Repairing Circuits with Transformers

Saarland University

Department of Computer Science

Master's Thesis

submitted by

Matthias Cosler

Saarbrücken, August 2022

Supervisor:	Prof. Bernd Finkbeiner, Ph.D.
Advisor:	Frederik Schmitt
	Dr. Christopher Hahn
Reviewer:	Prof. Bernd Finkbeiner, Ph.D.
	Dr. Markus Rabe
Submission:	August 22, 2022

Abstract

In this work, we present an approach for repairing faulty circuits using deep neural networks. Given a faulty circuit and a specification in LTL, we demonstrate that our approach repairs faulty circuits such that they satisfy corresponding specifications.

Since the 1950s, when introduced by Alonzo Church, reactive synthesis is a fundamental problem in computer science. Unfortunately, because the synthesis of temporal specifications is usually 2EXPTIME-complete, classic synthesis algorithms are slow and only able to synthesize from comparatively short and easy specifications. Using machine learning for such computationally hard problems gives new perspectives and recent advances show promising results. We improve these results by repairing mispredicted circuits using a Transformer-based model.

We introduce a new architecture, the separated hierarchical Transformer, that is designed to handle circuits and specifications as input sources. We introduce multiple datasets that include specifications, faulty circuits, and circuits that satisfy the specifications. We train separated hierarchical Transformer models with these datasets to repair faulty circuits towards circuits that satisfy a given specification. We show that the proposed architecture can learn from the synthetic data, that the model utilizes the repair circuit to solve more complex specifications, and that the model generalizes to out-of-distribution datasets. We show that we improve the state-of-the-art in Neural Circuit Synthesis by 6.8 percentage points on held-out instances and 11.8 percentage points on an out-of-distribution dataset from the reactive synthesis competition SYNTCOMP.

Acknowledgements

I would like to thank Prof. Finkbeiner for supporting me in working on this fascinating topic. I am grateful for the opportunity to write my master's thesis in your group. Many thanks to my advisors Frederik and Chris for many fruitful discussions, great brainstorming sessions, and an always open door for my questions. To my girlfriend, friends and parents, thanks for the support while working on my thesis and throughout my whole studies.

Erklärung
Ich erkläre hiermit, dass die vorliegende Arbeit mit der elektronischen Version übereinstimmt.

Statement
I hereby confirm the congruence of the contents of the printed data and the electronic version of the thesis.

Saarbrücken, 22 August, 2022

Contents

Chapter 1

Introduction

Already in the 1950s, Alonzo Church formulated the problem of reactive synthesis for circuits.

> "Given a requirement which a circuit is to satisfy (...). The synthesis problem is then to find recursion equivalences representing a circuit that satisfies the given requirement (or alternatively, to determine that there is no such circuit)." Alonzo Church, 1957 [Chu57]

Since then, reactive synthesis is a fundamental problem in computer science. Several formal logics have been invented to specify the requirements for systems, in formal verification and reactive synthesis. In this work, we consider the reactive synthesis of circuits for linear-time temporal logic (LTL). Using LTL, we can specify the properties of systems over time. Classical approaches to reactive synthesis from LTL specifications are typically either game-based [BL90] or bounded [Fay+17]. Although significant progress in optimization of the algorithms and tools such as BoSy [FFT17] or Strix [MSL18] have been made, reactive synthesis of LTL specifications is 2EXPTIME-complete [Ros91] and the current state-of-the-art is far from feasible for larger-scaled specifications and systems.

This motivates different perspectives and new approaches to reactive synthesis. With *Neural Circuit Synthesis from Specification Patterns* [Sch+21], Schmitt et al. recently showed that the problem can also be approached using neuro-symbolic methods by combining a Transformer-based architecture with the symbolic task of reactive synthesis. Although having promising results using Transformers for circuit synthesis, the work of Schmitt et al. [Sch+21] still leaves a gap of 20% mispredictions on held-out instances and 33% mispredictions on samples from the reactive synthesis competition SYNTCOMP [Jac+22b; Jac+22a].

The goal of this work is to introduce and implement a novel approach to automatically repair faulty circuits. This can be applied to Neural Circuit Synthesis to repair mispredicted circuits, lessen the gap, and advance in using machine learning for reactive synthesis and formal methods.

Solving the problem of circuit and program repair is an active field of research. Jobstmann et al. [JGB05; Job+12] show a game-based approach to repair programs using LTL specifications. In *Explainable Reactive Synthesis* [BFT20], the authors propose an approach for synthesizing reactive systems from LTL specifications iteratively through repair steps. *Live Synthesis* [FKM22; Nah+16] is a form of synthesis, where a system needs to be adjusted to a new specification in real-time. First, an (LTL) specification and a system that satisfies the specification are given. The specification is then updated and the system needs to be adjusted in real-time such that it satisfies the updated specification. Given the updated specification and the old system, the adjustment of the old system could also be interpreted as the repair of a malfunctioning system.

Contrary to the mentioned algorithmic approaches for circuit and program repair, in this work, we give an approach for circuit repair with LTL specifications using a deep learning architecture based on the Transformer [Vas+17]. We base this decision on two reasons. First, a deep learning model might be more suitable to repair mistakes that are made by another deep learning model. Faulty circuits produced by the network might not be logically close to a correct solution. Whereas for a deep learning model, such circuits could still be valuable because their textual representation might be close or because they might share valuable features that are not noticeable. Secondly, we base this decision on promising findings from Hahn et al. [Hah+21], showing that a Transformer can understand the semantics of temporal and propositional logics, and the work of Schmitt et al. [Sch+21] on reactive synthesis using a Transformer-based architecture.

The Transformer [Vas+17] is a sequence-to-sequence architecture for Machine Learning, originally designed for natural language processing. The Transformer architecture is one of the most revolutionizing and successful deep learning architectures of the past years. Transformer models can understand and produce complex patterns, and adhere to syntax and semantics, not only of natural language but also logic [Hah+21; Sch+21], proofs and mathematics: Transformers have been used for the learning of mathematical proofs and reasoning [Rab+20; PS20; Li+21], and applied to mathematical tasks such as integration or solving differential equations [LC19].

We introduce a new Transformer-based architecture, the **separated hierarchical Transformer** in Chapter 3. The separated hierarchical Transformer is an extension of the hierarchical Transformer [Li+21], designed to handle multiple sources of inputs. Other forms and applications of multi-source [ZK16; Nis+20; Lit+19], or more general multimodal machine learning [BAM19] are widespread, but rarely seen in logical or mathematical applications.

→ Chapter 3, p. 11

We create a **selection of datasets** for supervised learning for the circuit repair problem and train **experiments** based on the separated hierarchical Transformer architecture on these datasets. We present datasets in Chapter 4 and experiments in Chapter 5. We show

→ Chapter 4, p. 21
→ Chapter 5, p. 33

that our models repair circuits such that they satisfy the given specification with up to 86% accuracy on hold-out samples. In Chapter 6, we demonstrate that our approach can be combined with existing approaches such as Neural Circuit Synthesis [Sch+21] to repair mistakes and improve the task of reactive synthesis with machine learning. We make a significant improvement of 6.8 percentage points to a total of 84% on held-out-instances. An even greater improvement was made on out-of-distribution datasets with 11.8 percentage points on samples from the reactive synthesis competition SYNTCOMP.

➔Chapter 6, p. 43

Chapter 2

Reactive Synthesis

This chapter lays the theoretical background on reactive synthesis, linear-time temporal logic (LTL), and And-Inverter Graphs. We further present the work on Neural Circuit Synthesis [Sch+21], which partially is the foundation for the work in this thesis.

Reactive synthesis is the task to find a system that satisfies a given formal specification. We consider formal specifications that are formulas over a set of atomic propositions (AP) in LTL. The specification defines the desired behavior of a system based on a set of input and output variables. As the system, we consider circuits, more precisely a text representation of And-Inverter Graphs, called AIGER circuits. And-Inverter Graphs connect input and output edges using AND gates, NOT gates (inverter), and memory cells (latches).

We say that a circuit satisfies a specification if and only if, given any possible traces of assignments of input variables and the traces of assignments of output variables determined by the circuit, the specification holds. A specification is either *realizable* if a circuit exists that satisfies the specification or *unrealizable* if no circuit exists that can satisfy the specification.

Definition 2.1

For a specification s, we call a circuit c correct if either

- the specification s is realizable, and the circuit c satisfies the specification. Or
- the specification s is unrealizable, and the predicted circuit c is a counter strategy to the specification s.

2.1. Linear-time Temporal Logic (LTL)

LTL [Pnu77] combines boolean operators with temporal operators as *next* ($\bigcirc$) and *until* ($\mathcal{U}$). LTL is evaluated over trace models of atomic propositions. Boolean operators are evaluated over the propositions of states, whereas temporal operators refer to the states on which the propositions are evaluated. For example, *next* refers to a property on the state that comes relatively next to the current state, and *until* refers to a property that needs to hold until a second property will eventually hold.

$$\varphi := p \mid \varphi \wedge \varphi \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi$$

where p is an atomic proposition $p \in AP$. In this context, we assume that the set of atomic propositions AP can be partitioned into inputs I and outputs O: $AP = I\dot{\cup}O$.

Generally LTL is defined over a set of traces: $TR := (2^{AP})^{\omega}$. Let $\pi \in TR$ be trace, $\pi_{[0]}$ the starting element of a trace π and for a $k \in \mathbb{N}$ and be $\pi_{[k]}$ be the kth element of the trace π. With $\pi_{[k,\infty]}$ we denote the infinite suffix of π starting at k. We write $\pi \models \varphi$ for *the trace π satisfies the formula φ.*

For a trace $\pi \in TR$, $p \in AP$ and formulas φ:

- $\pi \models \neg\varphi$ iff $\pi \not\models \varphi$
- $\pi \models p$ iff $p \in \pi_{[0]}$; $\pi \models \neg p$ iff $p \notin \pi_{[0]}$.
- $\pi \models \varphi_1 \wedge \varphi_2$ iff $\pi \models \varphi_1$ and $\pi \models \varphi_2$.
- $\pi \models \bigcirc\varphi$ iff $\pi_{[1]} \models \varphi$
- $\pi \models \varphi_1\,\mathcal{U}\,\varphi_2$ iff $\exists l \in \mathbb{N} : (\pi_{[l,\infty]} \models \varphi_2 \wedge \forall m \in [0, l-1] : \pi_{[m,\infty]} \models \varphi_1)$

We use further temporal and boolean operators that can be derived from the ones defined above. That includes $\vee, \rightarrow, \leftrightarrow$ as boolean operators and the following temporal operators:

- $\varphi_1\,\mathcal{R}\,\varphi_2$ *(release)* is defined as $\neg(\neg\varphi_1\,\mathcal{U}\,\neg\varphi_2)$
- $\square\varphi$ *(globally)* is defined as *false* $\mathcal{R}\,\varphi$
- $\diamondsuit\varphi$ *(eventually)* is defined as *true* $\mathcal{U}\,\varphi$

2.2. And-Inverter Graphs

And-Inverter Graphs are graphs that describe hardware circuits. The graph connects input edges with output edges through AND gates, latches, and implicit NOT gates. We usually represent this graph by a text version called the AIGER Format [Bru+07]. The

AIGER format uses variable numbers that define variables. Variables can be interpreted as wired connections in a circuit or as edges in a graph, where gates and latches are nodes.

- A negation is implicitly encoded by distinguishing between even and odd variable numbers. Two successive variable numbers represent the same variable, the even variable number represents the non-negated variable, and the odd variable number represents the negated variable. The variable numbers `0` and `1` have the constant values `FALSE` and `TRUE`.
- Each input and output edge is defined by a single variable number, respectively.
- An AND gate is defined by three variable numbers. The first variable number defines the outbound edge of the AND gate, and the following two variable numbers are inbound edges. The value of the outbound variable is determined by the conjunction of the values of both inbound variables.
- A latch is defined by two variable numbers: an outbound edge and an inbound edge. The value of the outbound variable is the value of the inbound variable at the previous time step. In the first time step, the outbound variable is initialized as `FALSE`.

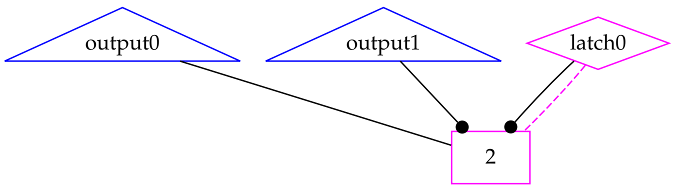
The AIGER format starts with a header, beginning with the letters `aag` and following five non-negative integers `M, I, L, O, A` with the following meaning:

```
M=maximum variable index
I=number of inputs
L=number of latches
O=number of outputs
A=number of AND gates
```

After the header, each line represents a definition of either input, latch, output, or AND gate in this order. The numbers in the header define the number of lines associated with each type. After the definition of the circuit, an optional symbol table might follow, where we can define names for inputs, outputs, latches, and AND gates. In this context, the circuit can either describe a satisfying system or a counter strategy to the specification.

Example 2.2.1. This example from [Bru+07] describes a toggle flip flop without inputs. The comments at the end of each line are not part of the actual format.

```
aag 1 0 1 2 0
2 3           latch 0
2             output 0
3             output 1
o0 output0
o1 output1
l0 latch0
```



The maximum variable index in the example is `1`, because variable numbers `2` and `3` represent the same variable, where `3` is the negation of `2`. We have no inputs, one latch, and two outputs. The last three lines are optional and define a symbol table. The latch has the inbound connection `3` and outbound connection `2`. A latch is visualized by a diamond shape with a dotted outbound connection. In the first time step `2` has the value `FALSE`, hence `3` has the value `TRUE`. Therefore, the outputs (triangle in the visualization) in the first time step are `output0: FALSE, output1: TRUE`. In the second time step, the latch outputs the value of `3` in the previous time step (`TRUE`) hence `2` has the value `TRUE` and `3` the value `FALSE`. The outputs in the second time step are `output0: TRUE, output1: FALSE`. If AND gates would be present, they would be shown as oval shapes in the visualization, with the inputs on the bottom and output(s) on the top. △

2.3. Circuit Repair

In this work, we target the problem of circuit repair. Circuit repair is closely related to reactive synthesis. Formally it can be seen as a subproblem of circuit synthesis. Given a specification and a (faulty) circuit, it is to find a circuit that satisfies the specification. Formally speaking, this problem is equally hard as circuit synthesis as we give no formal constraints on the (faulty) circuit. However, statistically speaking, the (faulty) circuit can help to find a satisfying circuit. Therefore, given a helpful (faulty) circuit, we anticipate circuit repair to work better in practice than circuit synthesis. The concept of circuit repair can be applied to improve any form of potentially unsound approach for circuit synthesis, which includes circuits designed by humans.

2.4. Neural Circuit Synthesis

The work of this thesis is strongly influenced by the work of Schmitt et al. [Sch+21] about Neural Circuit Synthesis. We will give a quick overview of this work, but refer to the original paper for more information. Neural Circuit Synthesis is one of the first applications of machine learning to reactive synthesis. The authors provide large datasets of pairs of LTL specifications and AIGER circuits for supervised learning. The specifications are generated based on patterns extracted from the synthesis competition SYNTCOMP [Jac+22b; Jac+22a], the corresponding circuits are generated using the synthesis tool Strix [MSL18]. The authors trained a hierarchical Transformer to predict AIGER circuits that satisfy the specification. In the case of an unrealizable specification, the Transformer should predict an AIGER circuit that inherently proves the unrealizability. The authors showed that their models achieved competitive results on held-out instances (79.9% accuracy), and two out-of-distribution benchmarks: the SYNTCOMP benchmarks (66.8% accuracy) and J.A.R.V.I.S [Gei+22] (40.0% accuracy). On a set of samples on which the synthesis tool Strix timed out after 120 seconds, the authors reached 30.1% accuracy.

Chapter 3

Setup and Architecture

In this chapter, we present the structure and architecture of this approach. We first show how the input and target are structured. Then we lay the background on the Transformer and hierarchical Transformer architecture. We also introduce a new architecture, the separated hierarchical Transformer. Further, we discuss how tokenization and positional encoding are applied. Lastly, we explain how we evaluate the network and why we use model checking in the evaluation process. We give an overview of the process in Fig. 3.1.

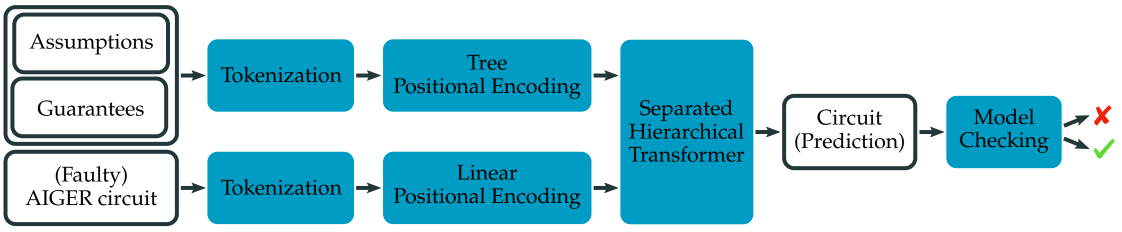


Figure 3.1.: Overview over the architecture pipeline

3.1. Data Format

We briefly cover the format of input and target of the architecture. We present more details on the format, datasets, and their generation in Chapter 4. →Chapter 4, p. 21

We consider an LTL specification and a (faulty) AIGER circuit as input sequences. The target feature is a correct AIGER circuit.

Specification We structure the specification into multiple input sequences: assumptions and guarantees. Assumptions and guarantees are lists of LTL formulas (see Sec. 2.1). The lists of assumptions and guarantees combined make a specification: →Sec. 2.1, p. 6

$$spec := (assumption_1 \land \cdots \land assumption_n) \rightarrow (guarantee_1 \land \cdots \land guarantee_m)$$

(Faulty) Circuit The second input feature is a sequence close to the format of an AIGER circuit (see Sec. 2.2). We enforce only a few constraints on the format: the sequence needs to be processable by our implementation of the parser and tokenizer (Sec. 3.5). This constraint does not necessarily imply that the sequence follows the correct AIGER syntax, nor does it imply that the circuit cannot be correct.

→ Sec. 2.2, p. 6
→ Sec. 3.5, p. 16

Target Circuit. The target circuit is a sequence representing a correct (Def. 2.1) AIGER circuit under consideration of the given specification. The circuit includes a symbol table, which, in this context, implicitly encodes whether the circuit is a satisfying system or a counter strategy.

→ Def. 2.1, p. 5

3.2. Transformer

The Transformer [Vas+17] is a sequence-to-sequence architecture for machine learning, originally designed for natural language processing. The Transformer architecture is one of the most revolutionizing and successful deep learning architectures of the past years. The foundation of the Transformer is an attention mechanism, allowing for a deeper and more flexible understanding of the input and target sequences. We will give a brief overview of the Transformer architecture and the attention mechanism in this section and Fig. 3.2.

Encoder Stack The encoder stack is designed to understand the input and transform it into a hidden representation. The encoder stack consists of multiple encoders, each of the same structure. We combine a multi-head self-attention mechanism (encoder attention) with a feed-forward network. Residual connections allow to skip the attention and or the feed-forward network. We normalize the results from the attention and the feed-forward network when combined with the residual connection.

Decoder Stack The decoder stack generates the output word-by-word based on the hidden representation and previously predicted words. The prediction of each word is called a prediction step. The decoder stack consists of multiple serially connected decoders; each decoder consists of multi-head self-attention on the output/target (decoder attention) combined with multi-head attention between encoder and decoder (encoder-decoder attention). A feed-forward network processes the resulting vector before softmax predicts the next word. Residual connections allow to skip the attention blocks and or the feed-forward network. As in the encoder, we normalize the results from the attention and the feed-forward network when combined with the residual connection.

