines are queried in the models for none of their end points given (node `A`), for one of their end points given (e.g., nodes `B`, `C`, and `I`), or for two of their end points given (e.g., nodes `D`, `E`, and `K`). The same observation also holds for arcs and links. None, one or two of the end points are known from previous search results, and a primitive is queried to continue this partial component. An arrow, for example, could be identified by first finding a link where none of its end points is given. Then, in different steps of the search process, both of the link's end points are tried as tip of the arrow, and the two lines that make up the arrow head are queried for.

If text would be specified for the components, it would be searched for in the text model right after every other primitive is identified. For example, for statements you could specify that the polygon given by the four junction points must intersect with the bounding box of some text written on the canvas. For an arrow, which does not occur in NSD, you could specify that the bounding box of some text must be close to one of the end points of the arrow. Using algorithmic geometry techniques, this can be computed efficiently.

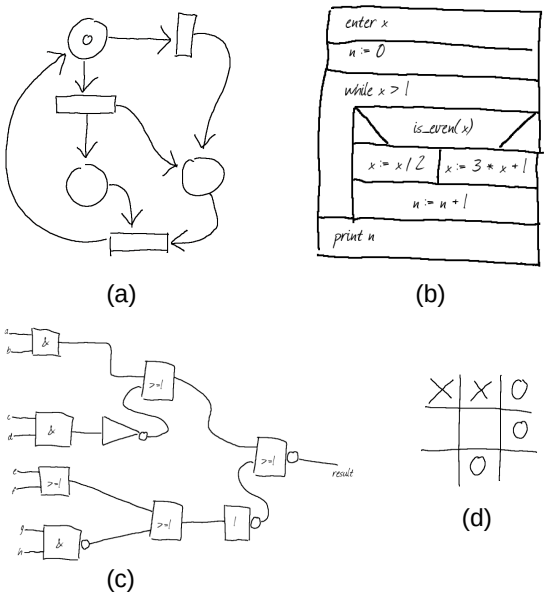

Figure 13: Examples for the case studies. (a) Petri net, (b) NSD, (c) logic gate, (d) tic-tac-toe.

6. Evaluation

This section describes five different examples we have implemented with our system as case studies. For each case study, its visuals are described briefly, and an example drawing is given. All of these examples are recognized correctly. Section 6.6 reports about the performance of the implementation for each case study, and discusses the lessons learned. These case studies are intended to show the range of domains where our approach can be applied, and the runtime of the implementation. Section 6.7 reports recognition rates obtained from a user study.

6.1. Petri nets

Also known as place/transition nets, Petri nets are used to model the behavior of (distributed) systems. The language consists of four different component types. *Places* are shown as circles and may contain one or more *tokens*. Although tokens are commonly shown as small filled circles, the filling is omitted and tokens are drawn as circles, too, as our recognizer cannot cope with such patterns. The analysis (cf. Section 2) reliably detects the difference. Finally there are *transitions*, drawn as rectangles, and open-headed *arrows*. Arrows connect either a place and a transition, or a transition and a place, but never two places or two transitions. Figure 13 (a) shows a simple Petri net consisting of three places, three transitions, one token, and eight arrows. Note that we modeled the arrow shafts to be links; otherwise, they could not be bent as shown in the figure.

6.2. Nassi-Schneiderman diagrams

NSD are used to visualize structured programs. Its four different types of components are shown in Figure 8 (actually NSD are more powerful and have more components, but for the sake of simplicity this subset is assumed). An algorithm for computing the Collatz sequence is given in Figure 13 (b). It consists of six statements, one loop, and one condition.

The mentioned figure also shows clustering and segmentation. For example, the left vertical line spanning from the first statement to the last is drawn in one stroke, although it contributes to four components (two statements, the loop, and the final statement). Hence this line is segmented. Additionally, each of these four components requires more lines than just this vertical line, so the recognizer also has to cluster lines.

6.3. Digital logic gates

Common boolean logic can also be expressed graphically. As operators `and`, `or` and `not` are assumed. All of them are drawn as rectangles, input on the left, output on the right. Operators `and` and `or` are always assumed binary, `not` is unary. The operators are distinguished by text written inside the rectangles. *&* stands for `and`, *>=1* stands for `or`, and *1* stands for `not`. A small circle, called a *bubble*, can be drawn between an operator and its output that means that the output is negated. This is mandatory for `not`; `and` and `or` become `nand` and `nor` this way. For `not`, a triangle can be drawn instead of the rectangle. In this case no text is necessary (cf. Figure 13 (c)). The figure represents the expression `result := not(a and b or not(c and d) or not(e or f or not(g and h)))`. The advantage of the graphical representation is its clarity for larger expressions.

