A Model-Based Recognition Engine for Sketched Diagrams

Florian Brieler, Mark Minas
Universität der Bundeswehr München
Department of Computer Science
85577 Neubiberg, Germany
{florian.brieler|mark.minas}@unibw.de

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 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 diagrams and single components are to be drawn.

In this paper we propose a recognition approach based on models, which is designed for the mentioned issue of clustering and segmentation. All strokes are fed into different models, where each model is responsible for a certain type of primitive, e.g., a line or an arc. The recognition of a component in the drawing is then decomposed into the recognition of its primitives, which can be directly searched for in the models. Finally, the identified primitives are assembled to the complete component.

In several case studies we also show the applicability and generality of our approach, as very different types of components can be recognized. Furthermore, the proposed approach is part of a complete system to sketch understanding which can not only recognize single components, but can also reason about diagrams as a whole, consisting of a set of these components.

1. Introduction

Electronic devices supporting free-hand input become more ubiquitous; recently, Apple released its *iPhone* and the very similar *iPod touch*, both mobile devices which are controlled by a touch-sensitive display. For several years Toshiba is selling and upgrading its *Protégé* series notebooks with can be converted to tablet computers. In industry, many more examples can be observed. Despite this technical evolution, often software for such devices is very basic, using the touch input as a bare replacement for a pointing device like a mouse, neglecting its special characteristics. While processing of hand-written text is quite matured (e.g., the hand-writing recognition module from *Microsoft Windows XP Tablet PC Edition*), processing of hand-drawn sketches is not.

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, and does not need to recognize any other components.

The issue of processing hand-drawn diagrams is twofold. First, the diagram components drawn on the canvas must be recognized; then, analysis of the identified components takes place (cf. Sec. 2). The latter is closely related to regular diagram processing (i.e., not hand-drawn); many matured approaches exist here, and research is still going on. Examples of such approaches are DiaGen [17], DiaMeta [18], AToM^3 [9], and Fujaba [10], many more exist. Despite a large body of research produced so far, the recognition process for hand-drawn diagrams is an open challenge. It can be described by the question "Given a set of strokes drawn on the canvas by the user, what diagram components are represented by these strokes?"

A challenge included in this question is that of *clustering* and *segmentation*. The former means to decide which different strokes must be combined to make one component. Segmentation means quite the opposite, regarding whether the same stroke contributes to more than one component. Clustering and segmentation are a central point for recognition, because their result must be known to decide about the actually drawn components, but it can only be known for sure after the recognition process identified all of those components; a typical chicken-and-egg problem. Examples for clustering and segmentation can be seen in Sec. 4.

There are many approaches to recognition, which can be

VL/HCC Workshop: Sketch Tools for Diagramming
Herrsching am Ammersee, Germany
15 September 2008
Editors: Beryl Plimmer & Tracy Hammond

arranged in several main categories. There are *image-based* approaches, like the one given by Kara [15]. 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.

The largest group of recognizers (regarding references in literature) are *feature-based* approaches. The work of Rubine [20] has to be mentioned here, as it is well-known and served as basis for several similar recognizers enhancing his original concept, e.g., [19]. Typically, feature-based approaches can be easily implemented, and there even exist frameworks providing recognizers ready for use [19, 14], which is a big advantage; sketching systems can be quicker implemented using these recognizers, getting rid of one of the pitfalls of the field. Nevertheless, segmentation and clustering are difficult here, too, 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.

In this paper we propose a different type of recognition. The driving idea was to overcome the mentioned issues. Nevertheless, it is possible to integrate already existing recognizers and related approaches in our recognizer. While useful for image- and feature-based approaches, we do not rely on training the system in any way. Adding support for training is subject to future work.

The remainder of this paper is as follows. Sec. 2 briefly describes the sketch understanding system the proposed recognizer is integrated in. Sec. 3 discusses the recognizer and how it works in detail. Sec. 4 shows the case studies we have performed. Sec. 5 gives related work. Finally, Sec. 6 concludes the paper and describes future work.

2. A sketch understanding system

In [3] we have described a generic sketch understanding system, i.e., it can be tailored to very different diagram types. Our system is no framework, but a full application; tailoring is done by specifications, from which source code is generated. An overview of the architecture of our system is depicted in Fig. 1. Solid arrows denote the control flow, rectangles denote processing units, and rounded boxes denote data.

