Sketch2Process: End-to-end BPMN Sketch Recognition Based on Neural Networks

Bernhard Schäfer, Han van der Aa, Henrik Leopold, and Heiner Stuckenschmidt

Abstract—Process models play an important role in various software engineering contexts. Among others, they are used to capture business-related requirements and provide the basis for the development of process-oriented applications in low-code/no-code settings. To support modelers in creating, checking, and maintaining process models, dedicated tools are available. While these tools are generally considered as indispensable to capture process models for their later use, the initial version of a process model is often sketched on a whiteboard or a piece of paper. This has been found to have great advantages, especially with respect to communication and collaboration. It, however, also creates the need to subsequently transform the model sketch into a digital counterpart that can be further processed by modeling and analysis tools. Therefore, to automate this task, various so-called sketch recognition approaches have been defined in the past. Yet, these existing approaches are too limited for use in practice, since they, for instance, require sketches to be created on a digital device or do not address the recognition of edges or textual labels. Against this background, we use this paper to introduce Sketch2Process, the first end-to-end sketch recognition approach for process models captured using BPMN. Sketch2Process uses a neural network-based architecture to recognize the shapes, edges, and textual labels of highly expressive process models, covering 25 types of BPMN elements. To train and evaluate our approach, we created a dataset consisting of 704 hand-drawn and manually annotated BPMN models. Our experiments demonstrate that our approach is highly accurate and consistently outperforms the state of the art.

Index Terms—Requirements engineering, business process modeling, graphics recognition and interpretation.

1 Introduction

Process models are key artifacts in various software engineering contexts. For instance, they are a commonly used means to capture business-related requirements [1], [2], [3] and also provide the basis for the development of process-oriented applications in low-code/no-code settings [4], [5], such as solutions for workflow orchestration and Robotic Process Automation (RPA).

To support modelers in creating, checking, and maintaining process models, dedicated modeling tools are available. Although these tools are generally considered as indispensable to capture process models for their later use, it is important to note that the creation of conceptual models often starts by sketching on a whiteboard or paper [6]. One of the main reasons for this is that modeling tools do not provide the means for effective communication and collaboration that are required for creating these models [7]. In this regard, drawing on a whiteboard or paper has been found to have great advantages. They are not only ubiquitous and easy to use [8], but also immediate [9]. This aspect of immediacy is of great importance, since it allows people involved in the model creation to continue their thought process or conversation without interruption [9].

However, starting with a hand-drawn model introduces the need to subsequently transform it into a digital counterpart that can be further processed by modeling and analysis tools [10]. If done manually, this transformation takes considerable time and effort, creating undesirable friction in the modeling process. This friction may also drive users away from sketching initial model versions in an analog manner, choosing instead to use collaborative modeling tools [11], [12]. Such tools enable modelers to jointly create models in a shared workspace, removing the need to conduct a manual transformation from an analog to a digital format. Nonetheless, these tools do not allow users to freely sketch processes, since they require modelers to stick to predefined constructs and functionality. They thus take away the aforementioned mentioned benefits of drawing models by hand.

Recognizing the benefits of hand-drawn models, as well as the effort associated with their manual transformation into digital counterparts, *sketch recognition* aims to automate this transformation task. Sketch recognition approaches have been defined for various diagrams, including UML models [13], flowcharts [14], and different kinds of process models [15], [16], [17]. Existing approaches, however, often have limiting requirements and assumptions. Some require sketches to be created on a digital device [13], [14], [15], whereas others are limited to the recognition of shapes, without being able to handle edges or textual labels [16], [17].

Against this background, we use this paper to introduce the first end-to-end sketch recognition approach for process models captured using the Business Process Model and Notation (BPMN)[1]. BPMN, a standard maintained by the Object Management Group (OMG)[2], is a flow-like notation that provides a rich set of graphical symbols, allowing users to specify which tasks should be executed in a process, by

- *Bernhard Schäfer, Han van der Aa, and Heiner Stuckenschmidt are with the Data and Web Science Group, University of Mannheim, Mannheim, Germany.*
  E-mail: {bernhard | han | heiner}@informatik.uni-mannheim.de
- *Henrik Leopold is with the Kühne Logistics University, Hamburg, Germany.*
  E-mail: henrik.leopold@the-klu.org

1. https://www.bpmn.org/
2. https://www.omg.org/

whom, using which data, and in which order. Our work focuses on these models because, on the one hand, they are commonly used by business users in software engineering contexts [2], [3], [18] and closely resemble the notations used by various low-code/no-code platforms [19], [20], such as the RPA solutions by *Microsoft*[3], *Appian*[4], or Mendix[5] (see [21] for a comprehensive overview). On the other hand, our choice is motivated by the conceptual complexity of the recognition task for BPMN models, owing to the notation's large number and high similarity of node types, which makes it more relevant to develop a tailored and accurate recognition approach.

As such, the primary contribution of our work is `Sketch2Process`, an approach that takes an image of a hand-drawn BPMN model as input and automatically transforms it into a BPMN XML file, compatible with existing process modeling tools. Conceptually, we build on a neural network-based architecture inspired by state-of-the-art work from the area of flowchart recognition [22]. In this manner, `Sketch2Process` considerably improves upon existing works in terms of both recognition quality and scope. It provides comprehensive transformation of hand-drawn BPMN models, including the proper handling of textual labels that were previously ignored. As a secondary contribution, we developed the `hdBPMN` dataset consisting of 704 hand-drawn and manually annotated BPMN models, which we here use to train and evaluate our approach. `hdBPMN` is publicly available and, therefore, can be used as a basis for the development and comparison of further transformation approaches. Experiments on this dataset demonstrate that `Sketch2Process` is highly accurate and consistently beats the state of the art.

The work presented in this manuscript represents a considerable extension in both scope and quality of our earlier work on Sketch2BPMN [23], a first approach for the (partial) recognition of hand-drawn BPMN models. In particular, we provide the following main extensions:

1) Aside from recognizing nodes and edges in hand-drawn BPMN models, `Sketch2Process` also targets the detection, recognition, and relation of textual labels to their respective nodes and edges. This makes `Sketch2Process` the first approach that provides end-to-end recognition of hand-drawn BPMN models.
2) We fully revised the component for edge relation detection by replacing Sketch2BPMN's heuristic detection technique with a neural network-based component. The new detection component considerably outperforms the previous version, particularly for the detection of more challenging hand-drawn edges, such as message flows.
3) We improved the overall recognition quality of our approach by incorporating a *crop augmentation procedure* during its training. This procedure allows our approach to handle model layouts not encountered in the training set, leading to better generalization.
4) Finally, we substantially extended the `hdBPMN` dataset. Specifically, we manually annotated all labels and individual words in the hdBPMN dataset and also increased the dataset size from 502 to 704 images. Overall, the dataset now contains more than 70,000 annotated elements, versus 20,000 annotations in the previous version.

The remainder is organized as follows. We provide an overview of existing sketch recognition approaches in requirements engineering in Section 2 and elaborate on the challenges of sketch recognition for BPMN models in Section 3. Section 4 presents our `Sketch2Process` approach. In Section 5, we introduce the `hdBPMN` dataset, which is used to evaluate our approach in Section 6. Finally, we discuss the implications and limitations of our work in Section 7 and conclude in Section 8.

2 Related Work

Freely sketching by hand is a natural and powerful way to convey information to others [24]. Given the advantages of free-hand sketching, a range of sketch recognition approaches have been proposed [25], which aim to transform such sketches into usable conceptual models. With early approaches dating back to the sixties [26], today, sketch recognition approaches are available for various types of diagrams [15], [27], [28], user interfaces [29], [30], and mechanical systems [31], [32].

Several of these approaches explicitly target the recognition of sketched diagrams in the context of requirements engineering, due to the wide-spread use of diagrams for requirements elicitation and documentation. We provide an overview of such works in Table 1. For each approach, we show the targeted artifact(s) and its recognition scope with respect to shapes, edges, and the labels of the diagram. For the labels, we also indicate whether the respective approach targets 1) *textblock detection*, i.e., identifies which parts of the image contain text, 2) *textblock handwriting recognition*, i.e., converts the handwritten text contained in a textblock into a digital counterpart, and 3) *textblock relation detection*, i.e., relates the textual label to a diagram shape or edge. In line with convention, we differentiate between online and offline sketch recognition in the remainder.

Online sketch recognition. Online approaches for sketch recognition require a sequence of hand-drawn strokes as input [14]. A stroke in this context is defined as a sequence of points that occur between pen-down and pen-up events. Typically, such input can only be provided by digital devices, such as tablets or smart boards, which limits the application of online sketch recognition approaches to these devices.

Existing approaches for online sketch recognition can be subdivided into geometry-based, stroke-based, and gesture-based approaches. In *geometry-based approaches*, higher-level shapes are recognized as a particular combination of strokes, using constraints to specify how strokes and subshapes fit together [25]. While many geometry-based approaches only target basic geometric shapes, such as circles, triangles, and squares [24], [25], [33], Brieler and Minas [34] build on these base approaches to recognize sketches of Petri nets and Nassi–Shneiderman diagrams. *Stroke-based approaches* pursue different strategies to group and classify the input stroke data and, in this way, recognize the drawn shapes and diagrams. For instance, the approach from Brieler and Minas [15] for

3. https://powerautomate.microsoft.com/en-us/robotic-process-automation/
4. https://appian.com/platform/complete-automation/robotic-process-automation-rpa.html
5. https://www.mendix.com/workflow/low-code-automation/

TABLE 1
Related Work

Category / Targeted artifact(s)	Name	Authors	Shapes	Edges	Labels TB-D*	Labels TB-HWR*	Labels TB-RD*
Online sketch recognition							
Geometry-based approaches							
Generic geometric shapes	-	Yu & Cai [24]	Yes	No	No	No	No
Generic geometric shapes	PaleoSketch	Paulson & Hammond [33]	Yes	No	No	No	No
Generic geometric shapes	-	Hammond & Paulson [25]	Yes	No	No	No	No
Petri nets, Nassi–Shneiderman diagram	DSketch	Brieler & Minas [34]	Yes	Yes	No	No	No
Stroke-based approaches							
Petri nets, Nassi–Shneiderman diagram	-	Brieler & Minas [15]	Yes	Yes	No	No	No
UML	AgentSketch	Casella et al. [13]	Yes	Yes	No	No	No
UML	-	Deufemia et al. [35]	Yes	Yes	No	No	No
Flowcharts, Finite automata	-	Bresler et al. [14]	Yes	Yes	Yes	No	Yes
Flowcharts	-	Julca-Aguilar et al. [36]	Yes	Yes	No	No	No
Gesture-based approaches							
UML, UI interfaces	SkApp	Schmidt & Weber [29]	Yes	Yes	No	No	No
UML	SUMLOW	Chen et al. [27]	Yes	Yes	No	Yes	No
Offline sketch recognition							
Stroke-based approaches							
Flowcharts	-	Costagliola et al. [37]	Yes	Yes	No	No	No
Flowcharts	-	Wu et al. [38]	Yes	Yes	No	No	No
Flowcharts	-	Bresler et al. [39]	Yes	Yes	Yes	No	No
EPC process models	-	Zapp et al. [17]	Yes	No	No	No	No
Object-based approaches							
Flowcharts	-	Julca-Aguilar and Hirata [28]	Yes	No	Yes	No	No
Flowcharts, Finite automata	Arrow R-CNN	Schäfer et al. [22]	Yes	Yes	Yes	No	No
BPMN models	Sketch2BPMN	Schäfer et al. [23]	Yes	Yes	No	No	No
BPMN models, Flowcharts	DiagramNet	Schäfer et al. [16]	Yes	Yes	No	No	No

*TB-D = Textblock detection, TB-HWR = Textblock handwriting recognition, TB-RD = Textblock relation detection.

Petri nets and Nassi–Shneiderman diagrams builds on a set of, so-called, transformers. Each transformer tries to interpret a given set of strokes in a different way (e.g. as a line, as an arc, or as a circle) and hands over the result to an assembly module, which identifies the drawn component based on the transformer input. The approach from Deufemia et al. [35] for UML class diagrams first segments the user's strokes and interprets them as primitive shapes before it exploits the domain context to cluster them into respective symbols of the target language. Similar approaches have been proposed for flowcharts [14] and for flowcharts and finite automata [36]. The main challenge for stroke-based approaches is that the way users draw shapes varies with respect to many dimensions including stroke order, stroke style, and stroke number. *Gesture-based recognition approaches* are similar to stroke-based approaches but additionally recognize definable gestures. For instance, SkApp [29] allows users to specify multi-touch gestures to efficiently create UML class diagrams and user interfaces. Similarly, SUMLOW [27], an approach for e-whiteboards, recognizes single and multi-stroke gestures for the specification of different types of UML diagrams.

As for the scope of online sketch recognition approaches, we can see that only two approaches address labels. First, the approach by Bresler et al. [14] provides basic support for textblock detection and textblock relation detection, though textblock handwriting recognition is not addressed. The detection of textblocks is accomplished with a text/non-text stroke separation algorithm. In case text-related strokes are located inside a shape, the stroke is related to the respective shape, otherwise they are grouped into textblocks based on spatiotemporal proximity and then assigned to the closest arrow. Second, SUMLOW from Chen et al. [27] defines gestures for handwriting recognition. However, since the approach is based on the interaction between user and e-whiteboard, there is no actual mechanism for textblock detection or textblock relation detection.