6.4. Tic-tac-toe

Apart from the technical examples given so far, it is even conceivable to implement simple paper-based games with our approach. As an example tic-tac-toe was chosen, because it is possible to give suitable rules for the analysis which is, as mentioned before, crucial to *DSketch*. Using the analysis, some reasoning about game situations can be done (because the system has no game engine, you cannot actually play against the machine). For the situation shown in Figure 13 (d) the system correctly tells that it is player X's turn.

6.5. GUI builder

Another example not regarding a traditional diagramming language is the GUI builder. The idea is to simply draw a window with some widgets, have the system recognize the drawing and generate an actual window from the recognized information. An example is given in Figure 14. This example also shows all widgets supported: combo boxes, text fields, checkboxes, radio buttons, regular buttons, and sliders. The graphical representation should become obvious from the drawing. The generated window is made in such a way that the correspondence to the drawing can be clearly seen.

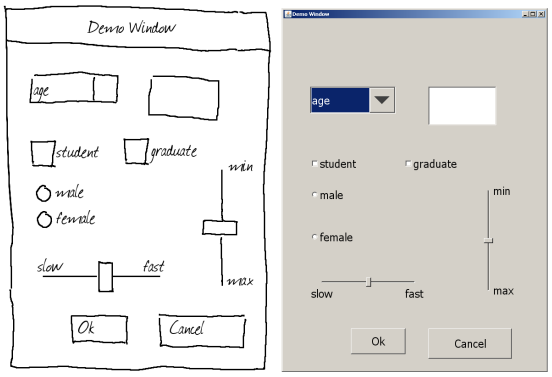

Figure 14: Example for the GUI builder. The right hand side shows the generated window.

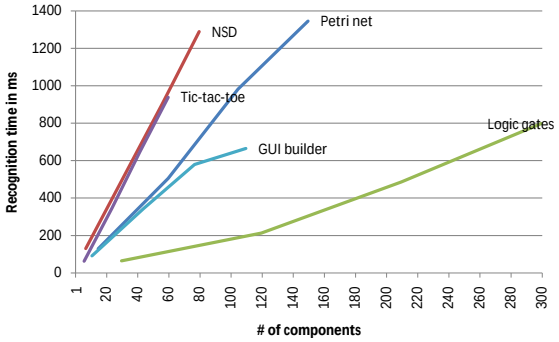

Figure 15: Runtime measurements for the five case studies for a linear increase in load. The vertical axis shows time in milliseconds, the horizontal axis shows the number of shapes that had to be recognized.

6.6. Runtime performance

To evaluate the performance of the prototypical implementation the five examples from above are used. For each case study the time for complete recognition (i.e., non-incremental) is measured, using one, four, 7 and 10 copies of the respective example, thus linearly increasing the load. Each setting is repeated 10 times, only the lowest value is reported (measured in milliseconds). This is valid because the implementation is deterministic, and does the same for each run. Extra time is thus consumed by the operating system. As hardware a PC was used running Windows XP on an Intel dual core CPU with 2.13GHz and 2GB of main memory.

The result is shown in Figure 15. In all cases the time for processing increases roughly linear with the input size. The average time to recognize *one actually drawn component* ranges from about 2ms for logic gates to about 16ms for tic-tac-toe and NSD. The small figure for logic gates is due to the large number of connections between the operators that are made by only one link and one text each, and can thus be recognized very fast. On the contrary, for tic-tac-toe, the board is a complex component that takes more time to be recognized; it consists of 12 primitives.

For NSD it could be observed that the number of false positives is very high due to the visual appearance of the single components. As shown in Section 5, the recognizer identifies an additional statement for each pair of consecutive statements, ignoring the horizontal line dissecting the pair. This issue, and a larger exemplary diagram consisting of 10 statements, three loops and two conditions, have been discussed in more detail in [22]. Here, the recognizer identified 56 components that would be passed to the analysis. As an optimization an option was added to the recognizer, allowing for dismissing a component if it contains one or more other components. Enabling this option, only 25 of the initial 56 components are kept, cutting the processing time for the full diagram (including analysis) from 2.5 seconds to 0.7 seconds. Still, the correct result was obtained. The drawback is that the error tolerance is reduced. In case of a false positive inside a component, the outer component is dismissed, although it may be correct.

