http://dx.doi.org/10.25673/111736

From Specification to Test Cases: A State-Machine-Based Approach using Image Recognition

Björn Otto[1], Robin Gröpler[2], Karsten Meinecke[3], Tobias Kleinert[4]

Abstract: Software testing enables the assessment and assurance of software quality. However, writing test cases manually is time-consuming and error-prone. To avoid this, test cases can be generated automatically using Model-based Testing (MBT). MBT derives tests from a test model like a finite state machine (FSM). Such FSMs are often part of specifications, but unfortunately, often only provided as image and not in machine-readable form. Therefore, in this work we present an approach to extract machine-readable representations of FSMs from images automatically using Neural Networks. Additionally, we evaluate the applicability of our approach using a real-world specification to generate test cases from it.

The number of software-intensive systems has drastically increased over the last decades. With them, the complexity of developing and maintaining software-intense systems has also increased. However, software quality measures have to keep up with these changes, which led to a huge number of software testing methods and approaches.

One key approach is to reduce the effort of defining and implementing tests manually. This can be achieved by generating tests in an automated or at least semi-automated way. Among various other methods, Model-based testing (MBT) has been shown to be suitable for test case generation [UPL12].

In MBT, tests are derived from a model. Here, one common type of model are finite state machines (FSM). From FSMs, tests can be derived by following paths through it. This approach has many advantages. First, manual effort is drastically reduced. Second, tests are parameterizable by defining a coverage criterion (like covering all nodes or transitions in the FSM). And last, the tests are efficient, in such that multiple tests avoid testing the same aspect of the application.

The crucial part of MBT is defining the model used for test generation. This can be done manually, which comes with some effort. Fortunately, manually defining the test model is often not needed, since many specifications already include state machines as part of the

[1] Institute for Automation and Communication, Werner-Heisenberg-Straße 1, 39106 Magdeburg, Germany bjoern.otto@ifak.eu
[2] Institute for Automation and Communication, Werner-Heisenberg-Straße 1, 39106 Magdeburg, Germany robin.groepler@ifak.eu
[3] Institute for Automation and Communication, Werner-Heisenberg-Straße 1, 39106 Magdeburg, Germany karsten.meinecke@ifak.eu
[4] RWTH Aachen University, Chair of Information and Automation Systems for Process and Material Technology, Turmstraße 46, 52064 Aachen, Germany kleinert@plt.rwth-aachen.de

system or behavioral description. However, the state machines are often rendered as images and are therefore not machine-readable.

To address this issue, this work will provide a two-fold contribution: First, we will provide an algorithm to automatically extract FSMs from images in human-readable specifications. Second, we will demonstrate the usefulness of our approach by automatically generating test cases from the extracted FSMs. Connecting these steps will result in the setup shown in Figure 1.

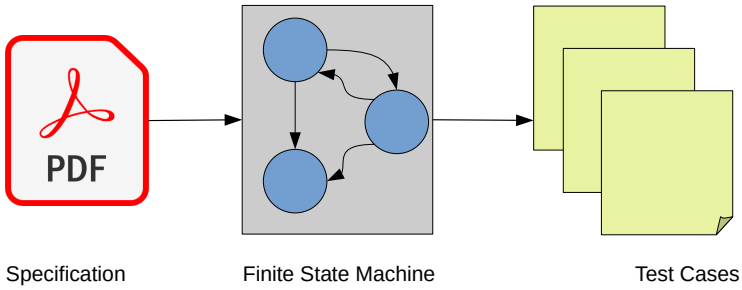


Fig. 1: Our test setup

This work is structured as follows: Section 1 lists related work, in Section 2, we describe our approach, which we evaluate in Section 3 using a fieldbus-related case study. Section 4 gives a conclusion.

1 Related Work

Our approach focuses on an end-to-end solution which yields test cases directly from a given specification by using image recognition. To best of our knowledge, this has not been studied in literature. Therefore, we focus on work related to individual phases of our approach.