The user draws a diagram using the *drawing tool*, i.e., the graphical user interface, which generates a set of strokes by capturing the event stream generated by the stylus, and a set of text written on the canvas (see Sec. 3.5). The strokes and the text are 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

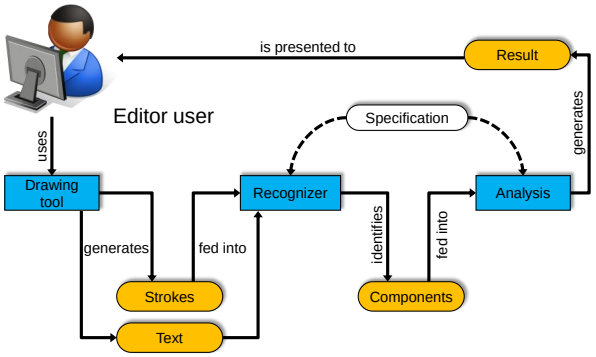

Figure 1. Architecture of our sketch understanding system [3].

components is then *analyzed*. As the components are tried to be added to a complete diagram, syntax and semantics of this diagram can be checked. Details of this process are described in [2]. 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 an usual application, the drawing tool is also equipped with editing capabilities, load and save functions and text input. The latter is currently subject to research in our group.

3. The model-based recognizer

This section describes the recognizer in detail. Its basic idea is to decompose a diagram component into its primitives. Then, these primitives are searched for. If found, the component can be assembled and passed on to the analysis step (see previous section). Decomposition of a component into primitives is also done by other approaches like SkG [7], and Ladder [13] (cf. Sec. 5).

3.1 Specification

A component consists of primitives and constraints on these primitives, e.g., regarding the length of a line, or the angle between two lines. In Fig. 2 the specification of an arrow with an open head and of a rectangle parallel to the axes is given. A graphical representation of the arrow and the rectangle, not showing the constraints, is depicted for clarity.

The arrow consists of three primitives, two for its head, and one for its shaft. Each primitive and each junction of primitives has a name (an identifier), as indicated by `from`, `to` and `id`. The `type` of the three lines means that they can be arbitrarily rotated and do not need to run parallel to the axes. This contrasts to the rectangle; here, we require

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

  <constraint type="fixedlength"
       compare="gt" ids="head,tail"
       length="80" />

  <constraint type="relativelength"
       required="true" compare="lt"
       ids="head,sideB,head,tail" />
  <constraint type="relativelength"
       required="true" compare="lt"
       ids="head,sideA,head,tail" />