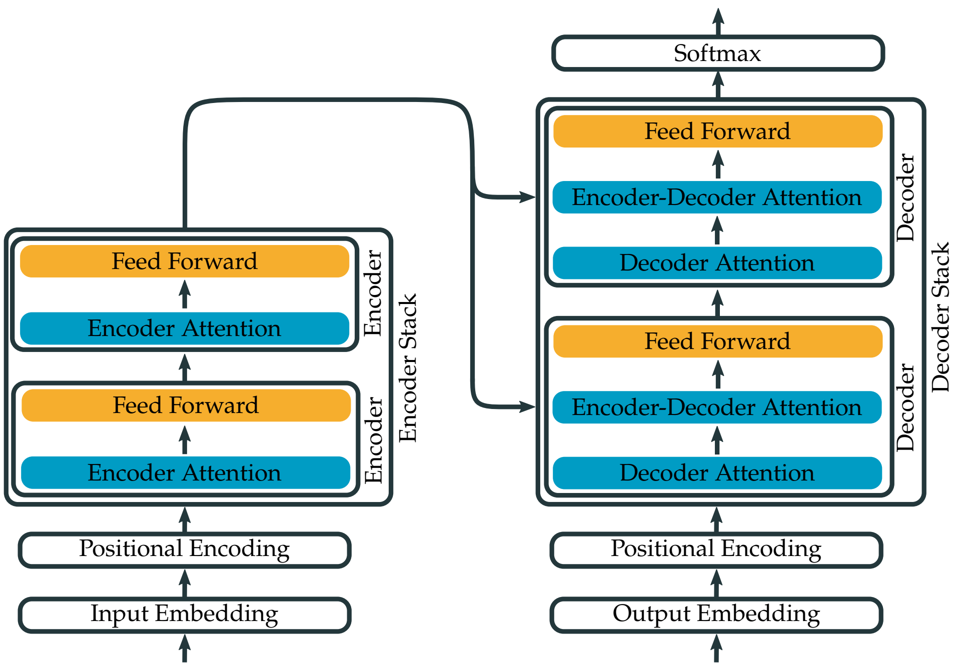


Figure 3.2.: The Transformer architecture. We show two encoders in an encoder stack and two decoders in a decoder stack.

Attention Attention is a method of learning relationships and connections between different words/positions of a sentence. The Transformer uses the attention mechanism in multiple steps. Encoder- and decoder attention are **self-attention** layers, meaning the attention is learned between positions of the same sentence, where encoder-decoder attention finds relationships between the hidden representation (output of the encoder) and the output of the decoder attention.

The underlying principle is the **scaled dot-product**. A query, key, and value vector are learned for each embedded token. Then, a score is calculated, stating how strong the connection of each token to any other token is, by calculating the dot product between the learned query vector of the current token and the learned key vector of any other token. This score is scaled before softmax is applied and multiplied with the learned value vector of each token. Let Q, K, V be the matrixes of all query, key, and value vectors packed together. d_k is the dimension of the key vectors. The attention is calculated as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

For the **encoder-decoder attention**, query vectors from the hidden representation of the decoder attention are used, while key and value vectors come from the output of the encoder attention stack. During training, a masked version of the self-attention in

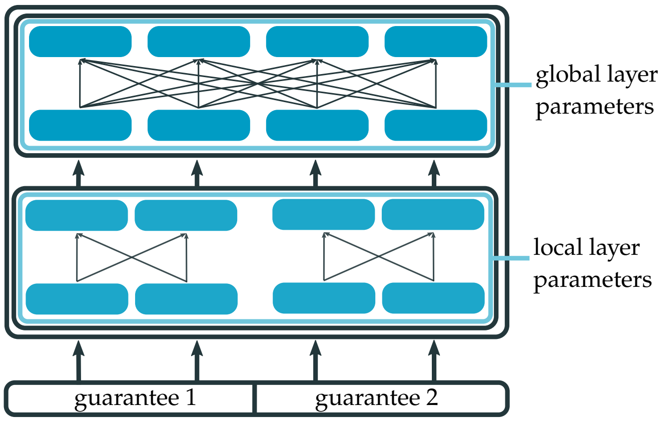


Figure 3.3.: The structure of global and local layers in the hierarchical Transformer. We omit the feed-forward network in this figure. Arrows between blue boxes (tokens) express the attention mechanism between tokens. As illustrated here, the specification can be partitioned into its guarantees.

the decoder is used. Only positions that already have a prediction should be visible to the self-attention. This method is called **masked self-attention** and prohibits cheating.

Instead of computing the scaled dot product just once, multiple versions, each called an attention head, are learned in parallel. The results are combined using a linear transformation. This is called **multi-head attention**.

3.3. Hierarchical Transformer

The hierarchical Transformer is a variation of the original Transformer. It was first proposed in Li et al. [Li+21]. A hierarchical Transformer has two types of hierarchically structured layers in the encoder. Local layers only see parts of the input, while the global layers handle the combined output of all local layers. First, the input is decomposed into multiple partitions before being fed into the local layers. Positional Encoding is applied separately to each part of the input, which is implemented by having one traditional encoder as local layers. All input partitions are fed independently and successively into the encoder. Therefore, model parameters are shared between the local layers. However, no attention can be calculated between tokens in different partitions because only one part of the input is visible to the local layer at once. The outputs of the local layer are concatenated and fed into the global layer. The global layer is just a standard encoder. An illustration of the architecture can be found in Fig. 3.3.

The hierarchical Transformer has been beneficial to understanding repetitive structures in mathematical [Li+21] and logical contexts such as LTL formula [Sch+21]. Local layers can learn these repetitive structures, while the global layer learns more general connections in the whole input sentence.

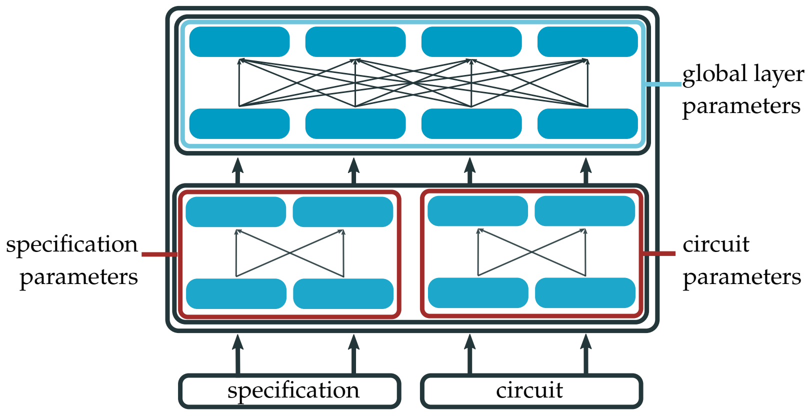


Figure 3.4.: The structure of global and local layers in the separated hierarchical Transformer. As applied in this work, the two local layers process circuits and specifications independently. Arrows between blue boxes (tokens) express the attention mechanism. We simplify this figure by not showing the feed-forward network.

3.4. Separated Hierarchical Transformer

We introduce a new architecture by extending the hierarchical Transformer to a separated hierarchical Transformer. A separated hierarchical Transformer has two types of local layers: Each **separated local layer** is an independent encoder, therefore, separated local layers do not share any model parameters, nor are attention calculations between tokens in two separated local layers possible, as shown in Fig. 3.4. **Non-separated local layers** are identic to local layers in the hierarchical Transformer. Non-separated local layers share model parameters, but no attention can be calculated between tokens in two local layers. A separated local layer contains one or more non-separated local layers. The results of the separated local layers are concatenated and fed into the global layer. While the number of non-separated local layers does not change the model size, multiple separated local layers increase the model size since they introduce more model parameters. Contrary to the hierarchical Transformer, which is typically good at handling multiple partitions of the input that are independent but of the same structure, type, and maximal length, we introduced the separated hierarchical Transformer to additionally handle multiple completely independent inputs, including structure, types, and length.

We give a comparison of the presented variations in Tbl. 3.5. A Transformer automatically performs the attention mechanism over the whole input sequence. Consequently, model parameters are shared over the whole input as well. The hierarchical Transformer can calculate independent attention mechanisms over partitions of the input while it uses the same model parameters for the whole input. The separated hierarchical Transformer can use independent attention mechanisms. It can share model →Tbl. 3.5, p. 16

	Independent Model Parameters	Independent Attention Calculations
Transformer	✗	✗
Hierarchical Transformer	✗	✓
Separated Hierarchical Transformer	✓	✓

Table 3.5.: Overview of the characteristics of the different architecture variations.

parameters between some partitions of the input and separate between model parameters of other partitions of the input. Independent parameters also imply independent attention mechanisms.

In this work, we use a separated hierarchical Transformer. We separate the specification and the (faulty) circuit using a separated local layer. The specification is further partitioned into its guarantees and assumptions, which we feed into non-separated local layers. Therefore, in local layers, attention calculation is limited to between tokens in the circuit, between tokens in each guarantee, and between tokens in each assumption, but not between the different partitions. Parameters are shared between all assumptions and guarantees but not between the specification and the circuit.

3.5. Tokenization and Embedding

While language models such as the Transformer need to embed arbitrary words of a natural language, we specialize in the Domain Specific Language (DSL) of LTL formulas (Sec. 2.1) and AIGER circuits (Sec. 2.2) with only a few symbols. For every symbol in the DSL, we introduce a token. In the LTL formula, we fix atomic propositions to five inputs $i_0 \cdots i_4$ and five outputs $o_0 \cdots o_4$ and introduce a token for each. Parentheses are redundant because we encode the syntax tree into the positional encoding (see Sec. 3.6). In the AIGER format, we fix the variable numbers to the range of 0 to 61, thereby indirectly limiting the size of the circuit. These limitations seem to be a good trade-off between token set size and expressible complexity. We only embed the circuit itself, not the symbol table, which can be part of the AIGER format. Therefore, we set a special token as a prefix to the circuit embedding. This token determines whether the circuit represents a satisfying circuit or a counter strategy, hence whether the specification is presumed to be realizable or not. We embed the tokens by applying one-hot encoding which we multiply with a learned embedding matrix. We call the embedding that results from one token, an embedding vector.

→Sec. 2.1, p. 6
→Sec. 2.2, p. 6
→Sec. 3.6, p. 17

3.6. Positional Encoding

A positional encoding of the input and target ensures that the Transformer can determine the order of embedding vectors and, therefore, the order of the words in the input/target sequences. Different techniques of positional encoding are possible, but they should meet some principles:

- Every position/token in the input should have a unique encoding.
- An easy-to-learn transformation can describe relative positions. This includes affine and linear transformations.

Linear Positional Encoding. As linear positional encoding for encoding sequences in general, Vaswani et al.[Vas+17] propose overlaying sine and cosine functions with different frequencies. This positional encoding has the advantage of easily generalizing to sequences longer than seen during training and having a fixed size for each token in potentially arbitrary long sequences. Further, relative positions are described by a linear transformation. In this work, we use a linear encoding for circuits (target circuit and (faulty) circuit)

Tree Positional Encoding. To consider tree-like structures, Shiv et al. [SQ19] introduced a tree positional encoding. This encoding incorporates the tree structure into the positional encoding and allows easy calculations between tree relations as siblings, parents, and children. For this encoding, we fix a maximal branching size and the maximal depth of the tree. The position of a token is defined by the path from the node to the tree's root node. Instead of a number, a one-hot encoding is used to describe the children-parent relation. Each node has a unique encoding, and an affine transformation describes relative positions. This work uses a tree encoding for LTL formulas representing assumptions and guarantees (input). The tree structure represents the syntax of the LTL formulas. We give an example of the tree encoding in Fig. 3.6.

→ Fig. 3.6, p. 18

3.7. Evaluation

We evaluate the model by querying the specification together with a faulty circuit. The model predicts a circuit and a realizability token. We call a prediction correct if the predicted circuit is correct (Def. 2.1) under consideration of the predicted realizability token and the given specification. We use beam search to improve the sequence accuracy while predicting the output token-by-token. Lastly, we model-check our predictions to verify our results.

→ Def. 2.1, p. 5

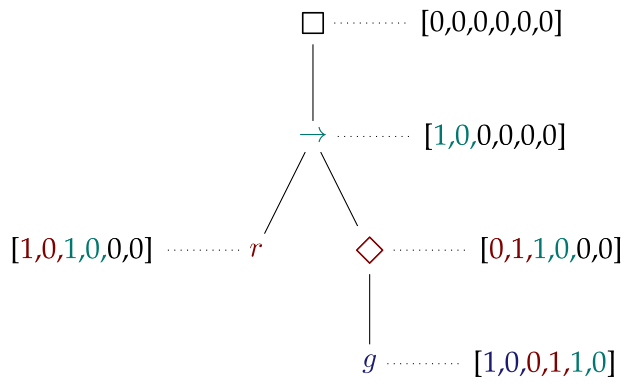


Figure 3.6.: Example tree positional encoding for the LTL request pattern $\square(r \rightarrow \diamond g)$. This example is taken from [Sch+21].

Beam Search. Beam search is a technique often combined with Transformers to improve the sequence accuracy of a Transformer model [Wu+16]. Beam search allows considering multiple distinct predictions simultaneously. Each such prediction is called a beam. A predefined parameter called beam size (bs) determines the number of predictions kept simultaneously. In each prediction step, we consider the bs likeliest predictions. Therefore, in the next prediction step, bs distinct histories are fed into the decoder. For each history, we again predict the bs likeliest words, leading to bs^2 distinct beams, of which we select the bs most promising ones by averaging the likelihood of all words in the beam. This process allows the model to rectify predictions of earlier steps that seemed optimal when the token was predicted.

Model Checking. Model checking is the counterpart to synthesis. Given a specification and an AIGER circuit, a model-checking algorithm decides whether the circuit is correct (Def. 2.1). We model-check each prediction to verify our results. Model checking is a PSPACE-complete operation. Therefore, it is much easier than solving reactive synthesis algorithmically (2EXPTIME-complete). Because of the hardness of reactive synthesis, we deal with comparatively small specifications and circuits. Model-checking the predictions is feasible for the size of our specifications and circuits. Combining our architecture with model checking makes our approach sound because we only accept model-checked circuits. We model-check using the tool nuXmv [Cav+14].

→ Def. 2.1, p. 5

Performance Measures. When measuring the accuracy of our models, multiple metrics can be considered: When considering only predictions that are identical to the target, we measure what we call the **syntactic accuracy** of the model. No model checking is necessary for measuring syntactic accuracy. We call predictions that fall under the syntactic accuracy a matching circuit or simply a match. However, in this case, it is more interesting to give a **semantic accuracy** by measuring circuits verified as correct by the model checker. This accuracy is typically higher than the syntactic accuracy, as it also considers alternative solutions. Contrary to syntactic accuracy, the semantic accuracy can also be calculated over datasets without target circuits.

Chapter 4

Datasets

In this chapter, we introduce a method for generating datasets for Neural Circuit Repair. All datasets we create are for supervised learning. All datasets have multiple splits, at least a *training*, *validation* and *test* splits. The number of samples in each dataset differs from dataset to dataset, depending on the generation technique. Most of the datasets have approximately 250000 samples, with typically around 10% in the *validation* and *test* set respectively and 80% in the *training* split. In Appendix A.1 we list exact numbers per dataset. All datasets we create are based on the datasets from [Sch+21].

➔ Appendix A.1, p. 63

We first explain the format the datasets have. Secondly, we discuss two methods for generating the datasets.

4.1. Format

A CSV file describes each split with several columns describing input, targets, and annotations. Annotations are additional characteristics not used during training but give valuable insight. A dataset is defined by a folder containing the CSV files of all dataset splits, along with some metadata and statistics.

In the following, we list input, target, and annotation columns:

Specification The specification is input to the network and consists of two columns: assumptions and guarantees. Both are a list of arbitrary LTL formulas and are generated based on SYNTCOMP 2020 benchmark [Jac+22b; Jac+22a]. The exact algorithm is described in [Sch+21].

(Faulty) Circuit. A possibly faulty circuit is an additional input to the network and the distinguishing feature between Neural Circuit Repair and Neural Circuit Synthesis. The circuit is supposed to be an AIGER circuit (Sec. 2.2), but we do not force the circuit to

➔ Sec. 2.2, p. 6

be syntactically correct. The circuit might satisfy or violate the specification. It is often close to a satisfying AIGER circuit in a loose sense. The circuit is to be repaired such that it eventually is a correct circuit (Def. 2.1), which is why it is called **repair circuit**. The following sections detail how to generate the repair circuit and what close in this context means.

→ Def. 2.1, p. 5

Target Circuit. The target circuit is a correct AIGER circuit; hence, it either satisfies the specification or is a counter strategy to the specification. This depends on whether the specification is realizable. It is generated from the specifications using the synthesis tool Strix [MSL18] or by evaluating Neural Circuit Synthesis while all samples are verified with nuXmv [Cav+14].

Annotations. Some columns in the CSV files are just annotations and not used for training. They should help to understand the underlying data and might be helpful to gain further insight.

- A status field describes how each sample was generated. In more detail, it describes the relationship between the repair circuit and target circuit, respectively specification. It can be *Violated* (repair circuit violates specification), *Satisfied* (repair circuit is correct, see Def. 2.1), *Match* (repair circuit is identical to target circuit and therefore also is correct), or *Changed* (repair circuit is a synthetically altered version of the target circuit).
- Input and output variables used in the specification.
- We include a hash that uniquely describes the specification.
- A realizability flag shows whether the specification is realizable. As the symbol table of the circuit indirectly encodes this information, we treat this column as an annotation.
- Lastly, we include an optional column containing the Levenshtein distance (Def. 4.1) between the repair circuit and the target circuit.

→ Def. 2.1, p. 5

→ Def. 4.1, p. 22

Definition 4.1 (Levenshtein Distance)

The Levenshtein distance is an edit distance metric, measuring the degree of distinction between two strings. Let s_1 and s_2 two given strings, then the Levenshtein distance $lev(s_1, s_2)$ is a the number of actions necessary to transform string s_1 into string s_2 or vice versa. Possible actions are deletions, insertions, and substitutions.

We use two approaches and their combination to generate datasets for neural circuit repair. Both approaches rely on existing datasets from the Neural Circuit Synthesis project. The following two sections explain both methods in detail. We also provide a Jupyter Notebook in the repository [1], guiding through the generation of the datasets with code.

4.2. Data generation through altering target circuits

The first method creates repair circuits using an altered target circuit version. There are two basic directions for introducing mistakes to the circuits. We could either introduce logic mistakes (similar to human mistakes by faulty reasoning) or introduce mistakes in the representation (similar to careless mistakes). We choose the latter direction for this work. The following paragraphs go into more detail about defining mistakes in the representation and what to consider when introducing such mistakes into a circuit.

- We introduce mistakes in the representation of the circuit. This means that introduced mistakes can be approximated using the Levenshtein distance. More mistakes lead to a higher Levenshtein distance. Also, comparing a more significant mistake to a less significant one should not lead to a smaller Levenshtein distance. This entails that a more significant mistake (following this definition) is not necessarily more significant in a logical sense. A circuit containing mistakes might even satisfy the specification. Indeed, this happened for an average of 1.7%. However, it strongly depends on the choice of parameters we use to create a dataset. All the created datasets have less than two percent of such samples. For per dataset statistics, have a look at Appendix A.1. → Appendix A.1, p. 63
- The mistakes we introduce should not be completely arbitrary, such as adding noise, but follow the syntactic of AIGER circuits and be comprehensible to a certain degree.

The following algorithm achieves these standards:

1. Given the parameters $changes_{max}$, $changes_{min}$ and $changes_{range-68}$, determine how many changes should be performed on the circuit by sampling a discrete truncated normal distribution, with mean 0, and $\sigma = \frac{changes_{range-68}}{2}$. We truncate with a lower bound of $changes_{min}$ and an upper bound of $changes_{max}$. See Fig. 4.1 for illustration. → Fig. 4.1, p. 24
2. For each change: given the parameter $P_{deteletes}$, we either delete a line with the probability $P_{deteletes}$ or change the number of a variable with the probability $1 - P_{deteletes}$

[1] https://github.com/MatCos/ml2/blob/main/notebooks/repair_datasets_creation.ipynb

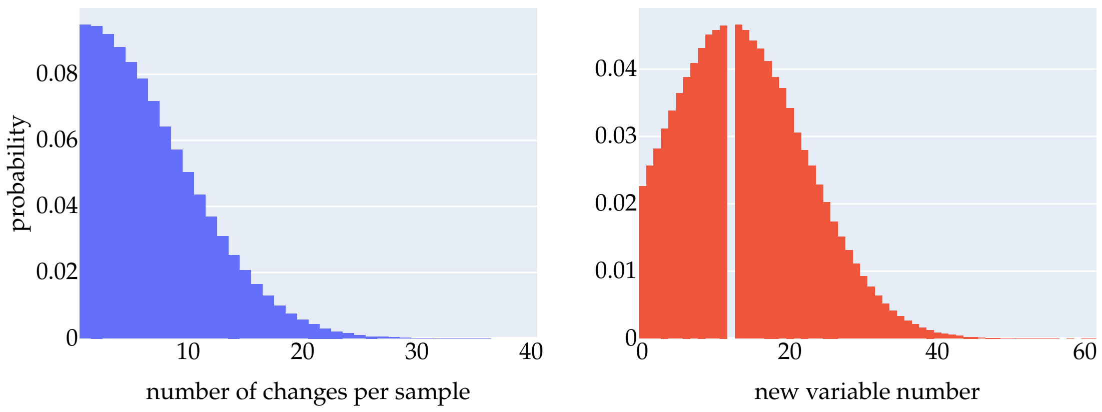


Figure 4.1.: On the left side, we depicted the probability density function for sampling the number of changes applied to a sample. Default values of $changes_{max} = 50, changes_{min} = 1, changes_{range-68} = 15$. On the right side, we show the probability density function for sampling a new variable number based on the exemplary old variable number 12. Default values of $var_{max} = 61, var_{min} = 0, var_{range-68} = 20$.