Offline sketch recognition. In comparison to online approaches, offline sketch recognition just requires an image as input, rather than depending on information on the sequence of strokes used to obtain the sketch. This makes offline recognition more widely applicable but also more complex, given that less data is available as input. Existing approaches for offline sketch recognition can be further subdivided into stroke-based and object-based approaches. *Stroke-based* offline approaches require that the strokes can be reliably reconstructed from a given image. Based on the reconstructed strokes, they then apply online recognition methods. Respective approaches have been proposed for flowcharts [37], [38], [39] and EPC process models [17]. It is important to point out stroke-based offline approaches are not applicable in the setting we address in this paper. Effective methods for the reconstruction of strokes from camera-based images of pen-and-paper drawings with complex backgrounds are simply not available [40], [41]. More recent *object-based* approaches use deep learning object detectors to detect diagram shapes, arrows, and textblocks in an image through, so-called, bounding boxes. Julca-Aguilar and Hirata [28] were the first to use the popular Faster R-CNN [42] object detection network to recognize flowchart elements. However, while object detectors can localize arrows through bounding boxes, they are not able to recognize the edges between shapes. Given this limitation, Schäfer et al.

proposed Arrow R-CNN [22], [43], which extends Faster R-CNN with a heuristics-based edge recognition component. Building on Arrow R-CNN, we proposed Sketch2BPMN [23] and DiagramNet [16] in earlier work, both targeting BPMN models. Sketch2BPMN [23] explicitly addresses the specifics of hand-drawn BPMN models. It, however, builds on a rule-based component for the identification of edges, which affects both the performance as well as the flexibility of the approach. DiagramNet [16] attempts to recognize edges using a learning-based approach. It, however, assumes that edges are mostly located in the area between two shapes, which means that it has difficulties recognizing edges that connect two shapes with a detour. A related approach that is not included in Table 1 is the BPMN-Redrawer approach [44] because it originally only addresses the recognition of *computer-generated* BPMN models. The approach consists of two off-the-shelf networks from the Detectron2 library [45], which are trained on a dataset of computer-generated BPMN model images. However, given this architecture, the approach could also be trained to recognize hand-drawn models.

As for the scope, it is again important to point out that the recognition of labels is hardly considered. In fact, there is no offline sketch recognition approach available that goes beyond textblock detection. This is highly problematic, since the overall conveyed semantics of all artifacts discussed above heavily relies on labels [46]. From a practical point of view, existing approaches are, therefore, only of limited use.

In summary, the review above shows that sketch recognition for requirements engineering is a highly active field of research, in which Sketch2BPMN [23] and DiagramNet [16] also specifically target the recognition of hand-drawn BPMN models. However, these approaches leave substantial gaps in terms of recognition quality and coverage, wholly omitting the consideration of textual labels. As described in Section 1, `Sketch2Process` addresses the limitations of the state of the art by:

1) addressing all three aspects of label recognition, making it the first approach that provides end-to-end recognition of hand-drawn BPMN models from images,
2) incorporating a neural network-based edge relation detection component that outperforms previous works considerably, especially for complex edges, and
3) improving the overall recognition quality through a crop augmentation procedure, which allows our approach to handle model layouts not encountered during training, leading to better generalization.

In this manner, `Sketch2Process` thus considerably improves upon the state of the art in terms of both scope and accuracy. As such, our approach is well-equipped to tackle the challenges described next.

3 Challenges of Hand-drawn BPMN Model Recognition

This section illustrates the challenges associated with the recognition of hand-drawn BPMN models. We discuss challenges specifically related to shapes, edges, and labels, as well as general challenges that occur when dealing with images of physical drawings. We will use to the exemplary drawing in Fig. 1, stemming from our `hdBPMN` dataset, to illustrate the challenges where applicable.

Shape recognition challenges. Shape recognition targets the identification of the nodes in a BPMN model, such as activities, events, gateways, resource pools, and data objects. From a recognition perspective, shapes are defined through a *bounding box*, capturing the location of the shape in the drawing, and a *shape type*, capturing its type. Compared to the recognition of other types of conceptual models (see Section 2), BPMN models have a considerably higher number of different node types, which increases the complexity of the recognition task.

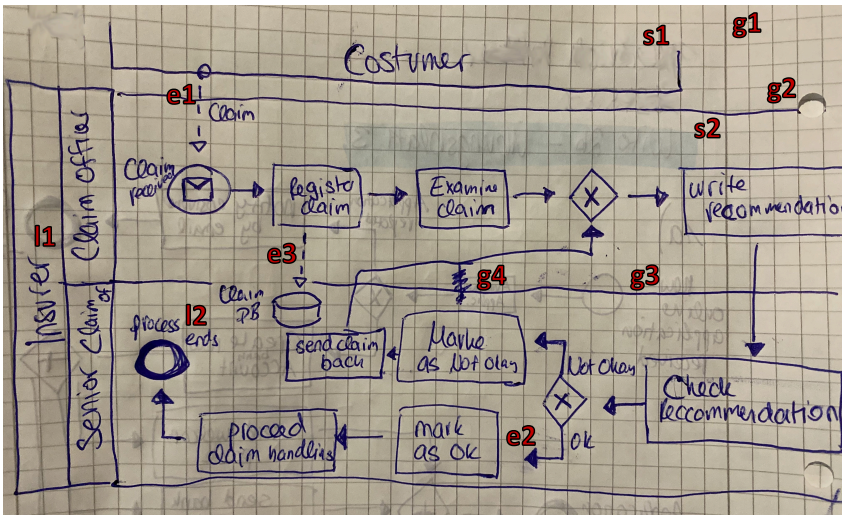
The recognition of shapes and their types in hand-drawn diagrams can be highly complex due to a variety of challenges. From a conceptual point of view, shape recognition in BPMN is complex because of the high similarity between certain types. Clear examples of this are events, which are depicted as circles, where the circle's line determines whether it is a start (single line), intermediate (double), or end (bold) event of a certain type. Similarly, activities, sub-processes, pools, and lanes are all depicted as rectangles, where the specific node type follows from the node's context, e.g., a lane encompassing several activities, or subtle differences, e.g., a small *plus* symbol in a square indicates that a node represents a sub-process.

While challenging in itself, differentiating among such similar node types becomes more complex due to the characteristics of hand-drawn models, such as drawn lines being incomplete, curved (when they should be straight), or interrupted, such as e.g., seen for issues s1 and s2 in Fig. 1. Furthermore, shapes in general are drawn in a broad range of styles, especially more complex ones such as events, databases, and certain gateways. This is, for instance, evidenced by the examples of intermediate (throwing) message events depicted in Fig. 2.

Edge recognition challenges. Edges in BPMN models indicate connections between nodes. Three edge types exist, which have different drawing styles and are used to connected different node types. Solid edges indicate the *control flow* of a process by connecting nodes such as activities, events, and gateways, as e.g., seen in Fig. 1 to indicate that the *Examine claim* activity occurs after *Register claim*. Dashed edges capture the *message flow* across organizational boundaries, as seen by the *Customer* sending a claim to the *Insurer* in the example. Finally, dotted edges capture *data associations*, showing the creation or retrieval of data, such as *Register claim* storing information in the *Claim DB*.

Each edge is defined through a sequence of waypoints (indicating the path of the edge), the edge type, and the source and target shape that the edge connects. Properly recognizing edges and their characteristics in an automated manner is complex, though. This complexity, for instance, results from edges commonly crossing each other or intersecting with other model elements, as e.g., seen for issue e1 in Fig. 1, where just a single dash appears before the edge intersects with the *Insurer* pool. Such edge interruptions make it hard for a recognition approach to identify which drawn lines belong to the same edge and which lines are actually separate ones.

Furthermore, drawn edges are often not properly connected to model nodes, as e.g., seen for issue e2 in



Depicted shape recognition issues:
- s1) Shape drawn incompletely;
- s2) Shape drawn using curved and interrupted lines.

Depicted edge recognition issues
- e1) Edge interrupted by other model element;
- e2) Edge not connected to corresponding nodes;
- e3) Data association drawn using dashes rather than dots.

Depicted label recognition issues
- l1) Rotated pool and lane labels;
- l2) Textblock words that appear disconnected.

Depicted general issues:
- g1) Additional lines due to paper type;
- g2) Punch-holes in paper obscuring the model;
- g3) Contents on other side of paper bleeding through;
- g4) Crossed out part of the drawing.

Fig. 1. Example of a hand-drawn BPMN model with various highlighted recognition challenges.

Fig. 2. Variations of intermediate throwing message events.

Fig. 1, which makes it harder to recognize the source and target of an edge. This recognition is particularly complex when there are multiple candidate shapes, such as seen in Fig. 3, where two pool boundaries and a task are very close to the startpoint of a message flow. Finally, the differentiation of message flows and data associations can be difficult since they are often drawn in a similar or even equal manner. For example, in Fig. 1 we observe that also data associations are drawn using dashes, rather than dots (see issue e3).

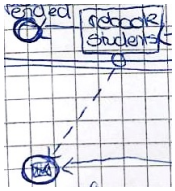


Fig. 3. Multiple edge candidates

Label recognition challenges. Labels are associated with shapes and edges in BPMN models. Some types, e.g., activities, require a label, whereas for others, e.g., control-flow edges, labels are optional. Each label, in contrast, should be associated with a specific shape or edge. As described in Section 2, label recognition can be decomposed into a sequence of three steps.

Textblock detection strives to locate the labels in BPMN models through bounding boxes. These boxes are referred to as *textblocks* in diagram recognition [14], [39]. The primary challenge here is to appropriately recognize which pieces of text in a diagram belong together, i.e., which form a single textblock. This can be highly complex, since it may be hard to discern which words actually belong together, for instance because they are apart from each other in the drawing (see e.g., issue l2) or even separated by (parts of) model shapes. The exception to this are activity labels, where we know that the label consists of the words located within the activity bounding box. Therefore, activity labels do not need to be detected through dedicated textblocks.

Textblock handwriting recognition aims to recognize the exact text that is contained within a textblock, i.e., to interpret the handwritten text. While handwriting recognition (HWR) methods for handwritten documents have been developed for decades, HWR for hand-drawn diagrams is largely unexplored and much more challenging [41], as textblocks can be rotated (see issue l1), in front of complex backgrounds, and with overlapping model element strokes.

Finally, *textblock relation detection* is concerned with finding the shape or edge that the textblock labels. This is challenging since textblocks may be in close proximity to multiple shape or edge candidates, as, e.g., in Fig. 4, making it hard to recognize the appropriate relation between textblock and model element.

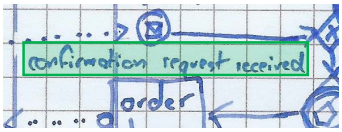


Fig. 4. Label candidates

General challenges. The complexity of detecting shapes, edges, and labels due to the aforementioned issues is amplified by various general challenges related to sketch recognition for hand-drawn models, such as:

- The paper on which a hand-drawn model was drawn may be lined, squared, or dotted. For instance, the graph paper in Fig. 1 adds numerous additional lines to the drawing, which may appear similar to lines used to denote model elements, such as resource pools (consider the thicker line denoted by issue g1).
- The image might contain additional contents that are visually similar to model elements. For example, Fig. 1 has punched holes (g2), which look similar to events, and visible model elements from the back side of the paper (g3).
- Drawing implements, such as pencils, may affect the clarity, consistency, and thickness of drawn lines, which can negatively affect the interpretability of a sketch. While Fig. 1 was drawn using a clear pen, a pencil would yield lines that look very similar to the lines that are already part of the paper (e.g., as indicated by g1).
- When hand-drawn models are captured using a camera,

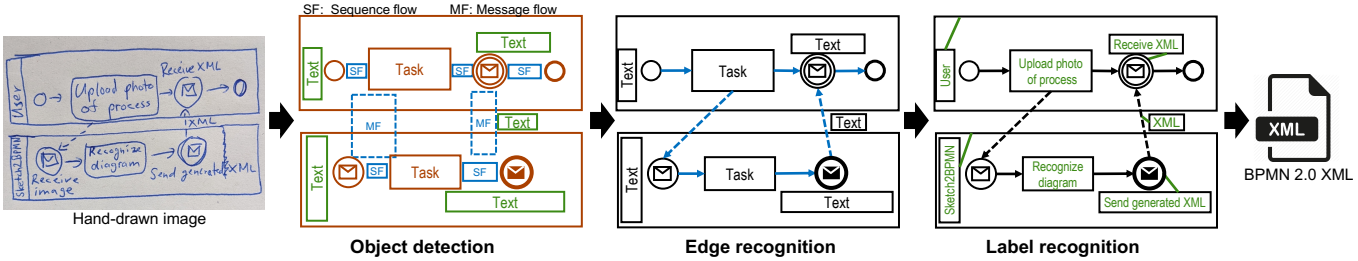


Fig. 5. Overview of our approach: Given an image, we first detect shapes, arrows, and textblocks as objects (object detection). Then, we identify the drawn arrow path and the shapes that each arrow connects (edge recognition). Next, we decode the textual content within each textblock and identify the shape or edge that each textblock labels (label recognition). Finally, we generate a BPMN 2.0 XML file.

rather than a dedicated scanner, additional quality issues may be introduced [40]. This includes images that are rotated or blurry, as well as those that include content beyond the paper or actually cut part of it off (e.g., the right-hand side of Fig. 1).

- Finally, it is important to recognize that establishing BPMN models is notoriously difficult [47], which means that it is not uncommon for modelers to make mistakes [48]. On the one hand, this can result in parts of a drawing being crossed out (issue g4), on the other hand, there is no guarantee that the final drawing is free of errors, which means that a recognition approach cannot depend on the syntactic correctness of the drawing.

In the next section, we propose our approach that aims to address the aforementioned challenges in order to accurately detect hand-drawn BPMN models.

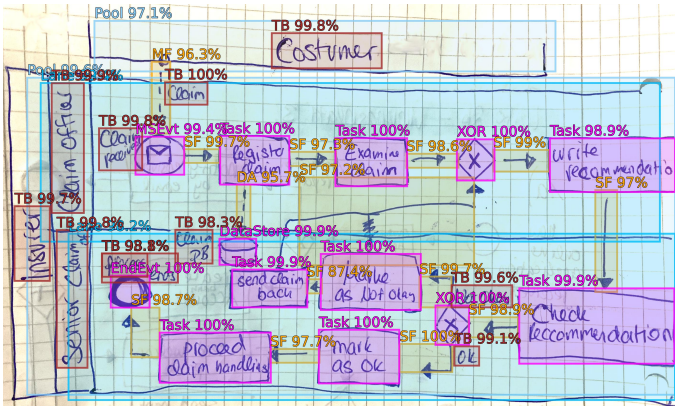


Fig. 6. *Object detection* output: the network has detected shape (blue), arrow (orange), and textblock (brown) objects.