  <constraint type="fixedangle"
       ids="sideA,head,head" angle="20"
       required="true" compare="gt" />
  <constraint type="fixedangle"
       ids="sideA,head,head" angle="70"
       required="true" compare="lt" />
  <!-- last two analog for sideB -->
</component>
```

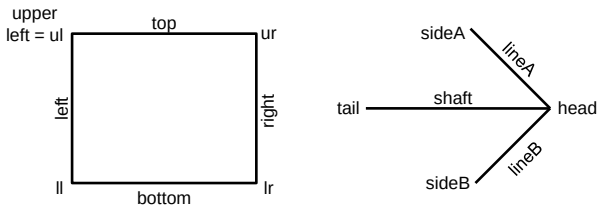

Figure 2. XML specification of an axially parallel rectangle and an arrow, and graphical representations.

the four sides to be either vertical (`up`, `down`) or horizontal (`left`, `right`).

For the arrow we require some further constraints in order to prohibit any three lines meeting in one junction 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 [units]. 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. Finally, the last two constraints require the angles between `sideA` and the shaft to be smaller than 70° and larger than 30°.

Although these two examples use only lines, we currently support three types of primitives: straight lines (as in the examples), arcs, and links. 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 are not specified by a `type`, instead having an attribute identifying the quadrant (1 to 4) where the arc is placed in, and an attribute telling about the sense of rotation. Links, on the other hand, simply connect two junctions 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 Fig. 2, requires the shaft to be straight. We found these three types of primitives sufficient for most domains (cf. Sec. 4). Dashed and dotted primitives are not supported.

3.2 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 virtually any other approach, we regard strokes 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 we do not use the timing information.

Fig. 3 shows a conceptual view of the recognizer, which 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 part of strokes) are meant to represent which primitives. To account for this unconsciousness, each (part of a) 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 *controller* can then query the

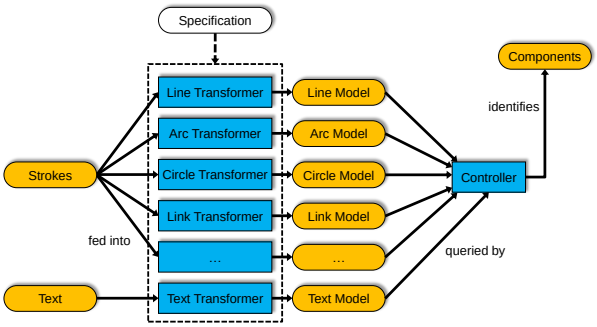

Figure 3. Conceptual view of the recognizer.

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 controller. Therefore, each model has associated a *transformer* which 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. Second, as the transformation takes place, the information is inevitably abstracted, thus enabling faster replies to the queries of the controller.

Additionally, the transformers also know the specification and may decide about which strokes to preprocess and which to discard. This way, the contents of the model can be adapted to the diagram language, and useless information can be discarded as early as possible, thus speeding subsequent processing. Also part of the preprocessing 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.

Until now we implemented five different models. One for each primitive, one especially designed for identifying circles, and one for text. Further models are conceivable and may be added to the system. In the following we discuss each model in detail, and then discuss the actual recognition process. The text model will be discussed separately in Sec. 3.5.

As mentioned before, the transformers low-level preprocess (filter, transform, and abstract) the information represented by the strokes. Fig. 4 shows the original drawing of a rectangle, a circle and an arrow, and graphical representations of the models. Preprocessing can clearly be seen by the bold, black lines, which indicate the data contained in the respective models.

The **line model** tries to interpret the whole drawing as if it consists of straight lines only. Basic vectorization al-

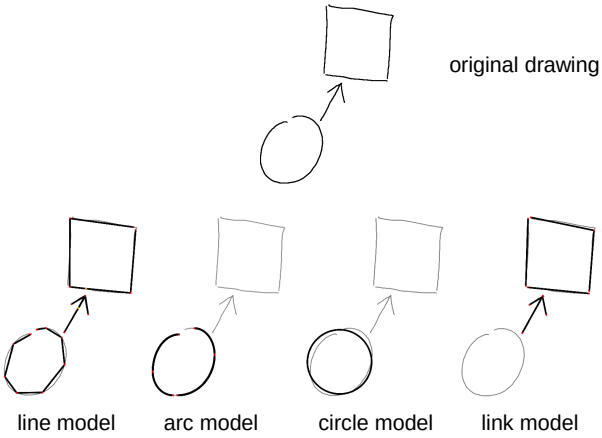

Figure 4. A simple exemplary sketch and graphical representations of the four models.

gorithms can be applied for this purpose. We decided for a very simple one, which proves to work very well and very fast for practical testing. The transformer applies the following steps on each stroke independently:

1. from a stroke each tuple is discarded which has a distance smaller than 5 [units] from its predecessor. This value is a threshold and can be set by the user. As 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 differs no more than 20° from 180°, which is another threshold. Without the first step, this second step fails, as most hardware has a very high sampling rate and usually consecutive samples are 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 controller.