- For deleting: uniformly choose a line from the AIGER circuit. We do not remove inputs or outputs to stay consistent with the dataset. All circuits and specifications in the dataset have the same number of inputs and outputs.
- For changing: uniformly choose a variable number to replace with a new value. The variable number can be an input, output, the inbound edge(s), or the outbound edge of a latch or AND gate. The new variable number is determined by sampling a discrete truncated normal distribution by choosing the mean as the old variable number and the parameter $var_{range-68}$ determining the width $\sigma = \frac{var_{range-68}}{2}$ of the distribution. The new variable number is not smaller than the parameter var_{min}, not larger than var_{max}, and cannot be the mean itself. For illustrations see Fig. 4.1 → Fig. 4.1, p. 24

3. The altered circuits are model-checked using nuXmv. For correct circuits (Def. 2.1) the status annotation is set to *Satisfied* and to *Changed* for all others. We calculate statistics to describe the dataset accurately. → Def. 2.1, p. 5

We generate a collection of different datasets covering a range of parameters. Datasets created based on this section are named with the prefix *exp-repair-alter*, with a number following, that simply iterates over all datasets from this section. We show the Levenshtein distance between altered circuit and target circuits in Fig. 4.2. An overview of more statistics and other final datasets can be found in Appendix A.1. → Appendix A.1, p. 63

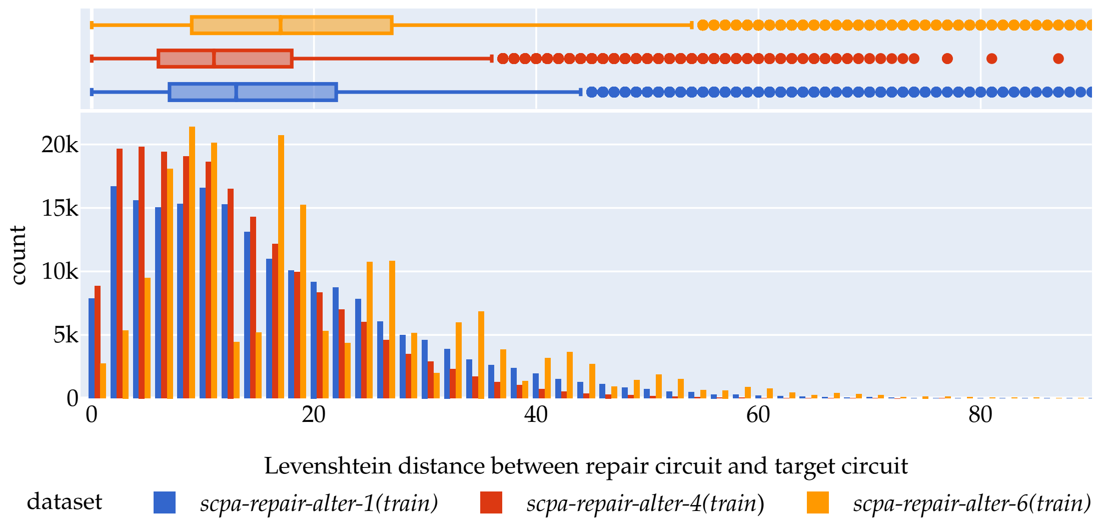


Figure 4.2.: Levenshtein distance between repair circuit and target circuit. *scpa-repair-alter-1*: $P_{deletes} = 0.2, changes_{max} = 50, changes_{range-68} = 15, var_{range-68} = 20$. *scpa-repair-alter-4*: $P_{deletes} = 0.1, changes_{max} = 50, changes_{range-68} = 15, var_{range-68} = 20$. *scpa-repair-alter-6*: $P_{deletes} = 1, changes_{max} = 20, changes_{range-68} = 5$.

4.3. Data generation through Neural Circuit Synthesis

Having the goal in mind of improving the state-of-the-art in Neural Circuit Synthesis, the second approach is based on the evaluation of Neural Circuit Synthesis. This approach allows us to train a network on actual evaluation data and, later, repair the results of the Neural Circuit Synthesis without out-of-distribution samples. However, we need to make several significant modifications before we can use the evaluation data for training.

In a first step, we train a network for Neural Circuit Synthesis. Architecture and parameter-tuning are based on [Sch+21]. Neural Circuit Synthesis uses a hierarchical Transformer (Sec. 3.4), and for evaluation beam search (Sec. 3.7) is applied. The next step is to evaluate the network with all original Neural Circuit Synthesis dataset samples. The original dataset has the same format described in Sec. 4.1 except for the repair circuit and related fields. The predictions are collected and model-checked against the specification and combined with the original dataset. This procedure creates a raw dataset consisting of the fields from the old dataset plus the prediction, now called the repair circuit, a status annotation with the result of the model checker, and an annotation column with the Levenshtein distance between the repair circuit and the target circuit. We repeated the evaluation for different beam sizes, giving us multiple datasets. Further, we have multiple unique predictions per specification for evaluations

→Sec. 3.4, p. 15
→Sec. 3.7, p. 18
→Sec. 4.1, p. 21

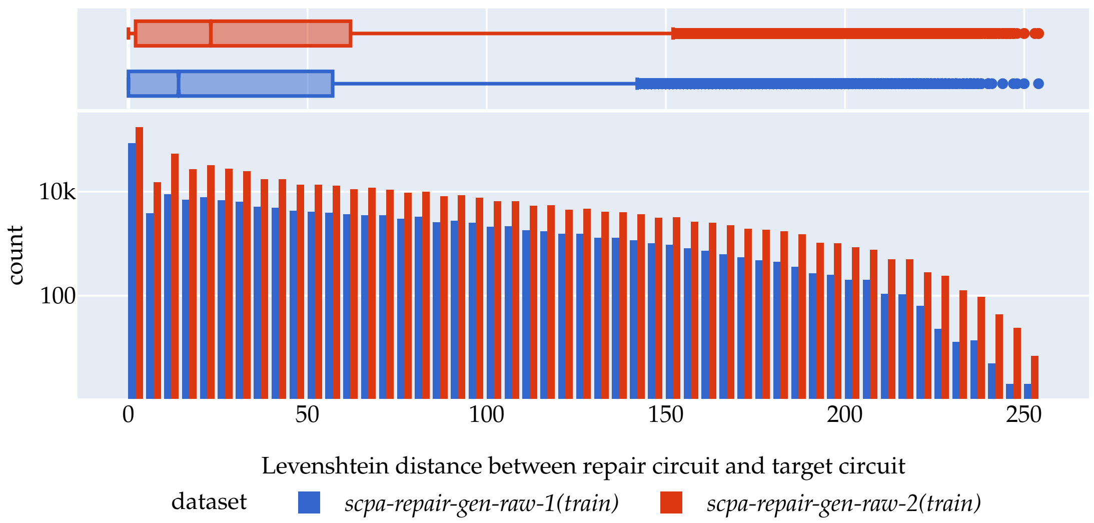


Figure 4.3.: Levenshtein distance between repair circuit and target circuit of the raw datasets based on beam size 1 (*...raw-1*) and beam size 3 (*...raw-2*). Notice the logarithmic scale of the y-axis.

where a beam size greater than one was used. Each prediction is combined with the original fields such as specification and target, leading to multiple unique samples in the new dataset that are based on a single evaluated sample.

In a first attempt, we trained experiments on these raw datasets. However, we only obtained bad results and unstable training. Therefore, we analyze the raw dataset more profoundly and determine several problems. First, we must choose a metric to calculate the difference between the repair circuits (evaluation output) and the target circuits. The repair circuits are based on the prediction of Neural Circuit Synthesis, which architecture is a sequence-to-sequence architecture, more precisely, a Transformer. Because of the architecture, we expect most mistakes in the representation of the circuit, hence several mispredicted tokens instead of logical mistakes (compare Sec. 4.2). Therefore, we propose the Levenshtein distance (Def. 4.1) between the repair circuit and target circuit as a metric.

→Sec. 4.2, p. 23
→Def. 4.1, p. 22

To get an overview, we show the histogram for the Levenshtein distance between the repair circuit and the target circuit in Fig. 4.3. This plot substantiates the choice of metric, as we can see that most mistakes in the repair circuit are small in the representation. Only comparatively few samples have a high Levenshtein distance.

4.3.1. Misleading Targets

One problem we could determine with the raw dataset is the problem of misleading targets. Neural Circuit Synthesis has to differ between semantic and syntactic accuracy. Correctly predicted circuits can either be identical to the target circuit (syntactic accuracy) or satisfy the specification with an alternative solution (semantic accuracy). Therefore, the repair circuit can satisfy the specification without being identical to the current target circuit. For beam size 1, we achieve a semantic accuracy of 55% vs. a syntactic accuracy of 32% on the Neural Circuit Synthesis task, leaving a gap of 23% samples where the predicted circuit is different from the current target circuit, although semantically correct. If we build a dataset for training from this evaluation data, the network would learn to repair already (semantically) correct samples. Not only is this unnecessary, as we only need semantically correct circuits, but the massive difference between the repair circuit and the target circuit might even hinder the network from gaining helpful information from the repair circuit. We call the target of such samples a misleading target as it leads the training signal in the wrong direction of repairing a correct circuit.

The misleading targets are not only found in *semantically correct but not matching* samples but can also be found in violating samples. Given a sample where the repair circuit does not satisfy the specification but is very close to some satisfying circuit (alternative target), we call the current target a misleading target. The training signal would direct towards learning the distant target instead of the close alternative target.

We apply two algorithms to the dataset to remove as many misleading targets as possible.

Removing misleading targets from semantically correct samples. Removing such misleading targets is comparatively easy. Semantically correct repair circuits are by definition already an alternative target so that we can replace the target circuit with the repair circuit. This eliminates the misleading training signal for semantically correct samples.

Removing misleading targets from violating samples. Given a violating sample, we need to determine whether the target might be misleading and if we find a better alternative target. In general, we cannot decide whether a target is misleading as we need to know all the alternative targets for this specification. Therefore, we choose to generate only a few alternative targets for each specification using the beam search. We evaluate each sample with different beam sizes from 1 to 4. This gives us multiple predictions for the same sample. Not surprisingly, not all of the predictions satisfy the specifications. In practice, for larger beam sizes, we often have groups of predictions that are close to each other, where one prediction is a satisfying circuit, and approximately 3 other predictions violate the specification. The satisfying circuit, however, might not be identical to the current target circuit, hence is an alternative target. Therefore using the beam search is an efficient way to find alternative targets that are close to the repair circuit. Algorithmically, for each violating sample, we now compare the current target and all alternative targets we discovered during beam-search. Based on the assumption

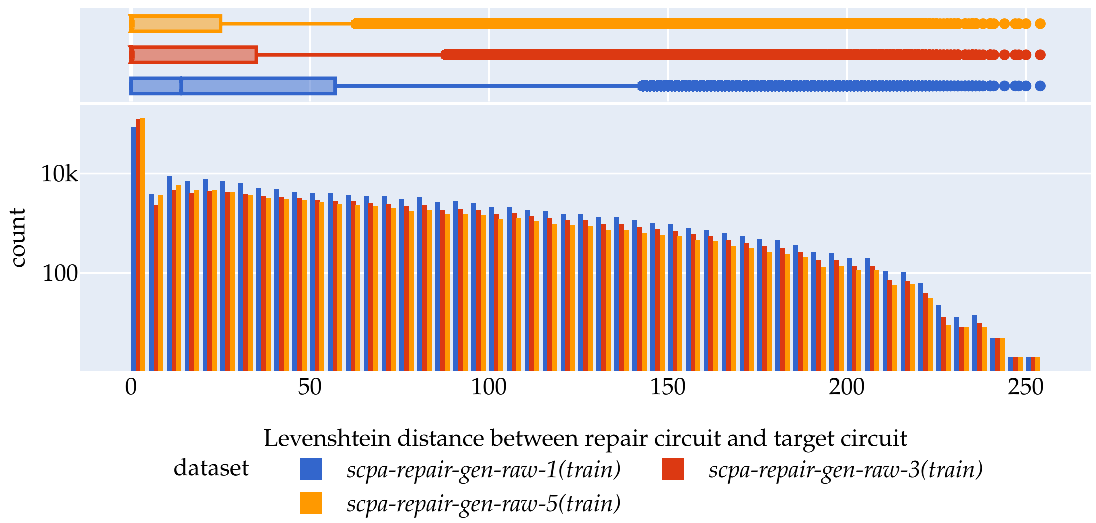


Figure 4.4.: Levenshtein distance between repair circuit and target circuit. *scpa-repair-gen-raw-1*: original raw dataset. *scpa-repair-gen-raw-3*: misleading targets from semantically correct samples removed. *scpa-repair-gen-raw-5*: misleading targets from violating samples removed. Beam size of 1 in all datasets. Notice the logarithmic scale of the y-axis.

that errors of the Neural Circuit Synthesis are in the representation, we choose the target with the smallest Levenshtein distance to the repair circuit as the new target.

Fig. 4.4 shows the effect of these changes on the distribution of the Levenshtein distance between the repair circuit and the (new) target. The difference between *scpa-repair-gen-raw-1* and *scpa-repair-gen-raw-3* shows how samples from almost all bins in the histogram are moved to the 0-bin, as the target circuit and repair circuit become identical. The match fraction (fraction of samples, where target circuit and repair circuit are identical) is 39% in *scpa-repair-gen-raw-1*, whereas in *scpa-repair-gen-raw-3*, the match fraction is 60%. For the next step, we removed misleading targets from violating circuits, perceptible in the difference between *scpa-repair-gen-raw-3* and *scpa-repair-gen-raw-5*. The max fraction stays identical. However, bins with a Levenshtein distance smaller than approximately 25 significantly grow, while other bins shrink. The mean of the Levenshtein distance of violating samples changes from 60.65 in *scpa-repair-gen-raw-3* to 50.64 in *scpa-repair-gen-raw-5*. The median changes from 50 to 37.

4.3.2. Filtering Dataset

We can further notice that a significant fraction of all samples contains a satisfying repair circuit. After applying the algorithms from the previous Sec. 4.3.1, these satisfying repair circuits are identical to the target circuit, resulting in a match fraction of 60%. As we

→Sec. 4.3.1, p. 26

exclude model checking from our problem of repairing circuits, these samples do not have much value during training, since they are easily solved by predicting the repair circuit. Since a significant fraction of the dataset has this feature, we experienced that it might guide the network to solely learn to copy the repair circuit instead of repairing it. We counteract by introducing a parameter $filter_{max-match}$, limiting the number of samples where the repair circuit is identical to the target circuit.

Additionally, we suspect that samples, where the edit distance between the repair circuit and the target circuit is considerably large, do not contribute to the learning process. We try to reason how the Neural Circuit Synthesis predicts such samples. We assume that in these cases, the Neural Circuit Synthesis either produced such wrong circuits, that it is not effective to try to repair these, or that they are close to an alternative target we could not find. Given the two explanations, such samples would be dispensable for the training process. We introduce the parameter $filter_{max-dist}$ to define the largest Levenshtein distance allowed in the dataset.

We choose two alternative methods to balance the dataset. First, we can remove inadvertent samples by undersampling the matches and removing samples with large edit distances. Alternatively, we can replace the repair circuit in such samples with an artificially altered target circuit based on the algorithm from Sec. 4.2. This can be set with the parameter $filter_{process} = (\textit{remove} \mid \textit{alter})$. → Sec. 4.2, p. 23

We generate a collection of different datasets covering a range of parameters. Datasets created based on this section are named with the prefix `exp-repair-gen`, with a number following, that simply iterates over all datasets from this section. Appendix A.1 gives an overview. We show the differences between the final dataset and the steps towards it for the default parameters in Fig. 4.5. *scpa-repair-gen-raw-5* shows the dataset after replacing misleading targets as the results of Sec. 4.3.1. *scpa-repair-gen-77* and *scpa-repair-gen-81* show two different final datasets, both are created with the same default parameters, but in *scpa-repair-gen-81*, we undersample by removing all samples that do not fit our filter criteria. In *scpa-repair-gen-77* we do not remove samples but create a new repair circuit by altering the target circuit Sec. 4.2 for all samples that do not fit the filter criteria. As clearly visible, no samples with a Levenshtein distance larger than 50 are part of the dataset *scpa-repair-gen-81* as such samples are removed. Further, only 10% of all samples are matches, as the parameter `max-match-fraction` defined it. This makes the distribution better balanced. Alternatively, as seen in *scpa-repair-gen-77*, a larger Levenshtein distance than 50 is possible when altering instead of removing. This is because we mix the distribution of *scpa-repair-gen-81* with the distribution of *scpa-repair-alter-1* from Fig. 4.2. We think that altering the circuits instead of removing them is generally better. We suspect that there is a codependency between easier specifications and correct circuits, as well as harder specifications and a larger Levenshtein distance. Following this assumption, the Neural Circuit Synthesis can be seen as an oracle, classifying which specifications are easier and which specifications are hard - by either giving a satisfying circuit or by calculating the degree it fails to give a satisfying circuit using the Levenshtein distance

→ Appendix A.1, p. 63
→ Fig. 4.5, p. 30
→ Sec. 4.3.1, p. 26
→ Sec. 4.2, p. 23
→ Fig. 4.2, p. 25

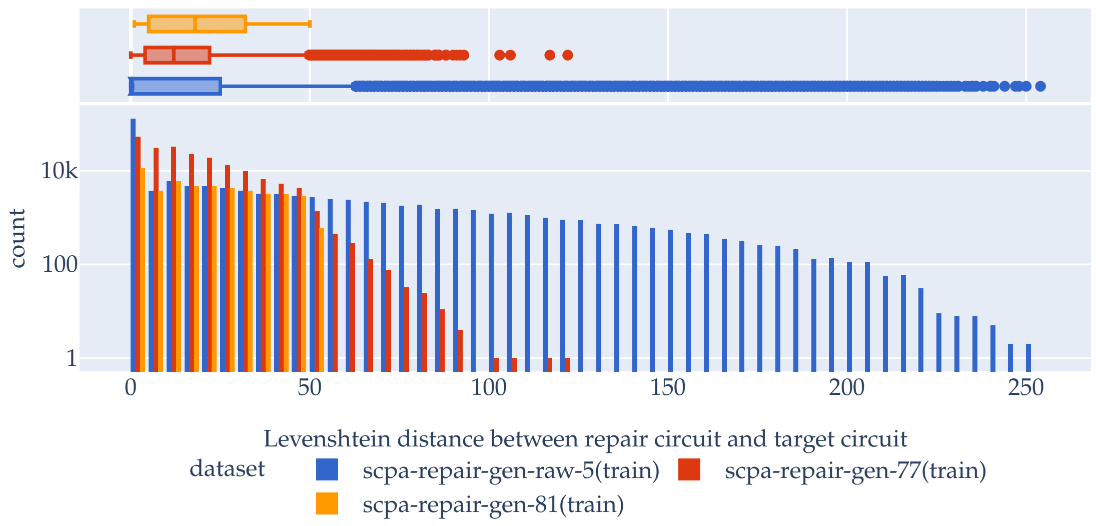


Figure 4.5.: Levenshtein distance between repair circuit and target circuit. Beam size of 1 in all datasets. Notice the logarithmic scale of the y-axis.
scpa-repair-gen-raw-5: misleading targets removed. *scpa-repair-gen-77*: final dataset with filters applied ($filter_{max-dist} = 50, filter_{max-match} = 0.1, filter_{process} =$ alter). *scpa-repair-gen-81*: final dataset with filters applied ($filter_{max-dist} = 50, filter_{max-match} = 0.1, filter_{process} =$ remove).

between violating circuit and target circuit. This would imply that easy specifications lead to matches in our raw dataset and harder specifications lead to large Levenshtein distances. After filtering by removing samples, easy specifications, as well as demanding specifications, are not included in the dataset. Altering the circuit, however, would allow all specifications in the dataset. Only the repair circuit will be changed.

4.4. Final Datasets

We now show three selected datasets, their parameters, and some statistics on these datasets. For more information on other datasets, we refer to Appendix A.1.