1.1 Diagram Extraction

Diagram extraction has been studied extensively in literature. Most work focuses on extracting UML diagrams from images. The authors of ReSECDI[Ch22] present a method to extract class information from a rendered UML class diagram. They combine shape detection techniques (lines, rectangles) to extract the required information. They evaluate the applicability on 80 images in total.

Other work focuses on piping and instrument diagrams (P&ID) like [Yu20]. In [GZS20], the authors leverage Faster Regional Convolutional Neural Networks (Faster RCNN) to analyze P&IDs. They evaluate their approach on a commercial nuclear power plant.

The authors of [KC13] present an approach with a similar architecture to ours. Their tool Img2UML extracts information from UML class diagrams in three phases: class detection, text recognition and relationship detection. The authors validate their tool using 10 different images.

Block diagrams are analyzed in [BL22]. Here, the authors extract series of triples from rendered block diagrams. A large language model then uses these triplets to summarize the contents of the diagram.

1.2 Model-Based Testing

Model-based testing is a well-studied field in research[Di07]. Especially FSMs have been shown to be a useful type of model for test case generation[LY96]. Model-based testing is applicable in a variety of fields. However, as our case study focuses on testing a network protocol, we consider work in this field in the following, only. In [TAB17], the authors present a model-based method to test MQTT brokers. Other approaches like [PA09] test TCP/IP implementations using model-based testing.

2 Approach

Our approach is divided into two stages: First, FSMs are extracted from the specification using image processing. Second, these FSMs are leveraged to generate test cases using model-based testing.

2.1 FSM Extraction

To goal of this step is to extract all FSMs out of the specification. For this, we first extract all images out of the specification. We then apply the pipeline shown in Figure 2 to each of the extracted images.

First, we segment the image to obtain a segmentation mask as shown in Figure 3. This mask reveals where nodes and edges can be found in the input image. Given this mask, we then focus on the patches masked as nodes. We consider each continuous region a node. Additionally, we can extract the node's text using OCR. Finally, we need to check, which nodes are connected by edges. We consider two nodes as connected, if there is a continuous path of pixels labeled as edge between them.

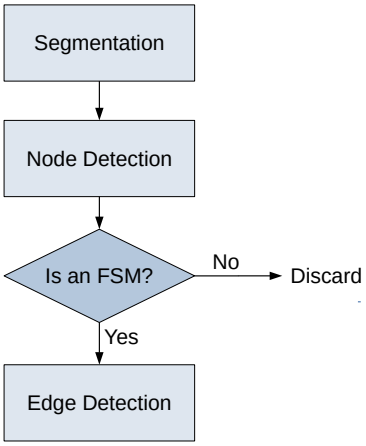
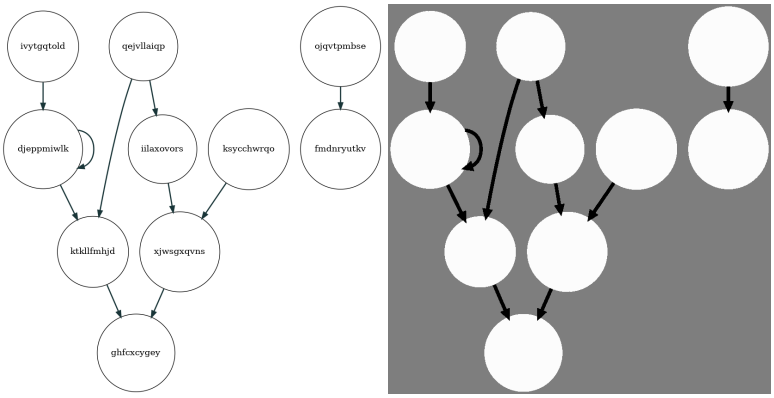


Fig. 2: FSM extraction

2.1.1 Segmentation

To goal of the segmentation step is to generate a mask image as shown in figure 3. The mask image is of the same size as the input image. For each input pixel the corresponding mask pixel has a value of 0, 1 or 2, where 0 indicates background, 1 a node and 2 an edge.