Finally, all lines are split at points of mutual intersection, and information is collected about lines close to each other.

The **arc model** works very similar to the line model. The idea here is to approximate each stroke by quarters of ellipses lying inside quadrants of the coordinate system. Straight lines are dismissed (cf. Fig. 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 sense of rotation changes. Consequently, the resulting sub-lists always turn left or right.

3. based on the direction of the connection between two consecutive points, for each sub-list the quarter ellipses can be obtained.

Unlike the line model and the arc model, the link model and the circle model are more driven by heuristics. For the **circle model** we implemented a feature-based recognizer which 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. 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 span width are calculated.

3. if the span width 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 span width) differs more than 10%, the stroke is dismissed.

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

All strokes which pass this series of tests are then stored as circles in the model, saving the center and the radius. Although the arc model itself contains all information necessary to identify a circle, we decided to add the mentioned circle model, because we found the arc transformer not being reliable enough. Adding an additional model is provided by design: our recognizer approach allows for adding different transformer-model-pairs for the same primitives, such that the overall robustness and reliability is augmented.

Finally, the **link model** regards every stroke as a link, unless its end points are very close to each other, thus the stroke forms some closed shape. Additionally, links may be split at points with a very acute angle. We had to add this mechanism as we observed that, for example, when drawing an open-headed arrow some users tended 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.

As this section shows, the actual 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 model. This means that very much of the research done so far is orthogonal to our approach, and can be seamlessly integrated.

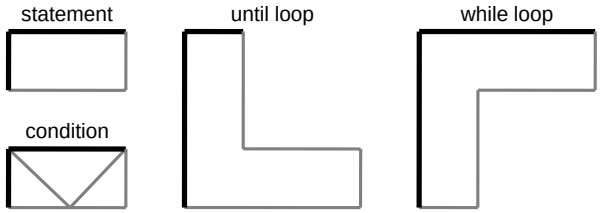

Figure 5. Diagram components of NSD. One possibility for common primitives is indicated by bold strokes.

3.3 Recognition

As mentioned before, the controller searches for primitives of a component one after another, until they are all 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 we rely on a *search plan*, which is precomputed based on the specification. As an example consider *Nassi-Shneiderman diagrams* (NSD). They have a simple syntax with only four different diagram components, as depicted in Fig. 5. Primitives common to all components are, for example, a vertical line which 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.

For NSD, the controller may first search for horizontal straight lines. Then, in the second step, the mentioned vertical line is searched. In the third step, another horizontal line connected to the vertical line at the lower end may be searched, and so on. Primitives are searched for jointly as long as possible, then the search process branches off search 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, we rely on some heuristics; in a greedy fashion, for the next step it is always taken that alternative which preserves the most components for joint search. The search plan is always a tree, and it is not unique in general. Fig. 6 depicts *one possible* search plan for NSD. The bold lines indicate for each node (i.e., step in the search process) that primitive which was recognized and added last. At every leaf the last primitive of a component is identified (indi-

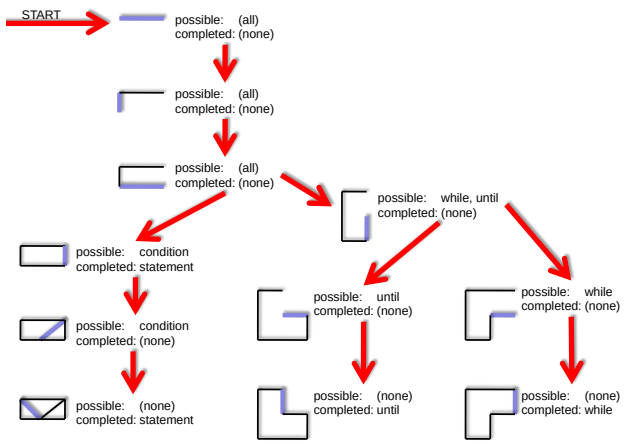

Figure 6. One possible search plan for NSD.

cated by *completed*), and statements are even 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. For the sake of simplicity, no constraints are assumed here.