➔ Appendix A.1, p. 63

We show the distribution of the Levenshtein distance of the selected datasets in Figure Fig. 4.6

➔ Fig. 4.6, p. 31

Dataset *scpa-repair-alter-19* The dataset *scpa-repair-alter-19* is based on altered circuit data. We do not delete any lines from circuits ($P_{deletes} = 0$), perform at most 100 changes per circuit, with approximately 68% samples having at most 50 changes ($changes_{max} = 100, changes_{range-68} = 50$). We sample the new variable number from a normal distribution around the old variable number with approximately 68% chance

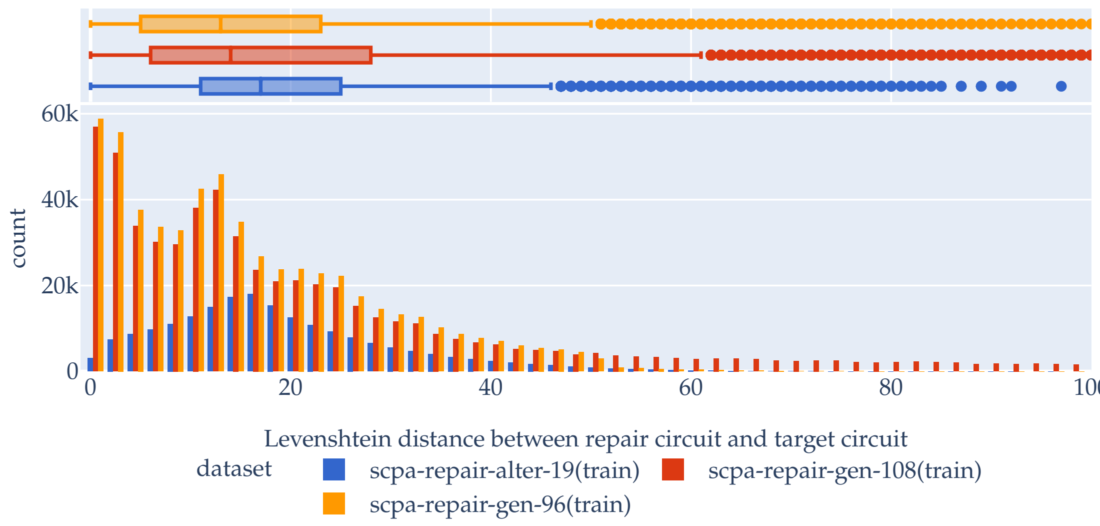


Figure 4.6.: Levenshtein distance between repair circuit and target circuit.

of a maximal distance of 20 ($var_{range-68} = 20$). This procedure leads to less than 1% of repair circuits that still satisfy the specification. 20% of all samples have a Levenshtein distance of less than 10 between the repair circuit and the target circuit. In total the Levenshtein distance in the dataset has a mean of 19.18 with a standard deviation of 12.97 and the median is at 17.

Dataset *scpa-repair-gen-108* The dataset *scpa-repair-gen-108* is based on evaluation circuit data and supplemented with altered circuits. We use a beam size of 3 to generate circuits ($parent_{beam-size} = 3$), then change misleading targets by finding minimal alternative targets, and finally filter the resulting samples. We only keep evaluation samples with a Levenshtein distance greater than 0 and smaller than 100 in the dataset ($filter_{max-match} = 0, filter_{max-dist} = 100$), hence for the rest we replace the repair circuits with altered circuits ($filter_{process} = alter$). For generating altered circuits, for 20% of all changes we delete a line from the circuit ($P_{deletes} = 0.2$), and perform at most 100 changes per circuit with approximately 68% samples having at most 15 changes ($changes_{max} = 100, changes_{range-68} = 15$). We sample the new variable number from a normal distribution around the old variable number with approximately 68% chance of a maximal distance of 20 ($var_{range-68} = 20$). This procedure leads to approximately 2% of altered circuits that still satisfy the specification. 35% of all samples have a Levenshtein distance of less than 10 between the repair circuit and the target circuit. In total the Levenshtein distance in the dataset has a mean of 21.14 with a standard deviation of 21.48 and the median is at 14.

Dataset *scpa-repair-gen-96* The dataset *scpa-repair-gen-96* is based on evaluation circuit data and supplemented with altered circuits. We use a beam size of 3 to generate circuits ($parent_{beam-size} = 3$), then change misleading targets by finding minimal alternative targets, and finally filter the resulting samples. We only keep evaluation samples with a Levenshtein distance greater than 0 and smaller than 50 in the dataset ($filter_{max-match} = 0, filter_{max-dist} = 50$), hence for the rest we replace the repair circuits with altered circuits ($filter_{process} = alter$). For generating altered circuits, for 20% of all changes we delete a line from the circuit ($P_{deletes} = 0.2$), and perform at most 50 changes per circuit with approximately 68% samples having at most 15 changes ($changes_{max} = 50, changes_{range-68} = 15$). We sample the new variable number from a normal distribution around the old variable number with approximately 68% chance of a maximal distance of 20 ($var_{range-68} = 20$). This procedure leads to approximately 2% of altered circuits that still satisfy the specification. 38% of all samples have a Levenshtein distance of less than 10 between the repair circuit and the target circuit. In total the Levenshtein distance in the dataset has a mean of 15.7 with a standard deviation of 12.77 and the median is at 13.

Chapter 5

Experiments

In this chapter, we document our training parameters, show some general results for all models and further analyze the evaluation of three selected models in more detail.

5.1. Hyperparameters and Training

All models are trained with the same hyperparameters but on different datasets. We trained a separated hierarchical Transformer with 4 heads in all attention layers, 4 stacked local layers in both separated local layers, and 4 stacked layers in the global layer. The decoder stack contains 8 stacked decoders. Embedding size in the decoder and encoder is 256 and all feed-forward networks have a size of 1024 and use the Rectified Linear Unit (ReLU) activation function. As in [Sch+21], we used the Adam optimizer [KB17] with $\beta_1 = 0.9, \beta_2 = 0.98$ and $\epsilon = 10^{-9}$. We also use 4000 warmup steps with an increased learning rate as proposed in [Vas+17]. We trained with a batch size of 256 for at least 20000 steps. We show the accuracy training curve for selected models in Fig. 5.1. All models are named after the dataset the model is trained on. Therefore, models containing *gen* are based on datasets from in Sec. 4.3 and models containing *alter* are based on datasets from Sec. 4.2. The second to last number determines the exact dataset, and the last number iterates over all models based on the same dataset. We refer to Appendix A.2 and Appendix A.1 for more details on (hyper-)parameters of experiments and datasets respectively.

→ Fig. 5.1, p. 34
→ Sec. 4.3, p. 25
→ Sec. 4.2, p. 23
→ Appendix A.2, p. 64
→ Appendix A.1, p. 63

5.2. Evaluation

All models have stable training and reach competitive semantic and syntactic accuracy. This indicates that hyperparameters are chosen well, datasets are balanced and the separated hierarchical Transformer architecture can be trained to solve the problem of

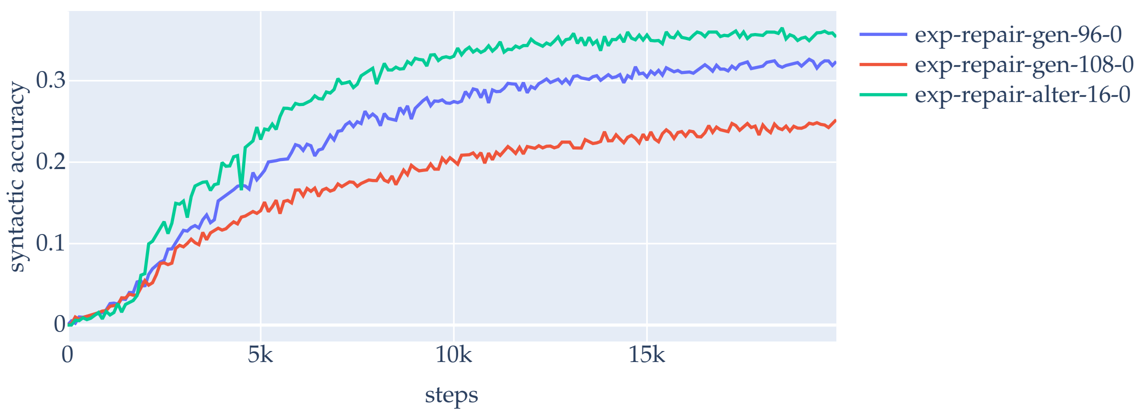


Figure 5.1.: Validation accuracy of selected models over training steps.

circuit repair. In Fig. 5.2, we show the semantic accuracy on the validation splits, with beam size $bs = 16$. The accuracy ranges from 63% up to 89%. However, because the datasets used to train each model, vary in difficulty, comparisons of different models by test or validation accuracy only have limited expressiveness. One indicator of the difficulty of a dataset is the Levenshtein distance between the repair circuit and the target circuit. Samples with a smaller Levenshtein distance are presumably easier to solve. Therefore we plot the mean and median of the Levenshtein distance on the y-axis of Fig. 5.2. We can infer the tendency that models with high validation accuracy were also trained and evaluated on easier datasets. This shows that comparisons between models should be treated with caution and we should not determine the best model based on the achieved test or validation accuracy. We compare the different models against each other using a common baseline in Chapter 6.

→Chapter 6, p. 43

For further analysis, we select three experiments. We chose two models based on predicted circuit data (Sec. 4.3) and one model based on altered circuit data (Sec. 4.2). All evaluations are based on the test split of the respective datasets and use beam search with a beam size of 16. We provide a Jupyter Notebook in the repository[1] to replicate the results for the selected experiments and all experiments we trained.

→Sec. 4.3, p. 25
→Sec. 4.2, p. 23

In Tbl. 5.3, we show some key results for the selected models. Surprisingly the number of correct beams per sample is relatively high, especially compared with the results from Neural Circuit Synthesis [Sch+21] which achieved 4.6 correct beams per sample. For the model *exp-repair-gen-96-0*, on average 6.57 out of 16 beams of the predicted circuits satisfy the specification. Looking only at solved samples, hence samples where at least the prediction of one beam satisfies the specification, even 7.75 satisfying circuits were found on average. Not only does the model find more alternative solutions, but it also finds the exact target circuit more often than in [Sch+21], which has a syntactic accuracy of 44.5%. While a higher syntactic accuracy seems intuitive as we support a repair

[1] https://github.com/MatCos/ml2/blob/main/notebooks/experiments.ipynb

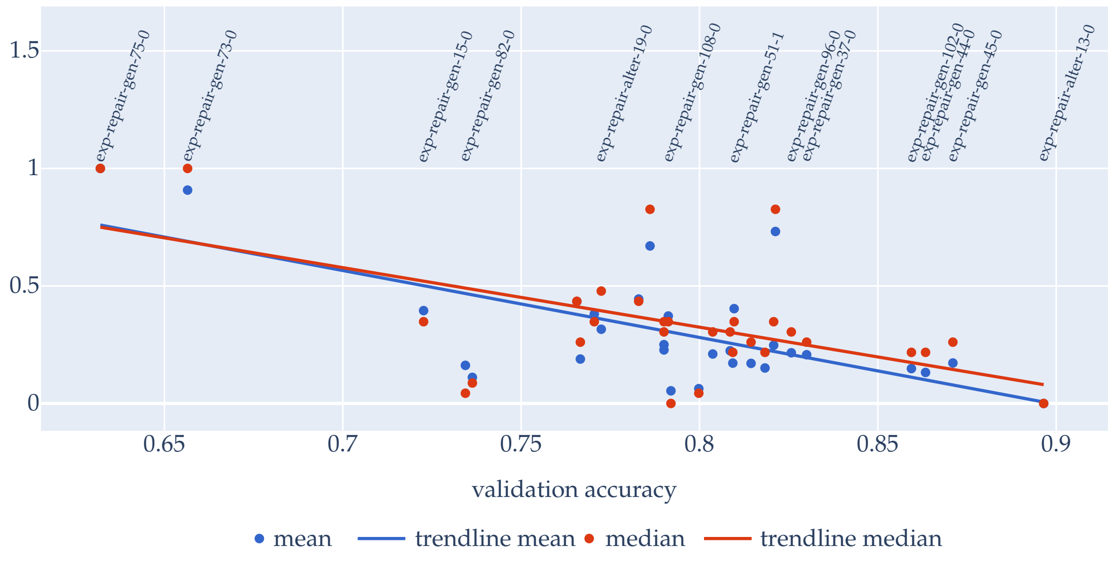


Figure 5.2.: Validation accuracy ($bs = 16$) of all trained models compared to the Levenshtein distance between the repair circuit and target circuit. Mean and median are scaled from smallest and largest values to 0 and 1. Only the labels of selected models are shown.

experiment	exp-repair-alter-19-0	exp-repair-gen-108-0	exp-repair-gen-96-0
dataset	scpa-repair-alter-19	scpa-repair-gen-108	scpa-repair-gen-96
semantic accuracy	$791/1024 = 77\%$	$803/1024 = 78\%$	$868/1024 = 84\%$
syntactic accuracy	$601/1024 = 59\%$	$455/1024 = 44\%$	$545/1024 = 53\%$
correct beams per sample	4.15	5.79	6.57
correct beams per solved sample	5.34	7.38	7.75

Table 5.3.: Key results of selected models.

circuit that is close to the target circuit, it surprises that the model is also more capable to find other solutions than the target circuit. This shows that the model can utilize the repair circuit beyond character editing toward the target circuit.

5.2.1. Difficulty Indicators

We will now look at three indicators for how complex/difficult it is to solve a sample. Naturally, these indicators cannot be extensive and only highlight one aspect of the hardness of a sample. We will first look at the specification size, then the target size, and finally the Levenshtein distance between the repair circuit and the target circuit.

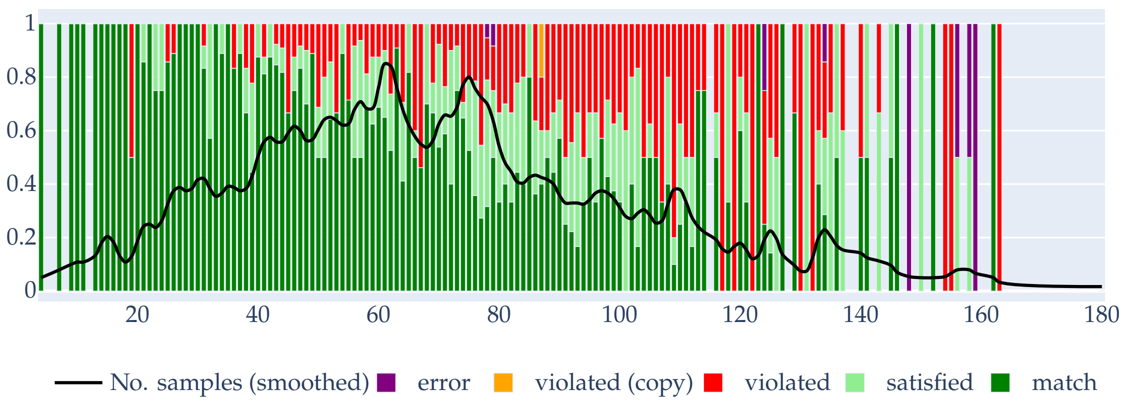
We categorize all samples based on the indicator into bins. For each bin, we show the percentage of all possible statuses:

- *Match*: prediction equals target.
- *Satisfied* prediction satisfies the specification, matches excluded.
- *Violated (Copy)* prediction violates the specification and is equal to the repair circuit.
- *Violated* prediction violates the specification, copies excluded.
- *Error* valid circuit could not be determined.

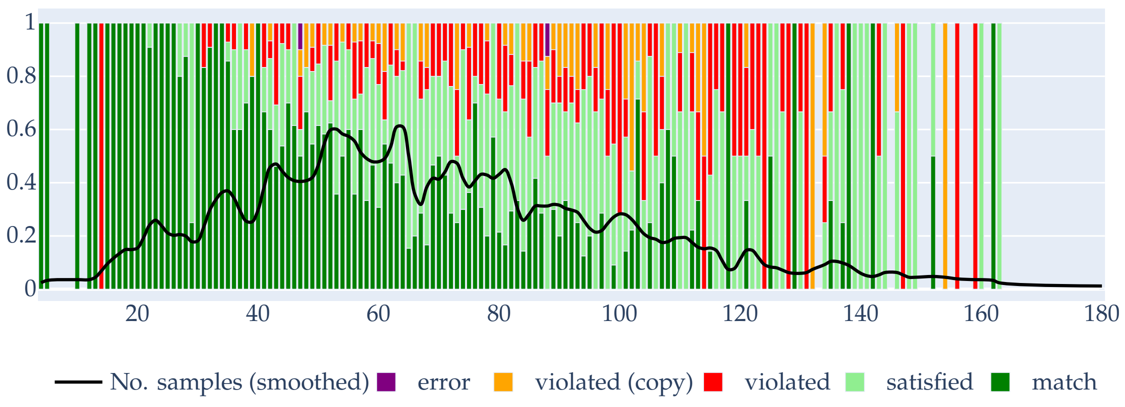
Specification Size In Fig. 5.4 we categorize all samples by specification size and show what percentage of samples in each bin is solved correctly. We determine the size of the specification by calculating the size of the abstract syntax tree (AST) of the formula. We also insinuate the number of samples in each bin with a line. All specifications in the three datasets come from the same distribution, hence the number of samples per bin is similar in all three experiments. We can see that in all experiments, performance goes down with larger specification sizes. This is not surprising, as larger specifications typically are more complicated. However, for *exp-repair-gen-96-0*, this is not as pronounced as in both other experiments. Especially with specification sizes from around 80 to 110, performance is significantly better than for both other models. Further, for model *exp-repair-alter-19-0*, we can see that for a significant portion of samples with larger specifications, no circuit could be produced. In total 8 of 1024 samples only produced errors in all beams. We also display if a model just copied the repair circuit. The model *exp-repair-alter-19* only produced one circuit that violates the specification and is a copy of the repair circuit. In both other models, significantly more samples have this status: *exp-repair-gen-108-0*: 65, *exp-repair-gen-96-0*: 31.

➔ Fig. 5.5, p. 38

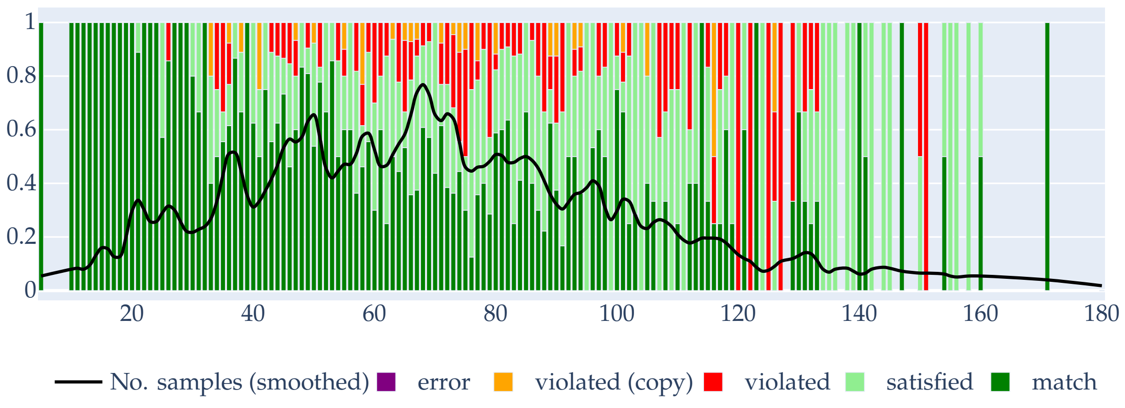
Target Circuit Size In Fig. 5.5 we categorize all samples by target circuit size and show what percentage of samples in each bin is solved correctly. We calculate the size of a circuit by counting AND gates and latches. We also show the number of samples in each bin with a line. All targets in the three datasets come from the same distribution, hence the number of samples per bin is similar in all three experiments. As in the previous section, we can see how samples with larger targets are solved less frequently, and as before, the effect is the smallest on the model *exp-repair-gen-96*, while the other models are comparable. The differences are highly interesting because all datasets (and consequently models) only differ in the repair circuit. All datasets have the same specifications and targets, hence all models have seen the same specifications and targets. However, these results differ greatly, showing that the repair circuit has an important influence on the training and evaluation of the models.