4 The Sketch2Process Approach

This section introduces `Sketch2Process`, our approach for recognizing a hand-drawn BPMN model from an image. As visualized in Fig. 5, `Sketch2Process` consists of three main steps: object detection, edge recognition, and label recognition. The *object detection* step aims to detect all objects (shapes, arrows, and textblocks) that are part of the input image, characterizing each object as a bounding box and a predicted object class. While this step completes the recognition of shapes in the drawing, detected arrow objects are further processed in the *edge recognition* step. In this step, we identify the source and target shapes that each arrow connects, along with the path that the drawn arrow follows. Afterwards, the *label recognition* step decodes the textual content of each textblock and assigns it as a label to a corresponding shape or edge. Finally, `Sketch2Process` generates and returns a BPMN 2.0 XML file that captures the detected BPMN model. In the following, we provide details on the individual approach steps (Sections 4.1 to 4.3) and the subsequent output generation (Section 4.4). Finally, given that we employ three neural networks throughout our approach, we show how these networks are connected and jointly trained in Section 4.5.

4.1 Object Detection

In this first step, our approach aims to detect all objects (shapes, arrows, and textblocks) contained in a provided image. The input to this step is the entire input image, which is resized so that the longer side is equal to 1,333 pixels, i.e., the standard format required by the Faster R-CNN network that we use here. The output of this step are the detected objects, which are captured as sets of shapes S, arrow objects A, and textblocks T. Each of these objects is represented as a tuple (b, c, s), with b as its bounding box, i.e., a rectangle encompassing the predicted area of the drawn object, c as the object's predicted class, and s as the classification score of this prediction. Fig. 6 shows the outcome of this step.

To operationalize the object detection step, we use the *Faster R-CNN* [42] network and adapt its training configuration to the details of our use case, as detailed below.

Faster R-CNN. Faster R-CNN is a popular neural network approach to detect objects in an image in the form of the previously defined (b, c, s) tuples. Faster R-CNN owes its popularity not only to its high level of accuracy, but also to its extensibility. It has been extended to address various computer vision tasks beyond object detection, such as human keypoint detection [49] and visual relationship detection [50]. Inspired by these extensions, we also use Faster R-CNN as our object detection network, and extend it with edge and label recognition network components.

Faster R-CNN operates on RGB images, representing them as three-dimensional arrays, of which the size of the first two dimensions corresponds to the image width and height (in pixels). The third dimension consists of three channels, where each channel represents the color intensity of a pixel for the three primary colors: red, green, and blue. Faster R-CNN detects objects using the *image features* $feat$, a

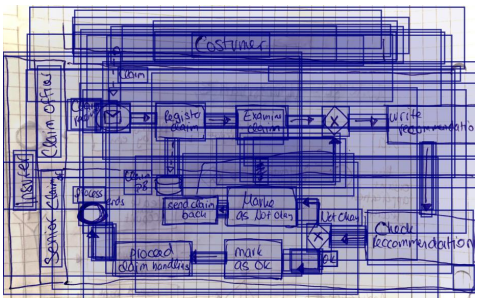


Fig. 7. *Object proposals*: Top 100 out of 984 proposals ranked by their objectness score.

learned three-dimensional representation of the image that is generated by its backbone network. The image features have a lower width and height than the original image, but a much higher number of channels. Rather than just capturing the intensity of primary colors, the channels of the image features correspond to various patterns that are learned by the network to detect and distinguish different object classes. For example, a channel can correspond to a specific stroke pattern, such as an arrowhead or a diagonal line segment, indicating for each channel pixel if this pattern can be found in the corresponding image region.

Given the image features $feat$, Faster R-CNN detects objects with a two-stage approach. The first stage generates a large set of (class-agnostic) *object proposals*, where each proposal is defined by a bounding box and a so-called objectness score.

Fig. 7 illustrates some of the predicted proposals for our running example. In the second stage of Faster R-CNN, a *box-head network* classifies each proposal and predicts a refined bounding box location. For this classification, the second stage predicts a score distribution over a set of predefined (foreground) object classes and a negative background class, with the sum of all scores for a given proposal equal to 1. Given a proposal's bounding box, the box-head network uses the *RoIAlign* [49] mechanism to cut out the part of the image features that corresponds to the proposal's image region, and then normalizes its size. The network uses these (smaller) image features to predict a refined bounding box b and the object class scores.

Detected objects. Finally, we turn a proposal into an object (b, c, s) by setting c as the most likely object class and s as the corresponding score. During inference, Faster R-CNN returns a set of objects, which we then divide into the aforementioned (disjoint) sets of shapes S, arrow objects A, and textblocks T, based on their predicted object classes. Note that with respect to shapes, set S only includes those objects with a classification score s equal to or above a score threshold for shapes, τ_s (we use 0.5 as a default and test further values in our evaluation experiments).

As can be seen in the output of Fig. 6, the objection detector aims to detect bounding boxes that as closely encompass shapes and textblocks as possible. For example, the bounding boxes of the *Check Recommendation* activity and *Costumer* [sic] textblock leave little space between the box and the actually drawn element.

By contrast, for arrow objects in A, the object detector is trained to establish bounding boxes so that the box connects the actual source and target shapes of the arrow, rather than just encompassing the arrow's drawn path. We describe the ground truth used for this purpose in Section 4.5. As, e.g., seen for the sequence flow arrows surrounding the XOR choices in Fig. 6, this can result in a bounding box that is considerably larger than the drawn arrow. By detecting arrow objects in this manner, it is considerably easier to turn them into edges that connect shapes, as done next.

4.2 Edge Recognition

In this step, our approach aims to recognize the BPMN edges indicated by the drawn arrows detected in the previous step. The input for this step consists of the detected shapes S and arrow objects A, and the image features $feat$, capturing the representation learned by Faster R-CNN. For each arrow object $a \in A$, our approach strives to recognize the two shapes that the edge connects and the edge's drawn arrow path. This is captured in the form of a tuple $e = (a, src, tgt, s, K)$, where $src, tgt \in S$ are the source and target shapes that the arrow connects, s its score, and K the drawn path of the arrow, represented as a sequence of keypoints. The output of this step, then, is a set of recognized edges E.

As visualized in Fig. 8, we decompose edge recognition into three stages. First, *edge candidate generation* produces a set of edge candidates for every detected arrow $a \in A$. Each edge candidate $src \xrightarrow{a} tgt$ relates an arrow a to a pair of possible source and target shapes in its proximity. Second, we use a trained *edge relation network* to process each edge candidate $src \xrightarrow{a} tgt$ and predict the likelihood s that arrow a indeed connects src and tgt, as well as the arrow path K that was drawn to connect src and tgt. Lastly, given the set of edge candidates detected for all arrows E_C, the *edge inference* procedure determines the final set of edges $E \subseteq E_C$. This procedure involves finding the most likely edge candidate for each arrow, and eliminating duplicate edges. In the following we introduce each stage in detail.

Edge candidate generation. Given an arrow $a \in A$, we first identify all potential pairs of source and target shapes that the arrow might connect. To that end, we take all shapes that are considered to be in proximity to the arrow's bounding box. Although the object detector already aims to connect an arrow's bounding box to its source and target shapes (see Section 4.1), we still need to account for minor prediction inaccuracies that could result in an arrow's bounding box not being fully connected to its corresponding shapes.

Therefore, we determine if an arrow a and a shape s are in proximity to each other by expanding the arrow's bounding box $a.b$ in a manner relative to its width and height. Specifically, we pad each side of $a.b$ by an arrow padding percentage pad_a of the box's weight and height, respectively.[6] We refer to this extended arrow bounding box as the (context-enriched) *arrow region*, $a.r$. Fig. 9 shows an example, with the arrow's bounding box in green and the arrow region in red. Based on the experiments reported on in Section 6.2, we found that a padding of $pad_a = 10\%$ is best suited for this purpose.

The set of candidate shapes for an arrow a then includes all shapes that overlap with the arrow region, i.e., it contains

6. To account for overly small or large arrows, we enforce a minimum padding value of 15px and a maximum of 100px.

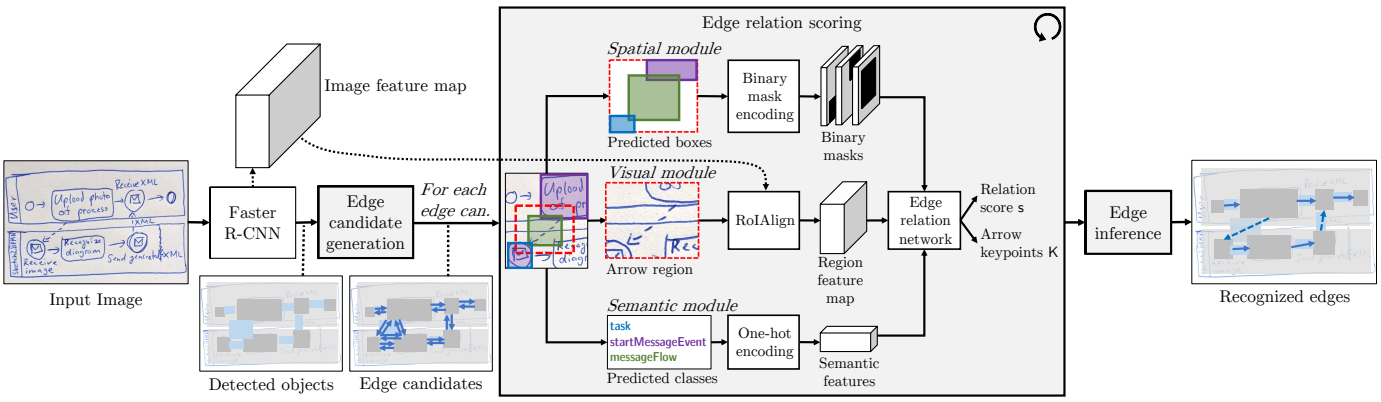


Fig. 8. *Edge recognition*: Given the objects detected by Faster R-CNN, *edge candidate generation* produces a set of edge candidates for every arrow. For each edge candidate $src \xrightarrow{a} tgt$, the *edge relation scoring* procedure predicts the relation score and arrow keypoints from the features extracted by three modules. Finally, the *edge inference* procedure identifies the most likely edge candidate for each arrow and eliminates duplicate edges.

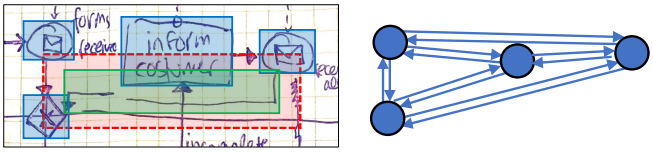


Fig. 9. *Edge candidate generation*: Given an arrow (green), each shape that intersects with the arrow region (red) is considered a candidate. After, we create $O(n^2)$ edge candidates (blue arrows) for the arrow and its n candidate shapes.

any shape $s \in S$ of which its bounding box $s.b$ intersects with the arrow's region $a.r$. For instance, in Fig. 9, the arrow region overlaps with the bounding boxes of all shapes, thus resulting in four shape candidates for the arrow. There are two exceptions to this procedure, related to the specifics of collaboration shapes in BPMN. First, we only consider pool shapes as candidates for arrows classified as message flow, as the other edge types do not connect to pools in BPMN. Second, we do not consider lane shapes, given that no edge type connects to these shapes at all. Removing invalid collaboration shape candidates greatly reduces the total number of candidates to evaluate, as, in the example of sequence flows, we would otherwise create a candidate shape for both the lane and pool that the arrow belongs to.

Given n candidate shapes, we create $n * (n-1)$ directed edge candidates of the form $src \xrightarrow{a} tgt$, with $src \neq tgt$, i.e., we consider all pair-wise combinations as potential source and target shapes for arrow a, as illustrated in Fig. 9. We intentionally do not apply any heuristics to further prune the set of shape pair candidates. For example, in Fig. 9, it seems unlikely that the predicted arrow bounding box connects the two leftmost shapes. However, in some cases arrows are drawn in such a way that they connect two shapes with a large detour. Instead of applying heuristics to detect such scenarios, we, therefore, opt for a learning-based approach that can exploit these spatial bounding box correlations to decide if an arrow connects a shape pair, as detailed in the second stage below.

While the number of edge candidates is quadratic with respect to the number of shape candidates, this is generally not problematic in practice. In particular, we observe that the majority of arrows (e.g., 71% in the training split of `hdBPMN`) have two candidate shapes, which results in only two edge candidates, one per direction. This is, e.g., seen for the two short sequence flow arrows in Fig. 9.

Edge relation scoring. In the second stage, we use an edge relation network to predict the likelihood that an edge candidate $src \xrightarrow{a} tgt$ indeed connects shape src to tgt, and to predict the drawn path of the arrow (captured as a sequence of keypoints K). As shown in Fig. 8, the network is provided an edge candidate as input, comprising the bounding boxes and class predictions of the three objects, and the arrow region $a.r$. As depicted, the edge relation scoring procedure uses three modules to analyze different kinds of features. This modular approach is inspired by existing works [51], [52], where relationships between objects in images are also detected using spatial, semantic, and a visual modules. In the following, we describe the three modules in detail, and explain how the edge relation network predicts the edge relation score and keypoints given the encoded features.

Spatial module. The spatial module encodes spatial features for each edge candidate, i.e., the (relative) locations of the bounding boxes of the arrow and the two associated shapes. For each predicted box of the three objects, the spatial module generates a 28×28 binary mask that indicates the location of the box within the arrow region $a.r$, as illustrated in Fig. 8. Each binary mask is initialized with zeros and then filled with ones for each bounding box pixel that is located within $a.r$. Specifically, we first compute the intersection bounding box of each shape and $a.r$. Next, we normalize the intersection boxes by dividing their x and y coordinates by the width and height, respectively, of $a.r$. Then, we multiply each coordinate by 28 and obtain boxes in the range $[0, 28]$, which we finally convert into binary masks.