A noteworthy point 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 which is searched for (the horizontal line in the example) is not restricted regarding its position on the canvas. The next primitive must then be connected to this first one, i.e., must have a common junction. 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 is that there are – in general – much less alternatives for a primitive if one or more of its junctions are already known. The recognition process is speeded. Additionally, required constraints are checked as soon as all necessary junctions and primitives are identified. If not satisfied, no further primitives need to be searched for, and this set of primitives can immediately be discarded. For connecting primitives, of course some threshold is applied, as it lies in the nature of hand-drawing that precision is lacking, and that there are some gaps in drawings of a component which is actually solid. Examples can be seen in Fig. 4; 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.

As it makes no sense to have the same (part of a) stroke belonging to two different primitives of the same component, the controller assures that no part of a stroke is used twice for one component. It can do so because each model includes information about parts of strokes in its replies.

3.4 Eliminating double findings

During testing we identified a typical behavior of the recognizer; the same component is often identified more than once, each time with slightly different junctions. The subsequent analysis step can handle this, but it is also this step which takes more time the more components there are. Accordingly, we have implemented three heuristics to suppress those double findings. One of two identified components of the same *type* (depending on the specification, e.g., statement, condition, while loop, or until loop) is discarded

- if the junctions with the same identifier from the two different components are very close to each other (within a range of some pixels).
- if exactly the same parts of strokes are used for both components, not regarding how they are assigned to different primitives.
- if the *base primitives* of both components use the same parts of strokes. Base primitives are those primitives where *all* junctions are connected to other primitives. For example, an arrow has no base primitives, and a rectangle has only base primitives.

We equip each identified component with a rating, which depends on how precisely each primitive is drawn, on how close the connections at the junctions are, and on how good the constraints are met. Whenever one of two components has to be discarded, that one is chosen which has the lower rating.

3.5 Text

As depicted in Fig. 3, text is also represented as a model, and can be queried by the controller. After all primitives of a component are identified, the controller constructs the regions where text may be added. The necessary information again comes from the specification. The controller then queries the text model and adds the respective texts to the components. Just like other primitives, text can be specified as required. In case that some text is required, but not present, the component is dismissed. Additionally, text can be checked against regular expressions to allow for some basic filtering.

To distinguish text from graphics, we do not rely on a *divider* (a piece of software can perform this determination), although [19] reports some advances. Instead, the drawing tool requires the user to explicitly indicate input of text. The input is then directly transformed to a string representation, which is fed into the text transformer.

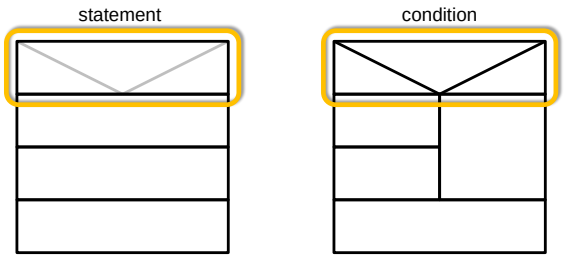

Figure 7. An example of ambiguity resolution by use of context information. The light gray lines on the left hand side are regarded as wrong.

3.6 Analysis

Subsequent to the recognizer, the analysis step is applied. At this point a common source of ambiguity can be discussed. As the search plan (Fig. 6) clearly shows, there is identified a statement for each identified condition. The system recognizes this ambiguity, but cannot decide for one component 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, which would clearly point out what choice is the right one. An example is given in Fig. 7. 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, the mentioned resolution is done by the analysis, which happens subsequent to the recognition of components. Apart from this example, analysis is not discussed in this paper, but in [2].

4. Case Studies

This section describes the different examples we have implemented with our system so far. For each case study, its visuals are described briefly, and an example drawing is given. All of these examples are recognized correctly. Sec. 4.6 reports 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 speed of our implementation. A user study which investigates recognition rates for a large body of participants is subject to future work.