(a) experiment exp-repair-alter-19-0

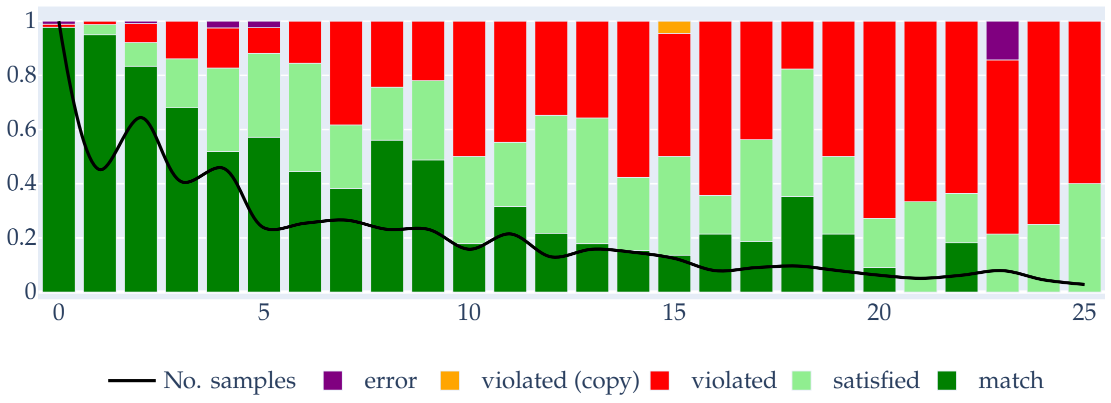


(b) experiment exp-repair-gen-108-0

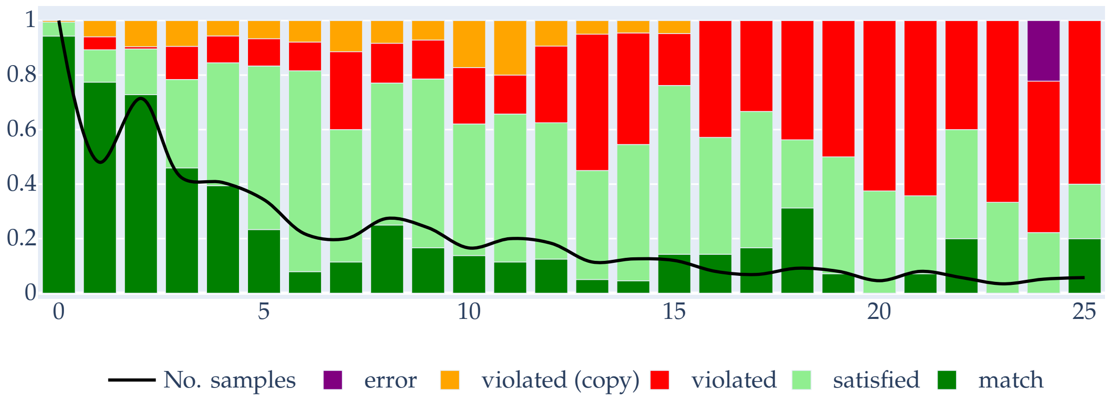


(c) experiment exp-repair-gen-96-0

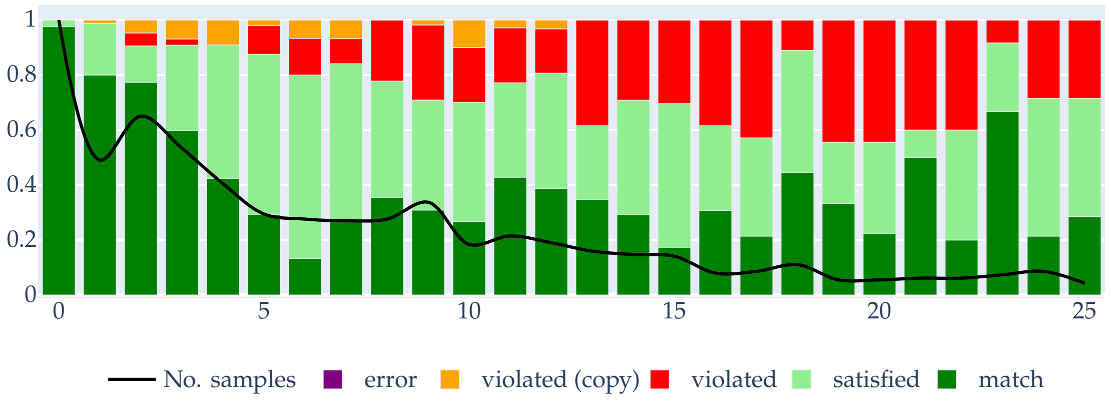
Figure 5.4.: The status of samples, categorized by the size of the specification AST. The line insinuates the number of samples in each bin, scaled from 0 to 1. For a better overview, the line is smoothed with a Hann-Window. Only the best result from all 16 results per sample is shown.



(a) experiment exp-repair-alter-19-0



(b) experiment exp-repair-gen-108-0



(c) experiment exp-repair-gen-96-0

Figure 5.5.: The status of samples, categorized by the target circuit size (sum of latches and AND gates). The line shows the number of samples in each bin, scaled from 0 to 1. Only the best result from all 16 results per sample is shown.

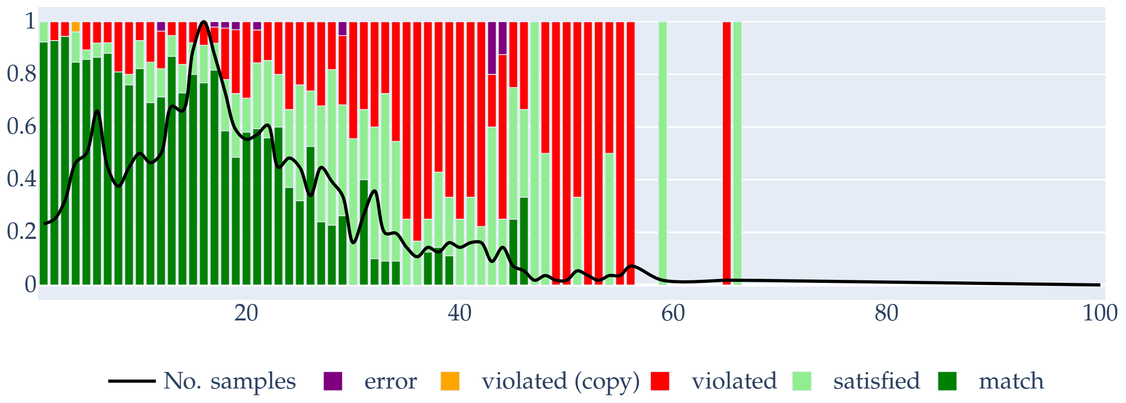
Levenshtein Distance In Fig. 5.6 we categorize all samples by the Levenshtein distance between the repair circuit and target circuit and show what percentage of samples in each bin is solved correctly. We also show the number of samples in each bin with a line. While the targets in the three datasets come from the same distribution, the repair circuit distribution is unique to each dataset. We already showed the distribution of the Levenshtein distance for the three datasets in Fig. 4.6. Comparing the three figures, we can see that the first model *exp-repair-alter-16-0* is, again, most affected by harder samples. Because of the parameters used to generate the repair circuits (see Sec. 4.4), only outliers have a larger Levenshtein distance than 50 in *exp-repair-alter-19-0* and *exp-repair-gen-96-0*.

→Fig. 5.6, p. 40
→Fig. 4.6, p. 31
→Sec. 4.4, p. 30

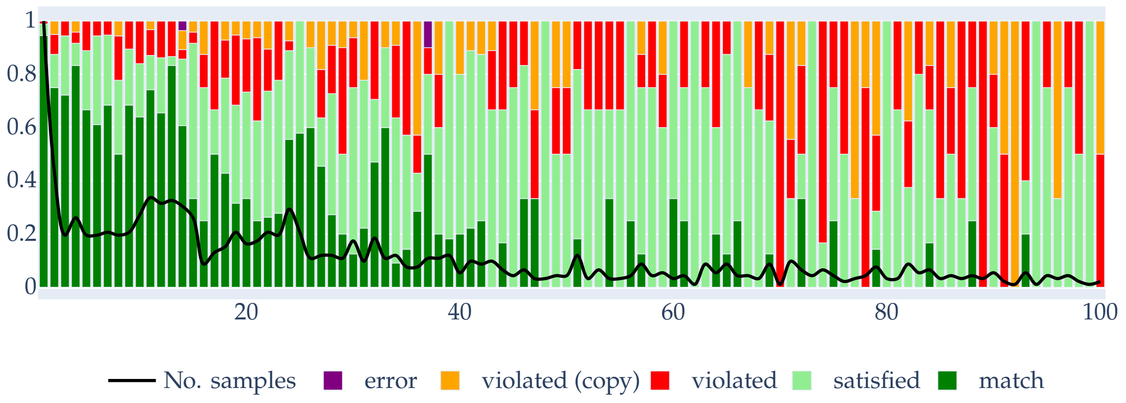
5.2.2. Levenshtein Distance Improvement

Although all models have good results, a significant amount of samples could still not be solved. In this section, we analyze whether a violating prediction is still an improvement. We calculate the difference between two Levenshtein distances: The distance between repair and target circuit $lev(r, t)$ and the distance between prediction and target circuit $lev(p, t)$. If the difference is below zero: $lev(p, t) - lev(r, t) < 0$, the model improved the repair circuit toward the target circuit. If not, the model might either improve toward an alternative target or just deteriorate. The more extreme the better/worse. In Fig. 5.7, we show a violin plot of the improvement. A violin plot visualizes a distribution by showing the probability density of the data. First, we can see that for all models, for violated or satisfied predictions, and for realizable or unrealizable specifications, the prediction is an improvement compared to the repair circuit. Since this holds for violating and satisfying predictions we conjecture that most violating predictions are still an improvement compared to the repair circuit, hence the status before applying the repair model. In the next Chapter 6, we analyze whether we can utilize this by repeatedly applying this approach to the predicted circuit(s) until we eventually might converge.

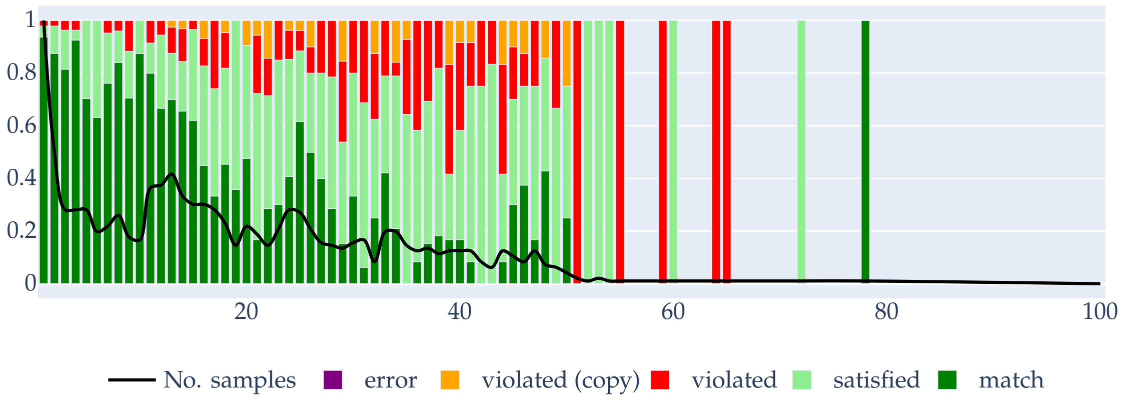
→Fig. 5.7, p. 41
→Chapter 6, p. 43



(a) experiment exp-repair-alter-19-0

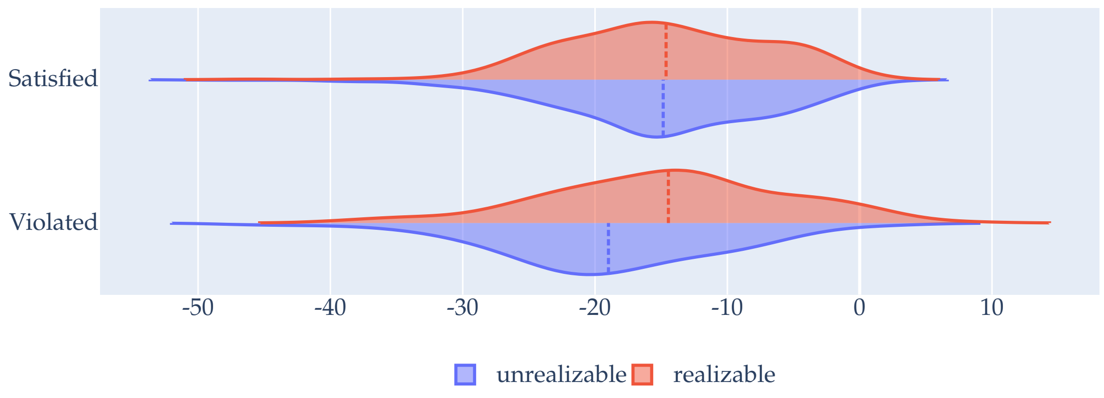


(b) experiment exp-repair-gen-108-0

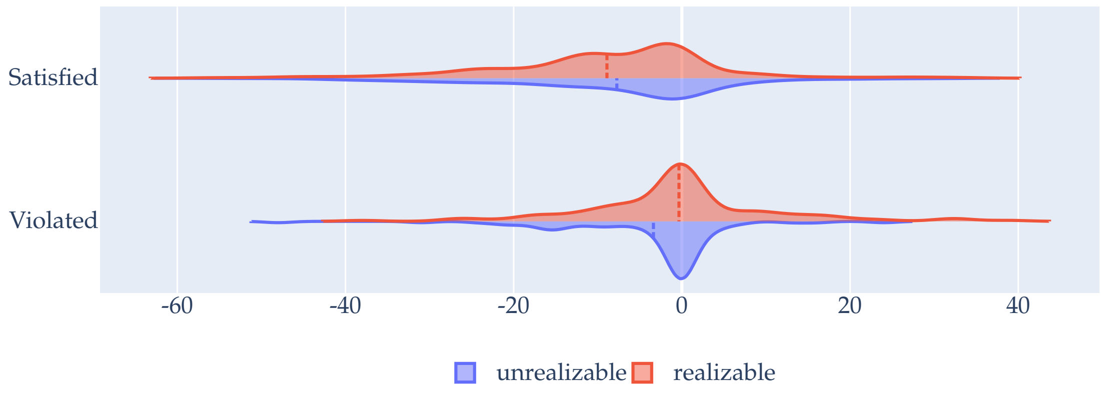


(c) experiment exp-repair-gen-96-0

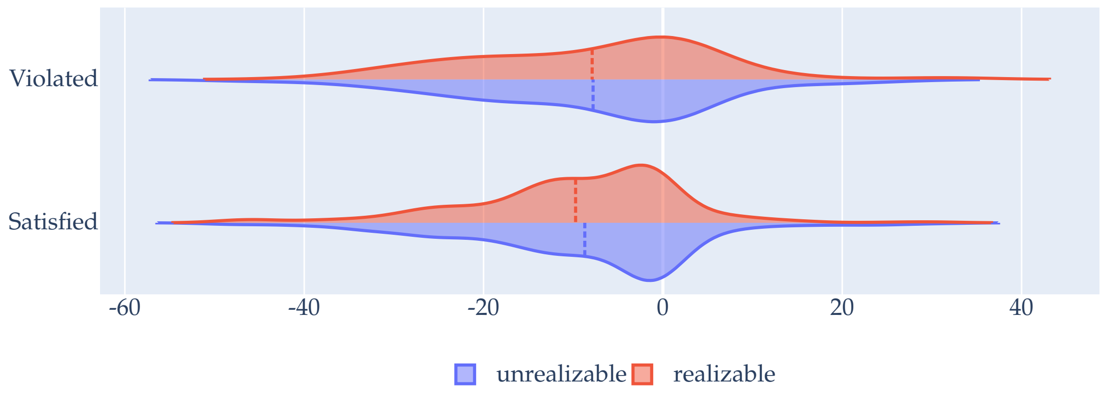
Figure 5.6.: The status of samples, categorized by the Levenshtein distance between the repair circuit and target circuit. The line shows the number of samples in each bin, scaled from 0 to 1. Only the best result from all 16 results per sample is shown.



(a) experiment *exp-repair-alter-19-0*



(b) experiment *exp-repair-gen-108-0*



(c) experiment *exp-repair-gen-96-0*

Figure 5.7.: Violin plot of the improvement of the Levenshtein distance from the repair circuit and prediction to the target. Smaller (negative) numbers indicate that the predicted circuit is closer to the target circuit than the repair circuit; larger (positive) numbers indicate that the predicted circuit is further from the target circuit than the repair circuit. The dashed line shows the mean of the displayed distribution. For better visibility, we removed extreme outliers and only show samples between -60 and 40.

Chapter 6

Improving Neural Circuit Synthesis

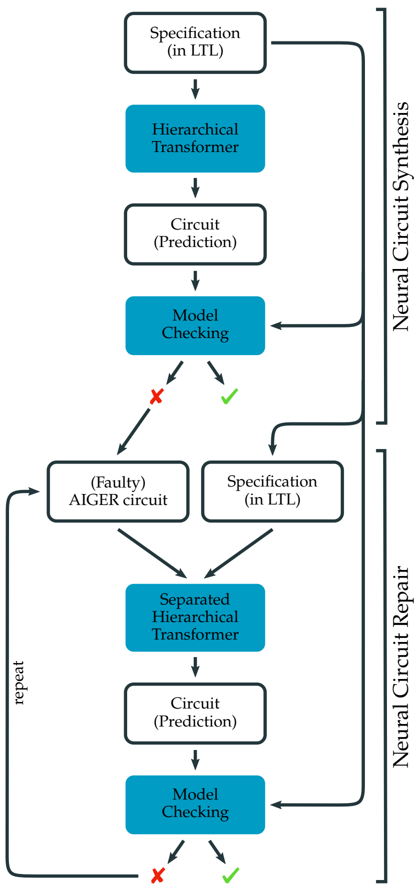


Figure 6.1.: Pipeline Structure

In this chapter, we present the results of combining this work with the work of [Sch+21]. We extend the previous work and improve the current state-of-the-art approach for Neural Circuit Synthesis.

We combine both approaches by using Neural Circuit Synthesis (base model, [Sch+21]) to predict circuits and Neural Circuit Repair (repair model, Chapter 5) to repair predictions that violate the specification. We call the following procedure *pipeline* and show an overview in Fig. 6.1.

➔Chapter 5, p. 33

- We first evaluate the base model. As a base model, we use the same model we trained for Sec. 4.3, which was trained on the dataset *scpa-2*. We call this the *0th iteration* of the pipeline or *base model evaluation*.

➔Sec. 4.3, p. 25

- If the predicted circuit violates the specification, we feed the specification together with the violating circuit into the repair model. We call this the *1st iteration* of the pipeline or first *repair model evaluation*.
- If the prediction of the repair circuit still violates the specification, we can repeat feeding the specification together with the violating cir-

cuit into the repair model until it is solved. We call this the *2nd to nth iteration* of the pipeline or the *repeats of the repair model*.

Using this approach we improve the results of Schmitt et al. [Sch+21] by 6.8 percentage points to a total of 83.9% on held-out instances. We could also see similar or even greater improvement on out-of-distribution benchmarks such SYNTCOMP [Jac+22a; Jac+22b] with a gain of 11.8 percentage points to a total of 75.9% and J.A.R.V.I.S [Gei+22] with a gain of 23.8 percentage points to a total of 66.7%. In the following, we give more details on these results and investigate which part of the pipeline holds the largest stake in the improvement.

6.1. Synthesizing an Arbiter

Before we present some generalized insights into the pipeline evaluation we give the demonstration of synthesizing an arbiter using the pipeline. The specification and results shown in this section are actual results using the repair model *exp-repair-gen-96-0*. An arbiter manages the access to a resource to which multiple processes can request access. The arbiter is a popular example in literature for reactive synthesis. We look at the more advanced version of an arbiter for 4 processes which was taken from the SYNTCOMP benchmark [Jac+22b; Jac+22a]. The requirements for a simple arbiter are as follows. First, only one process at once should have access to the resource. Secondly, if a process requested the resource, the arbiter should not ignore the request, hence the process should have access to the resource sometimes in the future. We can specify these requirements using LTL. For processes $p_0 \cdots p_3$, we use the requests $r_0 \cdots r_3$ as inputs and grants $g_0 \cdots g_3$ as outputs. A process p_i can request the resource by setting r_i to true and receives a grant if g_i is true. We create the following 5 guarantees that fully specify an arbiter.

$$\square((\neg g_0 \land \neg g_1 \land (\neg g_2 \lor \neg g_3)) \lor ((\neg g_0 \lor \neg g_1) \land \neg g_2 \land \neg g_3))$$

$$\square(r_0 \rightarrow \diamondsuit g_0)$$

$$\square(r_1 \rightarrow \diamondsuit g_1)$$

$$\square(r_2 \rightarrow \diamondsuit g_2)$$

$$\square(r_3 \rightarrow \diamondsuit g_3)$$

A circuit that satisfied the requirements does not need to consider the requests, as giving grants based on a round-robin scheduler would satisfy the requirements. The smallest circuit to satisfy the requirements, therefore, would give a one grant after the other to all processes.