The binary masks instruct the network which task it is supposed to solve, i.e., they indicate the source and target shape between which the network should try to recognize an edge. In addition, the spatial features can be used by the network to assess the likelihood that arrow a connects the source and target shapes, src and tgt. For instance, spatial features capture that corresponding source and target shapes are typically located on opposite sides of an arrow's bounding box, as seen for the example in Fig. 8. However, to

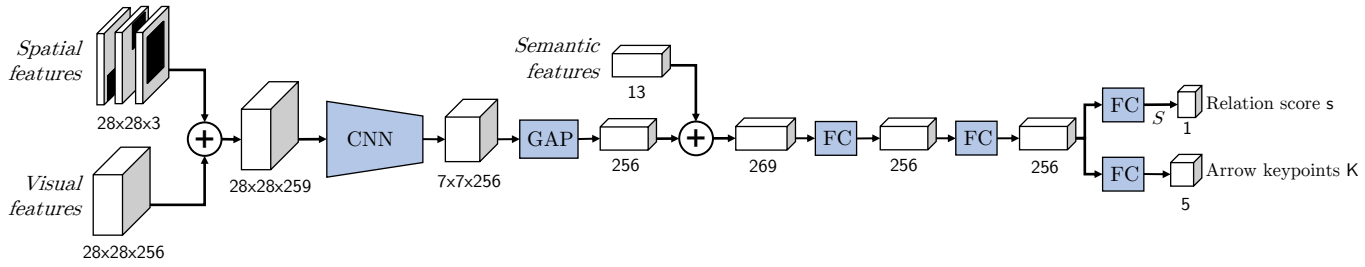


Fig. 10. *Edge relation network*: The concatenated visual and spatial features are processed by a convolutional neural network (CNN). Next, the feature vector obtained from the global average pooling (GAP) layer is fused with the semantic features. After two fully-connected layers (FCs), a network branch with a sigmoid function (*S*) predicts the edge relation score s, and a second branch predicts the arrow keypoints K.

actually recognize an edge, the network also needs the visual features of the arrow region. These features are generated by the visual module, as we describe next.

Visual module. The visual module generates a learned 28×28×256 visual feature representation of the arrow region $a.r$. These visual features can be used to assess the likelihood that an arrow connects to particular shapes, e.g., by considering the proximity and direction of a drawn arrow, whereas they are also used to identify the keypoints in the drawing, i.e., an arrow's start, end, and notable points in between. As discussed in Section 4.1, Faster R-CNN uses the *RoIAlign* mechanism to cut out the part of the image features that corresponds to the bounding box of an object proposal. Fig. 8 shows that the visual module uses the same mechanism to extract the features for $a.r$.

As discussed in the previous section, the network needs both spatial and visual features to solve the edge recognition task. Given just the visual features, the network lacks information about the shape pair it should evaluate. With both features, the network can learn to figure out if there is an arrow whose tail is in proximity to the source shape, whose head is in proximity to the target shape, and whose bounding box equals the provided arrow bounding box. Both the spatial and visual module rely on the bounding boxes of the edge candidate objects, but do not take into account the predicted classes. The next section therefore presents the semantic module, which leverages the predicted classes.

Semantic module. The semantic module provides the network with an encoded representation of the predicted classes of an edge. The network can use these features to learn class-specific modeling rules and conventions, e.g., that control-flow edges connect activities, events, and gateways, whereas data associations involve at least one data element. Given the large number of shape classes, many combinations of source and target shape classes have only a few training samples, and we observed that this leads to overfitting during training. Therefore, instead of directly using the predicted shape classes, we map the 21 shape classes into five more general shape groups: *activity*, *event*, *gateway*, *collaboration*, and *data elements*. From a process modeling perspective this is reasonable, as the majority of modeling rules are applied on shape group level. For example, the rule that a data association always connects a data store or data object can be simplified to a rule on data element level.

The predicted arrow class and the mapped shape groups are converted into vectors using one-hot encodings, which results in a vector of size 3 for the arrow, and two vectors of size 5 for the shapes. Finally, the three vectors are concatenated into a semantic feature vector of size 13.

Edge relation network. Given the input stemming from the three aforementioned modules, we use a neural network to predict the edge relation score and arrow path, as illustrated in Fig. 10. The architecture of our network largely follows our earlier work [16], which is in turn based on a network designed to predict relationships between form elements in a small document image dataset [52]. We first combine the visual and spatial features by concatenating the three binary masks as additional channels to the visual features, and obtain a 28×28×259 feature representation. The combined visual and spatial features are processed by a convolutional neural network (CNN). We use the same CNN architecture as in our earlier work [16], which consists of six depth-wise separable convolutional layers [53]. The spatial resolution of the features is downsampled twice by a factor of two, using strided convolutions in the third and last depth-wise convolution. This results in features of size 7×7×256. Next, we convert the three-dimensional features to a vector using global average pooling (GAP), which computes the mean over each of the 256 channels. We then concatenate the semantic vector, leading to an intermediate vector of size 269. We subsequently integrate these semantic features by applying two fully-connected layers (FCs), resulting in a final feature vector of size 256.

Given the final feature vector, the network predicts two outputs. First, a binary classification layer with a sigmoid function predicts the edge relation score s. A second linear layer predicts the arrow path as a sequence of arrow keypoints K, where each $(x, y) \in K$ represents a point within the image. Following our earlier work [16], we choose $|K| = 5$, and encode the arrow keypoint coordinates relative to the arrow region. The first and last keypoints in K indicate the arrow tail and head positions, and the three intermediate points capture the drawn path. The number of intermediate keypoints required to describe an arrow vary, e.g., an elbow arrow can be described with just one intermediate keypoint. Therefore, we remove superfluous intermediate keypoints in the last step of our approach, as we describe in Section 4.4.

Given the edge relation score s and the arrow keypoints K, we obtain a tuple $e = (a, src, tgt, l, K) \in E_C$ for each edge candidate. The next section details how we use the set of edge tuples E_C to determine the final set of edges.

Edge inference. The edge inference procedure determines

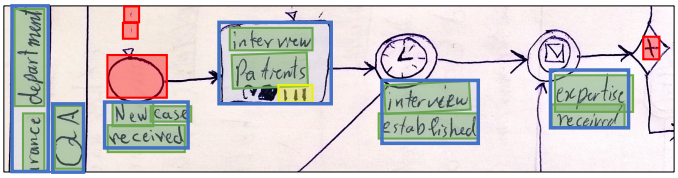


Fig. 11. *Textblock handwriting recognition*: For each textblock (blue), the label is obtained by detecting the reading order of its contained words (green), except for notation words (yellow).

the final set of edges $E \subset E_C$. For each arrow a, we choose the edge candidate e with the highest edge score $e.s$. In other words, we reduce the number of edge candidates to the number of arrows by only keeping the most likely edge per arrow. Next, we aggregate the edge score $e.s$ with the object detector score $e.a.s$ by taking the minimum of both, i.e., we set $e.s = \min(e.s, e.a.s)$.

Next, we remove all edges whose aggregated score is lower than a threshold τ_e (we use 0.5 as a default and test further values in our evaluation experiments), which results in $E'_C \subset E_C$. Last, we identify edges with the same connection, i.e., edges $e_1, e_2 \in E'_C$ with $e_1.src = e_2.src$ and $e_1.tgt = e_2.tgt$. We resolve these duplicates by only keeping the edge with the higher aggregated score $e.s$. As a result, we obtain the final set of edges E.

4.3 Label Recognition

In this step, our approach aims to recognize the BPMN labels indicated by the previously detected textblocks T (Section 4.1). The input for this step consists of the sets of shapes S, edges E, and textblocks T, as well as the image features $feat$. The output of this step is a set of labels L, where each label $l = (t, tgt, s, txt) \in L$ relates a textblock t, to a target shape or edge $tgt \in S \cup E$. Further, s represents the consolidated label score, and txt the textual content.

To achieve this, we decompose label recognition into two main parts. First, *textblock handwriting recognition* decodes the textual content within each textblock. Second, *textblock relation detection* strives to identify the shape or edge that each textblock labels, and, to eliminate duplicate textblocks that both relate to the same shape or edge. Besides the textblocks in T, our approach also decodes the textual content within the activities that are part of S. Therefore, we create a pseudo textblock for each activity. The pseudo textblock receives the same bounding box as its associated activity shape, but does not participate in textblock relation detection since its target shape is already known. We refer to the set of regular and pseudo textblocks as T'.

Below, we introduce the two parts to obtain L in detail.

Textblock handwriting recognition. Given a textblock $t \in T'$, the textblock handwriting recognition procedure tries to decode the textual content txt within the textblock. We decompose textblock handwriting recognition into two stages. First, in *image word recognition*, we use an off-the-shelf OCR service to recognize all words in an image. Second, the *textual content decoding* procedure identifies the words that belong to each textblock, and combines the words into a word sequence that represents the textual content of the textblock.

Image word recognition. In this stage, we try to identify all handwritten word objects $w = (b, d, txt) \in W$ within a given image. Each word w is defined by a bounding box b, the degree of its rotation angle d, and the word's textual content txt. To accomplish this, we leverage an off-the-shelf OCR service that supports handwritten text. Given a raw image as input, the OCR service returns a set of text lines, where each line consists of a sequence of words. As words can be arbitrarily rotated, the word bounding boxes also indicate the angle d. Since OCR services are commonly optimized towards handwritten documents, we observe that the returned text lines often combine lines of multiple textblocks, e.g., when two textblocks are next to each other. In Fig. 11, the service might recognize the text line "interview expertise", even though both words belong to different textblocks. We therefore discard the text line information and only keep the returned words W.

Next, we describe our approach for identifying the word sequences that represent the textual content of each textblock.

Textual content decoding. Given the textblocks O'_T and words W, we propose a procedure that decodes the textual content of each textblock. To this end, we first identify the words $W^t \subset W$ that belong to each textblock t, and, then, combine the words W^t to obtain the textual content txt.

To identify the words that belong to a textblock, we first compute the intersection over area (IoA) between all words and textblocks in the image. The IoA quantifies to what fraction of a word w is contained in a textblock t, and is defined as the overlap area of $w.b$ and $t.b$ divided by the area of $w.b$. We match each word to the textblock with the highest IoA, while only keeping words whose textblock IoA exceeds 50%. We use this threshold to account for minor word and textblock localization errors. In Fig. 11, the green boxes illustrate the words that have been assigned to their enclosing textblock. Further, the red boxes show that the OCR service returns many (false positive) words that actually are part of drawn shapes or edges. Among others, we frequently observe recognized "X" characters for exclusive gateway symbols, or multiple returned "-" or "I" characters for horizontal and vertical arrows. These false positives are discarded by only keeping words within textblocks. However, we also observe false positive words located within textblocks that often correspond to notational elements of BPMN, as illustrated by the yellow box in Fig. 11. We try to eliminate these with a filter list of common notation words, which includes recognized words for parallel task (e.g., "III", "111") and collapsed sub-process markers (e.g., "+"). Applying both the matching and filtering procedure yields the final set of words W^t for each textblock t.

In the second stage, a reading order detection algorithm decodes the textual content txt of each textblock t. Given the words W^t, the algorithm uses the bounding box and rotation angle d of each word to identify which words form a line, and in which order the lines should be read. The algorithm details can be found in the supplementary material.

Textblock relation detection. Similar to edge recognition (Section 4.2), we formulate textblock relation detection as a relationship detection problem and decompose it into three stages. First, *relation candidate generation* produces a set of shape and edge candidates for every textblock $t \in T$. Second,

relation scoring predicts the score for each candidate pair, where a candidate pair consists of a textblock and a related shape or edge. Last, the *relation inference* procedure tries to find the most likely shape or edge for each textblock, and to eliminate duplicate labels, i.e., multiple textblocks that have been related to the same shape or edge.

Relation candidate generation. Given a textblock t, we first identify all shapes and edges that the textblock might label. In edge candidate generation (cf. Section 4.2), the set of candidates includes all shapes that overlap with the arrow region. Here, the arrow region corresponds to the arrow bounding box extended proportionally to its width and height. We follow a similar procedure to generate textblock relation candidates, but also account for the following specifics of BPMN labels. First, we observe a large variance in textblock bounding box sizes, since the textual content can range from single digit to multi-line text phrases. Therefore, we do not pad each textblock relative to its dimensions. Instead, we compute the median size of all textblocks in the image, which we refer to as M. Here, the size of a textblock is defined as the mean of its width and height. To obtain the textblock region, we then pad the side of each textblock by a factor $pad_t * M$. Based on our experimental results described in Section 6.2, we set $pad_t = 1$.

In order to reduce the number of false positive candidates to evaluate, we also leverage two patterns regarding the positions of textblocks relative to the shape or edge that they label. First, we observe that edge bounding boxes often do not closely capture the drawn path of the arrow, especially for diagonal or elbow arrows. For example, even though the drawn path of the sequence flow in the center of Fig. 1 is far away from the "Not Okay" textblock, their two bounding box are in proximity. Therefore, we only consider an edge as relation target if its drawn path, identified by its keypoints K, intersects the textblock region. Second, we observe that pool and lane labels are commonly located near the boundary of the detected shape, as illustrated in Fig. 6. Therefore, we only consider a pool or lane as relation target if its border intersects the (extended) textblock region. This way, we avoid creating a candidate for every textblock that actually labels a shape or edge within the pool.

Fig. 12 shows an exemplary textblock in an image, along with the related candidate shapes (blue) and edges (orange) that intersect with the textblock region (red). As indicated, we do not consider activity shapes as relation targets, since we know the relation targets of the (pseudo) textblocks that we created for each activity. Further, we exclude data association arrows, as not a single such arrow is labeled in the `hdBPMN` dataset. In the next step, we use a network to evaluate each textblock relation candidate (t, tgt).

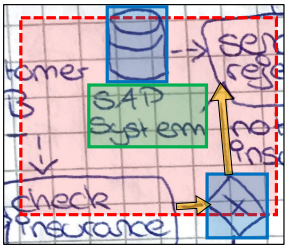


Fig. 12. Relation candidate generation

Relation scoring. In this second stage, we use a textblock relation network to determine the score of a relation candidate (t, tgt), i.e., the likelihood that textblock t labels a shape or edge, tgt. As in edge recognition, the network consists of three modules, and outputs a relation score. The main difference is that the textblock relation network operates on object pairs instead of triplets. As a consequence, the spatial module generates two instead of three binary masks. Further, since the source object of the textblock relation is always a textblock, the semantic module only encodes the predicted class of one object, namely the target shape or edge. Finally, the visual module extracts the features from the textblock region using RoIAlign, and thus works in the same way as with edge recognition. The network details can be found in the supplementary material.

Relation inference. The relation inference procedure determines the final set of textblock relations, and as part of this, also eliminates textblocks that have been identified as duplicates. This procedure is, again, performed in a similar manner as its corresponding part in the edge recognition step. For each textblock t, we choose the relation candidate (t, tgt) with the highest relation score s. We then aggregate the relation and the object detector score by taking their minimum, and remove all textblocks whose aggregated score is lower than a threshold τ_l (we use 0.5 as a default and test further values in our evaluation experiments). Last, we identify textblock duplicates, i.e., multiple textblocks that have been matched to the same shape or edge, and resolve these cases by only keeping the textblock with the highest score.