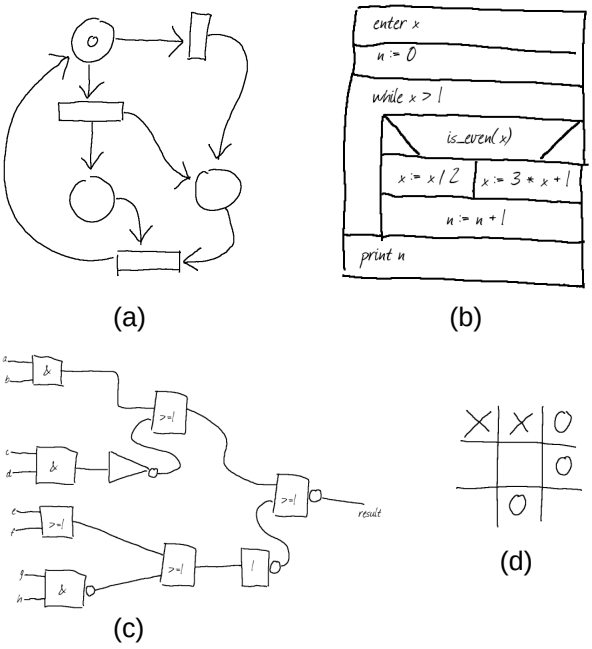

Figure 8. Examples for the case studies. (a) Petri net, (b) NSD, (c) logic gate, (d) tic-tac-toe.

4.1 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 depicted as circles and may contain one or more *tokens*. Although tokens are commonly depicted as small filled circles, we omit the filling and draw tokens as circles, too. The analysis step (cf. Sec. 2) reliably detects the difference. Finally there are *transitions*, which we depict 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. Fig. 8 (a) shows a simple Petri net consisting of three places, three transitions, one token, and 8 arrows.

4.2 Nassi-Schneiderman diagrams

NSD are used to visualize structured programs. Its four different types of components are depicted in Fig. 5 (actually NSD are more powerful and have more components, but for the sake of simplicity we assume this subset). An algorithm for computing the Collatz sequence is given in Fig. 8 (b). It consists of 6 statements, one loop, and one condition.

The mentioned figure also shows clustering and segmentation. The left vertical line spanning from the first state-

ment 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 has to cluster lines also.

4.3 Logic gates

Common boolean logic can also be expressed graphically. As operators we assume `and`, `or` and `not`. All of them are drawn as rectangles, input on the left, output on the right. `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, which 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. Fig. 8 (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.

4.4 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 we have chosen tic-tac-toe, because it is possible to give suitable rules for the analysis step here, which is, as mentioned before, crucial to our approach. Using the analysis step, we can also do some reasoning about game situations (because our system does not allow for interactivity, you cannot actually play). For the situation depicted in Fig. 8 (d) the system correctly tells that it is player X's turn.

4.5 GUI builder

Another example not regarding a traditional diagramming language is a 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 Fig. 9. This example also shows all widgets which we support: 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.

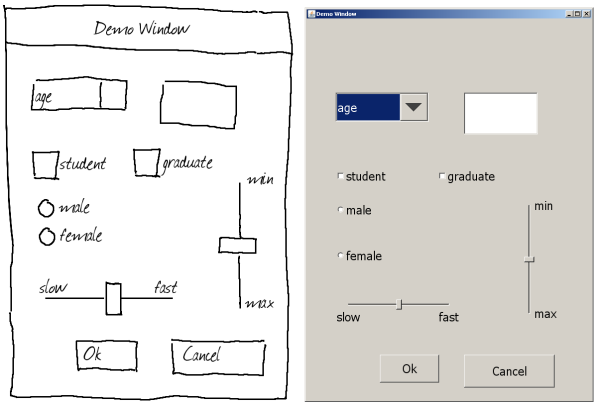

Figure 9. Example for the GUI builder. The right hand side shows the generated window.

4.6 Discussion

To evaluate the performance of our implementation we used the examples from above. For each case study we measured the time for recognition (skipping the time for analysis), using one, four, 7 and 10 copies of the respective example, thus linearly increasing the load. We repeated each setting 10 times and used the lowest value (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.

The result is depicted in Fig. 10. In all cases the time for processing seems to increase roughly linear with the input. However, some aspects like matching text to components is actually not linear, but worse. As the figure shows, the impact of these aspects is very little. The average time to recognize *one actually drawn component* ranges from about 2msec for logic gates to about 16msec for Tic-tac-toe and NSD. We believe that the small figure for logic gates is due to the large number of connections between the operators, which are made by only one link each, and can thus be recognized very fast. On the contrary, for Tic-tac-toe, and for NSD, the components are more complex, thus taking more time to be recognized; the board for Tic-tac-toe consists of 12 primitives, for example.

For NSD we observed that the number of false positives is very high due to the visual appearance of the single components. For example, the recognizer identifies for each two consecutive statements a third one, omitting the horizontal line dissecting the two. This issue, and a larger exemplary diagram consisting of 10 statements, three loops and two conditions, are discussed in more detail in [3]. Here, the recognizer identifies 56 components which would be passed to the analysis step. As an optimization we added an option to the recognizer, allowing for dismissing a component if it

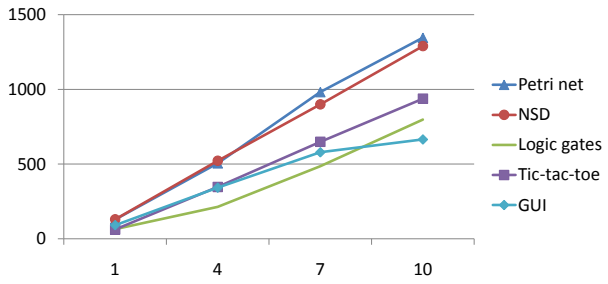

Figure 10. Runtime measurements for the five case studies for a linear increase in load. The vertical axis shows time in milliseconds, the horizontal axis shows the number of copies of the original diagrams from Fig. 8 and Fig. 9.