In Fig. 6.2, we show the result of the pipeline in different iterations. First, in Fig. 6.2a, the predicted circuit of the base model is not correct. It lacks two necessary concepts: The latches are not connected in a way they could count to memorize the 4 states we need. Secondly and more obvious, g_0 and g_1 are controlled by the same variable. That means they are either never `true` (which would violate the specification) or at some point `true` at the same time (which would also violate the specification). In Fig. 6.2b, we can see the first attempt to repair that circuit. While this circuit is still faulty because g_0 and g_1 are based on the same variable, progress was made towards a functioning counter. Latch 1 ($l1$) now is based on a combination of AND gates and negations that has the expressiveness to represent a counter. However, the counter does not work yet as the second AND gate should take the variable number 14 instead of 15 as input. In Fig. 6.2c we can see the result of feeding the previous circuit into the repair model again. Now the model predicted the correct circuit. Both latches build together a bit-wise counter that memorizes the state. The AND gates that are connected to outputs are the cross-product of the non-negated and negated versions of both latches such that always one AND gate outputs `true`. Lastly, each AND gate is connected to exactly one output. This circuit is an optimal solution in size for this requirement and also the output of the classical synthesis tool Strix [MSL18].

→ Fig. 6.2, p. 46
→ Fig. 6.2a, p. 46
→ Fig. 6.2b, p. 46
→ Fig. 6.2c, p. 46

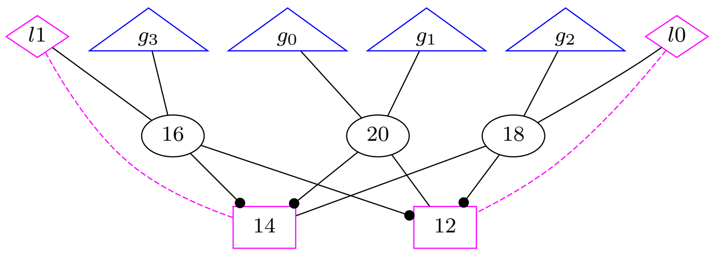
6.2. Improvement on held-out instances

For the remaining analysis, unless stated otherwise, we evaluate all repair models in the pipeline for 6 iterations (1 iteration base model and 5 iterations repair model) with beam size 16. We always keep the most promising beam after each iteration using the distance between the prediction and target circuit as a heuristic. We use the same 2048 samples, called *test_fixed*, which are randomly sampled from the test split of the Neural Circuit Synthesis dataset *scpa-2*. Therefore, in the first iteration of the repair model, we have the same samples for all evaluations of repair models. We made sure that neither the base model nor the repair model has seen samples or any parts thereof (i.e. specifications) from this split. We either show the pipeline accuracy improvement (percentage points gained after base model evaluation) or the pipeline accuracy (total percentage after the last pipeline iteration).

In all iterations of the pipeline, we use beam search (Sec. 3.7) with a beam size of 16. That entails that each iteration produces 16 distinct results per sample that we feed in. We have different options on how to handle these beams. One option is to feed all 16 results into the next iteration which leads to an exponential blowup in the number of repeats. This is feasible for small evaluation sets and a few iterations such as 3 as seen in the experiments on benchmarks (Sec. 6.4). More iterations would theoretically be possible and might lead to an even greater improvement, but 3 iterations are already sufficient for a huge improvement. For larger evaluation sets such as 2048 samples, this would theoretically still be possible but not efficient. Therefore we have a second option

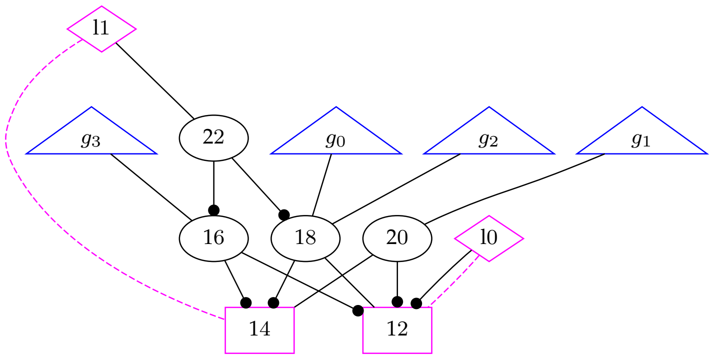
→ Sec. 3.7, p. 18
→ Sec. 6.4, p. 52

```
aag 10 5 2 5 3
2           input 0 (i0)
4           input 1 (r_2)
6           input 2 (r_0)
8           input 3 (r_3)
10          input 4 (r_1)
12 18       latch 0 (l0)
14 16       latch 1 (l1)
16          output 0 (g_3)
18          output 1 (g_2)
20          output 2 (g_0)
20          output 3 (g_1)
0           output 4 (o4)
16 15 13    and-gates
18 14 13    and-gates
20 15 12    and-gates
```



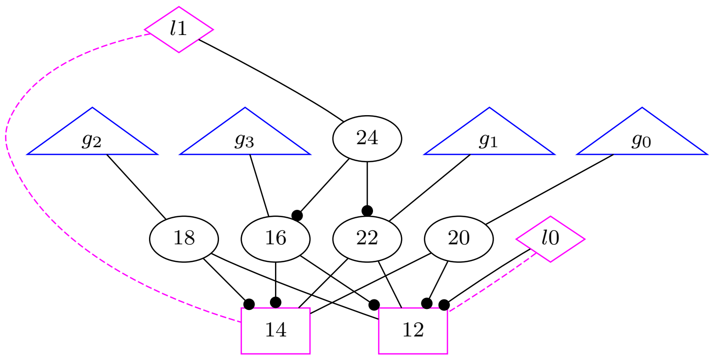
(a) Faulty circuit. Predicted in the base model (iteration 0).

```
aag 11 5 2 5 4
2           input 0 (i0)
4           input 1 (r_2)
6           input 2 (r_0)
8           input 3 (r_3)
10          input 4 (r_1)
12 13       latch 0 (l0)
14 22       latch 1 (l1)
16          output 0 (g_3)
18          output 1 (g_2)
18          output 2 (g_0)
20          output 3 (g_1)
0           output 4 (o4)
16 15 13    and-gates
18 15 12    and-gates
20 14 13    and-gates
22 19 17    and-gates
```



(b) Faulty circuit. Predicted in iteration 1 of the repair model.

```
aag 12 5 2 5 5
2           input 0 (i0)
4           input 1 (r_2)
6           input 2 (r_0)
8           input 3 (r_3)
10          input 4 (r_1)
12 13       latch 0 (l0)
14 24       latch 1 (l1)
16          output 0 (g_3)
18          output 1 (g_2)
20          output 2 (g_0)
22          output 3 (g_1)
0           output 4 (o4)
16 15 13    and-gates
18 15 12    and-gates
20 14 13    and-gates
22 14 12    and-gates
24 23 17    and-gates
```



(c) Correct circuit. Predicted in iteration 2 of the repair model.

Figure 6.2.: Predicted circuits for a 4-arbiter. In the graph, we omit inputs and outputs that are not connected to the circuit.

to select one promising beam per sample to keep for the next iteration. This can either be chosen at random or using a heuristic. To under-approximate the potential of the model if we would keep all beams, in the following, we use a heuristic on the distance between the prediction and the target.

In Fig. 6.3, we plot the accuracy improvement through the pipeline dependent on the validation accuracy. The validation accuracy is based on the distribution that was used to train the models, hence this is a different distribution for each dataset. The pipeline accuracy improvement is based on the same distribution for all models. We can see that models having a higher pipeline accuracy are trained with a dataset that included evaluation results (Sec. 4.3) instead of altered circuits (Sec. 4.2). This is not surprising, as these datasets are closer to the distribution, on which the pipeline accuracy is based. We can identify several clusters of models, of which one cluster (yellow) has relatively good validation accuracy and very good pipeline accuracy. All models in this cluster improve the synthesis accuracy by more than 5 percentage points, with the highest gain of 6.8 percentage points by the model *exp-repair-gen-96-0*.

→Sec. 4.3, p. 25
→Sec. 4.2, p. 23

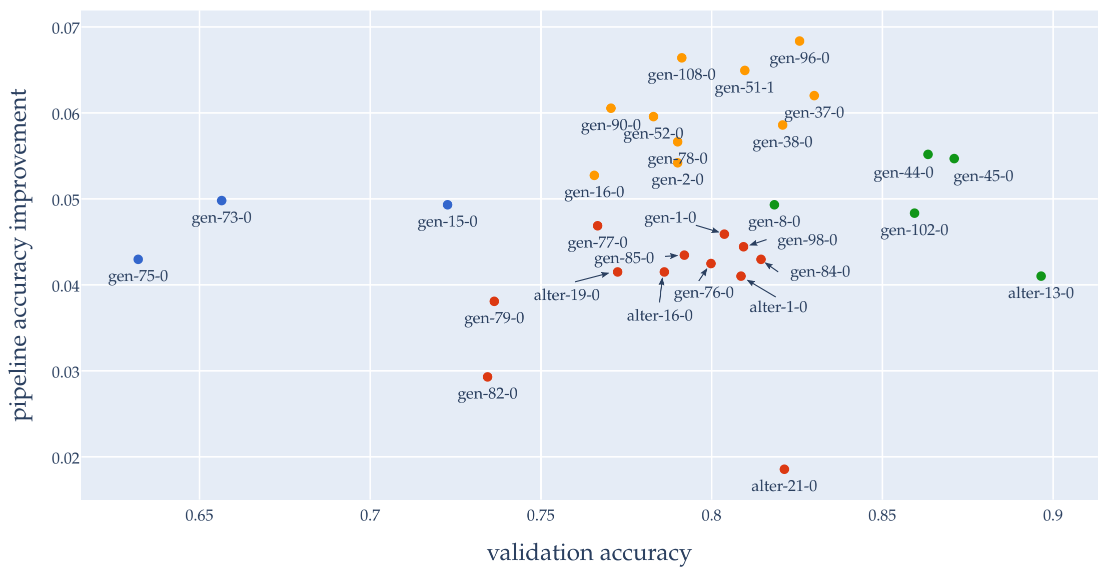


Figure 6.3.: Pipeline accuracy improvement (percentage points) compared to validation accuracy of all trained models. Colors are based on k-means clustering with 4 clusters.

In Fig. 6.4, we show the pipeline accuracy of three selected models after each iteration. *exp-repair-gen-96-0* and *exp-repair-gen-108-0* are the two models that have the highest pipeline accuracy of all models we trained and *exp-repair-alter-19-0* is the model that has the highest pipeline accuracy of all models whose training dataset is solely based on altered circuit data. For the syntactic accuracy (*Match*), we refer to Fig. A.1. As clearly visible, in all three models, the first iteration improved the most by up to 5.5

→Fig. 6.4, p. 48
→Fig. A.1, p. 59

percentage points, however, even the following iterations further improve the overall accuracy by up to 1.3 percentage points. For the model *exp-repair-alter-19-0*, we can see that the improvement is generally smaller but more distributed over all iterations. Models with the prefix *exp-repair-gen* have a major gain in the first iteration compared to later iterations. This is most likely because these models were trained with data that included circuits from evaluation predictions (Sec. 4.3). The distribution these models were trained on is similar to the distribution that is fed into the repair model in the first iteration. The model *exp-repair-alter-19-0* is trained on a different distribution, explaining the smaller improvement. Iterations 2-5 are out-of-distribution for all models.

➔Sec. 4.3, p. 25

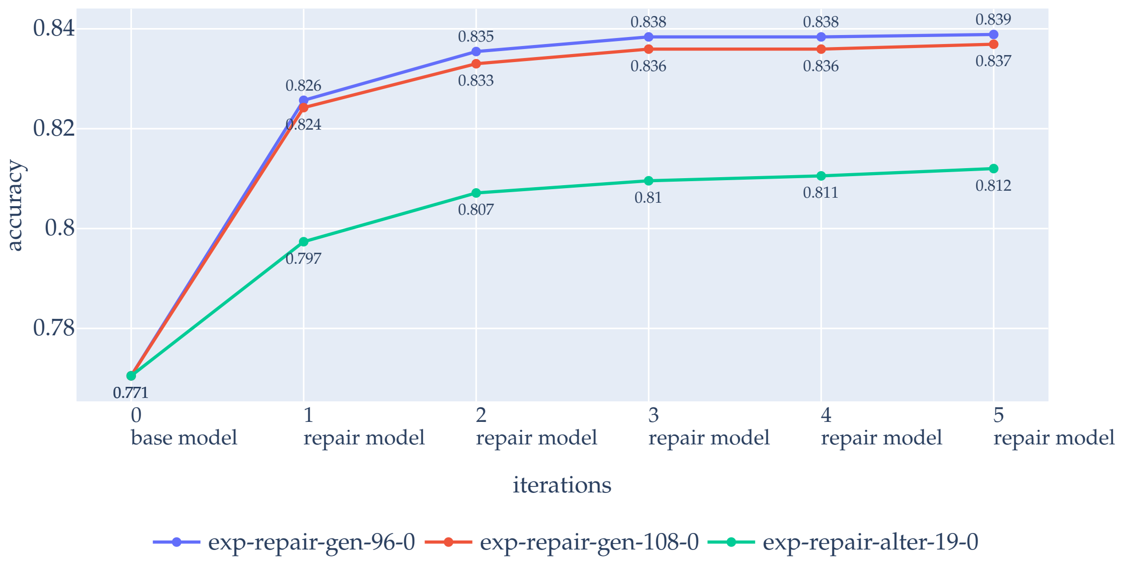


Figure 6.4.: Pipeline accuracy of selected models after each iteration step. A sample is rated as correct after iteration i if at least one prediction in evaluations from iterations $j <= i$ is correct. Based on a fixed set of held-out samples.

In Fig. 6.5, we show how the status changed between the base model and the last iteration. Samples that were already solved in the base model are shown in gray (unchanged), green samples are solved in at least one repair iteration but not in the base model (new), and red samples are neither solved in the repair model nor the base model (remaining). For the same plot but categorized by target circuit size instead of specification size, we refer to Fig. A.3. We can see that throughout all specification sizes, samples were solved that could not be solved by the base model. Most surprisingly, the repair model solves larger samples more often compared to smaller samples. This shows the advantage of the repair model, as for the base model, samples with smaller specification sizes are solved more frequently.

➔Fig. A.3, p. 60

We refer to Fig. A.4 and Fig. A.5 for the improvement between base model and first iteration only.

➔Fig. A.4, p. 61
➔Fig. A.5, p. 61

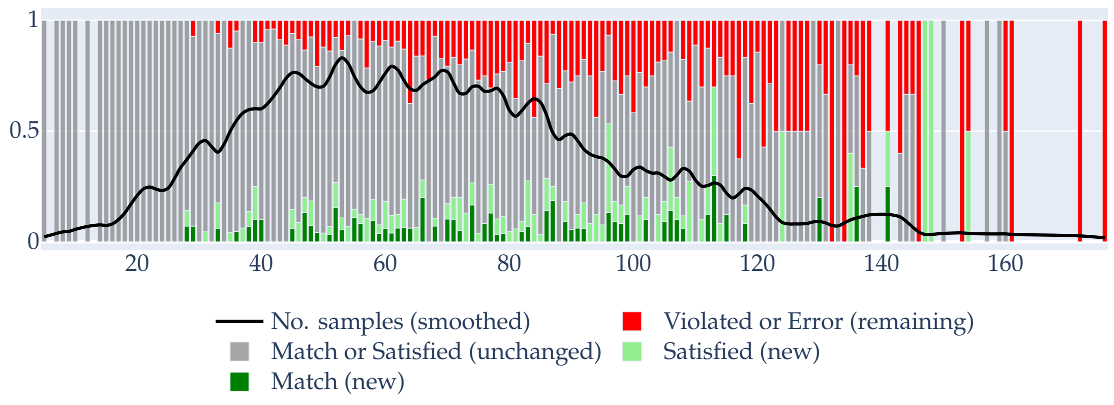


Figure 6.5.: The status improvement of samples between iteration 0 and 5, categorized by the AST size of the specification. The line insinuates the number of samples in each bin, scaled from 0 to 1. Per sample, we aggregate the best result from all iterations (base model and repair model) and all 16 beams. Based on a fixed set of held-out samples.

Finally, in Fig. 6.6, we compare the mean and median of the dataset that was used to train each repair model to the pipeline accuracy improvement. The plot is analogous to Fig. 5.2, where we showed the accuracy of each validation split. Contrary to that, we can see in this plot that the Levenshtein distance between the repair circuit and target circuit in the dataset each model was trained on has no obvious influence on the performance when measured as the improvement to the accuracy of the base model.

→Fig. 6.6, p. 50
→Fig. 5.2, p. 35

6.3. Improvement through repeats

In this section, we look at the effect of repeats in the pipeline. All evaluations in iterations larger than one are out-of-distribution evaluations because the distribution of (faulty) circuits predicted by the first iteration of the repair model is different from the distribution of the training data. As opposed to that, the model still performs exceptionally well in later iterations. We show this in Fig. 6.7, where we evaluate the model with *violated and satisfied* predictions from each previous iteration. We show the semantic accuracy in this plot, for the syntactic accuracy we refer to Fig. A.2. We show the accuracy *in* each iteration, which should not be confused with the accuracy *after* each iteration, where we rate a sample as correct if the prediction was correct in at least one earlier iteration. The question arises, as to why the model performs so well under these seemingly out-of-distribution conditions, but explanations at this point are highly speculative. Firstly it could be that the distributions are much closer to the distribution we trained on than expected. Specifications do not change through iterations, hence these are from the same distribution as we trained. Predicted circuits

→Fig. 6.7, p. 51
→Fig. A.2, p. 60

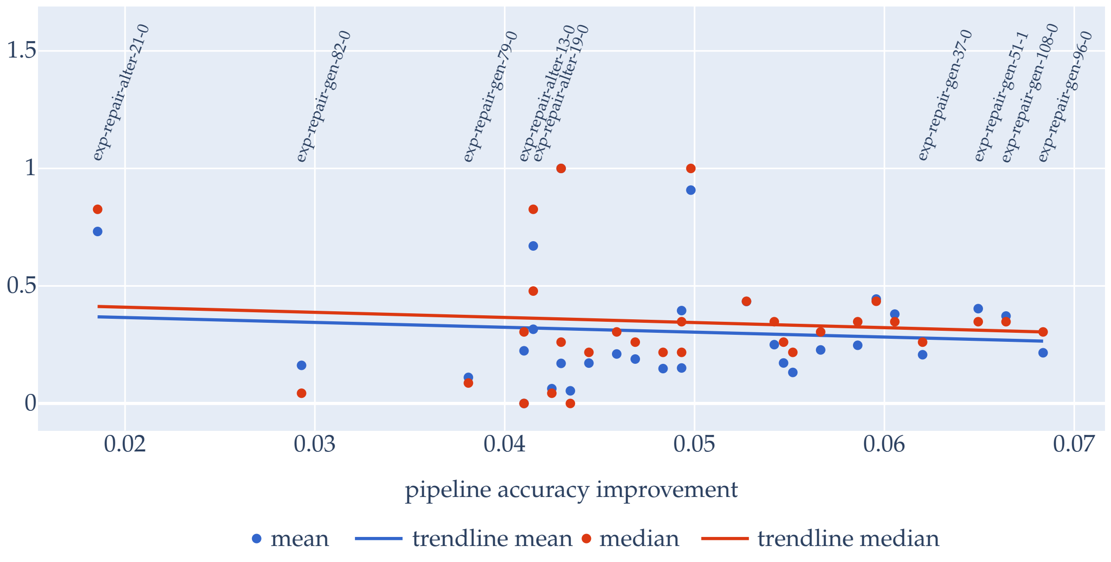


Figure 6.6.: Pipeline accuracy improvement of all trained models compared to the Levenshtein distance between repair circuit and target circuit. Mean and median are scaled from smallest and largest values to 0 and 1. Only the labels of selected models are shown.

from the base model and predicted circuits from the repair model might be very similar. Both are predicted by a model with very similar architectures (hierarchical Transformer vs. separated hierarchical Transformer), hence both models (base and repair) might produce similar mistakes, hence (faulty) circuits. Secondly, each iteration solves samples that were not solved before. By keeping the most promising beam out of all 16 predicted beams, beams that are already satisfied are kept with a higher probability. Therefore, more and more circuits we feed into the next iteration are already correct, hence the samples get easier after each iteration. This, however, does not explain the effect fully because the model not uncommonly fails on samples where the repair circuit is already correct.