4.4 Approach Output

The last step in our approach takes the final shapes S, edges E and labels L to create a process model in the BPMN 2.0 XML format. The XML format consists of two main schemata: the actual process model and the BPMN DI schema, which defines the shape and label bounding boxes and the waypoints of edges. For each predicted shape, edge, and label, we create a respective element in the XML file. When creating a BPMN DI edge element for each $e \in E$, we follow the typical convention and define the first and last waypoint as the points that intersect with the edge's source ($e.src$) and target ($e.tgt$) shapes, respectively. To that end, we shift the first and last predicted keypoint of $e.K$ to the nearest point on the bounding box boundary of the connecting shapes, except for gateways, where we connect the keypoint to the closest of the four diamond corner points. For the intermediate keypoints, we remove superfluous points with the Douglas-Peucker line-simplification algorithm [54]. Concretely, we remove every point where the distance between the original and induced simplified path is less than 5% of the total arrow length, resulting in a smoother representation as the final output of `Sketch2Process`.

4.5 Training

To operationalize `Sketch2Process`, we need to train and validate all neural network components on a dataset with an accompanying ground truth, split into a training and validation set. In this section, we describe the format of this ground truth, the overall training procedure, and details on our employed image augmentation pipeline.

Ground truth. Given an image containing a hand-drawn BPMN model, our approach uses a ground-truth annotation

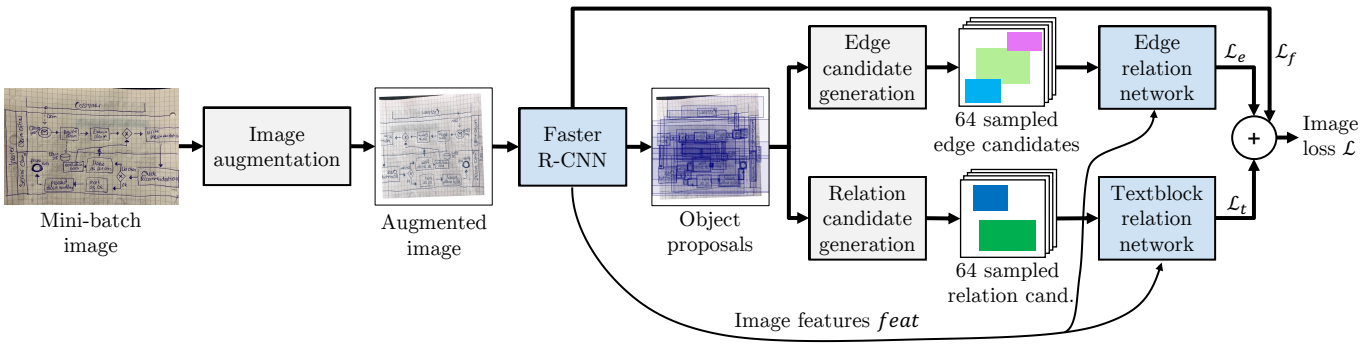


Fig. 13. *Training overview*: Given the augmented image, Faster R-CNN produces a large set of object proposals. Edge candidate generation identifies all proposals that sufficiently overlap with a ground-truth shape or edge to produce the set of edge candidates. It then samples 64 edge candidates, for which the edge loss $\mathcal{L}_e$ is computed by comparing the edge relation network outputs to their ground-truth counterparts. The textblock relation loss $\mathcal{L}_t$ is computed on 64 relation candidates using a similar procedure. Finally, the image loss $\mathcal{L}$ combines the losses of all networks.

that captures the shapes, edges, and labels contained in it. The formats for shapes and labels are straightforward and follow directly from the annotation procedure. Specifically, each *ground-truth shape* is a tuple (c, b), with c as the shape's class and b its bounding box, whereas each *ground-truth label* is a tuple (txt, tgt, b), with txt as the label's text, tgt its target shape or edge, and b its bounding box.

The ground truth of edges is slightly more intricate. Particularly, each *ground-truth edge* is a tuple (src, tgt, K, b), where src and tgt are the source and target shapes that the edge connects, K a sequence of keypoints (x and y coordinates, corresponding to BPMN waypoints) that are used to capture the edge's drawn path, and b the edge's bounding box. The first and last keypoints of K, respectively, correspond to the tail and head of the edge, whereas additional keypoints may be used to indicate an edge's bending points. Here, it is important to note that the tail and head keypoints should correspond to the points where the edge intersects or is supposed to intersect with its source and target shapes, rather than to the actually drawn start and endpoints of the edge. In this manner, we account for incompletely drawn edges that do not connect with their corresponding shapes (see issue e2 in Fig. 1). Then, the object detector trained on these annotations will strive to predict where an arrow bounding box *should* have ended (or started) if the edge had been drawn properly, rather than predicting the point where the edge ends (or starts) in the drawing (see also Section 4.1). Finally, an edge's bounding box b is automatically computed as the smallest box that contains all keypoints in K.

Training procedure. In the context of deep learning, which encompasses the neural networks we employ, the goal of model training is to find network parameters (i.e., weights) that maximize the performance on the validation set. We follow the de facto standard in deep learning and train the networks with stochastic gradient descent (SGD) [55]. SGD is an iterative method, where, at each step, k examples (images in our case) are randomly sampled from the training set. Here, k is referred to as the batch size, and the set of sampled examples is called a mini-batch. During each SGD iteration, a *loss function* quantifies the difference between the ground truth and the predicted outputs of the neural networks. We apply the loss function per image and obtain a mini-batch loss by averaging over the k image losses. Although we defer the technical details to the supplementary material, the intuition of how we compute an image loss is as follows.

As depicted in Fig. 13, we first apply an *image augmentation* pipeline (described below) and obtain a randomly augmented version of the input image. As mentioned in Section 4.1, Faster R-CNN uses the (augmented) image to produce the image features $feat$. Given these features, it generates a set of object proposals in the first stage. In the second stage, a box-head network uses the proposals and image features to predict the detected objects. Following the standard training procedure of this object detector [42], we compute the aggregated Faster R-CNN loss $\mathcal{L}_f$ by comparing both the proposals and the detected objects against the ground-truth objects. Concretely, this means that Faster R-CNN uses a ground truth consisting of all shapes, as well the bounding boxes and classes of edges and labels (representing arrow and textblock objects, as described in Section 4.1).

As Fig. 13 illustrates, we use both the object proposals and the image features $feat$ to train our edge and textblock relation networks, which follows related work that extends Faster R-CNN [49]. To this end, we randomly sample 64 edge candidates (for edge recognition) and 64 relation candidates (for label recognition) from the object proposals. We then compute an edge loss $\mathcal{L}_e$, which consists of a relation loss and a keypoint loss. For the *relation loss*, we use the ground-truth information on both shapes and edges to assess if an edge candidate is considered true positive or not, which results in a binary ground-truth relation score. We then compare the predicted and ground-truth relation score using binary cross entropy. For the *keypoint loss*, we compute the mean squared error between the predicted and ground-truth keypoints K of an edge (though the latter are resampled in an equidistant manner). The label relation loss $\mathcal{L}_l$ is calculated in the same manner as for edge relations, except that we are using the entire ground truth, as textblocks can be used to label both shapes and edges. Finally, we compute the image loss, $\mathcal{L} = \mathcal{L}_f + \mathcal{L}_e + \mathcal{L}_l$, as a weighted combination of the losses of the three networks.

Image augmentation pipeline. During training, we have to account for the particularities of the `hdBPMN` dataset in terms of its size (hundreds rather than many thousands of images) and diversity with respect to the means used to create and

digitize the hand-drawn models, such as the type of paper and drawing implement (see also Section 5.3).

Therefore, given a mini-batch image, we apply a randomly selected set of augmentation methods that are specifically suited to the data at hand. A code listing capturing the exact pipeline we use for this, based on the Albumentations library [56], is provided in the supplementary material, whereas we here focus on a description of the augmentation methods and their relevance.

In particular, we employ augmentation methods that randomly rotate, flip, and scale training images, as well as add Gaussian noise. These methods have been shown to work well for smaller datasets used in flowchart recognition [22]. On top of those, we also employ methods that are particularly aimed at the camera-based images in our dataset, which we introduced in a previous work [23]. This involves altering varying properties to reflect different kinds of camera-based images, for which we randomly change the brightness and contrast of the image, and shift the hue, saturation, and value color scale.

Finally, to improve generalization, we further introduce a new crop augmentation in this work, which cuts out a random part of the diagram and then enlarges the crop to the scale of the original diagram. Concretely, we crop an image patch whose scale compared to the original image is between 0.2 and 1.0, and whose ratio is between 0.5 and 1.5. Then, we keep all objects from the ground truth where at least 70% of the bounding box is located within the cropped image. To avoid having empty (cropped) images, we reapply the entire augmentation pipeline until the augmented image contains at least 3 objects. Overall, we observe that random cropping improves our recognition results, as during training the object detector sees variations of the same image that have been (1) stretched in horizontal or vertical direction and (2) where only a subset of the diagram symbols are visible.

Fig. 14 shows some augmented images obtained by applying our proposed pipeline on the running example.

Fig. 14. *Image Augmentation Example*: Our augmentation pipeline produces different variations of the same input image

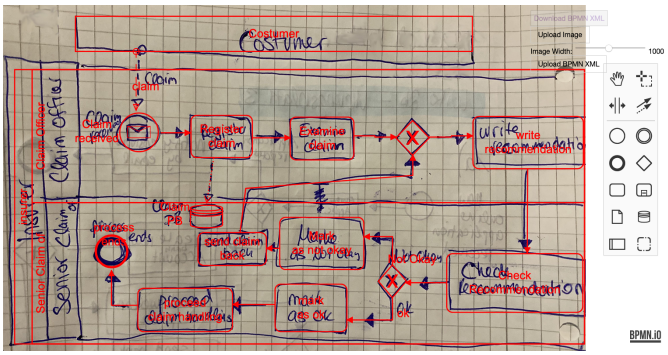


Fig. 15. Example of an annotated sketch in the *BPMN image annotator*. Shapes are sized and positioned to match their hand-drawn counterparts, while edges are modeled using waypoints to resemble the drawn arrows.

5 The hdBPMN Data Set

This section discusses the collection, annotation, characteristics, and splits of `hdBPMN`, the dataset we established for our work. The images and BPMN annotations are publicly available at: https://github.com/dwslab/hdBPMN.

5.1 Collection Procedure

We collected 704 images of hand-drawn BPMN models from 107 participants, all students at the University of Mannheim. Each image corresponds to a solution that was submitted by a student for a graded assignment in an exercise sheet or exam. The obtained models stem from 11 modeling tasks, 10 of which involved the establishment of a BPMN model on the basis of a textual process description, while the other involved the conversion of a Petri net into a BPMN model. Students were asked to draw their models on paper and embed a scan or photo of their drawing in a PDF, with the only constraint that the models should be readable.[7]

Note that, aside from splitting pages that covered multiple modeling tasks, we deliberately did not crop images, resulting in the occasional inclusion of background objects. Finally, we used an image editor to conceal personal details (e.g., names and student IDs). The resulting images were assigned names that follow a `taskID_participantID` convention to recognize drawings by the same participant.

5.2 Annotation

To train and evaluate our BPMN recognition approach, we annotated the hand-drawn shapes and edges in each image using the format described in Section 4.5. To this end, we developed an image annotation tool based on the open-source `bpmn-js` BPMN viewer and editor,[8] which we have made publicly available.[9] Fig. 15 depicts the annotation tool in action. The `bpmn-js` editor prevents users from modeling edges that are syntactically incorrect, e.g., an *end event* with an outgoing *sequence flow*. Since we want to be able to annotate

7. The modeling tasks and instructions are available in our repository.

8. https://github.com/bpmn-io/bpmn-js
9. https://github.com/dwslab/bpmn-image-annotator

TABLE 2
BPMN elements in the 704 annotated images

Type	Group	Elements and their frequencies
Shape	Activity	task (4,094), subprocess (collapsed) (133), subprocess (expanded) (7), call activity (15)
	Event	start (424), intermediate throw (7), end (936), message start (508), message interm. catch (507), message interm. throw (291), message end (199), timer start (86), timer intermediate catch (289)
	Gateway	exclusive (1,347), parallel (661) inclusive (3), event-based (171)
	Resource	pool (1,103), lane (688)
	Data ele.	data object (887), data store (219)
Edge		sequence flow (9,893), message flow (1,822), data association (1,773), annotation assoc. (170)
Label		textblock (12,501), word (32,009), text anno. (176)

such incorrect edges in the images, if they are present, we disabled the `bpmn-js` correctness rules in our annotation tool. Upon completion, the annotation is exported as a *BPMN 2.0 XML* file, which links the shapes, edges, and labels of the BPMN model to their corresponding locations in the image.

5.3 Dataset Characteristics

The 704 annotated images contain more than 70,000 annotated elements. As shown in Table 2, the models in the dataset are highly expressive, spanning 25 types of BPMN elements, including 4 types of activity shapes, 9 types of events, 4 types of gateways, and 4 types of edges. Largely owing to the different modeling tasks from which they stem, the individual BPMN models differ in terms of their size, complexity, and expressiveness (i.e., number of types covered). The models resulting from the 11 different modeling tasks have up to 15 activities and 26 events, 0 to 10 gateways, 0 to 8 resources, and 0 to 15 data elements. Some tasks result in simpler models (e.g., task1: 11.0 shapes and 11.9 edges on average) and others in more complex ones (e.g., task3: 25.8 shapes and 27.3 edges). Note that the running example depicted in, e.g., Fig. 1, represents a task of intermediate complexity.

Aside from the complexity of a particular process, the recognition difficulty of an image is affected by various other aspects, mainly corresponding to the *general challenges* highlighted in Section 3, such as the use of different paper types (blank, lined, or squared) and drawing implements (pen versus pencil), image-capturing issues (such as background objects, cut-off parts, and blurriness), the inclusion of crossed-out parts, and the presence of modeling errors.

As a result, the 704 images in the publicly-available `hdBPMN` dataset thus depict BPMN models that span a broad range of BPMN elements and have a high degree of diversity. Fig. 16 visualizes this by showcasing some of the different manners in which various kinds of shapes were drawn.