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 from 2.5 seconds to 0.7 seconds. The correct result was obtained anyway. The measurements reported in Fig. 10 also make use of this option. The drawback is that the error tolerance is reduced. In case of a false positive inside a component, this component is dismissed, although it may be correct.

Similar to NSD, we face a lot of false positives 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 step reliably selects the intended components. In the depicted case in Fig. 9, 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.

5. Related work

Conceptually very similar to our approach is that of Hammond and Davis, LADDER [12, 13]. The details are different, tough. Their description language allows specifying the same as we do, but additionally, editing behavior and more sophisticated constraints may be given. What we referred to as `required` constraints is distinguished as soft and hard constraints. Components may be defined hierarchically, using abstract components. Regarding recognition, they use a rule-based approach. Strokes are classified as primitives, and added as facts. Recognition of components is then represented as rules about the facts. Ambiguity on the level of components is solved by meta-rules, an analysis step as we have is not provided.

Also closely related to our work is the approach by Costagliola et al. [7, 8]. Domain-specific recognizers are generated from grammars in the SkG-formalism. 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 [14]. At the next level, the primitives are grouped into (partial) symbols, each having an *importance rate*. Based on this rate and the context, at the highest level it is finally decided for interpretations, and conflicting partial symbols may be pruned. The full system works incrementally. The reported recognition rates typically exceed 90%, with some exceptions. As a comparison, recognition rates are also taken without disambiguitation, which clearly reduces recognition rates.

From the same group as LADDER there are also other approaches available. [21] uses Hidden Markov Models, for example. Their recognizer takes into account the specific drawing styles of individual users. Reported recognition rates and run times for the recognizer are very good. [1] uses a Bayes net to reason about diagrams. The approach relies on three stages; first, there are hypotheses generated from the input strokes. Second, using the Bayes net it is determined how well these hypotheses fit the data. Third, further hypotheses are generated from the result of stage two. Using this approach it is even possible to recognize components only drawn partially, and context of a component may be considered.

Casella et al. [5, 6] propose a conceptually interesting generic framework for sketch understanding based on agents. They can include arbitrary symbol recognizers, but their organization is quite different from our approach. There is an individual agent for each available component type, called an SRA (*symbol recognition agent*). The SRAs autonomously search for components, but may communicate which each other to exchange context information. A central agent may solve conflicts between the SRAs. Use case diagrams from the UML serve as an example. Using a Java-based implementation, good recognition rates are reported. Similar to our findings, the result is clearly improved by use of context information.

The low-level framework *Cali* is based on features [11]. Segmentation cannot be performed, clustering is done automatically if strokes are drawn quickly one after another. All features calculated 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.) very reliable and very quick, 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 [4], and application for drawing user interfaces.

6. Conclusions and future work

In this paper we have proposed an alternative to conventional recognizers, which are based on image-processing techniques, or based on features. Our approach solves the issue of clustering and segmentation, and allows for integrating previous work due to its model-based concept. Several 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.

As future work we plan to include more primitives and respective transformer-model-pairs. Currently we are 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.

The integration of mechanisms well-known and understood for other types of recognizers is a completely open issue, such as training, or mediation techniques [16].