In Fig. 6.8, we show the status improvement after the first iteration. Samples that were solved in the base model or the first repair iteration are shown in gray (unchanged). Green samples are solved in at least one repeat but not in the base model or the first iteration (new). Red samples are neither solved in the repair model nor the base model (remaining). Improvements are rather small, which is not surprising as, after the first iteration, an accuracy of approximately 85% is reached. The remaining unsolved samples are expected to be hard, independent of the size of the specification, hence small improvements already indicate success. As in Fig. 6.5, we cannot recognize that samples with larger specification sizes are solved less frequently. In Fig. A.6, we show the same plot but categorized by target circuit size instead of specification size.

➔Fig. 6.5, p. 49
➔Fig. A.6, p. 62

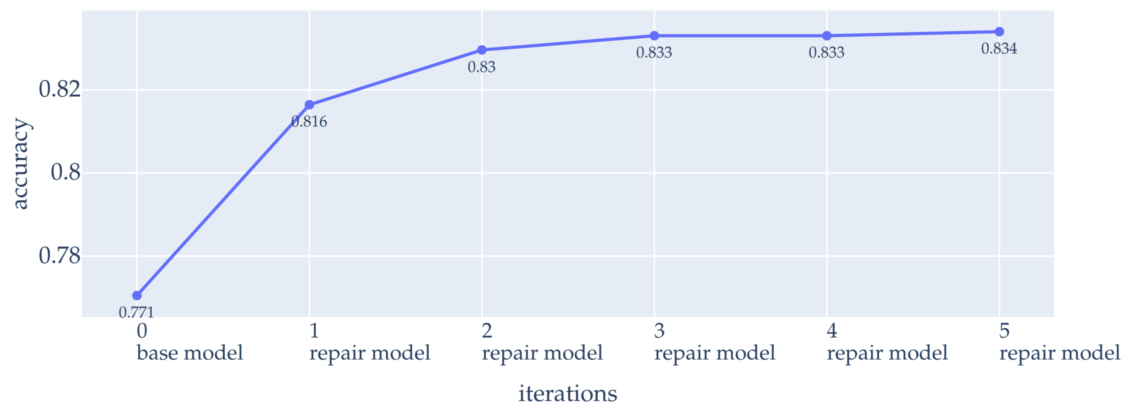


Figure 6.7.: Semantic accuracy in each iteration. Fixed test set. Beam size 16, keeping the best beam for the next iteration.

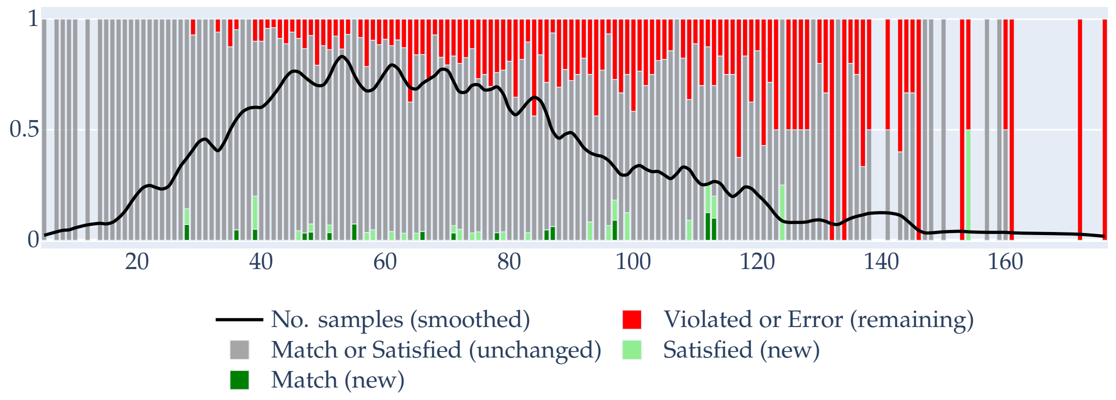


Figure 6.8.: The status improvement of samples between iteration 1 and 5, categorized by the AST size of the specification. Shows the effect of the repeats of the repair model. The line insinuates the number of samples in each bin, scaled from 0 to 1. Per sample, we aggregate the best result from all iterations (base model and repair model) and all 16 beams. Based on a fixed set of held-out samples.

6.4. Improvement on benchmarks

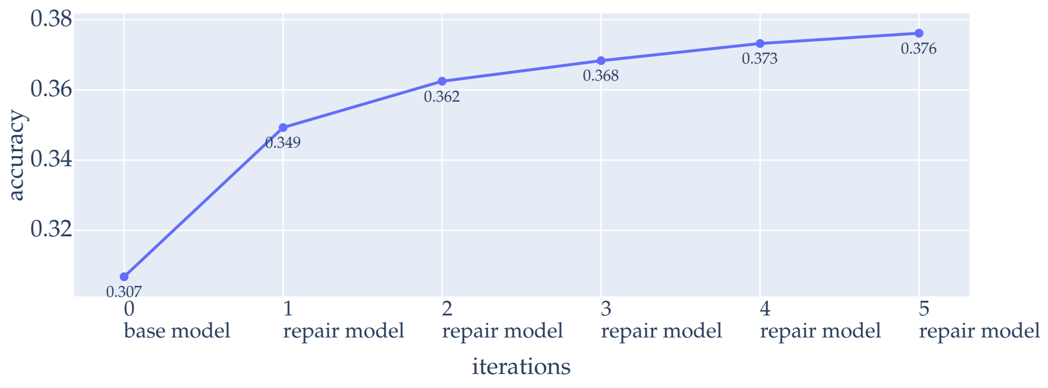
In this section we evaluate whether we can improve on benchmarks for reactive synthesis such as the reactive synthesis competition (SYNTCOMP) [Jac+22a; Jac+22b] or J.A.R.V.I.S [Gei+22], a set of specifications for Smart-Home appliances. We further look at a set of samples that is from the distribution of the dataset *scpa-2* but could not be used for training, validation, and test as the classical synthesis tool Strix [MSL18] timed out while attempting to synthesize a circuit. We call this set *timeouts*. Both benchmarks are out of distribution for the base model, hence also for the repair model. The set *Timeouts* is not out-of-distribution for the base model, but only contains hard specifications that Strix was not able to solve in 120 seconds.

We show the results in Fig. 6.9. We evaluate the three datasets in the pipeline with 6 iterations (1 iteration base model and 5 iterations repair model) with beam size 16 in all iterations. For all evaluations in this section, we do not use the under-approximating heuristic for which beam we keep. We keep a single beam chosen at random, while for SYNTCOMP and J.A.R.V.I.S, we additionally evaluate the pipeline keeping all beams with only 3 iterations. From the *timeouts* set, we sample 2048 samples for evaluation. From the SYNTCOMP and J.A.R.V.I.S benchmarks, we filter specifications that are too large for the model, with respectively 210 and 70 samples remaining which we all evaluate. Note that the different approaches for keeping beams, conceptually, cannot make a difference in the evaluation of the base model but only in future iterations. Here, the figure shows different values for the base model evaluation because of non-determinism when transforming the SYNTCOMP and J.A.R.V.I.S. specifications into our format. While two formulas of the same specification in two evaluations are always logically equal, they do not have to have the same abstract syntax tree and variable names - which leads to different results by the model.

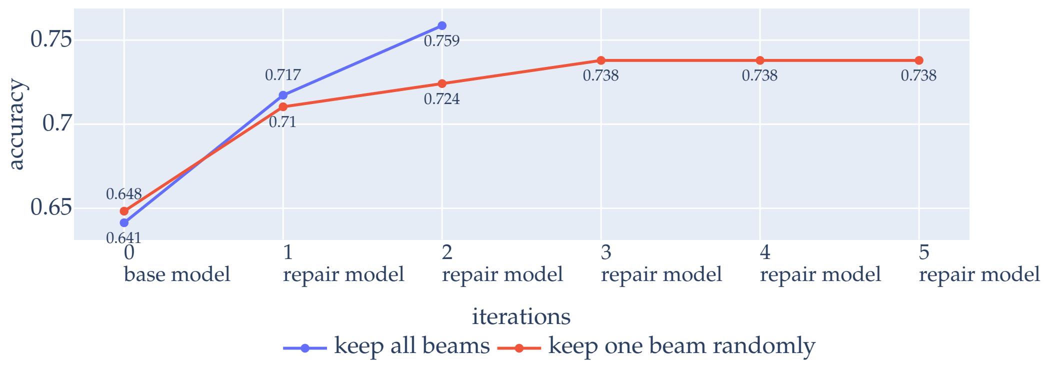
In Fig. 6.9a (*timeouts*), we see that the pipeline improves the accuracy of the base model by 6.9 percentage points. Additionally, repeats (iterations 2-5) have a larger stake in the improvement than in Fig. 6.4 with 2.5 percentage points. It seems that more than 6 iterations would even further improve the accuracy, as we have not yet reached a plateau at iteration 5.

➔Fig. 6.4, p. 48

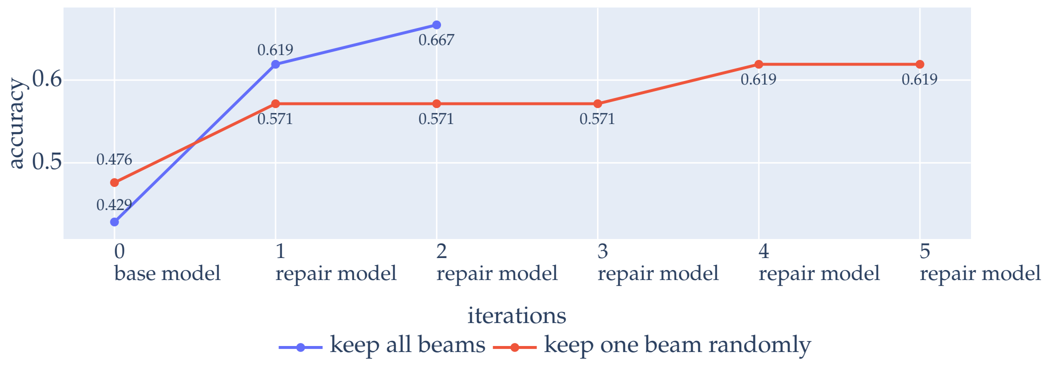
The Fig. 6.9b (SYNTCOMP) and Fig. 6.9c (J.A.R.V.I.S) show that we can improve the out-of-distribution benchmarks by 7.1 and 8.8 percentage points, respectively. We can also see that the pipeline performs considerably better when keeping all beams in all iterations. In future work, we can try to find a suitable heuristic that chooses the most promising beam instead of randomly choosing. This could improve the accuracy of the pipeline without the need to keep all beams in each iteration.



(a) Timeouts. Semantic accuracy. Keep one beam randomly after each iteration



(b) SYNTCOMP benchmark (out-of-distribution).



(c) J.A.R.V.I.S benchmark (out-of-distribution).

Figure 6.9.: Accuracy after each iteration in the pipeline. Beam size 16.

Chapter 7

Conclusion

In this work, we presented a method to repair circuits using a Transformer-based architecture. We introduced a new Transformer-based architecture and created several datasets for supervised learning. We trained models using this architecture and datasets that achieve promising results on the repair problem. We further combined our approach with Neural Circuit Synthesis [Sch+21], to demonstrate that our approach can be applied to existing approaches, improving the results significantly on a given set of hold-out samples and benchmarks such as the reactive synthesis competition SYNTCOMP. We provide code, guided notebooks, datasets, and trained models in our repository [1].

Separated Hierarchical Transformer. We introduced a new architecture that is based on the Transformer [Vas+17], more precisely the hierarchical Transformer [Li+21]. The separated hierarchical Transformer extends the hierarchical Transformer to specialize in handling multiple sources. We divide the encoder into multiple hierarchically structured encoder layers that each handle different input sources or different partitions of each input source. Some layers share parameters while others have *separated* parameters. In the lower layers, features in the partitions of the input can be learned without the greater context, while the higher layers combine the features learned from each partition and each input source and learn the greater context.

Datasets. We created a range of datasets that can be used for supervised learning of the repair problem. We gave instructions on how to generate these datasets and document findings that can be transferred to other distributions of circuits and other problems of similar nature. We demonstrated that these datasets can be used for training the separated hierarchical Transformer on the circuit repair problem. The datasets contain

[1] `https://github.com/MatCos/ml2`

specifications in LTL, target circuits in form of an And-Inverter Graph that satisfies the specification, and faulty circuits in form of a (faulty) And-Inverter Graph, that is close to the target circuit. Two methods were used to generate (faulty) circuits. First, we altered the target circuit to create broken circuits. Secondly, we used mispredicted circuits from Neural Circuit Synthesis as a base for our training data. For all datasets, we documented how they were generated and showed insightful statistics.

Repairing Systems. We trained multiple models that can repair a faulty circuit with high accuracy. We analyzed the evaluations by specification and target circuit size and learned that the performance does not depend on the size of the specification or target circuit - contrary to the results of Neural Circuit Synthesis [Sch+21]. We further analyzed the influence of the Levenshtein distance between the (faulty) circuit and the target circuit on the performance of the model. At least for some models, we could not infer that samples with larger distances are harder to solve. We also showed that our model(s) often do not only find one correct solution but, depending on the model, up to 7.6 correct distinct solutions per solved sample. We showed that, even for samples that were not solved, the result improved as the predicted circuit on average is closer to a correct circuit than the circuit we fed into the model.

experiment	exp-repair-alter-19-0	exp-repair-gen-108-0	exp-repair-gen-96-0
semantic accuracy	77%	78%	84%
syntactic accuracy	59%	44%	53%
correct beams per sample	4.3	5.8	6.6

Improving Neural Circuit Synthesis. We showed the potential of our approach by contributing to a competitive and well-researched problem. We applied our models to Neural Circuit Synthesis [Sch+21] to repair mispredicted circuits. We made significant improvements on held-out instances and benchmarks. We also found out that we can apply multiple iterations of the repair model. In the first iteration, the model attempts to repair the mispredictions of the synthesis model, while in further iterations the model attempts to repair the mispredictions of the repair model. Our method is not limited to improving a specific approach, we documented valuable insights and instructions, that can be transferred to improve other circuit synthesis approaches.

	baseline	after first iteration	after last iteration
test	77.1%	82.6% (+5.5)	83.9% (+6.8)
timeouts	30.7%	34.9% (+4.2)	37.6% (+6.9)
SYNTCOMP	64.1%	71.7% (+7.6)	75.9% (+11.8)
J.A.R.V.I.S	42.9%	61.9% (+19)	66.7% (+23.8)

7.1. Outlook

We see multiple angles from which we think our approach can still be improved and extended.

Graph Representations. We think that the AIGER format, while succinct, is not the most intuitive representation of a circuit. It is not unlikely that a Transformer model would understand other representations more easily. Using a graph encoding or ideas from the Graphormer [Yin+21], we could directly use the And-Inverter Graph as input, without needing the AIGER format as an intermediary text representation. One could also switch from circuits to transition systems, which are often easier to understand than circuits. Using transition systems instead of text representations of And-Inverter Graphs might also change the type of mistakes a network would produce. While in the AIGER format, small mistakes in terms of the edit distance to a correct solution generally do not mean that these are also logically close, for transition systems we anticipate this to be different because the time dimension is more explicit than in circuits.

Automatic Post-Editing. Instead of using a separated hierarchical Transformer, we could also try automatic post-editing, where the decoder is initialized with the circuit to repair. Given the initialization and the input from the encoder, the decoder could then predict the repaired circuit. The current architecture has multiple segments with almost identical functionality. A local separated layer of the encoder and the decoder both interpret AIGER circuits. While there are slight differences such that the encoder is trained on faulty circuits and the decoder only on satisfying circuits, most of the structure is the same in both. Moving the repair circuit to the decoder would spare the duplicated parameters and has the synergy potential of seeing even more circuit data in the decoder.

Error Trace. Model-checking circuits against a specification usually gives an error trace for violating circuits. The error trace is a trace of input/output assignments that are produced by the circuit but contradict the specification. Error traces can contain valuable information on the error in the circuit. In circuit repair, we could use the error trace as additional input, hopefully guiding the repair by further refining the type of error that needs to be repaired. This could further improve the results and make the model even more robust against out-of-distribution faulty circuits.

Repairing Programs & Human Errors Finally, we want to apply this approach to other domains, such as programs. The field of automated software repair [Mon18] is a very active research field, where recent additions such as Jigsaw [Jai+21] attracted attention. Automated software repair can be applied to human-written programs or machine-generated code for example by GitHub Copilot. The need to repair such programs is evident as it was recently shown that Copilot not only often produces buggy code but also introduces heavy security risks [Pea+22]. Existing repair approaches, however, often only rely on assertions and test cases but cannot consider formal specifications. We showed that our Transformer combines buggy programs with formal specifications to produce correct programs. Advances in the topic of formal temporal specifications paired with automated program repair could be a huge contribution in terms of security and verifiability of code.

Appendix A

Appendix

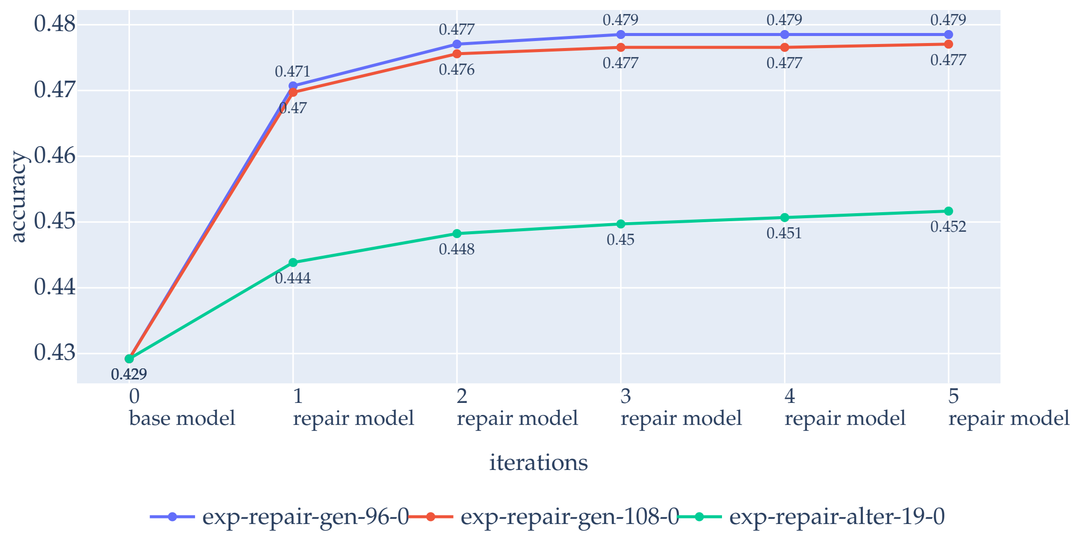


Figure A.1.: Pipeline syntactic accuracy of selected models in each iteration step. Based on a fixed set of held-out samples.

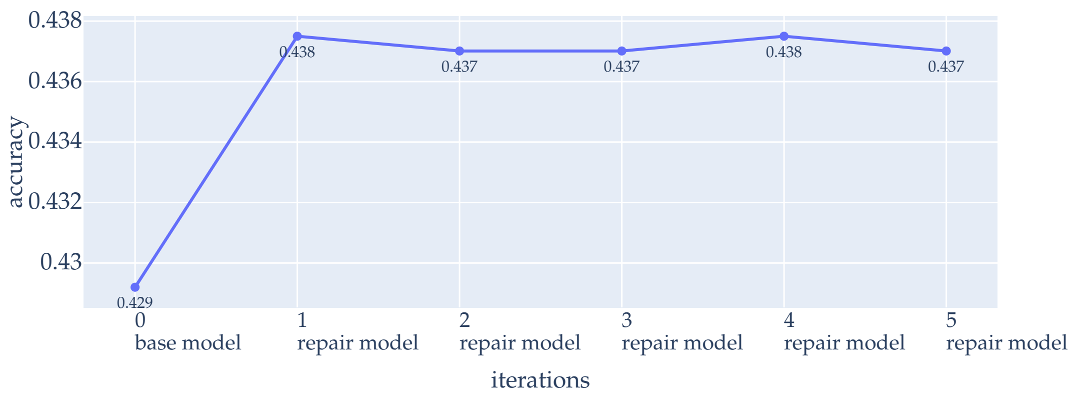


Figure A.2.: Syntactic accuracy in each iteration. Experiment exp-repair-gen-96-0, beam size 16, keeping the best beam for the next iteration.