5.4 Dataset Splits

Following related hand-drawn diagram datasets [14], [57], we split up the dataset into publicly available training, validation, and test parts. Each participant in the dataset contributed between one and nine diagrams. While the variability of factors such as writing style, writing medium and image capturing method is high between participants, there are substantial similarities between the diagrams of one participant. Therefore, we split the dataset by participants, such that the participants in the training, validation, and test set are disjoint. Specifically, we created a random 60%/20%/20% split over the participants, and assigned each diagram to the respective part. The resulting training/validation/test set contain 432/144/128 diagrams from 65/21/21 participants, respectively.

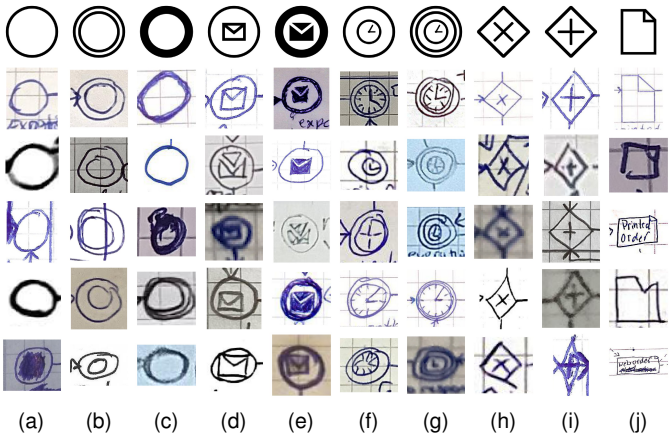


Fig. 16. Examples of hand-drawn *events* (start 16a, intermediate 16b, end 16c, message start 16d, message end 16e, timer start 16f, timer interm. 16g), *gateways* (exclusive 16h, parallel 16i), and *data objects* 16j.

6 Evaluation

To demonstrate the capability of our approach, we trained and optimized it using the training and validation set of `hdBPMN`, and conducted an evaluation using its test set. The evaluation results clearly demonstrate that our approach can reliably recognize hand-drawn BPMN models from images and, hence, remove undesirable friction in the modeling workflow.

6.1 Evaluation Setup

Below we elaborate on the details of our employed implementation and parameter settings, as well as the metrics and baselines used to evaluate our approach.

Implementation. Our neural network implementation is based on the Detectron2 [45] framework, which provides an extensible Faster R-CNN implementation based on PyTorch [58]. For OCR, we use the Microsoft Azure OCR service,[10] version v3.2. The service is asynchronous, i.e., we submit the image via a processing request, and then periodically check if the results are available. In order to optimize the overall runtime of our approach, we perform network inference while waiting for the OCR results.

Our code is available for research purposes upon request.[11] Readers are invited to try out their own sketches

10. https://docs.microsoft.com/en-us/azure/cognitive-services/computer-vision/overview-ocr

11. For proprietary reasons, requests for the source code of the implementation should be submitted to bernhard.schaefer@sap.com.

using our public demo:
http://sketch2bpmn.informatik.uni-mannheim.de/.

Parameter settings. `Sketch2Process` has several parameters, for which we analyze the effect of different choices. During inference, we use three score thresholds, one for shapes (τ_s), one for edges (τ_e), and one for labels (τ_l), to decide which elements to keep. For each score threshold, we explored choices in the range from 0.05 to 0.95. In addition, both our edge and label recognition components have a region size parameter, which we consider the most important parameter of the respective components, as they are used to identify elements in proximity during candidate generation, and to extract the visual features in the scoring procedures. For the arrow region size, we explored arrow padding percentages pad_a in the range from 0% to 50%, with a step size of 10%, and also evaluated 5% when we found that 10% performs best. For the textblock region size, we explored textblock padding factors pad_t in the range from 0.0 to 2.0, with a step size of 0.25.

Besides the mentioned parameters, Detectron2 has several training configurations, for which we apply the settings from previous works [16], [22]. Faster R-CNN can be equipped with different backbone networks. We use the ResNet-50-FPN [59] backbone throughout the experiments, which is relatively fast and provides a good speed-accuracy trade-off. For training, we largely follow the default Detectron2 configuration for training Faster R-CNN with a ResNet-50-FPN backbone. We use stochastic gradient descent with a batch size of 4, and a learning rate of 0.005. We train the model for 90k iterations, which means that the model sees 360k augmented images (90k batches of size 4). This takes about 32 hours on a Tesla V100 GPU with 16GB memory. During training, we multiply the learning rate after 50k and 70k iterations by a factor of 0.4. Here, we follow our previous work [16], where we found that the default factor of 0.1 decreases the learning rate too much. We initialize the model weights with models pretrained on the COCO dataset, which we obtain from the Detectron2 model zoo. Although the COCO dataset consists of images from a different domain than our target one, i.e., images of everyday scenes, we observed minor improvements with this transfer learning strategy in our previous works.

Finally, another important part of the training procedure is the image augmentation pipeline. To understand its impact on the overall results, we conducted an ablation study in which we compare a model trained using our full augmentation pipeline to two additional models: one with no augmentation at all, and one with just the default augmentations from the Detectron2 library (random resizing and horizontal flipping).

Metrics. To evaluate our approach, we compare a process model extracted by our approach to the one in the manually annotated image (see Section 5.2), i.e., the *ground truth*.

Object detection metrics. To quantify the *object detection* performance, we follow related work in diagram recognition [14], [38], [39]. A detected shape, edge, or textblock is considered a true positive if it has the correct class and its bounding box overlaps sufficiently with its ground-truth counterpart. Particularly, following other work [38], we consider this overlap sufficient if the bounding boxes have an overlap that exceeds an IoU threshold of 50%, which accounts for annotation inaccuracies in the ground-truth bounding boxes. To quantify object detection performance, we then use this notion of true positives to match the ground truth to the predicted objects and compute the standard *precision*, *recall*, and F_1 scores. For infrequent shape classes there can be zero predicted objects, in which case the recall is zero. As this leads to a division by zero in the calculation of the precision score, we handle this edge case by reporting n/a for both precision and F_1 scores. Similarly, we report a recall of n/a for classes that appear in the training, but not in the test set.

As the object detection step completes the recognition of shapes, we compute the macro shape F_1 score as the mean F_1 over all shape classes. Similarly, the micro shape F_1 score is computed as the F_1 over all shape predictions. This means that the micro score weights each class according to its relative frequency, and thus the score is biased towards more frequent classes.

Edge recognition metrics. To quantify the performance for edge recognition, we follow related work by determining if our approach is able to relate detected arrow objects to the correct (i.e., ground-truth) source and target shapes [14], [39]. We, again, compute precision, recall, and F_1 scores for this, where we consider an edge to be a true positive if both the source and target shapes are correctly identified.

Note that this means that edge recognition is affected by object detection accuracy for arrows, as well as for shapes: if an arrow object was not correctly detected, we cannot relate it to any shapes, whereas if a shape was not correctly detected, we cannot relate any edges to it, either.[12]

Label recognition metrics. We consider label recognition performance with respect to its two parts: textblock relation detection, for which we assess how well our approach relates textblocks to their corresponding shapes and edges, and textblock handwriting recognition, for which we evaluate the actual textual content detected in these textblocks.

For *textblock relation detection*, we follow the same evaluation procedure as for edge recognition, i.e., a label is a true positive if its textblock has been correctly detected and the associated shape or edge has been correctly identified. Note that, our conceptual contributions focus on this part of label recognition, whereas the rest primarily depends on an employed OCR service. Therefore, the overall label recognition performance of our approach that we report only includes these non-textual aspects.

For *textblock handwriting recognition*, we assess its two steps. We evaluate the *image word recognition* step by comparing the OCR performance to the ground-truth words annotated in an image. We consider a predicted word a true positive if its IoU score exceeds 50%, allowing us to report on precision, recall, and F_1 scores for the OCR service. In addition, we compare the predicted and ground-truth texts of the true positive words using the character error rate (CER), a common metric to evaluate HWR methods [60]. The CER is defined as the Levenshtein distance at character level between the prediction and ground truth, normalized by the ground-truth length. Prior to computing the CER, we lowercase the texts and replace line breaks with whitespaces,

12. Note that an edge is still considered correct if its associated shapes are incorrectly classified.

TABLE 3
Overall approach results

Approach	Shape		Edge		Label
	Micro F_1	Macro F_1	Micro F_1	Macro F_1	F_1
BPMN-Redrawer	89.2	73.7	63.1	47.5	—
Arrow R-CNN [22]	94.4	86.5	87.7	82.9	—
Sketch2BPMN [23]	94.8	87.3	90.7	86.2	—
DiagramNet [16]	95.6	**88.5**	85.5	78.8	—
Sketch2Process	**95.8**	88.3	**93.0**	**90.5**	**92.6**

as we observe inconsistent annotations for both. Finally, for the *textual content decoding* step, we assess how accurately our approach assigns words to the textblocks (and thus to labels). We evaluate this based on the ground-truth words, reflecting the accuracy of the decoding step in isolation, and based on the actually detected words, reflecting the end-to-end performance of the textblock handwriting recognition procedure.

Baselines. To demonstrate the efficacy of our approach, we compare its performance to related works. To this end, we train and evaluate the following systems on the `hdBPMN` dataset: BPMN-Redrawer [44], Arrow R-CNN [22], Sketch2BPMN [23], and DiagramNet [16] (see also Section 2). For the BPMN-Redrawer comparison, we use the training configurations provided in the open source implementation. For Arrow R-CNN, we use its default image augmentation methods, which are targeted at diagrams drawn on a white background. We also compare to the Sketch2BPMN approach, which extended Arrow R-CNN with additional augmentation methods and an improved rule-based edge relation procedure. Last, we compare to the DiagramNet [16] approach. DiagramNet does not predict arrow bounding boxes. Instead, the arrow bounding box are generated as the smallest bounding box that contains all predicted arrow keypoints. In the paper, the edge recognition evaluation metric considers neither arrow bounding boxes nor keypoints, and instead only considers if an arrow has been matched to the correct source and target shape. Our hypothesis is that, while DiagramNet is comparable to our approach in edge relation detection performance, it produces less accurate keypoints and bounding boxes. To verify this hypothesis, we compare our approach to DiagramNet using both evaluation procedures, the one proposed in our work, and the one used to evaluate DiagramNet originally [16].

6.2 Results

This section presents the results of our evaluation for the `hdBPMN` test set, first in terms of overall results, before taking a detailed look at the results per BPMN element class. In addition, we present a sensitivity analysis of the major parameters and the measured runtime of `Sketch2Process` and its major components.

Overall results and baselines. The overall results are presented in Table 3. As the cells with missing label F_1 scores indicate, our approach is the first work that addresses the recognition of all three diagram components. The results reveal that, for shape recognition, our approach is on par with DiagramNet, and outperforms all other approaches, especially on macro F_1. Note that micro and macro measures differ, because certain classes (e.g., *Tasks*) are much more common and easier to recognize than others (e.g., specific kinds of events). However, the overall trends are consistent across the two. For edge recognition, our approach considerably outperforms all other approaches, achieving both a micro and a macro F_1 score above 90. The rather low performance of the BPMN-Redrawer approach highlights that, especially for the recognition of hand-drawn edges, a dedicated network architecture is required.

As mentioned above, we also compared our edge recognition performance against DiagramNet [16] using the true positive definition from that work. Here, we observe that our approach has the highest macro F_1 (85.4), followed by DiagramNet (83.8) and Sketch2BPMN (80.8). These results indicate that, while DiagramNet underperforms all other approaches in edge recognition with our bounding box-based evaluation, it performs much better when only considering whether an edge actually exists between two shapes.

Regarding label recognition, we observe that our approach achieves an F_1 score of 92.6 (as indicated in Section 6.1, this corresponds to performance on textblock relation detection). Given that, to our knowledge, we are the first approach that addresses label recognition, we do not have prior work to compare to.

TABLE 4
Object detection results per class obtained for the test set

Group	Class	Prec.	Rec.	F_1	Count
Activity	Task	97.8	99.6	98.7	763
	Subprocess (collapsed)	96.2	80.6	87.7	31
	Subprocess (expanded)	n/a	0.0	n/a	3
	Call Activity	100.0	100.0	100.0	1
Event	Start Event	94.4	97.1	95.8	70
	Intermediate Event	n/a	n/a	n/a	0
	End Event	97.4	97.4	97.4	190
	Message Start Event	93.9	86.8	90.2	106
	Message Int. Catch E.	84.0	94.3	88.9	106
	Message Int. Throw E.	79.6	70.9	75.0	55
	Message End Event	81.1	76.9	78.9	39
	Timer Start Event	100.0	93.3	96.6	15
	Timer Intermediate E.	93.1	94.7	93.9	57
Gateway	Exclusive Gateway	97.6	98.0	97.8	247
	Parallel Gateway	96.1	96.8	96.4	126
	Inclusive Gateway	n/a	0.0	n/a	1
	Event-based Gateway	91.9	94.4	93.2	36
Collab.	Pool	96.6	99.0	97.8	203
	Lane	91.9	94.6	93.2	111
Data	Data Object	98.9	97.3	98.1	185
	Data Store	100.0	94.3	97.1	35
Arrow	Sequence Flow	97.6	95.6	96.6	1,887
	Message Flow	94.4	89.3	91.8	346
	Data Association	96.7	86.9	91.5	367
Text	Textblock	96.8	94.0	95.4	1,538
Overall	Macro avg.	94.3	84.8	89.3	6,518
	Micro avg.	95.9	95.1	95.5	6,518

Object detection. Table 4 provides detailed insights into the performance of our object detector, by depicting the results obtained per shape, arrow and text class. The table shows that our approach correctly recognizes the vast majority of objects, achieving an F_1 score of at least 90 for all arrow, text, and 14 out of 21 shape classes. For some shape types, the number of

TABLE 5
Edge recognition results per class obtained for the test set

Class	Prec.	Rec.	F_1	Count
Sequence Flow	95.9	93.5	94.7	1,887
Message Flow	91.0	85.3	88.1	346
Data Association	93.9	84.2	88.8	367
Macro avg.	93.6	87.7	90.5	2,600
Micro avg.	95.0	91.1	93.0	2,600

TABLE 6
Textblock handwriting recognition results obtained for the test set

Recognition task	Mean CER
Image word recognition	5.5
Textual content decoding (GT Words)	0.5
Textual content decoding (OCR Words)	8.8

TABLE 7
Textblock relation detection results per group obtained for the test set

	Predicted objects			Ground-truth objects			
Group	Prec.	Rec.	F_1	Prec.	Rec.	F_1	Count
Event	97.2	96.1	96.6	99.2	98.7	99.0	536
Gateway	n/a	0.0	n/a	n/a	0.0	n/a	3
Data element	98.1	94.5	96.3	100.0	98.6	99.3	220
Collaboration	93.8	87.8	90.7	100.0	98.7	99.4	312
Edge	92.4	83.3	87.6	97.5	92.7	95.1	467
Macro avg.	95.4	90.4	92.8	99.2	97.2	98.2	1,538
Micro avg.	95.3	90.1	92.6	99.0	96.7	97.8	1,538

data points is too low (in both the training and the test set), to sufficiently cover the spectrum of factors such as drawing styles and, therefore, to provide reliable evaluation results.