Similar to NSD, a lot of false positives occur for the GUI builder. For example, each rectangle is recognized as a text field, although most rectangles are part of other components as well. However, the analysis reliably selects the intended components, driven by the rating of the components (cf. Section 4.4). In the shown case in Figure 14, 28 components are identified, although only 11 are represented in the generated window. Due to the internal structure used for generating the window, the effect on the analysis can be neglected here, as it is not as severe as for NSD.

6.7. Recognition Rates

Next to the applicability and the runtime performance of a system, the actual recognition rates are of high interest. Therefore a user study was conducted. A total of 16 participants each drew an equal exemplary diagram from each of the five case studies discussed above. Each participant was given some minutes of preparation time to get used to the hardware (a Wacom DTU-710 pen display). Then, the five examples were drawn, in a random order for each participant. The participants were not given any feedback by the system before they finished drawing each example. This means that no participant could adapt his or her style of drawing to the system. Furthermore, the participants were encouraged to draw freely, as they would with pen and paper.

The average number of components identified by the recognizer (positives) was measured for each of the five example, and so was the average number of those components that were intended by the user (true positives). The recognition rates are obtained by dividing the number of true positives by the number of components that are actually contained in the example. The analysis is not considered for this measure. The result is shown in Table 3. Note that the number of false positives (i.e., positives that are no true positives) is quite high. Often, false positives cannot be avoided, for example, as has been shown in Figure 11, and has been discussed in the previous subsection. These false positives can be removed by suitable meta-rules, for example. However, in *DSketch* (cf. Section 2) we use a syntax analysis to select the true positives, which produces more reliable results.

The recognition rates range from about 81% up to 94%, depending on the example. We believe this is quite a good result, as the approach offers great flexibility in how sketches can be drawn (regarding clustering and segmentation), and as the participants of the user study could not adapt their drawing style to the system. Petri nets mark

	Number of components in example	Average number of positives	Average number of true positives	Recognition rate
Petri net	14	19.8	11.3	80.7%
NSD	6	10.6	5.6	93.3%
Logic gate	7	36.1	6.4	91.4%
Tic-tac-toe	6	5.3	5.3	88.3%
GUI builder	11	14.8	9.8	89.1%

Table 3: Number of components, average number of positives, average number of true positives, and average recognition rate for each of the five exemplary sketches from the user study.

the bottom in this comparison with 80.7%, while the average recognition rate of the other four examples always exceeds 88%. We have observed that, in the case of Petri nets, participants tended to draw arrowheads very sloppy, making their arrowheads look more like an arc instead of two straight lines. Also, transitions were often drawn hastily that led to corners that were too rounded in order to be recognized properly. We assume that both issues could be solved by showing the participants why their sketches were not recognized properly. However, for this study we did not give any feedback, as already mentioned above.

7. Conclusions and future work

In this paper there is proposed an alternative to other geometry-based recognizers. Our approach solves the issue of clustering and segmentation, and allows for integrating previous work, and further primitives, due to its model-based concept. Low-level preprocessing is performed by independent transformers. A search plan is automatically computed to dictate the order in which primitives are combined, which provides several benefits: subcomponents are always connected, and the order and timing of constraint checking can be statically precomputed. Further constraints can be easily added. Case studies show that our approach can be applied to recognize components from very different diagramming languages. The performance is good on a typical desktop computer for various sizes of diagrams. A user study shows good recognition rates, although there is always improvement possible on this issue.

The goal of this work has not been to be able to process as many different primitives as possible. Instead, the focus has been the overall architecture and concepts. However, we plan to include more primitives and respective transformer-model-pairs as future work. We are currently working on a model for hatched and solid regions, as these frequently occur in diagramming languages. For example, transitions in Petri nets are drawn solid, and in architecture plans and blueprints one can often find hatched regions, e.g., for walls. In this respect, we also plan to include low-level recognizers from other groups, either as replacement to our transformers, or in addition. The effect on the recognition rates cannot be predicted, but will be very interesting. Especially *PaleoSketch* seems promising, as the authors report very high recognition rates [36].

The evaluation of runtime performance has shown that runtime grows roughly linear with input size. Although this is a good result, we want to achieve a better runtime. There are two ideas that we want to investigate for this purpose. The first is to make recognition incremental. This would probably include to preserve all search states and continue the recognition process for these states as soon as new primitives are available. The second idea is to use several CPU cores for recognition. As all search states are independent, they serve as an obvious starting point for parallel computation. Furthermore, informal testing revealed that even for small diagrams the number of search states runs into the hundreds, and that there are mostly a couple of dozen search states waiting to be processed. Accordingly, runtime could be significantly increased by parallel computation.