(a) Input Image (b) Segmentation Mask

Fig. 3: Image segmentation

For the image segmentation, we use a Neural Network[Gu18]. Our network is a modified U-Net[Si21], which means it consists of an encoder and a decoder. The encoder part feeds the input image through 4 layers, each one halving the size of the image from 128x128 to 4x4 pixels. This way, the image's features are compressed to 16 values. For the encoder

part, we use a pre-trained net, namely the MobileNetV2[Sa18], which is a state-of-the-art model suited for image segmentation. The decoder part does the opposite then. It consists of 4 layers doubling the size of the image, again up to a size of 64x64 pixels. The final convolution layer then yields the final segmentation mask of size 128x128 pixels. As the trained model can only be applied to image of exactly 128x128 pixels, we need to split the input image into overlapping patches of this size and merge the segmentation results afterwards.

Training our neuronal network required a reasonable amount of training data. We generated this data synthetically as follows: First we generated a random adjacency matrix. We then serialized the matrix using the dot language. Here, we also selected random colors, shapes and line widths for nodes and edges. Finally, we rendered the dot file using the graph layout engine Graphviz[El04]. Additionally, we rendered the segmentation mask. By repeating this process, we were able to generate 10.000 images for training with least effort.

For training, we used the TensorFlow[PNW20] library. As the edges naturally occupy less pixels than nodes, we weighted them 5 times more, so that they were not simply ignored during optimization. We have chosen Sparse Categorical Crossentropy as loss function and trained the model over 20 epochs. This way, we were able to achieve a (pixel-wise) accuracy of 95% on our validation set.

2.1.2 Node Detection

The goal of the node detection step is to generate a list of node masks and their enclosed text (see Figure 4). For this, we first iterate over the pixels of the segmentation mask. For each pixel masked as node, we check if it is connected directly to another node pixel. If this is not the case, the pixel is assigned a new node number. Otherwise it is assigned the node number of the connected pixel. Using a disjoint-set data structure, this can be implemented efficiently.

Along with the pixels of each node we also obtain its enclosed text using Optical Character Recognition (OCR). As specifications often contain lots of other images besides actual FSMs, we also decide which images to discard at this point. We reject all images which have less than 2 nodes. Furthermore, we reject images where the node's pixels occupy less than 20% or more than 80% of the image.

2.1.3 Edge Detection

In the last step, we decide which nodes we consider being connected by edges in the image. For this, we iterate over the pixels of the segmentation mask, again. This time, however, we number the *edge* pixels in the same way we did for the node pixels in the step before. Next, we iterate the border pixels of each node to find out, to which edges each node is connected

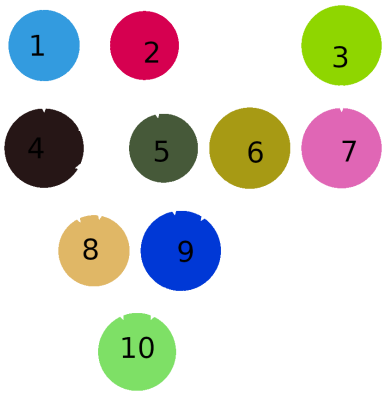


Fig. 4: Nodes mask

to. Finally, we can mark two nodes as being connected, if they are connected to at least one common edge.

To detect the direction of the edge, we use the following simple heuristic: We count the number of edge pixels adjacent to the node's pixels at each end of the edge. We then consider the side with more pixels to be the end, as it is most probably the arrow head.

2.2 Test Case Generation

For test case generation, we rely on a coverage-based approach[OK22]. For each extracted FSM, we generate a set of paths, which covers all nodes of the FSM[5]. Each path corresponds to one test case. For this, we follow the path and collect all nodes with their associated OCR-text in an array. We then pass this array to the final test execution. It interprets the node texts and executes operations accordingly. As this highly depends on the use-case, we cannot give a generalized approach for execution at this point.

For example, given the FSM in figure 5, we could obtain two test cases: `[A, B, D]` and `[A, C]`. It is then up to the text execution to map the labels `A, B, C, D` to actual test inputs and checks.