A post-hoc analysis of the results reveals that the most difficult task for our object detector is the correct classification of certain kinds of events. This comes as no surprise, though, since the difference between some of the eight kinds of events may only be due to marginal differences, such as a change in line thickness (start events), as well as different kinds of tiny envelopes (message events) and clocks (timer events). Especially in light of the diverse shapes in our dataset, as highlighted in Fig. 16, identifying such differences in hand-drawn models can already be highly complex for humans, let alone for an automated approach that lacks sufficient training examples for some of the rarer classes.

Edge recognition. The edge recognition results in Table 5 again demonstrate the overall strong performance of our approach, as well as that *sequence flows* (F_1 of 94.7) are easier to recognize than message flows (88.1) and data associations (88.8). To some extent, this can be attributed to the commonality of sequence flows and the fact that the latter two classes use dashed rather than continuous lines. However, it is also highly interesting to consider the different role of these edges from a process modeling perspective. Particularly, message flows connect (elements in) different pools, which are often placed relatively far from each other. This results in longer edges, which may also cross more nodes, and are, therefore, harder to analyze for an automated approach. For example, we observe that the median arrow path length is more than twice as high for message flows (447 pixels) as for sequence flows (152). For data associations, it is important to consider that elements related to the data perspective are often drawn last [61, p.177], whereas they also often are connected to multiple shapes, scattered throughout a model. These two factors thus commonly result in data associations that cross other edges or even shapes, which complicates their recognition.

By comparing the arrow object detection results in Table 4 with the edge recognition results in Table 5, we can quantify the effectiveness of our edge relation detection procedure. Concretely, the difference between the object detection and the edge recognition measures are arrows that were correctly detected, but where either the source or target shape was not correctly identified. Here, we observe that the F_1 delta is lowest for sequence flows (-1.9), and increases slightly for data associations (-2.7) and message flows (-3.7). For comparison, we also computed these differences for our rule-based approach in Sketch2BPMN [23]. Here, we find that the F_1 delta is much higher for sequence flows (-3.5) and message flows (-10.4), and slightly lower for data associations (-2.5). Overall, this shows the effectiveness of our improved edge recognition method, especially for complex arrows such as message flows.

Label recognition. In the following, we report the evaluation results of both parts of our label recognition procedure.

Textblock relation detection. Table 7 shows how well our approach is able to relate textblocks to corresponding shapes and edges, for each of the element groups. The table shows that event labels (F_1 of 96.6) and data element labels (F_1 of 96.3) are easier to detect and relate than collaboration labels (F_1 of 90.7) and edge labels (F_1 of 87.6). Since we do not (need to) detect activity labels through dedicated textblocks, we do not report results for activities.

To assess the accuracy of the relation detector independent of object detection errors, we also evaluate it using the ground-truth objects. Here, the table shows near perfect results for four out of five category groups, which demonstrates the effectiveness of our textblock relation detector. Overall, the recognition of edge labels is the most difficult task for our approach, which can be attributed to the fact that there are often multiple arrows in proximity to a textblock. For collaboration shapes (pools and lanes), the near-perfect results when using the ground-truth objects indicate that the errors are largely due to textblock and collaboration object detection errors. Here, a post-hoc analysis shows that the most difficult task is the detection of nested lanes and their labels, where the bounding boxes of parent and child lane are very similar (IoU of up to 97%).

Textblock handwriting recognition. Turning to the textual content within the detected textblocks, we assess both steps of the handwriting recognition part:

For the *image word recognition* step, the OCR service achieves a precision of 69.6% and a recall of 90.0% when it comes to detecting words in images (irrespective of their textual content). The low precision is largely due to single-character, false positive "words" that actually correspond to parts of drawn shapes or edges (e.g., "X" characters stemming from exclusive gateway symbols and dashes from dashed arrows), as mentioned in Section 4.3. Such false

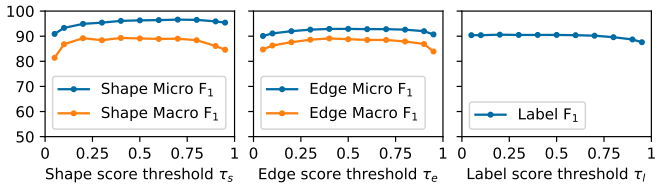


Fig. 17. Score threshold analyses conducted on the validation set

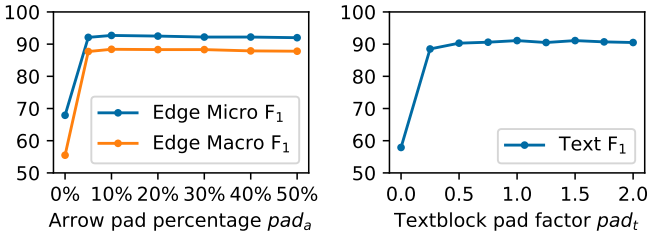


Fig. 18. Region size analyses conducted on the validation set

positives are not an issue for our approach, though, since they are eliminated during the textual content decoding step, given that they are not located within a textblock. For the true positive words identified by the OCR service, we observe a mean CER of 5.5%, as shown in Table 6.

For the *textual content decoding* step, the mean CER of 0.5% obtained when using the ground-truth words (GT words) shows that this step is very accurate in isolation. This means that if the OCR service would be perfect, the labels of the detected textblocks would be near-perfect as well. In terms of end-to-end performance of textblock handwriting recognition, we observe a CER of 8.8% when using the actually detected OCR words as input for textual content decoding. On top of character recognition errors made by the OCR service in true positive words (captured in the 5.5% CER for image word recognition), the 8.8% CER is also caused by words that the OCR service failed to detect, i.e., false negatives.

In summary, the vast majority of errors that happen during textblock handwriting recognition directly or indirectly stem from errors of the OCR service. It is, furthermore, worth highlighting that this does not imply that major corrections to the generated models are required. For instance, in the running example (cf. Fig. 15), only 18 of the 31 shapes and edges actually have labels, and among those, the median edit distance is 1, which means that only minor edits are required to fix the labels in the recognized model.

Sensitivity analysis. As mentioned, we analyze the sensitivity of the main parameters of `Sketch2Process`, including the settings for the different score thresholds, the arrow and textblock region sizes, and the image augmentation methods.

Score thresholds. Fig. 17 shows the results for the score thresholds τ_s, τ_e, and τ_l, of which we apply each in one component. We find that each threshold obtains similar results in the range from 0.4 to 0.7, where the difference between the highest and lowest F_1 score is at most 0.6. The performance only degrades substantially when using a very small or a very large threshold. This shows that our approach

TABLE 8
Image augmentation ablation study conducted on the validation set

Augmentation	Shape Mi. F_1	Shape Ma. F_1	Edge Mi. F_1	Edge Ma. F_1	Label F_1
No augmentation	92.2	81.8	86.1	78.2	86.0
Detectron2 (resize & hor. flip)	94.2	85.3	89.3	83.1	88.5
Sketch2Process (ours)	**95.7**	**87.9**	**92.8**	**88.0**	**90.6**

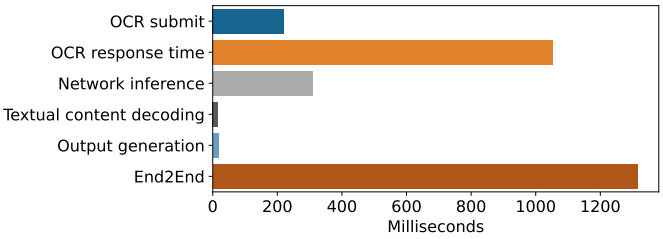


Fig. 19. Median runtime measures obtained for the validation set

is very robust to minor score threshold changes. Given the analysis, we decided to simply set all three thresholds to 0.5.

Region sizes. Fig. 18 shows the results for the region size configurations. For the arrow pad percentage pad_a, we observe that the edge F_1 scores are substantially lower when applying only the minimum padding, are almost identical in the range from 10% to 30%, and then slightly drop at 40% and 50%. As the number of edge candidates increases with the arrow region size, which leads to increased inference time, we use $pad_a = 10\%$ in our approach. Regarding the textblock region size, we observe that the results for the pad factor pad_t are very similar in the range from 0.75 to 2.0, with an F_1 between 90.5 and 91.1. As expected, the F_1 drops substantially when using no padding ($pad_t = 0$), as the majority of textblocks (53% in the training set) do not intersect with the shape or edge that they label. Similar to the arrow region size, the number of relation candidates increases with the textblock region size. We therefore employ $pad_t = 1.0$ for our approach.

Image augmentation. The results of our ablation study in Table 8 show the benefits of using image augmentation. Compared to using no augmentation, the performance of our approach consistently improves when using the default object detection augmentations provided by the Detectron2 library (random resizing and horizontal flipping), leading to F_1-score increases between 2.0 and 4.9. A further performance improvement of the same magnitude can be observed with the full `Sketch2Process` image augmentation pipeline, leading to further F_1-score improvements between 1.5 and 4.9. Notably, the recognition of the challenging edge categories (message flow and data association) benefits most from augmentation, which leads to the large edge macro F_1-score improvements (from 78.2 to 88.0).

Runtime. Finally, Fig. 19 shows the median runtime of the `Sketch2Process` components. Given an image to be processed, we first send the image to the OCR service (220 ms). We find that the time until the OCR results are available (1,125 ms) always exceeds the network inference time. Therefore, while waiting for the OCR results, we compute the results of all networks, i.e., the object detection,

edge relation detection, and textblock relation detection networks, which on average takes 308 ms on a Tesla V100 GPU. After network inference, we periodically check for the OCR results, which takes 695 ms on average. Once the OCR results are available, we run the textual content decoding procedure (17 ms) to identify the textual content of each textblock, and create the final BPMN XML as described in Section 4.4 (17 ms). On average, our approach processes an image in 1.3 seconds, most of which is spent on waiting for the OCR results.

7 Discussion

In this section, we reflect on the implications, for research and practice, and limitations of our work.

Implications. As for *implications for research*, the architecture and design of our approach may inform other research on sketch recognition addressing conceptual models. It is important to note that BPMN 2.0 is one of the most symbol-rich and complex modeling notations used for formal system specifications. Given the accuracy of our approach on BPMN models, we are, therefore, confident that the conceptual ideas presented in this paper can be transferred to also recognize hand-drawn models of other notations, such as UML. As for *implications for practice*, our approach allows organizations to benefit from the advantages of hand-drawn models, such as freedom and easy collaboration, without suffering from the consequences associated with the manual transformation. Our evaluation showed that our approach is highly accurate and, hence, only minor edits are required to fix the automatically recognized model. As a result, our approach may help organizations to establish a novel and more efficient approach to requirements elicitation and collection that starts on a whiteboard or even on a piece of paper. It is also worth noticing that our approach is highly accurate on computer-generated BPMN models and outperforms the existing state-of-the-art approach for this task [44][13]. Hence, our approach can also be used to support standardization efforts where BPMN models have been created with various tools and / or are only available as picture-based versions.

Limitations. Naturally, our work is subject to a number of limitations. First, it is important to highlight that the presented evaluation results should be considered in light of the characteristics of the `hdBPMN` dataset. Since the dataset was established using more than 100 different participants, spans models of varying size and complexity, and overall has images that differ a lot in terms of their characteristics and quality (cf., Section 5.3), we are confident that our results have a high external validity. Nevertheless, in practice, one may still encounter models that are, e.g., larger, more complex, or contain certain symbols less or more frequently.

Second, our label recognition component partially depends on the quality of the employed OCR service, which can lead to errors when dealing with handwriting (as seen in Section 6.2). However, this is not a critical issue for the conceptual validity or practical usefulness of our approach. Specifically, our work is independent of a particular service, which means that the employed OCR service can be replaced with improved versions in the future. Furthermore, the mistakes from this component can be quickly fixed manually or through a post-processing step that, for instance, builds on a dictionary.

Third, we would like to highlight that the successful application of our approach to other types of conceptual models requires retraining our approach on a respective dataset. While this is not a limitation per se, we recognize that the creation and manual annotation of such a dataset comes with considerable effort. However, this is a one-time effort, which, given the advantages of automatically recognizing model sketches, can be considered a good investment in many scenarios.

13. The details on the respective experiments are provided in the supplementary material.

8 Conclusion

In this paper, we addressed the problem of sketch recognition of hand-drawn BPMN models. Given that existing solutions for this problem left considerable room for improvement in terms of scope and recognition quality, we presented `Sketch2Process`, the first approach for comprehensive, end-to-end sketch recognition of BPMN models. `Sketch2Process` takes an image of a hand-drawn BPMN model as input and automatically transforms it into a format compatible with existing process modeling tools. For its training and evaluation, we created the `hdBPMN` dataset, consisting of 704 hand-drawn and manually annotated BPMN models. Experiments conducted on this dataset demonstrated that `Sketch2Process` is highly accurate and consistently outperforms the state of the art.

We identify several directions for future work. First, the application of OCR to hand-drawn BPMN models revealed the limitations of existing off-the-shelf OCR services. In future work it would, therefore, be interesting to investigate how OCR solutions can be improved with respect to the peculiarities of labels in hand-drawn BPMN models, such as text that is not written horizontally, but in arbitrary angles. Second, it is interesting to explore how well `Sketch2Process` can be adapted to other requirements modeling notations such, as UML. While this certainly requires a retraining, the question is how big a respective data set needs to be and to what extent generic aspects (e.g., edge relations) can be learned and transferred from the `hdBPMN` dataset. Finally, it is important to investigate the use of `Sketch2Process` in practice, striving to identify ways in which sketch recognition can further support tasks such as the efficient elicitation of requirements.