A thorough evaluation of our approach in terms of recognition rate by a user study is also necessary. From our experience gained so far we have learned that components not identified by the recognizer are always due to the models giving improper answers to the queries from the controller. Hence, the overall concept seems very reasonable and reliable, while the transformers miss some information sometimes. The evaluation will have to clarify this assumption.

References

[1] C. Alvarado and R. Davis. Dynamically constructed bayes nets for multi-domain sketch understanding. In *Proceedings of IJCAI-05*, pages 1407–1412, San Francisco, California, August 2005.

[2] F. Brieler and M. Minas. Ambiguity Resolution for Sketched Diagrams by Syntax Analysis Based on Graph Grammars. In C. Ermel, J. de Lara, and R. Heckel, editors, *Proc. GT-VMT '08*, volume 10 of *Electronic Communications of the EASST*. European Association of Software Science and Technology, 2008.

[3] F. Brieler and M. Minas. Recognition and processing of hand drawn diagrams using syntactic and semantic analysis. In *Proc. AVI '08*, pages 181–188. ACM Press, May 2008.

[4] A. Caetano, N. Goulart, M. Fonseca, and J. Jorge. Javasketchit: Issues in sketching the look of user interfaces. In *Proc. 2002 AAAI Spring Symposium - Sketch Understanding*, pages 9–14. AAAI Press, Menlo Park, 2002.

[5] G. Casella, G. Costagliola, V. Deufemia, M. Martelli, and V. Mascardi. An agent-based framework for context-driven interpretation of symbols in diagrammatic sketches. In *Proc. VL/HCC '06*, pages 73–80, Washington, DC, USA, 2006. IEEE Computer Society.

[6] G. Casella, V. Deufemia, and V. Mascardi. A multi-agent system for hand-drawn diagram recognition. *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, 2:739–743, September 2007.

[7] G. Costagliola, V. Deufemia, and M. Risi. Sketch grammars: A formalism for describing and recognizing diagrammatic sketch languages. In *ICDAR '05: Proceedings of the Eighth International Conference on Document Analysis and Recognition*, pages 1226–1231, Washington, DC, USA, 2005. IEEE Computer Society.

[8] G. Costagliola, V. Deufemia, and M. Risi. A multi-layer parsing strategy for on-line recognition of hand-drawn diagrams. In *Proc. VL/HCC '06*, pages 103–110, Washington, DC, USA, 2006. IEEE Computer Society.

[9] J. de Lara and H. Vangheluwe. Atom3: A tool for multi-formalism and meta-modelling. In *Proc. FASE '02*, pages 174–188, London, UK, 2002. Springer.

[10] T. Fischer, J. Niere, L. Turunski, and A. Zündorf. Story diagrams: A new graph grammar language based on the Unified Modelling Language and Java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Theory and Application of Graph Transformation (TAGT'98), Selected Papers*, number 1764, pages 296–309. Springer, 2000.

[11] M. J. Fonseca, C. Pimentel, and J. A. Jorge. Cali: An online scribble recognizer for calligraphic interfaces. In *Proc. 2002 AAAI Spring Symposium - Sketch Understanding*, pages 51–58, 2002.

[12] T. Hammond and R. Davis. Automatically transforming symbolic shape descriptions for use in sketch recognition. In *The 10th National Conference on Artificial Intelligence*. AAAI, July 2004.

[13] T. Hammond and R. Davis. Ladder, a sketching language for user interface developers. *Computers & Graphics*, 29(4):518–532, 2005.

[14] J. I. Hong and J. A. Landay. Satin: a toolkit for informal ink-based applications. In *Proc. UIST '00*, pages 63–72. ACM, November 2000.

[15] L. B. Kara and T. F. Stahovich. Hierarchical parsing and recognition of hand-sketched diagrams. In *Proc. UIST '04*, pages 13–22, New York, NY, USA, 2004. ACM.

[16] J. Mankoff, S. E. Hudson, and G. D. Abowd. Interaction techniques for ambiguity resolution in recognition-based interfaces. In *Proc. UIST '00*, pages 11–20, New York, NY, USA, 2000. ACM Press.

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

[18] M. Minas. Generating meta-model-based freehand editors. In *Electronic Communications of the EASST, Proc. GraBaTs '06*, September 2006.

[19] B. Plimmer and I. Freeman. A toolkit approach to sketched diagram recognition. In *Proceedings of the 21st British HCI Group Annual Conference*, pages 205–213, September 2007.

[20] D. Rubine. Specifying gestures by example. *SIGGRAPH Comput. Graph.*, 25(4):329–337, 1991.

[21] T. M. Sezgin and R. Davis. Hmm-based efficient sketch recognition. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI'05)*. ACM Press, January 2005.