References

[1] A. Apte, V. Vo, T. D. Kimura, Recognizing multistroke geometric shapes: an experimental evaluation, in: Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology (UIST '93), ACM Press, New York, NY, USA, 1993, pp. 121–128.

[2] J. Mankoff, G. D. Abowd, S. E. Hudson, OOPS: a toolkit supporting mediation techniques for resolving ambiguity in recognition-based interfaces, Computers & Graphics 24 (6) (2000) 819–834.

[3] H. Hse, A. R. Newton, Sketched symbol recognition using zernike moments, in: Proceedings of the 17th International Conference on Pattern Recognition (ICPR '04), IEEE Computer Society, Washington, DC, USA, 2004, pp. 367–370.

[4] C. Alvarado, R. Davis, Dynamically constructed bayes nets for multi-domain sketch understanding, in: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI '05), 2005, pp. 1407–1412.

[5] H. Dibeklioglu, T. M. Sezgin, E. Ozcan, A recognizer for free-hand graph drawings, in: Proceedings of the 1st International Workshop on Pen-Based Learning Technologies (PLT '07), IEEE Computer Society, Washington, DC, USA, 2007, p. 17.

[6] T. Igarashi, S. Matsuoka, H. Tanaka, Teddy: a sketching interface for 3d freeform design, in: Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99), ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999, pp. 409–416.

[7] S.-H. Bae, R. Balakrishnan, K. Singh, ILoveSketch: as-natural-as-possible sketching system for creating 3D curve models, in: Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology (UIST '08), ACM Press, New York, NY, USA, 2008, pp. 151–160.

[8] G. Costagliola, V. Deufemia, M. Risi, Sketch grammars: a formalism for describing and recognizing diagrammatic sketch languages, in: Proceedings of the

Eighth International Conference on Document Analysis and Recognition (ICDAR '05), IEEE Computer Society, Washington, DC, USA, 2005, pp. 1226–1231.

[9] R. Zanibbi, D. Blostein, J. R. Cordy, Recognizing mathematical expressions using tree transformation, IEEE Transactions on Pattern Analysis and Machine Intelligence 24 (11) (2002) 1455–1467.

[10] M. Minas, Concepts and realization of a diagram editor generator based on hypergraph transformation, Journal of Science of Computer Programming, Special Issue on Applications of Graph Transformations 44 (2) (2002) 157–180.

[11] M. Minas, Generating meta-model-based freehand editors, in: A. Zündorf, D. Varró (Eds.), Proceedings of the 3rd International Workshop on Graph Based Tools (GraBaTs'06), Natal (Brazil), Vol. 1 of Electronic Communications of the EASST, European Association of Software Science and Technology, 2006.

[12] J. de Lara, H. Vangheluwe, AToM^3: a tool for multi-formalism and meta-modelling, in: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE '02), Springer, London, UK, 2002, pp. 174–188.

[13] T. Fischer, J. Niere, L. Torunski, A. Zündorf, Story diagrams: a new graph rewrite language based on the Unified Modeling Language and Java, in: Selected Papers from the 6th International Workshop on Theory and Application of Graph Transformations (TAGT '98), Springer, London, UK, 2000, pp. 296–309.

[14] L. B. Kara, T. F. Stahovich, Hierarchical parsing and recognition of hand-sketched diagrams, in: Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology (UIST '04), ACM Press, New York, NY, USA, 2004, pp. 13–22.

[15] D. Rubine, Specifying gestures by example, Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '91) 25 (4) (1991) 329–337.

[16] M. J. Fonseca, C. Pimentel, J. A. Jorge, CALI: an online scribble recognizer for calligraphic interfaces, in: Papers from the 2002 AAAI Spring Symposium on Sketch Understanding, AAAI Press, Menlo Park, CA, USA, 2002, pp. 51–58.

[17] Q. Chen, J. Grundy, J. Hosking, An e-whiteboard application to support early design-stage sketching of UML diagrams, in: Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments (HCC '03), IEEE Computer Society, Washington, DC, USA, 2003, pp. 219–226.

[18] B. Paulson, P. Rajan, P. Davalos, R. Gutierrez-Osuna, T. Hammond, What!?! no Rubine features?: using geometric-based features to produce normalized confidence values for sketch recognition, in: Workshop on Sketch Tools for Diagramming (VL/HCC '08), 2008, pp. 57–63.