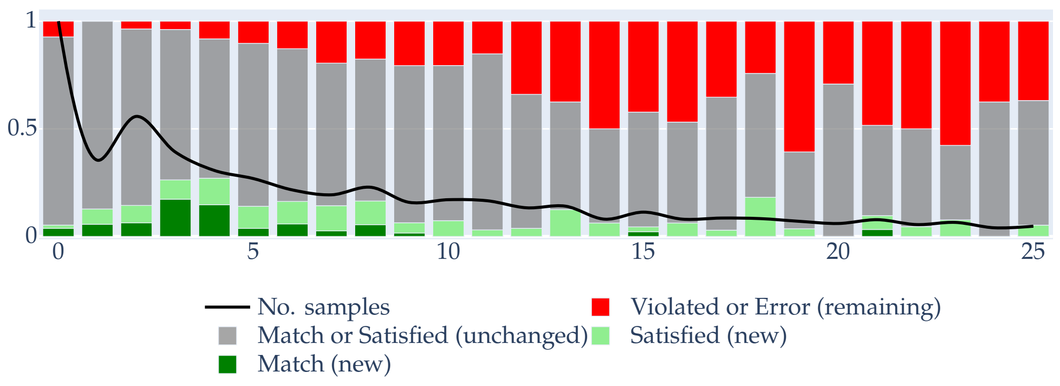


Figure A.3.: The status improvement of samples between iteration 0 and 5, categorized by the target circuit size (sum of latches and AND gates). The line shows the number of samples in each bin, scaled from 0 to 1. Per sample, we aggregate the best result from all iterations (base model and repair model) and all 16 beams. Samples that were already solved in the base model are shown in gray (unchanged). Green samples are solved in at least one repair iteration but not in the base model (new). Red samples are neither solved in the repair model nor the base model (remaining). Based on a fixed set of held-out samples.

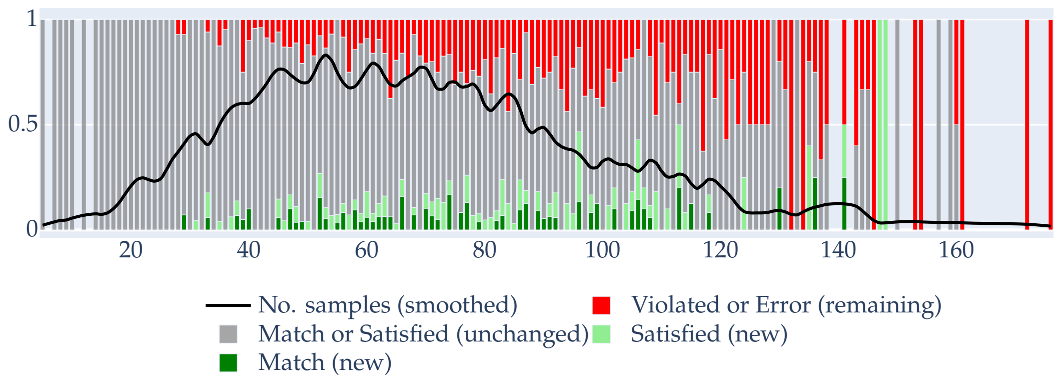


Figure A.4.: The status improvement of samples between iteration 0 and 1, categorized by the AST size of the problem. Only one iteration of the repair model is applied. The line insinuates the number of samples in each bin, scaled from 0 to 1. Per sample, we aggregate the best result from the base and repair model evaluation and all 16 beams. Samples that were solved in the base model are shown in gray (unchanged). Green samples are solved in the repair model but not in the base model (new). Red samples are neither solved in the repair model nor the base model (remaining). Based on a fixed set of held-out samples.

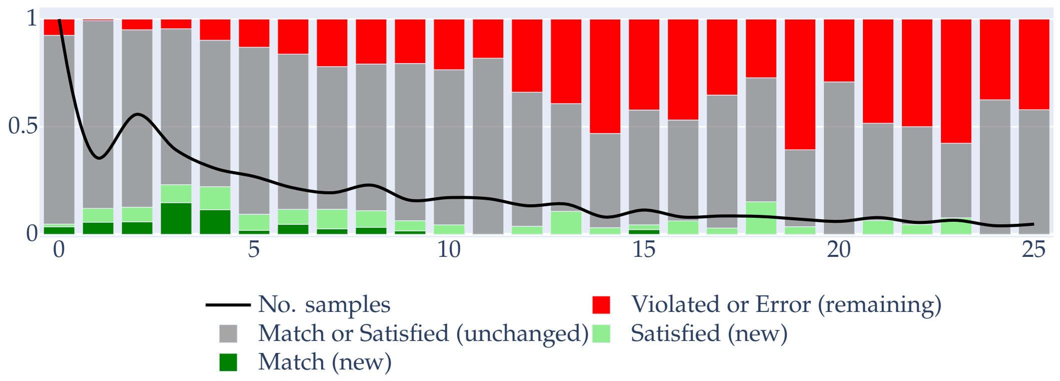


Figure A.5.: The status improvement of samples between iteration 0 and 1, categorized by the target circuit size (sum of latches and AND gates). Only one iteration of the repair model is applied. The line shows the number of samples in each bin, scaled from 0 to 1. Per sample, we aggregate the best result from the base and repair model evaluation and all 16 beams. Samples that were solved in the base model are shown in gray (unchanged). Green samples are solved in the repair model but not in the base model (new). Red samples are neither solved in the repair model nor the base model (remaining). Based on a fixed set of held-out samples.

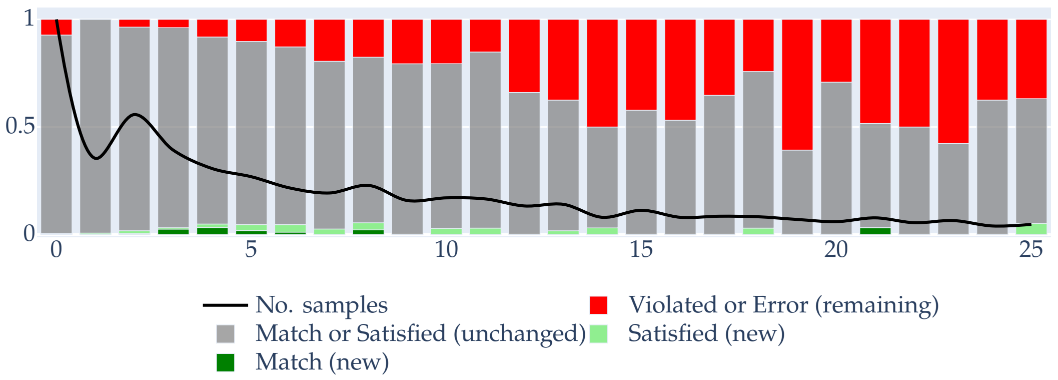


Figure A.6.: The status improvement of samples between iteration 1 and 5, categorized by the target circuit size (sum of latches and AND gates). Shows the effect of the repeats of the repair model. The line shows the number of samples in each bin, scaled from 0 to 1. Per sample, we aggregate the best result from all iterations (base model and repair model) and all 16 beams. Samples that were solved in the base model or the first repair iteration are shown in gray (unchanged). Green samples are solved in at least one repeat but not in the base model or the first iteration (new). Red samples are neither solved in the repair model nor the base model (remaining). Based on a fixed set of held-out samples.

A.1. Dataset Feature Table

The following table shows the parameters and some statistics of selected datasets. More extensive information can be found in the full table at the QR-code or using the Jupyter Notebook we provide in the repository[1].

SCAN OR CLICK

Name of dataset		scpa-repair-alter-1	scpa-repair-alter-4	scpa-repair-alter-6	scpa-repair-alter-19	scpa-repair-gen-77	scpa-repair-gen-81	scpa-repair-gen-96	scpa-repair-gen-108
alter parameters	$P_{deletes}$	0.2	0.1	1	0	0.2		0.2	0.2
	$changes_{min}$	1	1	1	1	1		1	1
	$changes_{max}$	50	50	20	100	50		50	100
	$changes_{range-68}$	15	15	5	50	15		15	15
	var_{min}	0	0		0	0		0	0
	var_{max}	61	61		61	61		61	61
	$var_{range-68}$	20	20		20	20		20	20
gen parameters	$filter_{max-dist}$					50	50	50	100
	$filter_{max-match}$					0.1	0	0	0
	$filter_{process}$					alter	remove	alter	alter
	change misleading targets					TRUE	TRUE	TRUE	TRUE
	$parent_{beam-size}$					1	1	3	3
	train size	200000	200000	200000	200000	196436	47761	581778	581778
	val size	25000	25000	25000	25000	24575	9073	72777	72777
	test size	25000	25000	25000	25000	24563	9214	72805	72805
statistics training split	status: violated	0	0	0	0	0.24	1	0.38	0.48
	status: satisfied	0	0	0	0	0.1	0	0	0
	status: match	0	0	0	0	0	0	0	0
	status: changed	1	1	1	1	0.66	0	0.61	0.51
	realizable	0.5	0.5	0.5	0.5	0.49	0.46	0.49	0.49
	satisfied and status changed	0.0145	0.0175	0.0125	0.00625	0.018		0.01925	0.02025
	Levenshtein distance mean	15.99	12.83	19.76	19.18	14.77	19.76	15.7	21.14
	Levenshtein distance median	13	11	17	17	12	18	13	14
	Levenshtein distance std	12.1	9.33	14.03	11.74	12.97	14.89	12.77	21.48
	Levenshtein distance < 10	0.35	0.43	0.29	0.2	0.42	0.31	0.38	0.35

[1] `https://github.com/MatCos/ml2/blob/main/notebooks/datasets.ipynb`

A.2. Experiment Feature Table

SCAN OR CLICK

The following table shows the parameters and some statistics of selected experiments. More extensive information can be found in the full table at the QR-code or using the Jupyter Notebook we provide in the repository[2].

				name	exp-repair-gen-96-0	exp-repair-gen-108-0	exp-repair-alter-19-0
parameters				name dataset	scpa-repair-gen-96	scpa-repair-gen-108	scpa-repair-alter-19
				training steps	20000	20000	20000
				warmup steps	4000	4000	4000
				embedding size	256	256	256
	input/target length			max circuit length	128	128	128
				max num properties (spec)	12	12	12
				max tree size per property	25	25	25
	network			dropout	0	0	0
				network sizes	1024	1024	1024
				activation	relu	relu	relu
	attention			heads (encoder and decoder)	4	4	4
				layers (encoder)	4	4	4
				layers (decoder)	8	8	8
evaluation	on dataset	val		accuracy	0.950	0.914	0.951
				loss	0.063	0.109	0.072
				accuracy per sequence	0.324	0.252	0.399
		test	semantic beam accuracy	beam size 1	0.584	0.547	0.550
				beam size 16	0.847	0.784	0.772
	on pipeline	test_fixed	Repeats: 5 Beam Base: 16 Beam Repair: 16 keep: best	sem_improvement_accuracy	0.068	0.066	0.041
				sem_overall_accuracy	0.839	0.837	0.812
				syn_improvement_accuracy	0.049	0.048	0.022
				syn_overall_accuracy	0.479	0.477	0.451
				enc_overall_accuracy	0.851	0.849	0.823
			Repeats: 10 Beam Base: 4 Beam Repair: 4 keep: best	sem_improvement_accuracy	0.090	0.085	0.038
				sem_overall_accuracy	0.764	0.758	0.711
				syn_improvement_accuracy	0.030	0.031	0.014
				syn_overall_accuracy	0.408	0.409	0.392
				enc_overall_accuracy	0.775	0.769	0.721

[2] https://github.com/MatCos/ml2/blob/main/notebooks/experiments.ipynb

Bibliography

[BAM19] Tadas Baltrušaitis, Chaitanya Ahuja, and Louis-Philippe Morency. "Multimodal Machine Learning: A Survey and Taxonomy". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 41.2 (Feb. 2019). Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 423–443. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2018.2798607.

[BFT20] Tom Baumeister, Bernd Finkbeiner, and Hazem Torfah. "Explainable reactive synthesis". In: *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2020, pp. 413–428.

[BL90] J. Richard Buchi and Lawrence H. Landweber. "Solving sequential conditions by finite-state strategies". In: *The Collected Works of J. Richard Büchi*. Springer, 1990, pp. 525–541.

[Bru+07] Robert Brummayer et al. "The AIGER And-Inverter Graph (AIG) Format Version 20070427". In: (2007).

[Cav+14] Roberto Cavada et al. "The nuXmv Symbolic Model Checker". en. In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 334–342. ISBN: 978-3-319-08867-9. DOI: 10.1007/978-3-319-08867-9_22.

[Chu57] Alonzo Church. "Applications of recursive arithmetic to the problem of circuit synthesis." en. In: *Summaries of the Summer Institute of Symbolic Logic*. Vol. 1. Ithaca, NY: Cornell Univ., 1957, pp. 3–50.

[Fay+17] Peter Faymonville et al. "Encodings of Bounded Synthesis". In: 2017, pp. 354–370. URL: https://doi.org/10.1007/978-3-662-54577-5_20 (visited on 08/19/2022).

[FFT17] Peter Faymonville, Bernd Finkbeiner, and Leander Tentrup. "BoSy: An Experimentation Framework for Bounded Synthesis". en. In: *Computer Aided Verification*. Ed. by Rupak Majumdar and Viktor Kunčak. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 325–

332. ISBN: 978-3-319-63390-9. DOI: 10.1007/978-3-319-63390-9_17. URL: https://www.react.uni-saarland.de/tools/bosy/.

[FKM22] Bernd Finkbeiner, Felix Klein, and Niklas Metzger. "Live synthesis". en. In: *Innovations in Systems and Software Engineering* (Mar. 2022). Publisher: Springer Verlag. ISSN: 1614-5046. URL: https://publications.cispa.saarland/3681/ (visited on 08/19/2022).

[Gei+22] Gideon Geier et al. *J.A.R.V.I.S. TSL/TLSF benchmark suite*. en. 2022. URL: https://github.com/SYNTCOMP/benchmarks (visited on 07/26/2022).

[Hah+21] Christopher Hahn et al. *Teaching Temporal Logics to Neural Networks*. Tech. rep. arXiv:2003.04218. arXiv:2003.04218 [cs, stat] version: 2 type: article. arXiv, Feb. 2021. DOI: 10.48550/arXiv.2003.04218. URL: http://arxiv.org/abs/2003.04218 (visited on 06/09/2022).

[Jac+22a] Swen Jacobs et al. *SYNTCOMP 2020 Results | The Reactive Synthesis Competition*. en-US. June 2022. URL: http://www.syntcomp.org/syntcomp-2020-results/ (visited on 06/27/2022).

[Jac+22b] Swen Jacobs et al. *The Reactive Synthesis Competition (SYNTCOMP): 2018-2021*. Number: arXiv:2206.00251 arXiv:2206.00251 [cs]. June 2022. DOI: 10.48550/arXiv.2206.00251. URL: http://arxiv.org/abs/2206.00251 (visited on 06/27/2022).

[Jai+21] Naman Jain et al. *Jigsaw: Large Language Models meet Program Synthesis*. arXiv:2112.02969 [cs]. Dec. 2021. DOI: 10.48550/arXiv.2112.02969. URL: http://arxiv.org/abs/2112.02969 (visited on 08/11/2022).

[JGB05] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. "Program repair as a game". In: *International conference on computer aided verification*. Springer, 2005, pp. 226–238.

[Job+12] Barbara Jobstmann et al. "Finding and fixing faults". In: *Journal of Computer and System Sciences* 78.2 (2012). Publisher: Elsevier, pp. 441–460.

[KB17] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. arXiv:1412.6980 [cs]. Jan. 2017. DOI: 10.48550/arXiv.1412.6980. URL: http://arxiv.org/abs/1412.6980 (visited on 07/29/2022).

[LC19] Guillaume Lample and François Charton. *Deep Learning for Symbolic Mathematics*. arXiv:1912.01412 [cs] 216 citations. Dec. 2019. DOI: 10.48550/arXiv.1912.01412. URL: http://arxiv.org/abs/1912.01412 (visited on 07/26/2022).

[Li+21] Wenda Li et al. *IsarStep: a Benchmark for High-level Mathematical Reasoning*. Tech. rep. arXiv:2006.09265. arXiv:2006.09265 [cs, stat] type: article. arXiv, Mar. 2021. DOI: 10.48550/arXiv.2006.09265. URL: http://arxiv.org/abs/2006.09265 (visited on 06/09/2022).

[Lit+19] Patrick Littell et al. "Multi-Source Transformer for Kazakh-Russian-English Neural Machine Translation". In: *Proceedings of the Fourth Conference on Machine Translation (Volume 2: Shared Task Papers, Day 1)*. Florence, Italy: Association for Computational Linguistics, Aug. 2019, pp. 267–274. DOI: 10.18653/v1/W19-5326. URL: https://aclanthology.org/W19-5326 (visited on 08/09/2022).

[Mon18] Martin Monperrus. "Automatic software repair: a bibliography". In: *ACM Computing Surveys (CSUR)* 51.1 (2018). Publisher: ACM New York, NY, USA, pp. 1–24.

[MSL18] Philipp J. Meyer, Salomon Sickert, and Michael Luttenberger. "Strix: Explicit Reactive Synthesis Strikes Back!" en. In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 578–586. ISBN: 978-3-319-96145-3. DOI: 10.1007/978-3-319-96145-3_31. URL: https://strix.model.in.tum.de/.

[Nah+16] L. Nahabedian et al. "Assured and correct dynamic update of controllers". In: *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 96–107. ISBN: 978-1-4503-4187-5. DOI: 10.1145/2897053.2897056. URL: https://doi.org/10.1145/2897053.2897056 (visited on 08/19/2022).

[Nis+20] Yuta Nishimura et al. "Multi-Source Neural Machine Translation With Missing Data". In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 28 (2020). Conference Name: IEEE/ACM Transactions on Audio, Speech, and Language Processing, pp. 569–580. ISSN: 2329-9304. DOI: 10.1109/TASLP.2019.2959224.

[Pea+22] Hammond Pearce et al. "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions". In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 754–768.

[Pnu77] A. Pnueli. "The temporal logic of programs". In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. Oct. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[PS20] Stanislas Polu and Ilya Sutskever. *Generative Language Modeling for Automated Theorem Proving*. arXiv:2009.03393 [cs, stat] 63 citations. Sept. 2020. DOI: 10.48550/arXiv.2009.03393. URL: http://arxiv.org/abs/2009.03393 (visited on 07/26/2022).

[Rab+20] Markus N. Rabe et al. *Mathematical Reasoning via Self-supervised Skip-tree Training*. arXiv:2006.04757 [cs, stat 22 citations. Aug. 2020. DOI: `10.48550/arXiv.2006.04757`. URL: `http://arxiv.org/abs/2006.04757` (visited on 07/26/2022).

[Ros91] Roni Rosner. *Modular synthesis of reactive systems*. The Weizmann Institute of Science (Israel), 1991.

[Sch+21] Frederik Schmitt et al. “Neural Circuit Synthesis from Specification Patterns”. In: *Advances in Neural Information Processing Systems*. Vol. 34. Curran Associates, Inc., 2021, pp. 15408–15420. URL: `https://proceedings.neurips.cc/paper/2021/hash/8230bea7d54bcdf99cdfe85cb07313d5-Abstract.html` (visited on 06/15/2022).

[SQ19] Vighnesh Shiv and Chris Quirk. “Novel positional encodings to enable tree-based transformers”. In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019. URL: `https://proceedings.neurips.cc/paper/2019/hash/6e0917469214d8fbd8c517dcdc6b8dcf-Abstract.html` (visited on 06/09/2022).

[Vas+17] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017. URL: `https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html` (visited on 07/07/2022).

[Wu+16] Yonghui Wu et al. *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*. arXiv:1609.08144 [cs]. Oct. 2016. DOI: `10.48550/arXiv.1609.08144`. URL: `http://arxiv.org/abs/1609.08144` (visited on 07/04/2022).

[Yin+21] Chengxuan Ying et al. “Do Transformers Really Perform Badly for Graph Representation?” In: *Advances in Neural Information Processing Systems*. Vol. 34. Curran Associates, Inc., 2021, pp. 28877–28888. URL: `https://proceedings.neurips.cc/paper/2021/hash/f1c1592588411002af340cbaedd6fc33-Abstract.html` (visited on 08/11/2022).

[ZK16] Barret Zoph and Kevin Knight. “Multi-Source Neural Translation”. In: *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2016, pp. 30–34.