3 Case Study

To assess the feasibility of our approach, we applied it to the IEC 61158-6-10 standard. This standard describes application layer protocols for fieldbusses. For this case study, we focused on the Profinet fieldbus. As input for our approach, we used the document given in

[5] Some nodes may be covered multiple times

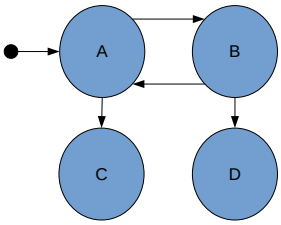


Fig. 5: Example FSM

pdf format. It consists of 1037 pages containing 274 figures in total. A few of these figures are actual state machines as the ones shown in Figure 6.

At first we converted each page to an image yielding 1037 images in total. Obviously, it would be better to consider only *figures*. However, we found these difficult to detect, because they are embedded as vector graphics within the text.

We then applied our image processing approach as described in Section 2.1. This resulted in 94 FSMs. For the test case generation and execution, we focused on the two FSMs shown in Figure 6. However, we still had to adjust the FSMs generated by our algorithm manually for two reasons: first, the extraction missed some nodes and edges, especially the intersecting ones. Second, we had to remove the loop from state K to the beginning as it would lead to only one test case iterating all the states multiple times. Finally, we used the Fences library[6] to select concrete paths through the FSMs, so that all transitions are covered. The results are shown in Table 1.

Tab. 1: Generated Test Cases

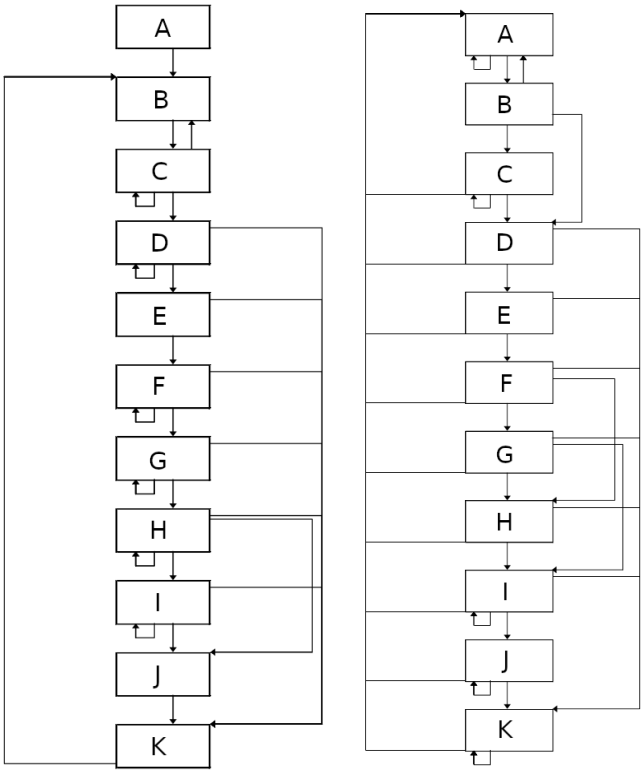
Target	States	Test Cases
Device	11	8
Controller	11	10

For the actual test execution, IEC 61158-6-10 already gives hints, how the states shall be implemented. Following this guides would result in complete test cases, which is out of the scope of this work.

4 Conclusion

In this work, we presented an end-to-end method for automated test case generation directly based on a specification. For this, we presented an approach to extract FSMs out of a specification and then use these to derive test cases. We evaluated the feasibility of our approach using the IEC 61158-6-10 standard. Evaluation showed, that our approach still

[6] https://github.com/ifak/fences



(a) Device (b) Controller

Fig. 6: State machines describing the interaction between a Profinet device and controller (from IEC 61158-6-10, node labels are simplified)

needs some manual effort. This has several minor reasons. First, the FSM extraction finds much more FSMs than we can use for actual testing. Also, the generated FSMs need some manual adjustments. And finally, the test execution still requires the states to be implemented manually.

Despite that, with the contributions made by this work, the effort of MBT can be drastically reduced since FSMs can be directly extracted out of corresponding specifications. Besides testing, our FSM extraction can be used to acquire models for design or development as well.

Future work should aim at improving the accuracy, especially for intersecting edges, i.e., FSMs with a non-planar representation.