[19] B. Plimmer, I. Freeman, A toolkit approach to sketched diagram recognition, in: Proceedings of the 21st British HCI Group Annual Conference (HCI '07), 2007, pp. 205–213.

[20] J. I. Hong, J. A. Landay, SATIN: a toolkit for informal ink-based applications, in: Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology (UIST '00), ACM Press, New York, NY, USA, 2000, pp. 63–72.

[21] F. Brieler, M. Minas, A new approach to flexible, trainingless sketching, in: Workshop on Visual Modeling for Software Intensive Systems (VL/HCC '05), 2005, pp. 43–50.

[22] F. Brieler, M. Minas, Recognition and processing of hand drawn diagrams using syntactic and semantic analysis, in: Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '08), ACM Press, New York, NY, USA, 2008, pp. 181–188.

[23] F. Brieler, M. Minas, Ambiguity resolution for sketched diagrams by syntax analysis based on graph grammars, in: C. Ermel, J. de Lara, R. Heckel (Eds.), Proceedings of the Graph Transformation and Visual Modeling Techniques (GT-VMT '08), Vol. 10 of Electronic Communications of the EASST, European Association of Software Science and Technology, 2008.

[24] J. A. Jorge, M. J. Fonseca, F. M. G. Pereira, Visual syntax analysis for calligraphic interfaces, in: 13° Encontro Portugues de Computacao Grafica, 2005.

[25] T. Hammond, R. Davis, Tahuti: a geometrical sketch recognition system for UML class diagrams, in: Papers from 2002 AAAI Spring Symposium on Sketch Understanding, 2002, pp. 59–66.

[26] T. Hammond, R. Davis, Automatically transforming symbolic shape descriptions for use in sketch recognition, in: The 19th National Conference on Artificial Intelligence (AAAI '04), AAAI Press, Menlo Park, CA, USA, 2004.

[27] T. Hammond, R. Davis, LADDER, a sketching language for user interface developers, Computers & Graphics 29 (4) (2005) 518–532.

[28] P. Taele, T. Hammond, Hashigo: A next-generation sketch interactive system for Japanese Kanji, in: 21st Innovative Applications Artificial Intelligence Conference (IAAI '09), 2009.

[29] T. A. Hammond, LADDER: a perceptually-based language to simplify sketch recognition user interface development, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA (2007).

[30] G. Costagliola, V. Deufemia, M. Risi, A multi-layer parsing strategy for on-line recognition of hand-drawn diagrams, in: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '06), IEEE Computer Society, Washington, DC, USA, 2006, pp. 103–110.

[31] R. Chung, P. Mirica, B. Plimmer, Inkkit: a generic design tool for the tablet pc, in: Proceedings of the 6th ACM SIGCHI New Zealand Chapter's International Conference on Computer-human Interaction (CHINZ '05), ACM, New York, NY, USA, 2005, pp. 29–30.

[32] L. Yeung, B. Plimmer, B. Lobb, D. Elliffe, Effect of fidelity in diagram presentation, in: Proceedings of the 22nd British HCI Group Annual Conference on HCI 2008 (BCS-HCI '08), British Computer Society, Swinton, UK, 2008, pp. 35–44.

[33] C. Alvarado, R. Davis, SketchREAD: a multi-domain sketch recognition engine, in: Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology (UIST '04), ACM, New York, NY, USA, 2004, pp. 23–32.

[34] A. Caetano, N. Goulart, M. Fonseca, J. Jorge, JavaSketchIt: issues in sketching the look of user interfaces, in: Papers from the 2002 AAAI Spring Symposium on Sketch Understanding, AAAI Press, Menlo Park, CA, USA, 2002, pp. 9–14.

[35] T. M. Sezgin, R. Davis, HMM-based efficient sketch recognition, in: Proceedings of the International Conference on Intelligent User Interfaces (IUI '05), ACM Press, New York, NY, USA, 2005.

[36] B. Paulson, T. Hammond, PaleoSketch: accurate primitive sketch recognition and beautification, in: Proceedings of the 13th International Conference on Intelligent User Interfaces (IUI '08), ACM, New York, NY, USA, 2008, pp. 1–10.

[37] T. Hammond, R. Davis, LADDER: a language to describe drawing, display, and editing in sketch recognition, in: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI '03), 2003, pp. 461–467.

[38] R. Patel, B. Plimmer, J. Grundy, R. Ihaka, Ink features for diagram recognition, in: Proceedings of the 4th Eurographics Workshop on Sketch-based Interfaces and Modeling (SBIM '07), ACM Press, New York, NY, USA, 2007, pp. 131–138.