References

[1] C. Ouyang, M. Dumas, W. M. V. D. Aalst, A. H. T. Hofstede, and J. Mendling, "From business process models to process-oriented software systems," *ACM transactions on software engineering and methodology (TOSEM)*, vol. 19, no. 1, pp. 1–37, 2009.

[2] H. Leopold, J. Mendling, and A. Polyvyanyy, "Supporting process model validation through natural language generation," *IEEE Transactions on Software Engineering*, vol. 40, no. 8, pp. 818–840, 2014.

[3] D. van der Linden, I. Hadar, and A. Zamansky, "What practitioners really want: requirements for visual notations in conceptual modeling," *Software & Systems Modeling*, vol. 18, no. 3, pp. 1813–1831, 2019.

[4] R. Waszkowski, "Low-code platform for automating business processes in manufacturing," *IFAC-PapersOnLine*, vol. 52, no. 10, pp. 376–381, 2019.

[5] P. Vincent, K. Iijima, M. Driver, J. Wong, and Y. Natis, "Magic quadrant for enterprise low-code application platforms," *Gartner report*, 2019.

[6] T. Gorschek, E. Tempero, and L. Angelis, "On the use of software design models in software development practice: An empirical investigation," *Journal of Systems and Software*, vol. 95, pp. 176–193, 2014.

[7] O. Badreddin, R. Khandoker, A. Forward, and T. Lethbridge, "The evolution of software design practices over a decade: A long term study of practitioners." *J. Object Technol.*, vol. 20, no. 2, pp. 1–1, 2021.

[8] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko, "Let's Go to the Whiteboard: How and Why Software Developers Use Drawings," in *SIGCHI*, 2007, pp. 557–566.

[9] J. Walny, S. Carpendale, N. Henry Riche, G. Venolia, and P. Fawcett, "Visual thinking in action: Visualizations as used on whiteboards," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2508–2517, 2011.

[10] C. Bartelt, M. Vogel, and T. Warnecke, "Collaborative creativity: From hand drawn sketches to formal domain specific models and back again." in *MoRoCo@ ECSCW*, 2013, pp. 25–32.

[11] P. Antunes, D. Simões, L. Carriço, and J. A. Pino, "An end-user approach to business process modeling," *Journal of Network and Computer Applications*, vol. 36, no. 6, pp. 1466–1479, 2013.

[12] R. Brown, J. Recker, and S. West, "Using virtual worlds for collaborative business process modeling," *Business Process Management Journal*, 2011.

[13] G. Casella, V. Deufemia, V. Mascardi, G. Costagliola, and M. Martelli, "An agent-based framework for sketched symbol interpretation," *Journal of Visual Languages & Computing*, vol. 19, no. 2, pp. 225–257, 2008.

[14] M. Bresler, D. Průša, and V. Hlaváč, "Online recognition of sketched arrow-connected diagrams," *Int. Journal on Document Analysis and Recognition*, vol. 19, no. 3, pp. 253–267, 2016.

[15] F. Brieler and M. Minas, "Recognition and processing of hand-drawn diagrams using syntactic and semantic analysis," in *Working conf. on Advanced visual interfaces*, 2008, pp. 181–188.

[16] B. Schäfer and H. Stuckenschmidt, "DiagramNet: Hand-Drawn Diagram Recognition Using Visual Arrow-Relation Detection," in *Int. Conf. on Document Analysis and Recognition*, 2021, pp. 614–630.

[17] M. Zapp, P. Fettke, and P. Loos, "Towards a Software Prototype Supporting Automatic Recognition of Sketched Business Process Models," *Wirtschaftsinformatik 2017*, 2017.

[18] C. Gonzalez Moyano, L. Pufahl, I. Weber, and J. Mendling, "Uses of business process modeling in agile software development projects," *Information and Software Technology*, vol. 152, p. 107028, 2022.

[19] A. Sahay, A. Indamutsa, D. Di Ruscio, and A. Pierantonio, "Supporting the understanding and comparison of low-code development platforms," in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2020, pp. 171–178.

[20] U. Frank, P. Maier, and A. Bock, "Low code platforms: promises, concepts and prospects. a comparative study of ten systems," ICB-Research Report, Tech. Rep., 2021.

[21] J. Wong, K. Iijima, A. Leow, A. Jain, and P. Vincent, "Magic quadrant for enterprise low-code application platforms," *Gartner report*, 2021.

[22] B. Schäfer, M. Keuper, and H. Stuckenschmidt, "Arrow R-CNN for handwritten diagram recognition," *Int. Journal on Document Analysis and Recognition*, Feb. 2021.

[23] B. Schäfer, H. van der Aa, H. Leopold, and H. Stuckenschmidt, "Sketch2BPMN: Automatic recognition of hand-drawn BPMN models," in *Int. Conf. on Advanced Information Systems Engineering*. Springer, 2021, pp. 344–360.

[24] B. Yu and S. Cai, "A domain-independent system for sketch recognition," in *Int. conf. on Computer graphics and interactive techniques in Australasia and South East Asia*, 2003, pp. 141–146.

[25] T. A. Hammond and R. Davis, "Recognizing interspersed sketches quickly," *Graphics Interface 2009 (GI '09)*, 2009.

[26] T. E. Johnson, "Sketchpad iii: a computer program for drawing in three dimensions," in *Spring joint computer conf.*, 1963, pp. 347–353.

[27] Q. Chen, J. Grundy, and J. Hosking, "Sumlow: early design-stage sketching of uml diagrams on an e-whiteboard," *Software: Practice and Experience*, vol. 38, no. 9, pp. 961–994, 2008.

[28] F. D. Julca-Aguilar and N. S. T. Hirata, "Symbol Detection in Online Handwritten Graphics Using Faster R-CNN," in *IAPR Int. Workshop on Document Analysis Systems*, Apr. 2018, pp. 151–156.

[29] M. Schmidt and G. Weber, "Recognition of multi-touch drawn sketches," in *Int. Conf. on Human-Computer Interaction*. Springer, 2013, pp. 479–490.

[30] A. Coyette, S. Schimke, J. Vanderdonckt, and C. Vielhauer, "Trainable sketch recognizer for graphical user interface design," in *IFIP Conf. on Human-Computer Interaction*. Springer, 2007, pp. 124–135.

[31] T. Kurtoglu and T. F. Stahovich, "Interpreting schematic sketches using physical reasoning," in *AAAI Spring Symposium on Sketch Understanding*, vol. 7885, 2002.

[32] O. Bergig, N. Hagbi, J. El-Sana, and M. Billinghurst, "In-place 3d sketching for authoring and augmenting mechanical systems," in *Int. Symposium on Mixed and Augmented Reality*. IEEE, 2009, pp. 87–94.

[33] B. Paulson and T. Hammond, "Paleosketch: accurate primitive sketch recognition and beautification," in *Int. conf. on Intelligent user interfaces*, 2008, pp. 1–10.

[34] F. Brieler and M. Minas, "A model-based recognition engine for sketched diagrams," *Journal of Visual Languages & Computing*, vol. 21, no. 2, pp. 81–97, 2010.

[35] V. Deufemia, M. Risi, and G. Tortora, "Hand-drawn diagram recognition with hierarchical parsing: An experimental evaluation," in *Information systems: crossroads for organization, management, accounting and engineering*. Springer, 2012, pp. 217–225.

[36] F. Julca-Aguilar, H. Mouchère, C. Viard-Gaudin, and N. S. T. Hirata, "A general framework for the recognition of online handwritten graphics," *Int. Journal on Document Analysis and Recognition*, 2020.

[37] G. Costagliola, M. De Rosa, and V. Fuccella, "Local context-based recognition of sketched diagrams," *Journal of Visual Languages & Computing*, vol. 25, no. 6, pp. 955–962, 2014.

[38] J. Wu, C. Wang, L. Zhang, and Y. Rui, "Offline Sketch Parsing via Shapeness Estimation," in *Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 2015.

[39] M. Bresler, D. Průša, and V. Hlaváč, "Recognizing Off-Line Flowcharts by Reconstructing Strokes and Using On-Line Recognition Techniques," in *ICFHR*, 2016, pp. 48–53.

[40] D. Doermann, Jian Liang, and Huiping Li, "Progress in camera-based document image analysis," in *Int. Conf. on Document Analysis and Recognition*, 2003, pp. 606–616 vol.1.

[41] S. Bhowmik, R. Sarkar, M. Nasipuri, and D. Doermann, "Text and non-text separation in offline document images: A survey," *Int. Journal on Document Analysis and Recognition*, vol. 21, no. 1-2, pp. 1–20, Jun. 2018.

[42] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," in *Conf. on Neural Information Processing Systems (NeurIPS)*, 2015, pp. 91–99.

[43] B. Schäfer and H. Stuckenschmidt, "Arrow R-CNN for Flowchart Recognition," in *2019 Int. Conf. on Document Analysis and Recognition Workshops*, 2019.

[44] A. Antinori, R. Coltrinari, F. Corradini, F. Fornari, B. Re, and M. Scarpetta, "Bpmn-redrawer: from images to bpmn models," in *BPM (Demos)*, 2022.

[45] Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo, and R. Girshick, "Detectron2," https://github.com/facebookresearch/detectron2, 2019.

[46] F. Pittke, H. Leopold, and J. Mendling, "Automatic detection and resolution of lexical ambiguity in process models," *IEEE Transactions on Software Engineering*, vol. 41, no. 6, pp. 526–544, 2015.

[47] S. Chakraborty, S. Sarker, and S. Sarker, "An exploration into the process of requirements elicitation: A grounded approach," *Journal of the association for information systems*, vol. 11, no. 4, p. 1, 2010.

[48] H. Leopold, J. Mendling, and O. Günther, "Learning from quality issues of bpmn models from industry," *IEEE software*, vol. 33, no. 4, pp. 26–33, 2015.

[49] K. He, G. Gkioxari, P. Dollar, and R. Girshick, "Mask R-CNN," in *Int. Conf. on Computer Vision (ICCV)*, 2017, pp. 2961–2969.

[50] J. Yang, J. Lu, S. Lee, D. Batra, and D. Parikh, "Graph R-CNN for Scene Graph Generation," in *Proceedings of the European Conf. on Computer Vision (ECCV)*, 2018, pp. 670–685.

[51] J. Zhang, K. J. Shih, A. Elgammal, A. Tao, and B. Catanzaro, "Graphical contrastive losses for scene graph parsing," in *Conf. on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[52] B. Davis, B. Morse, S. Cohen, B. Price, and C. Tensmeyer, "Deep Visual Template-Free Form Parsing," in *Int. Conf. on Document Analysis and Recognition*, Sep. 2019, pp. 134–141.
[53] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv:1704.04861 [cs]*, Apr. 2017.
[54] D. H. Douglas and T. K. Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature," *Cartographica*, vol. 10, no. 2, pp. 112–122, Dec. 1973.
[55] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.
[56] A. Buslaev, V. I. Iglovikov, E. Khvedchenya, A. Parinov, M. Druzhinin, and A. A. Kalinin, "Albumentations: Fast and Flexible Image Augmentations," *Information*, vol. 11, no. 2, p. 125, 2020.
[57] P. Gervais, T. Deselaers, E. Aksan, and O. Hilliges, "The DIDI dataset: Digital Ink Diagram data," *arXiv:2002.09303 [cs]*, 2020.
[58] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, 2019, pp. 8024–8035.
[59] T.-Y. Lin, P. Dollar, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature Pyramid Networks for Object Detection," in *Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 2117–2125.
[60] J. A. Sánchez, V. Romero, A. H. Toselli, M. Villegas, and E. Vidal, "ICDAR2017 Competition on Handwritten Text Recognition on the READ Dataset," in *ICDAR*, 2017, pp. 1383–1388.
[61] M. Dumas, M. L. Rosa, J. Mendling, and H. A. Reijers, *Fundamentals of Business Process Management*, 2nd ed. Springer, 2018.

Henrik Leopold Henrik Leopold is an Associate Professor at the Kühne Logistics University (KLU) and a Senior Researcher at Hasso Plattner Institute (HPI) at the Digital Engineering Faculty, University of Potsdam. He obtained his PhD degree in Information Systems from the Humboldt University Berlin, Germany. For his doctoral thesis, he received the German TARGION Award for the best dissertation in the field of strategic information management. Before joining KLU/HPI, Henrik held positions at the Vrije Universiteit Amsterdam as well as WU Vienna. His research is mainly concerned with leveraging technology from the field of artificial intelligence to develop techniques for process mining, process analysis, and process automation. Henrik has published over 100 scientific contributions, among others, in IEEE Transactions on Software Engineering, IEEE Transactions on Knowledge and Data Engineering, ACM Transactions on Management Information Systems, Decision Support Systems, and Information Systems.

Bernhard Schäfer Bernhard Schäfer is currently pursuing the Ph.D. degree with the Data and Web Science Group at the University of Mannheim in cooperation with SAP SE. He obtained his MSc degree in business informatics from the University of Mannheim in 2014. Before joining SAP SE, he worked as a data science consultant for inovex GmbH, Karlsruhe, Germany. His research interests include graphics recognition, data science, and artificial intelligence.

Heiner Stuckenschmidt Heiner Stuckenschmidt is a Full Professor in the Data and Web Science Group at the University of Mannheim, Germany. He received the Ph.D. degree from Vrije Universiteit Amsterdam in 2003, where he was a Post-Doctoral Researcher with the Artificial Intelligence Department. His group is performing fundamental and applied research in knowledge representation formalisms with a focus on reasoning techniques for information extraction and integration as well as machine learning for advanced decision making. He has published over 200 papers, including about 50 papers in international journals and over 100 papers in international peer reviewed conferences in computer science.

Han van der Aa Han van der Aa is a junior professor in the Data and Web Science Group at the University of Mannheim, Germany. Before that, he was an Alexander von Humboldt Fellow, working as a postdoctoral researcher in the Department of Computer Science at the Hum-boldt-Universität zu Berlin. He obtained a PhD from the Vrije Universiteit Amsterdam in 2018. His research interests include business process modeling, process mining, natural language processing, and complex event processing. His research has been published in renowned journals such as IEEE Transactions on Knowledge and Data Engineering, Decision Support Systems, and Information Systems, and at international conferences including BPM, CAiSE, SIGMOD, and IEEE ICDE.