Bibliography

[BL22] Bhushan, Shreyanshu; Lee, Minho: Block Diagram-to-Text: Understanding Block Diagram Images by Generating Natural Language Descriptors. In: Findings of the Association for Computational Linguistics: AACL-IJCNLP 2022. pp. 153–168, 2022.

[Ch22] Chen, Fangwei; Zhang, Li; Lian, Xiaoli; Niu, Nan: Automatically recognizing the semantic elements from UML class diagram images. Journal of Systems and Software, 193:111431, 2022.

[Di07] Dias Neto, Arilo C; Subramanyan, Rajesh; Vieira, Marlon; Travassos, Guilherme H: A survey on model-based testing approaches: a systematic review. In: Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007. pp. 31–36, 2007.

[El04] Ellson, John; Gansner, Emden R; Koutsofios, Eleftherios; North, Stephen C; Woodhull, Gordon: Graphviz and dynagraph—static and dynamic graph drawing tools. Graph drawing software, pp. 127–148, 2004.

[Gu18] Gu, Jiuxiang; Wang, Zhenhua; Kuen, Jason; Ma, Lianyang; Shahroudy, Amir; Shuai, Bing; Liu, Ting; Wang, Xingxing; Wang, Gang; Cai, Jianfei et al.: Recent advances in convolutional neural networks. Pattern recognition, 77:354–377, 2018.

[GZS20] Gao, Wei; Zhao, Yunfei; Smidts, Carol: Component detection in piping and instrumentation diagrams of nuclear power plants based on neural networks. Progress in Nuclear Energy, 128:103491, 2020.

[HC20] Hagberg, Aric; Conway, Drew: Networkx: Network analysis with python. URL: https://networkx. github. io, 2020.

[KC13] Karasneh, Bilal; Chaudron, Michel RV: Extracting UML models from images. In: 2013 5th International Conference on Computer Science and Information Technology. IEEE, pp. 169–178, 2013.

[LY96] Lee, David; Yannakakis, Mihalis: Principles and methods of testing finite state machines-a survey. Proceedings of the IEEE, 84(8):1090–1123, 1996.

[OK22] Otto, Björn; Kleinert, Tobias: A Flow Graph based Approach for controlled Generation of AAS Digital Twin Instances for the Verification of Compliance Check Tools. In: IECON 2022–48th Annual Conference of the IEEE Industrial Electronics Society. IEEE, pp. 1–6, 2022.

[PA09] Paris, Javier; Arts, Thomas: Automatic testing of tcp/ip implementations using quickcheck. In: Proceedings of the 8th ACM SIGPLAN Workshop on Erlang. pp. 83–92, 2009.

[PNW20] Pang, Bo; Nijkamp, Erik; Wu, Ying Nian: Deep learning with tensorflow: A review. Journal of Educational and Behavioral Statistics, 45(2):227–248, 2020.

[Sa18] Sandler, Mark; Howard, Andrew; Zhu, Menglong; Zhmoginov, Andrey; Chen, Liang-Chieh: Mobilenetv2: Inverted residuals and linear bottlenecks. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 4510–4520, 2018.

[Si21] Siddique, Nahian; Paheding, Sidike; Elkin, Colin P; Devabhaktuni, Vijay: U-net and its variants for medical image segmentation: A review of theory and applications. Ieee Access, 9:82031–82057, 2021.

[TAB17] Tappler, Martin; Aichernig, Bernhard K; Bloem, Roderick: Model-based testing IoT communication via active automata learning. In: 2017 IEEE International conference on software testing, verification and validation (ICST). IEEE, pp. 276–287, 2017.

[UPL12] Utting, Mark; Pretschner, Alexander; Legeard, Bruno: A taxonomy of model-based testing approaches. Software testing, verification and reliability, 22(5):297–312, 2012.

[Yu20] Yun, Dong-Yeol; Seo, Seung-Kwon; Zahid, Umer; Lee, Chul-Jin: Deep neural network for automatic image recognition of engineering diagrams. Applied sciences, 10(11):4005, 2020.